Timekeeping Virtualization for X86-Based Architectures

Author: Zachary Amsden <zamsden@redhat.com>

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\virt\kvm\x86\(linux-master)(Documentation)(virt)(kvm)(x86)timekeeping.rst, line 8)

Cannot extract compound bibliographic field "Copyright".

1. Overview

One of the most complicated parts of the X86 platform, and specifically, the virtualization of this platform is the plethora of timing devices available and the complexity of emulating those devices. In addition, virtualization of time introduces a new set of challenges because it introduces a multiplexed division of time beyond the control of the guest CPU.

First, we will describe the various timekeeping hardware available, then present some of the problems which arise and solutions available, giving specific recommendations for certain classes of KVM guests.

The purpose of this document is to collect data and information relevant to timekeeping which may be difficult to find elsewhere, specifically, information relevant to KVM and hardware-based virtualization.

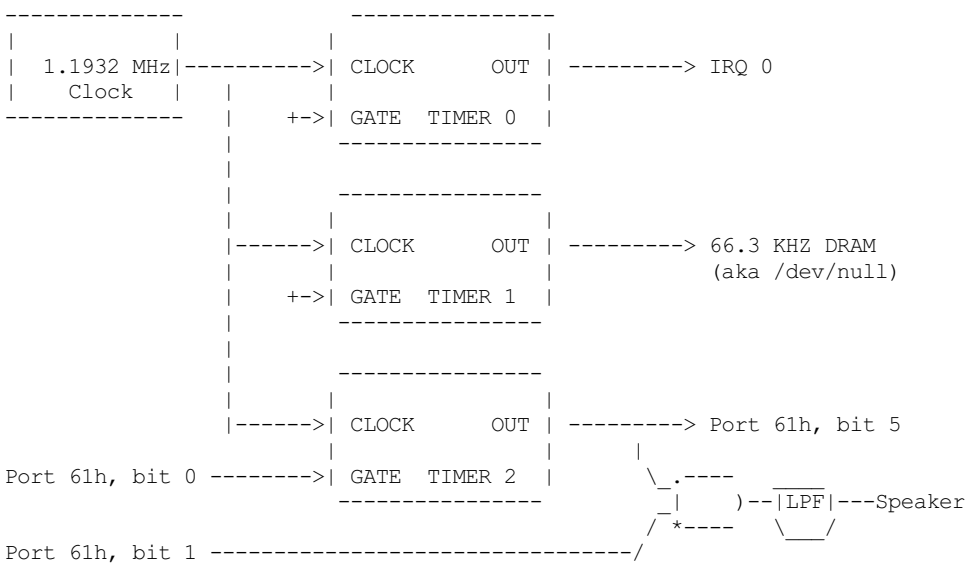
2. Timing Devices

First we discuss the basic hardware devices available. TSC and the related KVM clock are special enough to warrant a full exposition and are described in the following section.

2.1. i8254 - PIT

One of the first timer devices available is the programmable interrupt timer, or PIT. The PIT has a fixed frequency 1.193182 MHz base clock and three channels which can be programmed to deliver periodic or one-shot interrupts. These three channels can be configured in different modes and have individual counters. Channel 1 and 2 were not available for general use in the original IBM PC, and historically were connected to control RAM refresh and the PC speaker. Now the PIT is typically integrated as part of an emulated chipset and a separate physical PIT is not used.

The PIT uses I/O ports 0x40 - 0x43. Access to the 16-bit counters is done using single or multiple byte access to the I/O ports. There are 6 modes available, but not all modes are available to all timers, as only timer 2 has a connected gate input, required for modes 1 and 5. The gate line is controlled by port 61h, bit 0, as illustrated in the following diagram:



The timer modes are now described.

Mode 0: Single Timeout.
: This is a one-shot software timeout that counts down when the gate is high (always true for timers 0 and 1). When the count reaches zero, the output goes high.

Mode 1: Triggered One-shot.
: The output is initially set high. When the gate line is set high, a countdown is initiated (which does not stop if the gate is

lowered), during which the output is set low. When the count reaches zero, the output goes high.

Mode 2: Rate Generator.

The output is initially set high. When the countdown reaches 1, the output goes low for one count and then returns high. The value is reloaded and the countdown automatically resumes. If the gate line goes low, the count is halted. If the output is low when the gate is lowered, the output automatically goes high (this only affects timer 2).

Mode 3: Square Wave.

This generates a high / low square wave. The count determines the length of the pulse, which alternates between high and low when zero is reached. The count only proceeds when gate is high and is automatically reloaded on reaching zero. The count is decremented twice at each clock to generate a full high / low cycle at the full periodic rate. If the count is even, the clock remains high for N/2 counts and low for N/2 counts; if the clock is odd, the clock is high for (N+1)/2 counts and low for (N-1)/2 counts. Only even values are latched by the counter, so odd values are not observed when reading. This is the intended mode for timer 2, which generates sine-like tones by low-pass filtering the square wave output.

Mode 4: Software Strobe.

After programming this mode and loading the counter, the output remains high until the counter reaches zero. Then the output goes low for 1 clock cycle and returns high. The counter is not reloaded. Counting only occurs when gate is high.

Mode 5: Hardware Strobe.

After programming and loading the counter, the output remains high. When the gate is raised, a countdown is initiated (which does not stop if the gate is lowered). When the counter reaches zero, the output goes low for 1 clock cycle and then returns high. The counter is not reloaded.

In addition to normal binary counting, the PIT supports BCD counting. The command port, 0x43 is used to set the counter and mode for each of the three timers.

PIT commands, issued to port 0x43, using the following bit encoding:

```
Bit 7-4: Command (See table below)
Bit 3-1: Mode (000 = Mode 0, 101 = Mode 5, 11X = undefined)
Bit 0  : Binary (0) / BCD (1)
```

Command table:

```
0000 - Latch Timer 0 count for port 0x40
      sample and hold the count to be read in port 0x40;
      additional commands ignored until counter is read;
      mode bits ignored.

0001 - Set Timer 0 LSB mode for port 0x40
      set timer to read LSB only and force MSB to zero;
      mode bits set timer mode

0010 - Set Timer 0 MSB mode for port 0x40
      set timer to read MSB only and force LSB to zero;
      mode bits set timer mode

0011 - Set Timer 0 16-bit mode for port 0x40
      set timer to read / write LSB first, then MSB;
      mode bits set timer mode

0100 - Latch Timer 1 count for port 0x41 - as described above
0101 - Set Timer 1 LSB mode for port 0x41 - as described above
0110 - Set Timer 1 MSB mode for port 0x41 - as described above
0111 - Set Timer 1 16-bit mode for port 0x41 - as described above

1000 - Latch Timer 2 count for port 0x42 - as described above
1001 - Set Timer 2 LSB mode for port 0x42 - as described above
1010 - Set Timer 2 MSB mode for port 0x42 - as described above
1011 - Set Timer 2 16-bit mode for port 0x42 as described above

1101 - General counter latch
      Latch combination of counters into corresponding ports
      Bit 3 = Counter 2
      Bit 2 = Counter 1
      Bit 1 = Counter 0
      Bit 0 = Unused

1110 - Latch timer status
      Latch combination of counter mode into corresponding ports
      Bit 3 = Counter 2
      Bit 2 = Counter 1
      Bit 1 = Counter 0

      The output of ports 0x40-0x42 following this command will be:

      Bit 7 = Output pin
      Bit 6 = Count loaded (0 if timer has expired)
      Bit 5-4 = Read / Write mode
          01 = MSB only
```

```
            10 = LSB only
            11 = LSB / MSB (16-bit)
        Bit 3-1 = Mode
        Bit 0 = Binary (0) / BCD mode (1)
```

2.2. RTC

The second device which was available in the original PC was the MC146818 real time clock. The original device is now obsolete, and usually emulated by the system chipset, sometimes by an HPET and some frankenstein IRQ routing.

The RTC is accessed through CMOS variables, which uses an index register to control which bytes are read. Since there is only one index register, read of the CMOS and read of the RTC require lock protection (in addition, it is dangerous to allow userspace utilities such as hwclock to have direct RTC access, as they could corrupt kernel reads and writes of CMOS memory).

The RTC generates an interrupt which is usually routed to IRQ 8. The interrupt can function as a periodic timer, an additional once a day alarm, and can issue interrupts after an update of the CMOS registers by the MC146818 is complete. The type of interrupt is signalled in the RTC status registers.

The RTC will update the current time fields by battery power even while the system is off. The current time fields should not be read while an update is in progress, as indicated in the status register.

The clock uses a 32.768kHz crystal, so bits 6-4 of register A should be programmed to a 32kHz divider if the RTC is to count seconds.

This is the RAM map originally used for the RTC/CMOS:

```
Location    Size    Description
------------------------------------------
00h         byte    Current second (BCD)
01h         byte    Seconds alarm (BCD)
02h         byte    Current minute (BCD)
03h         byte    Minutes alarm (BCD)
04h         byte    Current hour (BCD)
05h         byte    Hours alarm (BCD)
06h         byte    Current day of week (BCD)
07h         byte    Current day of month (BCD)
08h         byte    Current month (BCD)
09h         byte    Current year (BCD)
0Ah         byte    Register A
                     bit 7   = Update in progress
                     bit 6-4 = Divider for clock
                                000 = 4.194 MHz
                                001 = 1.049 MHz
                                010 = 32 kHz
                                10X = test modes
                                110 = reset / disable
                                111 = reset / disable
                     bit 3-0 = Rate selection for periodic interrupt
                                000 = periodic timer disabled
                                001 = 3.90625 uS
                                010 = 7.8125 uS
                                011 = .122070 mS
                                100 = .244141 mS
                                   ...
                               1101 = 125 mS
                               1110 = 250 mS
                               1111 = 500 mS
0Bh         byte    Register B
                     bit 7   = Run (0) / Halt (1)
                     bit 6   = Periodic interrupt enable
                     bit 5   = Alarm interrupt enable
                     bit 4   = Update-ended interrupt enable
                     bit 3   = Square wave interrupt enable
                     bit 2   = BCD calendar (0) / Binary (1)
                     bit 1   = 12-hour mode (0) / 24-hour mode (1)
                     bit 0   = 0 (DST off) / 1 (DST enabled)
0Ch         byte    Register C (read only)
                     bit 7   = interrupt request flag (IRQF)
                     bit 6   = periodic interrupt flag (PF)
                     bit 5   = alarm interrupt flag (AF)
                     bit 4   = update interrupt flag (UF)
                     bit 3-0 = reserved
0Dh         byte    Register D (read only)
                     bit 7   = RTC has power
                     bit 6-0 = reserved
32h         byte    Current century BCD (*)
(*) location vendor specific and now determined from ACPI global tables
```

2.3. APIC

On Pentium and later processors, an on-board timer is available to each CPU as part of the Advanced Programmable Interrupt Controller. The APIC is accessed through memory-mapped registers and provides interrupt service to each CPU, used for IPIs and local timer interrupts.

Although in theory the APIC is a safe and stable source for local interrupts, in practice, many bugs and glitches have occurred due to the special nature of the APIC CPU-local memory-mapped hardware. Beware that CPU errata may affect the use of the APIC and that workarounds may be required. In addition, some of these workarounds pose unique constraints for virtualization - requiring either extra overhead incurred from extra reads of memory-mapped I/O or additional functionality that may be more computationally expensive to implement.

Since the APIC is documented quite well in the Intel and AMD manuals, we will avoid repetition of the detail here. It should be pointed out that the APIC timer is programmed through the LVT (local vector timer) register, is capable of one-shot or periodic operation, and is based on the bus clock divided down by the programmable divider register.

2.4. HPET

HPET is quite complex, and was originally intended to replace the PIT / RTC support of the X86 PC. It remains to be seen whether that will be the case, as the de facto standard of PC hardware is to emulate these older devices. Some systems designated as legacy free may support only the HPET as a hardware timer device.

The HPET spec is rather loose and vague, requiring at least 3 hardware timers, but allowing implementation freedom to support many more. It also imposes no fixed rate on the timer frequency, but does impose some extremal values on frequency, error and slew.

In general, the HPET is recommended as a high precision (compared to PIT /RTC) time source which is independent of local variation (as there is only one HPET in any given system). The HPET is also memory-mapped, and its presence is indicated through ACPI tables by the BIOS.

Detailed specification of the HPET is beyond the current scope of this document, as it is also very well documented elsewhere.

2.5. Offboard Timers

Several cards, both proprietary (watchdog boards) and commonplace (e1000) have timing chips built into the cards which may have registers which are accessible to kernel or user drivers. To the author's knowledge, using these to generate a clocksource for a Linux or other kernel has not yet been attempted and is in general frowned upon as not playing by the agreed rules of the game. Such a timer device would require additional support to be virtualized properly and is not considered important at this time as no known operating system does this.

3. TSC Hardware

The TSC or time stamp counter is relatively simple in theory; it counts instruction cycles issued by the processor, which can be used as a measure of time. In practice, due to a number of problems, it is the most complicated timekeeping device to use.

The TSC is represented internally as a 64-bit MSR which can be read with the RDMSR, RDTSC, or RDTSCP (when available) instructions. In the past, hardware limitations made it possible to write the TSC, but generally on old hardware it was only possible to write the low 32-bits of the 64-bit counter, and the upper 32-bits of the counter were cleared. Now, however, on Intel processors family 0Fh, for models 3, 4 and 6, and family 06h, models e and f, this restriction has been lifted and all 64-bits are writable. On AMD systems, the ability to write the TSC MSR is not an architectural guarantee.

The TSC is accessible from CPL-0 and conditionally, for CPL > 0 software by means of the CR4.TSD bit, which when enabled, disables CPL > 0 TSC access.

Some vendors have implemented an additional instruction, RDTSCP, which returns atomically not just the TSC, but an indicator which corresponds to the processor number. This can be used to index into an array of TSC variables to determine offset information in SMP systems where TSCs are not synchronized. The presence of this instruction must be determined by consulting CPUID feature bits.

Both VMX and SVM provide extension fields in the virtualization hardware which allows the guest visible TSC to be offset by a constant. Newer implementations promise to allow the TSC to additionally be scaled, but this hardware is not yet widely available.

3.1. TSC synchronization

The TSC is a CPU-local clock in most implementations. This means, on SMP platforms, the TSCs of different CPUs may start at different times depending on when the CPUs are powered on. Generally, CPUs on the same die will share the same clock, however, this is not always the case.

The BIOS may attempt to resynchronize the TSCs during the poweron process and the operating system or other system software may attempt to do this as well. Several hardware limitations make the problem worse - if it is not possible to write the full 64-bits of the TSC, it may be impossible to match the TSC in newly arriving CPUs to that of the rest of the system, resulting in unsynchronized TSCs. This may be done by BIOS or system software, but in practice, getting a perfectly synchronized TSC will not be possible unless all values are read from the same clock, which generally only is possible on single socket systems or those with special hardware support.

3.2. TSC and CPU hotplug

As touched on already, CPUs which arrive later than the boot time of the system may not have a TSC value that is synchronized with the rest of the system. Either system software, BIOS, or SMM code may actually try to establish the TSC to a value matching the rest of the system, but a perfect match is usually not a guarantee. This can have the effect of bringing a system from a state where TSC is synchronized back to a state where TSC synchronization flaws, however small, may be exposed to the OS and any virtualization environment.

3.3. TSC and multi-socket / NUMA

Multi-socket systems, especially large multi-socket systems are likely to have individual clocksources rather than a single, universally distributed clock. Since these clocks are driven by different crystals, they will not have perfectly matched frequency, and temperature and electrical variations will cause the CPU clocks, and thus the TSCs to drift over time. Depending on the exact clock and bus design, the drift may or may not be fixed in absolute error, and may accumulate over time.

In addition, very large systems may deliberately slew the clocks of individual cores. This technique, known as spread-spectrum clocking, reduces EMI at the clock frequency and harmonics of it, which may be required to pass FCC standards for telecommunications and computer equipment.

It is recommended not to trust the TSCs to remain synchronized on NUMA or multiple socket systems for these reasons.

3.4. TSC and C-states

C-states, or idling states of the processor, especially C1E and deeper sleep states may be problematic for TSC as well. The TSC may stop advancing in such a state, resulting in a TSC which is behind that of other CPUs when execution is resumed. Such CPUs must be detected and flagged by the operating system based on CPU and chipset identifications.

The TSC in such a case may be corrected by catching it up to a known external clocksource.

3.5. TSC frequency change / P-states

To make things slightly more interesting, some CPUs may change frequency. They may or may not run the TSC at the same rate, and because the frequency change may be staggered or slewed, at some points in time, the TSC rate may not be known other than falling within a range of values. In this case, the TSC will not be a stable time source, and must be calibrated against a known, stable, external clock to be a usable source of time.

Whether the TSC runs at a constant rate or scales with the P-state is model dependent and must be determined by inspecting CPUID, chipset or vendor specific MSR fields.

In addition, some vendors have known bugs where the P-state is actually compensated for properly during normal operation, but when the processor is inactive, the P-state may be raised temporarily to service cache misses from other processors. In such cases, the TSC on halted CPUs could advance faster than that of non-halted processors. AMD Turion processors are known to have this problem.

3.6. TSC and STPCLK / T-states

External signals given to the processor may also have the effect of stopping the TSC. This is typically done for thermal emergency power control to prevent an overheating condition, and typically, there is no way to detect that this condition has happened.

3.7. TSC virtualization - VMX

VMX provides conditional trapping of RDTSC, RDMSR, WRMSR and RDTSCP instructions, which is enough for full virtualization of TSC in any manner. In addition, VMX allows passing through the host TSC plus an additional TSC_OFFSET field specified in the VMCS. Special instructions must be used to read and write the VMCS field.

3.8. TSC virtualization - SVM

SVM provides conditional trapping of RDTSC, RDMSR, WRMSR and RDTSCP instructions, which is enough for full virtualization of TSC in any manner. In addition, SVM allows passing through the host TSC plus an additional offset field specified in the SVM control block.

3.9. TSC feature bits in Linux

In summary, there is no way to guarantee the TSC remains in perfect synchronization unless it is explicitly guaranteed by the architecture. Even if so, the TSCs in multi-sockets or NUMA systems may still run independently despite being locally consistent.

The following feature bits are used by Linux to signal various TSC attributes, but they can only be taken to be meaningful for UP or single node systems.

X86_FEATURE_TSC	The TSC is available in hardware
X86_FEATURE_RDTSCP	The RDTSCP instruction is available
X86_FEATURE_CONSTANT_TSC	The TSC rate is unchanged with P-states

X86_FEATURE_NONSTOP_TSC	The TSC does not stop in C-states
X86_FEATURE_TSC_RELIABLE	TSC sync checks are skipped (VMware)

4. Virtualization Problems

Timekeeping is especially problematic for virtualization because a number of challenges arise. The most obvious problem is that time is now shared between the host and, potentially, a number of virtual machines. Thus the virtual operating system does not run with 100% usage of the CPU, despite the fact that it may very well make that assumption. It may expect it to remain true to very exacting bounds when interrupt sources are disabled, but in reality only its virtual interrupt sources are disabled, and the machine may still be preempted at any time. This causes problems as the passage of real time, the injection of machine interrupts and the associated clock sources are no longer completely synchronized with real time.

This same problem can occur on native hardware to a degree, as SMM mode may steal cycles from the naturally on X86 systems when SMM mode is used by the BIOS, but not in such an extreme fashion. However, the fact that SMM mode may cause similar problems to virtualization makes it a good justification for solving many of these problems on bare metal.

4.1. Interrupt clocking

One of the most immediate problems that occurs with legacy operating systems is that the system timekeeping routines are often designed to keep track of time by counting periodic interrupts. These interrupts may come from the PIT or the RTC, but the problem is the same: the host virtualization engine may not be able to deliver the proper number of interrupts per second, and so guest time may fall behind. This is especially problematic if a high interrupt rate is selected, such as 1000 HZ, which is unfortunately the default for many Linux guests.

There are three approaches to solving this problem; first, it may be possible to simply ignore it. Guests which have a separate time source for tracking 'wall clock' or 'real time' may not need any adjustment of their interrupts to maintain proper time. If this is not sufficient, it may be necessary to inject additional interrupts into the guest in order to increase the effective interrupt rate. This approach leads to complications in extreme conditions, where host load or guest lag is too much to compensate for, and thus another solution to the problem has risen: the guest may need to become aware of lost ticks and compensate for them internally. Although promising in theory, the implementation of this policy in Linux has been extremely error prone, and a number of buggy variants of lost tick compensation are distributed across commonly used Linux systems.

Windows uses periodic RTC clocking as a means of keeping time internally, and thus requires interrupt slewing to keep proper time. It does use a low enough rate (ed: is it 18.2 Hz?) however that it has not yet been a problem in practice.

4.2. TSC sampling and serialization

As the highest precision time source available, the cycle counter of the CPU has aroused much interest from developers. As explained above, this timer has many problems unique to its nature as a local, potentially unstable and potentially unsynchronized source. One issue which is not unique to the TSC, but is highlighted because of its very precise nature is sampling delay. By definition, the counter, once read is already old. However, it is also possible for the counter to be read ahead of the actual use of the result. This is a consequence of the superscalar execution of the instruction stream, which may execute instructions out of order. Such execution is called non-serialized. Forcing serialized execution is necessary for precise measurement with the TSC, and requires a serializing instruction, such as CPUID or an MSR read.

Since CPUID may actually be virtualized by a trap and emulate mechanism, this serialization can pose a performance issue for hardware virtualization. An accurate time stamp counter reading may therefore not always be available, and it may be necessary for an implementation to guard against "backwards" reads of the TSC as seen from other CPUs, even in an otherwise perfectly synchronized system.

4.3. Timespec aliasing

Additionally, this lack of serialization from the TSC poses another challenge when using results of the TSC when measured against another time source. As the TSC is much higher precision, many possible values of the TSC may be read while another clock is still expressing the same value.

That is, you may read (T,T+10) while external clock C maintains the same value. Due to non-serialized reads, you may actually end up with a range which fluctuates - from (T-1.. T+10). Thus, any time calculated from a TSC, but calibrated against an external value may have a range of valid values. Re-calibrating this computation may actually cause time, as computed after the calibration, to go backwards, compared with time computed before the calibration.

This problem is particularly pronounced with an internal time source in Linux, the kernel time, which is expressed in the theoretically high resolution timespec - but which advances in much larger granularity intervals, sometimes at the rate of jiffies, and possibly in catchup modes, at a much larger step.

This aliasing requires care in the computation and recalibration of kvmclock and any other values derived from TSC computation (such as TSC virtualization itself).

4.4. Migration

Migration of a virtual machine raises problems for timekeeping in two ways. First, the migration itself may take time, during which

interrupts cannot be delivered, and after which, the guest time may need to be caught up. NTP may be able to help to some degree here, as the clock correction required is typically small enough to fall in the NTP-correctable window.

An additional concern is that timers based off the TSC (or HPET, if the raw bus clock is exposed) may now be running at different rates, requiring compensation in some way in the hypervisor by virtualizing these timers. In addition, migrating to a faster machine may preclude the use of a passthrough TSC, as a faster clock cannot be made visible to a guest without the potential of time advancing faster than usual. A slower clock is less of a problem, as it can always be caught up to the original rate. KVM clock avoids these problems by simply storing multipliers and offsets against the TSC for the guest to convert back into nanosecond resolution values.

4.5. Scheduling

Since scheduling may be based on precise timing and firing of interrupts, the scheduling algorithms of an operating system may be adversely affected by virtualization. In theory, the effect is random and should be universally distributed, but in contrived as well as real scenarios (guest device access, causes of virtualization exits, possible context switch), this may not always be the case. The effect of this has not been well studied.

In an attempt to work around this, several implementations have provided a paravirtualized scheduler clock, which reveals the true amount of CPU time for which a virtual machine has been running.

4.6. Watchdogs

Watchdog timers, such as the lock detector in Linux may fire accidentally when running under hardware virtualization due to timer interrupts being delayed or misinterpretation of the passage of real time. Usually, these warnings are spurious and can be ignored, but in some circumstances it may be necessary to disable such detection.

4.7. Delays and precision timing

Precise timing and delays may not be possible in a virtualized system. This can happen if the system is controlling physical hardware, or issues delays to compensate for slower I/O to and from devices. The first issue is not solvable in general for a virtualized system; hardware control software can't be adequately virtualized without a full real-time operating system, which would require an RT aware virtualization platform.

The second issue may cause performance problems, but this is unlikely to be a significant issue. In many cases these delays may be eliminated through configuration or paravirtualization.

4.8. Covert channels and leaks

In addition to the above problems, time information will inevitably leak to the guest about the host in anything but a perfect implementation of virtualized time. This may allow the guest to infer the presence of a hypervisor (as in a red-pill type detection), and it may allow information to leak between guests by using CPU utilization itself as a signalling channel. Preventing such problems would require completely isolated virtual time which may not track real time any longer. This may be useful in certain security or QA contexts, but in general isn't recommended for real-world deployment scenarios.