# TEMU

## *User Manual*

Version 5.0, 2026-11-20

© Terma GmbH, Germany, 2015-2023. Proprietary and intellectual rights of Terma GmbH, Germany are involved in the subject-matter of this material and all manufacturing, reproduction, use, disclosure and sales rights pertaining to such subject-matter are expressly reserved. This material is submitted for a specific purpose as agreed in writing, and the recipient by accepting this material agrees that this material will not be used, copied or reproduced in whole or in part, nor its content (or any part thereof) revealed in any manner or to any third party, except own staff, to meet the purpose for which it was submitted and subject to the terms of the written agreement.

# Table of Contents

1. Overview . . . 6
2. Documentation Overview . . . 7
2.1. Target Manuals . . . 7
2.2. Model Manuals . . . 7
3. System Requirements . . . 9
3.1. Linux . . . 9
4. Getting Started . . . 10
4.1. Installation . . . 10
4.2. TEMU Setup . . . 11
4.3. Licenses . . . 11
4.4. License Server . . . 13
4.5. Running the Emulator . . . 14
4.6. Setting up a Processor and Scheduler . . . 14
4.7. Loading and Running Software . . . 14
5. Configuration System . . . 16
5.1. Configuration File Location . . . 16
5.2. Configuration Sources . . . 16
5.3. YAML Format . . . 17
5.4. Scalars and Lists . . . 17
5.5. Access Patterns . . . 18
5.6. Queried Configuration Variables . . . 19
5.7. Defaults and Fallbacks . . . 22
5.8. Related Documentation . . . 23
6. Migration Guide . . . 24
6.1. C API Compatibility Headers . . . 24
6.2. Script and System Migration . . . 24
6.3. Command Aliasing Compatibility . . . 26
6.4. Recommended Approach . . . 26
7. Project Files . . . 28
7.1. Command Line Usage . . . 28
7.2. C API Usage . . . 28
7.3. File Format . . . 29
7.4. Project Metadata . . . 30
7.5. Path Mappings . . . 31
7.6. Configuration Values . . . 31
7.7. Runtime Paths . . . 33
7.8. Path Resolution . . . 33

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

7.9. Startup Scripts . . . 34
7.10. Targets . . . 34
7.11. Relationship to User Configuration . . . 35
8. Project Templates . . . 36
8.1. Initializing a Project . . . 36
8.2. Using the Generated Project . . . 36
8.3. Generated Startup Files . . . 37
8.4. Default Runtime Layout . . . 37
8.5. Adding a Plugin . . . 38
8.6. Generated Plugin Metadata . . . 38
8.7. Notes . . . 39
9. Support and Maintenance . . . 40
10. Time . . . 41
10.1. Time Base . . . 41
10.2. Temporal Decoupling . . . 43
11. Command Line . . . 44
11.1. Command Line Interface Options . . . 44
11.2. Command Syntax . . . 45
11.3. Variables . . . 45
11.4. Help Command . . . 45
11.5. Commands . . . 45
12. Libraries . . . 49
12.1. Deprecation Policy . . . 49
12.2. Experimental Application Programming Interfaces . . . 50
13. Object System . . . 51
13.1. Object Graph and Interface Properties . . . 53
13.2. Object System Functions . . . 53
13.3. Properties . . . 54
13.4. Pseudo Properties . . . 54
13.5. Interfaces . . . 54
13.6. Ports . . . 54
14. Running the Simulated System . . . 56
14.1. Schedulers . . . 56
14.2. Running, Stepping and Tracing . . . 58
15. Logging System . . . 60
15.1. Logging Backends . . . 60
15.2. Categories . . . 61
15.3. Severity Levels . . . 62
15.4. Filtering . . . 62

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

16. Events . . . 64
16.1. Events from Other Threads . . . 65
16.2. Notifications . . . 65
17. Processor Emulation . . . 67
17.1. Running a Processor . . . 67
17.2. Event System . . . 69
17.3. Multi-Core Emulation and Events . . . 69
17.4. CPU Interface . . . 70
18. Memory Emulation . . . 72
18.1. Memory Spaces . . . 72
18.2. Memory Transactions . . . 72
18.3. Address Translation Cache . . . 75
18.4. Memory Hierarchy and Caches . . . 75
18.5. Interfaces . . . 77
19. Network Simulation . . . 79
19.1. Network Traffic Capture . . . 79
19.2. Host Network Integration . . . 79
20. Components . . . 80
21. Snapshots . . . 81
21.1. Snapshot Restore Phases . . . 81
21.2. JSON Caveats . . . 81
22. Software Debugging . . . 83
22.1. CLI Based Software Debugging . . . 83
22.2. GDB Server . . . 86
23. OS Awareness . . . 89
23.1. OS Awareness . . . 89
24. Profiling and Coverage . . . 93
24.1. Enabling Profiling Mode . . . 93
24.2. Exporting Profiles . . . 93
25. Emulation Performance . . . 95
25.1. Performance Impact of Target Software . . . 95
25.2. Performance Impact from Host . . . 96
25.3. Performance Monitoring Capabilities . . . 97
25.4. Performance Improvement Capabilities . . . 98
26. Fault Injection . . . 105
26.1. Ensuring Determinism . . . 105
26.2. Device Faults . . . 105
26.3. Memory Faults . . . 105
26.4. Network Interception . . . 106

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

27. Scripting . . . 109
28. External Analysis Tools . . . 110
28.1. Valgrind and Binary Translation . . . 110
28.2. Process-Lifetime Allocations . . . 110
29. Examples . . . 112
29.1. Quick CPU Construction Using JSON Files . . . 112
29.2. Command Line CPU Construction . . . 112
29.3. Programmatic CPU Construction . . . 113
30. Modelling Guide . . . 117
30.1. Introduction . . . 117
30.2. Basic . . . 117
30.3. Advanced . . . 127
30.4. Examples . . . 131
31. TEMU System Debugger GUI . . . 136
31.1. Overview . . . 136
31.2. Views . . . 140
31.3. Customization . . . 196

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 1. Overview

This document is the TEMU (Terma Emulator) Software Users Manual. It describes the fundamental concepts and general usage of the TEMU libraries and command line interface.

TEMU is a multi-achictecture microprocessor emulator. It currently supports the ARMv7, SPARCv8 and PowerPC instruction sets. Currently this includes: ERC32, LEON2, LEON3, LEON4, Cortex-R4/5 and PPC750. The emulator can emulate multi-core processors.

TEMU is a full system emulator. It is capable of emulating (multi-core) microprocessors, memory and peripherals. Different devices are written as plugins. Including CPUs, memory and device modules. In-fact, systems are constructed by connecting different modules together. Thus, there is no hard-wiring of devices to any particular memory layout.

To give an example, in order to construct a LEON2 processor, one would first create and connect the CPU core, ROM, RAM and LEON2 SoC components.

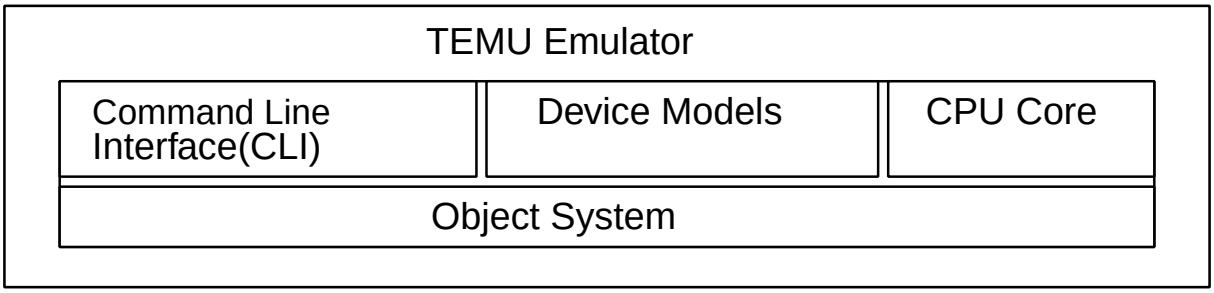



*Figure 1. Layers of the Terma Emulator*

There are two user interfaces for TEMU, the *Command Line Interface (CLI)* and the libraries (API). The CLI offers an interactive tool for running the emulator by itself. While the API allows the user to integrate the emulator with other simulators.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 2. Documentation Overview

This document is the software users manual. It gives a high level overview of the system. However, as TEMU is modular, this manual does not document everything. The details are described in different target, model and API manuals. The target and model manuals are *reference documents*, and describe the properties and interfaces implemented by the relevant model.

## 2.1. Target Manuals

Target manuals describe the usage of the processor emulators. There is one target guide per supported architecture (currently this include only the SPARCv8). Note that a CPU core does not contain any I/O models.

*Table 1. TEMU Target Manuals*

| Document | Description |
|---|---|
| ARMv7 Target Manual | Manual for all the ARMv7 CPU cores |
| SPARCv8 Target Manual | Manual for all the SPARC CPU cores |
| PowerPC Target Manual | Manual for all the PowerPC CPU cores |

## 2.2. Model Manuals

Each implemented I/O model has a manual describing the usage of the model. That includes how to configure the model and any known limitations. The models include not only device models, but also bus models.

The following table lists some of the manuals.

*Table 2. TEMU Model Manuals*

| Document | Note |
|---|---|
| Modelling Guide | How to write device models |
| GPIO Bus Model Manual | Manual for the built in GPIO bus model |
| UART Model Manual | Manual for the built in UART bus model |
| AMBA Bus Model Manual | Manual for the built in AMBA bus model |
| MEC Device Model Manual | Manual for the ERC32 memory controller |
| LEON2 Device Model Manual | Manual for the LEON2 on-chip devices |
| GRLIB GpTimer Device Model Manual | GRLIB manual |
| GRLIB IrqMp Device Model Manual | GRLIB manual |
| GRLIB AhbCtrl Device Model Manual | GRLIB manual |
| GRLIB ApbCtrl Device Model Manual | GRLIB manual |

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

| Document | Note |
|---|---|
| GRLIB AhbUart Device Model Manual | GRLIB manual |
| GRLIB FtmCtrl Device Model Manual | GRLIB manual |

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 3. System Requirements

## 3.1. Linux

### 3.1.1. Requirements

- x86-64 processor with `cmpxchg16b` instruction (this includes most x86-64 processors), and the BM1 and BM2 instruction set extensions (Haswell and later processors).
- GLIBC 2.29 or later



The technically exact GLIBC 2.29 requirement is set while compiling TEMU, it may change as TEMU evolves. The GLIBC requirement specified here reflects the build platform used, not the strict technical requirement, TEMU may run on earlier GLIBC, but there is no guarantee that this remains stable over patches.

### 3.1.2. Recommended Requirements

- At least two times the amount of RAM that the simulated system will contain.
- Qt5 installation via system package manager.

If the emulator is to be running in parallel mode, the recommendation is that at least one more physical processor cores is available than the number of threads the TEMU scheduler is configured with.

For example, if TEMU is running a quad-core processor model, with the scheduler configured to 4 threads, then 5 host processors should be available for use by TEMU.

### 3.1.3. Supported Distributions

The following Linux distributions are known to work with TEMU. They are seen as officially supported.

- RHEL 9
- Rocky Linux 9
- Debian 11
- SLES 15 SP3

Other distributions may work, including earlier versions, but they are not tested on regular basis.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 4. Getting Started

## 4.1. Installation

To install TEMU, the best approach is to use the RPM or DEB files. The latest versions can be downloaded from https://temu.terma.com/.

The following table illustrates which packages should be used on which operating system. Normally generic packages are available. For some older systems specific packages may be available.

*Table 3. Installation Package Suggestions*

| OS | Package Type |
|---|---|
| CentOS | .rpm |
| Debian | .deb |
| RedHat Enterprise Linux (RHEL) | .rpm |
| SUSE Linux for Enterprises (SLES) | .rpm |
| Ubuntu | .deb |
| Others | .tar.bz2 |

The following commands can be used to install the different types of packages (where `x.y.z` is the version number):

```
# Install RPM
$ rpm -ivh temu-x.y.z-x86_64-Linux.rpm

# Install DEB
$ dpkg -i temu-x.y.z-x86_64-Linux.deb

# Install Tarball (.tar.bz2)
$ bunzip2 temu-x.y.z-x86_64-Linux.tar.bz2
$ tar xvf temu-x.y.z-x86_64-Linux.tar
```

By default, the packages install TEMU in `/opt/temu/5.0`. The packages have also been created and bundled with all the normal dependencies they need. This include the standard C++ libraries, so there should be no problem to install and run the emulator on any Linux system. Note that testing is normally done on stable Debian (currently Jessie/8.0), RHEL7 and SLES11.

TEMU consist of a set of libraries and a command line tool. The libraries are normally installed in `/opt/temu/5.0/lib` and the tools in `/opt/temu/5.0/bin/`. The binaries and libraries have been liked with the `RPATH` option, so there is no need to set `LD_LIBRARY_PATH`.

There are also packages for a build which has asserts enabled. Asserts have a performance penalty, which at times can be heavy. Therefore, assert builds are opt-in. These packages installs under:

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
/opt/temu/5.0+asserts/
```

## 4.2. TEMU Setup

Before you start, you need to setup TEMU. The TEMU setup command, will create a *temu*-group on the system and add the current user to the group.

TEMU setup is using the `temu` command line tool:

```
$ sudo temu --setup
```



This command should preferably run using `sudo`. Using sudo, will ensure that the current user (the one invoking `sudo`) is added to the *temu*-group. If running the command as root, the *root* user will be added to the *temu*-group instead.

In case sudo is not available for your user, a sysadmin with *root* or *sudo* access needs to run the command.

In this case, the system administrator should add the user to the *temu*-group manually.

This can be done by running the following command as root:

```
usermod -a -G temu jdoe
```

## 4.3. Licenses

### 4.3.1. License Files

See https://temu.terma.com/ for more information on licenses. Note that you must have a valid license to run TEMU.

#### Making Sure the TEMU Group is Available



If the setup command was just executed, it is likely that license installation detailed in Section 4.3.1.2 will fail as the current shell will not be aware of the newly created *temu*-group.

To ensure that the newly created *temu*-group, where the user has been added, is made available to the current user it is possible to either logout and login, or one can run the following commands in the current shell:

```
exec newgrp temu ①
exec newgrp - ②
```

① Replace the current shell with one using *temu* as the primary group.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

② Replace the shell using *temu* as the primary group with one using the default group.

### Installing a License



Do not run the commands in this section as root. They are intended to be executed by a user in the *temu*-group. See Section 4.2 and Section 4.3.1.1 for more info.

A TEMU license is installed using the `temu` command line tool:

```
# DO NOT RUN THIS AS ROOT
$ temu --install-license my-license-file.json
```

It is possible to install multiple license files for the same user (e.g. to enable multiple machines).

### Revoking a License

To revoke a license in case of e.g. hardware migration, you should issue the revoke command on a license. To get the license ID, use the `--list-licenses` option to TEMU:

```
$ temu --list-licenses
| License ID       | Status
|------------------|---------------------------------------
| 123456789abcedf0 | hardware address error, installed
| abcdef0123456789 | valid, installed
```

Above you can see two licenses, one which does not match the current computer (hardware address error). To revoke a license for the current computer you should revoke a valid license. Typically, invalid licenses will be either marked as hardware address error or expired for time bounded licenses.

Next step is to revoke the license using the relevant license ID as listed above:

```
$ temu --revoke-license abcdef0123456789
------------------------ IMPORTANT NOTICE -------------------------
You are about to revoke a license, a revoked license invalidates the
license from use on your machine. A response code will be generated that
should be sent to Terma in order to be credited one license activation
NOTE: License revokations are only intended for use when retiring hardware
Revocations cannot be used to simulate a floating license, revocatiosn are
limited in numbers and excessive use is not allowed.
NOTE: In case of catastrophic failure of hardware, it may not be
possible to revoke the license, in that case you need to contact Terma
for an activation credit

Are you sure you wish to revoke the license (yes|no)?
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

Answer yes if you wish to proceed:

```
Revokation key: '123456789abcdef0...:abcdef0123456789...'
Send the key to Terma in order to be credited with one license activation
NOTE: You cannot use revocations as floating licenses, they are intended for
      hardware migration only.
```

Then copy the key and send it to Terma to be credited with one license activation.

## 4.4. License Server

An alternative to the node-locked licenses described in Section 4.3.1, are floating licenses.

Floating licenses require a license server to be running locally. The license server can be downloaded at https://tgss.terma.com/. Installation instructions for the license server are available here.

To use the license server from TEMU, create a file here (this is a per user setting): `~/.config/tgss/app/lm/config.yaml`

The file should look like the following:

```
server: "my-license-server.example.com:8080"
user: admin
password: password
```

In this case we use the default license manager user name and password, that should be changed using the license server web interface.

Now a license can be checked out from the server as follows with the TEMU command line:

*Listing 1. License Checkout with TEMU Command Line*

```
license-checkout pool='68728ED0-09D6-4732-BE8B-E51D426BC08B'
```



Licenses are checked out on a per-user-machine combination, and only one license can be the current one.

The command above can of-course be run using the `temu_execCommand()` function.

The pool-id is a UUID string, which can be determined using the license server web interface.

A checked out license can be released manually as follows:

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

*Listing 2. License Checkout with TEMU Command Line*

```
license-release
```



Floating licenses are automatically released after 24 hours.

## 4.5. Running the Emulator

To start the command line interface (CLI), simply run: `/opt/temu/5.0/bin/temu` or `/opt/temu/5.0+asserts/bin/temu`. The command line interface exists to run the emulator in stand alone mode.

## 4.6. Setting up a Processor and Scheduler

When TEMU is running it will normally display the `temu>`-prompt. This is the command prompt.

It is required to select the relevant scheduler, with the scheduler created, it is possible to use one of the bundled CPU configurations in `/opt/temu/5.0/share/temu/sysconfig/`. Common configurations that instantiate different types of systems are available. The command line scripts can be executed using the exec command. This can be done as illustrated in the following examples:

*Listing 3. Create a LEON2 System*

```
temu> init-scheduler variant=multi-level
temu> exec leon2.temu
```

*Listing 4. Create a Dual Core LEON3 System*

```
temu> init-scheduler variant=multi-level
temu> exec leon3-dual-core.temu
```

## 4.7. Loading and Running Software

When a system has been created, it is time to load and run software in the emulator. The example here assumes that the system was created as in the previous example. To load software which may be in ELF or SREC format the load-command can be used.



When running application software directly, as in contrast to have it loaded by boot software. Then the user needs to ensure that the boot software is simulated. This includes initializing the stack and frame pointers. On some systems timer registers must be initialized.

Execution of software in a single core system can be done by the run and step commands. The run-command runs the software for a given time (either cycles or seconds), while the step-command single steps the software instruction by instruction.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

*Listing 5. Load and Run Software Image*

```
temu> load obj=cpu0 file=rtems-hello.elf
info: cpu0 : loading section 1/1 0x40000000 - 0x4001ec20 pa = 0x40000000
temu> set-reg cpu=cpu0 reg="%fp" value=0x407ffff0
temu> set-reg cpu=cpu0 reg="%sp" value=0x407fff00
temu> cpu0.setPC pc=0x40000000
temu> run time=10.0
```



It is assumed that the user have access to application software and / or cross compilers and is familiar with how to use these tools.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 5. Configuration System

TEMU has a YAML-based configuration system with multiple configuration sources.



The project-file layer, list-valued configuration entries, `temu_configGetStringList()`, and the path-list configuration entries described on this page are new in TEMU 5.0.

The system is used for:

- user-scoped preferences
- project-scoped runtime setup
- command-line overrides
- internal API-provided configuration values

Configuration values are exposed through the public `temu_configGet*()` API and through a small set of CLI inspection commands.

## 5.1. Configuration File Location

By default, TEMU reads user configuration from:

```
~/.config/temu/config.yaml
```

The `~/.config/temu/` directory is created automatically during startup if it does not already exist.

The configuration file can be overridden on the command line:

```
temu --config-file /path/to/config.yaml
```

Project-scoped configuration can be loaded separately with:

```
temu --project /path/to/temu-project.yaml
```



`--project` is new in TEMU 5.0.

See Chapter 7 for the project-file format.

## 5.2. Configuration Sources

The currently active configuration layers are:

- built-in defaults
- user configuration file

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- project file
- environment overrides for selected settings
- command-line overrides
- runtime/API overrides

Higher-priority sources override lower-priority sources.

In general:

- project values override the user configuration file
- command-line options override project and user configuration
- runtime/API changes override all startup configuration

## 5.3. YAML Format

Configuration values are looked up by slash-separated paths such as `logging/stderr`. Each path component maps to one YAML level.

For example, the following YAML sets three TEMU configuration values:

```yaml
logging:
  stderr: true

license:
  checkout-script: /home/user/.config/temu/license-checkout.temu

python:
  lib3_13: /opt/python3.13/lib/libpython3.13.so
```

## 5.4. Scalars and Lists

The configuration system supports both scalar values and string/path lists.



List-valued configuration entries are new in TEMU 5.0.

Scalar values include:

- `bool`
- `string`
- signed integers
- unsigned integers
- floating-point values

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

List values are used for search paths such as plugin, script, Python, executable, and OS-awareness configuration paths.

List-valued entries may be written either as a plain sequence or as an operation mapping:

```
paths:
  executables:
    prepend:
      - build/images
    append:
      - deploy/images
```

The supported list operations are:

- `replace`
- `prepend`
- `append`
- `remove`

## 5.5. Access Patterns

The configuration API provides these lookup functions:

- `temu_configGetBool()`
- `temu_configGetString()`
- `temu_configGetSigned()`
- `temu_configGetUnsigned()`
- `temu_configGetDouble()`
- `temu_configGetStringList()`

> `temu_configGetStringList()` is new in TEMU 5.0.

The TEMU command line also exposes helper commands for inspecting values:

- `config-bool <path>`
- `config-str <path>`
- `config-int <path>`
- `config-uint <path>`
- `config-float <path>`
- `config-strlist <path>`
- `config-pathlist <path>`

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

## 5.6. Queried Configuration Variables

The following table lists the most visible built-in configuration variables and their typical effect.

*Table 4. Selected Built-In Configuration Variables*

| Path | Type | Default | Description | Related override |
|---|---|---|---|---|
| `gdb/log-commands` | `bool` | `false` | Enables extra logging of GDB remote protocol commands in the built-in GDB server. | `TEMU_GDB_SERVER_LOG_COMMANDS` |
| `license/checkout-script` | `string` | not set | If set, TEMU executes the referenced command file with immediately after successful license initialization. | none |
| `logging/stderr` | `bool` | `false` | Selects stderr instead of stdout for the standard text logging backend. | `TEMU_LOG_STDERR`, `--log-stderr` |
| `logging/with-cycles` | `bool` | `false` | Uses cycle counts instead of wall-clock timestamps in text logging. | `--log-with-cycles` |
| `logging/file/mode` | `string` | `disabled` | Controls file logging. Supported values are `disabled`, `explicit`, and `auto`. | none |
| `logging/file/path` | `string` | not set | Explicit path for file logging when `logging/file/mode` is `explicit`. | none |
| `logging/file/timestamp-format` | `string` | `%Y-%m-%dT%H-%M-%S` | Format string used when auto-generating log filenames. | none |

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

| Path | Type | Default | Description | Related override |
|---|---|---|---|---|
| `models/mec/debug` | `bool` | `false` | Enables extra debug behavior for the MEC model and emits an informational message when the device is created. | `TEMU_DEBUG_MEC` |
| `plugins/loader/debug` | `bool` | `false` | Enables debug tracing in the plugin loader, for example when searching plugin paths and resolving library names. | `TEMU_DEBUG_PLUGIN_LOADER` |
| `python/lib3_13` | `string` | not set | Overrides the shared-library path used when TEMU loads Python 3.13 support with `dlopen()`. If unset, TEMU tries a built-in list of candidate paths. | none |
| `python/paths` | `path list` | empty | Additional Python module search paths. | `--python-path` |
| `plugins/paths` | `path list` | `.` plus installation defaults | Additional plugin search paths. | `--lib-path`, `plugin-append-path`, `plugin-prepend-path`, `plugin-remove-path` |
| `scripts/paths` | `path list` | `.` plus installation defaults | Script search paths used for `exec` and other script resolution. | `--script-path` |
| `paths/executables` | `path list` | `.` | Search paths for resolving relative executable and image file names. | none |

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

| Path | Type | Default | Description | Related override |
|---|---|---|---|---|
| `paths/os-awareness-configs` | `path list` | `.` | Search paths for resolving relative OS-awareness YAML files. | none |
| `runtime/root` | `path` | not set | Base directory for generated runtime state. | none |
| `runtime/logs` | `path` | not set | Base directory for generated log files. | none |
| `runtime/snapshots` | `path` | not set | Base directory for snapshot files. | none |
| `snapshots/json/debug` | `bool` | `false` | Enables debug logging while deserializing JSON snapshots, including object creation and property restore progress. | `TEMU_DEBUG_SNAPSHOTS` |
| `assembler/aliases` | `bool` | `true` | Enables instruction aliases (also known as pseudo instructions) when assembling and disassembling target instructions. | The `config.assemblerAliases` property of CPUs. |

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

| Path | Type | Default | Description | Related override |
|---|---|---|---|---|
| `dbt/instruction-fusion` | `bool` | `true` | Enables instruction fusion in the TEMU's dynamic binary translator. This improves performance in cases such as RISC-V's auipc/lui + addi. Only leads to observable differences in execution for MMIO accesses scheduling an event within the same TEMU translation block. | none |
| `dbt/handle-event-within-block` | `bool` | `true` | Causes the binary translator to handle event scheduled within a translation block the same way as the interpreter, improving reproducability. Degrades performance of the binary translator slightly. | The `config.handleEventWithinBlock` property of CPUs. |

## 5.7. Defaults and Fallbacks

Each built-in configuration variable may have a registered default value.

If no active configuration source provides a value, the registered default is used.

The fallback parameter passed to `temu_configGet*()` is only used if:

- the key has no active configured value, and
- the key has no registered default

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

## 5.8. Related Documentation

- Chapter 7
- Chapter 11
- Chapter 15
- OS Awareness
- Chapter 21

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 6. Migration Guide

This page collects migration guidance for users moving between TEMU API and configuration styles.

The intent is to keep legacy code and scripts working where practical, while still encouraging migration to the newer interfaces.

## 6.1. C API Compatibility Headers

The public C API includes compatibility headers for older object-typing conventions:

- `temu-c/Support/Temu2Compat.h`
- `temu-c/Support/Temu3Compat.h`

These headers are included from core public headers such as `temu-c/Support/Objsys.h`.

The compatibility mode is controlled by defining `TEMU_TYPE_ERASED_OBJECTS` before including the TEMU headers.

```
#define TEMU_TYPE_ERASED_OBJECTS
#include "temu-c/Support/Objsys.h"
```

With `TEMU_TYPE_ERASED_OBJECTS` enabled, older code that expects type-erased `void *`-style object and time-source arguments can be kept source-compatible while it is being migrated to the typed `temu_Object *`, `temu_Component *`, and `temu_TimeSource *` forms used by newer TEMU releases.

Recommended migration strategy:

- keep the compatibility macro enabled while first moving to a newer TEMU release
- update model code to use the typed object pointers directly
- remove `TEMU_TYPE_ERASED_OBJECTS` once the codebase no longer depends on the legacy calling convention

See Chapter 12 for the general API compatibility and deprecation policy.

## 6.2. Script and System Migration

Older TEMU setups often construct complete boards through legacy `sysconfig` scripts. Newer setups should prefer YAML components, which are easier to maintain and compose.

For example, instead of constructing a whole machine through a large hand-written script, use:

```
import Component
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
Component.new name=board0 file=components/board.yaml
```

See Chapter 20 for the component model.

### 6.2.1. Object Naming Differences

Component instantiation intentionally namespaces contained objects. For example, a CPU that previously appeared as `cpu0` in a legacy system script may instead appear as `board0-cpu0` when instantiated from a component named `board0`.

To help with migration, TEMU supports object aliases. This allows compatibility wrappers to preserve older object names while still using YAML components as the source of truth.

```
import Component
Component.new name=board0 file=components/gr740.yaml

alias name=cpu0 object=board0-cpu0
alias name=mem0 object=board0-mem
```

This is particularly useful for:

- regression tests that still look up `cpu0`, `mem0`, or similar names
- legacy board scripts kept as thin wrappers around component instantiation
- staged migrations where new code uses component-qualified names and old code still uses legacy names

Recommended migration strategy:

- keep board topology in YAML components
- keep legacy `sysconfig` files only as compatibility wrappers where needed
- use object aliases to preserve old global names when required
- update new scripts and tests to use component-qualified names directly

### 6.2.2. Reset Semantics for Migrated Bundled Sysconfigs

When a bundled legacy `sysconfig` has been migrated to instantiate a YAML component, board-wide reset behavior may move from CPU-local reset fanout to the component's exported reset path, typically via a reset controller inside the component.

In practice, this means that scripts using migrated bundled sysconfigs may need to prefer:

```
reset obj=board0 warm=0
```

instead of:

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
reset cpu=cpu0 warm=0
```

for board-level reset behavior.

This change only applies to the bundled sysconfigs that TEMU has migrated to component-backed wrappers. Customer-owned legacy sysconfig scripts are not affected unless they are updated to use the bundled component-backed wrappers or otherwise adopt the same component/reset-controller structure.

## 6.3. Command Aliasing Compatibility

The `alias` command now supports command, class, and object aliasing:

```
alias name=my_echo command=echo
alias name=LegacyScheduler class=SingleLevelScheduler
alias name=cpu0 object=board0-cpu0
```

Legacy command-alias handling can be controlled through the compatibility configuration key:

`compatibility/commands/alias`

This key is versioned. For command aliasing only:

- values `<= 4` preserve the older behavior where aliasing an unknown command reports a diagnostic but does not fail the script
- values `>= 5` use the newer strict behavior where aliasing an unknown command fails

This setting can be placed in a user configuration file or in a project file.

```
compatibility:
  commands:
    alias: 4
```

See Chapter 5 and Chapter 7 for configuration layering and project scoping.

## 6.4. Recommended Approach

For actively maintained codebases:

- use the typed C API instead of relying on `TEMU_TYPE_ERASED_OBJECTS`
- use YAML components as the primary board description format
- keep compatibility settings and aliases only where they are needed for existing tests, scripts, or customer integrations

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- remove compatibility layers incrementally once dependent code has been migrated

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 7. Project Files

TEMU project files provide a way to apply project-scoped configuration, search paths, runtime settings, and startup scripts with a single file.



Project-file support described on this page is new in TEMU 5.0.

Project files can be loaded:

- from the command line with `temu --project <file>`
- with target selection from the command line using `temu --project <file> --target <name>`
- from the C API with `temu_loadProject(path, target)`

Project files are intended for settings that belong to a runnable project rather than to one user account.

## 7.1. Command Line Usage

To load a project file on startup:

```
temu --project ./temu-project.yaml
```

To load a specific target within the project:

```
temu --project ./temu-project.yaml --target bare-metal
```

When `--project` is given by itself, TEMU loads the project file, runs the configured startup scripts, and then enters interactive mode.

When `--project` is combined with other `--run-*` options or positional scripts, TEMU respects the command-line order. For example:

```
temu ./tests/paths.temu --project ./temu-project.yaml
```

runs `./tests/paths.temu` before the project is loaded, while:

```
temu --project ./temu-project.yaml ./post-startup.temu
```

loads the project first and runs `./post-startup.temu` afterwards.

## 7.2. C API Usage

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

The support library exposes the following function:

```
int temu_loadProject(const char *path, const char *target);
```



`temu_loadProject()` is new in TEMU 5.0.

The function:

- loads the project file
- applies its configuration values with project priority
- resolves project-relative paths
- runs configured startup scripts
- runs target-specific startup scripts when `target` is set

It returns `0` on success and non-zero on failure.

## 7.3. File Format

A project file is a YAML mapping.

The currently supported top-level keys are:

`project`
: Project metadata such as project name, vendor, copyright, and license.

`paths`
: Common project search paths mapped onto the normal configuration system.

`runtime`
: Runtime output locations such as the runtime root, log directory, and snapshot directory.

`config`
: Additional project-scoped configuration values.

`startup`
: Startup scripts to run after the project configuration has been applied.

`targets`
: Optional named target entries that can add target-specific configuration and startup scripts.

### 7.3.1. Example

```
version: 1
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```yaml
project:
  name: foo
  vendor: Vendor Name
  copyright: Acme Inc
  license: Commercial

paths:
  plugins:
    prepend:
      - build/plugins
  temu_scripts:
    prepend:
      - scripts/temu
  python:
    prepend:
      - scripts/python
  components:
    prepend:
      - components
  target:
    prepend:
      - target

runtime:
  root: .temu
  logs: logs
  snapshots: snapshots

config:
  logging:
    file:
      mode: auto

startup:
  temu-scripts:
    - scripts/temu/imports.temu
    - scripts/temu/startup.temu

targets:
  bare-metal:
    startup:
      temu-scripts:
        - scripts/temu/run-bare-metal.temu
```

## 7.4. Project Metadata

The `project` section describes the project itself.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

The currently used keys are:

- `project/name`
- `project/vendor`
- `project/copyright`
- `project/license`

These values are exposed through the configuration system when the project is loaded. The scaffolded plugin template uses this metadata for generated plugin metadata fields.

## 7.5. Path Mappings

The `paths` section is a convenience layer over the normal configuration system.

The currently supported mappings are:

- `paths.plugins` → `plugins/paths`
- `paths.temu_scripts` → `scripts/paths`
- `paths.python` → `python/paths`
- `paths.components` → `components/paths`
- `paths.target` → `target/paths`
- `paths.sources` → `sources/paths`

Each entry accepts either a plain list or explicit list operations:

- `replace`
- `prepend`
- `append`
- `remove`

Example:

```
paths:
  plugins:
    prepend:
      - build/plugins
  components:
    append:
      - third-party/components
```

## 7.6. Configuration Values

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

The `config` section maps directly to normal TEMU configuration keys. Nested YAML keys are flattened using `/`.

For example:

```
config:
  logging:
    stderr: true
```

sets the configuration key `logging/stderr`.

This section is intended for configuration values that do not have a dedicated top-level project-file section.

Examples include:

- `logging/stderr`
- `logging/file/mode`
- `logging/file/path`
- `logging/with-cycles`

### 7.6.1. Scalars

Scalar values in the project file replace lower-priority values from the user configuration file.

### 7.6.2. Lists

List-valued configuration entries may be written either as plain sequences or using list operations.

A plain sequence means `replace`:

```
config:
  plugins:
    paths:
      - build/plugins
      - deploy/plugins
```

The explicit list operations are:

- `replace`
- `prepend`
- `append`
- `remove`

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

## 7.7. Runtime Paths

The `runtime` section configures where generated runtime state is written.

The main keys are:

- `runtime/root`
- `runtime/logs`
- `runtime/snapshots`

Path resolution intentionally differs slightly between these keys:

- `runtime/root` is resolved relative to the project file directory
- `runtime/logs` and `runtime/snapshots` are interpreted relative to `runtime/root` unless set to absolute paths

So with:

```
runtime:
  root: .temu
  logs: logs
  snapshots: snapshots
```

the effective runtime directories are:

- `<project>/.temu/logs`
- `<project>/.temu/snapshots`

Similarly, `logging/file/path` is interpreted relative to `runtime/logs` when file logging mode is `explicit`.

## 7.8. Path Resolution

Relative paths in project files are generally resolved relative to the project file directory for project-owned path values.

This includes, for example:

- `paths.plugins`
- `paths.temu_scripts`
- `paths.python`
- `paths.components`
- `paths.target`
- `project` startup script paths

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- `runtime/root`

This makes project files relocatable as long as their referenced files remain in the same relative layout.

## 7.9. Startup Scripts

The `startup` section can contain:

`temu-scripts`
: TEMU command scripts (`.temu`, `.ts`, or similar).

`python-scripts`
: Python startup scripts.

The aliases `temu` and `python` are also accepted as list keys.

Relative startup script paths are resolved relative to the project file directory.

The startup scripts run after the project configuration has been applied. This means they can rely on:

- project-defined plugin paths
- component search paths
- script search paths
- runtime output directories

## 7.10. Targets

Projects may define an optional `targets` mapping for target-specific startup and configuration.

Each entry under `targets.<name>` may contain the same sections as the main project file, except that it is applied after the project-wide configuration. This allows a single project to share:

- component setup
- common imports
- common paths
- runtime settings

while selecting different payload or scenario scripts with:

```
temu --project ./temu-project.yaml --target bare-metal
```

Target startup scripts are appended after the project-wide startup scripts.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

## 7.11. Relationship to User Configuration

User configuration and project files are separate layers.

- user configuration is intended for user-scoped preferences
- project files are intended for project-scoped runtime setup

Project values override the normal user configuration file, but command-line options still take precedence over project values.

See also Chapter 8, Chapter 5, Chapter 11, Chapter 20, Chapter 15, and Chapter 21.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 8. Project Templates

TEMU can create and extend a project scaffold directly from the command line.

This is intended as a convenient starting point for new projects using project files, startup scripts, YAML components, CMake, and plugin development.

## 8.1. Initializing a Project

To create a new project:

```
temu --init-project foo
```

This creates a new directory `foo/` with a starter layout including:

- `temu-project.yaml`
- `CMakeLists.txt`
- `README.adoc`
- `cmake/`
- `lib/`
- `plugins/`
- `scripts/temu/`
- `scripts/python/`
- `components/`
- `target/`
- `tests/`
- `.temu/`

The generated scaffold includes:

- a project file with project metadata and default runtime settings
- automatic session file logging
- TEMU startup scripts
- a starter YAML component extending `at697f.yaml`
- a CMake build configured for external TEMU usage
- a minimal CTest-enabled sample test

## 8.2. Using the Generated Project

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

The generated project can be opened in place with:

```
temu --project=temu-project.yaml
```

If the project defines named targets, a specific target can be selected with:

```
temu --project=temu-project.yaml --target=bare-metal
```

To configure and build it:

```
mkdir -p build
cd build
cmake -DTEMU_ROOT=/opt/temu/5.0 -G Ninja ../
ninja
ninja test
```

For build-tree TEMU development, `TEMU_ROOT` may also point at a TEMU build directory that exports `TEMUConfig.cmake`.

## 8.3. Generated Startup Files

The scaffold creates two TEMU startup scripts:

`scripts/temu/imports.temu`
: Starts with `import Component` and is used to accumulate plugin imports.

`scripts/temu/startup.temu`
: Initializes the multi-level scheduler, instantiates the default component, and adds its processor to the scheduler.

The default component is:

- `components/board.yaml`

which extends `at697f.yaml` and overrides the default RAM and ROM sizes.

## 8.4. Default Runtime Layout

The generated project file uses:

```
runtime:
  root: .temu
  logs: logs
  snapshots: snapshots
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
config:
  logging:
    file:
      mode: auto
```

This means runtime artifacts are written below the project directory, for example:

- `.temu/logs/`
- `.temu/snapshots/`

## 8.5. Adding a Plugin

To add a new plugin scaffold to an existing project:

```
temu --project=temu-project.yaml --add-project-plugin=Bar
```

If `--project` is omitted, TEMU searches upwards from the current directory for `temu-project.yaml`.

This command:

- creates `plugins/Bar/`
- creates `plugins/Bar/CMakeLists.txt`
- creates `plugins/Bar/Bar.cpp`
- appends `add_subdirectory(Bar)` to `plugins/CMakeLists.txt`
- appends `import Bar` to `scripts/temu/imports.temu`

## 8.6. Generated Plugin Metadata

Generated plugin source files use the current scaffolded metadata pattern:

- vendor from `project.vendor`
- copyright holder from `project.copyright`
- license string from `project.license`
- ABI version from the installed or referenced TEMU headers
- local project version from the project CMake version
- local revision from Git when available, otherwise an all-zero revision string

The plugin source uses:

- `LOCAL_MAJOR_VERSION`
- `LOCAL_MINOR_VERSION`

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- `LOCAL_PATCH_VERSION`
- `LOCAL_GIT_REVISION`
- `TEMU_MAJOR_VERSION`
- `TEMU_MINOR_VERSION`
- `TEMU_PATCH_VERSION`

These local version defines are generated through:

- `cmake/TemuProject.cmake`
- `cmake/LocalVersion.h.in`

## 8.7. Notes

The scaffold is intentionally small and is meant as a starting point. You are expected to adapt:

- the project metadata
- the default component
- the startup scripts
- the build layout
- the generated plugin descriptions

See also Chapter 7, Chapter 20, Chapter 5, and Scripting.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 9. Support and Maintenance

TEMU support and maintenance is at present provided at the following location: https://degitlab-ext.terma.com/termade/temu-issues/temu-issues

It is possible to access the GitLab system for filing support tickets if TEMU is under warranty.

The GitLab system can be used to file bug reports, and user support requests.

Normally, TEMU is licensed with one year of support. Subsequent support and maintenance can be acquired by purchasing a maintenance agreement from Terma.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 10. Time

There are two types of time in a simulator like TEMU: *simulated real-time (SRT)* and *wall-clock time (WCT)*.

**SRT**

Simulated Real-Time is the simulated time. Simulated time can be frozen by stopping / pausing the simulator. It only advances when the simulator is running.

**WCT**

Wall-Clock Time is the physical time which progress independent of whether the simulator is paused or not. It thus represents the time as would be shown on a wall-clock.

TEMU *SRT* is maintained in steps. A step is the execution of one instruction.

This is combined with a conversion parameter and its inverse: *Cycles per Instruction (CPI)* and *Instructions per Cycle (IPC)*.



This differs form TEMU 2 and 3, where simulated time was maintained in cycles. The changes of time has been introduced for several reasons, primarily to better support super-scalar and multi-core processors. TEMU 2 and 3 modelled individual instruction timing, but this is no longer done in TEMU 4.

Simulated time is provided by a time source, and all models have a designated primary time source.

Time sources may have parent time sources for posting synchronized events.

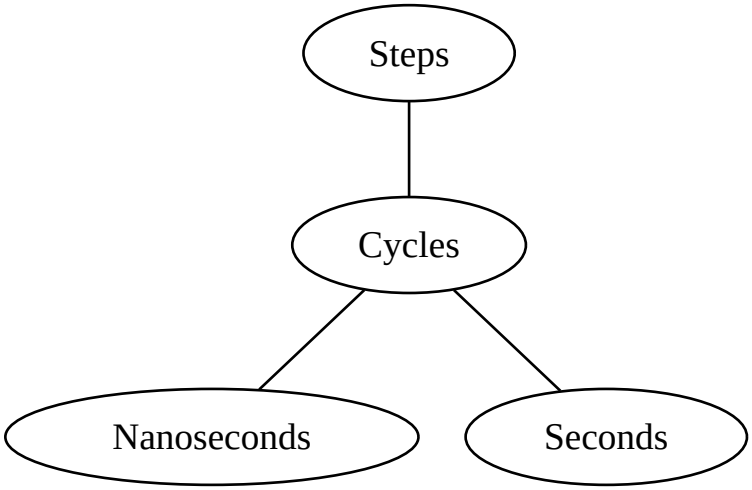
Time sources are special models that currently cannot be developed by the end user of TEMU, these currently include the following models:

- Processors
- Clocks
- Scheduler

## 10.1. Time Base

TEMU utilizes several notions of time that can be converted between each other.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.




### 10.1.1. Steps

A step is the execution of an individual instruction. Internally, the processor keeps track of time in steps.

In addition to advancing a step when running an instruction, when the processor is idle, TEMU advances the step counter as much as is needed to trigger the next timed event.



This differs from TEMU 2 and 3, where the step counter in idle mode was advanced by one when advancing the cycle counter to the next event. The change may be noticeable when stepping a processor in idle mode.

It is important to understand, that the TEMU event queues are driven using steps. Hence, the order of execution for events posted e.g. using different cycle or nanosecond offsets, is not defined if they convert to the same step offset.

### 10.1.2. Cycles

While steps is a notion of how many steps a processor has been executed, cycles is a direct measurement of time. Cycles is related to normal time (nanoseconds, seconds etc) using the time source's clock frequency.

The relation between steps and cycles is expressed using the *cycles per instruction (CPI)* and *instructions per cycle (IPC)* properties.

The CPI and IPC is currently expressed internally as *8:8* fixed point numbers. Though the exposure to the user is via double-properties. By default, TEMU 4 use a 1 cycle per instruction model.

This means that it is possible to adjust the cycles per instructions to any value in the range of [0, 256).

### 10.1.3. Nanoseconds

Nanoseconds is an integer value in TEMU. That means that sub-nanosecond time intervals cannot be expressed in TEMU nanoseconds.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

Sub nanosecond times must be expressed in seconds or cycles (with a higher clock frequency).

### 10.1.4. Seconds

Seconds in TEMU is a double precision floating point number. It is related to the cycle time using the clock frequency.

## 10.2. Temporal Decoupling

Processors in TEMU run temporally decoupled. That is, each processor maintain and advance its own time individually. In order to avoid running too much ahead of other processors, time is synchronized at regular intervals. This interval is known as the *time quantum*.

### 10.2.1. Time Quantum

The *Scheduler* is configured using a time quantum. When setting the quanta sizes, one need to consider emulator performance, simulated software behavior and other factors.

In general, a large quantum means that less overhead for managing time is spent. But on the other hand, large quanta may have an effect on the delivery of inter-processor interrupts. This in turn may impact the time of certain software operations.

The *Scheduler* model is configured using a quantum in steps.



There is no general "best quantum", it has to be experimentally determined. A good rule of thumb and strategy is to start with a quantum of 10000 steps and adjust it up and down for the highest performance using a binary search.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 11. Command Line

The command line interface (CLI) is easy to use and provides built in help for different commands. To start the command line tool do the following (assuming you are running bash).

```
# Set the PATH to include the temu command line application
$ PATH=/opt/temu/bin:$PATH

# Start TEMU
$ temu
no such file: '~/.config/temu/init'
no such file: './temu-init'
temu>
```

As can be seen above, the command line tool complains about two missing files. These are nothing to bother about at the moment. The files are used to automatically run a set of commands when you start the `temu` tool.

It is possible to get a list of commands by typing `help`. Help for an individual command (including lists of arguments the command takes) can be produced by typing `help CMDNAME`.

## 11.1. Command Line Interface Options

The command line interface support the execution of non-interactive batch sessions via the `--run-*` flags. Multiple run flags can be given to have scripts executed in order. On the first error in a script, TEMU will terminate and not proceed with the next script.

`--run-commands [filename]`

Run the TEMU command-script (CLI-script) in the given file in non-interactive mode. You can provide this option multiple times to execute multiple scripts in sequence. When the last script finishes, the emulator will quit.

`--run-command-string [cmd]`

Run [cmd] as a single command as if typed in the interactive command line.

`--run-script [filename]`

Run the Python script in the given file in a non-interactive TEMU-session. The option can be provided multiple times, and scripts will be executed in the sequence they are given on the command line.

`--interactive`

Enter interactive mode after processing `--run*` flags. Note that any failed scripts will still terminate TEMU before the interactive mode is entered.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

`--install-license [filename]`

Install a license file.

`--revoke-license [license-id]`

Revoke an installed license.

`--list-licenses`

List all installed licenses

When running both CLI scripts and Python scripts, the order will be as specified in the arguments to TEMU. It is possible to run a Python script first, followed by a CLI script or the other way around.

It is also possible to specify a list of scripts without the `--run-commands/--run-script` flags above, in that case the file type is inferred by the file extension, where the extension `.temu` will be treated as a TEMU script and the `.py` extension as a Python script. Passing file names this way will also result in a non-interactive session.

## 11.2. Command Syntax

Normally commands are named by a *noun-verb* format (but there are abbreviations as well). Commands take either a set of named arguments, but some (like the help command) also take positional arguments. In the named format, each argument is separated by a space, and defined using key-value pairs as e.g. `help command=memory-assemble`.

## 11.3. Variables

The command line allows for variables to be set. These can be set using the `var-set` command. Variables are expanded if they are given as argument values to commands. When used, variables are referenced as `$var` or `${var}`.

## 11.4. Help Command

Each command is self-documenting, typing help will show a list of available commands. Typing `help command=memory-assemble` will show the detailed help for the `memory-assemble` command, including all arguments and their types.

## 11.5. Commands

This section list some of the commands provided in the CLI. A full list can be generated by running the help command.

### 11.5.1. Snapshot Commands

There are two commands for working with snapshots, the save and restore command.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**snapshot-restore**

This command restores a serialized snapshot from a file. The read file should be a JSON file written by the save command.

**snapshot-save**

The save function writes a snapshot in JSON format to disk. Memory content is typically dumped as raw data in a binary blob (in an auxiliary file). The endianness of this blob is for RAM and ROM contents in the standard models is in host order where the unit size is the word size of the target. For the SPARCv8 target on an x86-64 host this means that the data is stored as sequence of little-endian 32-bit words.

### 11.5.2. Memory Commands

**memory-assemble**

This command assembles a string into memory.

**memory-disassemble**

This command disassembles memory contents. As assemblers are target dependent, the command takes a CPU parameter.

**memory-load**

Load executable file (SREC or ELF format). The Command automatically detects the format of the file.

**memory-read**

Read memory and write it to the console.

**memory-write**

Modify memory content.

**memory-map**

Map object to memory space. The command assigns an object to an address range in the memory space.

### 11.5.3. Object Commands

When dealing with the emulator object system in the CLI, there are a number of commands that are useful. These include the following:

**Class constructor commands**

Creates an object using the `<Class>.new` command generated for each registered class. The `name` parameter indicates the object name; this name should be unique. Additional arguments are class specific, consult the class documentation on the allowed arguments. Example `Leon3.new name=cpu0 CPUId=0`

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**object-connect**

Connect two objects together. The command connects an object reference property to an interface provided by another object. The command takes two parameters, parameter *a* is the property formed as `objname.propname`, parameter *b* is the interface reference that the property should refer to, this is formed as `objname:ifacename`. Examples: `connect a=cpu0.memAccess b=cpu0:MmuMemAccessIface`, `connect a=cpu0.memAccessL2 b=mem0:MemAccessIface`

**object-info**

This command prints the properties in an object.

**object-list**

List the names of all created objects.

**object-prop-write**

In order to assign property values using the property read and write mechanism this command provides that functionality. Depending on the model, a write may have side-effects (by invoking a write handler), side-effects are documented in the model manuals.

### 11.5.4. Plugin Commands

There are several commands in the CLI that helps you deal with and to load plugins. All of these commands have the prefix `plugin-`.

**plugin-append-path**

Add path to plugin search paths

**plugin-load**

Load a plugin

**plugin-remove-path**

Remove path from plugin search path

**plugin-show-paths**

Print the search paths for plugins

**plugin-unload**

Unload a plugin

### 11.5.5. Execution Commands

**run**

Run the scheduler or CPU for a given time

**step**

Step the scheduler or CPU for a given number of steps

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**trace**

Trace the scheduler or CPU for a given number of steps, each executed instruction is disassembled and printed out.

### 11.5.6. Other Commands

**quit (exit, q)**

Quit TEMU

**help**

Show help

**version**

Show version number

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 12. Libraries

The principal library is `libTEMUSupport.so`. Normally, you never need to directly link to any other library. Remaining libraries which implement CPUs and models, are loaded either in the command line interface by using the `plugin-load` or its alias `import`, or by `int temu_loadPlugin(const char *Path)` which is defined in `temu-c/Support/Objsys.h`.

To use the emulator as a library, simply link to `libTEMUSupport.so` and initialize the library with `temu_initSupportLib()`. The function will among other things ensure that there is a valid license file for you machine. In case there is no valid license file available, the function will *terminate your application*.

```
#include "temu-c/Support/Init.h"

int
main(int argc, const char *argv[argc])
{
  temu_initSupportLib(); // Initialise the TEMU library
  return 0;
}
```



`temu_initSupportLib()` will terminate your application if there is no valid license file on the system.

## 12.1. Deprecation Policy

TEMU versions are numbered as *Major#.Minor#.Patch#*. I.e. *2.0.1* is a bug fix for major version 2, minor version 0.

### 12.1.1. TEMU 2

This policy is in effect starting with TEMU 2.0.0 (and applies to the C-API). The policy will not change unless the major version is incremented.

Patch version increments are for bug fixes and they will be ABI compatible with previous releases of the same major-minor release (you will not need to recompile your models for them to remain functioning).

Minor version increments will remain source level API compatible, but may deprecate functionality and APIs. Deprecated APIs will be marked as such with GCC / Clang deprecation attributes and noted as deprecated in the release notes. Recompilation of user defined models is recommended as ABI may break (e.g. extra functions at the end of interfaces). Minor versions typically add non-invasive features (more models, additional simple API functionality etc).

Major version increments will remove deprecated functions and APIs. Although, models written using the C-API should in general remain compatible, however, no 100-percent-guarantee is made

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

for this. Major versions can add substantial new features.

### 12.1.2. TEMU 3

As of TEMU 3, the new versioning policy is semantic versioning.

### 12.1.3. Clarification on C++ APIs

At present, any public C++ APIs should be seen as unstable and subject to change without notice.

## 12.2. Experimental Application Programming Interfaces

New API functionality is introduced at regular intervals to help the end user of the system. While simple APIs will be introduced directly, more complex functionality is likely to go through an experimental release cycle (sometimes more than one). For example, source debugging support is being worked on at the time of writing. This is expected to first appear in the command line interface, followed by exposing some functionality via APIs, when these APIs are public, they will be marked as experimental with comments in the headers. Experimental means that the API is subject to change in ways that may be source incompatible, even between patch releases (e.g. between 2.0.0 and 2.0.1).

This way, new APIs can be introduced for public review, and be adapted based on user input.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 13. Object System

TEMU provides a light weight object system that all built in models are written in. The object system exist to provide a C API in which it is possible to define classes and create objects that support reflection / introspection. Conceptually this is similar to GOBJECT, but the TEMU object system is more tailored for the needs of an emulator and a lot simpler. There is also some correspondence to SMP2, but the interfaces are plain C which is needed in order to interface to the object system from the emulator core.

The key features of the object system are the following:

- Standardized way for defining classes and models in plain C.
- Ability to introspect models, even though they are written in C or C++.
- Automatic save and restore of state
- Access to object properties by name using scripts
- Standard way for defining interfaces (such as serial port interfaces etc)
- Easy to wrap in order to be able to write models in other languages (e.g. Python)

The object system accomplishes this by providing the following:

**Class**

Blueprint for objects, classes are created, registering properties and interfaces. Classes have names starting with a letter or underscore followed by any number of letters, digits and underscores.

**Object**

An instantiated class. Normally the TEMU object system takes care of instantiation, however externally created objects can also be registered with the object system (in order to have scripts build the object graph with external classes). Objects have names with the same naming rules as classes, except objects support object name separators with the minus character '-'. This is used to ensure objects inside components have unique global names.

**Property**

A named data member of a class (i.e. a field or instance variable). A property is accessible by name (e.g. using strings) and will be automatically serialized by the object system if needed. The system supports all basic fixed with integer types (from `<stdint.h>`), pointer sized integers (i.e. `uintptr_t` and `intptr_t`), floats, doubles and references to objects and interfaces. Property names start with a letter or underscore, followed by any number of digits, letters or underscores. Property names can be nested using a period '.' as separator.

**Pseudo Property**

A named data member of a class without directly backing storage. Pseudo properties are accessible by name, and will be automatically serialized by the object system if they have a set and get function associated. Pseudo properties are useful for custom check pointing logic (e.g.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

writing out raw data to a file) or for using external classes, or non standard layout types in the object system.

**Interface**

A collection of function pointers allowing classes to provide different behavior for a standardized interface. Similar to an interface in Java or an abstract class in C++. In TEMU this is implemented as structs of function pointers that are registered to a class.

**Port**

A property and interface with an inverse relationship. Connecting an interface property in the port to an interface in another object will automatically introduce a back link from the destination object to the original source object. If a port has been added combining property a and interface b in the source object, and property c and interface d in the destination object. Connecting `a→d` will introduce `c→b` automatically.

When setting up a simulator based on TEMU, the general approach is the following:

1. Create all the needed classes (e.g. load plugins)
2. Create all objects for the system (e.g. CPUs, ROM, RAM, MMIO models etc)
3. Connect objects (build the object graph)
4. Load target software in to RAM or ROM
5. Run the emulator

It is possible to query a class or object for properties and interfaces at runtime by specifying the property or interface name as a string.

For example there is a CPU interface that is common to all CPU models, this contain procedures for accessing registers. In addition, there is a SPARC interface which provides SPARC specific procedures (e.g. accessing windowed registers).

The most important core interfaces are the following:

- `MemAccessIface`
- `CpuIface`

An interface can be queried using the `temu_getInterface()` function. This function takes an object pointer as first argument and the interface name as second.

*Listing 6. Getting the Memory Interface for a Processor*

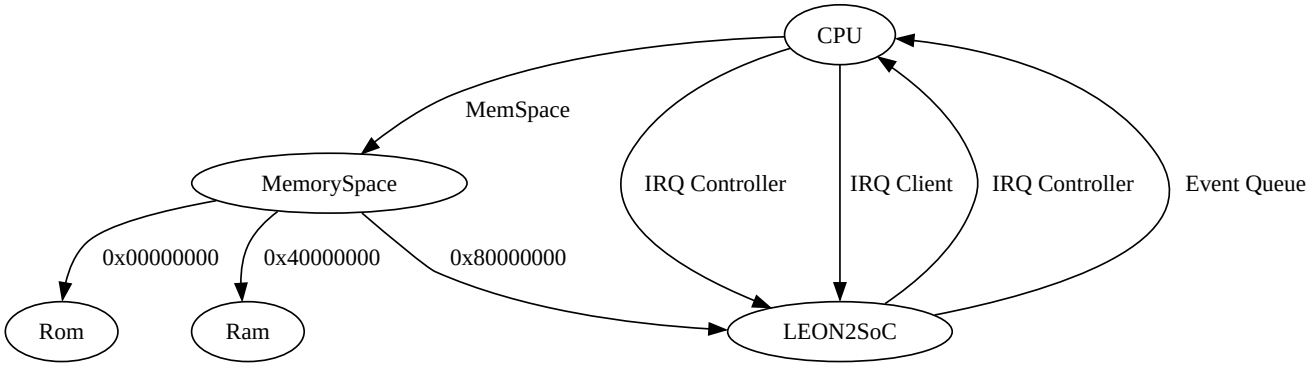
```
temu_Object *cpu = temu_objectForName("cpu0");
temu_MemAccessIface *iface = (temu_MemAccessIface*)temu_getInterface(cpu,
"MemAccessIface");
```

You need to cast the interface pointer to the correct type. The type mappings are provided in the model manuals.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

## 13.1. Object Graph and Interface Properties

The objects created in the object system are connected together by linking *interface properties* to actual interfaces. That is if an object A has an interface property, this interface property can refer to an interface implemented by some other object B. Under the hood this is a pointer pair with an object pointer and an interface pointer, the interface pointer is a pointer to the struct of function pointers implementing the relevant interface.




## 13.2. Object System Functions

This section lists the most important object system functions. The full documentation is in Doxygen based documentation, this is just a quick way to have an overview.

*Table 5. TEMU Object System Functions*

| Function | Description |
|---|---|
| `temu_addInterface()` | Add interface to class |
| `temu_addPort()` | Bind a property / interface pair as a port |
| `temu_addProperty()` | Add property to class |
| `temu_addPseudoProperty()` | Add pseudo property to class |
| `temu_checkSanity()` | Look for unconnected interface properties |
| `temu_classForName()` | Get a class object by name |
| `temu_classForObject()` | Get the class object for an object |
| `temu_connect()` | Connect an interface property to an interface |
| `temu_createObject()` | Create a new object from an internal class |
| `temu_deserialiseJSON()` | Restore the state of the emulator |
| `temu_disposeObject()` | Delete object |
| `temu_getInterface()` | Get interface pointer by name |
| `temu_getValue()` | Get property without side-effects |
| `temu_loadPlugin()` | Load a TEMU plugin |
| `temu_nameForObject()` | Get the name for the given object |

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

| Function | Description |
|---|---|
| `temu_objectForName()` | Get a named object |
| `temu_objsysClear()` | Delete all objects and classes |
| `temu_readValue()` | Get property by calling the read function |
| `temu_registerClass()` | Create a new class |
| `temu_serialiseJSON()` | Save the state of the emulator |
| `temu_setTimeSource()` | Set time source for object |
| `temu_setValue()` | Set property without side-effects |
| `temu_writeValue()` | Set property by calling the write function |

## 13.3. Properties

Properties are registered fields in a class. They are associated with the type and not with the object instance themselves. Properties have names, types and read- and write functions. Properties are saved in snapshots. Properties are associated with an offset in the model type, meaning they have backing storage.

Property names are legal if they start with a letter or underscore followed by any number of letters, digits or underscores. Properties support nesting via dots as well.

## 13.4. Pseudo Properties

Pseudo properties are properties without explicit backing storage, instead they are registered with not only the read and write functions, but also optional set and get functions. Set and get functions serves the same effect as accessing the raw data in a struct, and are used for snapshots. From the user interface point of view, pseudo properties behave the same as normal properties.

## 13.5. Interfaces

Interfaces are structs populated with function pointers. You can query an interface by name for a given object using `temu_getInterface()`.

Interface names are legal if they start with a letter or underscore followed by any number of letters, digits or underscores.

## 13.6. Ports

A very common case is where a source model is connected to a destination model, and the destination model must have a back link to the source model often to a different interface. For example, the IRQ interfaces have an upstream variant `IrqIface` and a downstream variant `IrqClientIface`. The upstream variant is used to raise interrupts, while the downstream interface have functions to acknowledge interrupts. To avoid the case where the user forgets to insert the

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

backward link, it is possible to pair interface properties and interfaces together using `temu_addPort()`.

When a port has been added to a class, the connect function will automatically insert the back links if connecting a port in a source object a port in the destination object.

In the interrupt controller case, assume that the class A has an interface reference property `irqController` and an interface `IrqIface` and class B an interface reference property named `irq` and an interface `IrqClientIface`. Then, the user would do the following:

```
temu_addPort(A, "irqController", "IrqIface", "downstream IRQ port");
temu_addPort(B, "irq", "IrqClientIface", "upstream IRQ port");

// Now normally without ports two connects would be needed
//temu_connect(a, "irqController", b, "IrqClientIface");
//temu_connect(b, "irq", a, "IrqIface");

// With ports, only one connect is needed, it will automatically
// add the reverse link. so, we get both of the links on one
// connect:
//   a.irqConroller -> b:IrqClientIface
//   b.irq -> a:IrqIface
temu_connect(a, "irqController", b, "IrqClientIface");
```

To list available ports in a class use the `class-info` command.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 14. Running the Simulated System

TEMU models and functionality is primarily built around the *TEMU Object System*.

Runnable items are no different, and they are implemented following a common type called a *Time Source*.

A time source is responsible for tracking its own time, clock rate, an event queue and a parent time sources.

It is possible to run any time source in TEMU, this includes:

- Clocks
- Processors
- Schedulers



The Machine model was removed in TEMU 5, following previous deprecation.

## 14.1. Schedulers

Schedulers in TEMU are the key driver of execution. There can only be one scheduler instantiated at any one time.

There are two schedulers at the moment:

- The initial multi-threaded scheduler implemented by the `Scheduler` class.
- A new multi-level scheduler which is required for reverse execution.

It is up to the user to select the scheduler by running one of the following lines:

```
// To pick the TEMU 4 single-level scheduler, run this line
init-scheduler model=single-level

// To pick the TEMU 5 multi-level scheduler, run this line:
init-scheduler model=multi-level
```

Alternatively, the C-API can be used:

```
temu_createScheduler(teSV_SingleLevel);
temu_createScheduler(teSV_MultiLevel);
```

### 14.1.1. Single-Level Scheduler

The single-level scheduler needs to be configured by e.g. setting the thread count and quanta.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

*Listing 7. Configuring a Single-Level Scheduler*

```
init-scheduler variant=single-level
sched.set-frequency frequency=cpu0.freq
sched.set-quanta quanta=20000

# Configure scheduler to run in 2 threads,
# binding cpu0 and cpu2 to thread 0,
# and cpu1 and cpu3 to thread 1.
sched.set-threads threads=2
sched.bind-to-group group=0 cpu=cpu0
sched.bind-to-group group=1 cpu=cpu1
sched.bind-to-group group=0 cpu=cpu2
sched.bind-to-group group=1 cpu=cpu3

# In single threaded mode, the processor will ignore flush
# instructions, but in multi-threaded mode we need to
# treat this as a proper instruction cache synchronization.
cpu0.config.exitOnSync = 1
cpu1.config.exitOnSync = 1
cpu2.config.exitOnSync = 1
cpu3.config.exitOnSync = 1

# The memory space must collect invalid page lists in multi-threaded mode,
# in single threaded mode the pages can be purged when written directly.
mem0.config.processorHasSync = 1
```

### 14.1.2. Multi-Level Scheduler

The multi-level scheduler supports different synchronization domains. Each domain has its own quanta, and may have one or more threads in it. Domains running processors are called leaf-domains, and leaf domains have parent domains. The top domain is called the root-domain. Domains thus form a tree, where the root domain defines a global synchronization quanta.

Different domains can handle e.g. separate multi-core processors. It is also possible to define systems that run separate processors in parallel, while maintaining full determinism.

*Listing 8. Setting up the Multi-Level Scheduler*

```
init-scheduler variant=multi-level

// Insert processors in scheduler
sched.add-cpu cpu=cpu0
sched.add-cpu cpu=cpu1
sched.add-cpu cpu=cpu2
sched.add-cpu cpu=cpu3

// Add scheduler domain
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
sched.add-domain id=1 quantum=10000
sched.set-domain-frequency id=1 frequency=cpu0.freq

// Attach CPUs to domain
sched.assign-cpu-domain cpu=cpu0 domain=1
sched.assign-cpu-domain cpu=cpu1 domain=1
sched.assign-cpu-domain cpu=cpu2 domain=1
sched.assign-cpu-domain cpu=cpu3 domain=1
sched.set-domain-workers id=1 workers=1

// Ensure CPU signals instruction sync to scheduler
cpu0.config.exitOnSync = 1
cpu1.config.exitOnSync = 1
cpu2.config.exitOnSync = 1
cpu3.config.exitOnSync = 1
mem0.config.processorHasSync = 1
```

## 14.2. Running, Stepping and Tracing

TEMU distinguishes between running and stepping a system.

Running implies we run the complete system. Stepping and tracing, implies we run the complete system, but we focus on a specific processor.

In single threaded mode, we run each processor in round-robin order. The same is done when stepping. However, when stepping, the processor under control is the last to run, meaning that processors after the one under control will still be at the beginning of their quanta.

Note that when stepping it may overrun of its quanta. This will in turn result in all other processors running their complete quanta. This way the behavior between run and step is identical.

Tracing is identical to stepping, except the last executed disassembled instruction is printed out.

Note that although the run, step and trace commands support working with single CPUs, this is not the normal approach for running a system.

```
// Run scheduler for 1 second
run time=1.0

// Step CPU0 two steps
step cpuid=0 steps=2

// Trace CPU1 ten steps
trace cpuid=1 steps=10
```

### 14.2.1. API

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

*Listing 9. Running an Already Created Scheduler in C*

```
// Run the scheduler for a 2 seconds
temu_runSecs(2.0);

// Step the cpu (which is a temu_TimeSource pointer) for 10 steps.
temu_step(cpu, 10);

// Stop the currently running scheduler and wait until it is stopped.
temu_stop();

// Stop the currently running scheduler, but return immediately.
// The function is async-safe and can be run in signal handlers.
temu_asyncStop();
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 15. Logging System

TEMU has a built in logging API for emitting logging messages. Logging messages are emitted with a C-style formatting string.

Messages are organized using firstly a category and secondly a severity.

## 15.1. Logging Backends

The TEMU logging system is constructed around a single logging frontend (which is not visible to users), connected to a logging backend.

Several backends exists in TEMU. It is not possible for users to add their own backends, however generic backends can be added by Terma on request.

The current logging backends include:

- Stdout logging backend
- Stderr logging backend
- Legacy logging function backend
- Advanced logging function backend

### 15.1.1. Stdout and Stderr Backends

The default backend is the stdout backend. The stderr backend is selected when launching `temu` with the `--log-stderr` command line flag.

### 15.1.2. Legacy Function Backend

The legacy function backend is enabled using the `temu_logSetFunc()` function:

```
void
myLoggingFunc(const char *msg) {
  printf("%s", msg);
}

temu_logSetFunc(myLoggingFunc)
```

The function receives a complete message string, including time stamp, source object name, category, severity AND a terminating linefeed.

### 15.1.3. Advanced Function Backend

The advanced function backend is a modernized logging backend, leaving more choices to the user.

It also lets you pass an arbitrary pointer as an extra argument.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

It is enabled by calling the `temu_logSetAdvancedFunc()` function.

```
void
myAdvancedLoggingFunc(void *userData, temu_Object *source,
                      unsigned category, temu_LogLevel severity,
                      const char *msg)
{
  auto logger = reinterpret_cast<ILogger*>(userData);

  if (source) {
    logger->emitMessage(source->Name, msg);
  }
}


// ILogger *logger = ...;
temu_logSetAdvancedFunc(myAdvancedLoggingFunc,
                        &logger);
```

The advanced logging backend does *not* append a linefeed after the message.

## 15.2. Categories

Logging categories exist to organize messages and hint on the reason for the message being emitted.

Categories also provide the option of filtering using severity level using a per category basis.

In the text log (*stdout* or *stderr*),

The following categories are built in:

**default**

The default category is not named when emitting messages.

**sim**

The sim category is for various simulation related issues. E.g. if a device logs the reception of a packet from another model, the message should probably be in the *sim* category.

**target**

The target category indicates that the message is triggered by target software, i.e. .

**config**

The config category is intended to be used when errors are detected in the configuration of a model. E.g. if the user configures the model to raise an invalid interrupt number.

Categories are identified using an integer, and the first 8 categories are reserved for TEMU

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

(although only the 4 above are defined at the moment). The default categories are all global at present.

An additional 8 categories can be defined by the user in a per class basis.

This is done using the `temu_addLoggingCategory()` function when the class is being registered.

## 15.3. Severity Levels

Severity levels allows the filtering and handling of messages. On stdout/stderr messages are color coded for the terminal, with red indicating an error, orange a warning etc.

The following severity levels are defined:

**fatal**

A fatal error occurred, message is printed and then the program is aborted.

**error**

An error occurred, but TEMU will keep on running. E.g. target software writes an illegal value to a register.

**warning**

Something was detected that could be a problem, but is not necessarily that.

**info**

Informational messages.

**trace**

High volume trace messages.

**debug**

Debugging messages, these are *NOT* emitted if the model emitting these is compiled with `-DNDEBUG=1`. The `+asserts` configuration of TEMU will not emit debug messages.



The debug severity messages are emitted with an inline function. This lets the compiler optimize away the calls when `-DNDEBUG` is enabled.

## 15.4. Filtering

TEMU applies two filters on a per category basis.

The first filter that is applied is the per object filter. The per object filter bits are stored in the `LoggingFlags` field in `temu_Object`.

If the message is passed on through the per object filter, the global filter is applied which is also per category.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

Filtering is typically controlled using the TEMU command line or `TScript` files using the `log-level` command.

However there is an API for controlling filtering as well.



Debug messages are controlled with the normal filters, but they are never emitted if `-DNDEBUG` is passed to the compiler. This setting is per compiled file in a model.

```
// Set global log level for all categories to warning
// Info messages or below will not be emitted.
temu_logSetLevel(teLL_Warning);

// Set the per object log level in myObj to error, for the sim category.
temu_objectSetLogLevel(myObj, teLC_SimCat, teLL_Error);
```

In the example above, after running the two lines:

- All messages, independent of source object will be dropped if logging level is *info* or below using the global filter.
- Messages from *myObj*, in the *sim* category will be dropped if the level is warning or below. Other messages from *myObj* will be filtered by the global level.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 16. Events

The event API is used to provides a common interface for pushing timed events on the event providers. The API is defined in `temu-c/Support/Events.h`. The API provides functions for posting events on CPU objects, and provides the ability to post in three different time bases (cycles, nanoseconds and seconds), and the ability to decide if events are synchronized on a single CPU or the parent machine object.

When posting events to a CPU, nano-second events are converted to cycles. This means that you will actually not have NS accuracy for the events. The accuracy is a function of the clock frequency. I.e. for a 100 MHz CPU, the accuracy is 10ns, while a 50 MHz CPU has an event posting accuracy of 20ns.

When posting machine synchronized events, the delta time must be at least in the next time quanta. If not, the event will be slipped to the start of the next quanta (and a warning will be printed to the log).

In the case synchronized events are used, the machine scheduler will adjust its quanta length to ensure that CPUs do not execute longer than needed. Note that synchronized events are executed after a CPU has returned to the machine object, potentially executing non-synchronized events before the machine event, even if the strictly speaking have a trigger time after.

In addition to the different types of timed events, it is possible to stack post an event, in which case it will be executed after the current instruction finishes (in case of CPU synchronized events), or after the current time quanta finishes in case of machine synchronized events.

Events are prioritized as follows:

- CPU synchronized immediate events (in FIFO order).
- CPU synchronized stacked events (in LIFO order).
- CPU synchronized timed events
- Board or domain synchronized stacked events
- Board or domain synchronized timed events
- Globally synchronized stacked events
- Globally synchronized timed events

```
int64_t temu_eventPublishStruct(const char *EvName,
                                temu_Event *Ev,
                void *Obj,
                                void (*Func)(temu_Event*));

int64_t temu_eventPublish(const char *EvName, void *Obj,
                          void (*Func)(temu_Event*));


typedef enum {
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
  teSE_Cpu,     // Trigger event when CPU reaches timer
  teSE_Board,   // Synchronise event to board / domain
  teSE_Global   // Global time queue
} temu_SyncEvent;

void temu_eventPostCycles(void *Q, int64_t EvID, int64_t Delta,
                          temu_SyncEvent Sync);
void temu_eventPostNanos(void *Q, int64_t EvID, int64_t Delta,
                         temu_SyncEvent Sync);
void temu_eventPostSecs(void *Q, int64_t EvID, double Delta,
                        temu_SyncEvent Sync);
void temu_eventPostStack(void *Q, int64_t EvID,
                         temu_SyncEvent Sync);
```

## 16.1. Events from Other Threads

The event queues in TEMU are protected by mutexes. It is possible to post events from other threads. However, care must be taken in case determinism is required (e.g. trigger time must be in next quanta).

Note that a lone CPU will post a null-event to ensure that the event handlers are triggered at regular intervals and not just when model events are executed.

A possible way to use this capability is when you integrate external hardware / models into your simulator. For example, a separate thread can run and wait on a file-descriptor or socket, when data is available, it reads out the data and posts a thread-safe callback function to be called by the main thread at a safe time. The callback can then take the data that was read from the file-descriptor and inject this over a virtual bus model or write it to emulated memory.

Note that in TEMU 2.2, specific APIs for doing file descriptor and timer monitoring has been added. These are available as the `temu_async*()` functions in the `temu-c/Support/Events.h`-header.

## 16.2. Notifications

A special type of event is an event that does not have a triggering time. They are instead triggered due to some action or event detected due to other logic inside the emulator. These events are posted using a pub-sub mechanism, with strings as the event key. A model or any other user code can listen for an event which is requested with a specific name. Note that when an event is published, it is assigned an integer ID which is used for fast notifications.

The functions for posting and listening to notifications include:

- `temu_publishNotification()`
- `temu_subscribeNotification()`
- `temu_unsubscribeNotification()`
- `temu_notify()`

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- `temu_notifyFast()`

In order to avoid event namespace collisions, all events issued in TEMU are prefixed with `temu.` or `temu::` (the latter is the new style). It is recommended that user published events use their own namespace.

The notification ID 0 is a special notification that is used for indicating "no event", this way it is easy to disable event emission and have the no-event base case be processed cheaply with a single compare and no function call needed.

The `temu_notifyFast`-function is an inline function that allows the compiler to get rid of the function call in case the notification id is the null notification.

Current events include (but this is by no means an exhaustive list):

**`temu.cpuErrorMode`**

CPU entered error mode (halted)

**`temu.cpuTrapEntry`**

CPU trap taken

**`temu.cpuTrapExit`**

CPU trap handler returned (for targets supporting this, e.g. `rett` instruction on SPARC)

**`temu.breakpoint`**

Breakpoint hit

**`temu.watchpointRead`**

Watchpoint read access

**`temu.watchpointWrite`**

Watchpoint write access

**`temu.mil1553Send`**

MIL1553 message sent, reported by bus object

**`temu.mil1553Stat`**

MIL1553 status report, reported by bus object

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 17. Processor Emulation

The processor emulation capability in TEMU is based on an instruction level simulation engine powered by LLVM. At present the processor emulation is interpreted, but does reach hundreds of MIPS (Millions of emulated Instructions Per wall-clock Second) on modern hardware.

A processor object is normally controlled by a scheduler. The scheduler runs several processors and makes sure they are synchronized accordingly.

For multi-processor systems, each processor core is temporally decoupled. They synchronize their time at specific quanta boundaries.

## 17.1. Running a Processor

For a simulator it is important to understand the flow and state transitions of a CPU core and when it terminates and the distinction between stepping and running.

### 17.1.1. CPU States

A CPU can be in three different states:

- Nominal
- Idling
- Halted

The nominal state indicates that the CPU is executing instructions.

Idling indicates that the CPU is not executing instructions but is advancing the CPU cycle counter and event queue. Idle mode is exited when IRQs are raised or the CPU is reset. Idle mode normally indicates either an idle loop (unconditional branch to itself) or power-down mode. In both cases, the CPU will simply forward time to the next event (or if no events are pending return from the core).

Halted mode indicates that the CPU is halted as would happen when a critical error is detected, on the SPARC the halted state corresponds to the SPARC error mode. When entering halted state the CPU core will return and the CPU will remain in halted state until it is reset. It is possible to run a halted core to advance time and execute events (e.g. if there are death event handlers or watchdogs that should reset the system).

### 17.1.2. CPU Exits

A CPU can exit (return from its step / run function) due to a number of reasons.

- Normal exit (step or cycle counter reach its target time)
- Transition to halted mode
- Breakpoint / watchpoint hit

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- Early exit (other reason which can be forced by event handlers or others)

### 17.1.3. Stepping

When a CPU is stepping (e.g. calling its step function), it will execute a fixed number of instructions. When a CPU enters idle mode, a step is seen as advancing to the next event. Except for the event advancement in idle mode, a step can be seen as executing a single instruction. Stepping is not normally done in a simulator, but is often done while debugging software. When the core is in error mode, a step will not advance time however.

When a scheduler is stepping, it is not the scheduler itself that is stepping, but *one* of its CPUs. Thus the `step` command takes an optional parameter `cpuidx` which can be set when one do not wish to step the default CPU.

### 17.1.4. Running

When a CPU is running, it is set to run `UINT64_MAX` steps, and a special end-run-event is posted at the target cycle time. When this end-run event is triggered, the core will stop executing after any stacked events have finished executing. Running a CPU is done in cycles (or in seconds, which is converted to an exact number of cycles).

### 17.1.5. Instruction Behavior

The emulator is interpreted (at present), in the current release an instruction is executed in the following order:

1. Fetch and decode instruction (may call fetch memory access handler)
2. Execute instruction semantics (may call memory access handlers, raise traps etc)
3. Increment program, step and cycle counters
4. Execute any pending events



This means that in an I/O model, if the model wants to terminate with an emergency stop, the step, cycle and program counters will not be updated. To leave the core after this, you need to post a stacked event, which will be executed in step 4. In particular you need to be careful with `raiseTrap()` and the `exitEmuCore()` functions defined in the CPU interface. Although, the `raiseTrap()` function will in general adjust the PC, step and cycle counters and also ensure pending events are executed, the exact results of doing this in a memory handler and an event handler does obviously have different behavior.



If a memory event handler calls `enterIdleMode()`, this will be entered after the program, step and cycle counters have been incremented. Thus, if you write to a power-down register, then the CPU will continue at the next instruction when returning from the interrupt handler that wakes the CPU. If the power-down system needs to be triggered at the current PC, then you need to use `exitEmuCore()`.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

## 17.2. Event System

A processor is the primary keeper of time in the emulator. The processor keeps track of the progress of time, by maintaining a cycle counter.

Some device models need to be able to post timed events on the CPUs event queue to simulate items such as DMA and bus message timing.

There is a standard API for event posting on CPU models. Timed events are fired at their expiration time, while stack posted events goes on a special event stack. The event will then be triggered after the current instruction has finished executing.

Events are tracked by an event ID which is associated with a function/object pair. Meaning that each object (e.g. an UART instance may have the same function posted as an event), however a single object should not post the same function multiple times while the event is still in-flight. Re-posting an event while in flight, will result in a the existing event being descheduled automatically and warning printed in the log.

## 17.3. Multi-Core Emulation and Events

Multi-core processors are simulated by adding several processors to a scheduler.

Multi-core processors are temporally decoupled and emulated by scheduling each core for a number of cycles on a single CPU (this window is called a CPU scheduling quanta).

This method guarantees full determinism even when emulating multi-core processors. The quanta length can be configured as low as a single nanosecond for the fastest processor, but this has a significant performance impact. The best value need to be experimentally determined for the relevant application, but something corresponding to 10 kCycles is probably a good start. Note that too long quantum means that Inter-Processor Interrupts (IPIs) and spinlocks may have a long response time.

Also, IPIs are typically raised as soon as the destination CPU is scheduled, this is either at the start of the next quanta (i.e. later in time) in case the destination CPU already being scheduled, or at the start of the current quanta (earlier in time) in case the destination CPU has not yet been scheduled.



Set the time quanta to 10 kCycles initially, this is a good starting point. This is also the default value.

The quanta length is set in whole nanoseconds. The quanta property can be set in the scheduler state object, and it will automatically be converted to cycles based on the individual processor's clock frequency.

Thus it is even possible to provide different CPUs with different clock frequencies.



The fact that processors are temporally decoupled does have impact on low level multi-threaded code, such as spin locks and lock free algorithms, where a CPU-core may have to wait excessively long for a spin lock if the owning CPU finishes its

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

quanta before releasing the lock. However, it also ensures that the emulation is deterministic.



IPIs are delivered at the start of either the current quanta or the next depending on whether the destination CPU has already been scheduled.

One variant for debugging locking issues is to run with a longer quanta at first and when approaching the locking code, reduce the quanta size to home in on the bug.

As the CPUs usually do not agree on time, the quanta length has an impact on the event system. When posting an event, it normally goes to a single CPU. However, in some cases it is needed to have the different cores agree on time. This is typically the case for timers and some other devices.

For these cases, the synchronized events exist. When posting a synchronized event, the sync level parameter determines if the event is enqueued in the CPU event queue, or the domain or scheduler queues.



Synchronized events should always be posted with a firing time in at least the next CPU scheduling quanta. If it isn't the event will be delayed until the next quanta and a warning noted in the log.

## 17.4. CPU Interface

The CPU interface provides a way to run processor cores and to access CPU state such as registers and the program counter.

```
typedef struct temu_CpuIface {
  void          (*reset)(void *Cpu, int ResetType);
  uint64_t      (*run)(void *Cpu, uint64_t Cycles);
  uint64_t      (*step)(void *Cpu, uint64_t Steps);

  void __attribute__((noreturn))
  (*raiseTrap)(void *Obj, int Trap);

  void (*enterIdleMode)(void *Obj);

  void __attribute__((noreturn))
  (*exitEmuCore)(void *Cpu, temu_CpuExitReason Reason);

  uint64_t      (*getFreq)(void *Cpu);
  temu_CpuState (*getState)(void *Cpu);
  void          (*setPc)(void *Cpu,
                         uint64_t Pc);
  uint64_t      (*getPc)(void *Cpu);
  void          (*setGpr)(void *Cpu,
                          int Reg,
              uint64_t Value);
  uint64_t      (*getGpr)(void *Cpu,
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
                        unsigned Reg);
  void          (*setFpr32)(void *Cpu,
                            unsigned Reg,
                uint32_t Value);
  uint32_t      (*getFpr32)(void *Cpu,
                            unsigned Reg);
  void          (*setFpr64)(void *Cpu,
                            unsigned Reg,
                uint64_t Value);
  uint64_t      (*getFpr64)(void *Cpu,
                            unsigned Reg);
  uint64_t      (*getSpr)(void *Cpu,
                          unsigned Reg);
  int           (*getRegId)(void *Cpu,
                            const char *RegName);
  uint32_t      (*assemble)(void *Cpu,
                            const char *AsmStr);
  const char*   (*disassemble)(void *Cpu,
                               uint32_t Instr);
  void          (*enableTraps)(void *Cpu);
  void          (*disableTraps)(void *Cpu);
  void          (*invalidateAtc)(void *Obj,
                                 uint64_t Addr,
                 uint64_t Pages,
                                 uint32_t Flags);
} temu_CpuIface;
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 18. Memory Emulation

Memory emulation in TEMU is very flexible, the memory system uses a memory space object to carry out address decoding. The memory space object enables the arbitrary mapping of objects to different address ranges. The emulator will handle the address decoding, which is done very efficiently through a multi-level page table.

## 18.1. Memory Spaces

TEMU provides dynamic memory mapping. Memory mapping is done using the `MemorySpace` class. A CPU needs one memory space object connected to it. The memory space object does not contain actual memory, but rather it contains a memory map. It is possible to map in objects such as RAM, ROM and device models in a memory space.

The requirement is that the object being mapped implements the `MemAccess` interface. It can optionally implement the Memory interface as well (in which case the mapped object will support block accesses).

The memory space mapping, currently implements a 36 bit physical memory map (which corresponds to the SPARCv8 architecture definition).

Because it would be inefficient to access through this structure and to build up the memory transaction objects for the memory access interface for every memory access (including fetches), the translations are cached in an Address Translation Cache. The ATC maps virtual to host address for RAM and ROM only. Note that there are six ATCs: one each for read, write and execute operations, and in different variants for user and supervisor privileges.

Memory may have attributes set in some cases (such as for example breakpoints, watchpoints and SEU bits). If memory attributes are set on a page, that page cannot be put into the ATC. Therefore, attribute bits should be set only in exceptional cases.

To map an object in memory, there are two alternatives, one is to use the command line interface command `memory-map`. The other is to use the function `temu_memoryMap()`.

## 18.2. Memory Transactions

Memory transactions are passed through the memory space via the `temu_MemAccessIface` interface. The memory hierarchy may need to pass along a lot of data for the simulation, so memory transactions are rather large objects that are passed as pointers.

By passing transaction objects around, users always have access to both virtual and physical addresses.

The memory transaction object contains a `Value` field. The field is 64 bit (8 byte) in size, however the data unit passed in those 64 bits can be either 1, 2, 4 or 8 bytes. The data unit is encoded in the `Size` field's lower two bits using the log-size of data unit size: Transaction target addresses must be aligned.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

*Table 6. Normal Memory Transaction Encoding, Size and Alignment*

| Size (62 MSB) | Size (2 LSB) | Transaction Size | Address Alignment |
|---|---|---|---|
| 0 | 0 | 8 bit / 1 B | 1 B |
| 0 | 1 | 16 bit / 2 B | 2 B |
| 0 | 2 | 32 bit / 4 B | 4 B |
| 0 | 3 | 64 bit / 8 B | 8 B |

### 18.2.1. Large Transactions

Large transactions are special transactions where more than one data unit is transferred. Devices can opt-in to large transactions by setting the `LargeTransactions` in its capabilities.

Since data units are at most 8 byte in size, this means that large transactions primarily handle transfers of data blocks of > 8 bytes.

The purpose of these is to for example emulate DMA traffic, (subject to IOMMU handling), send / transfer list loading, etc.

In addition, large transactions can be used to load memories like RAM and ROM with data from the simulation infrastructure.

In a *normal* transaction, the single data unit is transferred by copying it in the `Value` field. In a *large* transaction, the `Value` field is instead expected to contain a pointer to a buffer with the data being transferred.



The buffer can consist of either 1-, 2-, 4- or 8-byte data units.

The type of the buffer is still encoded in the `Size` field's lower two bits. A transaction is large if the upper 62 bits in the `Size` field is non-zero.

For large transactions, the `Size` field contains both the data unit type, and the number of data units that is pointed at in the `Value` field.

In the following example, the way to create a normal and large transaction is illustrated:

```
void
makeBuffer32(temu_MemTransaction *mt, size_t numWords, uint32_t *data)
{
  if (numWords == 1) {
    // Create a normal transaction
    mt->Value = data[0];
    mt->Size = 2;
  } else {
    // Create a large transaction
    // Value contains a pointer to the data
    mt->Value = (uintptr_t)(void*)data;
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
    // Data unit size is 4 bytes (use value 2 for encoding)
    // Number of words is the number of items going into the upper 62 bit
    mt->Size = (numWords << 2) | 2;
  }
}
```

Devices do not automatically support large transactions. Devices must opt-in to these using memory access capabilities.

In general, there is no need to support large accesses in a normal device, however if a custom memory is implemented (RAM/ROM/flash/etc), the device should opt-in to handle large transactions.



A large transactions are not subject to automatic endianness swap. Such devices should check the endianness bit in `mt.Flags` with `TEMU_MT_BIG_ENDIAN` or `TEMU_MT_LITTLE_ENDIAN` explicitly. This check should only be done for large transactions.

*Table 7. Large Memory Transaction Encoding, Size and Alignment*

| Size (62 MSB) | Size (2 LSB) | Transaction Size | Address Alignment |
|---|---|---|---|
| non-zero | 0 | 8 bit / 1 B | 1 B |
| non-zero | 1 | 16 bit / 2 B | 2 B |
| non-zero | 2 | 32 bit / 4 B | 4 B |
| non-zero | 3 | 64 bit / 8 B | 8 B |

### 18.2.2. Endianness

Endianness support for memory transaction is built by both a device specifying its endianness and the transaction endianness being set in the `Flags` field.

The following two constants can be used to set the `Flags` accordingly:

- `TEMU_MT_BIG_ENDIAN`
- `TEMU_MT_LITTLE_ENDIAN`

Devices can specify device endianness using the `Endianness` field in its capabilities. If no endianness is specified (i.e. the capabilities function in the `temu_MemAccessIface` is not implemented), the system assumes big endian for the device.

If the memory space detects that a transaction is sent to a device of opposite endianness, the memory space will automatically preform byte swapping for *non-large* transactions.

### 18.2.3. Capabilities

Memory transactions are routed by the memory space to the correct device. However, some device models might emulate little endian devices. Other devices may for example only support word

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

accesses.

By implementing the `getCapabilities` function in the memory access interface, a device can signal to the memory emulation system the endianness, or access sizes supported by the device.

## 18.3. Address Translation Cache

In order to get high performance of the emulation for systems with a paged memory management unit (MMU), the emulator caches virtual to physical to host address translations on a per page level. The lookup in the cache is very fast, but includes a two instruction hash followed by a tag check for every memory access (including instruction fetches).

In the case of an Address Translation Cache (ATC) miss, the emulator will call the memory space object's memory access interface which will forward the access to the relevant device model.

Only RAM and ROM is cached in the ATC, and only if the relevant page does not contain any memory attributes (breakpoints, SEU, MEU etc).

It is possible for models or simulators to purge the ATC in a processor if needed. The means to do this is provided in the CPU interface. Example is given below.

```
// Purge 100 pages in the ATC starting with address 0
Device->Cpu.Iface->invalidateAtc(Device->Cpu.Obj, 0, 100, 0);
```

Note that in normal cases, models do not need to purge the ATC and it can safely be ignored, it is mostly needed by MMU models (that cannot be modelled by the user at present).

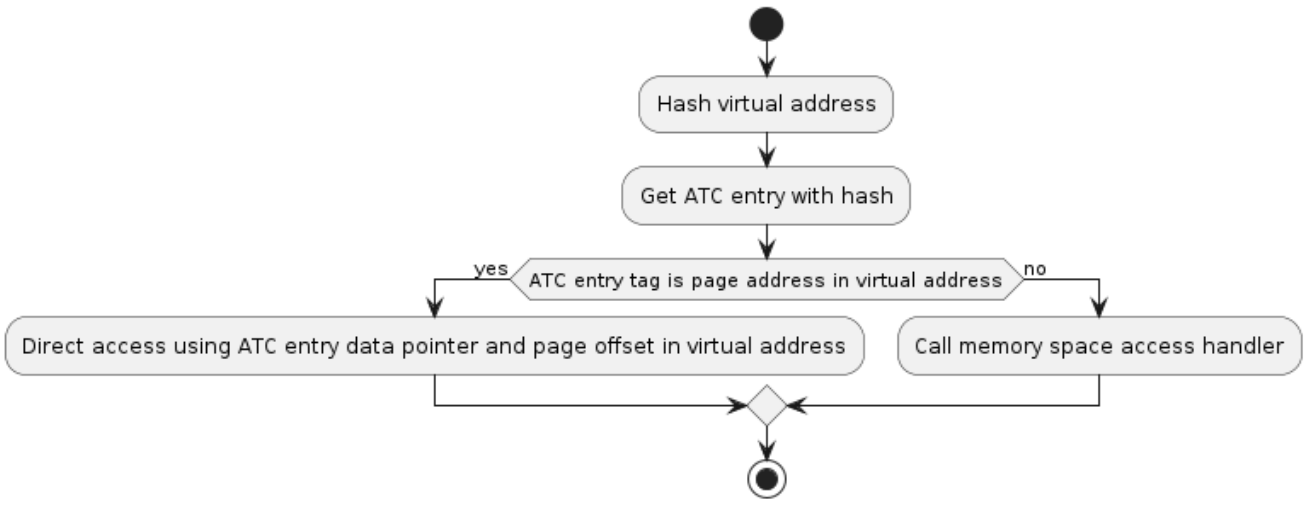



*Figure 2. Memory access flow using the ATC*

## 18.4. Memory Hierarchy and Caches

It is possible to manipulate the memory hierarchy when assembling your machine and connecting the object graph. A cache model can be inserted in the memory space object for more accurate

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

performance modelling. Note that, unless the cache estimates the needed stall cycles on a per page basis, this means that the ATC cannot be used while a cache model is connected. Cache models therefore clears the Page pointer field in the memory transaction object to ensure that the ATC is not used for the memory access.



When the ATC is disabled, the performance of the emulator drops considerably and when a cache model is used that emulates the cache in an accurate manner, it drops even more.

Cache models can be connected to the `preTransaction` and `postTransaction` interfaces. Caches should typically only cache RAM and ROM, at present, the user needs to set the TEMU_MT_CACHEABLE flag when mapping a device which is cacheable. In principle the MMU should handle this, but at present the SR-MMU does not use the cacheable bits.

Cache models and any other models suitable for handling the pre and post transaction semantics should provide a way to chain an additional model after it. This way, multiple levels of caches and tracing modules can be inserted at will.



At present, cacheable objects are only respected as such if they have a size in multiple of page sizes.

To insert a cache model, the typical command sequence is:

```
# Remember to set the cacheable flag on cacheable memories.
memory-map memspace=mem0 addr=0x40000000 length=0x8000000 \
           object=ram0 cacheable=1

# Connect pre- and post- MemTransactionIfaces
connect a=mem0.preTransaction b=l1Cache0:PreAccessIface
connect a=mem0.postTransaction b=l1Cache0:PostAccessIface
```

A pre-transaction handler will intercept memory transactions before they are executed, it can therefore modify written data. A post-transaction handler will intercept memory transactions after they have been executed, the post transaction handler can therefore modify read data.

Currently, the memory system will look at cache timing from the pre-transaction handlers, but the post transaction handler must be connected to ensure that it can clear the Page pointer in the `MemTransaction` object.

### 18.4.1. The Generic Cache Model

The emulator (as of TEMU 2.1) comes bundled with a generic cache model. This model can be used to emulate caches with different number of associativity and line sizes. Most standard cache parameters can be configured in the system. Including the replacement policy (at the moment LRU, LRR and RND are supported), line size, word size, number of sets and number of ways. The generic cache model can also be configured as a split (Harward-architecture) cache, where instructions and data have their own blocks.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

Note that when the cache is not split, the parameters (including the tags etc) will be turned into identical values.

The generic cache implements two copies of the cache interface, one for instructions and one for data. These are effectively identical if the caches are not split, so in that case which interface is not relevant.

### 18.4.2. Tracing Memory Accesses

It is possible to utilize the pre- and post-access handlers in the memory space to trace memory accesses. To do so, implement a model exposing the memory transaction interface. In the `postAccess` handler, the tracing model should clear the page pointer in the transaction object to disable ATC insertion of the memory access. Note that the pre access handler have access to the written value, and the post access handler have access to the read value. While the written value is normally there also in the `postAccess` handler, it will not be there for atomic exchange operations.

## 18.5. Interfaces

### 18.5.1. Memory Access Interface

The memory access interface defines the interface used by objects connected to the emulated memory system. The memory accesses are invoked by a CPU and can be either fetch, read or write operations.

```
typedef struct temu_MemTransaction {
  uint64_t Va; // 64 bit virtual for unified 32/64 bit interface.
  uint64_t Pa; // 64 bit physical address
  uint64_t Value; // Value
  uint64_t Size; // Size (see note above)
  uint64_t Offset; // Offset in bytes from start of mapping (used for determining
register)
  temu_InitiatorType InitiatorType;
  temu_Object_ *Initiator; // Initiating object (normally processor, may be null)
  void *Page; // Page will be cached in the ATC for this memoy page
  uint64_t Cycles; // CPU cycles this memory transaction take
  uint32_t Flags;  // Flags for use in the memory hierarchy.

  void *IR; // Internal pointer
} temu_MemTransaction;

// Exposed to the emulator core by a memory object.
struct temu_MemAccessIface {
  void (*fetch)(void *Obj, temu_MemTransaction *Mt);
  void (*read)(void *Obj, temu_MemTransaction *Mt);
  void (*write)(void *Obj, temu_MemTransaction *Mt);
  void (*exchange)(void *Obj, temu_MemTransaction *Mt); // Optional, deprecated and
not used
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
  void (*mapped)(void *Obj, uint64_t Pa, uint64_t Len); // Optional, called when
interface is mapped into memoryspace
  const temu_MemAccessCapabilities *(*getCapabilities)(void *Obj); // Optional, return
capabilities of device.
};
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 19. Network Simulation

TEMU supports simulated Ethernet links. TEMU provided a number of models for Ethernet communication:

- MAC models
- PHY models
- MDIO / MII bus model (if multiple PHYs are used).
- Ethernet link models
- Bridge models

Normally the `GenericPHY` is all that is needed for modelling a PHY. It is possible to implement additional specific PHYs if needed. The `GenericPHY` supports integration with the MDIO bus and provides for the basic MII register set.

## 19.1. Network Traffic Capture

The `EthernetLink` model supports packet capturing in the PCAP format. The PCAP format can either be captured in a file for offline analysis, or in a named pipe (or FIFO). In the latter case it is possible to use Wireshark for live monitoring of simulated network traffic.

## 19.2. Host Network Integration

To allow external software to connect to target software, TEMU 4.5 and later comes with host network integration support. This currently includes a NAT model that bridges between the real host network and the internal TEMU network.

Other protocols may have been added in newer versions of TEMU, consult the model reference for more details.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 20. Components

While individual objects are indeed very useful and can be instantiated multiple times connected different ways. The are problematic from a number of reasons.

Many systems consist of many different objects (e.g. an ASIC may have several processors, a couple of I/O models and potentially several additional devices).

Although, while the TEMU command line scripts can be used to construct arbitrary object graphs, it is problematic in the command line as one need to ensure the unique naming of different objects. In the case multiple processors need to be instantiated, several otherwise identical objects need to be created with unique names and attached to the correct CPU and / or memory space objects.

To solve this issue, TEMU provides an internal model called "Component". A component is a collection of objects and exported (and potentially renamed) interfaces.

If you are familiar with software components and e.g. the SMP2 simulator standard, this will sound familiar, and indeed. The TEMU components are modelled after such approaches.

While the generic Component class is useful to construct objects with a unique namespace, the real power comes in the component sub-classes. These sub-classes provide custom constructors and destructors that allow them to create a whole system when they are instantiated.

For example, the SPARCv8 target is bundled with the following components:

- erc32-component
- at697f-component
- ut699-component
- ut700-component
- ngmp-component

While these can be constructed manually (there are example scripts for this in the `sysconfig` directory), they are easier to instantiate using the components.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 21. Snapshots

As constructing the object graph can be quite complex, it is useful to do this once using the command line interface. The object graph can then be serialized to a JSON file. This is done using the `snapshot-save` and `snapshot-restore` commands (these have aliases `save` and `restore`).

A snapshot normally consist of the JSON file containing the object graph and property values, and separate binary blobs containing ROM and RAM contents.

The JSON snapshots are human readable, so, simple editing can be done on them by hand using a text editor.

## 21.1. Snapshot Restore Phases

The JSON snapshot mechanism follows a phased approach. This is important in case of implementing custom snapshot restore logic.

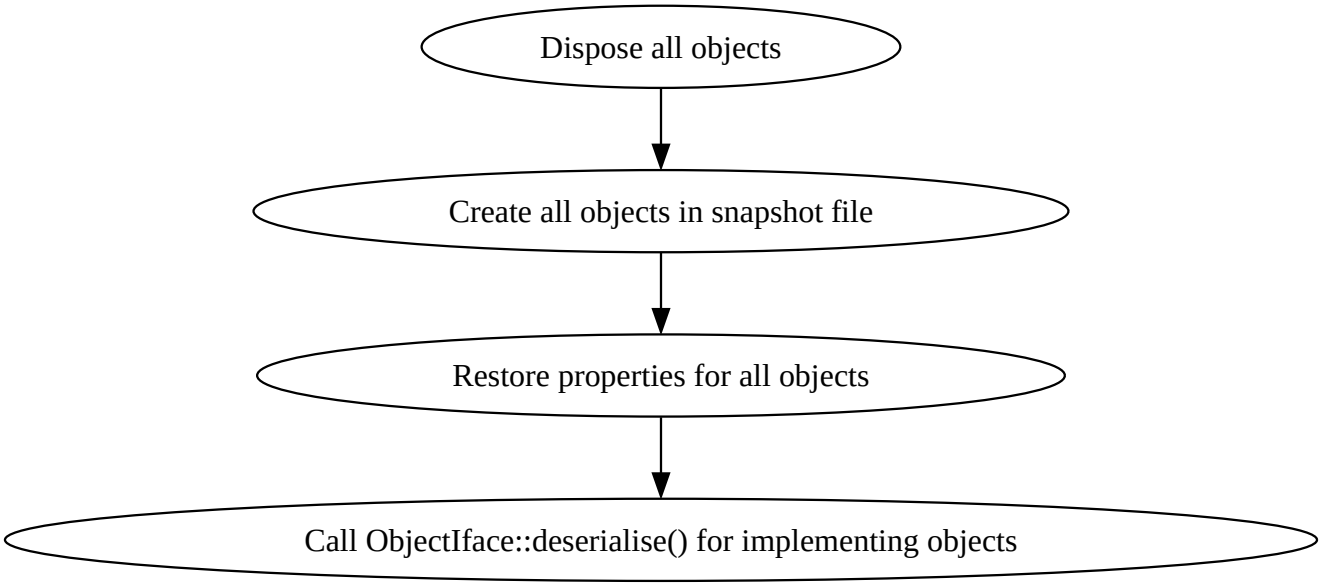



*Figure 3. Snapshot Restore Phases*

In particular this mean that the following holds:

- All non-registered pointers to TEMU objects will become invalid after objects have been recreated.
- Object constructors will be called without arguments. It is up to the model implementer to ensure that restore is successful using properties and custom restore logic.
- Processor and clock custom logic is responsible to restore the event queues. Do not register events outside the object constructor.

## 21.2. JSON Caveats

### 21.2.1. 64-Bit Values

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

JSON does not allow for larger than 53 bit integers to be stored in a portable way (as JavaScript uses doubles for storing integer values). In case a JSON file is edited, pay attention that when data of type `uint64_t` is serialized, it is split into two separate 32 bit values, thus the arrays storing the values will contain twice the elements that are actually in the object's property.

### 21.2.2. ROM and RAM Contents

Another issue is that JSON is not practical for storing RAM and ROM dumps which are needed if saving and restoring a snapshot not at time 0. Thus ROM and RAM is stored in a binary dump (which is host endian dependent) and the JSON file with the saved system configuration contain references to these RAM and ROM dump files.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 22. Software Debugging

There are two supported ways for debugging software with TEMU. Firstly the TEMU CLI supports assembler level debugging in itself. Secondly, TEMU is bundled with a GDB server, this server lets you start up a stand alone program (or run it from the CLI directly).

## 22.1. CLI Based Software Debugging

The TEMU command line interface will when reading ELF files, also load the symbol tables (there is also an API for inspecting ELF symbol tables). Thus, it is possible to disassemble named functions.

When disassembling code based on a function name or a virtual address, the disassembler prints special tokens indicating interesting addresses, such as the PC, nPC (for relevant targets) and trap table pointers. Only one token is printed, and the program counters will always have precedence over other tokens. These tokens are not printed when disassembling using physical addresses.

In addition to disassembling, it is possible to assemble instructions, and modify and inspect both registers and memory.

A global function can be disassembled using the `func=name` parameter, just give the name of the function to disassemble.

```
temu> dis cpu=cpu0 func=main
( pc) 40001934 040001934 9de3bf98 save %sp, 8088, %sp
(npc) 40001938 040001938 f027a044 st %i0, [%fp + 68]
      4000193c 04000193c f227a048 st %i1, [%fp + 72]
      40001940 040001940 c027bff8 st %g0, [%fp + 8184]
      40001944 040001944 82102001 or %g0, 1, %g1
      40001948 040001948 c227bffc st %g1, [%fp + 8188]
      4000194c 04000194c 82102000 or %g0, 0, %g1
      40001950 040001950 b0100001 or %g0, %g1, %i0
      40001954 040001954 81e80000 restore %g0, %g0, %g0
      40001958 040001958 81c3e008 jmpl %o7 + 8, %g0
      4000195c 04000195c 01000000 sethi 0, %g0
```

A local or static function can be disassembled by giving the function name prefixed with the file name and the scope resolution operator.

```
temu> dis cpu=cpu0 func=test.c::bar
      40001924 040001924 9de3bfa0 save %sp, 8096, %sp
      40001928 040001928 81e80000 restore %g0, %g0, %g0
      4000192c 04000192c 81c3e008 jmpl %o7 + 8, %g0
      40001930 040001930 01000000 sethi 0, %g0
```

### 22.1.1. Source Level Debugging

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

TEMU comes with built in source level debugging support, which is based on the DWARF debugging standard. This support is currently experimental and supports source listing, and symbolic and line based breakpoints. Since multiple applications with different DWARF data may be loaded at the same time (e.g. a boot loader and application, or multiple software partitions running under a hypervisor), the DWARF support is based around the notion of debugging contexts, which have been introduced in TEMU 2.2. Currently context management is manual and so is switching the active context. It is at present not possible to inspect symbolic data using the DWARF support.

There are at present three primary areas handled by the commands: context management (i.e. loading, unloading and switching the debugging context), source management (i.e. remapping paths due to moved source directories, and listing source code around the current address), break point handling (i.e. set and control break points). In addition to these, there are some DWARF specific commands that exist to support debugging of the source level debugging code.

The following commands exist at the moment, this explains the purpose of the commands:

**experimental-debug-load-ctxt**

Load an ELF file as a new debugging context.

**experimental-debug-list-contexts**

List all loaded debugging contexts.

**experimental-debug-set-context**

Set the current debugging context.

**experimental-debug-dispose-ctxt**

Remove debugging context.

**experimental-debug-list-source**

List source lines around the given address. Pass CPU argument to use the CPUs program counter. By default 5 lines is listed before and after.

**experimental-debug-add-path**

Add a path for searching for relatively named source files.

**experimental-debug-remap-path**

Add a remapping prefix. This is used to remap absolute paths to different directories. E.g. /home/foo/ to /home/bar/ will remap from one user dir to another one. This is particularly useful if the target software is built on a machine that is not the one you are debugging on.

**experimental-debug-break**

Add a breakpoint. Either at an address using the `addr` parameter or a named location using `loc`. Locations are in the form of `LINENUMBER` for locations in the current file (identified using the CPU argument). `+NUMBER` or `-NUMBER` for relative lines to the current one. `FILE:LINENUMBER` for explicit lines or FUNCNAME for a named function.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**experimental-debug-mute-break**

Prevent break point from printing message on a hit.

**experimental-debug-demute-break**

Ensure a break point prints a message on a hit.

**experimental-debug-ignore-break**

Ignore the break point (i.e. resume after hit, note that the message printing is controlled using the mute / un-mute commands), so it is easy to create a logging break point that does not stop the simulator.

**experimental-debug-stop-break**

Stop after the breakpoint has been hit. This is the default.

**experimental-debug-simulate-break**

Simulates a breakpoint hit at a given address. I.e. trigger the break point handler. This can be used to debug custom break point actions.

**experimental-debug-list-cu**

Print names of all compilation units in the current debugging context along with their starting address and implementation language.

**experimental-debug-print-linenum-prog**

Print the result of the line number program. This is DWARF specific and only useful for debugging line number and break location resolutions.

### 22.1.2. User Application Debugging

There are a number of problems when attempting to debug user applications. These stem from TEMU not being aware of the operating system it is debugging.

One impact of this, as mentioned before, is that the GDB server does not support inspection of OS threads.

Secondly, if an application runs with the MMU enabled, the default where the TEMU GDB server use the current processor's MMU for address translation, may not work.

The MMU issues can be dealt with on some targets by setting the `page-table-root` argument to the `gdb-server` command.

This argument shall be the physical address, of the root page table pointer for the application in question.



Determining the root pointer can be difficult.

On the SPARC, the best way is to try to stop the emulator inside the application, and then inspecting the MMU state.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

The page table root would then be calculated as:

*Listing 10. Calculating the Page Table Root for the Current Application on a SPARC Processor*

```
temu> hex((cpu0.mmuCtxtPtr << 4) + (cpu0.mmuCtxt * 4))
```

The above works for SPARC processors when the MMU is enabled.

If it is not possible stop when the application is running, it may be possible to use the page table walk mechanism to find the relevant pointer.

*Listing 11. Experimentally Determining the Page Table Root*

```
# Determine the current context
temu> cpu0.mmuCtxt
5

# Start with:
temu> cpu0.mmuCtxt = 0
temu> walk cpu=cpu0 addr=0x10000000
# If the walk fails pick the next mmuCtxt

temu> cpu0.mmuCtxt = 1
temu> walk cpu=cpu0 addr=0x10000000

# If the walk now passed we can calculate the page table root as:
temu> hex((cpu0.mmuCtxtPtr << 4) + (cpu0.mmuCtxt * 4))
'0x40001004'

# Restore the original mmuCtxt
temu> cpu0.mmuCtxt = 5

# Launch the GDB server using the computed page table root
temu> gdb-server cpu=cpu0 page-table-root=0x40001004
```



The above methods are all difficult to use in case the OS uses overlapping virtual address ranges for different applications. Contact your OS vendor for information on how to get the page table root if it is not practical using the methods above.

## 22.2. GDB Server

The GDB server is available the plugin: `GdbDebugger`. To instantiate the GdbServer execute the following commands:

```
import GdbDebugger
GdbServer.new name=gdb port=6666 mem=mem0
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
connect-timesource obj=gdb ts=cpu0
```

The new command also allows for two additional arguments:

cpu
: Processor model to limit breakpoints to this processor only.

debug
: Set this to 1 to enable logging of GDB RSP protocol messages.

GDB is connected in the same way as detailed in [target-debugging:::_legacy_gdb_server]. When GDB is disconnected, the TEMU scheduler is automatically resumed.

### 22.2.1. Limitations

#### Virtual to Physical Address Translation

The GDB server cannot set breakpoints before a virtual to physical address mapping has materialized.

If two applications share the same virtual addresses, but different physical addresses. Then when setting a breakpoint, you will automatically set it on all matching addresses. This can be confusing as GDB has no way of determining which application is running. To limit this problem, it is possible to limit breakpoints to specific processors.

The limitations are due to GDB not being designed for systems level debugging, but focused on debugging single applications.

#### Re-Starting of GDB Server

In addition, the GDB server terminates after a disconnect is done. To start it again two options exist:

*Listing 12. Start Server Thread Again*

```
gdb.start-server
```

*Listing 13. Delete and Create New Server*

```
gdb.delete

GdbServer.new name=gdb port=6666 mem=mem0
connect-timesource obj=gdb ts=cpu0
```

#### Concurrent Usage

As the GDB server may be launched for different targets by either multiple users on the same system, or the same user, port assignment in a TEMU script may fail.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

This could be an issue in the case a TEMU script is reused by multiple users.

Since TEMU scripts with failing commands either stop TEMU in batch mode, or return an error through the API.

End user applications must be prepared to deal with this.

Different options exists such as failing gracefully, or leaving it to the end user to start the server with a port number of choice.

*Listing 14. Graceful Failure in TScript*

```
if try GdbServer.new name=gdb mem=mem0 port=6666 {
  try connect-timesource obj=gdb ts=cpu0
  try gdb.start-server
}
```

*Listing 15. Graceful Failure in C*

```
if (temu_execCommandFile("gdb-script.temu")) {
  temu_printerr("Failed to create and start GDB server");
  gdbServerIsRunning = false;
}
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 23. OS Awareness

## 23.1. OS Awareness

Software-aware debug contexts are created with `ContextManager.add-software-context`.

```
ContextManager.new name=cm0
cm0.attach board=board0
cm0.add-software-context board=board0 name=boot config=./os-awareness/linux.yaml
```

The `config` argument points at a YAML file describing the OS-awareness plugin to use and the debug artifacts associated with that software context.

### 23.1.1. YAML Structure

The YAML document is a mapping with the following top-level keys:

`version`
: Optional schema version. Current examples use `1`.

`plugin`
: Required root plugin name such as `bare-metal`, `sparc-rtems`, `riscv-linux`, or `sparc-xng`.

`kind`
: Optional semantic discriminator for the node.

`main`
: Common debug artifact bindings for the current node.

`plugin-config`
: Plugin-specific configuration values.

`images`
: Optional mapping of child image identifiers to nested image nodes.

Relative paths in the YAML file are resolved relative to the YAML file itself, not relative to the current working directory.

### 23.1.2. `main`

The `main` mapping holds common debug artifact references. Supported keys currently include:

- `debug-info`: Main ELF, symbol-map, or other object file used to create the software context.
- `symbols`: Optional alternate symbol file.
- `source-root`: Optional source root used for source lookup/path mapping.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

Additional keys may be added later.

### 23.1.3. `plugin-config`

The `plugin-config` mapping contains plugin-specific values. Examples:

- Linux: `runqueue-symbol`, `hypervisor-root`, `second-level-root`
- XNG: `partition-file`

Key names may use `-` or `_`. The OS-awareness configuration layer treats `runqueue-symbol` and `runqueue_symbol` as equivalent.

### 23.1.4. `images`

`images` is a recursive mapping from a plugin-defined target identifier to a child node with the same structure as the root node.

The meaning of the key depends on the parent plugin:

- For Linux it is typically an absolute executable path in the target file system such as `/sbin/init`.
- For XNG it can be a partition name such as `Partition0`.
- For future hypervisor-style plugins it can be a guest, VM, or partition name.

Each child node may define:

- `kind`: Semantic type such as `executable`, `partition`, or `guest`
- `plugin`: Optional child plugin override
- `main`
- `plugin-config`
- nested `images`

This allows a hierarchical debug description, for example a hypervisor with guest partitions where one partition runs Linux and itself contains executable mappings.

### 23.1.5. Example: Bare Metal

```
version: 1
plugin: bare-metal

main:
  debug-info: ../bare-metal/bare-metal.elf
  source-root: ../bare-metal
```

### 23.1.6. Example: Linux Kernel with Executable Mapping

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
version: 1
plugin: riscv-linux

main:
  debug-info: ../RISC-V/Linux/hello-world/build/build/linux-6.13.12/vmlinux
  source-root: ../RISC-V/Linux/hello-world

plugin-config:
  runqueue-symbol: runqueues

images:
  /sbin/init:
    kind: executable
    main:
      debug-info: ../RISC-V/Linux/hello-world/build/images/fw_payload.elf
      source-root: ../RISC-V/Linux/hello-world
```

### 23.1.7. Example: XNG Hypervisor with Partition Mapping

```
version: 1
plugin: sparc-xng

main:
  debug-info: ../sparcv8-xng-hello-world/hello-world.elf
  source-root: ../sparcv8-xng-hello-world

plugin-config:
  partition-file: ../sparcv8-xng-hello-world/xml/module.xml

images:
  Partition0:
    kind: partition
    main:
      debug-info: ../sparcv8-xng-hello-world/Partition0.elf
      source-root: ../sparcv8-xng-hello-world
```

### 23.1.8. Example: Nested Hierarchy

```
version: 1
plugin: hypervisor

main:
  debug-info: ./hypervisor.elf

images:
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```yaml
  linux-guest:
    kind: guest
    plugin: riscv-linux
    main:
      debug-info: ./linux/vmlinux
    plugin-config:
      runqueue-symbol: runqueues
    images:
      /sbin/init:
        kind: executable
        main:
          debug-info: ./linux/init.elf
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 24. Profiling and Coverage



Profile and coverage collection support is currently experimental.

TEMU supports the collection of execution profiles. These profiles can also be used for coverage analysis.

At present the following metrics are collected:

- Taken branch counters
- Not-taken branches counters

As profiling has a negative performance impact, it needs to be enabled.

## 24.1. Enabling Profiling Mode

To enable profiling in the command line invoke the `enableProfiling` command method.



The memSpace property should be set to the CPUs memory space object. The memSpace property was introduced as an optional property in TEMU 2.2.6 in order to support forwards compatibility.

```
cpu0.memSpace=mem0
cpu0.enableProfiling
```

To enable profiling in the API call the `enableProfiling()` method in the `CpuIface`

```
cpuIfaceRef.Iface->enableProfiling(cpuIfaceRef.Obj);
```

## 24.2. Exporting Profiles

Profiles can be exported in YAML format with the `cov-write` command. Before `cov-write`, the profile state should be flushed.

```
cpu0.flushProfile
cov-write file='-'
```



`cov-write` can use the `-` name in order to specify `stdout`.

The exported format currently looks like the following:

```
---
branch-arcs:
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
  - {src: 0x4, tgt: 0x204, count: 1}
```

`src`

Marks out the physical address of the branch instruction.

`tgt`

Marks the physical address of the target of the branch.

`count`

Marks how many time that branch arc has been executed.

From the branch arcs (in combination with the binary), it is possible to derive information such as instruction coverage and branch coverage.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 25. Emulation Performance

TEMU is at its core an instruction set simulator. The instruction set simulator is optimized in the following ways:

- High performance interpreter with a pre-decode direct dispatch algorithm.
- Binary translator for hot code segments
- Idle and power-down mode detection
- Parallelization of multi-core processors

Although performance is both host and target software dependent, in general the raw rating for the different instruction set simulation modes are:

**Interpretation**

Should run at around 150-200 MIPS.

**Binary Translation**

Should run at around 300-500 MIPS.

**Idle**

Runs at infinite speed, but simulation events will throttle the speed.

## 25.1. Performance Impact of Target Software

Target software has a major impact on performance. The following cases are known to have a negative impact on emulation performance:

- Excessive I/O counts
  - Including register polling loops
- Round-tripping between binary translated code and interpreted code (traps, privilege mode changing instructions, floating point instructions on some targets).
- Self-modifying code
  - True self-modifying code (where code is actually written)
  - False self-modifying code (where a page containing code is written, but the write goes to a location on the page without code).
- MMU invalidations (as these flush the emulator ATCs, which triggers MMU checks). This can be especially bad when the target software is thrashing the MMU pages.
- Undetectable idle mode.
- Non-parallel idle mode, when running TEMU with multiple threads.

Many of the cases above, can be improved with better target software, in principle, better performing software (on the hardware), will perform better in the simulator.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

In other cases, there may be options, further detailed later in this section.

With respect to non-parallel idle mode, this applies to the simulation of multi-core processors. As the cores synchronize occasionally (configurable via the `Scheduler` quanta), if some cores are idling while another one is running, the threads running the idle cores will wait in a barrier for the core running code.

That means that the performance is bounded by the processor running code, and idle mode has little overall impact.

A solution in the case of real-time software is to plan the processor time partitioning for the software, so idle is triggered in parallel on all cores at the same time if possible.

Another potential issue is performance demanding parts of the target software. In some cases it is possible to replace these with simulation-specific variants.

*Example 1. Spacecraft Star Tracker*

In a spacecraft control software, a function or partition could be running image processing from star-tracker cameras.

The image processing function primarily produces a rotational vector or quaternion. And since the image processing may be slow, it makes sense to substitute it.

Such substitution could be done using the custom code pattern detector functionality discussed below, or by adding custom device models which would read the rotational parameters from the device model attached to the environmental models.

## 25.2. Performance Impact from Host

The host machine can impact performance in serious ways. Most noticeable is the impact from the lack of physical processors. To exploit parallelism, TEMU assigns different processor cores to different threads.

If there are not enough processors to run all the threads in parallel, execution of simulated processor cores will become serialized.

This can be especially noticeable in virtualized or containerized environments. Even if a virtual machine has 4 virtual processors, those processors are virtual, and may or may not run on separate physical processors.

It is always best to run TEMU on a physical machine, in the case of virtualized environments it may be needed to configure virtual machines and containers to ensure they are assigned physical processor cores.

Note that it is possible to explicitly set the *thread affinity* in order to control thread to host core allocation.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

## 25.3. Performance Monitoring Capabilities

TEMU supports several ways of measuring performance, many of these mechanisms are opt-in and must be explicitly enabled.

Performance is typically monitored using the built in `ExecutionProfiler` model. A separate execution profiler need to be attached to each processor.

Since performance measurements impact performance itself, the purpose of these is to pinpoint systems behavior, not give a realistic picture of a final deployed system.

```
# Create a profiler object, it will trigger a sampling event at 1Hz.
ExecutionProfiler.new name=prof0 freq=1.0
connect-timesource obj=prof0 ts=cpu0

# Enable the profiler to get statistics from the memory space (e.g. io-perf etc)
connect a=prof0.memorySpace b=mem0:MemorySpaceIface

# Enable low level performance collection on MMIO accesses
# This includes lock-time, and total I/O execution time.
# Statistics is global and not per CPU.
mem0.enable-io-perf

# Enable CPU execution time statistics
cpu0.config.measureExecTime = 1

# Enable scheduler performance metrics,
# the scheduler will measure total execution time and
# barrier wait and synchronization time.
# The barrier statistics only works with the spin-barriers,
# not the std-barriers.
sched.measureTime = 1

# Print out performance statistics when the sampling event is triggered
prof0.config.printExecTime = 1

# The profiler will collect statistics from the scheduler,
# this tells this profiler to get the statistics from a specific thread.
prof0.config.threadNumber = 0

# Open a CSV for writing
prof0.open-csv file="prof0.csv"

# Run scheduler for 10 seconds,
# this should result in 10 sampler events being printed.
# In addition, aggregated performance (which is non-intrusive)
# will be printed when done.
sched.run secs=10.0 perf=1
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
prof0.close-csv
```

The CSV-files are comma separated.

In addition, several models make it possible quickly view statistics using the command line interface, the convention is that the model implements a command named `pstat`.

This is implemented by for example the memory space and several processor models.

```
# Print memory space statistics
temu> mem0.pstat

Code Writes: 100
Self Modifying Code Writes: 100
I/O Reads: 1000
I/O Writes: 1000
Accumulated I/O Read Time: 100000
Accumulated I/O Write Time: 100000
Locks Taken: 2000
Accumulated Locking Time: 20000
```

## 25.4. Performance Improvement Capabilities

### 25.4.1. Infinite Bus Speed

Many buses offer the option to run the simulation in infinite speed mode. This is typically controlled by the bus controller configuration.

E.g. instead of simulating the time it takes to transmit a byte on the UART, it is possible to emit the byte immediately, when the data register is written.

This saves time for both the target software which might be waiting for the UART to finish transferring the data; and in the emulator, which can omit event posting (or at least fall back on stacked / immediate events, which are faster to post).

### 25.4.2. Quanta Size

The quanta size (number of instructions executed between time synchronization), is impacting the performance, especially when TEMU runs in multi-threaded mode.

Higher quanta sizes are preferred, however one should at minimum make sure that the quanta size is compatible with TEMU in single threaded mode, which introduce worst case temporal decoupling.

With higher quanta lengths, different processors run more instructions, before stopping, waiting

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

for other processors to catch up. This in turn results in less time spent in processor switching and synchronization which is overhead.

### 25.4.3. Instruction Cycle Adjustments

TEMU 4 assumes that one instruction takes one cycle. It is possible to tweak the instruction throughput, by setting either the *Cycles per Instruction* (CPI) or *Instructions per Cycle* (IPC).

To set the CPI, the following can be run in a TEMU script:

```
cpu0.cpi = 1.5
```

By setting the CPI to 1.5, performance should increase roughly 30-35%, since the instruction throughput has been reduced by a factor *1.0/1.5* (or around 67%).

The IPC parameter is the inverse of CPI, it is primarily used when one wish to model the effects of super scalar processors.

```
cpu0.ipc = 2 # Same as setting cpi = 0.5
```

Note that in the case of, real-time software, it is not certain that deadlines are kept when modifying these parameters. Sufficient experimental validation will be needed to ensure that the target software can keep on running.

### 25.4.4. Custom Idle Detector

As of TEMU 4.2, the previous internal idle pattern detector API, has been exposed as an interface to the processor models.

Internally, TEMU specify several idle instructions as built-in patterns. These idle patterns are per target and described in the respective target guides. It is possible to add additional patterns.

For example, the SPARCv8 two instruction `ba 0; nop` pattern is defined as follows:

```
// Match the two instructions in sequence,
// we match the whole instructions using 0xffffffff as masks
const temu_CodePatternEntry ba0nop[] = {
    {0x10800000, 0xffffffff}, // ba 0
    {0x01000000, 0xffffffff}  // nop or sethi 0, %g0
};

temu_CodePattern ba0nopPattern = {
    .PhysicalAddress = 0,
    .PhysicalAddressMask = 0, // Physical address is don't care
    .Action = tePA_Idle, // Trigger idle mode on entry
    .Callback = NULL, // Applicable to tePA_Call only
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
    .CallbackData = NULL, // Applicable to tePA_Call only
    .Parameter = 0, // 0 makes this an untagged idle
    .PatternLength = 2, // Number of instructions in pattern
    .Pattern = ba0nop // Instruction patterns
};

CodePatternIface->installPattern(cpu, &ba0nopPattern);
```

### 25.4.5. Tagged Idle Mode

Tagged idle mode works like idle as discussed above. The difference is that, in tagged idle the instructions will be executed once.

Whether to execute the instructions is controlled by the processor's `skipIdleTags` property.

The tag used for a particular idle mode pattern is controlled by the integer `.Parameter` field in the pattern.

If the pattern is non-zero, the tag bit will be checked before entering idle. If the processor property has the relevant bit set, idle will not be entered, but the bit will be cleared.

The bits are set by calling the `wakeUp` function in `temu_CpuIface`.

This means that we can specify a specific instruction as tagged idle, trigger on it, and then have it `wakeUp` when a model condition is reached.

There are two use cases for this, one is to force the processor to idle mode, when it is reading certain I/O registers. When the I/O registers change, the model can wake-up the processor, which forces it to run the I/O operation again. Although a model, could trigger idle at any I/O, the pattern mechanism makes it possible to single out specific instruction sequences.

The second use case is to optimize cross processor polling loops. This is a bit more complicated, but a program polling a memory location. Can trigger tagged idle mode in the same way as when polling an I/O register. The wake-up logic, will need to be triggered based on other conditions.

```
temu_CodePattern idleAtAddressPattern = {
    .PhysicalAddress = 0x40000000,
    .PhysicalAddressMask = 0xffffffff, // Trigger when running address above
    .Action = tePA_Idle, // Trigger idle mode on entry
    .Callback = NULL, // Applicable to tePA_Call only
    .CallbackData = NULL, // Applicable to tePA_Call only
    .Parameter = 1, // Non-zero makes this a tagged idle, tag is 1
    .PatternLength = 0,
    .Pattern = NULL
};

CodePatternIface->installPattern(cpu, &idleAtAddressPattern);
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
// From model when register state changes so it should wake up the software
cpuIf.Iface->wakeUp(cpuIf.Obj);
```

### 25.4.6. Skip Patterns

Skip patterns is a simple way to specify an instruction pattern that will be skipped. This can for example be the execution of scrubber logic, memory tests, etc.

The skip patterns are defined by setting the `.Action` field in the pattern, to `tePA_Skip`.

Note that the skipped instructions are assumed to take no time.

To some extent, the skip pattern is a bit like a branch. It would in some cases be possible to patch in a branch instruction to accomplish the same behavior. The patterns let you define a masked address, this in turn means that the scrubber or memory test can move due to target software changes.

```
temu_CodePattern skipPattern = {
    .PhysicalAddress = 0x40000000,
    .PhysicalAddressMask = 0xffffffff, // Trigger when running address above
    .Action = tePA_Skip, // Trigger skip instructions when running
    .Callback = NULL,
    .CallbackData = NULL,
    .Parameter = 10, // Skip 10 instructions, including the first
    .PatternLength = 0,
    .Pattern = NULL
};

CodePatternIface->installPattern(cpu, &skipPattern);
```

### 25.4.7. Arbitrary Call Operations

The general code pattern case is the `tePA_Call` action. The call action does not normally enter idle mode or skip instructions.

It is however possible to possible to:

- Change the program counter (will skip instructions)
- Change the processor state (e.g. enter idle mode)

That makes it possible to accomplish the same behavior as both the skip and tagged idle operations.

However the callback must take care of the state updates, making it a bit more difficult to implement.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

A use case is to for example install custom function replacements.

For example, a call to the `puts` function in the target software, could be eliminated. The pattern triggered call operation could read out string directly, and send it to the simulator log bypassing both I/Os and system call invocations.

Similarly, a call to an image processing kernel, could be eliminated, further enabling the call handler to update registers as expected.

E.g. if the kernel computed a quaternion for a star-tracker, the result could be produced by the call operation, by accessing the rotation directly from the simulator models.



Correctly using the `tePA_Call` operation to skip function and system calls is a per system / mission customization. To implement it correctly, good knowledge of the target's ABI will be needed.

To handle specific target function calls using this mechanism, the pattern should be defined so it is installed on the call instruction. To handle all system calls / function calls, there are two ways:

- In the case of system calls, intercept either all the relevant trapping instructions in the user software, or the system call handler in kernel space.
- In the case of function calls, intercept the first instruction in the called function.

The reason that the calling instruction cannot be intercepted, is that call and branch operation in most cases are relative. The pattern API can at present not match on complex conditions such as CPU state.

The example below shows how to specify a custom call handler. The handler is triggered at physical address 0x40000000. It will read out CPU state and modify it before returning.

```
void
myCallFunc(void *obj, void *data)
{
  temu_Object *cpu = (temu_Object*)obj;
  MyData *myData = (MyData*)data;

  uint32_t param0 = temu_cpuGetReg(cpu, 8); // Get parameter 0 (%o0)
  temu_logInfo(cpu, "Calling function with parameter %" PRIx32, param0);
  // Do something with myData

  // ...

  // Skip instruction and its delay slot
  temu_cpuSetPc(cpu, temu_cpuGetPc(cpu) + 8);
  temu_cpuSetReg(cpu, 8, 1); // Set return value (%o0) of function to 1
}
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
MyData myCallbackData;
// ...

temu_CodePattern callPattern = {
    .PhysicalAddress = 0x40000000,
    .PhysicalAddressMask = 0xffffffff, // Trigger when running address above
    .Action = tePA_Call, // Call function
    .Callback = myCallFunc,
    .CallbackData = myCallbackData,
    .Parameter = 0, // Not used
    .PatternLength = 0,
    .Pattern = NULL
};

CodePatternIface->installPattern(cpu, &callPattern);
```

### 25.4.8. Barrier Strategy

When running in parallel, the simulated time is kept in sync by running each processor core for a fixed duration of simulated time called the quanta.

The threads finishing their work will wait for the other threads in a barrier.

At the time of writing there are two barrier implementations, with different performance characteristics:

- Spin Barrier (the default)
- Std Barrier with backoff support (where threads *yield* or *sleep* if they wait too long).

The best strategy depends on the target software and needs to be experimentally determined.

*Listing 16. Barrier Control*

```
# Enable the default spin barrier.
sched.enable-spin-barrier

# Enable the standard barrier with backoff support.
sched.enable-std-barrier
```

### 25.4.9. Thread Affinity

It is possible to control the affinity and bind TEMU scheduler threads to specific host processor cores.

The configuration of affinity prevents thread migration, and gives more predictable performance.

This must be done before the number of threads is configured.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

*Listing 17. Affinity Example*

```
# Set affinity for the 2 threads
sched.set-affinity thread-id=0 host-core=4
sched.set-affinity thread-id=1 host-core=5

sched.set-threads threads=2
```

In the example above the we assign thread 0 to host CPU 4, and thread 1 to host CPU 5.

The host CPU number is the same as the one listed in `/proc/cpuinfo` on Linux.



Host cores should not select SMT (or hyperthreaded) cores on the same physical core.

There is no correct affinity setting for all host processors, and it should be experimentally determined on the deployed system.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 26. Fault Injection

When testing software, it is often necessary to test error handling in the software. Different devices support different types of error injection capabilities.

Typically, fault injection will need to be done using a TEMU plugin. In principle, the

This section discuss some approaches and pitfalls with fault injection in TEMU.

## 26.1. Ensuring Determinism

When implementing a custom fault injection module, pay attention to:

- Ensuring determinism if using random number generation. Random number generator state (seeds, etc) must be possible to snapshot.
- Implement a fault recorder in the injector that can replay any external fault injections.

## 26.2. Device Faults

Some devices report errors using registers and interrupts. It is possible to modify the register contents, and then raising interrupts using either the command line or the API.



It is often necessary to ensure that multiple registers and interrupts are consistent. This can be challenging, in itself.

### 26.2.1. Modifying Registers

Injecting faults by manipulating registers is possible, either manually using TEMU commands or using the API.

The main mechanism is that one issues a *set* operation to a property. Do not execute a *write* operation, since these may trigger register semantic effects.

### 26.2.2. Raising Interrupts

Interrupts can be raised using interrupt controller's commands. Another way is to connect a custom fault injection device to the interrupt controllers interrupt interface.

## 26.3. Memory Faults

### 26.3.1. Modifying Memory

Modification (corrupting memory contents) can be done by simply writing to the memory.

Note that these writes do not inject ECC errors per see, but simply modifies the content.

### 26.3.2. Correctable Memory Errors

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

A correctable memory error is set in the memory space subsystem, using the *upset* memory attribute.

Some memory controller models support the handling of these attributes natively.

If the memory controller lacks that support, it is possible to attach a custom handler to the memory space's `upsetHandlers` interface reference.

### 26.3.3. Uncorrectable Memory Errors

Uncorrectable memory errors can be injected using the *faulty* memory attribute. They are seen as uncorrectable in the sense of ECC, that is they get corrected by writing to the location.

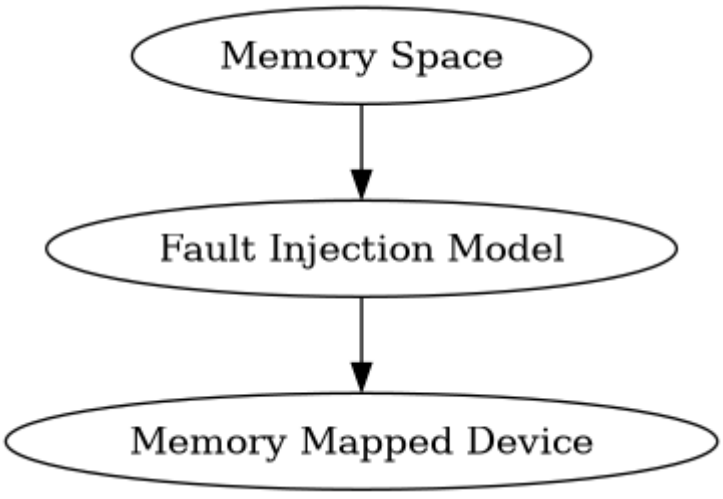
It is possible to add permanently uncorrectable errors by connecting a custom memory access interface to the memory space `faultyHandlers`.

Some memory controller models support the handling of these attributes natively. In some cases, the handling of uncorrectable errors have to be done manually. This is done in the same way as forcing stuck bits by connecting a custom interface to `faultyHandlers`.

### 26.3.4. Persistent Memory Errors

Stuck bits can be injected using the `preTransaction` and `postTransaction` properties in the memory space.

Another way is to add an intermediate device between a failed device, RAM or ROM and the memory space. This approach is more efficient than using the `preTransaction/postTransaction` handlers. The reason is that it does not add overhead to other devices.




For more complex use cases, nested memory spaces can be used.

## 26.4. Network Interception

Network traffic is a special topic. In principle bus models do not model hardware level integrity checks. For example, parity bits are not modelled in the UART or 1553 models. Only software visible

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

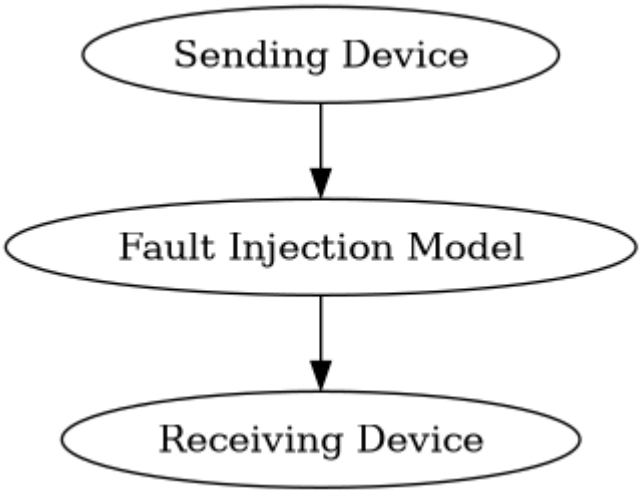
effects are modelled. Several bus models thus have a way of signalling bus model errors such as parity or CRC failures using error bits in frame and packet structures.

The way these features are implemented is specific to the bus model.

Network traffic (including buses such as Ethernet, 1553, SpaceWire and serial), can be handled by tapping the network traffic.

### 26.4.1. Point to Point Buses

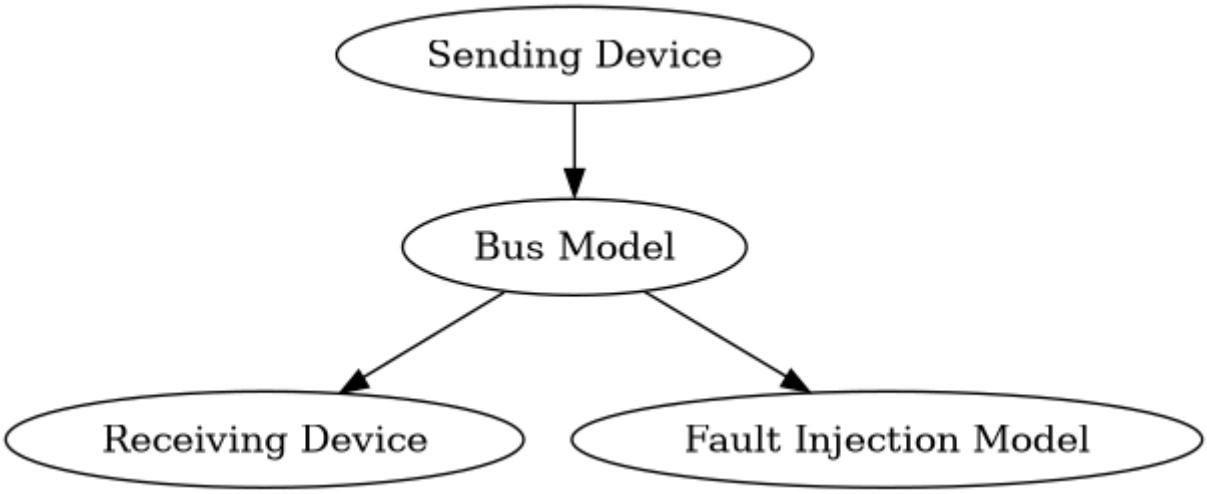
For point to point buses such as serial and SpaceWire, a tap device must be implemented.




This device will normally pass through interface invocations in two directions. To drop a bus message, one simply ignore transferring it to the destination. To inject extra messages, one can schedule events that sends pseudo random data.

### 26.4.2. Multi-Point Buses

For multi-point buses such as CAN, Ethernet and 1553, error injection on the frame or packet level can be done using the TEMU notification mechanism.




The notification will typically receive a pointer to the transmitted packet. The send notifications are

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

emitted before the frame is delivered, thus allowing the modification of the traffic in-flight.

Some bus interfaces allow for symbolic injection of certain bus specific errors, e.g. CRC errors, parity errors etc by setting an error bit in the packet structure.

In order to drop packets on multi-point bus, a tap device should be placed between the bus model and the receiving or sending device.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 27. Scripting

The command line interface provides a simple way to script the emulator, however it does not provide control flow and other more complex features. The command line interface therefore supports scripting with Python.

The scripting support is based on wrapping the C-API of the emulator using the `ctypes` Python package. Therefore it is possible to pass in Python functions where the API expects C-function pointers, for more details of how to do this, please consult the `ctypes` documentation on https://www.python.org::

The wrappers are installed in: `share/temu/wrappers/Python/`. The location is automatically detected by TEMU, so it is possible to use the wrappers by simply importing the packages from your python script.



Scripting wrappers typically strip the `temu` namespace from function names as the languages have their own support for namespaces or packages. Instead, they bundle the TEMU functions inside the TEMU package. To use the functions the relevant package must be imported using e.g. `import temu.c.support.cpu`. The C-subpackage is used for auto-generated wrappers of the C-API.

Python scripts can be executed via the `script-run` command from the CLI. The command takes either a file using the `file` argument, or a literal string using the `script` argument.

Another way to run a Python script is to start the CLI with the `--run-script` option, using this option a non-interactive execution of TEMU will be started (which stops after the script finishes). It is possible to run multiple script by specifying the `--run-script` option multiple times. As argument, pass a name of the script you want to execute.



Use the `--run-commands` option to specify a CLI script as well. The CLI scripts are normally more convenient for constructing CPUs and will be possible to use when running interactively as well.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 28. External Analysis Tools

TEMU can be run under external memory analysis tools such as Valgrind and sanitizer-enabled builds. There are two tool-specific interactions that are worth knowing about when interpreting reports.

## 28.1. Valgrind and Binary Translation

TEMU's binary translator emits host code at run time. On x86 hosts this generated code may use the host flag save and restore instructions `PUSHFW/POPFW`. Before Valgrind 3.3.1, the instructions in question were not properly supported.

If Valgrind fails while executing translated code, use a newer Valgrind build. If upgrading Valgrind is not possible, disable binary translation while running under Valgrind.

```
# Disable binary translation on a specific SPARC processor.
cpu0.disableTranslation

# Disable binary translation on a specific RISC-V processor.
cpu0.disable-dbt
```

Disabling binary translation reduces performance, but keeps execution on the interpreter path that is easier for Valgrind to instrument.

## 28.2. Process-Lifetime Allocations

Some TEMU support systems intentionally keep small global data structures alive until process exit. For example, the notification system keeps its global notification tables and synchronization objects allocated for the lifetime of the process. This avoids static destruction order problems during normal shutdown, where objects may still emit disposal notifications while other global objects are being torn down.

If integrating TEMU with LeakSanitizer compiled simulators or tools, then the notification system may trigger detection of leaks.

These can typically be suppressed as follows:

```
LSAN_OPTIONS=suppressions=/path/to/temu-lsan.supp ./my-simulator
```

Where the suppression file contains the following:

```
leak:notificationNameMap
leak:notificationMap
leak:libTEMUSupport
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

Similarly, when using Valgrind, these allocations may be seen as `still reachable`, especially when run with `--show-leak-kinds=all`. Such entries are expected and should not be treated as leaks, and may need suppression.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 29. Examples

## 29.1. Quick CPU Construction Using JSON Files

It is possible to quickly instantiate a system configuration including CPUs, memory and peripherals, this can be done by loading a JSON file with the serialized state of an existing system.

The JSON files are easy to understand, and can be edited by hand if needed (e.g. to change memory sizes).

Several examples of already defined JSON files are available in: `/opt/temu/5.0share/temu/sysconfig/`.

### 29.1.1. CLI

To load a system configuration in the current directory from the CLI:

```
snapshot-restore Leon2.json
```

### 29.1.2. API

To restore a JSON file from the API, call the `temu_deserialiseJSON` function with the file name as argument. The function returns non-zero on failure.

```
temu_deserialiseJSON("Leon2.json");
```

## 29.2. Command Line CPU Construction

Command line script for constructing a LEON2 CPU with on-chip devices. Note that constructing your own machine configuration from scratch is not trivial. Several CLI scripts are provided with the emulator and installed in `/opt/temu/5.0/share/temu/sysconfig/`.

```
import Leon2
import Leon2SoC
import Memory
import Console

Leon2.new name=cpu0
Leon2SoC.new name=leon2soc0
MemorySpace.new name=mem0
Rom.new name=rom0
Ram.new name=ram0

# Console is a virtual serial port sink that prints output to STDOUT
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
Console.new name=tty0

object-prop-write prop=rom0.size val=8192
object-prop-write prop=ram0.size val=8192

# Map in RAM and SOCs at the relvant address
memory-map memspace=mem0 addr=0x00000000 length=0x80000 object=rom0
memory-map memspace=mem0 addr=0x40000000 length=0x80000 object=ram0
memory-map memspace=mem0 addr=0x80000000 length=0x100 object=leon2soc0

connect a=cpu0.memAccess b=mem0:MemAccessIface
connect a=cpu0.memory b=mem0:MemoryIface
connect a=mem0.invalidaccess b=cpu0:InvalidMemAccessIface

# We only use the Leon2 SoC for the IRQ controller interface
# the interface is required by the CPU

connect a=leon2soc0.irqControl b=cpu0:IrqIface
connect a=cpu0.irqClient b=leon2soc0:IrqClientIface
connect a=leon2soc0.queue b=cpu0:EventIface
connect a=cpu0.devices b=leon2soc0:DeviceIface

connect a=tty0.serial b=leon2soc0:UartAIface
connect a=tty0.queue b=cpu0:EventIface

objsys-check-sanity

# Load binary (supports ELF files as well)
load obj=cpu0 file=myobsw.srec
set-reg cpu=cpu0 reg="%fp" value=0x40050000
set-reg cpu=cpu0 reg="%sp" value=0x40050000
run cpu=cpu0 pc=0x40000000 steps=1000000000 perf=1
```

Note that there are several of these configuration files available for different machine configurations.

## 29.3. Programmatic CPU Construction

To construct a CPU using the API, the following code sequence illustrates how. It is straight forward to translate the CLI construction (see previous section) to the C-API if needed.

```
#include "temu-c/Support/Init.h"
#include "temu-c/Support/Objsys.h"
#include "temu-c/Memory/Memory.h"
#include "temu-c/Target/Cpu.h"
#include "temu-c/Support/Loader.h"
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```c
#include <stdio.h>

int
main(int argc, const char *argv[argc])
{
  temu_CreateArg Args = TEMU_NULL_ARG;

  // Init support library, this will check your license
  // the program will terminate if you do not have a valid license
  temu_initSupportLib();

  // Look up the temu command and setup the plugin paths based on
  // its location.
  temu_initPathSupport("temu");

  // Load the plugins needed
  temu_loadPlugin("libTEMULeon2.so");
  temu_loadPlugin("libTEMULeon2SoC.so");
  temu_loadPlugin("libTEMUMemory.so");
  temu_loadPlugin("libTEMUConsole.so");

  // Create needed objects, no arguments are given (see init above)
  void *Cpu = temu_createObject("Leon2", "cpu0", &Args);
  void *L2SoC = temu_createObject("Leon2SoC", "leon2soc0", &Args);
  void *MemSpace = temu_createObject("MemorySpace", "mem0", &Args);
  void *Rom = temu_createObject("Rom", "rom0", &Args);
  void *Ram = temu_createObject("Ram", "ram0", &Args);
  void *Console = temu_createObject("Console", "tty0", &Args);

  // Allocate space for ROM and RAM
  temu_writeValueU64(Rom, "size", 0x80000, 0);
  temu_writeValueU64(Ram, "size", 0x80000, 0);

  // Map in ROM, RAM and the IO modules in the memory space
  temu_mapMemorySpace(MemSpace, 0x00000000, 0x80000, Rom);
  temu_mapMemorySpace(MemSpace, 0x40000000, 0x80000, Ram);
  temu_mapMemorySpace(MemSpace, 0x80000000, 0x100, L2SoC);

  // For the L2 (without MMU) we connect memAccess directly to the
  // memspace, for MMU systems, we will need to connect memAccessL2
  // instead, and to connect memAccess to the CPU memory access
  // interface. See /opt/temu/x/share/temu/sysconfig dir for exmples.
  temu_connect(Cpu, "memAccess", MemSpace, "MemAccessIface");
  temu_connect(Cpu, "memory", MemSpace, "MemoryIface");
  temu_connect(MemSpace, "invalidaccess", Cpu,
               "InvalidMemAccessIface");

  // In TEMU 2.2, we do not need to connect the reverse link
  // this has been automated through the "ports" mechanism.
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
temu_connect(Cpu, "irqClient", L2SoC, "IrqClientIface");

// Attach the L2SoC to its time source.
temu_setTimeSource(L2SoC, Cpu);

// Add Device to CPU device array, this is used to distribute
// CPU resets to device models.
temu_connect(Cpu, "devices", L2SoC, "DeviceIface");

// The console implements the serial interface and simply
// redirects it to stdout. For a GUI console use the ConsoleUI
// class instead
temu_connect(Console, "serial", L2SoC, "UartAIface");
temu_setTimeSource(Console, Cpu);

// Check sanity of the object graph, pass non-zero to enable
// automatic printouts (with info on which objects are not
// sane). 0 means the function is silent, and we only care
// about the result. Note, this is a debugging help, some
// interfaces do not need to be connected (e.g. the LEON2
// cache interfaces).
if (temu_checkSanity(1)) {
  fprintf(stderr, "Sanity check failed\n");
}

// Can pass CPU or MemorySpace, which you pass doesn't matter
// loadImage handles both SREC and ELF files.
temu_loadImage(Cpu, "hello");

// To get the CPU interface to run the CPU directly, we query
// for the interface. The CpuIface implements the basic CPU
// control functionality like RESET
temu_CpuIface *CpuIf = temu_getInterface(Cpu, "CpuIface", 0);

CpuIf->reset(Cpu, 0); // Cold-reset, 1 is a warm reset
CpuIf->setPc(Cpu, 0x40000000);        // Starting location

// Fake low level boot software setting up the stack pointers
CpuIf->setGpr(Cpu, 24+6, 0x40050000); // %i6 or %fp
CpuIf->setGpr(Cpu, 8+6, 0x40050000);  // %o6 or %sp

// You can step or run the CPU. Running runs for N cycles
// while stepping executes the given number of instructions
// as an instruction can take longer than a cycle, these are
// not the same. For multi-core systems, you will not run or
// step the CPU but rather a machine object, which will ensure
// that all of the CPUs advance as requested. Also a CPU in
// idle or powerdown mode does not advance any steps, but only
// cycles.
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
  CpuIf->run(Cpu, 1000000); // Run 1 M cycles
  //  CpuIf->step(Cpu, 1000000); // Step 1 M steps
  //  CpuIf->runUntil(Cpu, 1000000); // Run until absolute time is
                                     // 1000000 cycles

  // Step 10 instructions, but return early if until absolute time
  // reaches 1000000 cycles
  //  CpuIf->stepUntil(Cpu, 10, 1000000);
  return 0;
}
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 30. Modelling Guide

## 30.1. Introduction

This document provides an overview of how to write models in TEMU. Device models in TEMU are plugins, which can be written in C or C++ (using the C API).

To write a model, it is important to understand the TEMU object system. The object system provides a way to register a *classes*, *properties* and *interfaces*; and a way to instantiate classes to objects.

For users of the System-C, SMP2 or other simulation modelling frameworks, you will find that there are likely several similarities in the TEMU object system to these standards and framework. However, the emulator is intended to work with any external modelling standard and with any language, and the emulator therefor provides its own C level API. It is easy to integrate external modelling standards and frameworks using the external class feature in TEMU.

The TEMU object system is driven by the use of interfaces and interface references (see important interfaces for more info).

A very important interface for device models is the `MemAccessIface`. This Interface is the interface between the emulator core and the memory system. The interface provides a standardized way to implement read, write and fetch handlers for different memory mapped devices.

The most important interfaces are listed in Section 30.4.2.

### 30.1.1. Coding Conventions

TEMU provides a set of headers that define the API. These headers are located in the `include/temu-c/` directory that can be found in the installation directory (normally `/opt/temu/5.0/`). The headers provide a C-API for maximum compatibility (headers are C++ compatible). The headers follow a number of coding conventions.

- The namespace for TEMU is `temu`, and public functions, types and macros are prefixed with `temu_` or `TEMU_`.
- Enum members are prefixed with `teXYZ_` where XYZ is a per enum type abbreviation.
- Interface types (structs containing function pointers) are suffixed with Iface. This suffix is assumed by some macros, so custom interfaces should follow the same suffix rule.
- Interface reference types are suffixed with IfaceRef. This is assumed by some macros. There is a macro `TEMU_IFACE_REFERENCE_TYPE` that lets you define such reference types.



While the headers have a specific coding style, the only rule that the user have to consider is that custom interfaces must be suffixed with `Iface` and interface reference types with `IfaceRef`, although the latter is automatic if the macro is used.

## 30.2. Basic

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

### 30.2.1. The TEMU Object System

*Object System Glossary*

**class**

A description of a POD type with properties, interfaces and a name. Realized as a registered struct type.

**property**

A variable in a POD type, associated with an count (e.g. 1 for scalars, > 1 for arrays), an offset, a type and a name. A property can also have a set of accessor functions registered which enables the association of semantics with a property write or read.

**pseudo property**

A virtual variable for a class. A pseudo property is *ONLY* manipulated through accessor functions and does not have an associated storage location.

**interface**

A record of function pointer that is associated with a class. The function pointers take as first argument a pointer to the object.

**object**

An instance of a class.

As mentioned, TEMU device models are written using the object system. This entail a certain amount of boiler plate code that must be written to register a class in the object system. This boiler plate is typically written in a function with the signature `extern "C" void temu_pluginInit(void)` (or using the macro TEMU_PLUGIN_INIT). This function is called when a TEMU device plugin model is loaded by the object system. Normally, each plugin provides only one device class, but there is nothing preventing a plugin from registering multiple device model classes if needed.

Interfaces are one of the key concepts in the TEMU object system. They provide a way to have standardized functionality associated multiple classes. For example, the `IrqCtrlIface` provides a standard way to raise and lower interrupts. An object implementing interrupt control can use this to provide the functionality in a uniform way. Typical uses of that particular interface are processor models and external interrupt controllers.

#### Plugin Mechanism

TEMU is structured around plugins. Plugins are dynamically loaded code libraries that at load time registers any classes that the plugin provides. Plugins can use the TEMU_PLUGIN_INIT macro to define a function that will be called at plugin load time. Loading a plugin can be done using the `temu_loadPlugin()` function or in the command line interface using the `import` command. The `temu_loadPlugin()` function use the normal system paths (i.e. RPATH, LD_LIBRARY_PATH, RUNPATH, and system loader paths), while the command line interface has an additional plugin path variable that can be controlled using the `plugin-append-path`, `plugin-remove-path` and `plugin-show-paths` commands.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

To add the plugin initializer function to your file add the following code to your device model plugin:

*Listing 18. Plugin Initialisation Function*

```
#include "temu-c/Support/Objsys.h"

TEMU_PLUGIN_INIT
{
  temu_Class *Cls = temu_registerClass(...);
  //...
}
```

The plugin does not have to link to the emulator library since the plugin is loaded by the library. The system linker will resolve any references from the plugin to functions in the emulator libraries.

### Classes and Properties

A class description in TEMU is a registration of a POD-type in the object system, where the fields of the type are also registered as properties. An instance of a class is known as an object.

#### Classes and Instantiation

As mentioned, classes are managed by the TEMU object system. The classes are created by registering a name together with a create and dispose function. These functions are used to allocate and delete memory of objects of the relevant class. In the most basic implementation the create function simply returns a block of data allocated with `malloc()` or `new`, and the dispose function pass the object to the counterpart (`free()` and `delete` respectively).



All classes must inherit `temu_Object`. This is accomplished by ensuring that the first field in the struct corresponding to the class is `temu_Object Super;`.

The following snippet shows how to register a new internal class:

*Listing 19. Registering an Internal Class*

```
typedef struct {
  temu_Object Super; // Must be first field in struct
  // ...
} MyDevice;

// Register new internal class
temu_Class*
temu_registerClass(const char *ClsName,
                   temu_ObjectCreateFunc Create,
                   temu_ObjectDisposeFunc Dispose);

// Creating objects of internal classes
void* temu_createObject(const char *ClsName, const char *ObjName,
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
                        const temu_CreateArg *Args);
```

#### Interfacing with External API Models

TEMU3 does not support the registration of external objects and classes. Instead a wrapper class must be created which wraps the external class.

If the creation can be controlled by TEMU, it is convenient to create a TEMU class with a pointer (or a **std::unique_ptr<>** that instantiates the external object). If not, then the TEMU object should simply contain a normal non-managed pointer that can be used to access the external object.

For the cases where TEMU is allowed to manage the external object, it is also possible to use the **IndirectClass<T>** template if the models are implemented in C++.

### Interfaces and Interface References

An interface is a collection of function pointers, normally stored in a struct. These interface types are used to allow for model compatibility with different subsystems. An interface is registered to a class at class creation time. The first parameter of each function in an interface should by convention take an object pointer (i.e. a self or this pointer of type void*), as the object system API is expressed in C, the self/this pointer must be passed explicitly.

Interface references however are properties that refer to an object and interface pair, that is the interface reference bundle the self/this pointer together with the interface pointer. This is realized as a struct with two pointers (object and interface pointer, the fields are named `Obj` and `Iface` respectively). The object system header (`temu-c/Support/Objsys.h`) provides a macro to define a typed version of the interface reference struct. This macro is named TEMU_IFACE_REFERENCE_TYPE and will simply define a struct with the suffix `IfaceRef`. Note that the interface type (i.e. the struct with the function pointers) must have a suffix Iface for this macro to work.

There is also an untyped interface reference type with the name `temu_IfaceRef`.



Interface references are central to understand in order to be able to work with the object graph provided by the object system.

The emulator provides a function with the name `temu_connect()`, this function can be used to connect an interface reference in a specific object to an object and an interface implemented the latter object's class.

The following code snippet illustrate how to connect objects using the TEMU API.

*Listing 20. Connecting Objects*

```
// Equivalent to CLI: connect a=obja.propName b=objb:SomeIface
temu_connect(ObjA, "propName", ObjB, "SomeIface");
```

The first parameter to `temu_connect()` is an object pointer, the second parameter is the property

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

name (which must be either an interface reference property or an interface reference array property). Third is the destination object pointer, and fourth is the interface name for the destination object. Note that it is legal to connect an a property to an interface implemented by the object itself, to do this, pass the same pointer to the source and destination object arguments.

The connection function essentially fills the named property with a pointer for the destination object and the pointer for the interface struct, which is looked up by name. To call some function in the connected object one simply call the function in the interface, passing the object pointer as first parameter as illustrated in the following snippet:

*Listing 21. Calling a Function in an Interface*

```
ObjA->PropName.Iface->someFunc(ObjA->PropName.Obj);
```

It is possible to connect interface reference properties in two ways. In the first case the property is of type `teTY_IfaceRef`, in that case the connection is done by looking for the first free entry in the property field. In case all entries in the property is already connected, the connection operation fails.

The second case is when the type is `teTY_IfaceRefArray`, which is a dynamic array of interface references. This connection will always succeed assuming the destination object and interface are valid (and unless the system runs out of memory).



`temu_connect()` returns non-zero on failure (i.e. invalid property name, invalid interface name, invalid object pointers).



Connecting an interface reference property which is a static array will place the interface reference in the first free array entry (i.e. where the object and interface pointer pair are both NULL).

#### The Object Graph

The object system provides a registry of objects. The objects are connected to each other using interface references. This object connectivity is known as the object graph. As mentioned in Section 30.2.1.3, the objects are connected using the `temu_connect()` function.

### 30.2.2. Memory Mapped Devices

The primary ways to get an MMIO transaction is to implement the `MemAccessIface` in your model. The `MemAccessIface` has three functions (fetch, read and write). As arguments the functions take the object (device model pointer) and a `MemTransaction` object. The transaction object is important to fully understand.

By implementing the `MemAccessIface` the device model becomes compatible with the memory mapping functions.

The following snippet illustrates how to add support for the memory access interface:

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
#include "temu-c/Support/Objsys.h"
#include "temu-c/Memory/Memory.h"

void readFunc(void *Obj, temu_MemTransaction *MT);
void writeFunc(void *Obj, temu_MemTransaction *MT);

temu_MemAccessIface MemAccessIface = {
  NULL, // fetchFunc is optional (only used for RAM and ROM models)
  readFunc,
  writeFunc,
};

TEMU_PLUGIN_INIT
{
  temu_Class *Cls = temu_registerClass("MyClass", create, dispose);
  temu_addInterface(Cls, "MemAccessIface", "MemAccessIface",
                    &MemAccessIface);
}
```

The memory transaction object has an `Initiator` pointer, if the model needs to call functions that exits the emulator core directly (via longjump, the relevant functions in the CPU interface are tagged as noreturn functions), this should ONLY be done if there is an initiator object specified. Without an initiator object, the transaction is initiated from elsewhere (e.g. from a manual MMU table walk).



MODELS SHOULD NOT USE FUNCTIONS THAT LONGJMP TO THE EMULATOR CORE IF THERE IS NO INITIATOR OBJECT.

An object implementing the MemAccessIface can be mapped into a memory space. Normally the memory-mapped I/O model will provide a set of registers. There is no special register type, but most registers are implemented as properties, where the read and write functions act as register read and writes. It is necessary for the device model to provide functions implementing the `MemAccessIface` that dispatch reads and writes to the correct functions.

The normal way to do the dispatching is via a switch on the `Offset` field in the memory transaction object.

*Listing 22. Memory Mapped I/O Interface Usage*

```
#include "temu-c/Memory/Memory.h"

void
readFunc(void *Obj, temu_MemTransaction *MT)
{
  switch (Mt->Offset) {
  case 0:
    // Unwrap property
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
    Mt->Value = temu_propValueU32(readRegA(Obj, 0));
    break;
  case 4:
    Mt->Value = temu_propValueU32(readRegB(Obj, 0));
    break;
  }
  Mt->Cycles = 0; // Cost for this memory transaction in cycles
}
```



Registers are properties in the structure which have read and write handlers implementing the semantics. The properties can be written to from the API and the command line interface. This way unit testing of device models is very simple since device models can be stimulated without writing custom code for the target processor.

### 30.2.3. Device Resets and Mappings

The DeviceIface can be implemented by a device model for handling resets and mapping of devices to memory (a device should normally not need to know where it is mapped, but it may be used for automatic update of plug-and-play info that needs to reflect the MMIO address mappings).

The DeviceIface must be implemented in case the device must support resets. The reset function takes an int as parameter, specifying the reset type. By convention, 0 means a cold reset, while 1 means a warm reset.

```
#include "temu-c/Models/Device.h"

static void
reset(void *Obj, int ResetType)
{
  if (!ResetType) {
    // Cold reset
  }
  //...
}

static void
mapDevice(void *Obj, uint64_t Address, uint64_t Len)
{
  temu_logInfo(Obj, "device was mapped at %0.8x", (uint32_t)Address);
}

temu_DeviceIface DeviceIface = {
  reset,
  mapDevice,
};
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
TEMU_PLUGIN_INIT
{
  temu_Class *Cls = ...
  // To register device interface call the following in the plugin:
  // init function:
  temu_addInterface(Cls, "DeviceIface", "DeviceIface", &DeviceIface,
                    1, NULL);
}
```

### 30.2.4. Raising Interrupts

It is common that device models need to raise an interrupt. For this purpose there are two interfaces to consider. The `IrqCtrlIface` which is used to raise and / or lower interrupts. The `IrqCtrlIface` is implemented by processor models and interrupt controllers.

There is also `IrqClientIface` which is implemented by interrupt controllers so they support lazy interrupt evaluation. A device model will normally only need to bother about the `IrqCtrlIface`.

```
// Header for interface
#include "temu-c/Models/IrqController.h"

// Raise IRQ 1
Obj->IrqCtrl.Iface->raiseInterrupt(Obj->IrqCtrl.Obj, 1);

// Lower IRQ 1
Obj->IrqCtrl.Iface->lowerInterrupt(Obj->IrqCtrl.Obj, 1);
```

In most cases, a device model should be connected to an interrupt controller and not directly to the CPU.

When a multi-core system is emulated, an interrupt controller may need to distribute interrupts to different processors. There are the following alternatives for this:

- An interrupt is sent to the same CPU who initiated the interrupt (e.g. by MMIO or event execution).
- An interrupt is sent to another CPU than the one initiating the interrupt (e.g. an Inter-Processor Interrupt (IPI)).
- An interrupt is sent to several CPUs (multi- or broadcast interrupt).

In the case an IRQ is raised on the initiator CPU (or from a synchronized event) then the IRQ will be taken in natural order.

In the case the interrupt is raised on another CPU, the IRQ raising time will be at the start of the current quanta or at the start of the next quanta. That is, the IRQ may be raised earlier in apparent time for the other CPU than for the current CPU.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

### 30.2.5. Accessing Memory Contents

Some devices may need to access memory content in memory models, this can be done using the MemoryIface interface. The interface provides functions to read and write byte blocks from emulated memory. The interface is implemented by memory spaces, but would typically also be provided by memory models, CPU models and IOMMU models.

By convention, a memory space handles physical address offsets, while a CPU model would handle virtual addresses in this interface. It is only allowed to do memory content access in one device at a time. That is if one attempts to read or write a block which will span multiple device mappings (e.g. end of ROM followed by start of RAM), then the memory access will fail.

Memory content is accessed using the `MemoryIface`. Note that the model should normally be connected to the memory space for this. Although, some systems that contain an IOMMU would connect the device models directly to the IOMMU model instead.

Note that the read and write bytes functions will not trigger side effects. I.e. if the access is intended to be routed to a device model, the memory transaction interface should be used (but setting the initiator to NULL in the transaction object).

```
// Writing 128 bytes, data is given as an array of 32-bit words
Dev->Mem.Iface->writeBytes(Dev->Mem.Obj, 0x40000000, 128,
                           &Data[0], 2);

// Reading data, out data will be an array of 32-bit words
Dev->Mem.Iface->readBytes(Dev->Mem.Obj, &Data[0], 0x40000000, 128, 2);
```

### 30.2.6. Posting Timed Events

A device model that need to post timed events need to have an event queue object associated to itself. The event queue is provided by the CPU objects. That means that there are one event queue per CPU. For the cases where an event must be invoked at a synchronized time stamp (i.e. all CPUs having reached a specific time), then events can be posted by setting the `teSE_Board` sync qualifier when posting the events.

Events are registered at object construction time and have an associated ID. There are a number of ways to register events. This is done with the functions:

- `temu_eventPublishStruct()`
- `temu_eventPublish()`

The struct publication allows you to embed an event structure inside your object or to subclass the event struct (i.e. using the temu_Event struct type as the first field (normally called Super)). The second function creates an event struct in the global event struct registry, both functions return the Event ID which is globally unique in your program. This event ID is then typically saved in your class as a separate field (this field should not be registered as a property, event IDs are not guaranteed to remain the same the next run).

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

In order to post events, there are four functions available:

- `temu_eventPostCycles()`
- `temu_eventPostNanos()`
- `temu_eventPostSecs()`
- `temu_eventPostStack()`

Each of these take as parameter a queue object (a CPU object) and the event ID that was returned by thee `temu_eventPublish*()` functions.

All functions except the stack posting one post events relative to the current time as understood by the queue object.

The last argument to the functions are the "Sync" parameter. At present this can be either `teSE_Cpu`, or `teSE_Board`, this flag indicates whether the event should go on the CPU queue, or if it should be forwarded to the next level of scheduler. If the event is forwarded, it is:

1. The delta is converted to a nanosecond offset.
2. Rounded to the start of the next board quanta if less than it.
3. Inserted in the relevant queue.

Stacked events on the other hand, will be executed after the current instruction. A stacked event that is synchronized at board level will be executed at the end of the current time quanta.

```
// Header for event interface
#include "temu-c/Support/Event.h"

// Stack event (will be invoked after this instruction
// (or event) is handled)
temu_eventPostStack(Dev->Super.TimeSource, Dev->MyEventID, teSE_Cpu);

// Post an event 123 cycles in the future
temu_eventPostCycles(Dev->Super.TimeSource, Dev->MyEventID, 123, teSE_Cpu);

// Post a synchronized event at 42 ns in the future
temu_eventPostNanos(Dev->Super.TimeSource, Dev->MyEventID, 42, teSE_Board);

// Get delta time in cycles (useful in timer models)
delta = temu_eventGetCycles(Dev->Super.TimeSource, Dev->MyEventID);

// Deschedule event identified by function and sender pair
temu_eventDeschedule(Dev->Super.TimeSource, Dev->MyEventID);

// The object create function which is responsible for creating the
// object is responsible for registering the events.
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
void*
create(const char *Name, int Argc, const temu_CreateArg *Argv)
{
  MyDevice *Dev = new MyDevice;
  Dev->MyEventID = temu_eventPublish("myEventName", Dev, MyEventFunc);
}
```

#### Posting Initial Events

When an object is created it does not have an event queue associated with it. It is not possible to post events. To overcome this issue it is possible to implement the `ObjectIface` and its member `timeSourceSet`. This function will be called the time source property has been set and it is safe to post events.

## 30.3. Advanced

### 30.3.1. Custom Snapshots

While in most cases, snapshot-support is automatic for device model simply by registering the properties with the class. In some cases it may make sense to implement custom snapshots. To implement a custom snapshot a class must implement the `ObjectIface` (`temu-c/Support/Objsys.h`). This interface is optional but allows for custom serialisation routines to be implemented.

When saving a snapshot, the system first writes out all registered properties in an object (this process is known as serialisation). After the property serialisation, the optional `serialise` function in the (optional) `ObjectIface` is called.

When restoring a snapshot, the system first creates all objects by calling the constructors registered for a class, after this properties are restored, thirdly, if an object is of a class that implements the `deserialise` function that function is called.



When restoring a snapshot, the object constructors are called without arguments.

The snapshotting functions take two parameters, `BaseName` is the name of the file where the snapshot will be written, and `Ctxt` is a context pointer that can be used to insert and query additional property values in the snapshot file.

A typical use for the `BaseName` parameter is in the RAM and ROM models. These, by default, dump the raw data into binary files which will be named by adding a suffix to `BaseName`.

To serialize a special property in the custom snapshot interface, the following functions can be used:

```
// Write out a property
void temu_serialiseProp(void *Ctxt, const char *Name, temu_Type Typ,
                        int Count, void *Data);
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
// The following can be used to deserialise properties for objects.
// Get length of the property in the given context (i.e. number of
// elements)
int temu_snapshotGetLength(void *Ctxt, const char *Name);

// Get a value for the indexed property in the given context
temu_Propval temu_snapshotGetValue(void *Ctxt, const char *Name, int Idx);
```

To implement custom serialisation the `ObjectIface` is implemented. The following example shows how to match the `serialise` and `deserialise` functions for saving and restoring snapshots:

```
void
serialise(void *Obj, const char *BaseName, void *Ctxt)
{
  uint32_t Extra = 123;
  temu_serialiseProp(Ctxt, "myExtraProp", teTY_U32, 1, &Extra);
}

void
deserialise(void *Obj, const char *BaseName, void *Ctxt)
{
  assert(temu_snapshotGetLength(Ctxt, "myExtraProp") == 1);
  temu_Propval PV = temu_snapshotGetValue(Ctxt, "myExtraProp", 0);
}

temu_ObjectIface ObjIface = {
  serialise,
  deserialise,
  NULL, // checkSanity
};
```

### 30.3.2. DMA Access Emulation

To emulate DMA, the standard approach is as follows:

1. When DMA transaction starts, the whole data block is copied and the estimated time is computed and an event is posted at the end of transaction time.
2. When the transaction event is called, the event handler raises an interrupt.



Transferring all the data when posting the event is preferred over transferring it in the event handler. This way data may be sourced from the stack without problems, if data must be copied to memory at the event time, then a buffer must be allocated on the heap for this purpose.

The following example illustrates emulation of DMA transactions:

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

*Listing 23. DMA Emulation Example*

```
void
dmaFinished(void *Sender, void *Data)
{
  MyDevice *Dev = (MyDevice*)Sender;
  Dev->IrqCtrl.Iface->raiseInterrupt(Dev->IrqCtrl.Obj, 1);
}

void
dmaStart(MyDevice *Dev)
{
  uint64_t TransactionTime = MyDevice->DmaSize * NsPerByte;
  Dev->Mem.Iface->writeBytes(Dev->Mem.Obj, Dev->DmaAddr, Dev->DmaSize,
                             Dev->TransactionData, 2);
  Dev->Queue.Iface->postDeltaEvent(Dev->Queue.Obj, dmaFinished, Dev, NULL,
                                   TransactionTime, TEMU_EVENT_NS);
}
```

### 30.3.3. Proxy Objects

An interesting debugging technique is to use proxy objects where one implement an interface and simply forwards the calls to the same interface but as provided by another class. This is called a proxy object, and such an object can for example log calls or provide other interesting diagnostics.

Similar techniques can also be used to implement cache models in the emulator.

### 30.3.4. Using and Implementing Bus Models

The complexity of issuing a transaction on a non-MMIO bus depends on the type of bus. A point to point bus (e.g. serial) may be implemented by simple interfaces in the models where models connect directly to each other and do not strictly speaking need a separate bus model class.

However, multi-point links typically needs a bus model to work, this bus-model is for example responsible for routing messages to the correct model.



Even point-to-point buses like serial could use a bus model object in order to ensure that a bus client behaves appropriately (e.g. verify bandwidth usage), or to implement hardware flow-control emulation, or to be able to inject errors.



There is no 'generic' bus model as connecting a *SpaceWire* connector to a serial port or to a *MIL-1553-STD* bus makes no sense. Thus each bus needs its own interface (and optionally a bus-model component). This approach also ensures a degree of type-safety.

When constructing a more complex bus model, one way is to provide a connect function, and ensure that users simply set the bus property in the models connecting to the bus. When this property is set, the connect function in the bus-object is called with whichever parameters are

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

relevant for the bus in question.

There are also other issues to taken into account. For example, if the bus model is frame-based, in order to be able to interrupt a transaction (e.g. after half of it was sent), it may be useful to be able to issue two messages issued of one. That is, one begin transaction and one end transaction message. The exact requirements depend on the accuracy of the model one need. However, if one only models working transactions, then often a single message is needed.

*Listing 24. Bus Model Example*

```
typedef struct {
  uint16_t Dest;
  uint16_t *Data;
} MyTransaction;

typedef struct {
  void (*receive)(void *Obj, MyTransaction *T)
} MyDeviceIface;
TEMU_IFACE_REFERENCE_TYPE(MyDevice);

typedef struct {
  void (*connect)(void *Bus, uint16_t Addr, MyDeviceIfaceRef Device)
  void (*send)(void *Obj, MyTransaction *T)
} MyBusIface;
TEMU_IFACE_REFERENCE_TYPE(MyBus);

typedef struct {
  MyBusIfaceRef Bus;
} MyDevice;

MyDeviceIface MyIfaceImpl = {
  receive
};

// Connect to the bus when the bus property is written
void
writeBus(void *Obj, temu_Propval PV, int Idx)
{
  MyDevice *Dev = Obj;

  Dev->Bus = temu_propValueIfaceRef(PV);

  MyDeviceIfaceRef DevIface = {
    Dev, MyIfaceImpl;
  };

  Dev->Bus.Iface->connect(Dev->Bus.Obj, 123, DevIface);
}
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# 30.4. Examples

## 30.4.1. A Simple Model Example

In this example we implement a model with two registers that can be accessed using the read and write procedures in the memory access interface. We have a device which contain two registers (*RegA* and *RegB*). These registers are mapped at offset 0 and 4 from the device base address.

To implement this model, we implement read and write functions for registers a and b, memory transaction handlers which decode the offset and calls the relevant read or write handler. The read and write functions are important as they provide a standard interface for the TEMU runtime to call register reads and writes; this is especially useful in device model tests which does not need to deal with the rather complex memory access interface. In addition, it simplifies device debugging for the same reason when loading a device in the TEMU command line interface.

The model here can be compiled with GCC or clang using:

```
# With GCC
g++ -fPIC -shared mydevice.cpp -o libMyDevice.so

# With clang:
clang++ -fPIC -shared mydevice.cpp -o libMyDevice.so
```

Note that you do not need to link to the support library (`libTEMUSupport.so`) as the symbols from that library are resolved when the plugin is loaded.

The built model can be loaded form the command line with the import command.

*Listing 25. Example Class*

```
#include "temu-c/Support/Objsys.h"
#include "temu-c/Memory/Memory.h"

#include <stdint.h>

typedef struct {
  temu_Object Super;

  uint32_t RegA;
  uint32_t RegB;
} MyDevice;

void*
create(const char *Name, int Argc, const temu_CreateArg *Argv)
{
  MyDevice *Device = new MyDevice;
  memset(Device, 0, sizeof(MyDevice));
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
  return Device;
}

void
destroy(void *Obj)
{
  MyDevice *Dev = reinterpret_cast<MyDevice*>(Obj);
  delete Dev;
}

void
writeRegA(void *Obj, temu_Propval Val, int Idx)
{
  MyDevice *Dev = reinterpret_cast<MyDevice*>(Obj);
  // Semantics of register write goes in here

  Dev->RegA = temu_propValueU32(Val);
}


temu_Propval
readRegA(void *Obj, int Idx)
{
  MyDevice *Dev = reinterpret_cast<MyDevice*>(Obj);
  // Semantics of register read goes in here

  return temu_makePropU32(Dev->RegA);
}


void
writeRegB(void *Obj, temu_Propval Val, int Idx)
{
  MyDevice *Dev = reinterpret_cast<MyDevice*>(Obj);
  // Semantics of register write goes in here

  Dev->RegB = temu_propValueU32(Val);
}

temu_Propval
readRegB(void *Obj, int Idx)
{
  MyDevice *Dev = reinterpret_cast<MyDevice*>(Obj);
  // Semantics of register read goes in here

  return temu_makePropU32(Dev->RegB);
}
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
void
readFunc(void *Obj, temu_MemTransaction *Mt)
{
  switch (Mt->offset) {
  case 0:
    Mt->Value = temu_propValueU32(readRegA(Obj, 0));
    break;
  case 4:
    Mt->Value = temu_propValueU32(readRegB(Obj, 0));
    break;
  }
  Mt->Cycles = 0;
}

void
writeFunc(void *Obj, temu_MemTransaction *Mt)
{
  switch (Mt->offset) {
  case 0:
    writeRegA(Obj, temu_makePropU32(Mt->Value), 0);
    break;
  case 4:
    writeRegB(Obj, temu_makePropU32(Mt->Value), 0);
    break;
  }
  Mt->Cycles = 0;
}

temu_MemAccessIface MemAccessIface = {
  NULL, // fetch
  readFunc,
  writeFunc,
};

extern "C" void
temu_pluginInit(void)
{
  temu_Class *Cls = temu_registerClass("MyDevice", create, destroy);

  temu_addProperty(Cls, "regA", teTY_U32, 1,
                   offsetof(MyDevice, RegA),
           writeRegA, readRegA,
           "Register A");

  temu_addProperty(Cls, "regB", teTY_U32, 1,
                   offsetof(MyDevice, RegB),
           writeRegB, readRegB,
           "Register B");
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

```
  temu_addInterface(Cls, "MemAccessIface", &MemAccessIface);
}
```

### 30.4.2. Important Interfaces

*Listing 26. Memory Access Interface*

```
#include "temu-c/Memory/Memory.h"

typedef struct temu_MemTransaction {
  uint64_t Va;      //!< 64 bit virtual for unified 32/64 bit interface.
  uint64_t Pa;      //!< 64 bit physical address
  uint64_t Value;   //!< Resulting value (or written value)

  //! Log size of the transaction size it is at most the size of the
  //! CPUs max bus size. In case of SPARCv8, this is 4 bytes (double
  //! words are issued as two accesses).
  uint8_t Size;

  //! Used for device models, this will be filled in with the offset
  //! from the start address of the device (note it is in practice
  //! possible to add a device at multiple locations (which happens in
  //! some rare cases)).
  uint64_t Offset;
  void *Initiator; //!< Initiator of the transaction
  void *Page;      //!< Page pointer (for caching)
  uint64_t Cycles; //!< Cycle cost for memory access
} temu_MemTransaction;

// Exposed to the emulator core by a memory object.
typedef struct temu_MemAccessIface {
  void (*fetch)(void *Obj, temu_MemTransaction *Mt);
  void (*read)(void *Obj, temu_MemTransaction *Mt);
  void (*write)(void *Obj, temu_MemTransaction *Mt);
} temu_MemAccessIface;
```

*Listing 27. IRQ Interface*

```
#include "temu-c/Models/IrqController.h"

typedef struct temu_IrqControllerIface {
  void (*raiseInterrupt)(void *Obj, uint8_t Irq);
  void (*ackInterrupt)(void *Obj, uint8_t Irq);
} temu_IrqCtrlIface;
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

*Listing 28. Device Interface*

```
#include "temu-c/Models/Device.h"

typedef struct temu_DeviceIface {
  void (*reset)(void *Obj, int ResetType);
  void (*mapDevice)(void *Obj, uint64_t Address, uint64_t Len);
} temu_DeviceIface;
```

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# Chapter 31. TEMU System Debugger GUI

## 31.1. Overview

### 31.1.1. Introduction

The TEMU System Debugger is a graphical system debugger for TEMU targets. It connects to debug targets via the Target Communication Framework (TCF) protocol, allowing you to:

- Control execution (run, pause, step, reverse).
- Inspect source code (locally) and disassembly (target).
- Examine memory, registers, and object properties.
- Inspect Symbols and memory values with addition memory area type assumtion.
- Manage breakpoints, watchpoints, and complex breakpoint patterns.
- Navigate call stacks and variables.
- Review timeline events.
- Run TEMU commands from the integrated terminal.

The interface uses a flexible dock-based layout.

### 31.1.2. Getting Started

#### Prerequisites

- TEMU installed and configured with your target system
- TCF server running in TEMU.

Example TCF server setup:

```
// TCF server is in the debugging plugin
import Debugging

TCFServer.new name=tcfServer
tcfServer.contextManager = cm0
connect-timesource obj=tcfServer ts=l2-cpu
tcfServer.start-server port=9000
```

#### Quick Start

1. Launch the TEMU System Debugger.
2. Connect via TCF → Connect (or toolbar button):
    - Enter the port (default: 9000).

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- Click Connect.

3. Run Software in TEMU.
4. Begin debugging — use the breakpoints and run controls to debug.

### 31.1.3. Main Interface Overview

The TEMU System Debugger main window combines a central code workspace with multiple dockable tool panels. The arrangement is optimized for switching quickly between source-level and system-level debugging tasks.

#### Central Workspace

The center of the window hosts the source and disassembly views. Use <menu>View <font size="1.15em"><color rgb="b12146">›</color></font> Central Layout <font size="1.15em"><color rgb="b12146">›</color></font> Source Only</menu>, <menu>View <font size="1.15em"><color rgb="b12146">›</color></font> Central Layout <font size="1.15em"><color rgb="b12146">›</color></font> Disassembly Only</menu>, or <menu>View <font size="1.15em"><color rgb="b12146">›</color></font> Central Layout <font size="1.15em"><color rgb="b12146">›</color></font> Split View</menu> to choose the layout.

- `Ctrl` + `1` switches to source only.
- `Ctrl` + `2` switches to disassembly only.
- `Ctrl` + `3` restores the split view.
- In split mode, a horizontal splitter separates the two panes and remembers the last usable size ratio.
- If another action needs a hidden pane, such as jumping to source or disassembling a symbol, the debugger automatically makes that pane visible.

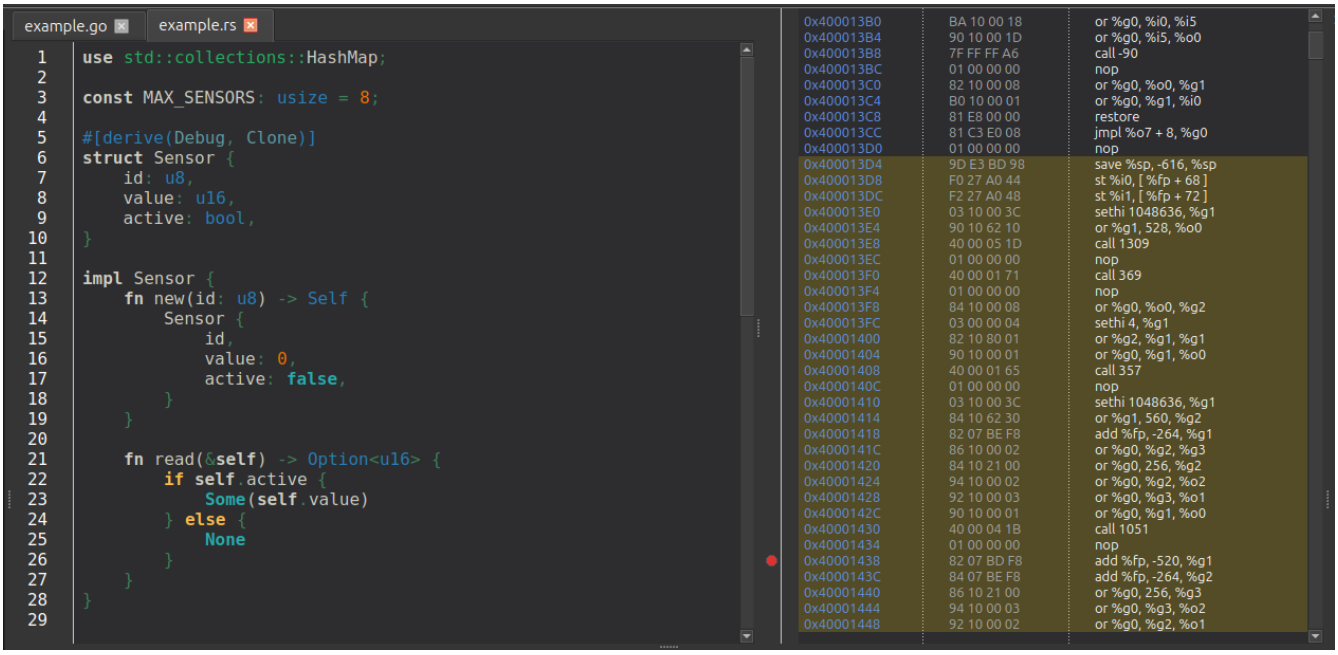



*Figure 4. Split central layouts*

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

### Run Control Toolbar

The run control toolbar is docked at the top of the window by default. It contains the execution buttons, reverse-debugging controls, context selection, and the connection menu described in Section 31.2.9.

### Default Dock Layout

Around the central workspace, the debugger creates several dock widgets. Their default placement is:

**Left side**

Components, Breakpoints / Watchpoints, Complex Breakpoints, and Project Explorer. These views are tabbed together so you can switch between them without consuming extra horizontal space.

**Right side**

Stack Trace & Variables and the Inspector View. The Inspector provides symbol, memory, and timeline event inspection capabilities in a unified tabbed interface.

**Bottom side**

Terminal, Memory, Registers, and Object Properties are tabbed together, while Logs and Timeline form a second bottom-side group.

Every dock can be moved, floated, or merged into a different tab group. More detail is covered in Section 31.3.1.

### Menu Bar

The main menus expose the high-level GUI controls:

**File**

Contains application-level actions such as <menu>File <font size="1.15em"><color rgb="b12146">›</color></font> Quit</menu>.

**View**

Holds the central layout actions for source, disassembly, and split view.

**Main window**

Controls global dock behavior, including animated docks, nested docks, tabbed docks, vertical tabs, and grouped dragging.

**TCF**

Provides connection and run-control commands from the same services used by the run control toolbar.

**CTF**

Contains timeline polling controls and target-context selection for trace collection.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

#### Dock Widgets

Contains the dock-corner assignment menus.

#### About

Opens the About dialog.

#### Status Bar

The status bar shows two kinds of information:

- Temporary status messages appear on the left.
- The Simulated Real Time (SRT) display appears on the right and updates as time information arrives from the target.

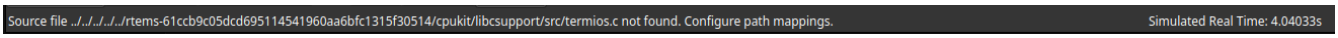



*Figure 5. Status bar with temporary messages on the left and simulated time on the right*

### 31.1.4. Connecting to a Debug Target

#### Opening the Connection Dialog

Access the connection dialog via: - TCF → Connect from the menu bar - Connect button on the run control toolbar (chain link icon)

#### Connection Settings

- IP Address: Enter the hostname or IP address of the machine running the TCF server. Use localhost or 127.0.0.1 if TEMU is running locally.
- Port: The TCF server port number. Default is 9000. This must match the port configured in your TEMU TCF server settings.

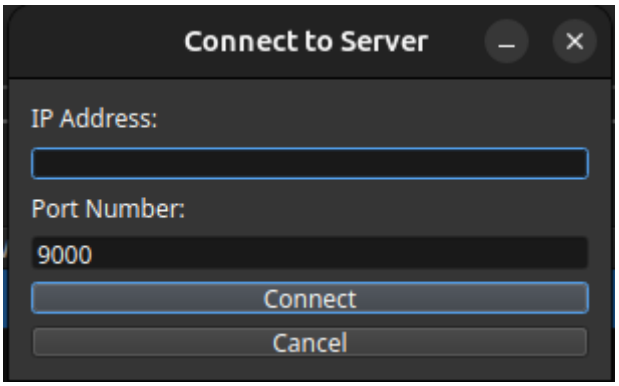



*Figure 6. Connection dialogue with defualt placeholder*

#### Connection Process

- Click Connect — the button changes to "Connecting..."
- The debugger establishes a TCF protocol handshake
- On success, the dialog closes and the Component Tree populates with available contexts

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- On failure, an error dialog appears with the reason

#### After Connecting

- The Component Browser shows available CPUs, peripherals, Software, Objects, CUs and symbols
- Select a CPU or Thread context from Run control toolbar to enable debugging features
- The context dropdown in the toolbar lists available execution contexts

#### Disconnecting

- TCF → Disconnect from the menu bar or use the Disconnect toolbar button
- The debugger returns to an unconnected state. All views clear until a new connection is established.

#### Launching TEMU (Not Enabled)

If TEMU is not already running, you can launch it directly from the debugger: - TCF → Launch TEMU or the toolbar Launch button - To stop: TCF → Terminate TEMU

# 31.2. Views

## 31.2.1. Component Browser

The Component Browser is the main navigation tree for live target contexts, boards, symbols, and TEMU object paths. It is the quickest way to move from a high-level target overview into specific GUI tools such as memory, registers, disassembly, stack traces, watchpoints, and object inspection and memory/symbol inspections.

#### Opening and Refreshing the Tree

The Components dock populates after a successful TCF connection. If the target disconnects, the tree is cleared and repopulates on the next connection.

The dock title bar includes a refresh button. Use it when the target has created or removed contexts and you want the browser to reload immediately.

#### Structure and Default Expansion

The tree is hierarchical:

**Root nodes**

Top-level containers such as boards or object groupings.

**Intermediate nodes**

CPUs, peripherals, software contexts, threads, symbols, or object folders.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**Leaf nodes**

The most specific item available for the selected branch, such as a concrete symbol or object.

After a refresh, the debugger automatically expands the main board structure so you can reach the interesting runtime nodes quickly. The Symbols container remains collapsed until you open it explicitly.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

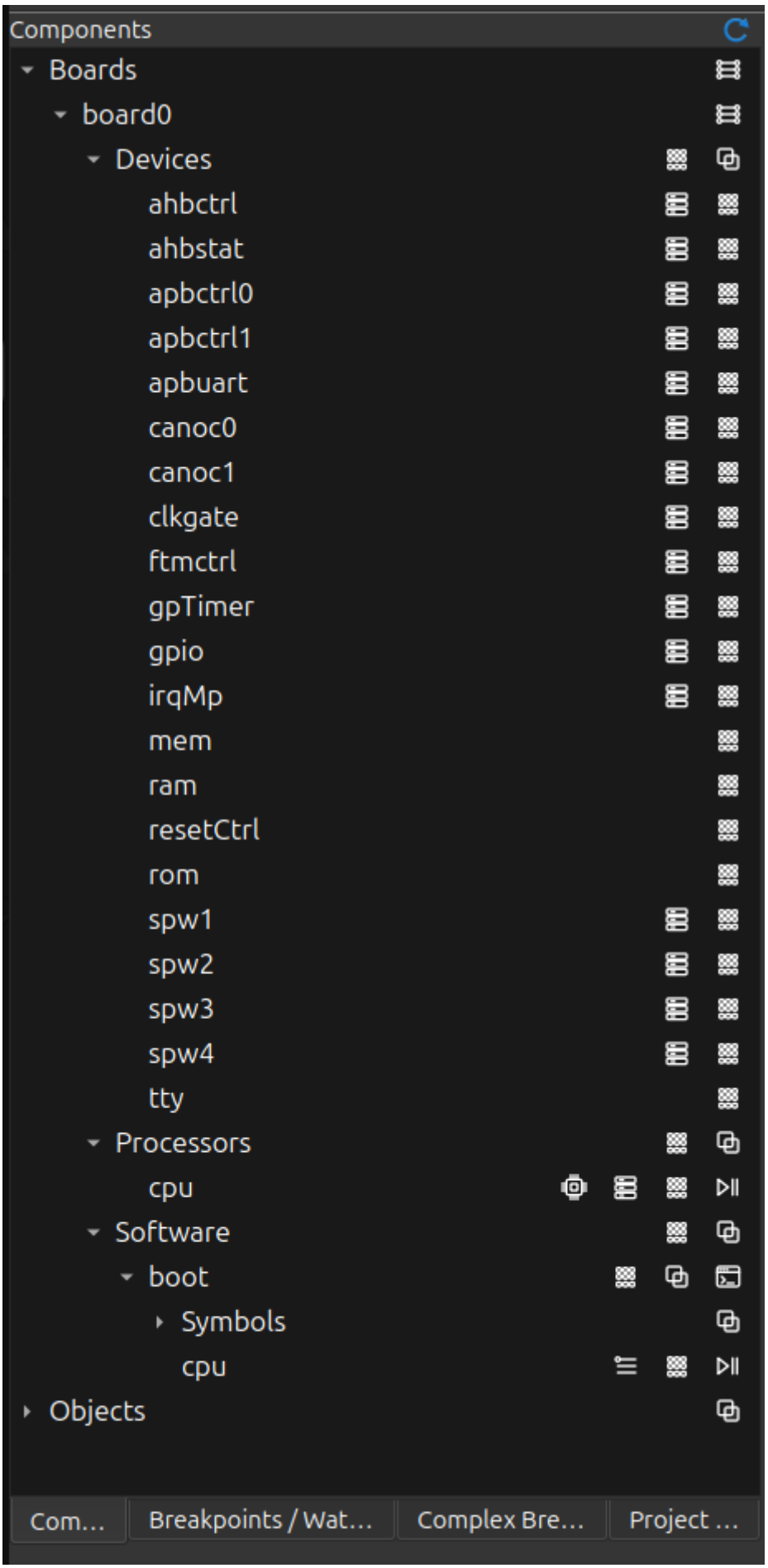



*Figure 7. Component Browser showing boards, contexts, and symbols*

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

#### Navigation

- Use the disclosure triangles to expand or collapse a branch.
- Hovering a node shows a tooltip with the available metadata for that context.
- Selecting a different row updates the debugger's current context tracking.
- Double-clicking a row requests a stack trace for that context.

#### Context Menu Actions

Right-click a node to open the context-sensitive menu. The available actions depend on the type of node you selected.

**Get Stack Trace**

Requests a stack trace for the selected context and opens the stack/variables panel with the result.

**Get Registers**

Loads registers for the selected context into the Registers panel.

**View Memory**

Opens the Memory view at the selected context when the memory service is available.

**Get Disassembly Capabilities**

Available on CPU contexts. Use it before disassembling when you want the debugger to query supported instruction-set options from the target.

**Disassemble Symbol**

Available on symbol and compilation-unit nodes. Opens the disassembly view for the selected symbol.

**Add Watchpoint**

Available on symbol leaves. Creates a watchpoint for that symbol and routes it to the watchpoint system.

**Inspect Symbol**

Available on symbol leaves. Opens the Inspector view with a detailed tree showing the symbol's type structure, all fields, nested composites, arrays, and current values. See Section 31.2.11 for more details.

**Inspect Object Properties**

Available under the `/Objects` branch. Opens the Object Properties panel for the selected object path.

#### Object Properties Panel

The Object Properties panel is tightly coupled to the Component Browser. When you choose **Inspect**

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**Object Properties**, the bottom dock shows a hierarchical property tree for the selected object.

The table provides these columns:

**Name**

The object or property name.

**Kind**

The property kind or class information.

**Value**

The current value when one is available.

**ID**

The underlying object or property identifier.

Expanding an object row requests its properties on demand, so large object trees do not need to load everything at once. The refresh button on the dock title bar asks the debugger to rebuild the browser and restore the expanded objects where possible.

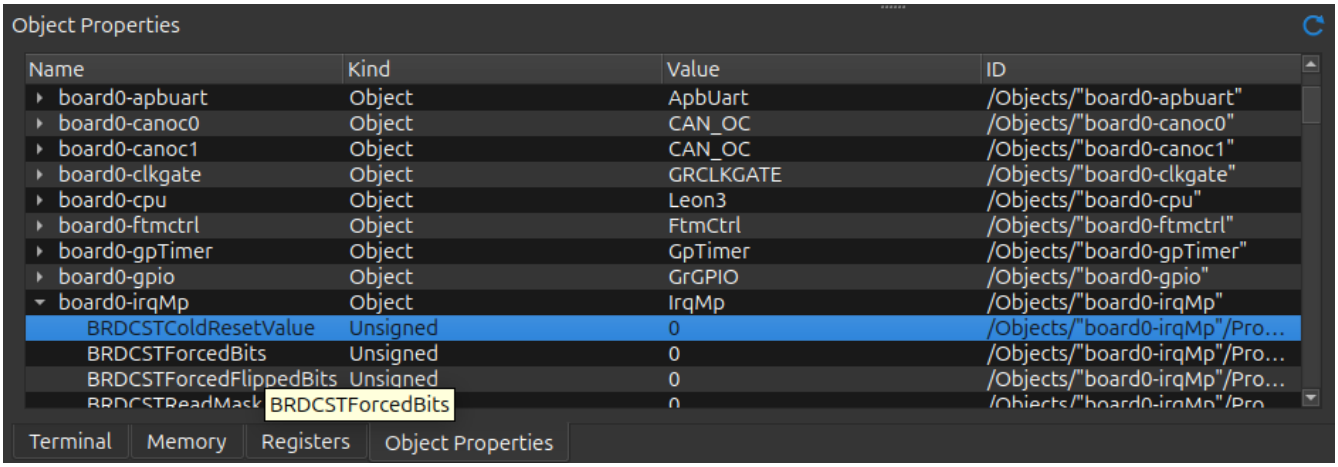
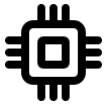



*Figure 8. Object properties panel*

### Capability Icons on Nodes

Nodes in the Component Browser can show multiple icons on the right side of the row. Each icon describes either a capability that the selected context exposes or a special role that the node plays in the tree.

| Icon | Meaning |
|---|---|
|  | **CPU** — the node is a processor execution context. This is typically the context you use for disassembly and CPU-focused navigation. |
|  | **Thread** — the node represents a thread context. Thread nodes are the usual entry point for stack traces and thread-specific run control. |

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

| Icon | Meaning |
|---|---|
|  | **Registers** — the context exposes register access. The **Get Registers** action is meaningful on these nodes. |
|  | **Memory** — the context exposes memory access. The **View Memory** action is available when this capability is present. |
|  | **Run Control** — the context can be suspended, resumed, or both. These nodes participate directly in debugger execution control. |
|  | **Contains children** — the node is a container or grouping node with nested contexts below it. This is used for structural folders such as the **Symbols** container and other expandable groupings. |
|  | **Object** — the node belongs to the `/Objects` hierarchy and represents an inspectable object rather than one of its properties. |
|  | **Property** — the node is part of the `/Objects/⋯/Properties/⋯` hierarchy. Use these nodes when inspecting object properties in the Object Properties panel. |
|  | **Software** — the node belongs to the software/application branch of the target model. This helps distinguish software contexts from board, CPU, or object nodes. |
|  | **Symbol** — the node is in the symbol hierarchy, such as a symbol leaf or an entry under an application's **Symbols** branch. These nodes are the ones used for symbol inspection, watchpoints, and symbol-based disassembly. |
|  | **Generic context** — the node has no more specific icon match, so the browser shows the default bus-style marker. You will commonly see it on generic buses, peripheral groupings, or other structural contexts. |



The Component Browser is read-only. Change state through the appropriate debugger service, scripting workflow, or other dedicated GUI action rather than by editing the tree directly.

### 31.2.2. Project Management & File Explorer

The Project Management & File Explorer panel is the main place to create, organize, and maintain project resources. Use it to manage project structure, files, and mappings between your local workspace (source files) and target paths.

#### Key Panel Capabilities

- Create and open projects
- Browse files and folders in a tree view
- Perform file and folder operations
- Configure local-to-target path mappings

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

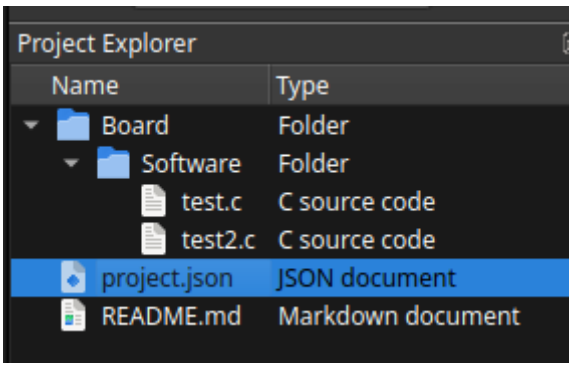



*Figure 9. Project Manager View with template file setup*

#### Creating Projects

Use this workflow to create a new project:

1. Open the Project Management & File Explorer panel.
2. Right click.
3. Select **New Project**.
4. Enter project metadata:
    a. Project Root Directory
    b. Project Name
5. Confirm with **Create**.

#### Expected Result

- A new project is added to the explorer.
- The root directory appears in the file tree.

#### Tips

- Use a clear, unique project name.
- Keep project roots inside a dedicated workspace directory.

#### File Operations

Use the explorer context menu (right-click) or toolbar actions.

#### Create a New File

1. Select a target folder.
2. Click **New File**.
3. Enter file name and extension.
4. Press Enter to create.

#### Create a New Directory

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

1. Select a parent folder.
2. Click **New Folder** (or **New Directory**).
3. Enter folder name.
4. Press Enter to create.

#### Rename File/Folder

1. Select the item.
2. Choose **Rename**.
3. Enter the new name.
4. Confirm to apply changes.

#### Delete File/Folder

1. Select the item.
2. Choose **Delete**.
3. Confirm the deletion prompt.

#### Best Practices

- Rename before moving when possible to reduce broken references.
- Review delete confirmations carefully, especially for directories.
- Use version control to recover accidental changes.

#### Path Mapping

Path Mapping links local project paths to target/runtime paths so files are resolved correctly during debugging.

#### Why It Matters

- Ensures source paths match target execution paths.
- Improves symbol resolution and breakpoint accuracy.
- Prevents file mismatch during remote operations.

#### Configure Mapping

1. Open project settings.
2. Navigate to **Path Mapping**.
3. Add a mapping entry:
    a. Local Path (workspace path on Linux, e.g. /home/user/project/src)
    b. Target Path (runtime/remote path, e.g. /opt/app/src)
4. Save configuration.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

5. Validate mapping with a test run or sync action.

| Local Path | Target Path |
|---|---|
| `Project/sourceapp` | `compSys/approot` |
| `Project/sourceapp/config` | `compSys/approot/config` |

#### Troubleshooting

- If breakpoints are not hit, verify path case and exact directory structure.
- If files do not sync, confirm mapping points to existing directories.
- If multiple mappings overlap, prioritize the most specific path rules.

## 31.2.3. Source Code View

### Source Code View

The **Source Code View** is the main editor used when you inspect the target's program text. It provides a tabbed interface for opening, browsing and editing source files while you are debugging.

#### Opening & managing files

- **Double-click** a file in the **File Explorer** (or any project tree) to open it in a new tab. If the file is already open the existing tab will be activated instead.
- Recent files remain visible as tabs for one-click access.
- Drag tabs to reorder, or right-click a tab for options such as **Close**, **Close Others** or **Split Editor**.
- Cycle through open tabs with `Ctrl+Tab` / `Ctrl+Shift+Tab`.

Tabs are identified by their full canonical path, so you won't accidentally open the same file twice from different locations.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

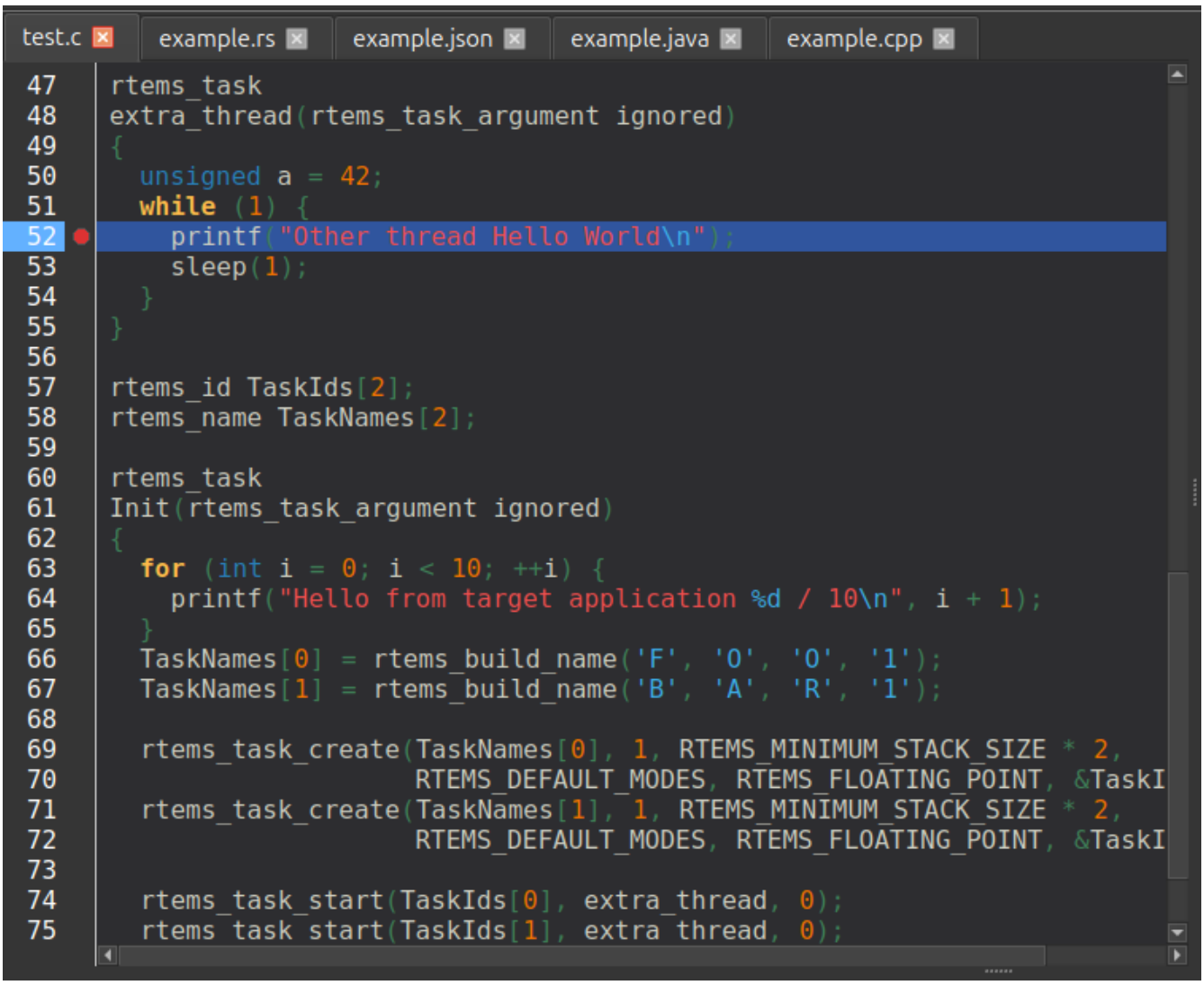



*Figure 10. Source view with suspended thread on a Breakpoint*

#### Navigation

- Explorer clicks are the fastest way to bring a source file into view.

#### Execution¬Point Indicators

- When the debugger is paused the line currently being executed is highlighted in the active source tab.
- If several threads are stopped, each thread is marked with a coloured band beside its line; hovering shows the thread's ID or name.
- Picking a different stack frame in the call-stack panel moves the highlight to that frame's location.
- The view scrolls automatically to keep the highlighted line visible when you step through code.

#### Breakpoints from the source view

- Click in the left **gutter** (the margin beside line numbers) to toggle a breakpoint on that line. A red circle marks an active breakpoint.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- Breakpoints set here are kept in sync with the debugger and remain between sessions.
- Disabled breakpoints appear as hollow circles.

#### Scrolling & zooming

- Use the mouse wheel to scroll through lines.

#### Additional

- Line numbers are always shown and follow the current execution highlight.
- If no file is loaded the editor displays **No source loaded**.

The source editor is designed for speed; it renders text itself rather than relying on a full document model, so large files load quickly.

### 31.2.4. Disassembly View

#### Introduction

The disassembly view in the TEMU System Debugger lets you inspect the raw machine instructions that correspond to the program you are debugging. It provides a window into what the CPU sees, helping you understand low-level behavior, investigate crashes, and verify that your code is executing as expected.

#### Opening the Disassembly View

The disassembly window is usually accessible from the main menu On bootup or can be switched to via the 'View' menu in the system toolbar. It becomes enabled once you connect to a target and a CPU with ISA capabilities is available. If the view is empty, you may need to select a thread and disassembele a stack frame in the callstack contextmenu or resume execution until instructions are available.

#### Basic Layout

The view is divided into three columns:

**Address**
: shows the memory address of the instruction.

**Opcode**
: displays the raw bytes of the instruction in hexadecimal form.

**Instruction**
: shows the human-readable assembly mnemonic and operands.

You can resize each column by dragging the separators in the header.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

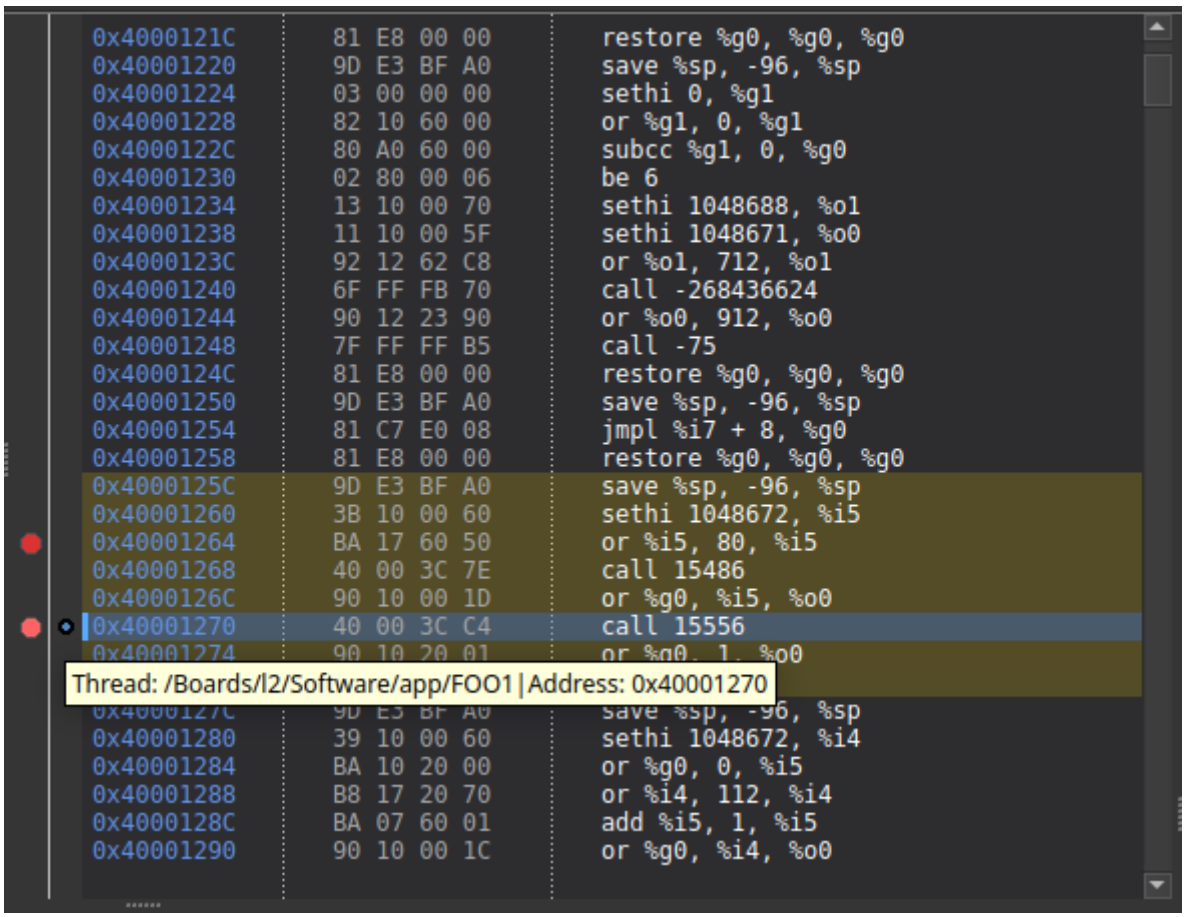



*Figure 11. Disassembly View stopped at a BP with Thread Details*

### Navigating the Code

- **Scroll** with the mouse wheel or use the scrollbar on the right. The view loads additional instructions on demand, so you can scroll through large regions without pre-loading everything.
- **Jump to address** by right-clicking a breakpoint in the breakpoint view.
- **Scroll to line** by clicking on an entry and pressing the down/up arrows or PgDn/PgUp keys.

The view employs an infinite scrolling mechanism: as you approach the top or bottom of the cached region, additional instructions are fetched automatically, keeping the display responsive.

### Selecting and Copying Instructions

Click and drag over the lines to select one or more instructions. The selection highlights across all three columns.

- **Select All** – use the Edit menu or Ctrl+A to select the entire contents of the view.
- **Copy** – press Ctrl+C or choose "Copy" from the context menu to place the selected lines on the clipboard. The copied text preserves the address, opcode and instruction columns.

### Breakpoints and Gutter Plugins

The disassembly view integrates with the breakpoint system:

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- Clicking in the left-hand gutter toggles a breakpoint on that address. A small red indicator appears.
- A custom gutter may show additional information such as thread pointers or execution markers; these are provided by plug-ins and appear automatically if enabled.

You can inspect or navigate to breakpoints using the dedicated breakpoint panel.

### Execution Point Highlighting

When the program is running, the currently executing instruction is highlighted in a distinct color. The view supports multiple execution points, one for each thread.

To jump to an execution point explicitly, use the context selection dropdown that is located in the Run Control Bar. The relevant line is centered in the view.

### Context Menu Actions

Right-clicking within the disassembly view opens a context menu offering:

- **Copy** – copy selected lines or the line under the cursor. The option to copy from address, opcode or instruction will be available in the menu depending on the region you click.

### Working with Symbol Highlights

Symbols imported from the debug information (e.g. function or variable names) are automatically highlighted in the disassembly when you navigate to their addresses. You can also manually specify a range corresponding to a particular memory sector via the memory view; the view will draw a shaded rectangle around that region.

### Choosing Instruction Set Architecture (ISA)

If the target supports multiple ISAs (for example, in a mixed-mode system), the view will offer a selection list of available architectures. Choose the ISA that matches the code you are examining to ensure correct decoding of opcodes.

The debugger retrieves the list of supported ISAs from the target when a connection is established and caches it for reuse. The current ISA selection affects all disassembly requests.

### Chunked Disassembly

To preserve performance when scrolling large address ranges, the view performs disassembly in chunks. Each chunk consists of a set number of instructions (typically a few thousand) and is stored in an internal cache. When you move to a new area of memory, only the necessary chunks are fetched from the target.

### Error Handling

If the debugger cannot retrieve disassembly data (for example, the target is not connected or the address is invalid), an error message appears briefly in the status bar. You can retry by

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

re-requesting disassembly from the context menu or by re-establishing the connection.

#### Tips and Best Practices

- When working with self-modifying code, refresh the view frequently to ensure you see the latest bytes.

### 31.2.5. Memory Inspection

This document describes the *Memory View* in the TEMU System Debugger. It is written for users of the debugger and explains the interface and interaction patterns you will use when examining live memory. No programming knowledge is required.

#### Overview

The Memory View presents a block of target memory organized in a familiar "hex editor" layout. An address column on the left shows the start of each row, followed by a grid of hexadecimal byte values and an ASCII translation on the right. You can scroll, jump to addresses, copy values, and even request disassembly of instructions found in memory. The control panel at the top lets you customise the view and navigate through contexts provided by the debugger.




*Figure 12. Highlighted area in the memory view*

#### Opening the Memory View

The view is available from the main menu or via a tab button labelled **Memory** around the dock. When you first open it no memory is displayed. The view will automatically load the memory associated with the current debugger context once the connection is established.

#### Control Panel

At the top of the Memory View is the control panel, which contains all of the interactive widgets you will use frequently.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

#### Address Navigation

**Address**

A hexadecimal text box where you may enter any address you wish to inspect.

**Type the address with or without the `0x` prefix and press Enter**

or click the **[ Go ]** button to jump. The view aligns the requested address to the beginning of a row.

**Go**

Executes the jump to the address entered in the box.

**Refresh**

Forces the view to re-read memory at the current address. This is useful if the target process has modified memory and you want to see the latest contents immediately.

#### Display Options

**Bytes per row**

A spin box allowing you to choose how many bytes are shown on each line. Valid values range from 8 to 64. Changing this value automatically reloads the view and keeps your current position.

**Freeze**

A checkbox that toggles auto-refresh. When unchecked the view periodically reloads memory to reflect changes in the target (default behaviour). Checking **Freeze** stops automatic updates and locks the view at the current contents. Selecting data will also enable the freeze automatically so that the selection does not blur.

#### Copy Mode

A single button shows the current copy mode. Clicking it cycles through three modes:

**[ Linked ]**

Copies all three representations together: the hex bytes, the ASCII rendering, and the raw byte string.

**[ Hex ]**

Copies only the hexadecimal byte values.

**[ ASCII ]**

Copies only the printable ASCII rendering (non-printable bytes become `.`).

A tooltip on the button explains these options in more detail. The button's appearance changes to reflect the active mode.

#### Context Information

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

A descriptive label shows which memory context is currently active. This typically reflects the application or memory device you are connected to and updates automatically when the debugger switches contexts. While contexts are being discovered the label displays a message such as "Discovering contexts..." and the address controls are disabled until a valid context is selected.

### Scrolling and Navigation

Memory is displayed in an "infinite scroll" container. You can use the vertical scroll wheel, drag the scrollbar, or press the **Page Up**/**Page Down** keys to move through memory. The view fetches data as needed, so you can navigate across a very large address space without loading everything at once.

Clicking on an address (the value shown in the address column) will take you directly to that address. You can also use the horizontal scrollbar to adjust the display offset, effectively shifting the columns left or right if they do not fit on screen.

The view supports keyboard navigation:

`` / ``
: Move the caret one byte at a time.

`Ctrl+A`
: Selects all visible memory.

`Ctrl+C`
: Copies the current selection according to the active copy mode.

### Selection and Copying

You can select a range of bytes by clicking and dragging in either the hex or ASCII area. Selection extends across rows naturally. Cross-row selection is preserved when you scroll or resize the window. Selected bytes are highlighted and the freeze checkbox will be enabled automatically to prevent the display from changing while you examine or copy data.

When a selection exists the view emits information about it; the log output (if enabled) will show both the hex and ASCII representation of the selected data. Use **Ctrl-C** or the context menu to copy the selection in the chosen format. The context menu also offers convenient commands such as "Copy Address" or "Copy Hex" regardless of the current copy mode.

### Context Menu

Right-clicking anywhere in the memory area brings up a context menu with additional actions:

**Copy**
: Copies the currently selected bytes.

**Copy Address**
: Places the address under the cursor on the clipboard.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**Copy Hex/ASCII**

Explicit control over which representation is copied.

**Select All**

Selects every byte currently visible.

**Inspect Memory**

Available when bytes are selected. Opens the Inspector view and attempts to automatically infer a type for the selected memory region. The Inspector displays the memory contents interpreted as a structure based on heuristics. See Section 31.2.11 for more details.

**Assume Type…**

Available when bytes are selected. Opens a dialog allowing you to manually select a type (e.g., `struct`, `int`, `float`) to interpret the selected memory. After choosing a type, the Inspector view displays the memory formatted according to that type's definition. This is useful when inspecting dynamically allocated memory or memory-mapped I/O where no symbol information exists.

**Disassemble**

If a memory region at the clicked address contains executable code, you can request disassembly. The disassembled instructions will appear in the disassembly view or a pop-up depending on configuration.

This menu adapts to the context; it remains disabled until a memory row is under the pointer.

#### Integration with Disassembly and Other Views

The Memory View is tightly connected with other components of the debugger:

- **Disassembly requests** – double-clicking (or using the context menu) on an instruction triggers a disassembly request. The disassembly window will show the corresponding machine instructions for the selected address.
- **Inspector** – selecting bytes and using **Inspect Memory** or **Assume Type…** opens the Inspector view with the memory interpreted as a typed structure. This integration allows seamless navigation from raw bytes to structured data views.
- **Context manager** – when the debugger switches to a new process/thread or load module, Memory View updates automatically. You can see which context is active in the context info label and you may manually switch by using the context manager widget elsewhere in the UI.

#### Tips and Tricks

- Use the **Bytes per row** setting to make the view wider or narrower depending on your monitor or to align with natural word sizes (32-bit vs 64-bit).
- The freeze checkbox is handy when you're watching a changing buffer such as a log file or shared memory region; enable it to pause the view and step through at your own pace.
- Change the copy mode before making a selection if you know you only need one representation;

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

this avoids having to post-process the clipboard contents.

#### Troubleshooting

- If memory fails to load, ensure that you have a valid debugging context and that the process is still running. The context info label will show errors if the memory service reports a problem.
- Very large caches are managed automatically, but if you notice sluggishness you can clear the view by clicking **Refresh** or by switching to a different address range. The application will keep only a limited amount of data in memory and will discard old pages as you move around.
- When the address input box refuses to accept an entry, check that you are using hexadecimal digits only. The box accepts both upper- and lower-case letters.

#### Keyboard Shortcuts Summary

| Keys | Action |
|---|---|
| `Enter` (in address box) | Jump to address |
| `Go` | Jump to address |
| `Refresh` | Reload memory |
| `Ctrl`+`A` | Select all |
| `Ctrl`+`C` | Copy selection (current mode) |
| `←` / `→` | Move caret/selection |
| `Page Up` / `Page Down` | Scroll one page |

### 31.2.6. Breakpoint Management

#### Overview

Breakpoints are the primary mechanism for stopping execution in the debugger. TEMU provides several tightly-integrated user interfaces for creating, inspecting and controlling breakpoints:

- **Gutter icons** in disassembly and source-code panes.
- A searchable **breakpoint tree** with context and properties.
- Visual cues (colours, hovers) and a context menu for fine-grained control.
- A separate **Complex Breakpoints** dock for multi-stage event patterns.

This document explains how to use those features from the user's perspective. No knowledge of the underlying code is required; the descriptions concentrate on what you can do and how the interface behaves.

#### Breakpoint terminology

A **disassembly breakpoint** refers to a specific memory address. A **line breakpoint** refers to a source-file/line pair. All breakpoints belong to a **context** (e.g. a software context or thread). Enabled

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

breakpoints are active – the debugger will stop when they are hit. Disabled breakpoints remain visible but won't pause execution.

### Adding and removing breakpoints

#### Using the gutter

When a disassembly or source pane is visible a narrow "gutter" runs down the left-hand side of the text.

- **Create/toggle** – click the circle in the gutter next to an instruction (disassembly) or a source line.
- An empty circle indicates no breakpoint; clicking it sets a new one.
- A filled red circle denotes an enabled breakpoint; clicking it disables or removes the breakpoint depending on its current state.
- **Hover feedback** – moving the mouse over the gutter shows a pale marker even if no breakpoint exists. This makes it easy to target a line.

#### Breakpoint tree view

Open the **Breakpoints / Watchpoints** dock (normally tabbed with the Component Browser and Project Explorer on the left side of the window). The upper half of that dock lists the regular breakpoints.

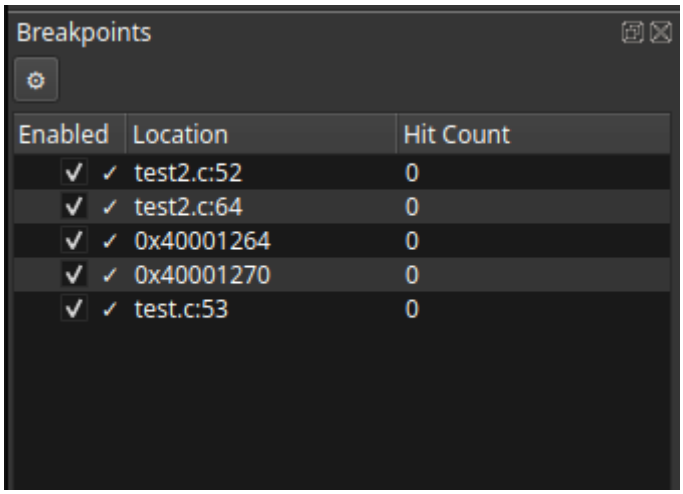



*Figure 13. Tree listing breakpoints*

- **Add** – use the toolbar + button, choose "Add breakpoint" and enter an address or file/line. The current context is used by default.
- **Enable/disable/delete** – select one or more entries and use the toolbar buttons or the right-click context menu.
- **Navigate** – double-click an entry or choose "Go to" from the menu; the disassembly/source view will scroll to that location and make it current.
- **Copy/Clear All** – the menu provides clipboard copy and a "clear all" command to remove everything in one go.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

For data-access stops, see Section 31.2.7.

#### Context¬sensitive additions

Breakpoints are context-aware.

When adding from the tree view or gutter the current context (software, thread, etc.) is automatically attached.

If you change context later, the tree filters accordingly, but existing breakpoints remain intact.

### Seeing breakpoints in the views

#### Colours & icons

- **Enabled** breakpoints appear as a solid red circle in the gutter.
- **Disabled** breakpoints are grey, with a diagonal line through the circle.
- **Hover** – when the mouse is over a line without a breakpoint, a faint red dot previews where a new breakpoint would appear.
- **Execution point** – the currently executing line is highlighted with a distinct background colour; this is separate from the breakpoint indicator.

These visual cues are theme-aware.

#### Disassembly view behaviour

The gutter plugin keeps breakpoints synchronized with the underlying model, so enabling or removing a breakpoint anywhere (gutter/tree/script) updates the display immediately.

#### Source view behaviour

- **Line-breakpoints** appear as circles in the gutter beside the source line.
- **Clicking the gutter** toggles the breakpoint for that line.
- **Multiple files** – each open file has its own gutter and breakpoints; the tree view reflects all of them.

When you set a line breakpoint, the controller attempts to map the local file path to the target's path using the project's path-mapping rules – this means breakpoints survive rebuilds and directory renames.

### Working with breakpoint properties

Property dialogs let you:

- Give a breakpoint a condition.
- Specify whether it should trigger only after a certain number of hits.
- Associate it with particular threads or contexts.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- Add TEMU and Python scripts to individual breakpoints, these scipts will trigger in the emulator not the Debugger.

See the TEMU scripting documentation: Scripting.

Access the dialog by right-clicking a breakpoint in the gutter or selecting “Edit” from the tree-view context menu.

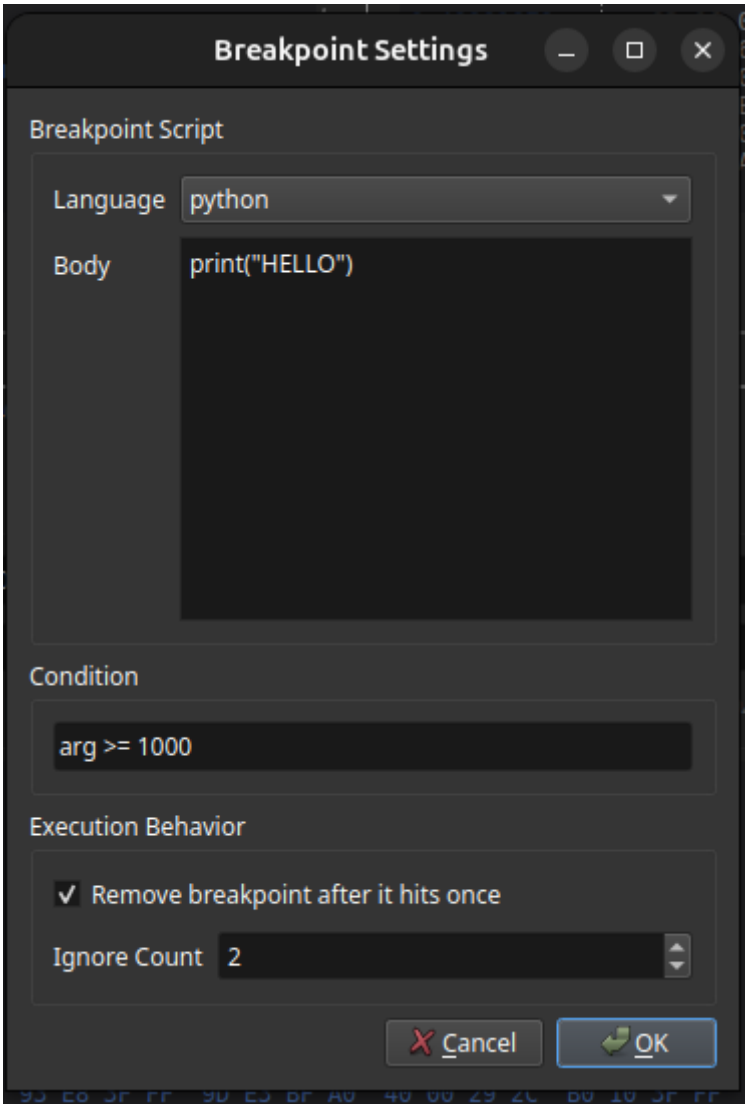



*Figure 14. Breakpoint settings dialogue*

#### Breakpoint settings dialog

The dialog is split into three functional areas:

**Breakpoint Script**

Lets you attach a script to the breakpoint. The **Language** field currently offers `python` and `tscript`, and the script body editor uses syntax highlighting that follows the selected language.

**Condition**

An optional expression guard that must evaluate true before the breakpoint stops execution.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**Execution Behavior**

Contains the **Remove breakpoint after it hits once** checkbox and the **Ignore Count** field.

In the disassembly gutter, right-clicking an existing breakpoint opens the same properties workflow for the address under the mouse.

#### Complex Breakpoints

Complex breakpoints are managed in a dedicated dock tab labelled **Complex Breakpoints**. Use that view when you need a pattern of multiple operations instead of a single stop point.

See Section 31.2.8.

#### Tips & tricks

- **Synchronize** – if you've added breakpoints via a script or another tool, click the "Refresh" button in the tree view (or use the menu) to pull them into the UI.
- **Disable all** – use the tree view's toolbar menu to quickly disable all breakpoints without deleting them.

#### Troubleshooting

- If a breakpoint does not appear where expected, verify that the disassembly pane is displaying the context you think it is (check the context selector at the top of the view).
- Line breakpoints on generated code may not align with the source; use the navigator to jump to the physical address before setting one.

### 31.2.7. Watchpoint Management

Watchpoints stop or notify on data access rather than on code location. In the TEMU System Debugger they are managed from the lower half of the **Breakpoints / Watchpoints** dock.

#### Opening the Watchpoint View

Open the **Breakpoints / Watchpoints** dock from the left-side tab group. The dock is vertically split:

- **Breakpoints** in the upper section
- **Watchpoints** in the lower section

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

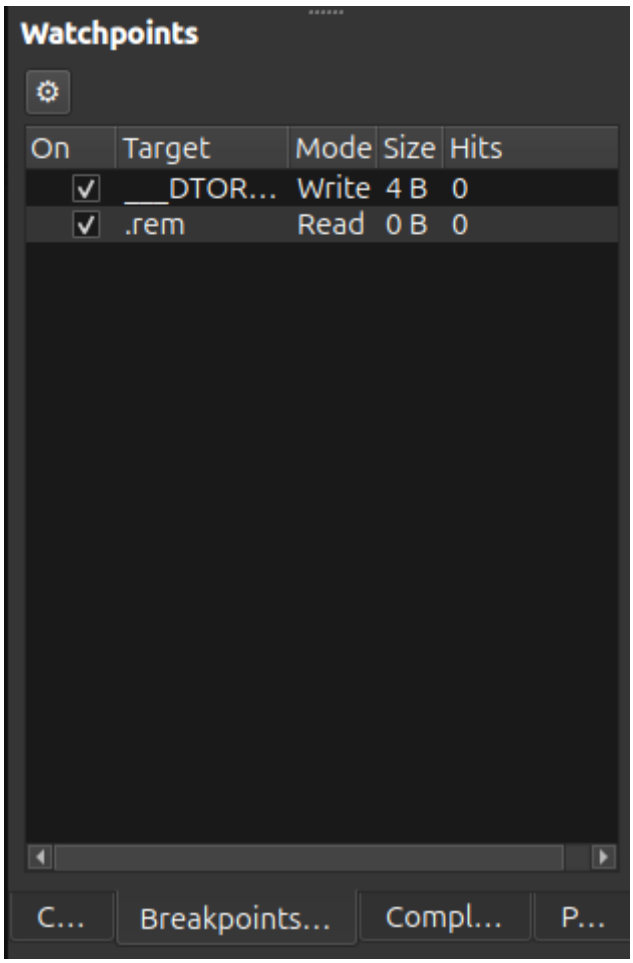



*Figure 15. Watchpoint tree view*

### Adding Watchpoints

You can create watchpoints from several GUI entry points:

**Gear menu**

Click the gear button at the top of the watchpoint section and choose **Add Watchpoint...**.

**Component Browser**

Right-click a symbol leaf and choose **Add Watchpoint**.

**Source Code View**

Right-click an identifier symbol and choose **Set Watchpoint on "<name>"**.

The add/edit dialog is the main interface when you need to configure the target, access mode, size, or other conditions manually.

### Watchpoint Settings Dialog

The **Watchpoint Settings** dialog is divided into three groups.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

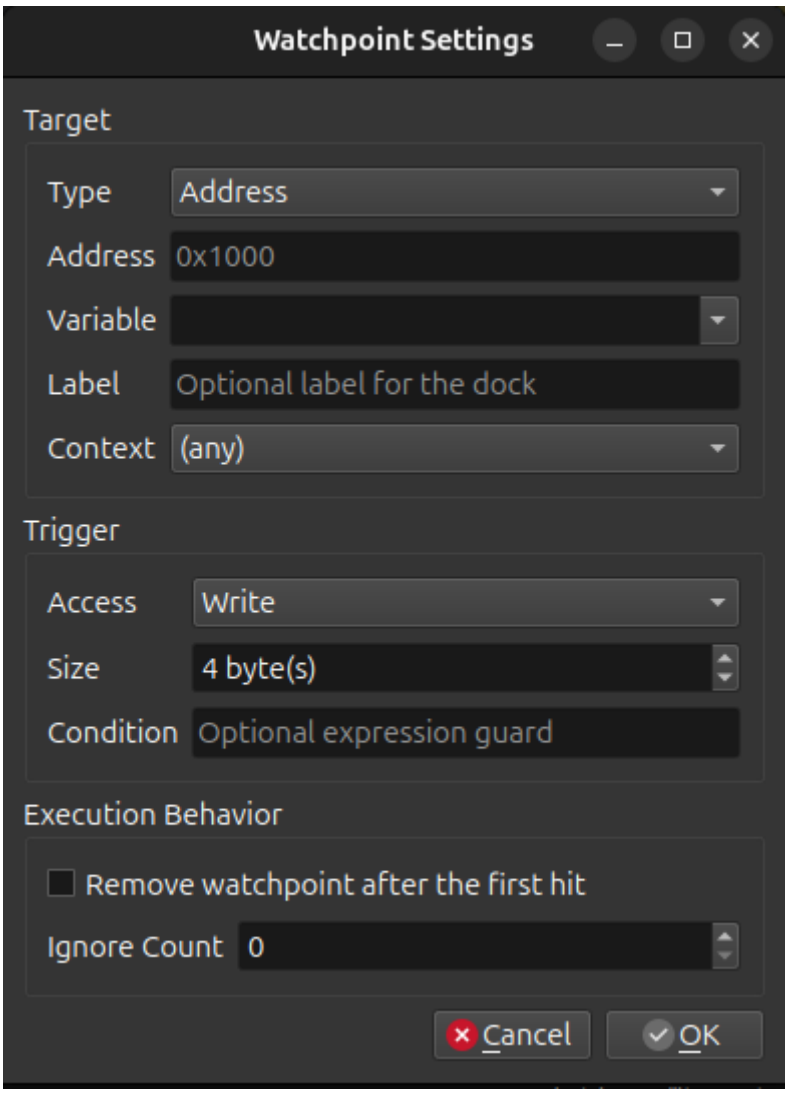



*Figure 16. Watchpoint settings dialog*

#### Target

### Type

Choose whether the watchpoint targets an **Address** or a **Variable**.

### Address

Enter a hexadecimal address such as `0x1000`.

### Variable

Search known component variables from an editable drop-down list.

### Label

Optional text used to make the watchpoint easier to recognize in the list.

### Context

Select the software context that should own the watchpoint.

#### Trigger

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**Access**

Choose **Read** or **Write**.

**Size**

Set the watched size in bytes.

**Condition**

Provide an optional expression guard.

#### Execution Behavior

**Remove watchpoint after the first hit**

Creates a one-shot watchpoint.

**Ignore Count**

Skips the first N hits before the watchpoint becomes active.

### Managing the Watchpoint List

The watchpoint table shows one row per watchpoint with these columns:

**On**

A check box that enables or disables the watchpoint.

**Target**

The watched address, symbol, or custom label.

**Mode**

The access type, such as **Read** or **Write**.

**Size**

The watched size in bytes.

**Hits**

The current hit count.

Hovering a row shows more detail, including the selected context, condition, symbol path, ignore count, and whether the watchpoint is one-shot.

### Toolbar and Context Menu Actions

The gear menu above the list provides:

- **Add Watchpoint...**
- **Enable Selected**
- **Disable Selected**
- **Edit Watchpoint**

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- **Delete Selected**
- **Clear All**

Right-clicking a watchpoint row opens a smaller context menu with:

- **Edit Watchpoint**
- **Delete**

Deleting selected rows and clearing the list both ask for confirmation first.

#### Navigation Behavior

Double-click a watchpoint row to navigate to its address. If the watchpoint belongs to another context, the debugger switches to that context first and then routes the navigation to the disassembly workflow.

#### Tips

- Use symbol-based watchpoints when you want the GUI to track the watched item by name instead of typing an address manually.
- Use one-shot watchpoints for temporary investigations where the first access is the only one that matters.

### 31.2.8. Complex Breakpoints

Complex breakpoints let you build multi-step breakpoint patterns instead of a single stop condition. They are managed from the **Complex Breakpoints** dock, which is tabbed with the regular breakpoint tools by default.

#### Opening the Complex Breakpoints Dock

Select the **Complex Breakpoints** tab from the left-side dock group. The view contains a toolbar and a tree listing the currently defined patterns.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

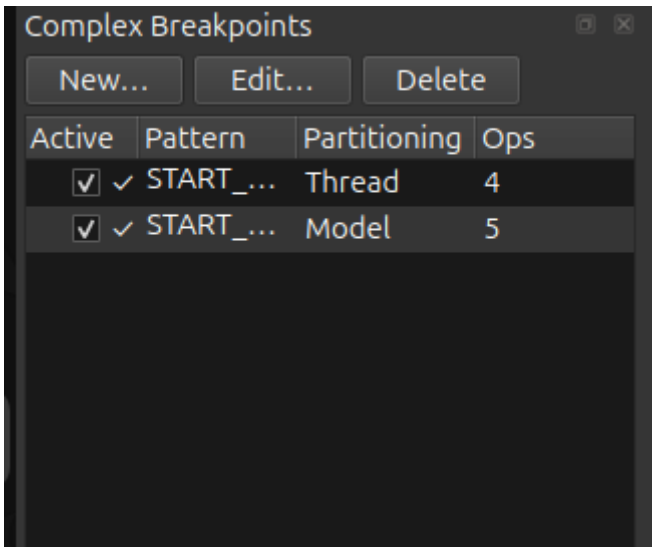



*Figure 17. Complex breakpoint listings*

### Pattern List

Each row in the list summarizes one complex breakpoint:

**Active**
: Shows whether the pattern is enabled.

**Summary**
: A compact operation-by-operation description.

**Partitioning**
: Shows whether the pattern is keyed per **Thread** or per **Model**.

**Ops**
: The number of operations in the pattern.

The summary is truncated in the table when the pattern becomes long, but the full operation breakdown is available in the tooltip.

### Creating, Editing, and Deleting Patterns

The dock toolbar provides three actions:

**New**
: Opens the pattern builder for a new complex breakpoint.

**Edit**
: Opens the selected pattern in the same builder dialog.

**Delete**
: Removes the selected pattern after confirmation.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

The row context menu offers the same actions. Double-clicking a row is a shortcut for **Edit...**.

### Pattern Builder Layout

The builder dialog is split into three main areas.

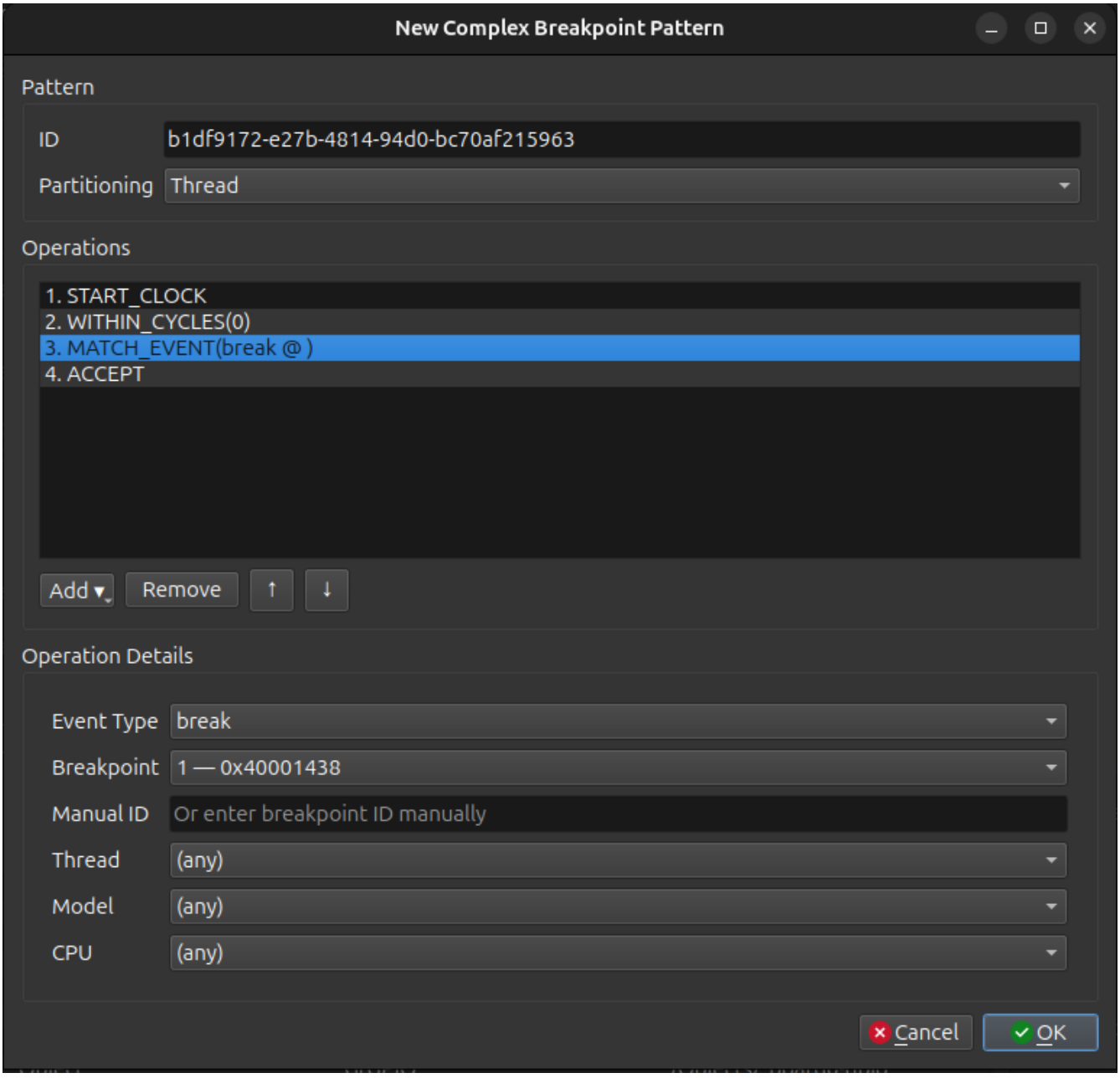



*Figure 18. Complex breakpoint pattern builder dialog*

#### Pattern Header

**ID**

A generated identifier that is shown read-only.

**Partitioning**

Choose **Thread** to track state independently per thread, or **Model** to track it per hardware object.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

#### Operations List

The center list holds the pattern program in execution order. Use the toolbar beneath it to:

- add a new operation from a drop-down menu,
- remove the selected operation,
- move the selected operation up,
- move the selected operation down.

The available operation types are:

- `NOP`
- `START_CLOCK`
- `WITHIN_NS`
- `WITHIN_CYCLES`
- `WITHIN_STEPS`
- `MATCH_EVENT`
- `NOT_EVENT_UNTIL_NEXT`
- `CAPTURE`
- `ACCEPT`

#### Operation Details

Selecting an operation opens its parameter page in the lower part of the dialog.

**MATCH_EVENT**

Choose the event type and any optional restrictions. The current GUI supports `break`, `ethPkt_tx`, and `uartByte_tx`.

For `break` events you can either pick an existing breakpoint from the list or type a breakpoint ID manually. You can also narrow the match to a specific thread, model, or CPU.

**WITHIN_NS / WITHIN_CYCLES / WITHIN_STEPS**

Enter the numeric window value for the selected unit.

**CAPTURE**

Choose the capture slot index from `0` to `3`.

**NOT_EVENT_UNTIL_NEXT**

Choose the inhibitor event that cancels the pattern if it occurs before the next match.

**NOP / START_CLOCK / ACCEPT**

These operations do not expose extra fields in the details pane.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

#### Validation Rules

The dialog only enables **OK** when:

- the pattern contains at least one operation, and
- the final operation is `ACCEPT`.

This prevents saving incomplete patterns that the debugger cannot execute.

#### Reading the Summary Column

The summary column shows the same operation order used by the builder. Examples include:

- `MATCH_EVENT(break @ <breakpoint-id>)`
- `WITHIN_CYCLES(100)`
- `CAPTURE(slot 1)`
- `NOT_EVENT_UNTIL_NEXT(break)`

This makes it possible to review a pattern at a glance without reopening it.

### 31.2.9. Execution Control

#### Overview

The Run Control Bar is the primary interface component for controlling program execution within the TEMU System Debugger. It provides both forward and reverse execution controls, context selection, and connection management all from a single toolbar.




*Figure 19. The run control bar*

#### Accessing the Run Control Bar

The toolbar normally appears at the top of the main window. It can be docked to any edge of the application window or floated as a separate panel. Drag the bar by its grip to move it.

#### Context Selection

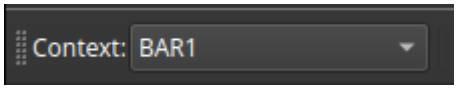



*Figure 20. Context Selection dropdown*

- A labelled dropdown (`Context:`) lets you choose the CPU or thread context for execution. Only contexts that support execution are listed.
- When context discovery is in progress, the dropdown is disabled and shows "Discovering contexts...".

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- Selecting a context updates the entire debugger to operate on that context. The dropdown reflects the current active context when it changes elsewhere in the UI.
- The minimum width ensures names are readable; long names may be truncated.

### Forward Execution Controls

The forward controls are grouped together and separated from other categories by visual separators.



*Figure 21. Forward Run Buttons*

**Continue**

Resumes execution from the current location. Shortcut: `F5`. Icon: .

**Pause**

Suspends execution at the nearest suspendable point. Shortcut: `F6`. Icon: .

**Step Over Instruction**

Execute the next instruction, treating function calls as a single step. Shortcut: `Shift` + `F10`.

**Step Into Instruction**

Execute the next instruction and enter any called functions. Shortcut: `Shift` + `F11`.

**Step Over Line**

Step to the next source line, skipping over function calls. Shortcut: `F10`.

**Step Into Line**

Step to the next source line, entering calls. Shortcut: `F11`.

**Finish Function**

Resume until the current function returns. Shortcut: `Shift` + `F5` `.

Tooltips appear when hovering over each button, including the shortcut key for quick reference.

### Reverse Execution Controls

Reverse debugging buttons mirror the forward controls and allow stepping backwards through execution history. Icons and tooltips make it clear that time is moving in reverse.



*Figure 22. Reverse Run Buttons*

**Reverse Continue**

Resume execution backwards until a breakpoint or user stop.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**Reverse Over Instruction**

Step backwards one instruction, skipping calls.

**Reverse Into Instruction**

Step backwards into any called function.

**Reverse Over Line**

Step backwards one source line, skipping calls.

**Reverse Into Line**

Step backwards into a line's function calls.

**Reverse To Call**

Go backwards until reaching the last call instruction (opposite of Finish).

### Active Context Controls

When working with multiple contexts the following buttons help re-synchronize with the active context state:



*Figure 23. Resume/Reverse Run Buttons*

**Resume Until Active**

Run forwards until the context becomes active again.

**Reverse Until Active**

Run backwards until the context becomes active.

These buttons are useful in multi-core or multi-thread scenarios where contexts may be scheduled in and out.

### Connection Controls

A connection icon appears at the right-hand end of the bar. Clicking it opens an overflow menu with the following options:



*Figure 24. The right is the connection button*

**<menu>Connection Button <font size="1.15em"><color rgb="b12146">›</color></font> Connect</menu>.**

Open the connection dialog to attach to a TCF agent. This launches a pop-up that temporarily hides other menus and dialogs.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**<menu>Connection Button <font size="1.15em"><color rgb="b12146">›</color></font> Disconnect</menu>**

Detach from the current TCF agent immediately.

**<menu>Connection Button <font size="1.15em"><color rgb="b12146">›</color></font> Launch TEMU</menu>**

Start the TEMU emulator instance if supported.

**<menu>Connection Button <font size="1.15em"><color rgb="b12146">›</color></font> Terminate TEMU</menu>**

Stop the emulator process.

The menu is accessible via mouse or keyboard and remains the same regardless of whether the bar is docked or floating.

### Tooltips, Icons & Shortcuts

Every button has a descriptive icon and a tooltip that appears on hover. Most actions also provide a keyboard shortcut displayed in the tooltip. Familiarity with these shortcuts can greatly speed up debugging.

### Mouse/Keyboard Interaction

- Click a button to invoke its action.
- Use the keyboard shortcuts for faster operation.
- The context drop-down responds to mouse clicks and keyboard navigation (arrow keys, type-ahead).
- The bar supports standard Qt drag-and-drop docking behavior and can be hidden or shown via the `View` menu.

### Behaviour During Context Discovery

When the debugger is refreshing available contexts, the context dropdown is disabled and a message informs the user. Execution buttons remain functional but will log a message if no valid context is selected.

### Guidelines & Tips

- Always select the correct context before issuing run or step commands. An invalid context will be ignored, and a message appears in the debug log.
- Reverse debugging requires that the target supports recording execution history (TEMU 5 does). Reverse Execution must also be enabled in TEMU before it can be used in the debugger. Use reverse controls carefully—they may take longer than forward steps.
- When connecting to remote targets, the connection pop-up will take focus; ensure other modal dialogs are closed first.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

#### Troubleshooting

- If the buttons appear greyed out, ensure that a context is selected and the TCF client is connected.

## 31.2.10. Register Inspection

This guide explains how to use the **Register Widget** inside the TEMU System Debugger.

#### Overview

The Registers panel shows CPU and peripheral register information for the quried debugging context. It displays groups of registers and individual fields in a single, scrollable, hierarchical table. You can:

- Browse register names and descriptions
- Read current values in hex
- Edit writable registers directly
- Expand/collapse groups to reveal fields
- See visual indicators for read-only vs writable items
- Observe automatic updates when execution state changes

The layout is optimized for keyboard and mouse, with a familiar spreadsheet-style interface.

#### Opening the Registers Panel

1. Connect to a target or start a debug session.
2. Choose <menu>ToolBar <font size="1.15em"><color rgb="b12146">›</color></font> Views <font size="1.15em"><color rgb="b12146">›</color></font> Registers</menu> (or click the register icon on the tab bar of the bottom dock).
3. From the component tree, use the context menu to select 'Get Register' for a component that may have registers.
4. The panel will then populate.

If no target is connected or the context has no register information, the panel will not populate anything.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

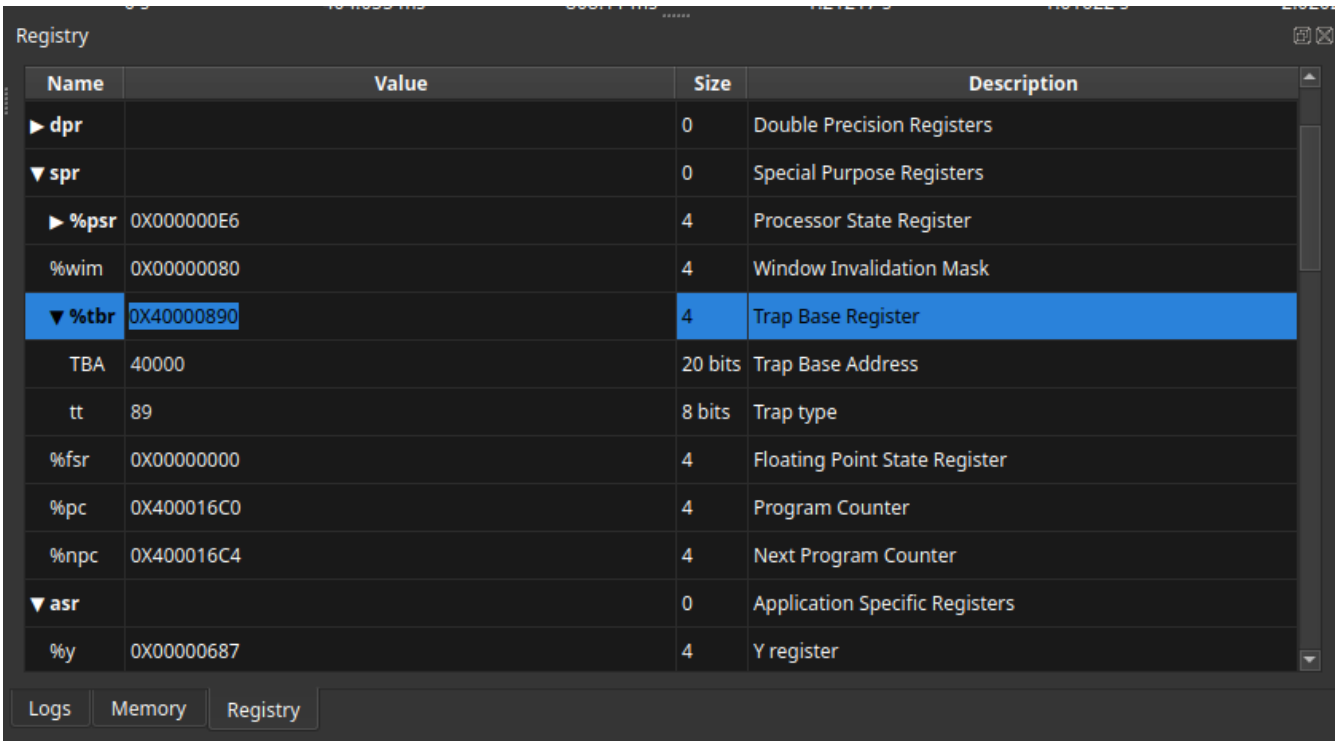



*Figure 25. CPU register view*

### Table Columns

The register view uses four columns:

**Name**
: the register or field name. Parent registers (those with children) are shown in bold and can be expanded or collapsed. Indentation indicates hierarchy level.

**Value**
: the current contents expressed in hexadecimal. Fields under registers update when you expand the parent or when the register value changes.

**Size**
: the size of the register or field (bytes or bits).

**Description**
: human-readable comments describing the purpose or semantics.

Each row is selectable. Clicking anywhere in the name column expands or collapses the item if it has children.

### Navigation and Interaction

#### Hierarchy Management

- Click the triangular indicator or name in the **Name** column to expand/collapse registers that group fields or sub-registers.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- Expanded parents show a downward arrow (▼); collapsed ones show a rightward arrow (□).
- Nested registers/fields are indented by four spaces per level to help visually parse the structure.

#### Value Updates

- When you expand a register or when the debugger halts or steps, the panel automatically queries the target and updates the **Value** column.
- Field values (children with type **Field**) are computed from the parent register value and displayed next to the parent.
- If the value cannot be obtained an error text will appear (for example, "Error fetching value").
- When register values change due to program execution, the panel reflects the new values within a moment.

#### Color Coding and Visual Cues

- Read-only registers/fields have a light gray background and darker text to indicate they cannot be edited.
- Writable values use the normal background so you can recognise editable entries quickly.
- Fields with all bits cleared appear in dark gray text; fields with all bits set turn orange; intermediate values use a lighter gray. This helps spot status flags and controls at a glance.
- Top-level items with no parents or children are styled differently: they are bold, larger, and centered across all columns.

#### Editing Register Values

1. Double-click the **Value** cell of a register that is writable (not read-only).
2. Type a new hexadecimal number. You may include the `0x` prefix or just the digits.
3. Press **Enter** or click outside the cell to commit, or **Esc** to cancel.

If the field is not writable, an attempt to edit will be rejected and a warning dialog will appear: "Register '<name>' is not writable".

Invalid input (non-hex digits) is automatically reverted to the previous value. Upon successful entry, any child fields are recalculated and the debugger is notified of the new register value. The register will then be read back from the target to confirm the update.

### Error Handling

- If the widget receives malformed data from the backend, a descriptive message such as "Error: Unexpected data format for registers." is shown.
- Connectivity issues or missing services may prevent values from loading; in that case, individual cells will display an error message.

### Tips and Best Practices

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- Expand a register before editing to see how its fields will be affected by a new value.
- Use the **Size** and **Description** columns to verify you are modifying the correct register, especially in CPUs with large register sets.
- Collapsing a parent hides all its children, which can help declutter the view while stepping through code.
- You can navigate the table with arrow keys once a row is selected. Press **Tab** to move between columns if you prefer keyboard editing.

#### Accessibility and Keyboard Support

- Standard Qt table keyboard navigation applies: arrows for movement, **CTRL** + **Home**/**End** to jump to the start or end of the table.
- Selection follows the row; clicking anywhere on a row selects the entire row.

#### Closing the Panel

The view can be reopened at any time; its state (expanded/collapsed registers) is transient and will reset when you reconnect or query another context.

### 31.2.11. Inspector View

The Inspector is a powerful multi-purpose panel for examining symbols, memory regions, and timeline events within the TEMU System Debugger. It provides a unified interface for deep inspection of program state, type inference from raw memory, and event timeline analysis. The Inspector organizes inspections into three categories with support for pinning multiple items simultaneously, allowing you to maintain several inspection contexts while debugging.

#### Overview

The Inspector view presents three distinct categories of inspection data:

**Global Symbols (SYM)**

Inspect global symbols by name and context. The Inspector reads the symbol’s type definition and displays all fields, nested structures, arrays, and primitive values in a hierarchical tree. You can search fields, toggle between hex and decimal formats, and dereference pointers to inspect referenced memory.

**Memory Inference (MEM)**

Interpret raw memory at any address as a typed structure. When you know the type of data stored at a memory address but the debugger has no symbol for it, you can "assume" a type and view the bytes formatted according to that type definition. This is particularly useful for inspecting dynamically allocated structures, memory-mapped hardware registers, or serialized data.

**Timeline Events (EVT)**

Examine individual timeline events or event aggregates. Click on a timeline event to view its

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

metadata, duration, kind, lane, and associated properties. Aggregate mode displays a range of events and provides quick navigation back to the timeline.

The Inspector dock supports multiple pinned entries per category. Each entry appears as a tab within its category card, preserving your inspection history as you navigate through the debugging session.

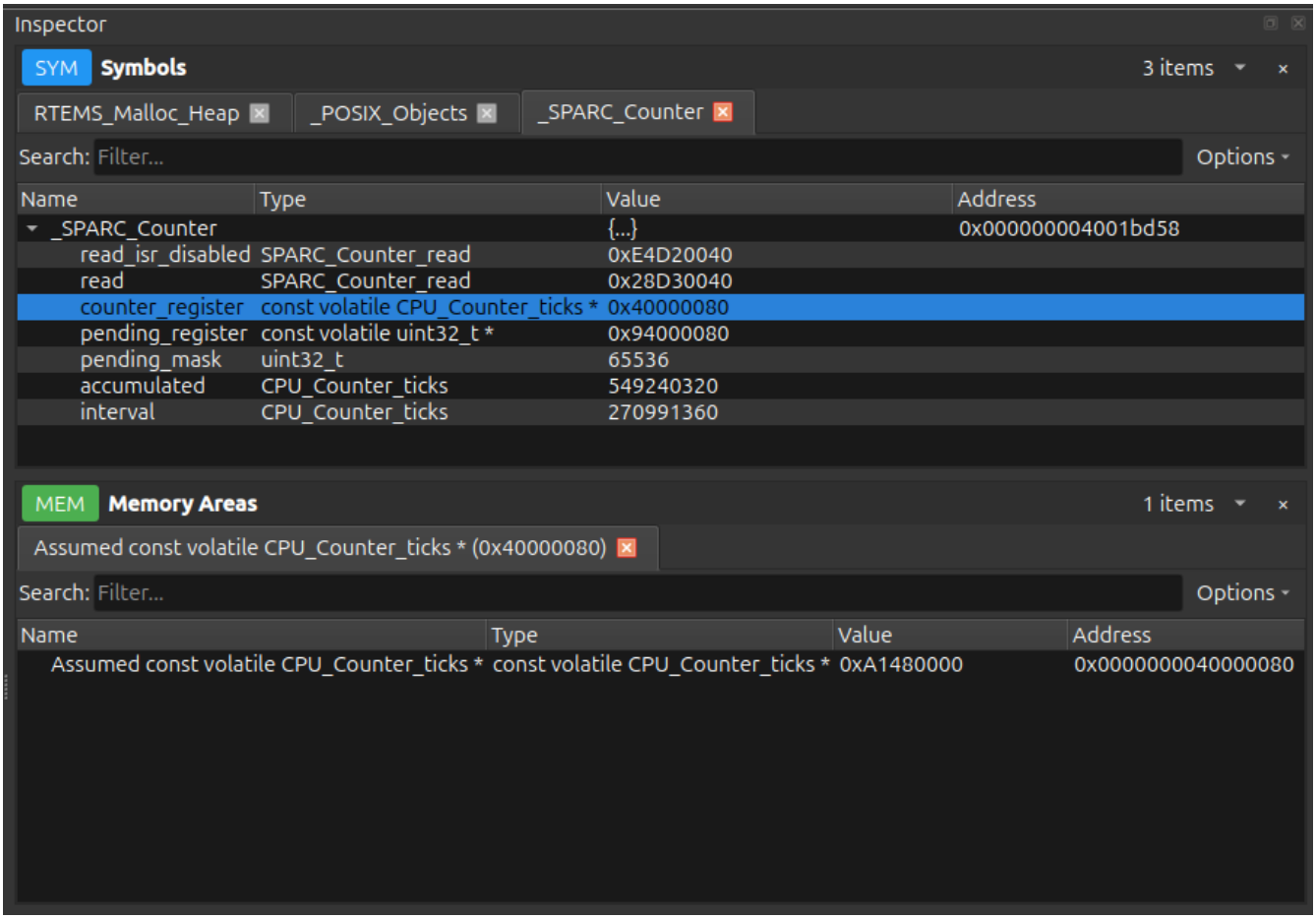



*Figure 26. Inspector view showing Symbol and Memory inspection categories*

### Opening the Inspector

The Inspector dock is available from the main menu or via a dock tab button. When first opened, the view displays "No items pinned." To populate the Inspector, trigger an inspection action from one of the following locations:

- Right-click a symbol leaf in the Component Browser and select **Inspect Symbol**
- Right-click memory bytes in the Memory View and choose **Inspect Memory** or **Assume Type...**
- Click an event in the Timeline view to inspect its metadata
- Right-click a pointer variable in the Variables panel and select **Dereference**

Each inspection request creates or updates a tab in the appropriate category.

### Inspector Layout

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

The Inspector is divided into three category cards displayed vertically:

- **SYM** – Global Symbol inspections
- **MEM** – Memory Inference inspections
- **EVT** – Timeline Event inspections

Each category card is collapsible and contains a tab widget for managing multiple pinned items. The category badge uses color coding for quick identification:

#### Tab Widget

Tabs within a category represent individual inspection entries. Tab titles show the symbol name, memory address, or event identifier. You can switch between tabs to view different inspections without losing data. Each tab displays:

- Header information (symbol name, type, context, or address)
- Status indicator (Loading, Populated, or Error)
- Tree widget or event metadata panel

### Global Symbol Inspection (SYM Category)

Global symbol inspection displays the complete type structure and current values for a selected symbol.

#### Triggering Symbol Inspection

1. Open the **Component Browser** dock.
2. Expand the tree to reveal the **Symbols** container under a software context.
3. Continue expanding until you reach a symbol leaf node.
4. Right-click the symbol leaf.
5. Select **Inspect Symbol** from the context menu.

The Inspector will create or update a tab in the SYM category card with the symbol's name as the tab title.

#### Symbol Tree Columns

The symbol inspection tree uses four columns:

**Name**

The symbol, field, or array element name. Nested fields are indented to show hierarchy. Composite structures display `\{⋯}` and can be expanded. Arrays display `[n]` where n is the element count.

**Type**

The data type as reported by the symbol service.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**Value**

The current value formatted according to the type. Primitive types show numeric values or characters. Pointers show hexadecimal addresses. Strings embedded in char arrays are displayed with quotes.

**Address**

The memory address where the field or symbol resides. Only populated for symbols with location information.

#### Pointer Dereferencing

Pointer fields in the symbol tree can be dereferenced to inspect the referenced memory:

1. Locate a pointer field in the tree (identified by the pointer type, e.g., `int*`).
2. Right-click the pointer row.
3. Select **Dereference Pointer** from the context menu.

The Inspector will query the memory at the pointer's address and attempt to interpret it according to the pointer's base type. A new tab appears in the MEM category showing the dereferenced structure.



Pointer dereferencing requires valid memory access and symbol type information. If the pointer is null or points to unmapped memory, an error message is displayed.

#### Search and Filtering

The Inspector tree includes a search field at the top of the tab content. Type any substring to filter visible rows. The tree retains parent nodes if any child matches the search term, making it easy to locate deeply nested fields.

Search is case-insensitive and matches against the Name column.

#### Format Toggle (Hex/Decimal)

Use the options button (gear icon) in the toolbar to toggle between hexadecimal and decimal value display.

- **Hex**: Values display as `0x···` hexadecimal
- **Decimal**: Values display as base-10 integers

The format preference applies to all numeric fields in the tree.

### Memory Inference (MEM Category)

Memory inference interprets raw bytes at a given address as a specific type, even when no symbol information exists for that address.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

#### Assume Type Dialog

When you select **Assume Type...**, a dialog appears allowing you to search for and select a type:

1. Enter a search term in the **Search Type** field (e.g., `struct`, `int`, `float`, or a partial type name).
2. Press **Enter** or wait briefly for the type list to populate.
3. Select the desired type from the list.
4. Click **OK** to apply the type interpretation.

The dialog fetches types from the debugger's symbol service based on the active software context. Common types like `int`, `float`, `char`, and `void` are available, along with all user-defined structures, unions, and typedefs.

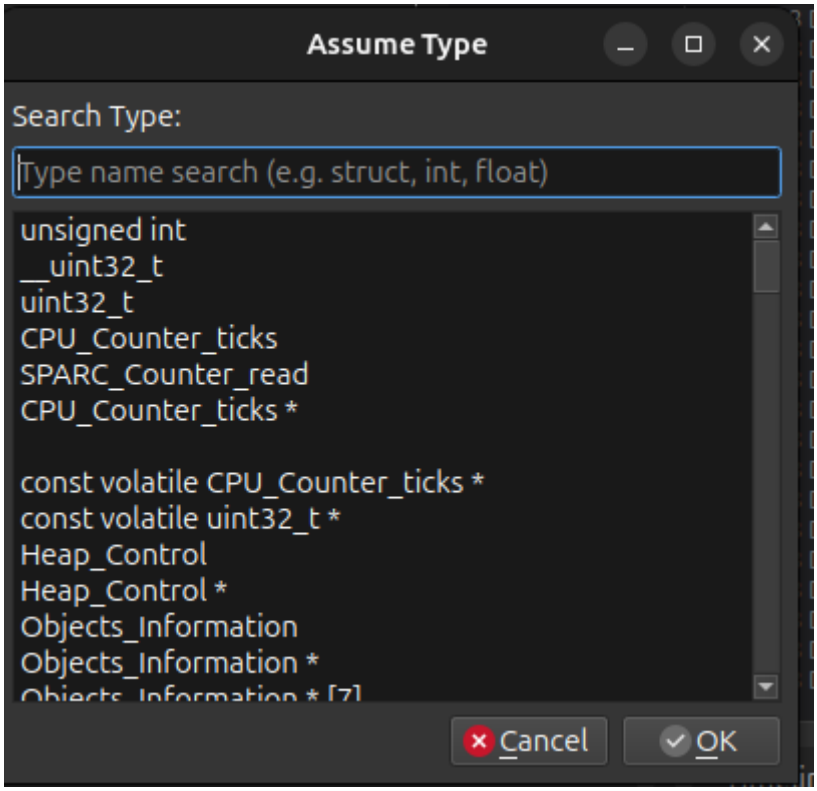



*Figure 27. Assume Type dialog for selecting a structure type*

#### Memory Inspection Display

Once a type is selected, the Inspector creates a new tab in the MEM category. The tab title shows the memory address in hex format (e.g., `0x10004000@contextID`).

The tree structure is identical to symbol inspection.

The Inspector reads raw bytes from the target memory and interprets them according to the type definition, taking into account endianness and alignment.

#### Endianness Handling

Memory inference respects the target context's endianness setting. Multi-byte values are interpreted as big-endian or little-endian according to the context properties.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

If you dereference a pointer from a symbol or memory tree, the endianness is inherited from the parent inspection context.

### Timeline Event Inspection (EVT Category)

Timeline event inspection displays metadata and properties for individual events or event aggregates from the Timeline view.

#### Triggering Event Inspection

Click any event rectangle in the Timeline view to inspect it. The Inspector creates a tab in the EVT category with the event's unique identifier as the tab title.

For event aggregates (multiple events within a time range), click the aggregate indicator to open a grouped inspection showing all events in the range.

#### Event Metadata Display

Below the metadata, a tree or list displays all key-value properties associated with the event. Properties vary by event type and may include function names, return values, CPU context IDs, and other debugging information.

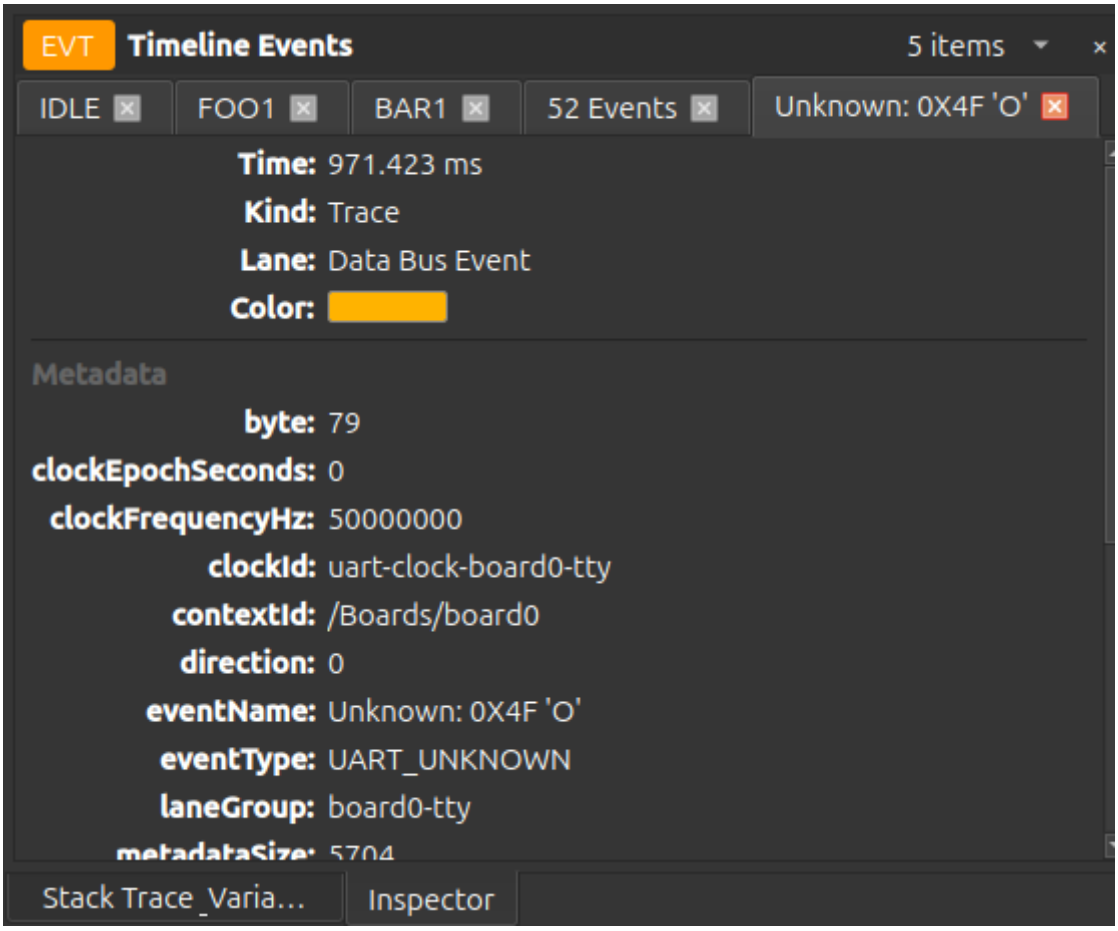



*Figure 28. Inspecting a timeline event showing metadata and properties*

#### Event Selection and Navigation

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

Clicking an event in the Inspector timeline tab highlights it in the Timeline view. Use the **Zoom to Range** button to adjust the timeline zoom level to focus on the selected event.

#### Aggregate Event Display

When an aggregate is inspected, the EVT tab lists all events within the selected time range.

Click any row to navigate to that individual event in the timeline.

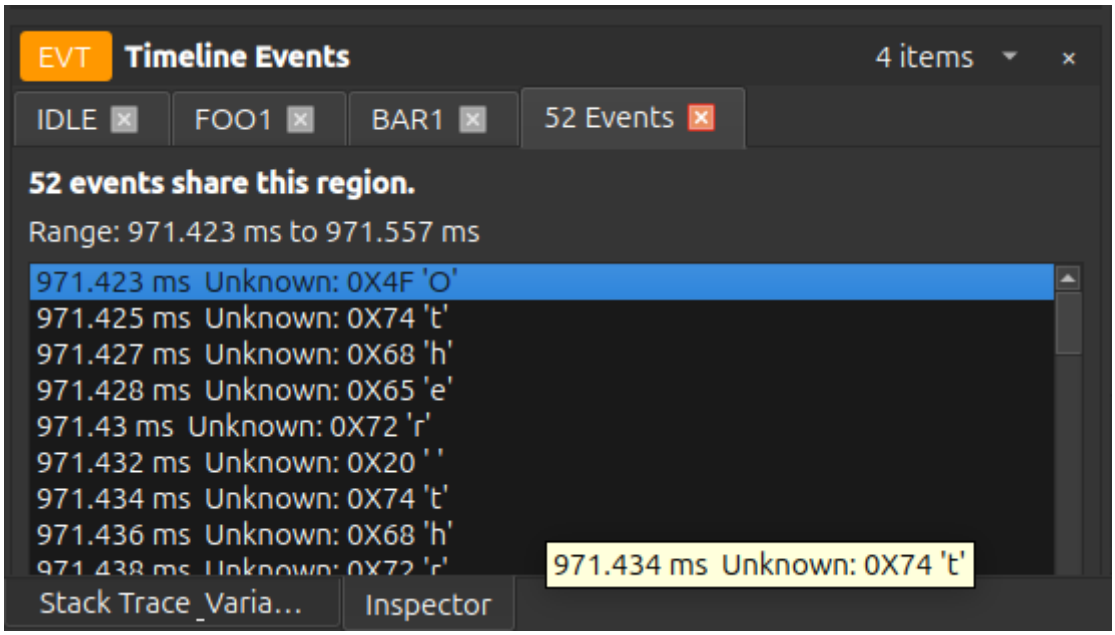



*Figure 29. Inspecting multiple timeline events in an aggregate view*

### Error Handling

If an inspection fails, the Inspector displays an error message in the tab content area:

- **Symbol not found**: The requested symbol does not exist in the current context.
- **Memory read failed**: The memory address is invalid or inaccessible.
- **Type resolution failed**: The assumed type could not be loaded from the symbol service.
- **Context unavailable**: The target context disconnected or is no longer valid.

Errors are shown in red text with a brief description. You can unpin the failed tab and retry after correcting the issue.

### Keyboard Shortcuts

The Inspector supports standard keyboard navigation:

`Ctrl` + `F`
: Focus the search field.

`Up` / `Down`
: Navigate tree rows.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

`Left` / `Right`

Collapse or expand tree nodes.

`Ctrl` + `C`

Copy the selected value.

#### Integration with Other Views

The Inspector works with other debugger components:

**Component Browser**

Symbol inspection requests originate from symbol leaves in the Component Browser tree.

**Memory View**

Memory inspection and Assume Type actions trigger memory inference tabs in the Inspector. Selecting bytes in the Memory View and choosing an inspection action automatically populates the MEM category.

**Timeline**

Clicking timeline events opens the EVT category with event metadata. You can zoom back to the timeline from the Inspector, creating a bidirectional workflow.

**Variables Panel**

Pointer variables can be dereferenced. The Inspector opens a MEM tab showing the pointed-to memory symbol.

**Stack Trace**

When debugging with call stack information, the Inspector can display symbols in the current stack frame context.

#### Troubleshooting

**Memory inference shows incorrect values:** Verify the assumed type matches the actual data layout. Check the endianness setting for the memory context. Ensure the memory address is aligned according to the type's alignment requirements.

**Pointer dereference fails with "Memory read failed":** The pointer may be null, uninitialized, or point to unmapped memory. Use the Memory View to verify the pointer's value before dereferencing.

**Assume Type dialog is empty:** The symbol service may not have loaded types for the current context. Ensure you have connected to a target with debug symbols and the software context is active.

### 31.2.12. Stack Trace & Variables

**Stack Trace and Variables** panel in the TEMU System Debugger gives the user an insight into the

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

current state of the call stack and stack frames as well variable views. A stack trace may be gotten for each suspendable context.

### Layout

When the debugger is running, the panel is normally placed on the right- hand side of the main window. It is divided vertically into two linked views by a movable splitter:

**Stack trace**
: the upper half shows the call stack for the current thread.

**Variables**
: the lower half displays the contents of local variables and parameters for whatever stack frame is selected above.

You can drag the splitter bar up or down to give more space to either view.

### Stack Trace Viewer

The stack trace view is a simple table with three columns:

**Stack Frame**
: the function name or, if the name is unavailable, a placeholder such as "<unknown>".

**IP**
: the current instruction pointer for that frame (shown in hexadecimal).

**FP**
: the stack frame pointer (also hexadecimal).

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

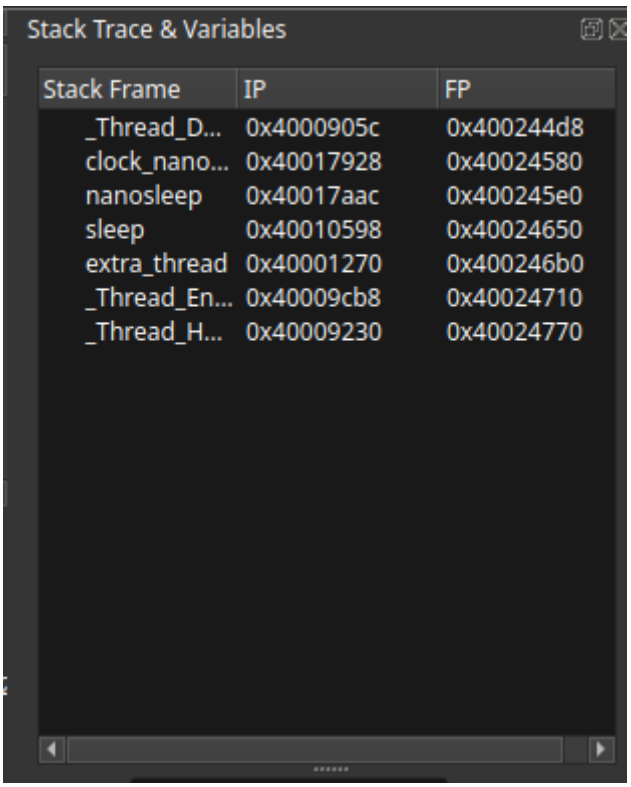



*Figure 30. A Stack Trace with callstack listed*

#### Selecting a Frame

Click on any row to make it the active frame. As soon as a frame is selected, the variables view beneath will refresh to show the locals and arguments that were in scope at that point in the execution. The debugger automatically keeps the two panes in sync.

#### Context Menu

Right-clicking a frame opens a small menu with the following action:

- **Disassemble Function** – request that the disassembly view show the machine code for the function associated with that frame. This is especially useful when debugging at the assembly level. If the frame does not have symbol information, the menu item will be disabled.

The context menu only appears when you click on a non-empty row. Use it after the program has paused (for example, on a breakpoint or exception).

### Variables Viewer

The variables view shows a hierarchical tree of the variables that are in scope for the currently selected frame. Each row has three columns:

**Variable**

The name of the variable or structure field.

**Value**

The current value. This field can show primitive types, strings, pointer addresses, array contents,

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

etc. Long values are truncated with a tooltip that appears when you hover the mouse over the cell.

**Type**

The data type as reported by the debugger's symbol information.

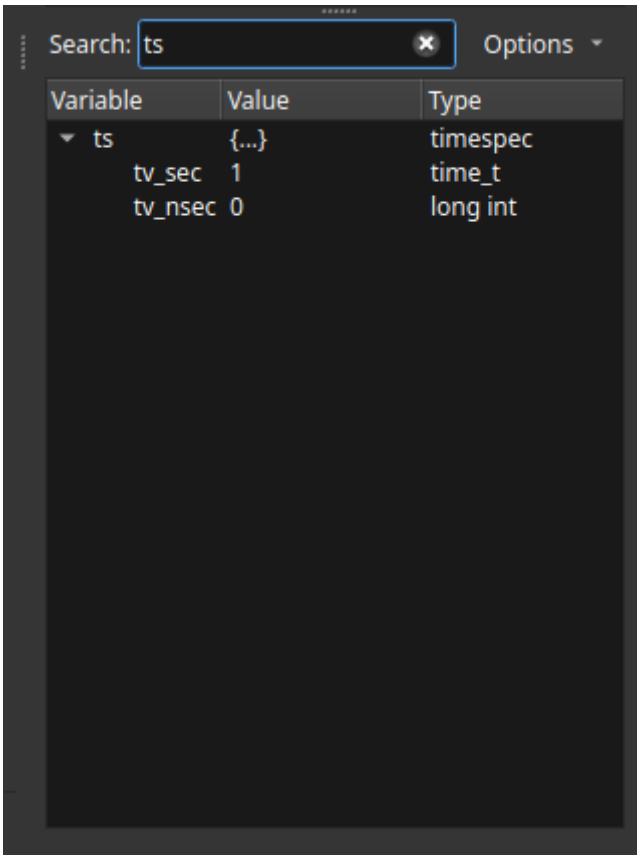



*Figure 31. A Composite struct in the variable view*

#### Expanding Nodes

Composite types (structures, classes, arrays, etc.) are shown with a small triangle that you can click to expand and collapse. Expanding a pointer will trigger the debugger to fetch the pointed-to data and show it as child entries.

#### Searching and Filtering

At the top of the variables pane is a **Search:** field. Type any substring to filter the tree; only variables whose names match the search text (case-insensitive) will remain visible. Parent entries are also shown if any of their children match, which makes it easy to find a field buried deep inside a structure.

The search field has a clear button to quickly remove the filter and show everything again.

#### Display Options

Next to the search field is an **Options** button. Clicking it reveals a menu with two mutually exclusive settings:

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- **Decimal** – display numeric values in the usual base-10 form (the default).
- **Hex** – show numbers in hexadecimal. This affects all values in the table, including addresses, and is often more convenient when working with low-level code or memory dumps.

Changing the option updates the values immediately.

#### Context Menu

Right-click anywhere in the variables tree to open a context menu with the following actions:

- **Clear Variables** – empties the view. This is helpful when you want to remove stale data before stepping the target or switching threads.
- **Dereference to Memory Inspector** – available when right-clicking a pointer variable. The Inspector view opens with the memory at the pointer's address interpreted. See Section 31.2.11 for more details on memory type inference.

There is no "copy" action; to copy a value simply select the cell and use the normal keyboard shortcuts for your platform.

#### Pointer Handling

When the debugger knows that a value is a pointer, it automatically displays the address in hexadecimal and, if the pointed-to type is available, attempts to read the memory at that address. If the read succeeds, the pointer entry will be annotated so that you can expand it and see the dereferenced fields. This happens transparently, without any extra clicks.

If the debugger is unable to dereference a pointer (for example, if the address is invalid or the type is unknown), the value column will show an error message such as "<error>".

#### Tooltips

Hovering the mouse over a value cell will show the full text of the value in a tooltip. This is particularly useful when values are truncated to fit in the column.

### Thread Changes and Clearing

If the debugger switches threads (for example, when stepping or when a thread becomes active), the stack trace view will be updated automatically for the new thread. The variables pane is cleared and then repopulated when you click a frame.

### Tips

- Resize the splitter to give more room to the pane you are currently interested in.
- Use the search field to quickly locate a variable when working with large data structures.
- Switch between hex and decimal displays depending on whether you are looking at addresses/masks or human-readable numbers.
- Right-click a frame and choose **Disassemble Function** to jump to the code for that stack frame

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

in the disassembly view.

## 31.2.13. CTF & Timelines

Make the most of the CTF tracing facilities and associated timeline views in the TEMU System Debugger.

### Overview

The debugger can connect to a running TEMU instance using the TCF protocol and request Common Trace Format (CTF) data from software contexts. When tracing information is available it is presented in one or more timeline panels that allow you to see events, thread activity, variable changes and more. The timeline subsystem is designed to be flexible: you can zoom, pan, detach views, follow the latest data, and combine multiple series in a single container.

### Getting Started with CTF Tracing

*Enabling polling*

- In the main toolbar and under the **CTF** item, use the \[Start Polling] and \[Pause Polling] toolbar buttons to control automatic fetching of CTF data. Polling will continue in the background until you pause it or close the debugger.
- The drop-down list next to the buttons shows available software contexts; if none are selected the first context will be used.
- The bottom status bar displays **Simulated Real Time** and other time messages as data arrives.



Variable sampling must be set in TEMU by creating a Trace Sampler model and connect it to the system timesource such as a cpu. Next attach properties of the models which will be displayed in series.

```
TraceSampler.new name=sam
connect-timesource obj=sam ts=l2-cpu
sam.add prop=wave.output
sam.add prop=wavier.output
sam.add prop=wavierer.output
sam.add prop=l2-cpu.pc
```

### Timeline Container and Docking

The **Timeline** area is implemented as a dockable panel that you can move around, tile with other docks, or turn into a separate window. It is made up of a `TimelineContainer` which hosts one or more individual timelines.

- Tabs appear along the top when multiple timelines are present. Click a tab to switch between them.
- Right-click a tab to open a context menu with the **Pull Out Timeline** action. This will detach the timeline into its own floating window.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- Detached timelines remain linked to the container and will reattach when you close the floating window (unless you tagged them as transient).

#### Arranging Timelines

- Drag the title bar of a dock to reposition it. Use the **View** menu to enable animated docks, nested docks, or tabbed docks.
- You can have multiple timelines side-by-side or stacked vertically by dragging dock borders.

### Working with a Single Timeline

Each timeline represents a sequence of events over time. It may contain multiple series (for example, variable values) or a single stream such as context-switch events.

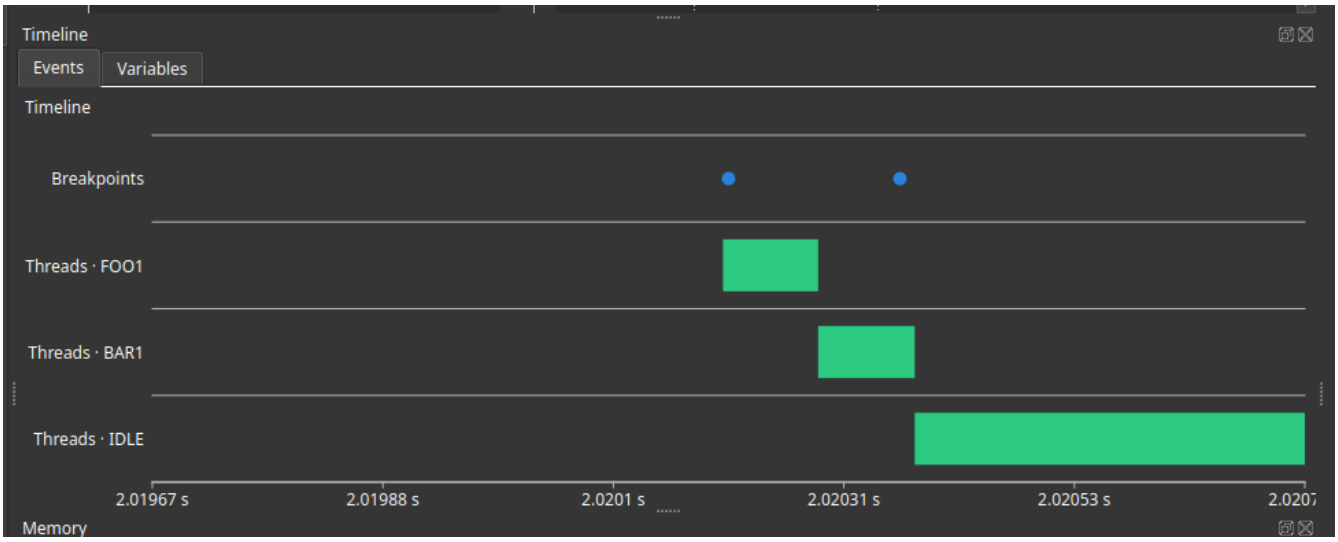



*Figure 32. Event Timeline with context switches and breakpoint events*

#### Time Range and Current Time

- The horizontal axis shows the current time range. You can set the span by dragging with the middle-mouse button or using the zoom controls.
- A vertical marker indicates the current time. It moves automatically when **Follow Latest** is enabled (see below) or you can click on the timeline to move it manually.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

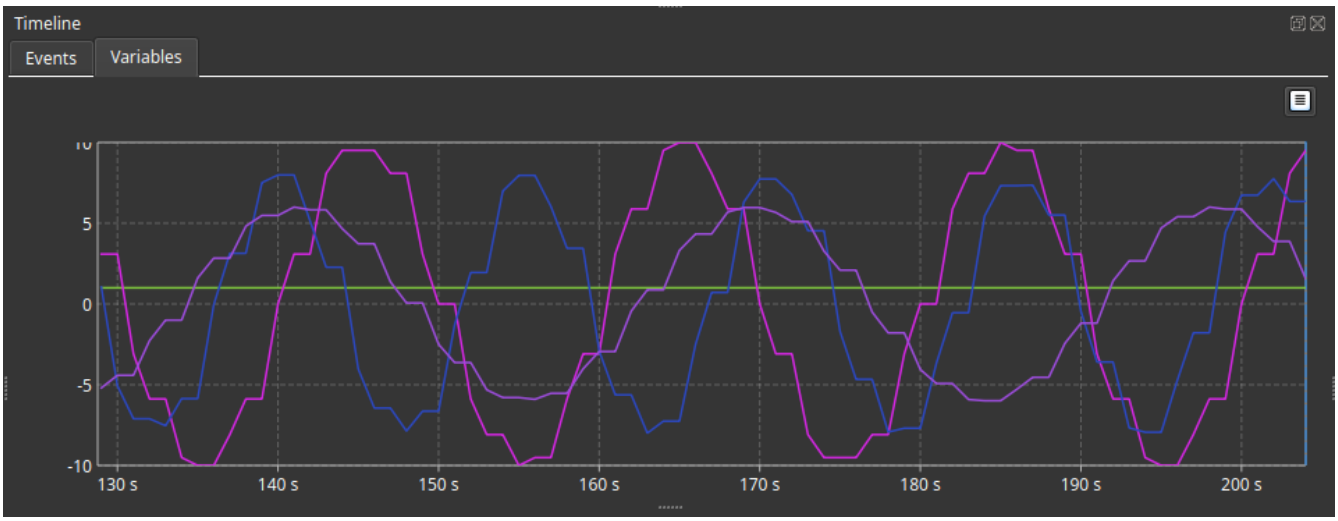



*Figure 33. Variable Timeline with with 4 variable series set*

#### Zooming and Panning

### Mouse Wheel

rotate to zoom in (forward) or out (backward).

### Middle-Button Drag

click and hold the middle mouse button, then move left or right to pan the view.

#### Follow Latest

Right-click anywhere on a timeline and choose **Follow Latest** from the context menu, or toggle the button in the timeline control bar. When enabled the view will automatically scroll so that the newest data points are always visible. This is especially handy during live polling.

#### Detaching Series

Some timelines (such as variable traces) may contain multiple named series within a single view. A context-menu or toolbar action allows you to "pull out" individual series into their own timeline so you can compare or arrange them independently.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

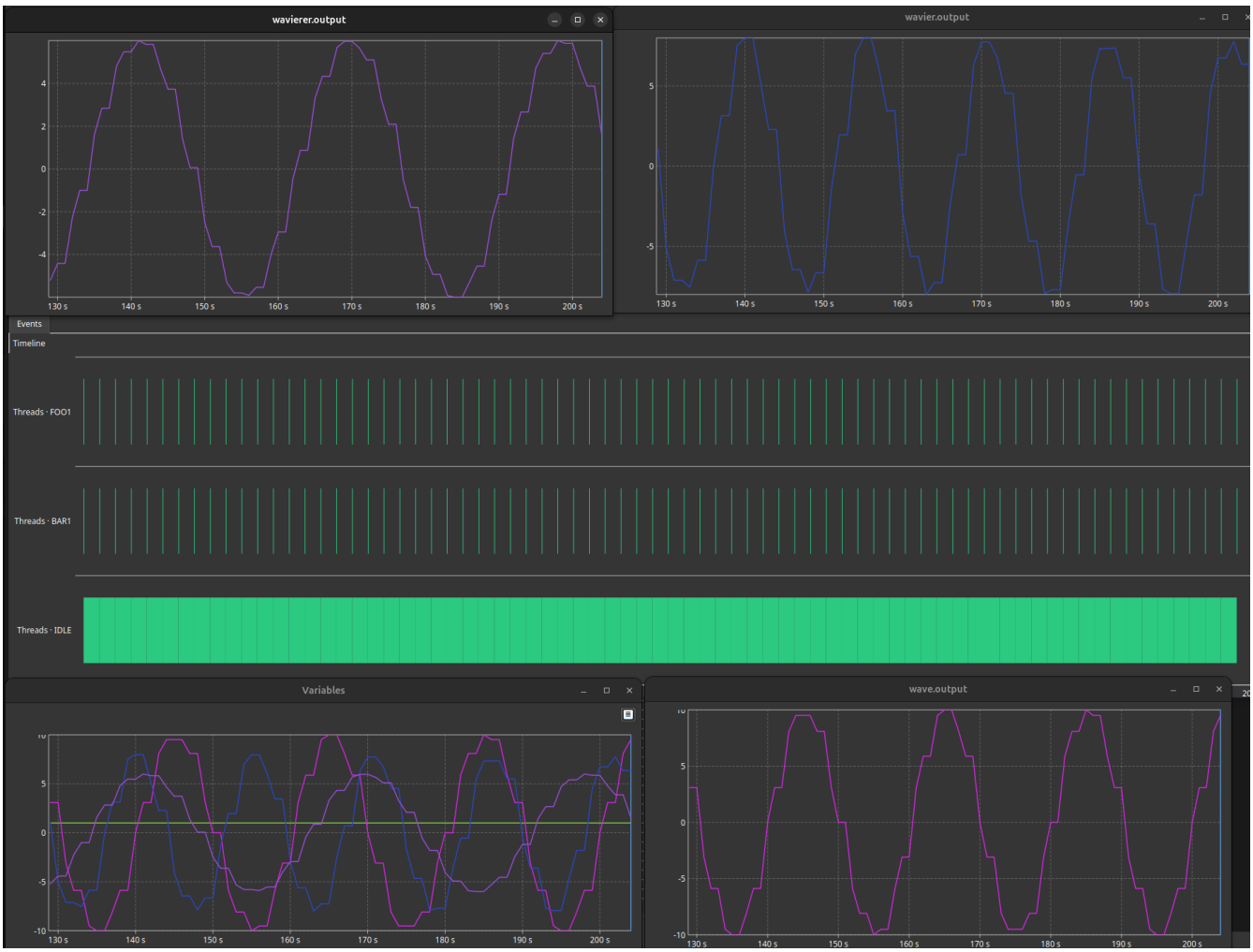



*Figure 34. Variable series poped out from the main variable timeline view*

### Contexts and Events

The debugger displays the context identifier for the trace in the timeline header. Events are translated from raw CTF records into human-readable entries such as thread switches, function calls, or variable updates.

- Thread switch events are automatically highlighted and can optionally display the previous and next thread names.
- You can inspect the fields of any event by hovering or using a dedicated event list (depending on the timeline type).

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

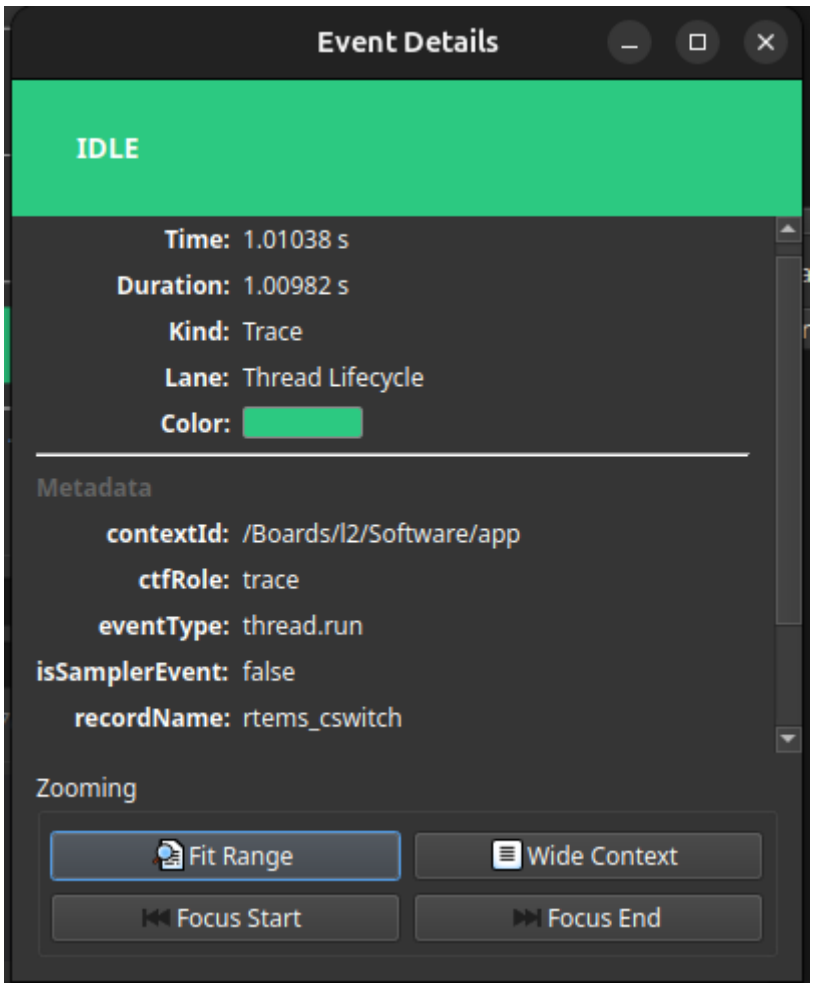



*Figure 35. Popout with thread information related to the clicked event*

### Breakpoint Integration

When you set breakpoints and run the debugger, the timeline controller may receive timestamps for each hit. These times appear as markers on suitable timelines, allowing you to correlate trace data with breakpoint activity.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

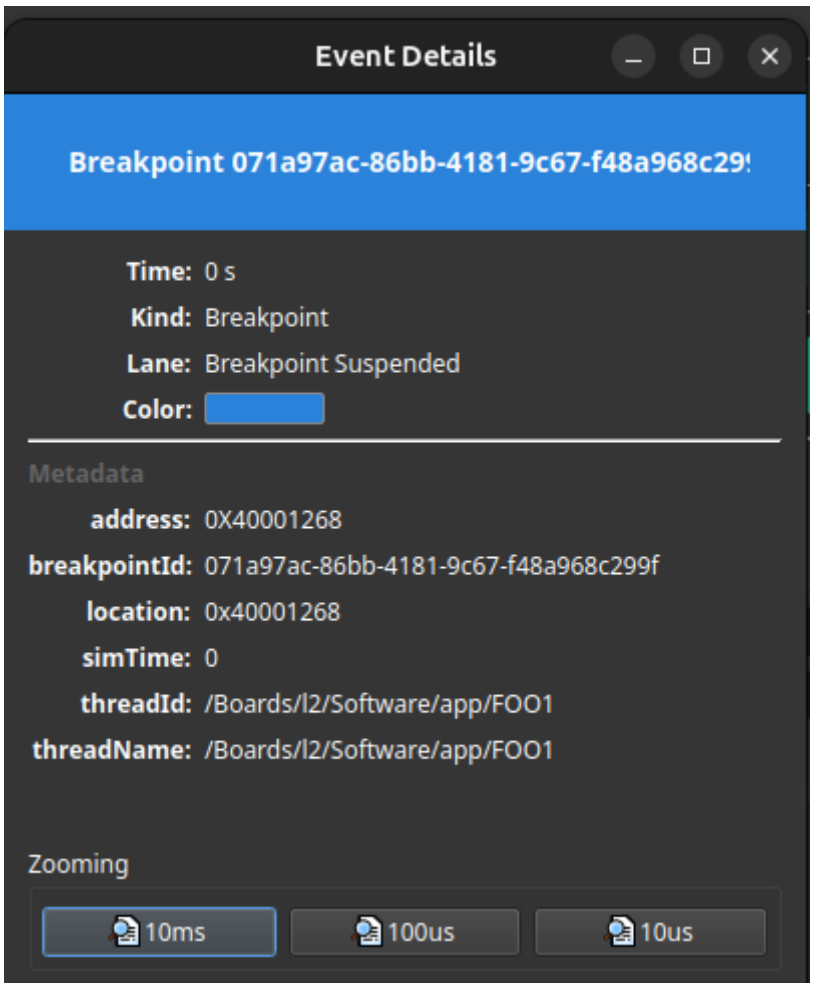



*Figure 36. Popout with breakpoint information related to the clicked event*

### Tips & Tricks

- Use the **View** → **Timeline** menu or toolbar to show/hide the timeline dock on demand.
- When the timeline window becomes crowded, detach some views and resize them independently.
- Combine CTF data with logging and source views by arranging docks side by side – you can drag the timeline dock next to the log or source editor.

### Troubleshooting

- If no CTF contexts appear, ensure the TEMU instance was started with tracing enabled and that the TCF connection is active (check the status bar). Reconnect or restart polling if necessary.
- If a detached timeline window disappears unexpectedly, look for a checkbox in its title bar labeled **Keep on Close** – disabling it will make the timeline transient.

### Keyboard Shortcuts

- `Space` – Open Variable timeline filter options.
- `Alt` + `←/→` – switch between timeline tabs (if numbered).
- `Ctrl` + `Wheel` – zoom centered around cursor (if supported).

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

## Appendix

### CTF Contexts

A "context" typically corresponds to a running software process or thread within TEMU. You may see several contexts listed if the system under test has multiple processors or execution environments.

### Timeline Types

The system supports different kinds of timeline widgets:

**Variable Timelines**
: Shows numeric values over time and can be plotted on separate scales.

**Event Timelines**
: Displays discrete events as markers or colored bands.

**Custom Timelines**
: May be provided by plugins or future extensions.

## 31.2.14. TEMU CLI Terminal

The debugger includes an interactive TEMU command terminal for running emulator commands without leaving the GUI. The dock is labelled **Terminal**.

### Opening the Terminal

The Terminal dock is placed in the bottom dock area by default and is tabbed with Memory, Registers, and Object Properties.

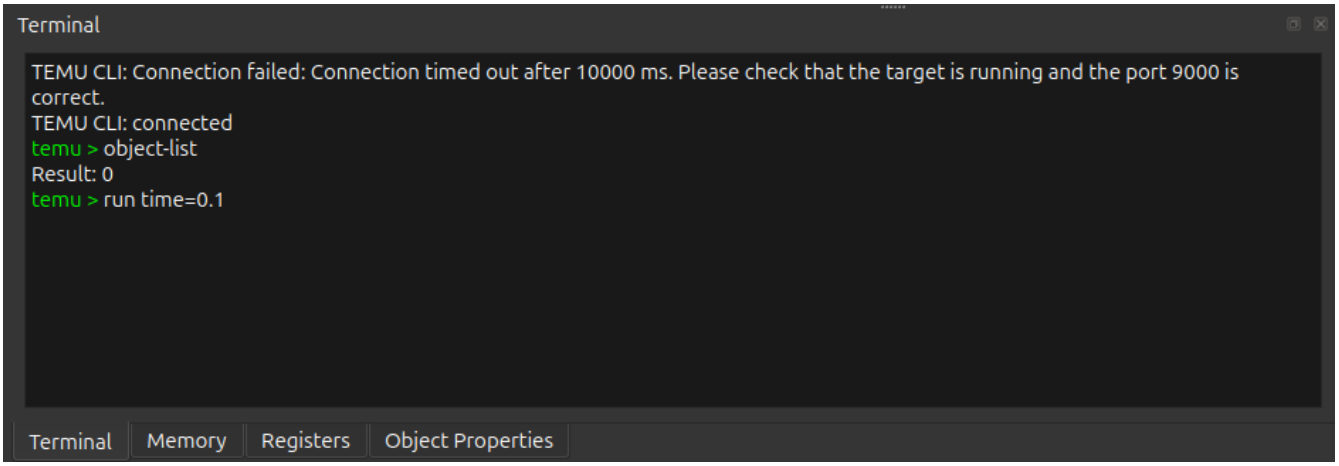



*Figure 37. TEMU CLI terminal dock*

### Connection State

The terminal is only interactive when the TEMU service is available.

- Before a connection is ready, the dock shows service-status messages and input is disabled.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

- After a successful connection, the terminal prints a connected message and shows the prompt `temu >`.
- On disconnect or connection failure, the terminal prints the reason and disables further input until the service becomes available again.

### Running Commands

Type a command after the prompt and press `Enter`. The debugger sends the command to the TEMU service and appends the result to the terminal output before printing a new prompt.

The prompt color reflects the last outcome:

- green after a successful command result,
- red after an error or failing result.

A current limitation of this Terminal is that results are not streamed to this terminal but will appear in the real emulators terminal.

The `run` command, with no `time=*`, will continue running the software indefinitely and can be stopped using the command `stop`.

### Completion and Editing

The terminal supports interactive completion:

- Typing starts a delayed completion request.
- Press `Tab` to request completion immediately.
- When the completion popup is visible, use the arrow keys to move through the suggestions and press `Enter` to accept one.
- Press `Esc` to close the completion popup.

The widget protects the prompt area, so editing is limited to the current input line after `temu >`.

### Command History

The terminal keeps a local command history for the current session.

- Press `Up` to move to an older command.
- Press `Down` to move toward newer commands or back to the unfinished draft you were typing.

This is useful when repeating the same inspection command with only minor changes.

### Tips

- Use the terminal for quick one-off TEMU commands while keeping the graphical views open beside it.
- If you rely on scripted breakpoint actions, see Scripting for the emulator-side scripting model.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

# 31.3. Customization

## 31.3.1. Customizing the Layout

The debugger uses Qt dock widgets throughout the interface, so you can reshape the workspace without leaving the main window.

### Moving, Docking, and Floating Panels

Drag a dock by its title bar to move it to another edge of the window. While dragging, the window shows dock targets that indicate where the panel will land.

- Drop onto an edge to dock the panel there.
- Drop into the center of another dock area to tab the panel with the existing one.
- Drag a dock away from the main window to float it in its own window.
- Drag a floating dock back onto the main window to re-dock it.

### Working with Tabbed and Nested Docks

Tabbed docks are the default way to group related tools such as:

- Components, Breakpoints / Watchpoints, Complex Breakpoints, and Project Explorer
- Terminal, Memory, Registers, and Object Properties
- Logs and Timeline

If nested docking is enabled, you can also split a dock area into multiple rows or columns instead of keeping everything in a single tab stack.

### Main Window Options

The Main window menu controls how dock widgets behave globally:

**Animated docks**
: Turns the docking animation on or off while moving panels.

**Allow nested docks**
: Lets one dock area contain multiple side-by-side dock groups.

**Allow tabbed docks**
: Allows docks to be combined into tabs.

**Force tabbed docks**
: Forces dropped docks to join an existing tab group instead of creating a side-by-side split.

**Vertical tabs**
: Places dock tabs vertically where Qt supports it.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.

**Grouped dragging**

Moves an entire tab group together instead of only the selected dock.

#### Dock Corner Assignments

The Dock Widgets menu lets you choose which dock area owns each corner of the main window:

- <menu>Dock Widgets <font size="1.15em"><color rgb="b12146">›</color></font> Top left corner</menu>
- <menu>Dock Widgets <font size="1.15em"><color rgb="b12146">›</color></font> Top right corner</menu>
- <menu>Dock Widgets <font size="1.15em"><color rgb="b12146">›</color></font> Bottom left corner</menu>
- <menu>Dock Widgets <font size="1.15em"><color rgb="b12146">›</color></font> Bottom right corner</menu>

Each submenu offers the two relevant dock areas for that corner. This is useful when you want the left or right dock stacks to extend fully into the lower corners instead of yielding that space to the bottom docks.

#### Refresh Buttons on Custom Dock Title Bars

Some docks use a custom title bar with a refresh button. This is especially useful for the Component Browser and Object Properties panels, where a refresh updates the visible runtime structure after the target changes.

#### Re-open a Removed Dock

Docks that have been closed can be re-opened by right clicking on an empty area. This will produce a list of all views with checkboxes attached. If a dock had been removed it’s mark will be un-checked, to open the dock again, check the box and it will appear in it’s previous state.

## 31.3.2. Troubleshooting

Common issues and solutions for the TEMU System Debugger GUI.

#### Connection Issues

- TCF connection problems: Check server status, port, and firewall settings.

#### Logs View

- Use the logging panel for diagnostics and error messages.

#### General Tips

- Restart the debugger if issues persist.
- Consult the documentation for further help.

The use and/or disclosure,etc. of the contents of this document (or any part thereof) is subject to the restrictions referenced on the front page.