TEMU

Tutorial

Version latest, 2024-05-14

Table of Contents

Chapter 1. Getting Started

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

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>
```

1.2. Command Line Interface

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

```
$ PATH=/opt/temu/2.2.0/bin:$PATH
```

Then create a directory for the rest of the tutorial:

```
$ mkdir tutorial && cd tutorial
```

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 executing the leon3.temu script in `/opt/temu/latest/share/temu/sysconfig`. The sysconfig directory is automatically available for the exec command in TEMU:

```
temu>
exec leon3.temu
:info: rom0 : map device at 0x00000000 □ 0x00100000
:info: ram0 : map device at 0x40000000 □ 0x48000000
:info: ftmctrl0 : map device at 0x80000000 □ 0x80000100
:info: apbuart0 : map device at 0x80000100 □ 0x80000200
```

```
:info: irqMp0 : map device at 0x80000200 … 0x80000300
:info: gpTimer0 : map device at 0x80000300 … 0x80000400
:info: ahbstat0 : map device at 0x80000f00 … 0x80001000
:info: apbctrl0 : map device at 0x800ff000 … 0x80100000
:info: ahbctrl0 : map device at 0xfffff000 … 0x00000000
0.000000: warning: cpu0 : sanity check: dCache not connected
0.000000: warning: cpu0 : sanity check: iCache not connected
0.000000: warning: cpu0 : sanity check: machine not connected
:warning: mem0 : sanity check: faultyHandlers not connected
:warning: mem0 : sanity check: postTransaction not connected
:warning: mem0 : sanity check: preTransaction not connected
:warning: mem0 : sanity check: upsetHandlers not connected
:warning: mem0 : sanity check: user1Handlers not connected
:warning: mem0 : sanity check: user2Handlers not connected
:warning: mem0 : sanity check: user3Handlers not connected
```

As can be seen above, when creating the LEON3, 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 script 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:

```
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=Leon3`.

Correspondingly, an object can be inspected by using the `object-info` command. Try `object-info obj=cpu0`.

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 to get all CPU related commands by typing cpu-`Tab`.

```
temu> cpu-<TAB>
Completions:
  cpu-disable-trap-events
  cpu-enable-trap-events
  cpu-reset
  cpu-run
  cpu-set-pc
  cpu-set-reg
  cpu-show-regs
  cpu-step
temu> cpu-show-regs cpu=cpu0
%g0 = 0x00000000
%g1 = 0x00000000
%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
temu> exec leon3.temu
temu> load obj=mem0 file=./rtems-hello.prom
:info: mem0 : loading big-endian sparc 32-bit
:info: mem0 : loading segment 1/1 0x00000000 - 0x00015de0 pa = 0x00000000
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: irqMp0 : read unknown register @ 0x80000220
Hello World
Hello World over printk() on Debug console
1.276503: info: cpu0 : 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. Secondly, at 1.276503 seconds, cpu0 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=cpu0 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.

Chapter 2. 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.

Create a new file in your tutorial directory named `simulator.c` This file will be compiled using the command:

```
$ cc simulator.c -I/opt/temu/2.2.0/include -L/opt/temu/2.2.0/lib \
     -lTEMUSupport -o simulator
```

And executed using:

```
$ LD_LIBRARY_PATH=/opt/temu/2.2.0/lib ./simulator
```

Then we can write the following program in that file:

```
#include <stdint.h>
#include "temu-c/Support/Init.h"
#include "temu-c/Support/CommandLine.h"
#include "temu-c/Support/Objsys.h"
#include "temu-c/Support/Loader.h"
#include "temu-c/Support/Cpu.h"

int
main(int argc, const char *argv[])
{
  // Initialise TEMU support library, this is mandatory, it will
  // among other things ensure you have a valid license.
  temu_initSupportLib();
  // Init path support will populate various TEMU paths based on the
  // location of the temu command. Note that this does not work in
  // case "temu" is a shell alias. Also "temu" must be visible in your
  // $PATH. If you do not want PATH to be modified, you can specify the
  // complete TEMU path here. The call to this function is optional but
  // it makes sure that the temu_execCommandFile() works without an
  // absolute path (e.g. /opt/temu/2.2.0/share/temu/sysconfig/leon3.temu)
  temu_initPathSupport("temu");
  // Create a system based on the bundled LEON3 configuration.
  temu_execCommandFile("leon3.temu");

  // cpu0 is created by the leon3.temu script
  void *cpu0 = temu_objectForName("cpu0");
```

```
    // You can load an image to any object implementing MemoryIface
    temu_loadImage(cpu0, "rtems-hello.prom");
    // Run CPU for the given number of cycles
    temu_cpuRun(cpu0, UINT64_C(1000000000));
    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.

Chapter 3. 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 Mach-O bundles on macOS). On Linux, you can compile the simple plug-in as:

```
$ cc -fPIC -shared first.c -I/opt/temu/2.2.0/include -o libfirst.so
```

The hello world plug-in is a very small:

```
#include <stdio.h>
#include "temu-c/Support/Objsys.h"
TEMU_PLUGIN_INIT
{
  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. Use of 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>
#include "temu-c/Support/Objsys.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_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 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> create-object class=MyModel 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> create-object class=MyModel name=bar \
      args="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, arguments for the create function have their type inferred by the way the argument is formatted, e.g. 123 is an integer, while 123.0 is a floating point value, anything that isn't a number will be passed as a string:

```
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);
  }
}
```

Also add `#include <inttypes.h>`, to get the PRIu64 macro defined. Now recompile the plug-in, restart TEMU and reload the plug-in. Try the "object-create" 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 object-prop-write command. In the example given, you can say:

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

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

Chapter 4. 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 static void
writeA(void *obj, temu_Propval pv, int idx)
{
  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");
```

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> object-create class=MyModel name=foo
args="baz:42,foobar:42.0,blah:bar3"
temu> object-prop-write prop=foo.a val=42
:info: foo : updating a from 0 to 42
temu> object-prop-read prop=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);
  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> exec ut700.temu
temu> import mmio
temu> object-create class=MyModel name=foo
```

The model then needs to be mapped to memory, this is done with the "memory-map" command, but first, let's see which address is free in the UT700 system that you just 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=mem0
Memory Space 'mem0'
Address     Length           Object
----------- ---------------- ------
0x000000000            1 MiB rom0
0x040000000          128 MiB ram0
0x080000000          256   B ftmctrl0
```

```
0x080000100          256   B apbuart0
0x080000200          256   B irqMp0
0x080000300          256   B gpTimer0
0x080000900          256   B gpio0
0x080000a00          256   B spw1
0x080000b00          256   B spw2
0x080000c00          256   B spw3
0x080000d00          256   B spw4
0x080000f00          256   B ahbstat0
0x0800ff000            4 KiB apbctrl0
0x0801ff000            4 KiB apbctrl1
0x0fff20000          256   B canoc0
0x0fff20100          256   B canoc1
0x0fffff000            4 KiB ahbctrl0
```

As can be seen, after ahbstat0, there is some free space: `0x80000f00 + 256 = 0x80001000`, and this area is not mapped.

```
temu> memory-map memspace=mem0 object=foo 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`.

tion, 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> assemble cpu=cpu0 instr="st %g1, [%g2 + 0]"
temu> assemble cpu=cpu0 instr="ld [%g2 + 0], %g3"
```

The program will by default end up at address 0 and 4. Enter the command `disassemble cpu=cpu0 addr=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`:

```
temu> cpu-set-reg cpu=cpu0 reg="%g2" value=0x80001000
temu> cpu-set-reg cpu=cpu0 reg="%g1" value=42
temu> step cpu=cpu0 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!

Chapter 5. 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)`, 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=cpu0
```

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 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 run the processor until it hits error mode, just run the following command:

```
temu> step obj=cpu0 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.

Chapter 6. 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: `irqMp0`. It is easy to use the connect command to attach your MMIO model to the IRQ controller:

```
temu> connect a=foo.irq b=irqMp0: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 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 );
}
```

Chapter 7. 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],
```

```
  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=mem0: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.

Chapter 8. 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 in should never be done from the receiving function. The reason is that, the sending device is not done with updating its state and need to finish executing the sending function. Thus, responses should always be posted using events.

8.1. 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 does ofcourse 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;
tty->uart.Iface->write(tty->uart.Obj, 'x');
```

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 the UT700 scripts 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> object-create class=ApbUart 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=mem0 addr=0x80001100 length=256 object=apbuart1
temu> connect a=cpu0.devices b=apbuart1:DeviceIface
temu> connect a=apbuart1.irqCtrl b=irqMp0:IrqIface
temu> connect-timesource obj=apbuart1 ts=cpu0
temu> connect a=apbctrl0.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> object-create class=MyTTY name=tty
temu> connect-timesource obj=tty ts=cpu0
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,now when running you should see the logging message from the model and the output on `apbuart0` 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
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).

8.2. 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);       \
  }

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 in the UT700 configuration:

```
temu> object-create class=MyDevice name=sigdev
temu> connect a=gpio0.outSignals[0] b=sigdev:signals[0]
temu> connect a=gpio0.outSignals[4] b=sigdev:signals[2]
```

And provide a small RTEMS program for triggering the updates:

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

8.3. 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 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,
};
```

Plug-in init:

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

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> import BusModels
temu> object-create class=SimpleCANBus name=canbus0
```

Then instantiate your CAN device and connect it and the canoc0 controller (created by the `ut700.temu`-script) to the bus, this connection is done using the can-connect command:

```
temu> object-create class=MyCANDevice name=candev0
temu> can-connect bus=canbus0:CanBusIface dev=candev0:can_a
temu> can-connect bus=canbus0:CanBusIface
```

```
dev=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
    = (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.

8.4. 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 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.

Chapter 9. External Models

External models have been deprecated in TEMU 2.2. Use pseudo properties instead

Simulators often consist of models for different frameworks. The TEMU object system has direct support for interfacing with external models using the notion of an external class. With this notion, the user can provide a set of wrapper functions for pseudo-property access, and a set of interfaces to integrate external models with e.g. TEMU bus models. Create a struct or C++ class to simulate an external model and register this class with TEMU:

```
temu_Class *c = temu_registerExternalClass("myextclass");
temu_addInterface(c, ...);
temu_addPseudoProperty(c, ...);
```

Now new objects can then be added as follows:

```
temu_addObject("myextclass", "myobjectname", obj);
```

It should now be possible to use the connect function / command and to list the objects in the command line interface. Note that external objects are not saved in snapshots and they cannot be used as processors, machines, or memory spaces.

Chapter 10. Physical Networks

U 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.

```
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)
{
  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");
    }
  }
}
```

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, void *source, void *notinfo)
{
  temu_TrapEventInfo *info = (temu_TrapEventInfo *)notinfo;
  temu_logInfo(source, "trap taken %x", info->TrapId);
}
void
trapExit(void *arg, void *source, 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 *cpu0 = temu_objectForName("cpu0");
  temu_CpuIface *iface = temu_getInterface(cpu0, "CpuIface", 0);
  iface->enableTrapEvents(cpu0);
  temu_subscribeNotification("temu.cpuTrapEntry", cpu0, NULL, trapEntry);
  temu_subscribeNotification("temu.cpuTrapExit", cpu0, 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.

Chapter 12. Scripting

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"

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");
  temu_logInfo(NULL, "mycommand executed with object % s",
               temu_nameForObject(obj));
}

data_t data;

TEMU_PLUGIN_INIT
{
  void *cmd = temu_createCmd("mycommand", mycmdfunc,
                             "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/latest}/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("mem0"), "st %g1, [%g2]", 0x40000000)
cpuSetPc(objectForName("cpu0"), 0x40000000)
# Set %g2 (used for address of store)
cpuSetReg(objectForName("cpu0"), 2, 0x80001000)
cpuStep(objectForName("cpu0"), 1) # Cpu will write data to device register
print "%x" % (getValueU32(objectForName("mydevice"), "myreg", 0))
```