Timer

Timers are mainly useful if we want to develop a multitasking kernel, the timers are used to keep track of the time a task has been running and decided wheter it has to give space to another task in the queue, or not.

In this chapter we are going to see how to enable and configure a timer, using the one provided by the APIC, and having an IRQ generated everytime the timer expires.

Types

There are different sources when talking about timers on modern computer, it depends on the architecture, and the hardware, in this this document we are going to set-up the Apic Timer, but to calibrate it properly we are going to use another timer the PIT.

The Programmable Interrupt Timer (aka PIT) is a legacy hardware present in x86-* architectures (in more modern architecture is not really present but it is emulated) that was once used to generate an irq everytime a counter reaches zero. And the irq were sent to another hardware that was responsible of handling the hardware interrupt requests (IRQs) the PIC8259.

On more modern hardware the PIC processor was replaced by the APIC, and this one came with it's own timer.

But here the catch: one of the easiest way to calibrate properly the Apic timer is still to use the legacy PIT.

Few words about the PIT

Even if we are going to use the PIT only to set-up correctly the APIC timer, is worth spending few words on it. Especially to understand why we are going to use certain values.

First the PIT has basically 4 communication ports:

I/O Port	Description
0x40	Channel 0 data Port
0x41	Channel 1 data Port
0x42	Channel 2 data Port
0x43	Mode/Command register

The PIT 8253/4 Chips has a strange clock rate, that is 1,193,180 Hz that is more or less 1,19Mhz, the reason behind it is again some legacy from the past, you can read it on the OSDEV Wiki page about the PIT (See the useful links section). Reminder: 1 Hz is number of cycles per second.

The pit counter is 16bits, this means it can't count up to 1 second, because the clock rate is more than 16 bits. So we must count to fractions of it.

Everytime the the counter reaches 0, an irq is generated. So let's say for example we want to create an interrupt every millisecond (1/1000 of second) to know how many cycles are needed we need to divide the clock rate by the duration we want (1ms):

```
1,193180 (clock rate) / 1000 (1ms) = 1193.18 (cycles in 1ms)
```

Now one problem is that we can't use decimal number so we need to round up to the closer integer, that is in this case 1193. but that means that we lose accuracy... Yeah, and there is not much that can be done about it.

The programming of the PIT is pretty straightforward, there is only one configuration byte, and basically just one command to send. For more information about programming the PIT please refer to the useful links section, here i will give a very short explain of the configuration byte, and what we values are we going to set for our calibration purpose.

The table below shows theocnfiguration byte:

Bits	Description
0	It describe how the channel will operate if in Binary mode or BCD Mode, for our purpose we will use Binary mode.
1 - 3	Operating Mode there are basically 5 operating modes, we are going to use the "rate generator", identified by 010
4 - 5	Access mode it tells how the channel how to read/write the counter register. It can be: low/high/low first then high
6 - 7	Select the channel we want to use, consider that channel 1 is unavailable. We are going to use channel 0

In our case we want:

- Binary Mode (0)
- Operating mode 2: (010)
- Access mode: Low first then high (11)
- channel 0: (00)

That translates into byte: 00110100.

With that byte now we must:

- Write it into the pit using Mode command register the port (0x43)
- Send two consecutive writes to the channel 0 data port (0x40), with the two bytes for the counter (low first then high), the value of the counter depends on how much time you want between IRQs, for example if we want 1ms of delay between each IRQ then we need to write the value 1193,

in hexadecimal: 0x4A9, so we will send the lower byte First 0xA9 followed by the higher byte: 0x04.

And remember: we need to set an IRQ handler for the PIT Irqs, for how to do this please refer to the APIC chapter

Why we need the calibration?

Because according to the intel ia32 documentation the APIC frequency is the same of the bus frequency or the core crystal frequency (divided by the chosen frequecny divider) so this basically depend on the hardware we are running.

There are different ways to calibrate the apic timer, the one explained here is the most used and easier to understand.

IRQ

The PIT timer is connected to the old PIC8259 IRQ0 pin, now if we are using the APIC, this line is connected to the Redirection Table entry number #2 (offset: 14h and 15h).

While the APIC timer irq is always using the lapic LVT entry 0.

Steps for calibration

These are at a high level the steps that we need to do to calibrate the APIC timer:

1. Configure the PIT Timer
2. Configure the APIC timer
3. Reset the APIC counter
4. Wait some time that is measured using another timing source (in our case the PIT)
5. Compute the number of ticks from the APIC counter
6. Adjust it to the desired measure
7. Divide it by the divider chosen, use this value to raise an interrupt every x ticks
8. Mask the PIT Timer IRQ

Configure the PIT Timer (1)

Refer to the paragraph above, what we want to achieve on this step is configure the Pit timer to generate an interrupt with a specific interval (in our example 1ms), and configure an IRQ handler for it.

Configure the PIT Timer: IRQ Handling The irq handling will be pretty easy, we just need to increment a global counter variable, that will count how many ms passed (the "how many" will depend on how you configured the pit),

just keep in mind that this variable must be declared as volatile. The irq handling function will be as simple as it seems:

```
void irq_handler() {
    pit_ticks++;
}
```

Configure the APIC Timer (2)

This step is just an initialization step for the apic, before proceeding make sure that the timer is stopped, to do that just write 0 to the Initial Count Register.

The APIC Timer has 3 main register:

- The Initial Count Register at Address 0xFEEO 0380 (this has to be loaded with the initial value of the timer counter, every time time the counter it reaches 0 it will generate an Interrupt
- The current Count Register at Address 0xFEE0 0390 (current value of the counter)
- The Divide Configuration register at 0xFEE0 03E0 The Timer divider.

In this step we need first to configure the timer registers:

- The Initial Count register should start from the highest value possible. Since all the apic registers are 32 bit, the maximum value possible is: (0xFFFFFFFF)
- The Current Count register doesn't need to be set, it is set automatically when we initialize the initial count. This register basically is a countdown register.
- The divider configuration register it's up to us (i used 2)

Wait some time (4)

In this step we just need to make an active waiting, basically something like:

```
while(pitTicks < TIME_WE_WANT_TO_WAIT) {
    // Do nothing...
}
```

pitTicks is a global variable set to be increased every time an IRQ from the PIT has been fired, and that depends on how the PIT is configured. So if we configured it to generate an IRQ every 1ms, and we want for example 15ms, will need the vlaue for `TIME_WE_WANT_TO_WAIT` will be 15.

Compute the number of ticks from the APIC Counter and obtain the calibrated value (5,6,7)

That step is pretty easy, we need to read the APIC current count register (offset: 390).

This register will tell us how many ticks are left before the register reaches 0. So to geth the number of ticks done so far we need to:

```
apic_ticks_done = initial_count_reg_value - current_count_reg_value
```

this number tells us how many apic ticks were done in the `TIME_WE_WANT_TO_WAIT`. Now if we do the following division:

```
apic_ticks_done / TIME_WE_WANT_TO_WAIT;
```

we will get the ticks done in unit of time. This value can be used for the initial count register as it is (or in multiple depending on what frequency we want the timer interrupt to be fired)

This is the last step of the calibration, we can start the apic timer immediately with the new calibrated value for the initial count, or do something else before. But at this point the PIT can be disabled, and no longer used.

Useful links

- Ehtereality osdev Notes - Apic/Timing/Context Switching
- OSdev Wiki - Pit page
- Brokern Thron Osdev Series