QEMU – SCENARIO ENGINE
TEST DEVELOPMENT HOW-TO

Real-Time emulation with Qemu

08/07/2015

OPEN WIDE, SAS au capital de 129 350 euros
N° SIRET: 437 664 394 00016 RCS PARIS B 437 664 394
Code APE: 6202A Siège Social : 23/25 rue Daviel 75013 PARIS
Tél : 01 42 68 28 00 - Fax : 01 53 80 30 37

SOMMAIRE

1 INTRODUCTION

1.1 The goal

The Qemu scenario engine is a framework designed to test the software running on the guest machine.

It allows interacting with the guest devices by stimulating inputs and reading outputs. It also provides a utility to schedule time based events that can trigger some actions like changing an input or checking an output.

1.2 Real-time emulation

The original purpose of the scenario engine is to measure and validate delays of the guest software in order to check if it respects real time constraints. To achieve this goal, Qemu have to be run with the -icount shift=0,nosleep parameter. These option ensure that Qemu will have deterministic execution times. The scheduler can also be used to trigger some actions and checks at precise dates.

In order to get deterministic execution the whole system, including the scenario engine and the device emulators, have to rely only on the Qemu virtual clock (QEMU_CLOCK_VIRTUAL). The 'nosleep' mode of the -icount option will make the virtual clock “purely” virtual by removing any reference to the real clock in its computation. This prevents the host OS from introducing latency in the guest clock.

This kind of setting allows testing software delays on emulated hardware as if it was on real hardware. The latency induced by the host OS scheduling and Qemu self execution is completely removed and the clock computation only rely on guest CPU instruction counter and timers deadlines, which approach “real” hardware time.

Adjusting shift value of the -icount option will modify the amount of virtual time added for each instruction by Qemu. This allows Qemu to approach real execution time. However, this parameter is not very flexible (1 instruction = 2^n nanoseconds).

1.3 Components

The three main components of the scenario engine are the device emulators API for interacting with the guest I/O, the scenario source file which is the core of the test and the scheduler, set up by an event file, which triggers user defined events at very accurate timestamps during the test.

1.3.1 The scenario source file

The scenario source file component contains the actual test. The user implements its test algorithms in that file, using the C programming language, and uses the emulators API to interact with the guest software. The user can also register custom callbacks to get event notifications from these emulators.

This test program can also use the scheduler to trigger virtual time based events. The scheduler component is described in the next section.

1.3.2 The event scheduler

The event scheduler sets up time based events. These events will call a specific function when they expire. They are described in an event description file which contains timestamps, each with a parameter which will be passed to the callback.

1.3.3 The device interaction APIs

The device API is a set of basic interaction functions integrated to a device emulator. They are used by the scenario engine to provide simple I/O interaction between the test scenario and the guest software. Those functions are used by the test algorithm to stimulate the external interface of the device:

- They can simulate a check on the device output by reading the device current state.
- They can simulate a change on the device input by writing in the device current state.

Those APIs can also include a callback to a user specified function when an event occurs. This call is used to notify the test scenario of a guest I/O event. This function is set at engine initialization time with the event callback registration utility function.

The device API component is intended to be generic and to be used in many different test cases. This API allows a user to write tests and device interactions independently of the targeted emulator and with no knowledge of Qemu's internal architecture

In the case of a *character device*, a common interaction API is already implemented and can be used for every device emulator of this kind. To use the emulator event notification a specific *chardev* family event callback register function has to be called.

2 HOW TO USE THE SCENARIO ENGINE

2.1 The scenario source file

The scenario file is the core of the test scenario. It contains the user's test algorithms, the device event callbacks and the scheduler callback.

It is a C source file which will be built with Qemu if the scenario engine is enabled.

2.1.1 Header files

The file *utils.h* has to be included, it contains declarations of “utility” functions like the event callback registration function. Each device emulator might also provide headers that need to be included by the scenario for device specific functions and callbacks.

2.1.2 Initialization

The entry point of the scenario engine is the initialization function. It is mandatory for a scenario to implement that function. The emulator event callbacks should be registered within this function, and the scheduler started if necessary. The callback registered at initialization time are the only calls to the scenario engine that will happen afterwards.

2.1.3 Event callback registration

To register event callbacks, one needs to call the generic event callback registration function which finds the desired devices and registers the custom callback. This function is *scenario_register_emulator_cb* and its parameters are the device type that will call the callback (ex. “pl061”), the event callback register function of this device emulator (ex. *pl061_scenario_cb_register*) and the user defined callback. This user callback have to implement the device specific prototype defined in the device emulator's scenario header.

The character devices case

In the case of a *chardev* emulator, one needs to use the *chardev* specific callback register function. This function is *scenario_register_chardev_cb,*. Its parameters are the *chardev* unique identifier and the user callback. This callback have to implement the *scenario_chardev_cb_t* prototype. Those elements are defined in “*include/sysemu/char.h*” as following:

- Callback prototype:

```
typedef int scenario_chardev_cb_t(
     struct CharDriverState *s, const uint8_t *buf, int len);
```

- Callback registration function:

```
void scenario_register_chardev_cb(
     const char *name, scenario_chardev_cb_t *cb, bool create);
```

2.2 The event scheduler

To start the event scheduler, the function “scenario_scheduler_start” have to be called. It will initialize the scheduler by reading the event description file and scheduling the first event. Its parameters are the callback function to call on an event, an opaque pointer to pass to the callback and the name of the event description file.

- The user specified callback have to implement the predefined prototype declared in the scheduler header (“*scheduler.h*”)

- The event description file name can be specified using the *-scenario* Qemu option. It can be retrieved with the “filename” parameter of the scenario initialization function.

2.3 The event description file

To use the event scheduler in the test scenario, an event description file needs to be filled with events. An event is a timestamp with a unique optional associated parameter.

- The timestamps are in nanoseconds since the event scheduler start (*scenario_scheduler_start* call)

- Each timestamp can have an associated parameter. This parameter is optional and there can be only one for each timestamp. This parameter will be passed to the callback along with the opaque pointer and the timestamp itself.

2.3.1 Syntax

The event description file is composed of an event list. Here is the syntax:

```
#<timestamp>
[%<associated parameter>]
…
;<comment>
…
```

Exemple:

```
; Event description file example
#10000000
%toggle_pin1
#10000100
; check for pin 0 toggling
%check_pin0
#20000000
#30000000
#50000000
%stop_exec
```

Each line beginning with a '#' is a timestamp representing a new event. The scheduler will trigger the callback at the specified time. The timestamp value and its parameter, if any, will be passed to the callback. This parameter is the line following the timestamp, which begin with '%'. It is passed as a string containing the entire line with the beginning '%' removed. If this line is omitted, an empty string will be passed to the callback.

Lines beginning with ';' are full line comments ignored by the scheduler and can be inserted anywhere.

The user callback have to implement the following prototype :

```
void callback(void *opaque, uint64_t timestamp, char *parameter);
     - opaque : the opaque pointer passed at scheduler startup
     - timestamp : timestamp value as written in the event file
     - parameter : a string representing the associated parameter
```

The previous example runs like this:

```
@start_time+10000000ns : callback(opaque, 10000000, "toggle_pin1")
@start_time+10000100ns : callback(opaque, 10000100, "check_pin0")
@start_time+20000000ns : callback(opaque, 20000000, "")
@start_time+30000000ns : callback(opaque, 30000000, "")
@start_time+50000000ns : callback(opaque, 50000000, "stop_exec")
```

2.4 Usage

In order to run the scenario engine, it needs to be enabled at build time by specifying the *--enable-scenario-engine* option to the *configure* script. Then the *-scenario file=<event_file_name>* option needs to be passed to Qemu at launch. If there is no event file, the *file=* option still have to be specified but left empty (this limitation is due to Qemu's options parsing).

3 DEVICE INTERACTION API DEVELOPMENT

In this section, we will describe how to develop a scenario API for a device emulator which does not already have one. This API is necessary for the scenario to interact with the virtual device.

An interaction API is mainly composed of two parts whose development will be detailed in this section:

- Basic interaction function, as said in the introduction. Allowing the scenario to send orders to the virtual device
- Callback infrastructure, allowing the virtual device to notify the scenario of events.

We will describe the PL061 API in order to explain how to develop a device interaction API. (The PL061 is a GPIO controller for ARM processors)

Every device emulator in Qemu includes a structure representing the state of the virtual device. In our example, this structure is named PL061State. Every virtual PL061 controller attached to the virtual machine is represented by an instance of PL061State. It is referred as *device instance* in this section.

Notice: In the case of a *character device* the interaction API does not need to be developed. A generic API for this kind of device is already implemented. See #2.1.The scenario for more details.

3.1 Scenario API conditional compilation

All the scenario API code should be surrounded by preprocessor instructions in order to avoid its compilation when the scenario engine is not enabled at build time.

Here are the instructions that should be added to each scenario API parts of the emulator:

```
#ifdef CONFIG_SCENARIO
    <API_code>
#endif
```

There, API_code is compiled only if *–enable-scenario-engine* is specified at *configure* time.

NB: those instructions will not be specified in further listings, but should be added in the actual code.

3.2 Scenario API header

First, we need to create a new C header file to declare all API functions. This file will be used by the scenario source file to call emulator API functions. This file is the only link between the device emulator and the scenario source file. The conventional naming is *<emulalor_file_name>_simu.h*.

3.3 Interaction functions

The interaction of the guest machine with the scenario will happen through a set of functions to query and set guest device registers, to check outputs and act on inputs. The calls implemented are simple in order to be as generic as possible..

- Write GPIO data register function (input simulation):

```
void pl061_simu_write(void *opaque, uint8_t mask, uint8_t value)
{
     PL061State *s = (PL061State *)opaque;

     s->data = (s->data & ~mask) | (value & mask);
     pl061_update(s);
}
```

- Read GPIO data register function:

```
uint8_t pl061_simu_read(void *opaque, uint8_t mask)
{
     PL061State *s = (PL061State *)opaque;

     return (s->data & mask);
}
```

With these functions, it is possible to read and write the data register of the controller. These simple functions cover almost every use case of this API.

The device instance is passed to the callbacks via the opaque pointer.

These functions also need to be declared in the header file mentioned above.

3.4 Event notification

To notify events from the guest device, a scenario-defined callback has to be called at a specific point of the emulator code. The callback infrastructure contains:

- New variables in the structure representing the device instance
- A registration function to save the user defined callback
- The actual callback function call

3.4.1 Specification of the event callback prototype

We need to define a new function type which will be the event callback prototype for this emulator so the user will have to implement a callback of this type. We put this new type declaration in the emulator header.

Here is the PL061 one:

```
typedef void simu_event_cb_t(void *opaque, const char *type, unsigned int simu_id);
```

The PL061 event callback has 3 parameters: an opaque pointer, which is the device emulator instance, a string describing the device type triggering the event, and an integer which is the device id (see next section).

3.4.2 Appending new variables to the device structure

Once the signature of the callback is known, two variables need to be added in the structure representing the device instance. In the PL061 case:

- a pointer to a function of the type that was defined in the preceding section

  ```
  simu_event_cb_t *simu_cb;
  ```

- an unsigned integer which will be an identifier used by test scenario. In the case where there is more thane one instance of a device in the virtual machine, this id will be used for differentiating them.

  ```
  unsigned int simu_id;
  ```

3.4.3 The callback register function

Finally we need a callback register function. . It has to implement the prototype *scenario_emulator_cb_register_t* defined in *scenario/utils.h* to allow the discovery and registration utility function to call it.

Here is the register function for the PL061 emulator:

```
void pl061_simu_register_cb(void *opaque, unsigned int simu_id, void
*opaque_cb)
{
     PL061State *s = PL061(opaque);
     simu_event_cb_t *cb = (simu_event_cb_t *) opaque_cb;

     if (!(s->simu_cb)) {
          s->simu_cb = cb;
          s->simu_id = simu_id;
     }
}
```

3.4.4 Event callback call

Now, we just have to call the user callback at the correct point in the emulator code.

In the PL061 case, we call the callback at the end of the *pl061_update* function, as it is called after each GPIO I/O.

```
     if (s->simu_cb) {
          s->simu_cb(s, TYPE_PL061, s→simu_id);
     }
```

3.5 Deterministic execution notice

If the test scenario is used to measure any delay, the used device emulators must only use the Qemu virtual clock (QEMU_CLOCK_VIRTUAL) and Qemu have to be launched with *-icount sleep=off*.

If those requirements are met, the guest runs in a deterministic manner and no more latency is introduced by the host OS scheduling.

4 CONCLUSION

This scenario engine provides a framework to easily write and run tests against the guest software.

4.1 A complete framework

If a user wants to develop a test which uses device emulators already bundled with a scenario API, then it is easy for him to code his test. He does not need to know all the Qemu device emulators architecture to interact with the guest software. The only thing to implement is the test algorithm, with the help of utilities (the API functions, the event scheduler). Then the test can be run by enabling the scenario engine at build time and at launch time.

4.2 Extensible

This framework provides a structure for building new device emulator APIs. Even if each device type has its own API, a common architecture brings consistency to the whole system.

4.3 Build for real time tests

Running a scenario with the *icount* option in *sleep=off* mode and using the event scheduler, one can check delays of the guest software. It can be used for validating that deadlines are met in a real time OS.

4.4 Usable for other tests

The scenario engine is a framework to help a user to create and run scenarios interacting with the guest machine's inputs and outputs.