Programmable Interval Timer

From OSDev Wiki

The **Programmable Interval Timer** (**PIT**) chip (also called an 8253/8254 chip) basically consists of an oscillator, a prescaler and 3 independent frequency dividers. Each frequency divider has an output, which is used to allow the timer to control external circuitry (for example, IRQ 0).

Contents

The Oscillator

The oscillator used by the PIT chip runs at (roughly) 1.193182 MHz. The reason for this requires a trip back into history (to the latter half of the 1970's)...

The original PC used a single "base oscillator" to generate a frequency of 14.31818 MHz because this frequency was commonly used in television circuitry at the time. This base frequency was divided by 3 to give a frequency of 4.77272666 MHz that was used by the CPU, and divided by 4 to give a frequency of 3.579545 MHz that was used by the CGA video controller. By logically ANDing these signals together a frequency equivalent to the base frequency divided by 12 was created. This frequency is 1.1931816666 MHz (where the 6666 part is recurring). At the time it was a brilliant method of reducing costs, as the 14.31818 MHz oscillator was cheap due to mass production and it was cheaper to derive the other frequencies from this than to have several oscillators. In modern computers, where the cost of electronics is much less, and the CPU and video run at much higher frequencies the PIT lives on as a reminder of "the good ole' days".

Frequency Dividers

The basic principle of a frequency divider is to divide one frequency to obtain a slower frequency. This is typically done by using a counter. Each "pulse" from the input frequency causes the counter to be decreased, and when that counter has reached zero a pulse is generated on the output and the counter is reset. For example, if the input signal is 200 Hz and the counter is reset to a value of ten each time, then the output frequency would be 200/10, or 20 Hz.

The PIT has only 16 bits that are used as frequency divider, which can represent the values from 0 to 65535. Since the frequency can't be divided by 0 in a sane way, many implementations use 0 to represent the value 65536 (or 10000 when programmed in BCD mode)

The PIT chip has three separate frequency dividers (or 3 separate channels) that are programmable, in that the value of the "reset counter" is set by software (the OS). Software also specifies an action to be taken when the counter reaches zero on each individual channel. In this way, each channel can be used in one of several "modes" - for example, as a frequency divider (where the count is automatically reset) or as a "one shot" timer (where the count isn't automatically reset).

Each PIT channel also has a "gate input" pin which can be used to control whether the input signal (the 1.19MHz one) gets to the channel or not. For PIT channels 0 and 1, the associated gate input pin is not connected to anything. The PIT channel 2 gate is controlled by IO port 0x61, bit 0.

PIT Timer Accuracy

The accuracy of the PIT timer depends on the quality of the oscillator used, and is typically accurate to within +/- 1.73 seconds per day. There are many causes for this inaccuracy, however because of this there isn't much point in specifying times or frequencies to more than five or six digits.

Outputs

Channel 0 is connected directly to IRQ0, so it is best to use it only for purposes that should generate interrupts. Channel 1 is unusable, and may not even exist. Channel 2 is connected to the PC speaker, but can be used for other purposes without producing audible speaker tones.

Channel 0

The output from PIT channel 0 is connected to the PIC chip, so that it generates an "IRQ 0". Typically during boot the BIOS sets channel 0 with a count of 65535 or 0 (which translates to 65536), which gives an output frequency of 18.2065 Hz (or an IRQ every 54.9254 mS). Channel 0 is probably the most useful PIT channel, as it is the only channel that is connected to an IRQ. It can be used to generate an infinte series of "timer ticks" at a frequency of your choice (as long as it is higher than 18 Hz), or to generate single CPU interrupts (in "one shot" mode) after programmable short delays (less than an 18th of a second).

When choosing an operating mode, below, it is useful to remember that the IRQ0 is generated by the *rising edge* of the Channel 0 output voltage (ie. the transition from "low" to "high", only).

Channel 1

The output for PIT channel 1 was once used (in conjunction with the DMA controller's channel 0) for refreshing the DRAM (Dynamic Random Access Memory) or RAM. Typically, each bit in RAM consists of a capacitor which holds a tiny charge representing the state of that bit, however (due to leakage) these capacitors need to be "refreshed" periodically so that they don't forget their state.

On later machines, the DRAM refresh is done with dedicated hardware and the PIT (and DMA controller) is no longer used. On modern computers where the functionality of the PIT is implemented in a large scale integrated circuit, PIT channel 1 is no longer usable and may not be implemented at all.

Channel 2

The output of PIT channel 2 is connected to the PC speaker, so the frequency of the output determines the frequency of the sound produced by the speaker. This is the only channel where the gate input can be controlled by software (via bit 0 of

I/O port 0x61), and the only channel where its output (a high or low voltage) can be read by software (via bit 5 of I/O port 0x61). Details of how to program the PC speaker can be found here.

I/O Ports

The PIT chip uses the following I/O ports:

```
I/O port     Usage
0x40         Channel 0 data port (read/write)
0x41         Channel 1 data port (read/write)
0x42         Channel 2 data port (read/write)
0x43         Mode/Command register (write only, a read is ignored)
```

Each 8 bit data port is the same, and is used to set the counter's 16 bit reload value or read the channel's 16 bit current count (more on this later). The PIT channel's current count and reload value should not be confused. In general, when the current count reaches zero the PIT channel's output is changed and the current count is reloaded with the reload value, however this isn't always the case. How the current count and reload value are used and what they contain depends on which mode the PIT channel is configured to use.

The Mode/Command register at I/O address 0x43 contains the following:

```
Bits         Usage
 6 and 7      Select channel :
                 0 0 = Channel 0
                 0 1 = Channel 1
                 1 0 = Channel 2
                 1 1 = Read-back command (8254 only)
 4 and 5      Access mode :
                 0 0 = Latch count value command
                 0 1 = Access mode: lobyte only
                 1 0 = Access mode: hibyte only
                 1 1 = Access mode: lobyte/hibyte
 1 to 3       Operating mode :
                 0 0 0 = Mode 0 (interrupt on terminal count)
                 0 0 1 = Mode 1 (hardware re-triggerable one-shot)
                 0 1 0 = Mode 2 (rate generator)
                 0 1 1 = Mode 3 (square wave generator)
                 1 0 0 = Mode 4 (software triggered strobe)
                 1 0 1 = Mode 5 (hardware triggered strobe)
                 1 1 0 = Mode 2 (rate generator, same as 010b)
                 1 1 1 = Mode 3 (square wave generator, same as 011b)
 0            BCD/Binary mode: 0 = 16-bit binary, 1 = four-digit BCD
```

The "Select Channel" bits select which channel is being configured, and must always be valid on every write of the mode/command register, regardless of the other bits or the type of operation being performed. The "read back" (both bits set) is not supported on the old 8253 chips but should be supported on all AT and later computers except for PS/2 (i.e. anything that isn't obsolete will support it).

The "read back" command is discussed later.

The "Access Mode" bits tell the PIT what access mode you wish to use for the selected channel, and also specify the "counter latch" command to the CTC (more on the "counter latch" command latter). These bits must be valid on every write to the mode/command register. For the "read back" command (also discussed later), these bits have a different meaning. For the remaining combinations, these bits specify what order data will be read and written to the data port for the associated PIT channel. Because the data port is an 8 bit I/O port and the values involved are all 16 bit, the PIT chip needs to know what byte each read or write to the data port wants. For "lobyte only", only the lowest 8 bits of the counter value is read or written to/from the data port. For "hibyte only", only the highest 8 bits of the counter value is read or written. For the "lobyte/hibyte" mode, 16 bits are always transferred as a pair, with the lowest 8 bits followed by the highest 8 bits (both 8 bit transfers are to the **same** IO port, sequentially -- a word transfer will not work).

The "Operating Mode" bits specify which mode the selected PIT channel should operate in. For the "read back" command and the "counter latch" command, these bits have different meanings (see the information corresponding to these commands below). There are 6 different operating modes. Each operating mode will be discussed separately later.

The "BCD/Binary" bit determines if the PIT channel will operate in binary mode or BCD mode (where each 4 bits of the counter represent a decimal digit, and the counter holds values from '0000' to '9999'). 80x86 PCs only use binary mode (BCD mode is ugly and limits the range of counts/frequencies possible). Although it should still be possible to use BCD mode, it may not work properly on some "compatible" chips. For the "read back" command and the "counter latch" command, this bit has different meanings (see the information corresponding to these commands below).

Operating Modes

While each operating mode behaves differently, some things are common to all operating modes. This includes:

Initial Output State
: Every time the mode/command register is written to, all internal logic in the selected PIT channel is reset, and the output immediately goes to its initial state (which depends on the mode).

Changing Reload Value
: A new reload value can be written to a PIT channel's data port at any time. The operating mode determines the exact effect that this will have.

Current Counter

The current counter value is always either decremented or reset to the reload value on the *falling* edge of the (1.193182 MHz) input signal.

Current Counter Reload

In modes where the current count is decremented when it is reloaded, the current count is not decremented on the same input clock pulse as the reload - it starts decrementing on the *next* input clock pulse.

Mode 0 - Interrupt On Terminal Count

For this mode, when the mode/command register is written the output signal goes low and the PIT waits for the reload register to be set by software, to begin the countdown. After the reload register has been set, the current count will be set to the reload value on the next falling edge of the (1.193182 MHz) input signal. Subsequent falling edges of the input signal will decrement the current count (if the gate input is high on the preceding rising edge of the input signal).

When the current count decrements from one to zero, the output goes high and remains high until another mode/command register is written or the reload register is set again. The current count will wrap around to 0xFFFF (or 0x9999 in BCD mode) and continue to decrement until the mode/command register or the reload register are set, however this will not effect the output pin state.

The reload value can be changed at any time. In "lobyte/hibyte" access mode counting will stop when the first byte of the reload value is set. Once the full reload value is set (in any access mode), the next falling edge of the (1.193182 MHz) input signal will cause the new reload value to be copied into the current count, and the countdown will continue from the new value.

Note: despite the misleading name of this mode, it only generates interrupts on channel 0.

Mode 1 - Hardware Re-triggerable One-shot

This mode is similar to mode 0 above, however counting doesn't start until a rising edge of the gate input is detected. For this reason it is not usable for PIT channels 0 or 1 (where the gate input can't be changed).

When the mode/command register is written the output signal goes high and the PIT waits for the reload register to be set by software. After the reload register has been set the PIT will wait for the next rising edge of the gate input. Once this occurs, the output signal will go low and the current count will be set to the reload value on the next falling edge of the (1.193182 MHz) input signal. Subsequent falling edges of the input signal will decrement the current count.

When the current count decrements from one to zero, the output goes high and remains high until another mode/command register is written or the reload

register is set again. The current count will wrap around to 0xFFFF (or 0x9999 in BCD mode) and continue to decrement until the mode/command register or the reload register are set, however this will not effect the output pin state.

If the gate input signal goes low during this process it will have no effect. However, if the gate input goes high again it will cause the current count to be reloaded from the reload register on the next falling edge of the input signal, and restart the count again (the same as when counting first started).

The reload value can be changed at any time, however the new value will not affect the current count until the current count is reloaded (on the next rising edge of the gate input). So if you want to do this, clear and then reset bit 0 of IO port 0x61, after modifying the reload value.

Mode 2 - Rate Generator

This mode operates as a frequency divider.

When the mode/command register is written the output signal goes high and the PIT waits for the reload register to be set by software. After the reload register has been set, the current count will be set to the reload value on the next falling edge of the (1.193182 MHz) input signal. Subsequent falling edges of the input signal will decrement the current count (if the gate input is high on the preceding rising edge of the input signal).

When the current count decrements from two to one, the output goes low, and on the next falling edge of the (1.193182 MHz) input signal it will go high again and the current count will be set to the reload value and counting will continue.

If the gate input goes low, counting stops and the output goes high immediately. Once the gate input has returned high, the next falling edge on input signal will cause the current count to be set to the reload value and operation will continue.

The reload value can be changed at any time, however the new value will not effect the current count until the current count is reloaded (when it is decreased from two to one, or the gate input going low then high). When this occurs counting will continue using the new reload value.

A reload value (or divisor) of one must *not* be used with this mode.

This mode creates a high output signal that drops low for one input signal cycle (0.8381 uS), which is too fast to make a difference to the PC speaker (see mode 3). For this reason mode 2 is useless for producing sounds with PIT channel 2.

Typically, OSes and BIOSes use mode 3 (see below) for PIT channel 0 to generate IRQ 0 timer ticks, but some use mode 2 instead, to gain frequency accuracy (frequency = 1193182 / reload_value Hz).

Mode 3 - Square Wave Generator

For mode 3, the PIT channel operates as a frequency divider like mode 2, however the output signal is fed into an internal "flip flop" to produce a square wave (rather than a short pulse). The flip flop changes its output state each time its input state (or the output of the PIT channel's frequency divider) changes. This causes the actual output to change state half as often, so to compensate for this the current count is decremented twice on each falling edge of the input signal (instead of once), and the current count is set to the reload value twice as often.

When the mode/command register is written the output signal goes high and the PIT waits for the reload register to be set by software. After the reload register has been set, the current count will be set to the reload value on the next falling edge of the (1.193182 MHz) input signal. Subsequent falling edges of the input signal will decrement the current count twice (if the gate input is high on the preceding rising edge of the input signal).

Note: under normal circumstances the output state will be low 50% of the time when the mode/command register is written. The output will then go high, which will generate an immediate (perhaps spurious) IRQ0. The other 50% of the time the output will already be high, and there will be no IRQ0 generated.

For even reload values, when the current count decrements from two to zero the output of the flop-flop changes state; the current count will be reset to the reload value and counting will continue.

For odd reload values, the current count is always set to one less than the reload value. If the output of the flip flop is low when the current count decrements from two to zero it will behave the same as the equivalent even reload value. However, if the output of the flip flop is high the reload will be delayed for one input signal cycle (0.8381 uS), which causes the "high" pulse to be slightly longer and the duty cycle will not be exactly 50%. Because the reload value is rounded down to the nearest even number anyway, it is recommended that only even reload values be used (which means you should mask the value before sending it to the port).

Note: This even value limitation on the reload value in mode 3 reduces the number of possible output frequencies in half. If you want to be able to control the frequency of IRQ0 to a somewhat higher degree, then think about using mode 2 instead for channel 0.

On channel 2, if the gate input goes low, counting stops and the output goes high immediately. Once the gate input has returned high, the next falling edge on input signal will cause the current count to be set to the reload value and operation will continue (with the output left high).

The reload value can be changed at any time, however the new value will not effect the current count until the current count is reloaded (when it is decreased

from two to zero, or the gate input going low then high). When this occurs counting will continue using the new reload value.

A reload value (or divisor) of one must *not* be used with this mode.

Mode 4 - Software Triggered Strobe

Mode four operates as a retriggerable delay, and generates a pulse when the current count reaches zero.

When the mode/command register is written the output signal goes high and the PIT waits for the reload register to be set by software. After the reload register has been set, the current count will be set to the reload value on the next falling edge of the (1.193182 MHz) input signal. Subsequent falling edges of the input signal will decrement the current count (if the gate input is high on the preceding rising edge of the input signal).

When the current count decrements from one to zero, the output goes low for one cycle of the input signal (0.8381 uS). The current count will wrap around to 0xFFFF (or 0x9999 in BCD mode) and continue to decrement until the mode/command register or the reload register are set, however this will not affect the output state.

If the gate input goes low, counting stops but the output will not be affected and the current count will not be reset to the reload value.

The reload value can be changed at any time. When the new value has been set (both bytes for "lobyte/hibyte" access mode) it will be loaded into the current count on the next falling edge of the (1.193182 MHz) input signal, and counting will continue using the new reload value.

Mode 5 - Hardware Triggered Strobe

Mode 5 is similar to mode 4, except that it waits for the rising edge of the gate input to trigger (or re-trigger) the delay period (like mode 1). For this reason it is not usable for PIT channels 0 or 1 (where the gate input can't be changed).

When the mode/command register is written the output signal goes high and the PIT waits for the reload register to be set by software. After the reload register has been set the PIT will wait for the next rising edge of the gate input. Once this occurs, the current count will be set to the reload value on the next falling edge of the (1.193182 MHz) input signal. Subsequent falling edges of the input signal will decrement the current count.

When the current count decrements from one to zero, the output goes low for one cycle of the input signal (0.8381 uS). The current count will wrap around to

0xFFFF (or 0x9999 in BCD mode) and continue to decrement until the mode/command register or the reload register are set, however this will not effect the output state.

If the gate input signal goes low during this process it will have no effect. However, if the gate input goes high again it will cause the current count to be reloaded from the reload register on the next falling edge of the input signal, and restart the count again (the same as when counting first started).

The reload value can be changed at any time, however the new value will not affect the current count until the current count is reloaded (on the next rising edge of the gate input). When this occurs counting will continue using the new reload value.

Counter Latch Command

To prevent the current count from being updated, it is possible to "latch" a PIT channel using the latch command. To do this, send the value CC000000 (in binary) to the mode/command register (I/O port 0x43), where 'CC' corresponds to the channel number. When the latch command has been sent, the current count is copied into an internal "latch register" which can then be read via. the data port corresponding to the selected channel (I/O ports 0x40 to 0x42). The value kept in the latch register remains the same until it has been fully read, or until a new mode/command register is written.

The main benefit of the latch command is that it allows both bytes of the current count to be read without inconsistencies. For example, if you didn't use the latch command, then the current count may decrease from 0x0200 to 0x01FF after you've read the low byte but before you've read the high byte, so that your software thinks the counter was 0x0100 instead of 0x0200 (or 0x01FF).

While the latch command should not affect the current count, on some (old/dodgy) motherboards sending the latch command can cause a cycle of the input signal to be occasionally missed, which would cause the current count to be decremented 0.8381ms later than it should be. If you're sending the latch command often this could cause accuracy problems (but if you need to send the latch command often you may wish to consider redesigning your code anyway).

Read Back Command

The read back command is a special command sent to the mode/command register (I/O port 0x43). The "read back" is not supported on the old 8253 chips but should be supported on all AT and later computers except for PS/2 (i.e. anything that isn't obsolete will support it).

For the read back command, the mode/command register uses the following format:

```
Bits         Usage
 7 and 6      Must be set for the read back command
 5            Latch count flag (0 = latch count, 1 = don't latch count)
 4            Latch status flag (0 = latch status, 1 = don't latch status)
 3            Read back timer channel 2 (1 = yes, 0 = no)
 2            Read back timer channel 1 (1 = yes, 0 = no)
 1            Read back timer channel 0 (1 = yes, 0 = no)
 0            Reserved (should be clear)
```

Note: Be careful with bits 4 and 5 - they are inverted.

Bits 1 to 3 of the read back command select which PIT channels are affected, and allow multiple channels to be selected at the same time.

If bit 5 is clear, then any/all PIT channels selected with bits 1 to 3 will have their current count copied into their latch register (similar to sending the latch command, except it works for multiple channels with one command).

If bit 4 is clear, then for any/all PIT channels selected with bits 1 to 3, the next read of the corresponding data port will return a status byte (discussed below).

Read Back Status Byte

After sending a read back command with bit 4 clear, reading the data port for each selected channel will return a status value with the following format:

```
Bit/s        Usage
 7            Output pin state
 6            Null count flags
 4 and 5      Access mode :
                 0 0 = Latch count value command
                 0 1 = Access mode: lobyte only
                 1 0 = Access mode: hibyte only
                 1 1 = Access mode: lobyte/hibyte
 1 to 3       Operating mode :
                 0 0 0 = Mode 0 (interrupt on terminal count)
                 0 0 1 = Mode 1 (hardware re-triggerable one-shot)
                 0 1 0 = Mode 2 (rate generator)
                 0 1 1 = Mode 3 (square wave generator)
                 1 0 0 = Mode 4 (software triggered strobe)
                 1 0 1 = Mode 5 (hardware triggered strobe)
                 1 1 0 = Mode 2 (rate generator, same as 010b)
                 1 1 1 = Mode 3 (square wave generator, same as 011b)
 0            BCD/Binary mode: 0 = 16-bit binary, 1 = four-digit BCD
```

The bottom six bits return the values that where programmed into the mode/command register when the channel was last initialized.

Bit 7 indicates the state of the PIT channel's output pin at the moment that the

read-back command was issued.

Bit 6 indicates whether a newly-programmed divisor value has been loaded into the current count yet (if clear) or the channel is still waiting for a trigger signal or for the current count to count down to zero before a newly programmed reload value is loaded into the current count (if set). This bit is set when the mode/command register is initialized or when a new reload value is written, and cleared when the reload value is copied into the current count.

Reading The Current Count

To read the current count using the "lobyte only" or "hibyte only" access modes, you can just do an "in al,0x40" (for PIT channel 0) without problems. For frequencies higher than 4.7 KHz it can be easiest to set the high byte of the reload value to zero, and then use the "lobyte only" access mode to minimize hassles.

For the "lobyte/hibyte" access mode you need to send the latch command (described above) to avoid getting wrong results. If any other code could try set the PIT channel's reload value or read its current count after you've sent the latch command but before you've read the highest 8 bits, then you have to prevent it. Disabling interrupts works for single CPU computers. For example, to read the count of PIT channel 0 you could use something like:

```
read_PIT_count:
        pushfd
        cli
        mov al, 00000000b    ; al = channel in bits 6 and 7, remair
        out 0x43, al         ; Send the latch command

        in al, 0x40          ; al = low byte of count
        mov ah, al           ; ah = low byte of count
        in al, 0x40          ; al = high byte of count
        rol ax, 8            ; al = low byte, ah = high byte (ax =
        popfd
        ret
```

Setting The Reload Value

To set the reload value, just send the value/s to the corresponding data port. For the "lobyte only" or "hibyte only" access modes this only takes a single "out 0x40,al" (for PIT channel 0).

For the "lobyte/hibyte" access mode you need to send the low 8 bits followed by

the high 8 bits. You must prevent other code from setting the PIT channel's reload value or reading its current count once you've sent the lowest 8 bits. Disabling interrupts works for single CPU computers. For example:

```
set_PIT_count:
        pushfd
        cli
        out 0x40, al        ; Set low byte of reload value
        rol ax, 8           ; al = high byte, ah = low byte
        out 0x40, al        ; Set high byte of reload value
        rol ax, 8           ; al = low byte, ah = high byte (ax = c
        popfd
        ret
```

It should be noted that a reload value of zero can be used to specify a divisor of 65536. This is how the BIOS gets an IRQ 0 frequency as low as 18.2065 Hz.

PIT Channel 0 Example Code

The following example code was written for NASM, but hasn't been tested.

The idea is to provide a single routine to initialize PIT channel 0 for any (possible) frequency and use IRQ 0 to accurately keep track of real time in milliseconds since the PIT was configured.

For the sake of accuracy, the initialization code will calculate the number of whole milliseconds to add to the "system timer tick" each IRQ, and the number of "fractions of a millisecond" to avoid drift. This may be important, for example if the PIT is set for 700 Hz it'd work out to (roughly) 1.42857 mS between IRQs, so keeping track of whole milliseconds only would lead to huge inaccuracies.

Hopefully, everyone is familiar with fixed point mathematics. For example, with the "32.32" notation I'll be using, if the high dword is equal to 0x00000001 and the low dword is equal to 0x80000000 then the combined value would be 1.5. In a similar way, the fraction 0.75 is represented with 0xC000000, 0.125 is represented with 0x20000000 and 0.12345 would be represented with 0x1F9A6B50.

To begin with, this following code contains all of the data used by this example. It is assumed that the ".bss" section is filled with zeros.

```
section .bss
system_timer_fractions:  resd 1         ; Fractions of 1 mS since
system_timer_mS:         resd 1         ; Number of whole mS since
IRQ0_fractions:          resd 1         ; Fractions of 1 mS betwee
```

```
IRQ0_mS:                resd 1          ; Number of whole mS betwe
IRQ0_frequency:         resd 1          ; Actual frequency of PIT
PIT_reload_value:       resw 1          ; Current PIT reload value
section .text
```

Next is the handler for IRQ 0. It's fairly simple (all it does it add 2 64 bit fixed point values and send an EOI to the PIC chip).

```
IRQ0_handler:
        push eax
        push ebx

        mov eax, [IRQ0_fractions]
        mov ebx, [IRQ0_mS]                      ; eax.ebx = amount of
        add [system_timer_fractions], eax       ; Update system timer
        adc [system_timer_mS], ebx              ; Update system timer

        mov al, 0x20
        out 0x20, al                            ; Send the EOI to the

        pop ebx
        pop eax
        iretd
```

Now for the tricky bit - the initialization routine. The PIT can't generate some frequencies. For example if you want 8000 Hz then you've got a choice of 8007.93 Hz or 7954.544 Hz. In this case the following code will find the nearest possible frequency. Once it has calculated the nearest possible frequency it will reverse the calculation to find the actual frequency selected (rounded to the nearest integer, intended for display purposes only).

For some extra accuracy, I also use "3579545 / 3" instead of 1193182 Hz. This is mostly pointless due to inaccurate hardware (I just like being correct).

```
;Input
; ebx   Desired PIT frequency in Hz

init_PIT:
   pushad

   ; Do some checking

   mov eax,0x10000                  ;eax = reload value for slowe
```

```
   cmp ebx,18                      ;Is the requested frequency t
   jbe .gotReloadValue             ; yes, use slowest possible f

   mov eax,1                       ;ax = reload value for fastes
   cmp ebx,1193181                 ;Is the requested frequency t
   jae .gotReloadValue             ; yes, use fastest possible f

   ; Calculate the reload value

   mov eax,3579545
   mov edx,0                       ;edx:eax = 3579545
   div ebx                         ;eax = 3579545 / frequency, e
   cmp edx,3579545 / 2             ;Is the remainder more than h
   jb .l1                          ; no, round down
   inc eax                         ; yes, round up
.l1:
   mov ebx,3
   mov edx,0                       ;edx:eax = 3579545 * 256 / fr
   div ebx                         ;eax = (3579545 * 256 / 3 * 2
   cmp edx,3 / 2                   ;Is the remainder more than h
   jb .l2                          ; no, round down
   inc eax                         ; yes, round up
.l2:


; Store the reload value and calculate the actual frequency

.gotReloadValue:
   push eax                        ;Store reload_value for later
   mov [PIT_reload_value],ax       ;Store the reload value for l
   mov ebx,eax                     ;ebx = reload value

   mov eax,3579545
   mov edx,0                       ;edx:eax = 3579545
   div ebx                         ;eax = 3579545 / reload_value
   cmp edx,3579545 / 2             ;Is the remainder more than h
   jb .l3                          ; no, round down
   inc eax                         ; yes, round up
.l3:
   mov ebx,3
   mov edx,0                       ;edx:eax = 3579545 / reload_v
   div ebx                         ;eax = (3579545 / 3) / freque
   cmp edx,3 / 2                   ;Is the remainder more than h
   jb .l4                          ; no, round down
   inc eax                         ; yes, round up
.l4:
```

```
   mov [IRQ0_frequency],eax          ;Store the actual frequency i


; Calculate the amount of time between IRQs in 32.32 fixed point
;
; Note: The basic formula is:
;           time in ms = reload_value / (3579545 / 3) * 1000
;       This can be rearranged in the follow way:
;           time in ms = reload_value * 3000 / 3579545
;           time in ms = reload_value * 3000 / 3579545 * (2^42)/(2
;           time in ms = reload_value * 3000 * (2^42) / 3579545 /
;           time in ms * 2^32 = reload_value * 3000 * (2^42) / 357
;           time in ms * 2^32 = reload_value * 3000 * (2^42) / 357

   pop ebx                           ;ebx = reload_value
   mov eax,0xDBB3A062                ;eax = 3000 * (2^42) / 357954
   mul ebx                           ;edx:eax = reload_value * 300
   shrd eax,edx,10
   shr edx,10                        ;edx:eax = reload_value * 300

   mov [IRQ0_mS],edx                 ;Set whole mS between IRQs
   mov [IRQ0_fractions],eax          ;Set fractions of 1 mS betwee


; Program the PIT channel

   pushfd
   cli                               ;Disabled interrupts (just in

   mov al,00110100b                  ;channel 0, lobyte/hibyte, ra
   out 0x43, al

   mov ax,[PIT_reload_value]         ;ax = 16 bit reload value
   out 0x40,al                       ;Set low byte of PIT reload v
   mov al,ah                         ;ax = high 8 bits of reload v
   out 0x40,al                       ;Set high byte of PIT reload

   popfd

   popad
   ret
```

Note: you also need to install an IDT entry for IRQ 0, and unmask it in the PIC chip (or I/O APIC).

Of course it's easier to configure the PIT to a fixed value, but where's the fun in that? :-)

Uses for the Timer IRQ

Using the IRQ to Implement `sleep`

The PIT's generating a hardware interrupt every *n* milliseconds allows you to create a simple timer. Start with a global variable x that contains the delay:

```
 SEGMENT .DATA
 CountDown DD 0
Next, every time the timer interrupt is called, decrement this vari
 SEGMENT .TEXT
 [GLOBAL TimerIRQ]
 TimerIRQ:
     PUSH EAX
     MOV EAX, CountDown
     OR EAX, EAX ; quick way to compare to 0
     JZ TimerDone
     DEC CountDown
 TimerDone:
     POP EAX
     IRETD
Finally, create a function <tt>Sleep</tt> that waits the interval,
 [GLOBAL _Sleep]
 _Sleep:
     PUSH EBP
     MOV EBP, ESP
     PUSH EAX
     MOV EAX, [EBP + 8] ; EAX has value of sole argument
     MOV CountDown, EAX
 SleepLoop:
     CLI ; can't be interrupted for test
     MOV EAX, CountDown
     OR EAX, EAX
     JZ SleepDone
     STI
     NOP ; NOP a few times so the interrupt can get handled
     NOP
     NOP
     NOP
     NOP
     NOP
```

```
    JMP SleepLoop
SleepDone:
    STI
    POP EAX
    POP EBP
    RETN 8
```

In a multitasking system, consider using a linked list or array of these CountDown variables. If your multitasking system supports interprocess communication, you can also store the semaphore/exchange where two processes can talk to, have the interrupt send a message to the waiting process when the timer is done, and have the waiting process block all execution until that message comes:

```
#define COUNTDOWN_DONE_MSG 1
struct TimerBlock {
    EXCHANGE e;
    u32int CountDown;
} timerblocks[20];

void TimerIRQ(void) /* called from Assembly */
{
    u8int i;

    for (i = 0; i < 20; i++)
        if (timerblocks[i].CountDown > 0) {
            timerblocks[i].CountDown--;
            if (timerblocks[i].CountDown == 0)
                SendMessage(timerblocks[i].e, COUNTDOWN_DONE_MESSAC
        }
}

void Sleep(u32int delay)
{
    struct TimerBlock *t;

    if ((t = findTimerBlock()) == nil)
        return;
    t->CountDown = delay;
    WaitForMessageFrom(t->e = getCrntExch());
}
```

In your documentation, note the interval of the timer. For example, if the timer interval is 10 milliseconds per tick, tell the programmer to issue

```
Sleep(100);
```

to sleep for a single second.

Using the IRQ for Preemptive Multitasking

The timer IRQ can also be used to perform preemptive multitasking. To give the currently running task some time to run, set a threshold, for example of 3 ticks. Use a global variable like the one before but go up from 0, and when that variable hits 3, switch tasks. How you do so is up to you.

See Also

Articles

- Time And Date
- RTC

Threads

External Links

- Programmable Interval Timer on Wikipedia
- The PIT: A System Clock (http://www.osdever.net/bkerndev/Docs/pit.htm) on osdever

Categories: Common Devices | Time
