TEMU
Tutorial

Version 5.0, 2026-11-20

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

Table of Contents

1. Getting Started . . . 3
1.1. Installation . . . 3
1.2. Command Line Interface . . . 4
2. General Usage . . . 9
2.1. Opening a Project . . . 9
2.2. Loading Software . . . 9
2.3. Running and Stepping . . . 10
2.4. Scheduler Quanta and Domains . . . 10
2.5. Inspecting Objects . . . 12
2.6. Working with Memory . . . 13
2.7. Saving and Restoring State . . . 13
2.8. Running Project Targets . . . 13
3. Embedding . . . 14
3.1. TEMU CMake Support . . . 14
3.2. Embedded Simulator . . . 14
4. First Plugin . . . 17
5. Memory Mapped Devices . . . 22
6. Events . . . 26
7. Raising Interrupts . . . 28
8. Memory Access . . . 30
9. Bus Models . . . 32
9.1. Serial Ports . . . 32
9.2. Signals . . . 35
9.3. CAN . . . 37
9.4. Ethernet . . . 40
9.5. MIL-STD-1553 . . . 43
9.6. SpaceWire . . . 48
10. Physical Networks . . . 50
10.1. Ethernet NAT Bridge . . . 51
11. Notifications . . . 53
12. Scripting . . . 54
13. Adding Custom Commands . . . 55
13.1. TScript . . . 56
13.2. Python . . . 56

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

Chapter 1. Getting Started

This is the tutorial for the TEMU emulator. It has been tested against TEMU v5.0.

A note of nomenclature:

- Shell prompts (e.g. bash) are prefixed with `$`.
- Emulator prompt is prefixed with `temu>`.

1.1. Installation

TEMU binaries are available for download at the TEMU website. For the tutorial, it is assumed that you use a nightly build. Visit the website and click on the *Downloads* tab. You should then be able to download the TEMU nightly release.

There are two variants, one with asserts enabled, and one without asserts. The assert version will catch many errors at runtime and exit your program. This is especially useful during development. The other version is more high performant and is more suitable for deployment.

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

Installing the package is done as follows:

RPM

```
$ rpm -ivh temu-<vers>-generic-Linux-x86_64-<date>-<rev>.rpm
```

DEB

```
$ dpkg -i temu-<vers>-generic-Linux-x86_64-<date>-<rev>.deb
```

The default installation location will be `/opt/temu/<vers>/`.

To run TEMU, you need a valid license file. Before continuing with the tutorial, you should ensure that a license is installed. If not, evaluation license can be requested from the TEMU website.

Licenses are installed with the TEMU command line:

```
$ temu --install-license <license-file>
```

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

1.2. Command Line Interface

It is recommended that TEMU is added to your PATH variable, e.g.:

```
$ PATH=/opt/temu/{temu-version}/bin:$PATH
```

Then create a TEMU project for the rest of the tutorial:

```
$ temu --init-project tutorial
$ cd tutorial
```

The project contains a `temu-project.yaml` file, startup scripts in `scripts/temu/`, and a default board component in `components/board.yaml`. The generated board component extends `at697f.yaml`. For this tutorial, change `components/board.yaml` to extend `leon3.yaml`:

```
name: TutorialBoard
extends: leon3.yaml
```

TEMU is started by running the temu command:

```
$ temu
:info:: no such file: '/home/johndoe/.config/temu/init.temu' ①
:info:: no such file: './temu-init.temu' ②
temu>
```

① Missing user-global start-script, can be added by user.

② Missing directory local start script, can be added by user.

As can be seen, two messages are printed about missing start up scripts.

For now, simply type in the `quit` command and then return. Now, lets create a simple TEMU start-up script in the tutorial directory:

```
$ echo 'echo "Hello Emulator"' > temu-init.temu
$ cat temu-init.temu
echo "Hello Emulator"
$ temu
:info:: no such file: '/home/maho/.config/temu/init.temu'
Hello Emulator
temu>
```

As can be seen, the TEMU start-up script is now executed. We can now create a LEON3 device by loading the project. The project startup script imports component support, initializes the multi-level

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

scheduler, creates a `board0` component from `components/board.yaml`, and adds the component processor to the scheduler:

```
$ temu --project=temu-project.yaml
:info: board0-rom : map device at 0x00000000 - 0x00100000
:info: board0-ram : map device at 0x40000000 - 0x48000000
:info: board0-ftmctrl : map device at 0x80000000 - 0x80000100
:info: board0-apbuart : map device at 0x80000100 - 0x80000200
:info: board0-irqMp : map device at 0x80000200 - 0x80000300
:info: board0-gpTimer : map device at 0x80000300 - 0x80000400
:info: board0-ahbstat : map device at 0x80000f00 - 0x80001000
:info: board0-apbctrl : map device at 0x800ff000 - 0x80100000
:info: board0-ahbctrl : map device at 0xfffff000 - 0x00000000
0.000000: warning: board0-cpu : sanity check: dCache not connected
0.000000: warning: board0-cpu : sanity check: iCache not connected
0.000000: warning: board0-cpu : sanity check: machine not connected
:warning: board0-mem : sanity check: faultyHandlers not connected
:warning: board0-mem : sanity check: postTransaction not connected
:warning: board0-mem : sanity check: preTransaction not connected
:warning: board0-mem : sanity check: upsetHandlers not connected
:warning: board0-mem : sanity check: user1Handlers not connected
:warning: board0-mem : sanity check: user2Handlers not connected
:warning: board0-mem : sanity check: user3Handlers not connected
```

As can be seen above, when creating the LEON3 component, a number of messages are printed. The first sequence of info messages tell you that a device has been mapped into simulated memory. The next number of messages are warnings. They are emitted due to non-connected interface references in some device models.

As the default LEON3 component does not connect any cache models, the warnings are nothing to worry about in this case.

```
<time>: <severity> : <object> : <message>
```

The timestamp is the current time of the objects time source if it has one. The severity is one of several severity levels, including: debug, info, warning, error and fatal.



The fatal message type is combined with an abort call. Hence, the emulator will terminate after emitting the message. These messages are of great use when debugging models.

The object is the name of the object that emits the message. The message is any text string.

The warnings above are emitted by the objsys-check-sanity command. We can get help for this command by using the help command:

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

```
temu> help objsys-check-sanity
Command 'objsys-check-sanity'
    Check sanity of object system
Options:
    NONE
Aliases:
    objsys-check-sanity
```

If you type the objsys-check-sanity command, the same warnings will be printed. You can also use tab completion for commands.

Try:

objs `Tab` c `Tab`

And you will see that the command is completed.

You can get a full list of commands by typing `help` without any arguments.

Two interesting commands are: `class-list`, and `object-list`. These will list classes and objects that are registered with the TEMU object system. Another command, which is useful is the `class-info` command. The `class-info` command takes the argument `class=ClassName`. It prints out properties, interfaces and ports that the class provides. Go ahead, try out `class-info class=Component` or `class-info class=Leon3`.

Correspondingly, an object can be inspected by using the `object-info` command. Try `object-info obj=board0-cpu`.

As you may notice, argument names for commands, and certain argument values.

Although, commands are often shortened with nicer to type aliases, most commands have a full noun-verb name. Thus, it is possible to get all CPU inspection and setup commands by typing cpu-`Tab`. Execution itself should normally go through the top-level commands `run`, `step`, `trace`, and `stop`. They use the active scheduler, which is the multi-level scheduler in the tutorial project.

```
temu> cpu-<TAB>
Completions:
  cpu-disable-trap-events
  cpu-enable-trap-events
  cpu-reset
  cpu-set-pc
  cpu-set-reg
  cpu-show-regs
temu> cpu-show-regs cpu=board0-cpu
%g0 = 0x00000000
%g1 = 0x00000000
```

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

```
%g2 = 0x00000000
%g3 = 0x00000000
%g4 = 0x00000000
```

Commands in TEMU are structured around the form:

```
noun-verb-command-name arg=value arg2=value
```

Note that arguments are typed (e.g. object name, string, file name, integer, real number et.c.). This means that if you type tab for completing an object argument, the object name will be completed. The same goes for file names.

Now, let us try something more fun. Copy the rtems-hello.c file to your tutorial directory. Compile it and convert it to a ROM image.

```
$ cp /opt/rtems-4.10/src/samples/rtems-hello.c ./
$ PATH=/opt/rtems-4.10/bin:$PATH
$ PATH=/opt/mkprom2:$PATH
$ sparc-rtems-gcc rtems-hello.c -o rtems-hello.elf
$ mkprom2 rtems-hello.elf -o rtems-hello.prom
$ temu --project=temu-project.yaml
temu> load obj=board0-mem file=./rtems-hello.prom
:info: board0-mem : loading big-endian sparc 32-bit
:info: board0-mem : loading segment 1/1 0x00000000 - 0x00015de0 pa = 0x00000000
temu> sched.start workers=1
temu> run time=10.0
MkProm2 boot loader v2.0
Copyright Gaisler Research - all rights reserved
system clock : 50.0 MHz
baud rate : 19171 baud
prom : 512 K, (2/2) ws (r/w)
sram : 2048 K, 1 bank(s), 0/0 ws (r/w)
decompressing .text to 0x40000000
decompressing .data to 0x40022200
decompressing .jcr to 0x400233c0
starting rtems-hello.elf
:target/warning: board0-irqMp : read unknown register @ 0x80000220
Hello World
Hello World over printk() on Debug console
1.276503: info: board0-cpu : error mode due to 'trap_instruction' trap
(tt = 0x80) @ 0x40004a74
```

As you see, the output on the serial port is displayed on the console output. There are two items to note in the output above. Firstly, an error message about a read from an unknown register. This is expected behaviour, as there are two IRQ controllers from Cobham Gaisler, but in RTEMS (and Linux), the same driver is used for both the controllers. The message can be ignored in this case.

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

Secondly, at 1.276503 seconds, board0-cpu enters error mode (which means the CPU halts execution due to a trap being raised while traps are disabled), the type of trap is 'trap_instruction', with the %tbr.tt field being 0x80. The instruction causing the trap is located at address 0x40004a74. As the emulator comes with an embedded assembler and disassembler, we can disassemble this instruction:

```
temu> disassemble cpu=board0-cpu addr=0x40004a74
40004a74 040004a74 91d02000 ta %g0 + 0
```

If you disassemble the hello binary with the objdump tool, we can locate the function that executes the trap instruction:

```
$ sparc-rtems-objdump -d rtems-hello.elf | less
```

In less you can search by typing `/<search string>`, so typing `/40004a74` gives us a hit in the syscall function and will display the same code you can see with the built-in TEMU disassembler. The trap here is actually fully expected, as the rtems-hello application call the exit(0) at the end. To simulate the normal behaviour of exit(), RTEMS ensures that the CPU enters error mode.

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

Chapter 2. General Usage

Before writing models or embedding TEMU in another simulator, it is useful to know the normal workflow for using an existing project. This section uses the tutorial project created in the previous chapter. It assumes that `components/board.yaml` extends `leon3.yaml` and that the project startup scripts create the `board0` component.

2.1. Opening a Project

Start TEMU from the project directory:

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

The project file applies the project search paths, runtime directories, and startup scripts. In the generated tutorial project, `scripts/temu/imports.temu` imports component support and `scripts/temu/startup.temu` initializes the scheduler, creates `board0`, and adds the board CPU to the scheduler:

```
temu> object-list
```

The component owns the board objects, so the CPU and memory space are named `board0-cpu` and `board0-mem`. The scheduler object is named `sched`.

The relevant startup commands are:

```
init-scheduler variant=multi-level
sched.set-stop-mode mode=cli
Component.new name=board0 file=board.yaml
sched.add-cpu cpu=board0-cpu
```

TEMU 5 runs target processors through a scheduler. If you create a project manually, call `init-scheduler` before running software and add every processor that should execute with `sched.add-cpu`. The generated project startup script does this for the tutorial board.

2.2. Loading Software

Target software is loaded into an object that implements the memory interface. For the tutorial LEON3 board, use the component memory space:

```
temu> load obj=board0-mem file=./rtems-hello.prom
```

If the image contains an entry point, TEMU will use it when the loader supports that format. You

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

can inspect or adjust the CPU state before running:

```
temu> regs cpu=board0-cpu
temu> cpu-set-pc cpu=board0-cpu pc=0x40000000
```

2.3. Running and Stepping

The tutorial project uses the multi-level scheduler. After loading software and finishing scheduler configuration, start the scheduler once:

```
temu> sched.start workers=1
```

Then use the normal execution commands. `run`, `step`, `trace`, and `stop` are the preferred user-facing commands, also when the active scheduler is the multi-level scheduler. They route execution through the scheduler that was configured during startup:

```
temu> run time=1.0
temu> run cycles=100000
```

Use `step` when you need instruction-level control for one processor. Other processors in the scheduler may still run when the step crosses a scheduler quantum:

```
temu> step cpuidx=0 steps=1
temu> trace cpuidx=0 steps=5
```

Use `stop` to request that a running multi-level scheduler stops.

When a program stops because of a trap, breakpoint, or error mode, inspect the CPU and disassemble around the program counter:

```
temu> regs cpu=board0-cpu
temu> disassemble cpu=board0-cpu addr=0x40000000 count=8
```

Use the scheduler object explicitly in startup scripts for scheduler configuration, and use the top-level execution commands for normal interactive work.

2.4. Scheduler Quanta and Domains

The scheduler quantum controls how far processors can run before the scheduler synchronizes time and dispatches pending synchronized events. Larger quanta reduce scheduler overhead, while smaller quanta make cross-processor and event timing synchronization more frequent.

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



The multi-level scheduler is the preferred scheduler for new TEMU 5 projects. It is newer and less tested than the single-level scheduler. For systems that would use only one MLS domain, the single-level scheduler may still be faster.

For the default single-domain tutorial board, adding the CPU is enough:

```
temu> init-scheduler variant=multi-level
temu> Component.new name=board0 file=board.yaml
temu> sched.add-cpu cpu=board0-cpu
temu> sched.start workers=1
```

If no domains are configured, the multi-level scheduler creates a default domain when it starts. To choose a CPU quantum explicitly, create a leaf domain and assign the CPU to it before `sched.start`:

```
temu> init-scheduler variant=multi-level
temu> Component.new name=board0 file=board.yaml
temu> sched.add-cpu cpu=board0-cpu
temu> sched.add-domain id=0 quantum=1000
temu> sched.assign-cpu-domain cpu=board0-cpu domain=0
temu> sched.start workers=1
```

For systems with several processors, add every processor before starting the scheduler. For a normal multi-core board, the usual deterministic setup is to place all CPUs for that board in the same leaf domain and run that domain with one worker thread. This keeps the CPUs ordered by the scheduler instead of by host-thread timing:

```
temu> sched.add-cpu cpu=board0-cpu0
temu> sched.add-cpu cpu=board0-cpu1
temu> sched.add-domain id=0 quantum=1000
temu> sched.assign-cpu-domain cpu=board0-cpu0 domain=0
temu> sched.assign-cpu-domain cpu=board0-cpu1 domain=0
temu> sched.set-domain-workers id=0 workers=1
temu> sched.start workers=1
```

You can create more workers for higher performance, for example one worker per CPU in a large leaf domain. That trades determinism for speed because host-thread scheduling can affect the order in which runnable CPUs make progress.

Use explicit domains when groups of processors need their own synchronization quantum. A leaf domain owns processors. A parent domain synchronizes child domains at a coarser boundary:

```
temu> sched.add-domain id=100 quantum=50000
temu> sched.add-domain id=0 quantum=1000 parent=100
temu> sched.add-domain id=1 quantum=2000 parent=100
```

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

```
temu> sched.add-cpu cpu=board0-cpu0
temu> sched.add-cpu cpu=board0-cpu1
temu> sched.assign-cpu-domain cpu=board0-cpu0 domain=0
temu> sched.assign-cpu-domain cpu=board0-cpu1 domain=1
temu> sched.start workers=2
temu> sched.tree
```

In this example, `board0-cpu0` synchronizes at 1000-cycle leaf boundaries, `board0-cpu1` synchronizes at 2000-cycle leaf boundaries, and both domains meet at the 50000-cycle parent-domain boundary. Domains, CPU assignment, domain worker counts, and domain frequencies must be configured before `sched.start`. Use `sched.set-domain-workers id=… workers=…` and `sched.set-domain-frequency id=… frequency=…` for those optional domain settings.

For multi-board simulations, a common deterministic layout is one leaf domain per board. Put the CPUs of each board in that board's leaf domain, run each board domain with one worker, and connect the boards through delay/replay-capable links so cross-board traffic is recorded and delivered at deterministic synchronization points. The domain quantum should be chosen together with the link latency; cross-domain communication needs enough delay that delivery can be ordered at domain boundaries.

Multi-board systems do not have to use separate domains. If deterministic ordering is more important than parallelism, or if the board-to-board communication is too tightly coupled for delayed links, put all board CPUs in a single leaf domain and run it with one worker.

2.5. Inspecting Objects

The object system is the main way to inspect a running simulation. Use `object-list` to find object names and `info` to inspect one object:

```
temu> object-list
temu> info obj=board0-cpu
temu> info obj=board0-mem
```

Use `class-info` when you need to know which properties, interfaces, and ports a class provides:

```
temu> class-info class=Leon3
temu> class-info class=MemorySpace
```

Properties can also be read and written directly:

```
temu> board0-cpu.freq
temu> board0-cpu.freq = 50000000
```

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

2.6. Working with Memory

The memory space can print its current mappings:

```
temu> pp obj=board0-mem
```

This is useful before adding a device model, but it is also useful during normal usage because it shows which address ranges are occupied by ROM, RAM, and peripherals.



The `pp` command is an abbreviation for *pretty-print*, it works for every model that implements the `temu_ObjectIface::printObject` function.

The CPU-specific assembler and disassembler can be used for small experiments:

```
temu> board0-cpu.assemble instr="nop" addr=0x40000000
temu> disassemble cpu=board0-cpu addr=0x40000000 count=1
```

2.7. Saving and Restoring State

Snapshots let you preserve a configured or partially executed system. Project scaffolds place runtime files below `.temu/`, so this tutorial saves snapshots there:

```
temu> tsnap-write file=.temu/snapshots/after-load.tsnap
temu> run time=0.1
temu> tsnap-read file=.temu/snapshots/after-load.tsnap
```

The snapshot stores the object graph and properties, in a binary format.

2.8. Running Project Targets

Projects can define named targets for common workflows. If a project contains a target named `hello`, start it directly from the shell:

```
$ temu --project=temu-project.yaml --target=hello
```

Use targets for repeatable usage flows such as loading one software image, setting up a debugger, or running a test script. Keep ad-hoc interactive experiments in the shell until the command sequence is stable enough to move into a project startup script.

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

Chapter 3. Embedding

Now that you know how to use TEMU in stand alone mode, we will look on how we can integrate TEMU in a simulator. We will do this by creating our own program for driving TEMU. It is easy enough to create the simulator that drives TEMU. In fact, the command line interface that was used in the previous section, is a relatively simple program that just invokes the TEMU API.

3.1. TEMU CMake Support

TEMU installs a CMake package configuration file. External projects should point CMake at the TEMU installation or build tree with `-DTEMU_ROOT=⋯`, then use `find_package(TEMU REQUIRED CONFIG)`. The package defines imported targets such as `temu::support`, `temu::scheduler`, `temu::snapshots`, and `temu::flux`.

For a project with one embedded executable, the relevant CMake pattern is:

```
cmake_minimum_required(VERSION 3.20)
project(MyTemuSimulator C)

find_package(TEMU REQUIRED CONFIG)

add_executable(simulator simulator.c)
target_include_directories(simulator PRIVATE ${TEMU_INCLUDE_DIRS})
target_link_libraries(simulator PRIVATE
  temu::support
  temu::scheduler
  temu::snapshots
  temu::flux)
```

Configure and build your project with:

```
$ cmake -S . -B build -DTEMU_ROOT=/opt/temu/{temu-version}
$ cmake --build build
```

For plug-ins, build a `MODULE` library and link it with `temu::support`.

3.2. Embedded Simulator

Create a new file in your tutorial directory named `simulator.c`. Then write the following program in that file:

The example loads the tutorial project, starts the multi-level scheduler, and then runs through the multi-level scheduler API. It also shows the single-level scheduler fallback for projects that still use it. This matches the command-line flow: configure the scheduler first, start it once when using the multi-level scheduler, and then execute the target through the scheduler.

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

```
#include "temu-c/Flux/Flux.h"
#include "temu-c/Scheduler/MultiLevelScheduler.h"
#include "temu-c/Snapshots/Snapshots.h"
#include "temu-c/Support/CommandLine.h"
#include "temu-c/Support/Events.h"
#include "temu-c/Support/Init.h"
#include "temu-c/Support/Loader.h"
#include "temu-c/Support/Objsys.h"
#include "temu-c/Support/Project.h"
#include "temu-c/Support/Scheduler.h"
#include "temu-c/Support/Time.h"
#include <stdint.h>

int
main(int argc, const char *argv[])
{
  // Initialise TEMU support library and additional support,
  // this is mandatory, it will
  // among other things ensure you have a valid license.
  temu_initConfigDirs();
  temu_initSupportLib();
  temu_initScheduler();
  temu_initSnapshotLib();
  temu_initFluxLib();
  // Load the tutorial project. The project startup scripts initialize the
  // scheduler, create board0 from components/board.yaml, and add its CPU to
  // the scheduler.
  if (temu_loadProject("temu-project.yaml", NULL)) {
    return 1;
  }

  // board0-mem is created by the board0 component.
  void *board0_mem = temu_objectForName("board0-mem");
  // You can load an image to any object implementing MemoryIface
  temu_loadImage(board0_mem, "rtems-hello.prom");
  // Start and run through the multi-level scheduler used by the tutorial
  // project. The fallback shows how the same embedded simulator can support
  // projects that still use the single-level scheduler.
  temu_TimeSource *mls = temu_getMultiLevelScheduler();
  temu_Sched *sched = temu_getMultiLevelSchedulerInstance();
  if (mls && sched) {
    if (temu_schedStart(sched, 1)) {
      return 1;
    }

    temu_StopInfo stop = {0};
    uint64_t freq = temu_eventQueueGetFreq(mls);
    uint64_t cycles = (uint64_t)temu_secsToCycles(10.0, freq);
```

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

```
    if (temu_schedRunFor(sched, cycles, &stop)) {
      return 1;
    }
  } else {
    temu_runForTime(10.0);
  }
  return 0;
}
```

When you run the program above, you should see the same output as in the previous section.

Now you know the basics of using the TEMU APIs and compiling. We will move on to plug-in development in the next couple of sections.

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

Chapter 4. First Plugin

The main way of extending TEMU with additional models is to write plug-ins. In-fact, virtually all TEMU models are defined in their own plug-in that needs to be imported by the TEMU plug-in system. Plug-ins are shared libraries on Linux and bundles on macOS. Build plug-ins as CMake `MODULE` libraries and link them with `temu::support`:

```
cmake_minimum_required(VERSION 3.20)
project(MyTemuPlugin C)

find_package(TEMU REQUIRED CONFIG)

add_library(first MODULE first.c)
target_include_directories(first PRIVATE ${TEMU_INCLUDE_DIRS})
target_link_libraries(first PRIVATE temu::support)
```

Configure and build the plug-in project with:

```
$ cmake -S . -B build -DTEMU_ROOT=/opt/temu/{temu-version}
$ cmake --build build
```

The hello world plug-in is a very small:

```
#include <stdio.h>
#include "temu-c/Support/Objsys.h"
#include "temu-c/Support/config.h"

TEMU_PLUGIN_MD_DEF_V1 = {.MDVersion = 1,
                         .Vendor = "Terma",
                         .PluginName = "TutorialHelloWorld",
                         .Major = 1,
                         .Minor = 0,
                         .Patch = 0,
                         .ABI_MAJOR = TEMU_MAJOR_VERSION,
                         .ABI_MINOR = TEMU_MINOR_VERSION,
                         .ABI_PATCH = TEMU_PATCH_VERSION,
                         .Flags = 0,
                         .Revision = TEMU_GIT_REVISION,
                         .License = "Tutorial",
                         .Description = "Tutorial hello-world plugin",
                         .Copyright = "Terma",
                         .ExportControl = "None"};

TEMU_PLUGIN_INIT
{
```

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

```
  printf("Hello TEMU plug-in\n");
}
```

After the plug-in has been compiled, you can load it in the command line interface as follows:

```
temu> import first
Hello TEMU plug-in
```



Subsequent imports will not re-run the plug-in initialisation function. The example also defines plug-in metadata with `TEMU_PLUGIN_MD_DEF_V1`. TEMU reads this metadata when loading the plug-in and uses it to identify the plug-in, its version, the TEMU ABI version it was compiled against, licensing information, and descriptive text. New plug-ins should define this metadata next to the plug-in initialisation function:

```
TEMU_PLUGIN_MD_DEF_V1 = {.MDVersion = 1,
                         .Vendor = "Terma",
                         .PluginName = "TutorialHelloWorld",
                         .Major = 1,
                         .Minor = 0,
                         .Patch = 0,
                         .ABI_MAJOR = TEMU_MAJOR_VERSION,
                         .ABI_MINOR = TEMU_MINOR_VERSION,
                         .ABI_PATCH = TEMU_PATCH_VERSION,
                         .Flags = 0,
                         .Revision = TEMU_GIT_REVISION,
                         .License = "Tutorial",
                         .Description = "Tutorial hello-world plugin",
                         .Copyright = "Terma",
                         .ExportControl = "None"};
```

Use the `TEMU_PLUGIN_INIT` macro to declare the name and signature for the plug-in initialisation function. The advantage of using this (in contrast to e.g. the compilers library constructor / init attributes) is that `TEMU_PLUGIN_INIT` is guaranteed to work on all platform that TEMU supports.

In addition, the `TEMU_PLUGIN_INIT` macro will ensure the correct name-mangling is done if the plug-in is compiled as a C++ plug-in.

The key use of the plug-in initialisation function is to register classes with the TEMU object system. We will modify the plug-in file as follows:

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

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

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

typedef struct {
  temu_Object Super;
  uint32_t a;
} MyModel;

static void*
create(const char *name, int argc, const temu_CreateArg argv[])
{
  MyModel *model = malloc(sizeof(MyModel));
  memset(model, 0, sizeof(MyModel));
  return model;
}

static void
dispose(void *obj)
{
  MyModel *model = (MyModel*)obj;
  free(model);
}

TEMU_PLUGIN_MD_DEF_V1 = {.MDVersion = 1,
                         .Vendor = "Terma",
                         .PluginName = "TutorialFirstPlugin",
                         .Major = 1,
                         .Minor = 0,
                         .Patch = 0,
                         .ABI_MAJOR = TEMU_MAJOR_VERSION,
                         .ABI_MINOR = TEMU_MINOR_VERSION,
                         .ABI_PATCH = TEMU_PATCH_VERSION,
                         .Flags = 0,
                         .Revision = TEMU_GIT_REVISION,
                         .License = "Tutorial",
                         .Description = "Tutorial object model plugin",
                         .Copyright = "Terma",
                         .ExportControl = "None"};

TEMU_PLUGIN_INIT
{
  temu_Class *c = temu_registerClass("MyModel", create, dispose);
  temu_addProperty(c, "a", offsetof(MyModel, a), teTY_U32, 1, NULL, NULL,
                   "property a");
}
```

Restart TEMU, and re-load your plug-in with the "import" command. Now, if you type class-list, you will see your new model. If you type `class-info class=MyModel`, you will see the meta info you

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

registered with the class (interfaces, properties and ports). The temu_registerClass() function takes as parameters, the name of the class (which must be unique), a constructor that is responsible for allocating and intialising objects of the class, and a destructor that is responsible for freeing the object and cleaning up any resources it may have acquired (closing open files etc). The next step is to create an instance of the class:

```
temu> MyModel.new name=foo
```

Typing "object-list" will let you see that foo is created with the type MyModel. Go ahead and inspect your now created object:

```
temu> object-info obj=foo
foo : MyModel <>
a : uint32_t = 0 = 0x0
object.timeSource : <<object>> = 0
```

As you can see, the property you registered has the value 0. This is expected as the first thing done in the constructor was to clear the object returned by malloc.

If you look at the signature of the create function, you see that it takes the object name, an argument count and an argument array as parameters. The name will be the name you pass to the name parameter of the create-object function. The argc/argv values are set using the optional "args" parameter. The args parameter is a string, with key-value pairs, but as the command interpreter also use key-value pairs, the syntax is slightly different. Using the args can be done as follows:

```
temu> MyModel.new name=bar baz=42 foobar=42.0 blah=fourtytwo
```

At present, your model class does not handle the arguments, so lets add some code to handle this in the constructor function:

```
for (int i = 0; i < argc ; ++i) {
  if (!strcmp(argv[i].Key, "baz") && temu_isDiscrete(argv[i].Val)) {
    printf("create argument baz = %" PRIu64 "\n",
           temu_asUnsigned(argv[i].Val));
  } else if (!strcmp(argv[i].Key, "foobar") && temu_isReal(argv[i].Val)) {
    printf("create argument foobar = %f\n", temu_asDouble(argv[i].Val));
  } else if (!strcmp(argv[i].Key, "blah") && temu_isString(argv[i].Val)) {
    printf("create argument blah = %s\n", argv[i].Val.String);
  } else {
    printf("unknown argument '%s'\n", argv[i].Key);
  }
}
```

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

And add this to the plugin initialization to attach the arguments to the new command:

```
void *newCmd = temu_classGetCmd(c->Super.Class, "new");
assert(newCmd != NULL);
temu_classCmdAddParam(newCmd, "baz", teCOK_Int, 0, "Baz");
temu_classCmdAddParam(newCmd, "foobar", teCOK_Double, 0, "Foo Bar");
temu_classCmdAddParam(newCmd, "blah", teCOK_String, 0, "Blah");
```

Also add `#include <inttypes.h>`, to get the PRIu64 macro defined, and `#include <assert.h>` for `assert`. Now recompile the plug-in, restart TEMU and reload the plug-in. Try the `MyModel.new` command above with the same arguments again. Note that, although arguments are a useful way to simplify model creation scripts, arguments suffer from the lack of built-in documentation. You therefore need to document all arguments carefully for any users of your model. There is another way of configuring your models that is more self-documenting and that is to use the property mechanism in the object system. To write a property in an object, use the assignment operator. In the example given, you can say:

```
temu> foo.a = 42
temu> object-info obj=foo
foo : MyModel <>
a : uint32_t = 42 = 0x2a
object.timeSource : <<object>> = 0
```

As can be seen, executing the property assignment updated the property value that was registered with the MyModel class.

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

Chapter 5. Memory Mapped Devices

In the previous exercise, you implemented a small model and had a brief interaction with the TEMU object system. This section will expand on the previous knowledge and provide the first practical working model which expose memory mapped registers to actual target software.

We will start with the previous model and extend it with:

- Property Accessor For the Registered Property
- Memory Access Interface

For this exercise we should rename our model as `mmiomodel.c`, and when compiling ensure the output is named `libmmio.so`.

A property will have a default accessor created for itself if no accessor is provided, the previous example registered the property `a`, with the `MyModel`-class, but provided NULL instead of a write and read accessor.

The first step now is to add accessor functions:

Add `#include "temu-c/Support/Logging.h"` to your model include directives.

Implement the accessor functions:

```
static void
writeA(void *obj, temu_Propval pv, int idx)
{
  MyModel *model = (MyModel*)obj;
  temu_logInfo(model, "updating a from %u to %u", model->a,
    pv.u32);
  model->a = pv.u32;
}
static temu_Propval
readA(void *obj, int idx)
{
  MyModel *model = (MyModel*)obj;
  temu_logInfo(model, "reading a as %u", model->a);
  return temu_makePropU32(model->a);
}
```

Then change the line:

```
temu_addProperty(c, "a", offsetof(MyModel, a), teTY_U32, 1,
  NULL, NULL,
  "property a");
```

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

To:

```
temu_addProperty(c, "a", offsetof(MyModel, a), teTY_U32, 1,
  writeA, readA,
  "property a");
```

Now, recompile the plug-in and load it again and modify the property manually.

```
temu> MyModel.new name=foo baz=42 foobar=42.0 blah=bar3
temu> foo.a = 42
:info: foo : updating a from 0 to 42
temu> foo.a
:info: foo : reading a as 42
42
```

As you can see, when writing or reading a property, the property accessor will be called. This allows us to implement registers as properties with accessors (which especially simplify model testing, as one do not need to access registers from embedded software doing memory accesses). The example above also illustrated the use of the TEMU logging system.

To finalise our model for integration in the TEMU memory system, we need to add an implementation of the memory access interface. This interface is defined in `temu-c/Memory/Memory.h`, which needs to be included.

```
static void
memRead(void *obj, temu_MemTransaction *mt)
{
  temu_Propval pv;
  mt->Value = 0;
  switch (mt->Offset) {
  case 0:
    pv = readA(obj, 0);
    break;
  default:
    temu_logWarning(obj,
      "register at offset %d not implemented for reads",
      (int)mt->Offset);
     return;
  }
  mt->Value = temu_propValueU32(pv);
  mt->Cycles = 100;
}
static void
memWrite(void *obj, temu_MemTransaction *mt)
{
  temu_Propval pv = temu_makePropU32(mt->Value);
```

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

```
  switch (mt->Offset) {
  case 0:
    writeA(obj, pv, 0);
    break;
  default:
    temu_logWarning(obj,
      "register at offset %d not implemented for writes",
      (int)mt->Offset);
    return;
  }
  mt->Cycles = 100;
}

temu_MemAccessIface MemAccessIface = {
  NULL,
  memRead,
  memWrite,
  NULL
};
```

And add the following in the plug-in initialisation function:

```
temu_addInterface(c, "MemAccessIface",
                  TEMU_MEM_ACCESS_IFACE_TYPE,
                  &MemAccessIface, 0,
                  "memory access interface");
```

Compile the model, restart TEMU and run the following commands:

```
$ temu --project=temu-project.yaml
temu> import mmio
temu> MyModel.new name=foo
```

The model then needs to be mapped to memory. First, let's see which address is free in the LEON3 component that the project created. Some classes implement a pretty printing function in the optional object interface. This pretty printer can be invoked using the `object-print` command; the memory space class has one such implementation that prints out existing memory mappings:

```
temu> object-print obj=board0-mem
Memory Space 'board0-mem'
Address     Length           Object
----------- ---------------- ------
0x000000000            1 MiB board0-rom
0x040000000          128 MiB board0-ram
0x080000000          256   B board0-ftmctrl
```

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

```
0x080000100          256   B board0-apbuart
0x080000200          256   B board0-irqMp
0x080000300          256   B board0-gpTimer
0x080000f00          256   B board0-ahbstat
0x0800ff000            4 KiB board0-apbctrl
0x0fffff000            4 KiB board0-ahbctrl
```

As can be seen, after `board0-ahbstat`, there is some free space: `0x80000f00 + 256 = 0x80001000`, and this area is not mapped.

```
temu> board0-mem.map iface=foo:MemAccessIface addr=0x80001000 length=256
:info: foo : map device at 0x80001000 - 0x80001100
```

If you execute the object-print command again, you should now see the newly mapped device with the name `foo`.

We can use the built in assembler to create some instructions that can read and write to the register at offset 0 from the mapping, the offset is used in the switch. We create a small assembler program to trigger the memory accesses by using the built in assembler:

```
temu> board0-cpu.assemble instr="st %g1, [%g2 + 0]" addr=0
temu> board0-cpu.assemble instr="ld [%g2 + 0], %g3" addr=4
```

The program will by default end up at address 0 and 4. Enter the command `board0-cpu.disassemble va=0 count=10` and see that you got the correct instructions assembled in memory. The next step is to set the `%g2` register to the address of the device's register and then set some value to write to the register in `%g1`. After the device is configured, start the multi-level scheduler and use the top-level `step` command:

```
temu> board0-cpu.setReg reg="%g2" value=0x80001000
temu> board0-cpu.setReg reg="%g1" value=42
temu> sched.start workers=1
temu> step cpuidx=0 steps=2
:info: foo : updating a from 0 to 42
:info: foo : reading a as 42
```

Congratulations, your first TEMU MMIO device is now working!

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

Chapter 6. Events

A common task for memory mapped devices is that they need to simulate delays, TEMU is a discrete event simulation framework, and we can post events on the event queue provided by a CPU model. There are two types of "normal" events that can be posted, firstly stacked events, which are executed at the end of the instruction that triggered the event posting, and secondly timed events (with a delay in cycles, nanoseconds or seconds). Note that all events are rounded to a number of cycles and no event will be triggered in the middle of an instruction.

There are two items to take care of for event posting, firstly an event must be registered in order to provide snapshot capabilities, and secondly an event must be posted. The registration is done in the model's constructor, and is very easy, first define an event function with type `void (*evfunc)(temu_Event*)`:

```
void
evfunc(temu_Event *ev)
{
  temu_logInfo(ev->Obj, "event triggered");
}
```

Then add an `int64_t` event id to your model and then call the following in the constructor:

```
model->eventId = temu_eventPublish("mymodel.event", model, evfunc);
```

The event publication ensures that the event queues can be restored later on (after serialisation) without having to do anything else on your side.

There is one additional thing that needs to be done, your model needs to know where any event will be posted, and this is done by attaching a time source to your model. A time source property is already available in the `temu_Object` field, so the only thing needed is to connect to it in the command line:

```
temu> connect-timesource obj=foo ts=board0-cpu
```

In TEMU, time is provided by processor models. That means that there are multiple time-source if there are multiple processors in the system. A processor and the devices that uses the processor as a time source form a clock domain. This means that two events triggered by different processors may not be strictly ordered in apparent time, however they will be well ordered with other events triggered by the same processor.

The next step is to ensure that an event is actually posted by your model.



You should not enqueue an event that is already enqueued, if you do this, a warning will be emitted, and the already enqueued event will be dequeued before

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

> enqueuing the event again.

Go ahead, you can easily stack post an event in your model, by adding the following in your read and write accessors:

```
temu_eventPostStack(model->Super.TimeSource,
                    model->eventId, teSE_Cpu);
```

The last parameter is a synchronisation parameter, that are useful for multi-processor systems. In multi-processor systems, events can be posted so all processors will have the same time (plus a few cycles depending on the cycle cost time of the last instruction in each processor). To post a synchronised event, change it to teSE_Machine. However, for stacked events, machine posted events are events that are executed at the start of the next quanta. To step the processor through the multi-level scheduler, start the scheduler after the model has been configured and then use the top-level `step` command:

```
temu> sched.start workers=1
temu> step cpuidx=0 steps=2
0.000000: info: foo : updating a from 0 to 42
0.000002: info: foo : event triggered
0.000002: info: foo : reading a as 42
0.000004: info: foo : event triggered
```

As you notice, the first event is executed at time 0.000002 and not at time 0.000000, this is because events are triggered after an instruction has been executed, and that means that the CPU time is updated before the checking of the event queue.

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

Chapter 7. Raising Interrupts

A common need for MMIO models is to raise interrupts. The means for doing this depend on the type of interrupt interface that is used. For example, the signal interface can be used to raise and lower interrupts for devices that supports this, but more common is to use the direct interrupt interface as this offers some advantages as interrupts may be numbered.

The interrupt interface is defined in `temu-c/Models/IrqController.h`. There are two interfaces defined there, the `temu_IrqCtrlIface` and the `temu_IrqClientIface`. The client interface is for implementing dedicated interrupt controllers that needs to be notified about interrupt acknowledgement, while the controller interface is the way in which a device need to interface to an existing interrupt controller. Most users will be interested in the controller interface only.

Add the field:

```
temu_IrqCtrlIfaceRef irq;
```

To your model struct and register this field in the plug-in initialisation function:

```
temu_addProperty(c, "irq", offsetof(MyModel, irq),
  teTY_IfaceRef, 1,
  NULL, NULL, "interrupt controller");
```

To raise and lower interrupts, we need to insert code for this somewhere in the model. We can do this in the event handler (or in the register read/write handlers):

```
MyModel *model = (MyModel*)ev->Obj;
model->irq.Iface->raiseInterrupt(model->irq.Obj, 14);
```

The code above will raise and lower an interrupt in the connected interrupt controller (note that if no NULL checks are done, then this will result in a crash if you have not connected the interface).

The LEON3 and UT700 model comes with the IrqMp interrupt controller, in both cases it is named: `board0-irqMp`. It is easy to use the connect command to attach your MMIO model to the IRQ controller:

```
temu> connect a=foo.irq b=board0-irqMp:IrqIface
```

The connect command will create a link from the interface reference property named `irq` registered with the `temu_addProperty()` call to the named interface "IrqIface" implemented by the IrqMp class.

Note that if you need to figure out what interfaces a model implements, either look in the device

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

manual or use the `class-info` command.

Now we need some software to handle the interrupts. The straight forward method is to use RTEMS for this. We can work from the `rtems-hello.c` example you used earlier in the tutorial, and rework the `Init()`-function to ensure that we explicitly install an interrupt handler and to trigger the interrupt by writing to the register that was added in your device model:

```
rtems_isr
handleIRQ(rtems_vector_number vector)
{
  // Note, printf is interrupt driven and will hang
  // hang in ISRs unless you reenable IRQs here
  // thus we use printk which is polling.
  printk("External interrupt received with vector 0x%x\n",
         vector);
  }
}
rtems_task
Init(rtems_task_argument ignored)
{
  rtems_isr_entry oldHandle;
  rtems_status_code status = rtems_interrupt_catch(handleIRQ, 0x1e, &oldHandle);
  *(volatile unsigned*)(0x80000240) |= (1 << 14); // IRQ 14
  *(volatile unsigned*)(0x80001000) = 42;
  exit( 0 );
}
```

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

Chapter 8. Memory Access

A common task in many models is to access memory, reasons for this may include DMA transactions to be simulated, RMAP access from SpaceWire controllers, accessing send-lists in a bus controller etc.

There are two ways to access memory. One is to use the same memory access interface that you have implemented for register address decoding (by connecting to the memory space object), the other option is to use the MemoryIface, this interface provides two methods, readBytes and writeBytes. The nice thing with these is that they allow you to read out large blocks of data and potentially swap data to host endianess if needed for different data sizes.

The MemoryIface is provided by the MemorySpace class, so the easy way to get access to it is to go through an instance of that.

Working from the previous model, add an interface reference to the model:

```
temu_MemoryIfaceRef mem; // In the device struct
// In the plug-in init function:
temu_addProperty(c, "memory", offsetof(MyModel, mem),
    teTY_IfaceRef, 1,
    NULL, NULL, "memory space");
```

Now, let's extend the device model's register write function, we are going to treat the written value as an address and turn our device into a word inverter, we can write a new RTEMS application for triggering this process:

```
rtems_task
Init(rtems_task_argument ignored)
{
  volatile unsigned data[4] = {0x01234567, 0x89abcdef,
                               0xaaaaaaaa, 0x55555555};
  printk("data before: %x %x %x %x\n",
         data[0], data[1], data[2], data[3]);
  *(volatile unsigned*)(0x80001000) = (unsigned)&data[0];
  printk("data inverted: %x %x %x %x\n",
         data[0], data[1], data[2], data[3]);
  exit(0);
}
```

In the write handler, you can add a snippet like this:

```
  uint32_t buffer[4] = {0};
  // Read out 16 bytes of type word (1 << 2 bytes each)
  model->mem.Iface->readBytes(model->mem.Obj, &buffer[0],
```

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

```
  pv.u32,16, 2);
  temu_logInfo(model, "before invert: %x %x %x %x",
  buffer[0], buffer[1], buffer[2], buffer[3]);
  for (int i = 0 ; i < 4 ; ++i) {
    buffer[i] = ~buffer[i];
  }
  model->mem.Iface->writeBytes(model->mem.Obj, pv.u32, 16,
     &buffer[0], 2);
```

You need to connect the memory interface as well:

```
temu> connect a=foo.memory b=board0-mem:MemoryIface
```

Now, when running it, you will see that data is copied back and forth between the models and properly updated by the device model.

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

Chapter 9. Bus Models

TEMU comes with several transactional bus models bundled. This section will show how you can interface with these bus models from your device models. We will also introduce the concepts of ports and interface arrays in this section.

In general, responses from a bus device over the bus a message was received on should never be sent directly from the receiving function. The sending device and the bus model are still updating their state and need to finish executing the current send path first. Direct responses can otherwise recursively activate the bus model, or recursively activate a controller that starts another transmission before the previous transaction has unwound. Thus, responses should always be posted using events.

9.1. Serial Ports

The serial port is a byte oriented bus model, it provides the ability to send individual bytes to a receiver. TEMU is delivered with several serial port terminals, the most important ones are the Console and the `ConsoleUI` plugins. These provide a non-interactive serial port terminal that prints anything received to stdout (this is the one you have been using so far), and an interactive version respectively. The interactive serial port terminal in the `ConsolUI` plug-in can be used if Qt4 is installed on your system, it provides a console application with a degree of VT100 emulation. This console will send data back to TEMU in case you type in it.

The rest of this exercise focus on the creating of a device communicating via the serial port. The device will when receiving data, print the character using the TEMU logging functions, and will respond automatically with a character. The response will be executed from a stack posted event handler.

Create a new file with a model, we can call it MyTTY. To communicate with the serial port we need two fields in the device struct (in addition to the temu_Object field):

```
int64_t eventId;
temu_SerialIfaceRef uart; // Defined in temu-c/Bus/Serial.h
```

The constructor needs to publish the event:

```
tty->eventId = temu_eventPublish("tty.event", tty, evfunc);
```

The event function will send a byte to the other end of the serial port:

```
MyTTY *tty = (MyTTY*)ev->Obj;
if (tty->uart.Iface && tty->uart.Iface->write) {
  tty->uart.Iface->write(tty->uart.Obj, 'x');
```

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

```
  }
```

To implement the serial port interface you need to provide two functions to the interface, firstly a receive function, and secondly a (clear to send) cts function. CTS is experimental, so you do not need to fill it in at the moment.

```
static void
receive(void *obj, uint8_t data)
{
  MyTTY *tty = (MyTTY*)obj;
  temu_logInfo(obj, "received character '%c'", (char)data);
  temu_eventPostStack(tty->Super.TimeSource, tty->eventId, teSE_Cpu);
}
static void
cts(void *obj)
{
}
static temu_SerialIface SerialIface = {
  receive,
  cts,
};
```

The device need to be registered in the plug-in initialisation function:

```
TEMU_PLUGIN_INIT
{
  temu_Class *c = temu_registerClass("MyTTY", create, dispose);
  temu_addProperty(c, "uart", offsetof(MyTTY, uart), teTY_IfaceRef, 1,
    NULL, NULL, "Target serial port");
  temu_addInterface(c, "UART_A", TEMU_SERIAL_IFACE_TYPE,
    &SerialIface, 0, "serial port interface");
}
```

Then compile the model and load it in TEMU. A problem now is that the LEON3 and UT700 components only come with one APBUART instance. This is used by the normal serial port and is connected to the console (which is what has let you see the output of the RTEMS printf/printk calls). So, we cannot use this in case we want to print the received data in RTEMS. So we will add an extra UART to the system:

```
temu> ApbUart.new name=apbuart1
temu> object-prop-write prop=apbuart1.config.infiniteUartSpeed val=1
temu> object-prop-write prop=apbuart1.config.fifoSize val=1
temu> memory-map memspace=board0-mem addr=0x80001100 length=256 object=apbuart1
temu> connect a=board0-cpu.devices b=apbuart1:DeviceIface
temu> connect a=apbuart1.irqCtrl b=board0-irqMp:IrqIface
```

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

```
temu> connect-timesource obj=apbuart1 ts=board0-cpu
temu> connect a=board0-apbctrl.slaves b=apbuart1:ApbIface
```

As the extra APBUART is created and mapped into the system, we can now create the serial port device you have implemented:

```
temu> MyTTY.new name=tty
temu> connect-timesource obj=tty ts=board0-cpu
temu> connect a=tty.uart b=apbuart1:UartIface
temu> connect a=apbuart1.tx b=tty:UART_A
```

Note that we connect firstly `tty.uart` to `apbuart1:UartIface` and secondly `apbuar1.tx` to `tty:UART_A`, that is as the connection is bidirectional.

Now we can create a small RTEMS program with the following init function:

```
rtems_task
Init(rtems_task_argument ignored)
{
  *(volatile unsigned*)(0x80001108) = 0x3; // Enable RX and TX
  *(volatile unsigned*)(0x80001100) = 'y'; // Put byte in datareg
  // Loop until data is in RX
  while ((*(volatile unsigned*)(0x80001104) & 1) == 0) ;
  int data = *(volatile unsigned*)(0x80001100);
  printk("got byte: %c\n", data);
  exit( 0 );
}
```

Build it with `sparc-rtems-gcc` and `mkprom2` as usual and load this into the emulator. When running, you should see the logging message from the model and the output on `board0-apbuart` in the console.

As you probably noticed above, you need to connect a bidirectional bus connector twice. However, this is error prone and easy to forget, if you delete the second connect, the emulator will crash as your serial device does not check whether it is calling a NULL function. There are two ways to avoid this, firstly is to run the `objsys-check-sanity` command, after all devices have been created and connected, that will let you know that there is an unconnected interface reference in the `tty` object. The second method is to add a port.

A port is an association within a TEMU class that establish a relationship between an interface reference property and an interface. The connect command can then establish a reverse link automatically. In the command:

```
temu> connect a=tty.uart b=apbuart1:UartIface
```

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

```
temu> connect a=apbuart1.tx b=tty:UART_A
```

If there is a port associating `tty.uart` and `tty:UART_A`, and there is a second port that associates `apbuart1:UartIface` and `apbuart1.tx`, then we will automatically get a bidirectional connection. In the plug-in initialisation function add the following after the `uart` interface reference property is registered AND after the `UART_A` interface is registered:

```
temu_addPort(c, "uart", "UART_A", "Serial port");
```

Now you only need to execute one connect command to connect the APBUART with your serial port device (note that you can use either of the two connect commands).

9.2. Signals

Another simple bus interface is the `SignalIface`, the signal interface provide a simple single bit control involving the two `raise()` and `lower()` methods.

We will create a new model (you do not need any properties in it except the `temu_Object` field) and add 4 signal interfaces to it. While it is possible to add multiple interfaces with different names (`signal_a`, `signal_b`, et.c.). It is much more convenient to add multiple interfaces in an interface array, first define two functions that implements the SignalIface (in `temu-c/Bus/Signal.h`) but extend the functions with an integer index parameter, then add a macro to synthesise the actual implementation in multiple copies:

```
static void
raise(void *obj, int idx)
{
  temu_logInfo(obj, "signal %d raised", idx);
}
static void
lower(void *obj, int idx)
{
  temu_logInfo(obj, "signal %d lowered", idx);
}
#define DEF_SIGNAL(n)    \
  static void            \
  raise ## n (void *obj) \
  {                      \
    raise(obj, n);       \
  }                      \
  static void            \
  lower ## n (void *obj) \
  {                      \
    lower(obj, n);       \
  }
```

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

```
DEF_SIGNAL(0)
DEF_SIGNAL(1)
DEF_SIGNAL(2)
DEF_SIGNAL(3)

temu_SignalIface SignalIface[] = {
  { raise0, lower0 },
  { raise1, lower1 },
  { raise2, lower2 },
  { raise3, lower3 },
};
```

By implementing the interface like above, you can have a single implementation for the signal handling functions. This may of-course in some cases not be appropriate (e.g. if a signal need specific semantics), but in many cases it will work fine if the behaviour can be specialised via an index parameter.

To register an interface array with this device, use the following code:

```
TEMU_PLUGIN_INIT
{
  temu_Class *c = temu_registerClass("MyDevice", create, dispose);
  temu_addInterfaceArray(c, "signals", TEMU_SIGNAL_IFACE_TYPE,
                         &SignalIface[0], 4, sizeof(SignalIface[0]),
                        "signal interfaces");
}
```

Now we can connect to this from the GPGPIO device provided by the UT700 component. Before starting TEMU for this exercise, set the project component in `components/board.yaml` to extend `ut700.yaml`:

```
name: TutorialBoard
extends: ut700.yaml
```

```
$ temu --project=temu-project.yaml
temu> MyDevice.new name=sigdev
temu> connect a=board0-gpio.outSignals[0] b=sigdev:signals[0]
temu> connect a=board0-gpio.outSignals[4] b=sigdev:signals[2]
```

And provide a small RTEMS program for triggering the updates:

```
rtems_task
Init(rtems_task_argument ignored)
{
```

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

```
  *(volatile unsigned*)(0x80000908) = 0xffff; // Turn all pins to output
  *(volatile unsigned*)(0x80000904) = 1; // Set pin 0
  *(volatile unsigned*)(0x80000904) = 1 << 4; // Set pin 4, clear other pins
  exit( 0 );
}
```

When running the program we will see first how the pins are updated when set as output pins, and secondly how they are toggled when modifying the GPIODOR register.

9.3. CAN

CAN is a protocol, very common in automotive systems. It is also being used in several spacecraft and by hobbyists. CAN is a multi-node bus, meaning that a bus model is needed for it. TEMU comes with a model of the OpenCores CAN controller (with the AMBA P&P extensions for GRLIB compatibility) and a simple bus model. With simple, it is meant that the bus model broadcasts every message and does not provide any built-in routing or filtering. Broadcasting is potentially expensive for large CAN networks, and for these cases it is possible to replace the simple CAN bus model with a custom one.

You need to implement three functions in a CAN device: connected, disconnected and receive. The `temu_CanDevIface` is defined in `temu-c/Bus/Can.h`. The CAN connected function is called when the device has been connected to a CAN bus, it needs to ensure that the bus interface reference is saved in the device model:

```
void
connected(void *obj, temu_CanBusIfaceRef bus)
{
  MyDevice *dev = (MyDevice *)obj;
  dev->bus = bus;
  const char *busName = temu_nameForObject(bus.Obj);
  temu_logInfo(obj, "connected to CAN bus %s", busName);
}
```

The disconnected function is called whenever the user disconnects a CAN device from the bus (e.g. to simulate someone/something severing the cable). This function should clear the CAN bus property:

```
void
disconnected(void *obj)
{
  MyDevice *dev = (MyDevice *)obj;
  // Stop any in-flight events
  // Disconnect the bus iface reference
  dev->bus.Iface = NULL;
  dev->bus.Obj = NULL;
  temu_logInfo(obj, "disconnected");
```

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

```
}
```

The third function is the receive function which is responsible for handling incoming frames, note that the device probably need to filter out messages:

```
void
receive(void *obj, temu_CanFrame *frame)
{
  MyDevice *dev = (MyDevice *)obj;
  int rtr = temu_canIsRemoteTransmissionRequest(frame);
  int len = frame->Length;
  temu_logInfo(obj, "received CAN frame RTR = %d, len = %d", rtr,
               (int)frame->Length);
  if (len > 8)
    len = 8;
  if (!rtr) {
    for (int i = 0; i < len; ++i) {
      temu_logInfo(obj, "\tFrame data %u", (unsigned)frame->Data[i]);
    }
  }
  temu_canSetAck(frame);
}
```

Finally, the interface struct should be constructed and it needs to be registered in the CAN device plugin initialisation function:

```
static temu_CanDevIface CanIface = {
    connected,
    disconnected,
    receive,
    NULL,
};
```

Plug-in init:

```
TEMU_PLUGIN_INIT
{
  temu_Class *c = temu_registerClass("MyCANDevice", create, dispose);
  temu_addProperty(c, "bus", offsetof(MyDevice, bus), teTY_IfaceRef, 1, NULL,
                   NULL, "CAN bus object");
  temu_addInterface(c, "can_a", TEMU_CAN_DEV_IFACE_TYPE, &CanIface, 0,
                    "CAN interface");
}
```

This exercise uses the CAN controller provided by the UT700 component. Before starting TEMU, set

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

the project component in `components/board.yaml` to extend `ut700.yaml`:

```
name: TutorialBoard
extends: ut700.yaml
```

When running TEMU, there are two items that must be done in order to use the CAN bus and the new CAN device, the first is to create a CAN bus object (CAN is a multi-node bus and therefore needs a bus object for message routing). The SimpleCANBus class is available in the BusModels plugin:

```
$ temu --project=temu-project.yaml
temu> import BusModels
temu> SimpleCANBus.new name=canbus0
```

Then instantiate your CAN device and connect it and the `board0-canoc0` controller created by the UT700 component to the bus. This connection is done using the `can-connect` command:

```
temu> MyCANDevice.new name=candev0
temu> can-connect bus=canbus0:CanBusIface dev=candev0:can_a
temu> can-connect bus=canbus0:CanBusIface
dev=board0-canoc0:CanDevIface
```

The last thing is to have some embedded software that can be used to control the CAN OC controller:

```
struct CAN_OC_Device {
  uint8_t Ctrl;
  uint8_t Command;
  uint8_t Status;
  uint8_t Interrupt;
  uint8_t AcceptCode;
  uint8_t AcceptMask;
  uint8_t BusTiming[2];
  uint8_t unused0[2];
  uint8_t TxID[2];
  uint8_t TxData[8];
  uint8_t RxID[2];
  uint8_t RxData[8];
  uint8_t unused1[1];

  uint8_t ClockDivider;
};
rtems_task
Init(rtems_task_argument ignored)
{
  volatile struct CAN_OC_Device *CAN_OC
```

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

```
    = (struct CAN_OC_Device*)0xfff20000;
  printk("OBSW: Will send a CAN message\n");
  CAN_OC->Ctrl = 0; // Exit reset mode and disable Irqs
  CAN_OC->TxID[0] = 0xaa;
  CAN_OC->TxID[1] = 0xe1; // Length is 1
  CAN_OC->TxData[0] = 42; // Payload
  CAN_OC->Command = 1; // Transmission request
  sleep(1);
  printk("OBSW: Just sent the CAN message\n");
  exit( 0 );
}
```

Now run the emulator and see the logging done by the OBSW and the CAN device.

9.4. Ethernet

TEMU Ethernet support is split into MAC, PHY, and link layers. The usual topology is:

```
MAC -> PHY -> EthernetLink -> PHY -> MAC
```

The MAC is the device model that target software controls, for example a board Ethernet controller. The PHY models link state, MDIO/MII registers, auto-negotiation, and forwarding to the Ethernet link. `EthernetLink` routes frames between connected PHY devices.

9.4.1. Implementing a MAC

A custom Ethernet controller should implement `temu_MACIface`, defined in `temu-c/Bus/Ethernet.h`. The interface callbacks are:

- `connected` and `disconnected`, called when the PHY is connected to or disconnected from a link.
- `up` and `down`, called when the Ethernet link state changes.
- `receive`, called by the PHY when a frame is delivered to the MAC.
- `getMAC` and `setMAC`, used by the PHY and link routing code to track the active MAC address.

The MAC model should keep an interface reference to a PHY:

```
typedef struct {
  temu_Object Super;
  temu_PHYIfaceRef phy;
  uint64_t mac;
} MyMAC;
```

The MAC interface forwards link events into the device model and exposes the currently configured address:

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

```
static void
connected(void *obj)
{
  temu_logInfo(obj, "ethernet connected");
}

static void
disconnected(void *obj)
{
  temu_logInfo(obj, "ethernet disconnected");
}

static void
up(void *obj)
{
  temu_logInfo(obj, "ethernet link up");
}

static void
down(void *obj)
{
  temu_logInfo(obj, "ethernet link down");
}

static int
receive(void *obj, temu_EthFrame *frame)
{
  MyMAC *mac = (MyMAC *)obj;
  /* Copy the frame into the receive DMA path and raise the device IRQ. */
  return 0;
}

static uint64_t
getMAC(void *obj)
{
  MyMAC *mac = (MyMAC *)obj;
  return mac->mac;
}

static void
setMAC(void *obj, uint64_t addr)
{
  MyMAC *mac = (MyMAC *)obj;
  mac->mac = addr;
}

static temu_MACIface MACIface = {
    connected, disconnected, up, down, receive, getMAC, setMAC};
```

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

When target software transmits a frame, the MAC model builds a `temu_EthFrame` and sends it through the connected PHY. The frame buffer contains the Ethernet frame data, including destination MAC, source MAC, EtherType or length field, payload, and space for the CRC bytes. The frame does not need a valid CRC unless the receiving model checks it:

```
static int
sendFrame(MyMAC *mac, temu_EthFrame *frame)
{
  if (!mac->phy.Iface || !mac->phy.Iface->send) {
    return 0;
  }

  return mac->phy.Iface->send(mac->phy.Obj, frame);
}
```

As for other bus models, avoid sending response frames directly from the receive path when the response is caused by an incoming frame. Post a TEMU event and send from that event instead, so the current Ethernet link activation can finish before another transmission starts.

The plugin registration should expose both the MAC interface and the PHY reference:

```
temu_addInterfaceReference(cls, "phy", offsetof(MyMAC, phy),
                           TEMU_PHY_IFACE_TYPE, 1, 0, NULL, NULL,
                           "PHY device.");

temu_addInterface(cls, "MACIface", TEMU_MAC_IFACE_TYPE, &MACIface, 0,
                  "Ethernet MAC interface.");
```

9.4.2. Using GenericPHY

For most simulated equipment, use `GenericPHY` instead of writing a PHY model. It implements `PHYIface` and `MDIOIface`, provides standard MII registers, forwards frames between the MAC and the link, tracks link-up state, and participates in auto-negotiation.

Create an Ethernet link, create a generic PHY, connect the MAC to the PHY, connect the PHY back to the MAC, and then connect the PHY to the link:

```
$ temu --project=temu-project.yaml
temu> import BusModels
temu> EthernetLink.new name=eth0
temu> GenericPHY.new name=phy0
temu> MyMAC.new name=mac0
temu> connect a=mac0.phy b=phy0:PHYIface
temu> connect a=phy0.macDevice b=mac0:MACIface
temu> eth-connect link=eth0 device=phy0:PHYIface
```

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

The object method is equivalent:

```
temu> eth0.connect device=phy0:PHYIface
```

`GenericPHY` accepts optional creation arguments to restrict advertised speeds:

```
temu> GenericPHY.new name=phy10 base10=1
temu> GenericPHY.new name=phy100 base100=1
temu> GenericPHY.new name=phy1000 base1000=1
```

`EthernetLink` can also capture traffic to a PCAPNG file:

```
temu> eth-enable-capture link=eth0 file=eth0.pcapng
```

For live inspection with Wireshark, use a named pipe:

```
temu> eth-enable-capture link=eth0 file=/tmp/temu-eth0.pcapng named-pipe=1
```

9.5. MIL-STD-1553

TEMU models MIL-STD-1553 as a bus with one active bus controller and up to 31 remote terminals. The bus model is available from the `BusModels` plugin as the `MilStd1553Bus` class. The bus controller drives the transaction phases, while remote-terminal models receive command and data phases and post their responses as events. The phased design is intentional: the bus model tracks whether the current word stream is a command, data, or status phase, so each device model can handle the phase it is given instead of reconstructing the whole transaction locally. This is especially important for RT-to-RT transfers. The receiving RT sees the receive command and then the data phase in the same way it would for a BC-to-RT transfer, while the transmitting RT sees the transmit command and replies with status and data. As a result, RT models that follow the phase interface handle RT-to-RT transfers without special-case code.

The MIL-STD-1553 interface is defined in `temu-c/Bus/MilStd1553.h`. A device model that can be connected to the bus must implement `temu_Mil1553DevIface`:

- `connected` is called when the device is connected to a bus address.
- `disconnected` is called when the device is removed from a bus address.
- `receive` is called for command, data, and status phases.
- `busEnteredIdle` is called on the bus-controller device when the bus enters the idle state, for example after a reset.

The remote-terminal example keeps the bus reference and the assigned RT address after connection. It also tracks whether the current command expects a following data phase:

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

```
  uint8_t expectData;
  uint8_t pendingBroadcast;
  uint8_t pendingSubAddr;
  uint8_t pendingWordCount;

  temu_Mil1553BusIfaceRef bus;
  uint16_t rtAddr;

  struct {
    uint8_t infiniteSpeed;
  } config;
```

The connect and disconnect callbacks store and clear the bus interface reference. This is the reference the RT uses later when it sends status or data words back to the bus:

```
void
connected(void *obj, temu_Mil1553BusIfaceRef bus, int rtAddr)
{
  GenericRTU *rtu = (GenericRTU *)obj;
  rtu->bus = bus;
  rtu->rtAddr = rtAddr;
  rtu->expectData = 0;
}

void
disconnected(void *obj, temu_Mil1553BusIfaceRef bus, int rtAddr)
{
  (void)bus;
  (void)rtAddr;
  GenericRTU *rtu = (GenericRTU *)obj;
  rtu->bus.Obj = NULL;
  rtu->bus.Iface = NULL;
  rtu->rtAddr = 0;
  rtu->expectData = 0;
}
```

Responses should be posted through TEMU events instead of being sent directly from the `receive` callback. This lets the bus transaction advance at either zero model delay or at a delay based on `temu_mil1553TransferTime`. It also prevents recursive bus activation while the current command or data phase is still being processed. The example sends a status word before the data phase for RT transmit commands:

```
void
txDataEvent(temu_Event *ev)
{
  GenericRTU *rtu = (GenericRTU *)(ev->Obj);
```

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

```
  // Send status first
  temu_Mil1553Msg msg;
  uint16_t statWord;

  msg = temu_mil1553CreateStatMsg(&statWord, rtu->rtAddr);
  if (!rtu->bus.Iface || !rtu->bus.Iface->send) {
    return;
  }
  rtu->bus.Iface->send(rtu->bus.Obj, rtu, &msg);

  // Then send data
  uint16_t sa = (rtu->cmd >> 5) & 0x1f;
  uint16_t wc = rtu->cmd & 0x1f;
  wc = wc == 0 ? 32 : wc;

  msg = temu_mil1553CreateDataMsg(rtu->txData[sa], wc);

  rtu->bus.Iface->send(rtu->bus.Obj, rtu, &msg);
}

void
statusEvent(temu_Event *ev)
{
  GenericRTU *rtu = (GenericRTU *)(ev->Obj);

  temu_Mil1553Msg msg;
  uint16_t stat;

  msg = temu_mil1553CreateStatMsg(&stat, rtu->rtAddr);
  if (!rtu->bus.Iface || !rtu->bus.Iface->send) {
    return;
  }
  rtu->bus.Iface->send(rtu->bus.Obj, rtu, &msg);
}
```

The `receive` callback handles the incoming bus phases. For RT receive commands it records the pending sub-address and waits for a data phase before sending status. For RT transmit commands it schedules the status and data response immediately. Broadcast receive commands do not produce a status response:

```
void
receive(void *obj, temu_Mil1553Msg *msg)
{
  GenericRTU *rtu = (GenericRTU *)(obj);

  if (msg->MsgTyp == teMT_Cmd) {
    receiveCommand(rtu, msg);
```

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

```
    return;
  }

  if (msg->MsgTyp == teMT_Data) {
    if (!rtu->expectData) {
      temu_logError(rtu, "unexpected MIL-STD-1553 data phase");
      return;
    }

    uint16_t words = msg->WordCount;
    if (words > rtu->pendingWordCount) {
      words = rtu->pendingWordCount;
    }
    memcpy(rtu->rxData[rtu->pendingSubAddr], msg->Data, words * 2);
    rtu->expectData = 0;

    if (!rtu->pendingBroadcast) {
      postStatus(rtu);
    }
  }
}
```

The TEMU 5 bus interface also has an explicit reset path. Bus-controller models call `reset` on the bus interface when they need to abort or recover a transaction, such as after a timeout. When the bus returns to idle, the bus model calls the controller device's `busEnteredIdle` callback with a `temu_Mil1553BusIdleInfo` record. An RT-only model can still provide the callback as a no-op or use it to clear any pending local state:

```
void
busEnteredIdle(void *obj, temu_Mil1553BusIdleInfo *idleInfo)
{
  (void)idleInfo;
  GenericRTU *rtu = (GenericRTU *)obj;
  rtu->expectData = 0;
}
```

Finally, the device interface must include all four callbacks and must be registered by the plugin:

```
temu_Mil1553DevIface MilBusDevIface = {
    .connected = connected,
    .disconnected = disconnected,
    .receive = receive,
    .busEnteredIdle = busEnteredIdle,
};
```

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

```
TEMU_PLUGIN_INIT
{
  temu_Class *cls = temu_registerClass("GenericRTU", create, dispose);

  temu_addProperty(cls, "config.infiniteSpeed",
                   offsetof(GenericRTU, config.infiniteSpeed), teTY_U8, 1, NULL,
                   NULL, "Infinite speed mode");

  temu_addProperty(cls, "rtAddr", offsetof(GenericRTU, rtAddr), teTY_U16, 1,
                   NULL, NULL, "Remote terminal address, set on connect.");

  temu_addProperty(cls, "milbus", offsetof(GenericRTU, bus), teTY_IfaceRef, 1,
                   NULL, NULL, "Milbus interface reference");
  temu_requireInterface(cls, "milbus", TEMU_MIL1553_BUS_IFACE_TYPE);

  temu_addInterface(cls, "Mil1553DevIface", TEMU_MIL1553_DEV_IFACE_TYPE,
                    &MilBusDevIface, 0, "Milbus device interface");
}
```

9.5.1. Bus Setup

Create the MIL-STD-1553 bus from the `BusModels` plugin, instantiate the tutorial RT, and connect it to an RT address:

```
$ temu --project=temu-project.yaml
temu> import BusModels
temu> import TutorialMilbusRTU
temu> MilStd1553Bus.new name=milbus0
temu> GenericRTU.new name=rt1
temu> milbus-connect bus=milbus0 rt=rt1 addr=1
```

If the board model provides a MIL-STD-1553 controller, connect it as the bus controller:

```
temu> milbus-setbc bus=milbus0 bc=board0-b1553brm
```

The object methods can also be used when that is more convenient:

```
temu> milbus0.connect rt=rt1 addr=1
temu> milbus0.setBC bc=board0-b1553brm
```

Disconnecting an RT is explicit:

```
temu> milbus-disconnect bus=milbus0 addr=1
```

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

The object passed to `milbus-setbc` must implement `Mil1553DevIface`. The bus controller is responsible for initiating command sequences and for resetting the bus when a transaction times out or is aborted. Remote terminals normally only send status and data responses to command phases that are addressed to them.

9.5.2. Timing

MIL-STD-1553 transfers run at 1 Mbit/s, and TEMU accounts for 20 bits per 16-bit word. The helper `temu_mil1553TransferTime(words)` returns the transfer time in nanoseconds. The tutorial RT has `config.infiniteSpeed` enabled by default so responses are posted on the current event stack. Disable it when the RT model should delay responses according to the modeled bus transfer time:

```
temu> rt1.config.infiniteSpeed = 0
```

9.6. SpaceWire

SpaceWire is another protocol supported by TEMU. Although the SpaceWire wire protocol is byte oriented, individual byte handling is in simulation terms only needed for simulating the SpW link state automata. The TEMU SpW model is simplified into a packet based model, however to properly connect in a SpW network, the SpW link state simulation must be implemented.

The SpW protocol uses the `temu_Buff` *copy-on-write* type, this is useful as a device may queue up buffers with virtually zero copying. The buff type also provides ability to shrink the buffer from the head or the tail,without any overhead. Removing bytes from the head is very useful in SpW for the simulation of wormhole routing. As noted, TEMU does not transfer control characters (NULL, FCT), so the state machine is simulated through the following functions of the SpaceWire port interface (`temu_SpwPortIface`):

- `void (signalLinkStateChange)(void Device, temu_SpwLinkState LinkState)`
- `temu_SpwLinkState (getOtherSideLinkState)(void Device)`

The idea is that the each device can communicate its change of state to the other side calling `signalLinkStateChange` function on the other side port and get the other side state calling `getOtherSideLinkState` function on the other side port. Therefore, each device can implement its custom state machine (i.e. customized version of how to control the way that the system enter the started state), based on its current state and the state changes notified by the other side.

From a software developer point there is not much interest in phases other then Run and Ready. As such, the GRSPW2 for example, supports only Ready, Connecting and Run states at present.

The GRSPW2 will stay in Ready state if the link is disabled.

It will go and stay into connecting state if allowed to connect (LS bit set to 1) or the autostart bit is set and the other end signal a change in state to connecting.

It will go into run state if receiving a run state or a connecting from the other end while being in

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

connecting state.

It will go into ready state if link is disabled.

It will go into connecting state if it is in run state and the other side signal a disconnection through any phase other then run or connecting.

A working SpaceWire terminal example is provided in the examples directory.

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

Chapter 10. Physical Networks

TEMU is not only intended to be used in isolation. The event system provides a mechanism in which the user can register to be notified in a normal TEMU event that data is available on a socket. The API for this is declared in `temu-c/Support/Events.h` and includes the functions: `temu_asyncSocketAdd()`, `temu_asyncSocketRemove()`, `temu_asyncTimerAdd()` and `temu_asyncTimerRemove()`. The socket functions allows for the registration of a socket listening function that is scheduled in the main emulator loop. The functions actually handles any file descriptor (including pipes). The use of this function is that the provided callback will be executed when other emulator events are executed, and thus everything that is safe to do there will be safe to do in the provided callback. Internally, the async APIs uses the epoll mechanism on LINUX and kqueue on macOS and other BSD based systems. It is relatively straight forward to extend the serial port device example with a device that speaks to a telnet connection instead of printing messages to stdout. This can be done by opening a TCP socket and then waiting for a connection when the time source has been connected (by implementing the relevant function in `temu_ObjectIface` (see `temu-c/Support/Objsys.h`)). This will block the timesource-set command until you have connected the telnet client). Note that sockets should be made non-blocking, or you risk hanging the read access.

```
#include "temu-c/Bus/Serial.h"
#include "temu-c/Support/Events.h"
#include "temu-c/Support/Logging.h"
#include "temu-c/Support/Objsys.h"
#include <errno.h>
#include <fcntl.h>
#include <stdint.h>
#include <unistd.h>

typedef struct {
  temu_Object Super;
  int fd;
  temu_SerialIfaceRef uart;
} MySerialDevice;

static
void asyncCb(void *data)
{
  MySerialDevice *d = (MySerialDevice*)data;
  uint8_t ch = 0;
  int res;
  do {
    res = read(d->fd, &ch, 1);
  } while (res < 0 && errno == EINTR);

  if (res == 1) d->uart.Iface->write(d->uart.Obj, ch);
}
static void
timeSourceSet(void *obj)
```

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

```
{
  MySerialDevice *d = (MySerialDevice*)obj;
  if (d->Super.TimeSource) {
    // Open socket
    // d->fd = ...
    fcntl(d->fd, F_SETFL, O_NONBLOCK);
    if (temu_asyncSocketAdd(d->Super.TimeSource, d->fd, TEMU_ASYNC_READ,
      asyncCb, d) != d->fd) {
      temu_logError(d, "could not add async listner");
    }
  }
}
```

10.1. Ethernet NAT Bridge

`NATBridge` connects a simulated Ethernet network to host TCP sockets. It is useful when target software should accept a TCP connection from the host without creating a TAP/TUN device or changing host network configuration.

The bridge is part of the `BusModels` plugin and acts as an Ethernet PHY on the simulated link. It has its own simulated MAC address and IPv4 address. Incoming host TCP connections are mapped to configured target IPv4 addresses and TCP ports inside the simulated network.

10.1.1. Setup

Create an Ethernet link and NAT bridge, configure the bridge address, connect it to the link, and add the TCP port mappings:

```
$ temu --project=temu-project.yaml
temu> import BusModels
temu> EthernetLink.new name=eth0
temu> NATBridge.new name=nat0
temu> nat0.set-mac mac=02:00:00:00:00:01
temu> nat0.set-ip ip=192.168.0.1
temu> eth-connect link=eth0 device=nat0:PHYIface
```

Connect the board Ethernet PHY to the same link. For a board MAC that uses a `GenericPHY`, this looks like:

```
temu> GenericPHY.new name=board0-phy0
temu> connect a=board0-eth0.phy b=board0-phy0:PHYIface
temu> connect a=board0-phy0.macDevice b=board0-eth0:MACIface
temu> eth-connect link=eth0 device=board0-phy0:PHYIface
```

If the board component already creates a PHY object, connect that object to `eth0` instead.

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

Add a NAT mapping for each TCP service that should be reachable from the host. This example maps host port `10023` to target address `192.168.0.10`, target port `23`:

```
temu> nat0.add-nat name=telnet internal-addr=192.168.0.10 internal-port=23 external-
port=10023
temu> nat0.start-nat
```

After the target software has configured its MAC and IP address and started the TCP service, connect from the host:

```
$ telnet localhost 10023
```

Use `pmap` to inspect the configured port mappings:

```
temu> nat0.pmap
```

10.1.2. ARP

The NAT bridge uses ARP to resolve target MAC addresses. If the target has not yet sent ARP traffic, ask the bridge to send an ARP request:

```
temu> nat0.do-arp-request ip=192.168.0.10
temu> nat0.parp
```

The target software must be configured so that `nat0` and the target Ethernet interface are on the same IPv4 subnet, or the target must route replies back to the NAT bridge address.

10.1.3. Notes

`NATBridge` is a TCP bridge, not a full host network adapter. It forwards configured TCP connections between host sockets and the simulated Ethernet network. Use `EthernetLink` capture when debugging packet-level behavior:

```
temu> eth-enable-capture link=eth0 file=eth0.pcapng
```

For line-oriented debug protocols, `enable-crlf-substitution` can replace incoming CRLF sequences with `\0`:

```
temu> nat0.enable-crlf-substitution
```

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

Chapter 11. Notifications

In additional to the normal timed and stacked events, it is possible to add modules that listen to special notifications. Such notifications include for example bus traffic notifications (which can be used to tap some of the bus models), and trap taken and return from trap notifications. Normally, notifications are disabled, but they can be enabled using certain programming interfaces.

Notification listening can be handle in plug-ins (plug-ins do not need to register any models) or they can be added in your own program that drives the emulator (see [notifications:index:::using-temu-in-a-simulator]). As an example, in order to handle traps and rett instructions (return from trap), you can register to receive two notifications as follows:

```
void
trapEntry(void *arg, temu_Object *source, void *notinfo)
{
  (void)arg;
  temu_TrapEventInfo *info = (temu_TrapEventInfo *)notinfo;
  temu_logInfo(source, "trap taken %x", info->TrapId);
}
void
trapExit(void *arg, temu_Object *source, void *notinfo)
{
  (void)arg;
  (void)notinfo;
  temu_logInfo(source, "rett executed");
}
```

Note that the actual type for the notinfo pointer should be documented in the respective manuals and it differ depending on the type of notification. The actual registration for subscription can be done as follows:

```
  void *board0_cpu = temu_objectForName("board0-cpu");
  temu_CpuIface *iface = temu_getInterface(board0_cpu, "CpuIface", 0);
  if (iface && iface->enableTrapEvents) {
    iface->enableTrapEvents(board0_cpu);
  }
  temu_subscribeNotification("temu.cpuTrapEntry", board0_cpu, NULL, trapEntry);
  temu_subscribeNotification("temu.cpuTrapExit", board0_cpu, NULL, trapExit);
```

There is a command to list all published notifications and possible source objects. Type "notification-list" in the command line to see these.

Note that notifications are in a global namespace (unlike timed events, which are associated with an object). This allows you to for example listen to all notifications with a certain event, without specifying the source object.

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

Chapter 12. Scripting

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

Chapter 13. Adding Custom Commands

In TEMU 2.2 a command API has been added. It is now possible to register commands using either the TEMU API or the python wrappers. A plug-in can include `temu-c/Support/CommandLine.h` and then register commands using the `temu_createCmd` function:

```
#include "temu-c/Support/Objsys.h"
#include "temu-c/Support/CommandLine.h"
#include "temu-c/Support/Logging.h"
#include "temu-c/Support/config.h"

typedef struct {
  int a;
} data_t;

int
mycmdfunc(void *ctxt)
{
  void *data = temu_cmdGetData(ctxt); // returns dataptr
  // Return pointer to obj in param "foo"
  void *obj = temu_cmdGetOptionAsObject(ctxt, "foo");
  (void)data;
  temu_logInfo(NULL, "mycommand executed with object %s",
               temu_nameForObject(obj));
  return 0;
}

data_t data;

TEMU_PLUGIN_MD_DEF_V1 = {.MDVersion = 1,
                         .Vendor = "Terma",
                         .PluginName = "TutorialCustomCommand",
                         .Major = 1,
                         .Minor = 0,
                         .Patch = 0,
                         .ABI_MAJOR = TEMU_MAJOR_VERSION,
                         .ABI_MINOR = TEMU_MINOR_VERSION,
                         .ABI_PATCH = TEMU_PATCH_VERSION,
                         .Flags = 0,
                         .Revision = TEMU_GIT_REVISION,
                         .License = "Tutorial",
                         .Description = "Tutorial command plugin",
                         .Copyright = "Terma",
                         .ExportControl = "None"};

TEMU_PLUGIN_INIT
{
  void *cmd = temu_createCmd("mycommand", mycmdfunc,
```

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

```
                                "some documentation string", &data);
  temu_cmdAddOption(cmd, "foo", teCOK_Object, 1, "object name", NULL);
}
```

13.1. TScript

13.2. Python

There are three methods in which Python scrips can be executed inside TEMU. The first option is to use the script-run command. The second option (which is experimental at time of writing) is to prefix a line with an exclamation mark '!'. This tells the command interpreter to execute the line as a Python script. The third option is to run temu in non-interactive mode by adding the flags `-run-script "foo.py"` when starting TEMU.

Python scripts are especially convenient for writing tests of device models (and you could also implement a device model in Python, but at present you need to manually wrap functions using the ctypes library, which is not convenient).

The Python wrappers are defined in `/opt/temu/5.0}/share/temu/wrappers/Python`, this directory is automatically added to the PYTHON_PATH by TEMU. It defines a number of packages based on ctypes, under the `temu.c` umbrella package. The current API is a simple wrapper of the TEMU C-API (but stripping the `temu_` prefix of all functions as these become redundant with python packages), note that not all APIs are wrapped at present.

For example:

```
from temu.c.support.objsys import *
from temu.c.support.cpu import *
from temu.c.support.assembler import *
assembleToMemory(objectForName("board0-mem"), "st %g1, [%g2]", 0x40000000)
cpuSetPc(objectForName("board0-cpu"), 0x40000000)
# Set %g2 (used for address of store)
cpuSetReg(objectForName("board0-cpu"), 2, 0x80001000)
cpuStep(objectForName("board0-cpu"), 1) # Cpu will write data to device register
print "%x" % (getValueU32(objectForName("mydevice"), "myreg", 0))
```

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