T-EMU: User Manual

Prepared

Mattias Holm
Technical Project Manager

Checked

Dan Søren Nielsen
QA Manager

Approved

Michela Alberti
Director of Operations

Record of Changes

Author	Description	Rev	Date
Mattias Holm	Initial Version	1.0	2015-03-01
Mattias Holm	Clarify whole document and add multicore discussion	1.1	2015-07-27
Mattias Holm	Added section about the GDB server. Added description on instruction semantics. Added section about cache emulation. Added section about CLI variable support.	1.2	2015-09-14

Table of Contents

1. Introduction

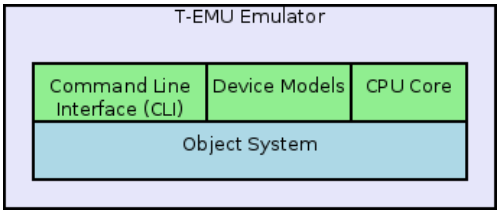
This document is the T-EMU (Terma Emulator) Software Users Manual. It describes the fundamental concepts and general usage of the T-EMU libraries and the command line interface.

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

T-EMU 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 T-EMU, 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 T-EMU 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 1. T-EMU 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 2. T-EMU 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
GRLIB AhbUart Device Model Manual	GRLIB manual
GRLIB FtmCtrl Device Model Manual	GRLIB manual

3. Getting Started

3.1. Installation

To install T-EMU, the best approach is to use the RPM or DEB files. The latest versions can be downloaded from http://t-emu.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:

```
# Install RPM
$ rpm -ivh t-emu-2.0.0-generic-Linux-x86_64.rpm

# Install DEB
$ dpkg -i t-emu-2.0.0-generic-Linux-x86_64.deb

# Install Tarball (.tar.bz2)
$ bunzip2 t-emu-2.0.0-generic-Linux-x86_64.tar.bz2
$ tar xvf t-emu-2.0.0-generic-Linux-x86_64.tar
```

By default, the packages install T-EMU 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.

T-EMU 2.0 consist of a set of libraries and a command line tool. The libraries are normally installed in `/opt/temu/2.0.0/lib` and the tools in `/opt/temu/2.0.0/bin/`. The binaries and libraries have been liked with the RPATH option, so there is no need to set `LD_LIBRARY_PATH`.

There are also packages for a build which has asserts enabled. Asserts have a performance penalty, which at times can be heavy. Therefore, assert builds are opt-in. These packages installs under: `/opt/temu/2.0.0+asserts/`

3.2. License Files

T-EMU will check your computer for a valid license file.

By default, T-EMU will look for a license file in the following locations:

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

See http://t-emu.terma.com/ for more information on licenses. Note that you must have a valid license to run T-EMU.

3.3. Running the Emulator

To start the command line interface (CLI), simply run `/opt/temu/2.0.0/bin/temu` or `/opt/temu/2.0.0+asserts/bin/temu`. The command line interface exists to run the emulator in stand alone mode.

3.4. Creating a New Machine

When T-EMU is running it will normally display the `t-emu>` 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/2.0.0/share/temu/sysconfig/`. Common configurations that instantiate different types of systems are available. The command line scripts can be executed using the exec command. This can be done as illustrated in the following examples:

Create a LEON2 System.

```
t-emu> exec /opt/temu/2.0.0/share/temu/sysconfig/leon2.temu
```

Create a Dual Core LEON3 System.

```
t-emu> exec /opt/temu/2.0.0/share/temu/sysconfig/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.

Note

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 and CPUs.

Load and Run Software Image.

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

Note

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. 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 T-EMU
$ temu
no such file: '~/.config/temu/init'
no such file: './temu-init'
t-emu>
```

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`.

4.1. Command Line Interface Options

The command line interface support the execution of non-interactive sessions via the `--run-commands` flag.

`--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-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.

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.

4.2. 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`.

4.3. 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

4.4. 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.

4.5. Commands

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

4.5.1. 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.

4.5.2. 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.

4.5.3. 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.

4.5.4. 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

4.5.5. 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

4.5.6. Other Commands

script-run	Run python script
temu-quit	Quit T-EMU
temu-help	Show help
temu-version	Show version number

5. 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 T-EMU library
  return 0;
}
```

Warning

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

5.1. Deprecation Policy

T-EMU 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 T-EMU 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.

5.1.1. Clarification on C++ APIs

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

5.2. The Object System

T-EMU 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 T-EMU 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.

Object	An instantiated class. Normally the T-EMU 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).
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.
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 T-EMU this is implemented as structs of function pointers that are registered to a class.

When setting up a simulator based on T-EMU, 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 on-board 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).

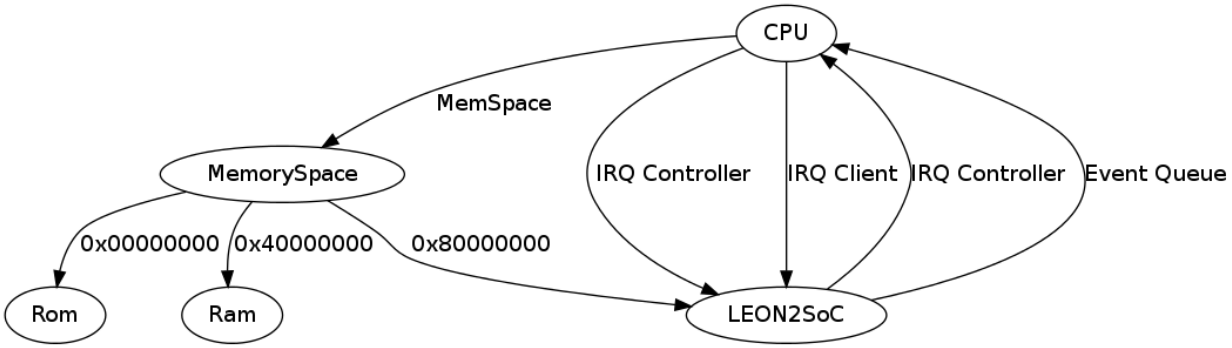
The most important core interfaces are the following:

- MemAccessIface
- MemoryIface
- CpuIface
- EventIface

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

5.3. 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.



5.4. 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. T-EMU Object System Functions

Function	Description
temu_getValue()	Get property without side-effects
temu_readValue()	Get property by calling the read function
temu_setValue()	Set property without side-effects
temu_writeValue()	Set property by calling the write function
temu_registerClass()	Create a new class
temu_registerExternalClass()	Create a new external class
temu_addProperty()	Add property to class
temu_addInterface()	Add interface to class
temu_getInterface()	Get interface pointer by name
temu_objsysClear()	Delete all objects and classes
temu_createObject()	Create a new object from an internal class
temu_addObject()	Register an externally created object
temu_disposeObject()	Delete object
temu_addNamedFunction()	Add a named function pointer
temu_classForName()	Get a class object by name
temu_classForObject()	Get the class object for an object

Function	Description
temu_objectForName()	Get a named object
temu_nameForObject()	Get the name for the given object
temu_loadPlugin()	Load a T-EMU plugin
temu_connect()	Connect an interface property to an interface
temu_serialiseJSON()	Save the state of the emulator
temu_deserialiseJSON()	Restore the state of the emulator
temu_checkSanity()	Look for unconnected interface properties

5.5. Interfaces

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

5.5.1. 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);
} temu_ObjectIface;
```

5.5.2. 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;
```

5.5.3. Memory Interface

The memory interface is a common interface for memory storage devices. It provides procedures for writing and reading larger blocks of memory.

```
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;
```

5.5.4. 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;
```

5.5.5. Event Interface

The event interface is used to expose an event queue provider and provides a common interface for pushing timed events on the given queue. The event interface is typically implemented by a CPU object, but is also provided by the machine objects which are essentially CPU schedulers. The sender parameter should be a pointer to the object that is posting the event (e.g. a timer object).

Normally, the CPU queues are based on cycles, while the machine object queue uses nanoseconds as event time.

In order to be able to post events using nanoseconds as unit, the event routines takes a flag parameter and the TEMU_EVENT_NS can be ored with other flags in order to specify events in nano-seconds instead. When posting events to a CPU, nano-second events are converted to cycles, which means that you do not have NS accuracy of the events, but 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.

Another flag option is the TEMU_EVENT_SYNC, which means that the event will be synchronised, such an event must be posted in at least the next time-quanta. And it will end up in the event queue of the machine object (if one exists). If a synchronised event is posted, with a triggering time in the current time quanta, the post fails with an error noted in the log.

In the case synchronised events are used, the machine scheduler will adjust its quanta length to ensure that CPUs do execute longer than needed. Note that synchronised events are executed after a CPU has returned, potentially executing non-synchronised events.

The event system supports three types of events, there are stacked events on the current CPU, these can be posted by an event handler or MMIO in order to execute an event *after* the current instruction finishes.

Events are prioritised as follows:

- Stacked events (in LIFO order).
- Normal timer events
- Synchronised events

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

An invariant is that when an emulator core returns, there should be no pending stacked or normal events to be triggered at the current time.

```
typedef struct {
  void (*stackPostEvent)(void *Obj,
                         void (*Ev)(void *, void *),
                         void *Sender, void *Data, uint32_t Flags);

  void (*postDeltaEvent)(void *Obj,
                         void (*Ev)(void *, void *),
                         void *Sender, void *Data,
                         int64_t Cycles, uint32_t Flags);

  void (*postAbsoluteEvent)(void *Obj,
                            void (*Ev)(void *, void *),
                            void *Sender, void *Data,
                            int64_t Cycles, uint32_t Flags);
  int64_t (*getEventDeltaTime)(void *Obj,
                               void (*Ev)(void *, void *),
                               void *Sender);
  int64_t (*getEventAbsoluteTime)(void *Obj,
                                  void (*Ev)(void *, void *),
                                  void *Sender);
  void (*descheduleEvent)(void *Obj,
                          void (*Ev)(void *, void *),
                          void *Sender, uint32_t Flags);

  void (*registerEvent)(void *Obj, const char *EvName,
                        void (*Ev)(void *, void *), uint32_t Flags);

  void (*postCallback)(void *Obj, temu_ThreadSafeCb Cb, void *Arg);


} temu_EventIface;
```

There is also a function API that can be used to implement the EventIface, this API is however only useful if you implement your own Machine or CPU objects. As CPU models must be implemented by Terma at present, this is in reality restricted to custom machine objects. This API is therefore not described in this document.

5.6. 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. Thus, the function postCallback is provided in the event interface (and in the Cpu.h header there is a wrapper of this function for machine and cpu objects called temu_postCallback()). This function can be used to post an event from a separate thread. The event will be called when the emulator thinks it is safe, which is when the CPU event timer expires (or when the machine quanta expires). Note that a lone CPU will post a null-event to ensure that the event handlers are triggered at regular intervals and not just when model events are executed.

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

5.7. Publish-Subscribe Events

A special type of event is an event that does not have a triggering time. They are instead triggered due to some action 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.

The functions for posting and listening to un-timed events include:

- temu_publishEvent()
- temu_subscribeEvent()
- temu_unsubscribeEvent()
- temu_notifyEventHandlers()
- temu_notifyEventHandlersFast()

When an event is published, the publication function returns an event id for the publishing object. The publishing object then use this event id to notify any event listners.

In order to avoid event namespace collissions, all events issued in T-EMU are prefixed with "temu.". It is recommended that user published events use their own namespace.

The event ID 0 is a special event 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.

Current events include:

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

6. Processor Emulation

The processor emulation capability in T-EMU 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 between the processor core and memory system 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.

6.1. 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.

6.1.1. 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).

6.1.2. 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)

6.1.3. 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.

6.1.4. 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.

6.1.5. 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

Note

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.

Note

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()`.

6.2. 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 interface for event posting that the CPU models implement. All CPUs implement `stackPostEvent()`, `postDeltaEvent()` and `postAbsoluteEvent()` as part of the CPUs *EventIface* struct. Delta- and absolute events are fired at the 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 function-object pairs, 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.

6.3. 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 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 cycle 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.

Tip

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.

Important

This approach to multi-core emulation 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 ensures that the emulation is deterministic.

Important

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

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).

Important

Synchronised events should always be posted with a firing time in at least the next CPU scheduling quanta. Otherwise, CPUs that have finished their quantas will not be in sync.

7. Memory Emulation

Memory emulation in T-EMU 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.

7.1. Memory Spaces

T-EMU 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 three ATCs: one each for read, write and execute operations.

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.

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_mapMemorySpace()`.

7.2. 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).

7.3. 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 at arbitrary places in the hierarchy 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.

7.3.1. The Generic Cache Model

The emulator (as of T-EMU 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. 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.

8.1. JSON Caveats

8.1.1. 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.

8.1.2. 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.

9. GDB Server

T-EMU (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, as a stand-alone command line tool, and as a separate command in the T-EMU 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.

Note

C++ libraries are in general not guaranteed to be API stable. A simple stand-alone C wrapper may be provided in the future.

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

--paths-file [path]	A T-EMU CLI script whose purpose is to set any needed paths.
--machine-file [path]	A T-EMU 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.

Tip

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.

Important

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.

Tip

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.

Warning

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.

9.1. Example

Starting the standalone GDB server:

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

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
```

10. 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 T-EMU, so it is possible to use the wrappers by simply importing the packages from your python script.

Note

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.support.cpu*.

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 T-EMU 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.

Tip

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.

11. Examples

11.1. 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/`.

11.1.1. CLI

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

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

11.1.2. 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");
```

11.2. 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.

11.3. 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 <stdio.h>

int
main(int argc, const char *argv[argc])
{

  temu_initSupportLibrary();
  temu_initPathSupport("temu");

  temu_loadPlugin("libTEMULeon2.so");
  temu_loadPlugin("libTEMULeon2SoC.so");
  temu_loadPlugin("libTEMUMemory.so");
  temu_loadPlugin("libTEMUConsole.so");

  void *Cpu = temu_createObject("Leon2", "cpu0");
  void *L2SoC = temu_createObject("Leon2SoC", "leon2soc0");
  void *MemSpace = temu_createObject("MemorySpace", "mem0");
  void *Rom = temu_createObject("Rom", "rom0");
  void *Ram = temu_createObject("Ram", "ram0");
  void *Console = temu_createObject("Console", "tty0");

  // Allocate space for ROM and RAM
  temu_propWriteU64(Rom, "size", 0x80000, 0);
  temu_propWriteU64(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
  temu_connect(Cpu, "memAccess", MemSpace, "MemAccessIface");
  temu_connect(Cpu, "memory", MemSpace, "MemoryIface");
```

```
temu_connect(MemSpace, "invalidaccess", Cpu,
             "InvalidMemAccessIface");

temu_connect(L2SoC, "irqControl", Cpu, "IrqIface");
temu_connect(Cpu, "irqClient", L2SoC, "IrqClientIface");
temu_connect(L2SoC, "queue", Cpu, "EventIface");

// 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.
temu_connect(Console, "serial", L2SoC, "UartAIface");
temu_connect(Console, "queue", Cpu, "EventIface");

// 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.
if (temu_checkSanity(0)) {
  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, "myobsw.srec")

// 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;
}
```