TEMU

User Manual

Version latest, 2024-05-14

Table of Contents

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.

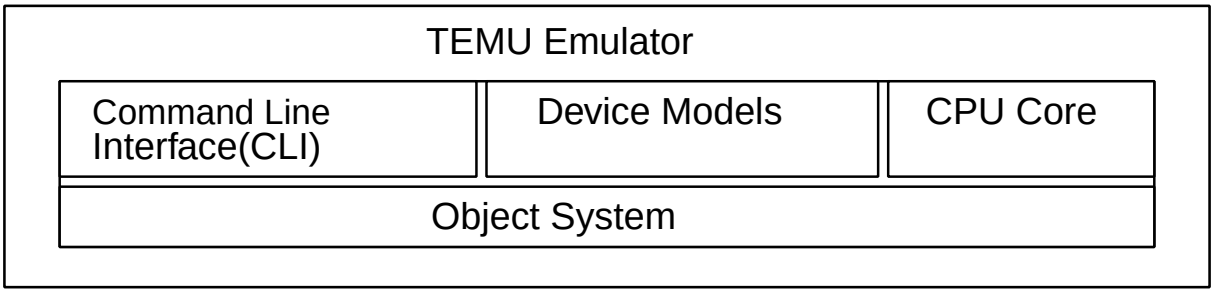


Figure 1. Layers of the Terma Emulator

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.

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

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

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.

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/latest`. 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/latest/lib` and the tools in `/opt/temu/latest/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:

`/opt/temu/latest+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. License Files

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

4.3.1. Making Sure the TEMU Group is Available

If the setup command was just executed, it is likely that license installation detailed in Section 4.3.2 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.

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

4.3.2. 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 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).

4.3.3. 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)?
```

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. Running the Emulator

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

4.5. Creating a New Machine

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

To create a new machine, it is possible to use one of the bundled CPU configurations in `/opt/temu/latest/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 1. Create a LEON2 System

```
temu> exec leon2.temu
```

Listing 2. Create a Dual Core LEON3 System

```
temu> exec leon3-dual-core.temu
```

4.6. 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 run and step commands can run and step both machines, clocks and CPUs.

Listing 3. 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> run obj=cpu0 pc=0x40000000 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.

Chapter 5. 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.

Chapter 6. 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.

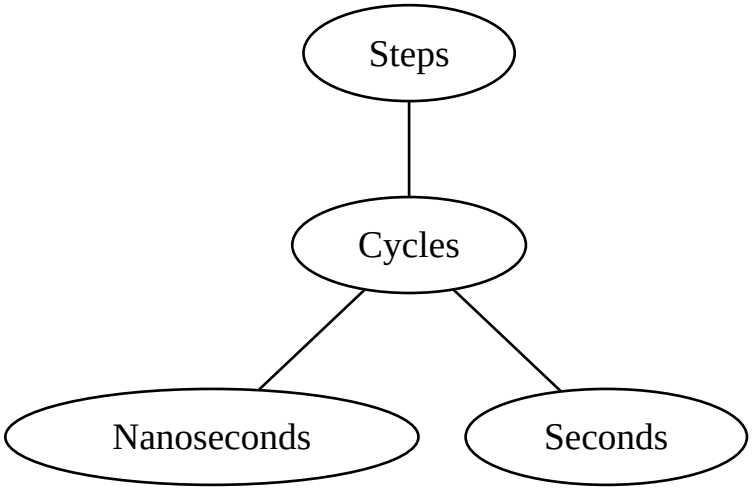
Time sources may have parent time sources for posting synchronized events.

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

- Processors
- Clocks
- Machine
- Scheduler

6.1. Time Base

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



6.1.1. 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.

6.1.2. 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).

6.1.3. Nanoseconds

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

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

6.1.4. Seconds

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

6.2. 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*.

6.2.1. Time Quantum

The *Scheduler* and *Machine* are 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 *Machine* model is configured using a quantum in nanoseconds. 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.

Chapter 7. 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`.

7.1. 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.

`--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.

7.2. 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`.

7.3. 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}`.

7.4. 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.

7.5. Commands

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

7.5.1. Snapshot Commands

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

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.

7.5.2. 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.

7.5.3. 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:

object-create

Creates an object, the command takes two or three parameters. The class parameter indicates the class of the object to be created, name indicates the object name (this name should be unique) and the third optional parameter `args` allows you to list a number of arguments formatted as `name:value` pairs in a comma separated list. The arguments are class specific, consult the class documentation on the allowed arguments. Example `object-create class=Leon3 name=cpu0 args="cpuid:0"`

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 objects created with object-create.

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.

7.5.4. 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

7.5.5. Execution Commands

run (object-run)

Run the machine or CPU for a given time

step (object-step)

Step the machine or CPU for a given number of steps

7.5.6. Other Commands

script-run

Run python script

temu-quit

Quit TEMU

temu-help

Show help

temu-version

Show version number

Chapter 8. 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.

8.1. 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.

8.1.1. 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

for this. Major versions can add substantial new features.

8.1.2. TEMU 3

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

8.1.3. Clarification on C++ APIs

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

8.2. 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.

Chapter 9. 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.

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:

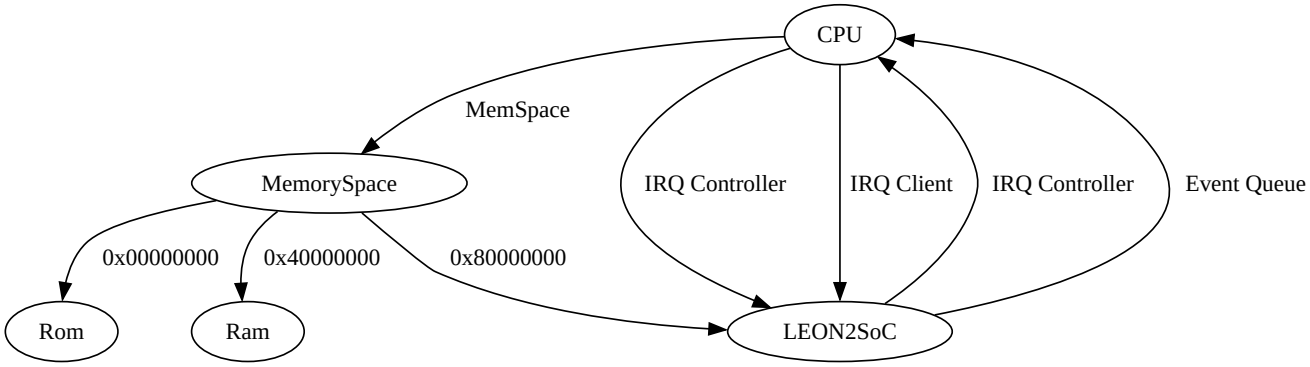
- MemAccessIface
- MemoryIface
- CpuIface

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. For example, `temu_getInterface(cpu, "MemAccessIface")` will return the pointer to the memory access interface structure provided by the CPU object (or NULL if not available). You need to cast the interface pointer to the correct type. The type mappings are provided in the model manuals.

9.1. 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.



9.2. 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 4. 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
`temu_objectForName()`	Get a named object
`temu_objsysClear()`	Delete all objects and classes

Function	Description
`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

9.3. 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.

9.4. 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.

9.5. 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.

9.6. 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 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.

Chapter 10. 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
- Machines
- Schedulers

The scheduler was introduced in TEMU 4 to expose a parallel scheduler. The machine in turn is the legacy single threaded scheduler. A processor and clock are bottom level time sources.

To run machines, processors or clocks the user invokes the global run-command, passing the time source as the `obj`-parameter.

However for the scheduler, the corresponding command method in the scheduler should be invoked.

10.1. Stepping and Running

TEMU distinguishes between running and stepping a system.

When running a system, the whole system runs freely for some time. At the end of the run (ignoring early stops such as breakpoints), the local time for each processor should be roughly the same.

When stepping a system, we are instead controlling the exact number of executed instructions for one of the processors. When the processor in question exceeds it's quanta, other processors may run.

10.2. Processors and Clocks

Processor and clocks are the lowest level in the scheduler hierarchy. Processors and clocks can be run using the `CpuIface` or the `ClockIface`.

10.2.1. Command Line

```
run obj=cpu0 time=1.0
```

```
# Step a processor for two instructions
step obj=cpu0 steps=2
```

10.2.2. API

```
// Run processor for 10000 cycles
cpuIfaceRef.Iface->run(cpuIfaceRef.Obj, 10000);

// Run processor for 10000 steps
cpuIfaceRef.Iface->step(cpuIfaceRef.Obj, 10000);
```

10.3. Machine Model

The Machine model is deprecated as of TEMU 4.0; in favour of the Scheduler.

The Machine-model was the traditional way of running multi-core processors in TEMU 2 and 3.

The machine model is an object that collects several processors together, enabling them to be run as one unit.

The model runs processors in a *round-robin* schedule. Each processor is executed for a *quanta*, before running the next processor. The length which is configurable on the *machine* in simulated nanoseconds.

As the machine is using nanoseconds, it cannot be used for processors running faster than *1 GHz*.

10.3.1. Command Line

```
// Step processor number 1 in the machine for 10 steps
step obj=machine0 cpuidx=1 steps=10
```

10.3.2. API

```
// Run the machine for 10000 nanoseconds
machineIfaceRef.Iface->run(machineIfaceRef.Obj, 10000);
```

10.4. Scheduler

The scheduler model is currently not controllable by the GDB Server. To use the TEMU GDB Server, the Machine model must be used instead.

The TEMU Scheduler is a globally available object. The last created Scheduler-instance is

automatically registered as the *current scheduler*.

When running the *scheduler*, the processors registered in it will run in parallel if so configured. However, when stepping the scheduler, the scheduler falls back to serial *round-robin* scheduling in order to remain deterministic.

10.4.1. Command Line

Listing 4. Creating a Scheduler

```
Scheduler.new name=sched
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
```

Listing 5. Running and Stepping the Scheduler

```
# Run the simulator for 10 seconds
sched.run secs=10.0

# Step processor 2 for 10 steps
sched.step cpu=cpu2 steps=10
```

10.4.2. API

Listing 6. 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();
```

Chapter 11. 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.

11.1. 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

11.1.1. 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.

11.1.2. 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.

11.1.3. 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.

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.

11.2. 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

(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.

11.3. Severities

Severities 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 severities 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.

11.4. 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.

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.

Chapter 12. 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 stacked events (in LIFO order).
- CPU synchronized timed events
- Machine synchronized stacked events
- Machine synchronized timed events

That means that machine synchronized events will not be executed until all the stacked events and the normal timer events have been executed.

```
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 {
  teSE_Cpu,     // Trigger event when CPU reaches timer
```

```
  teSE_Machine, // Synchronise event on machine
} 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);
```

12.1. Events from Other Threads

Note that it is in general not possible to post events from an other thread than the one controlling the execution of the emulator. To solve this issue, the function `temu_postCallback()` is available. This function is thread safe, and posts an event to be executed alongside the rest of the emulator event queue executions.

The event will be called when the emulator thinks it is safe while it is running, which is when the CPU event timer expires (or when the machine quanta expires by other means). 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. For machines, each CPU runs at most a quanta of time, so the check can be done at quanta expiration.

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.

12.2. 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()`
- `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

Chapter 13. 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.

The processor models provide static instruction timing which is useful in order to predict performance in certain cases. Timing does not take pipeline dependencies into account, so there is no simulation of branch prediction, pipeline stalls or superscalar execution. It is possible to insert user provided cache models in the memory space object such models can add more timing accuracy to the emulation at the expense of performance.

A processor object can be embedded inside a machine object. The machine objects can be used in order to control multiple processors as a group. This is the primary way that multi-core, and multi-computer systems are supported.

When running a machine with multiple processors, the processors are temporally decoupled, and the machine synchronizes the processors at various time points. These time points include a mandatory time-quanta, and synchronized events.

13.1. Running a CPU or Machine

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.

13.1.1. 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).

13.1.2. 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
- Early exit (other reason which can be forced by event handlers or others)

13.1.3. 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 machine is stepping, it is not the machine 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 which is the current CPU. As the "current" CPU can change (e.g. when the CPU finishes its scheduling quanta), it is advisable to set this parameter.

13.1.4. 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).

When machines are run, the CPUs part of the machine will all advance for the time given to the machine. In this case, it is not possible to specify time in the unit "cycles" as each CPU in a machine may have a different clock frequency. Instead, the machine is executed for a given number of nanoseconds.

13.1.5. 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()`.

13.2. 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.

13.3. Multi-Core Emulation and Events

Multi-core processors are simulated by creating a machine object, and adding multiple CPU cores to it, and associating a single memory space object which all the cores (in fact, a non shared memory multi-computer system is a machine object with separate memory spaces for each CPU).

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 machine 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 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.

It is possible to manipulate the machine's time-quanta during execution.

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. For these cases, the machine object allows for the posting of synchronized events. These will ensure that the CPU scheduling window is aborted before the quanta is finished and all processor will agree on time (within the granularity of the worst case instruction time).

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.

13.4. 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,
                          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;
```

Chapter 14. 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.

14.1. 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()`.

14.2. 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:

0

1-byte transaction

1

2-byte transaction

2

4-byte transaction

3

8-byte transaction

14.2.1. 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.

TEMU 2 only supported single data unit transactions. Large data blocks could be transferred with the `MemoryIface`, however these were not subject to IOMMU handling. Large transactions is a new feature in TEMU 3. The `MemoryIface` is now deprecated in favour of large transactions.

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;
    // 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.

14.2.2. 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.

14.2.3. 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 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.

14.3. 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).

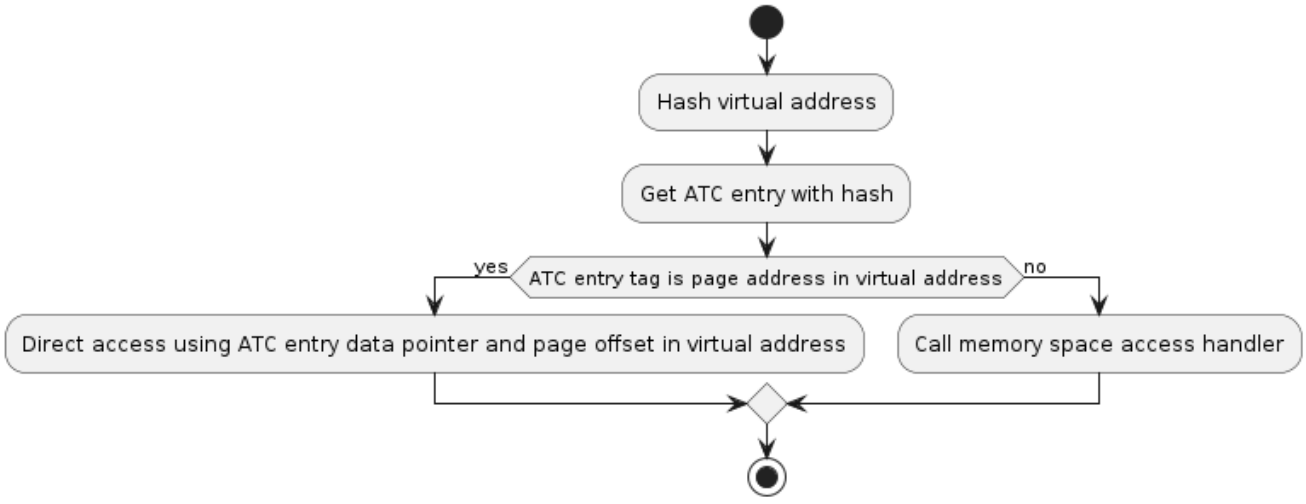


Figure 2. Memory access flow using the ATC

14.4. 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 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.

14.4.1. 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.

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.

14.4.2. 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.

14.5. Interfaces

14.5.1. 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
  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.
};
```

14.5.2. Memory Interface

Deprecation Notice

The memory interface has been deprecated in favor of large memory transactions. The memory interface is scheduled to be removed in TEMU 4.

The memory interface is a common interface for memory storage devices. It provides procedures for writing and reading larger blocks of memory. The interface takes an offset from the base address of the object is mapped (normally you use a memory space object to cover the physical address space).

The Size parameter is in bytes, and the Swap parameter specify the log-size in bytes of the data units to read or write. Note that the address/offset is assumed to be aligned at the unit size and the size will be truncated if it does not represent a whole number of data units.

I.e. when reading 64 bit words, the size should be 8, 16, 24, ... and the swap argument should be set to 3.

```
typedef struct temu_MemoryIface {
  void (*readBytes)(void *Obj,
                    void *Dest, uint64_t Offs, uint32_t Size,
            int Swap);
  void (*writeBytes)(void *Obj,
                     uint64_t Offs, uint32_t Size, void *Src,
             int Swap);
} temu_MemoryIface;
```

Chapter 15. 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.

Chapter 16. 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.

16.1. Snapshot Restore Phases

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

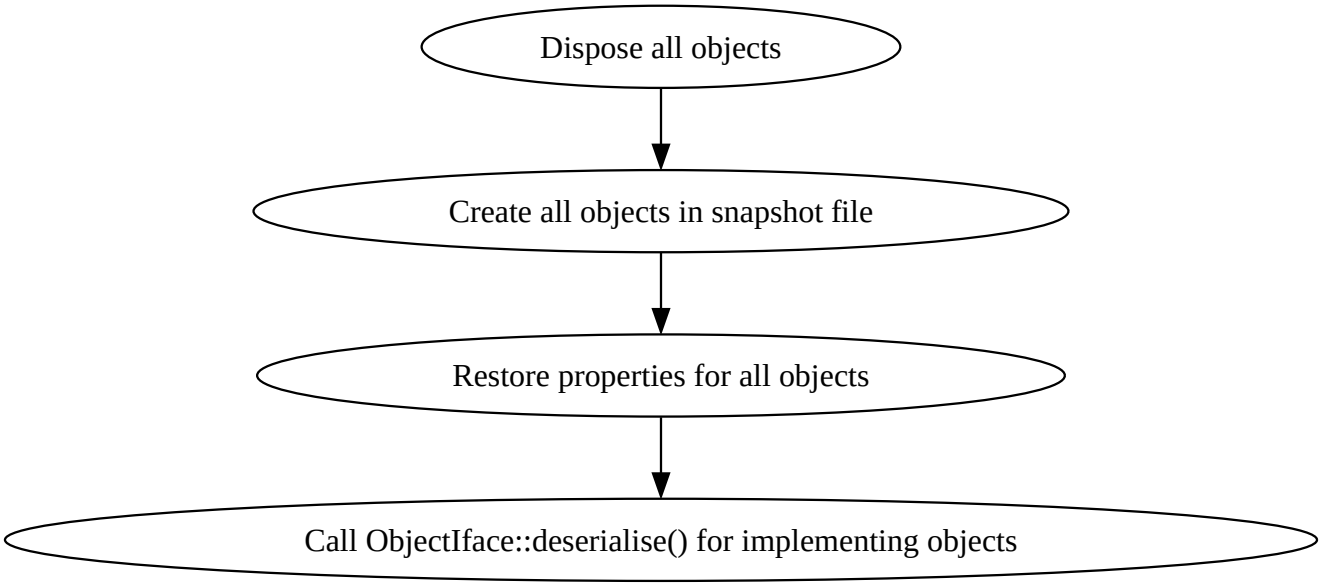


Figure 3. Snapshot Restore Phases

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 implementor 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.

16.2. JSON Caveats

16.2.1. 64-Bit Values

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.

16.2.2. 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.

Chapter 17. 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).

17.1. 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
```

17.1.1. Source Level Debugging

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.

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.

17.2. Legacy GDB Server

The legacy GDB server assumes it have full control over the running processors. It only works with separate CPUs and machine objects. For instructions relating to the scheduler compatible GDB server introduced in TEMU 4.1, please see Section 17.3.

TEMU (as of 2.1) comes with a built-in non-intrusive GDB server speaking the GDB remote protocol. The server is launched using the `gdb-server` command.

The server uses the `SO_REUSEADDR` socket option, so it is always possible to connect to the same port after disconnection without further delay.

To start the gdb-server inside the interactive CLI simply run the `gdb-server` command. It takes three parameters, `machine` or `cpu`, `port` and `page-table-root`.

When the server is running, GDB has control over the execution of the emulator, you can quit the GDB server command by interrupting it in the CLI (using `Ctrl` + `C`) or by disconnecting GDB.

If no arguments are given, the command defaults to `machine0`, `cpu0` and port `6666` for the different arguments respectively. The `page-table-root` argument is only used if set explicitly.

The GDB server supports multi-core debugging, by exposing the cores as different threads. The multi-core support is limited in some ways, for example, breakpoints cannot be set per core.

Uploading binaries via the GDB remote protocol is very slow. It is recommended to load the binaries in the TEMU CLI and then specify which file you are debugging to GDB.

The GDB server does not know about operating system threads. Instead it treats a GDB-thread as a CPU core (numbering the threads from 1 for CPU 0 and up).

The GDB remote debugging protocol is not designed to be interfaced with emulators. One issue is that in order to inspect the stack, GDB will issue memory read commands to the remote target. This causes a problem in an emulator since many stack entries (especially on the SPARC target) will be in CPU registers. Thus, when the GDB program asks to read memory which is on the stack and in registers, the server will return the register content and not the memory content. Consequently if memory referring to register-shadowed memory content is modified, the remote target will write both memory and registers.

17.2.1. 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 page table root would then be calculated as:

Listing 7. 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 8. 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.

17.2.2. Example

Starting the GDB-server in the CLI:

```
temu> gdb-server machine=machine0 port=6666
Starting GDB server... (^C to stop)
GDB connected.
```

Connecting with GDB

```
(gdb) target remote localhost:6666
(gdb) file my-application.elf
```

17.3. New GDB Server

With the introduction of the new scheduler a new GDB server running in a separate thread was needed.

The new 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
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 Section 17.2. When GDB is disconnected, the TEMU scheduler is automatically resumed.

17.3.1. Limitations

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.

Chapter 18. 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.

18.1. 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);
```

18.2. 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:
```

```
  - {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.

Chapter 19. 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.

19.1. 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.

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.

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

19.2. 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.

19.3. Performance Improvement Capabilities

19.3.1. 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).

19.3.2. 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’s, different processors run more instructions, before stopping, waiting for other processors to catch up. This in turn results in less time spent in processor switching and synchronization which is overhead.

19.3.3. 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.

19.3.4. 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
    .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);
```

19.3.5. 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);



// From model when register state changes so it should wake up the software
cpuIf.Iface->wakeUp(cpuIf.Obj);
```

19.3.6. 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);
```

19.3.7. 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.

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 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
}


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);
```

Chapter 20. 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 done using a TEMU plugin. In principle, the

This section discuss some approaches and pitfalls with fault injection in TEMU.

20.1. 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.

20.2. 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.

20.2.1. 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.

20.2.2. 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.

20.3. Memory Faults

20.3.1. 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.

20.3.2. Correctable Memory Errors

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.

20.3.3. 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.

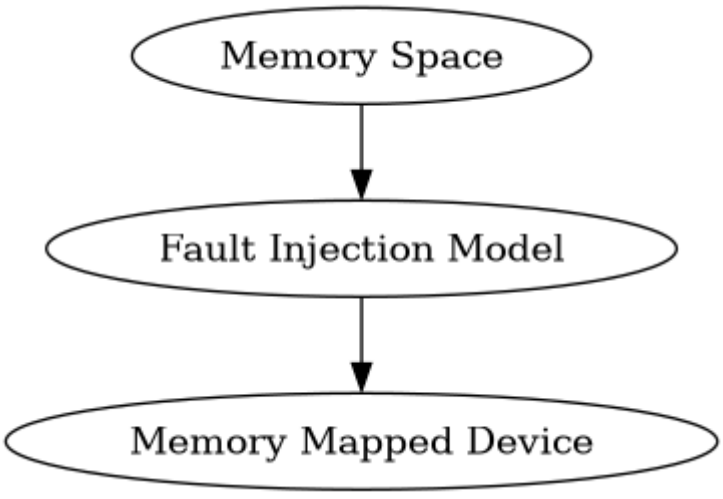
It is possible to add permanently uncorrectable errors by connecting a custom memory access interface to the memory space `faultyHandlers`.

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`.

20.3.4. 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.

20.4. 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

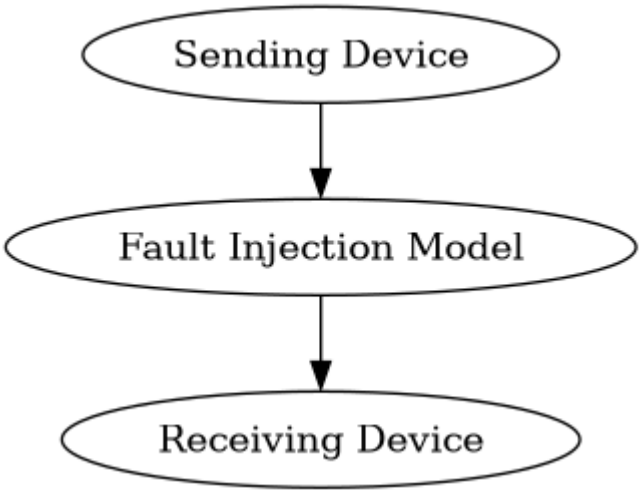
effects are modelled. Several bus models thus have a way of signalling bus model errors such as parity or CRC failures using error bits in frame and packet structures.

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.

20.4.1. Point to Point Buses

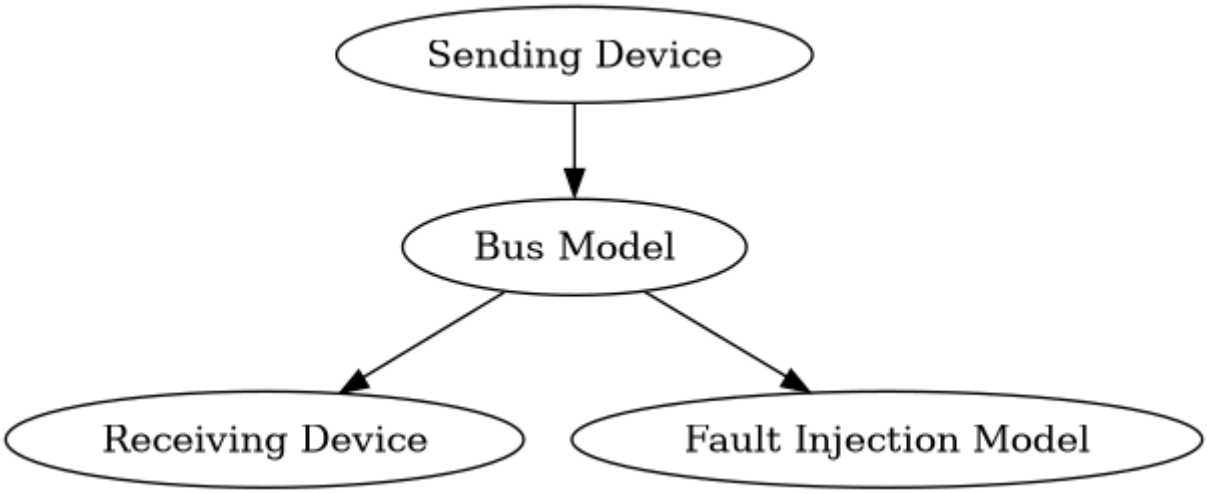
For point to point buses such as serial and SpaceWire, a tap device must be implemented.



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.

20.4.2. 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

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.

Chapter 21. 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. Thus, construct CPUs and machines using the CLI, and then run the more sophisticated code using Python.

Chapter 22. Examples

22.1. 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/lateststhare/temu/sysconfig/`.

22.1.1. CLI

To load a system configuration in the current directory from the CLI:

```
snapshot-restore Leon2.json
```

22.1.2. 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");
```

22.2. 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/latest/share/temu/sysconfig/`.

```
import Leon2
import Leon2SoC
import Memory
import Console

object-create class=Leon2 name=cpu0
object-create class=Leon2SoC name=leon2soc0
object-create class=MemorySpace name=mem0
object-create class=Rom name=rom0
object-create class=Ram name=ram0

# Console is a virtual serial port sink that prints output to STDOUT
```

```
object-create class=Console 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.

22.3. 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"
```

```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.
```

```
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.
```

```
  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;
}
```

Chapter 23. Modelling Guide

23.1. 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 23.4.2.

23.1.1. 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/latest/`). 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.

23.2. Basic

23.2.1. The TEMU Object System

Object System Glossary

class

A description of a POD type with properties, interfaces and a name. Realised 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 standardised 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.

To add the plugin initialiser function to your file add the following code to your device model plugin:

Listing 9. 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 10. 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,
```

```
                        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 conventent 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 contiain 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 realised 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 11. 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

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 12. 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 23.2.1.3, the objects are connected using the `temu_connect()` function.

23.2.2. 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:

```
#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 13. 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
```

```
    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.

23.2.3. 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,
};
```

```
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);
}
```

23.2.4. 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.

23.2.5. 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);
```

23.2.6. 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 synchronised time stamp (i.e. all CPUs having reached a specific time), then events can be posted by setting the teSE_Machine 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).

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_Machine, this flag indicates whether the event should go on the CPU queue, or if it should be forwarded to the machine object. If the event is forwarded, it is:

1. The delta is converted to a nanosecond offset.
2. Rounded to the start of the next machine time 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 machine synchronised 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 synchronised event at 42 ns in the future
temu_eventPostNanos(Dev->Super.TimeSource, Dev->MyEventID, 42, teSE_Machine);

// 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.
```

```
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.

23.3. Advanced

23.3.1. 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 snapshoting 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 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
};
```

23.3.2. 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:

Listing 14. 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);
}
```

23.3.3. 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.

23.3.4. 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 milbus 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

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 15. 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);
}
```

23.4. Examples

23.4.1. 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 16. 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));
```

```
  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);
}
```

```
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");
```

```
  temu_addInterface(Cls, "MemAccessIface", &MemAccessIface);
}
```

23.4.2. Important Interfaces

Listing 17. 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 18. 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;
```

Listing 19. Device Interface

```c
#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;
```