TERMA

TEMU
User Manual

Mattias Holm

Version 1.8, 2018-03-15

Table of Contents

Table 1. Record of Changes

Rev	Date	Author	Note
1.8	2018-03-15	MH	Added section on source level debugging without GDB.
1.7	2018-01-24	MH	Documented the new license manager.
1.6	2017-07-13	MH	Added section about plugin loading.
1.5	2016-11-23	MH	Add sections about properties and pseudo properties. Add section about components.
1.4	2016-03-01	MH	Rewrite section on event handling for new event API. Add section on software debugging using the CLI.
1.3	2015-11-30	MH	Updated package locations for Python wrappers. Refer to C99 interface to the GDB server and mention the removal of the C++ interface.
1.2	2015-09-14	MH	Added section about the GDB server. Added description on instruction semantics. Added section about cache emulation. Added section about CLI variable support.
1.1	2015-07-27	MH	Clarify whole document and add multicore discussion
1.0	2015-03-01	MH	Initial version

1. Introduction

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

TEMU is a microprocessor emulator that supports the SPARCv8 processors ERC32, LEON2, LEON3 and LEON4. The emulator can emulate multi-core processors.

TEMU is a full system emulator, meaning that it is capable of emulating (multi-core) microprocessors, memory and peripherals. Different devices are written as plugins, meaning that the system supports both pluggable CPU, memory and device modules. In-fact, systems are constructed by connecting different modules together, meaning that there is no hard-wiring to any special memory layout or on-chip devices.

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

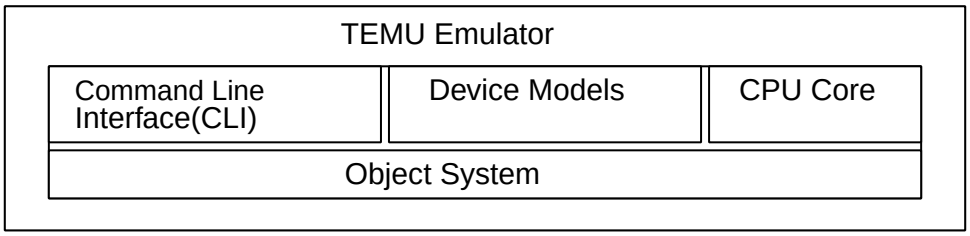


Figure 1. Layers of the Terma Emulator

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 and with its own models, while the API allows the user to integrate the emulator with other simulators.

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. In general, the target and model manuals document the properties and interfaces these systems implement. They are however not tutorials and those documents are not intended to help you get started.

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 2. TEMU Target Manuals

Document	**Description**
SPARCv8 Target Manual	Manual for all the SPARC CPU cores

2.2. Model Manuals

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

The following table lists some of the manuals.

Table 3. TEMU Model Manuals

Document	**Note**
Modelling Guide	How to write device models

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

3. Getting Started

3.1. Installation

To install TEMU, the best approach is to use the RPM or DEB files. The latest versions can be downloaded from http://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 4. 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-generic-Linux-x86_64.rpm

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

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

By default, the packages install TEMU in /opt/temu/x.y.z. 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/x.y.z/lib and the tools in /opt/temu/x.y.z/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/x.y.z+asserts/

3.2. License Files

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

3.2.1. Installing a License

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

```
$ 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).

3.2.2. 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.

3.2.3. Legacy Licenses

Legacy TEMU licenses are still supported.

TEMU will check your computer for a valid license file at start-up.

By default, TEMU will look for a license file in the following locations (in this order):

```
${TEMU_LICENSE_FILE}
./temu-license.json
~/.config/temu/license.json
```

3.3. Running the Emulator

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

3.4. 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/x.y.z/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
```

3.5. 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), the user needs to ensure that assumptions made by the application software about the environment provided by the boot software are valid. On the SPARC, this implies in many cases that the stack and frame pointer are initialised to point out the end of RAM. But some systems (e.g. Linux) assume that also timer registers are initialised.

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.

4. Plug-In Support

TEMU is build as a set of plug-ins. Almost every model, including the processor models are located in plugins that are loaded at run-time.

A plugin can be loaded using the command plugin-load (aliased as import). Or the API function temu_loadPlugin(). Loading is done in the following way:

1. If the plugin contains a directory separator (i.e. a '/' on Linux and macOS). Then the plugin is loaded directly, assuming it is a file.
2. If the plugin does not contain a directory separator, then TEMU first tries to find the plugin in the internal plugin path. That path can be manipulated with the commands: plugin-append-path and plugin-remove-path. The current path can also be printed using plugin-show-path. The plugin path is searched, attempting to locate the plugin with the name libTEMU<plugin-name>.so first, and secondly as lib<plugin-name>.so.
3. If the plugin is not found in the internal plugin path, then a last attempt is made to use the system library paths. If falling back to the system paths, then the plugin must be named "libTEMU<plugin-name>.so".

When a plugin is loaded, the first thing that is done is to execute the "temu_pluginInit()" function that must be implemented in the plugin. That function typically registers classes and commands with the TEMU runtime library.

5. Command Line Interface

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. But 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.

5.1. 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.

5.2. 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.

5.3. 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}.

5.4. 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.

5.5. Commands

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

5.5.1. Checkpointing Commands

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

checkpoint-restore

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

checkpoint-save

The save function writes a checkpoint in JSON format to disk. Memory content is typically dumped as raw data in a binary blob (in an auxiliary file). The endianess 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.

5.5.2. 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 object as a parameter.

memory-load

Load executable file (srec or elf). The Command automatically detects the format of the file by both extension and binary analysis.

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.

5.5.3. 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.

Example:

```
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.

5.5.4. 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

5.5.5. 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

5.5.6. Other Commands

script-run

Run python script

temu-quit

Quit TEMU

temu-help

Show help

temu-version

Show version number

6. 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 initialise 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.

6.1. 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.

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 bugfixes 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.

6.1.1. Clarification on C++ APIs

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

6.2. 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.

6.3. The 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:

- Standardised 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. It is also possible to define *external classes*, these are special classes which describe object created outside the emulator. 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 serialised 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 serialised 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 behaviour for a standardised 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).

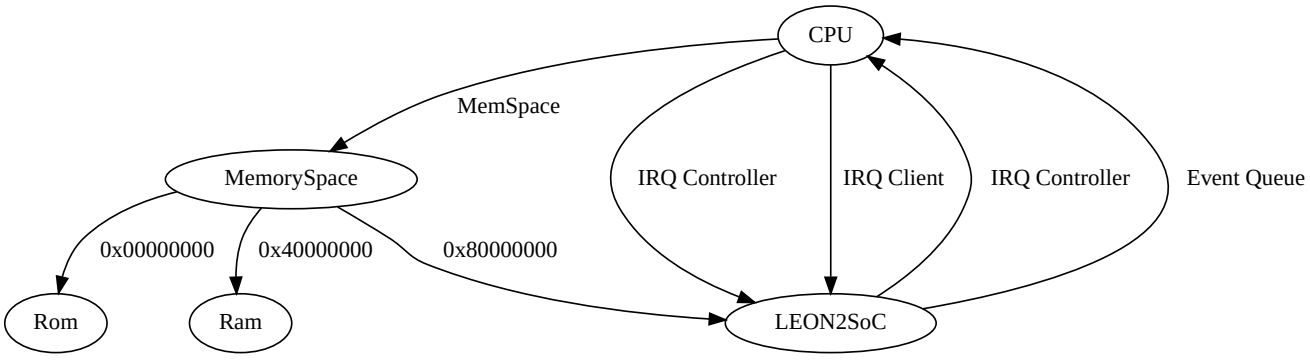
The most important core interfaces are the following:

- 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.

6.4. 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.



6.5. Object System Functions

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

Table 5. TEMU Object System Functions

Function	Description
temu_addInterface()	Add interface to class
temu_addObject()	Register an externally created object
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
temu_readValue()	Get property by calling the read function
temu_registerClass()	Create a new class
temu_registerExternalClass()	Create a new external 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

6.6. 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 checkpointed. 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.

6.7. 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 check-pointing. From the user interface point of view, pseudo properties behave the same as normal properties.

6.8. 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.

6.8.1. Object Interface

The object interface provides a way to add support functions for the object system, for example custom serialise and deserialise functions, and custom sanity checkers for a class.

```
typedef struct {
  void (*serialise)(void *Obj, const char *BaseName, void *Ctxt);
  void (*deserialise)(void *Obj, const char *BaseName, void *Ctxt);
  int (*checkSanity)(void *Obj, int Report);
  void (*timeSourceSet)(void *Obj);
} temu_ObjectIface;
```

6.8.2. 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;    //!< Virtual address
  uint64_t Pa;    //!< Physical address
  uint64_t Value; //!< Resulting value (or written value)

  //! 2-log of the transaction size.
  uint8_t Size;

  uint64_t Offset; //!< Offset from model base address
  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;
```

6.8.3. Memory Interface

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;
```

6.8.4. 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;
```

6.9. 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 backlink, 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.

6.10. Event API

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 synchronised 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 10 ns, while a 50 MHz CPU has an event posting accuracy of 20 ns.

When posting machine synchronised 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 synchronised events are used, the machine scheduler will adjust its quanta length to ensure that CPUs do not execute longer than needed. Note that synchronised events are executed after a CPU has returned to the machine object, potentially executing non-synchronised 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 synchronised events), or after the current time quanta finishes in case of machine synchronised events.

Events are prioritised as follows:

- CPU synchronised stacked events (in LIFO order).
- CPU synchronised timed events
- Machine synchronised stacked events
- Machine synchronised timed events

That means that machine synchronised 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);
```

6.10.1. 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.

6.10.2. 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.". 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 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

7. 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 close to 100 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 synchronises the processors at various time points. These time points include a mandatory time-quanta, and synchronised events.

7.1. 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.

7.1.1. 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 powerdown 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 happend 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).

7.1.2. 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)

7.1.3. 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.

7.1.4. 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.

7.1.5. Instruction Behaviour

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 behaviour.

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().

7.2. 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.

7.3. 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 quantas 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 synchronised 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).

Synchronised 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.

8. 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.

8.1. 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). It does this by defining a two level page table (with 12 bits per level). 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().

8.2. 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).

8.3. 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.

8.3.1. 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.

8.3.2. Tracing Memory Accesses

It is possible to utilise 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.

9. 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.

10. Checkpointing

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 serialised to a JSON file. This is done using the checkpoint-save and checkpoint-restore commands (these have aliases save and restore).

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

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

10.1. JSON Caveats

10.1.1. 64-Bit Values

JSON does not allow for larger than 53 bit integers to be stored (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 serialised, 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.

10.1.2. 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 checkpoint 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.

11. 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).

11.1. CLI Based Software Debugging

The TEMU command line interface will when reading ELF files, also load the symtables (there is also an API for inspecting ELF symtables). 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
```

11.1.1. 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 / unmute 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.

11.2. GDB Server

TEMU (as of 2.1) comes with a built-in non-intrusive GDB server speaking the GDB remote protocol. The server is available in three formats, as a C library (the old C++ library has been replaced with a stable C-interface in v2.2), as a stand-alone command line tool, and as a separate command in the TEMU command line interface.

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 stand-alone GDB server application, simply execute the temu-gdbserver command. The command takes the following arguments:

--paths-file [path]

A TEMU CLI script whose purpose is to set any needed paths.

--machine-file [path]

A TEMU CLI script whose purpose is to construct a machine object. This is executed after the paths-file has been loaded and executed.

--machine [machinename]

Name of the machine object that should be controlled with the GDB server. The default is "machine0".

--cpu [cpuname]

Name of the cpu object that should be controlled with the GDB server. The default is "cpu0". Note that this is only used in case the machine file does not define a machine object, i.e. you are intending to run a single core system without a machine object.

--port [portnum]

TCP port of the GDB server. The default is port 6666.

To start the gdb-server inside the interactive CLI. Simply run the gdb-server command. It takes two parameters, machine (or cpu)and port. 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 ^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 GDB server supports multicore debugging, by exposing the cores as different threads. The multicore 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 that instead of uploading them via the remote, load the binaries in the 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). This behaviour may not be what you want when debugging certain type of application software (where you expect a thread to correspond to an OS thread), however, it is very useful when debugging the early boot issues and issues related to CPU scheduling in the operating system.

In order to debug OS threads, it is recommended that you write a console model that binds an emulated serial link to a TCP/IP port for connecting to with GDB. This type of debugging will rely on having a GDB remote stub in your target software that talks to the serial port, and thus the debugging will be intrusive in contrast to the non-intrusive GDB server library that is part of the emulator.

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.

11.2.1. Example

Starting the standalone GDB server:

```
$ temu-gdbserver --port 6666 --machine-file leon3-dual.temu --machine machine0
```

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
```

12. Scripting Support

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 http://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.

13. Examples

13.1. 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 serialised 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/share/temu/sysconfig/.

13.1.1. CLI

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

```
checkpoint-restore Leon2.json
```

13.1.2. 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");
```

13.2. 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/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.

13.3. 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"

#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;
}
```