

PC Speaker and Timing Functions

The functions for [PC speaker sound effect](#) playback are inextricably linked to those responsible for game timing due to the fact that they all use the same underlying hardware.

- [A Taste of the Programmable Interval Timer](#)
- [Timekeeping](#)
- [StartSound\(\)](#)
- [PCSpeakerService\(\)](#)
 - [Sound Output](#)
- [WaitHard\(\)](#)
- [WaitSoft\(\)](#)

A Taste of the Programmable Interval Timer

The Intel 8253 Programmable Interval Timer (**PIT**) of the IBM PC was a complex beast. If we limit the scope of our discussion to just the necessary information to understand sound generation, things will be much easier.

At its core, the PIT is a decrementing counter (actually, there are three distinct counter channels, but sound generation only uses one of them). The programmer loads a 16-bit value between 0 and 65,535 into a register in the PIT, and the counter starts decrementing toward zero. Once it reaches zero, it automatically resets to the original value stored in the register and starts over. The countdown rate is derived from an external frequency reference generated by a quartz crystal on the system board. This crystal oscillates at 315/22 MHz (or 14,318,181.81 Hz). This frequency is divided by 12, yielding 1,193,181.81 Hz, which is the rate the PIT counts at.

A sense of poise and rationality

Why does a 315/22 MHz crystal exist in the first place, I hear nobody ask? This is exactly four times the 315/88 MHz color subcarrier frequency of NTSC television broadcasts, and is a crystal frequency that has been widely used since the 1950s. IBM wasn't just planning for the eventuality of connecting the PC to television sets – there was a certain safety in choosing a widely-available and

easily-sourced electrical component as the timing basis for the entire PC ecosystem.

The frequency choice itself dates back to the 1950s, when engineers were trying to modify black-and-white television broadcasts in the United States to include color information without creating interference on older TVs that didn't understand the additional signals. Part of this modification involved reducing the frame rate to 29.97 frames per second – an artifact that survives to this day.

The original PC and PC/XT also divided the 315/22 MHz frequency by three to produce the 4.77 MHz CPU clock – another weird number with a surprisingly pragmatic origin.

If the counter is closer to its starting value than zero, the output of the PIT is a high electrical signal. If the counter is closer to zero, the output is low. The end result is a square wave, high half of the time and low the other half, with the frequency controlled by the value in the PIT register. This square wave signal is amplified and fed into the speaker. The connection to the speaker can be deactivated without changing any timer parameters by opening a gate controlled through – I kid you not – the system's keyboard controller.

The end result sounds something like this:

(Download [MP3](#), [AAC](#), or [WAV](#).)

Note: The actual speaker in most PCs was a chintzy piece of junk buried inside a metal box, so it probably didn't sound as bright and "buzzy" as this rendering. This is a frequent problem¹ when trying to recreate PC speaker sound effects on modern audio hardware.

The programmer is free to change the PIT register values at any time. The current cycle will run to completion (i.e. the counter will reach zero), and then the updated counter value will be used starting on the next cycle.

As a practical example, say we wanted to play a middle C through the speaker. Middle C is 261.626 Hz,² and the PIT clock runs at 1,193,181.81 Hz. Dividing the latter by the former and rounding to the nearest integer value yields 4,561. This is the value that must be written to the PIT to produce the

desired tone. The PIT counter starts at the programmed value, and decrements by 1 each time the external clock ticks. While the counter is above 2,280 ($4,561/2$ as an integer), the output signal is high; when the counter drops below that threshold, the output signal is low. Once the counter reaches 0, it automatically resets to the initial value of 4,561 and the cycle repeats.

The output signal of the PIT in this configuration is a square wave of just about 261.605 Hz, which is as close to the desired frequency as we can get with the granularity offered by the system. To create a higher pitch, say the 1,000 Hz tone they use to censor expletives on TV, the counter value would need to be lower: 1,193. This inverse relationship between sound frequency and PIT counter values is key to reasoning about the behavior of the hardware.

Timekeeping

Earlier it was hinted that the Programmable Interval Timer had three timer channels (0–2), of which only channel 2 was used for PC speaker output. The other two channels have functions that are not directly related to sound generation.

The output of PIT channel 1 is connected to the system's memory controller circuitry. DRAM chips need to be electrically "refreshed" periodically to keep their contents from fading away. This timer output triggers the necessary refresh sequence to keep the memory contents intact. Since programs generally have no idea about the electrical characteristics of the memory installed in the system, timer channel 1 is considered a "do not touch" part of the PIT. Its output cannot be directly read or used, and no good can come from messing with it.

How much, really?

In PC hardware of the era, memory values could last about two milliseconds before they began to decay from lack of a refresh. Since all bus communication must be suspended during a refresh, it wasn't practical to pause the machine for long enough to refresh all of the memory every two milliseconds. Instead, smaller regions are refreshed more frequently – a common value

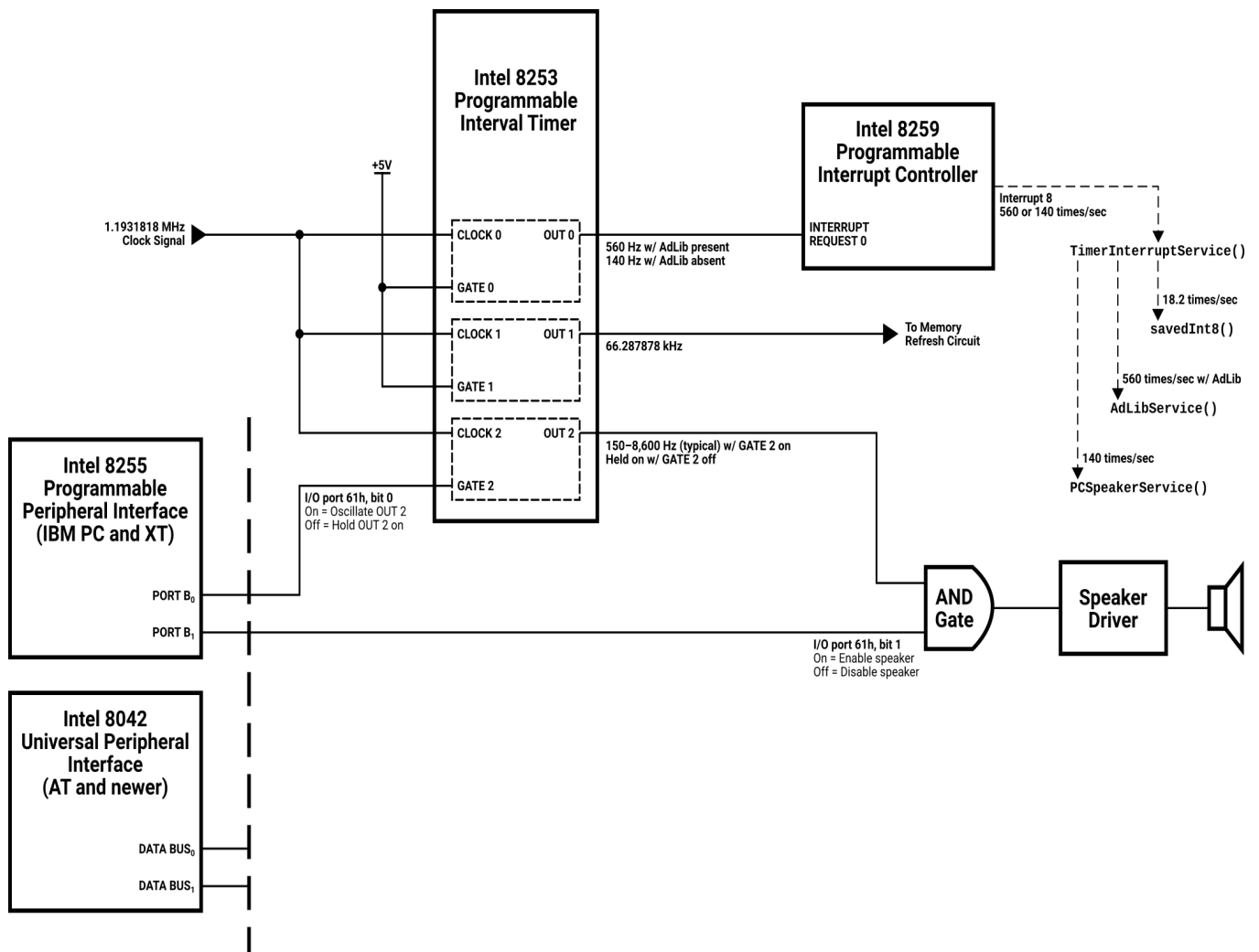
was to refresh 1/128 of the memory every ~15 μ s. This means that PIT channel 1 is firing more than 64,000 times every second!

The output of PIT channel 0 is connected to the system's interrupt controller, and generates interrupt 8 (or IRQ 0) each time it fires. Channel 0's counter is initially loaded with a zero value, which permits the timer countdown to cover the entire span of 65,536 possible values. This results in an output rate of ~18.20651 Hz or one interrupt every 54.92541 ms. This gives software some external sense of the passage of time, no matter how fast the CPU actually performs work.

The system's BIOS tracks the total number of timer interrupts that have occurred since the system was powered on. Older systems (like the IBM PC and PC/XT) lacked any other form of onboard real-time clock circuitry, so DOS had to prompt for the current time/date each time the system booted and perform calculations based on the BIOS timer interrupt count to track the current time of day.

PIT channel 0 was available to be reprogrammed to generate interrupts at any supported frequency, provided care was taken to ensure that the BIOS interrupt handler continued to be called at 18.2 Hz to avoid clock drift and other unpleasantness. The game reprograms this timer channel to run at 140 Hz (or four times that – 560 Hz – if AdLib music output is being used), which serves as the timing basis for PC speaker sound effect playback and constant-time delays.

The following diagram shows the relationship between the timer channels, the hardware that supports them, and the interrupts that connect the timers to software:



Timer Channels, Support Hardware, and Interrupts During Gameplay

In this diagram, timer channels 0 and 2 are shown in their in-game state, and the handler functions for interrupt vector 8 reflect the interrupt handling arrangement once the `Startup()` function has completed all of its initialization tasks. A system capable of running the game would have the Intel 8042 UPI, but the wiring for the older 8255 PPI used in the IBM PC and XT is included to show how the design evolved over time.

StartSound()

The `StartSound()` function queues the new sound effect identified by `sound_num` for playback and immediately returns. It does not block, and it may have no observable effect if a sound with greater priority is already playing.

`sound_num` is a one-based number. A total of 65 sound effects are available, numbered 1 to 65.

```
void StartSound(word sound_num)
{
    if (soundPriority[sound_num] < activeSoundPriority)
        return;

    activeSoundPriority = soundPriority[sound_num];
    isNewSound = true;
    activeSoundIndex = sound_num - 1;
    enableSpeaker = false;
}
```

The `soundPriority[]` of the provided `sound_num` is checked against `activeSoundPriority` – if the currently-playing sound has a higher priority than the sound that wants to be started, nothing occurs and the function returns. (If there is no sound playing, `activeSoundPriority` holds a zero value and the function continues.)

Once it's been determined that the new sound has sufficient priority to replace whatever is currently playing, `activeSoundPriority` is updated to the priority of the new sound. `isNewSound` is set true to inform the PC speaker service that it must reset its playback position so the new sound starts from the beginning.

`activeSoundIndex` holds the zero-indexed sound effect number that is currently playing. It is updated by correcting the one-indexed `sound_num` value. `enableSpeaker` is set false, apparently in an effort to silence any sound that might already be playing, but this assignment has no effect (see `PCSpeakerService()`, in the “if (`isNewSound`)” block).

Great shot kid; that was one in a million.

Since this function does not pause interrupt handling, it's possible for `PCSpeakerService()` to run at absolutely any time – even in the middle of an assignment operation. This means that, for example, the PC speaker service could run after `isNewSound` is

set but before `activeSoundIndex` is updated, causing a single sample of incorrect sound data to play.

If you can get this to occur, and notice an audible effect, you're a much more observant and patient person than I am.

The function returns, having prepared the sound service for playback but without actually making sound yet. That will occur asynchronously after execution has moved on to other things.

PCSpeakerService()

The `PCSpeakerService()` function is called in response to the timer interrupt event and sends a new fragment of sound effect data to the PC speaker. The PC speaker service is called at a constant rate of 140 Hz, regardless of what is happening elsewhere in the program. This service also maintains the game tick counter that governs the speed of the entire game.

```
void PCSpeakerService(void)
{
    static word soundCursor = 0;
    gameTickCount++;
```

`soundCursor` is a counter that tracks the playback position within the currently-playing sound effect. It is private to this function, and holds its value for subsequent calls.

`gameTickCount` is a global free-running counter that is incremented at a constant rate, which occurs here. Other functions in the game (e.g. `WaitHard()` and `WaitSoft()`) rely on this counter to provide consistent delays across all systems the game may run on. Each tick count represents 1/140 of a second.

```
if (isNewSound) {
    isNewSound = false;
    soundCursor = 0;
    enableSpeaker = true;
}
```

If `isNewSound` is true, a new sound effect has been requested to play since the last time the PC speaker service ran. Flip `isNewSound` back to false to

acknowledge the request, reset the `soundCursor` position to the start of the sound, and set `enableSpeaker` true to enable the speaker output.

```
if (*(soundDataPtr[activeSoundIndex] + soundCursor) ==
END_SOUND) {
    enableSpeaker = false;
    activeSoundPriority = 0;

    outportb(0x0061, inportb(0x0061) & ~0x02);
}
```

This block checks for the end marker in the sound effect data. `soundDataPtr[]` holds the pointers to all of the available sound effects, `activeSoundIndex` holds the numeric identifier of the sound that is currently playing, and `soundCursor` holds the offset within that sound. Combining all of these and dereferencing the pointer results in a word value, which could either be a fragment of sound effect data or the sentinel value `END_SOUND` (FFFFh). If `END_SOUND` appears in the data, the end of the sound effect has been reached and playback must stop.

Note: This “end sound” check runs all the time, even if no sound is being played. If `enableSpeaker` is false, `soundCursor` never advances and this block repeatedly tests the same word in the last sound effect that was played. This word tends to be `END_SOUND`, meaning this block executes rather gratuitously while no sounds are active.

In the case where `END_SOUND` is seen, `enableSpeaker` is set to false to disable further output and `activeSoundPriority` is zeroed, allowing any subsequent sound effect to play even if its priority is very low.

The call to `inportb()` reads one byte from the I/O port at address 61h, which addresses the control register of the system’s keyboard controller. The bit in position 1 is turned off, and the resulting value is written back to I/O port 61h via `outportb()`, leaving the other bits at their current values. Bit 1 of port 61h is defined as the “speaker data enable” bit, and setting it to zero disables PC speaker output at the hardware level. This is a quick and effective way to quickly silence sound output.

```
if (enableSpeaker) {
```



```

        /* Send the next piece of sound effect data to the
hardware */
        ...
    } else {
        outportb(0x0061, inportb(0x0061) & ~0x02);
    }
}

```

If `enableSpeaker` is true, read the next piece of sound effect data and reprogram the hardware with it (detailed [below](#)). Otherwise, no sound effect is currently being played and the PC speaker hardware should be silenced; use `inportb()` and `outportb()` to facilitate that. This is an exact duplicate of the earlier speaker control code from the `END_SOUND` check.

This duplication means that, if conditions are right, each run of the PC speaker service could command the already-silent speaker to silence itself twice.

Sound Output

When `enableSpeaker` is true, the meat of the sound effect output is as follows:

```

    word sample = *(soundDataPtr[activeSoundIndex] +
soundCursor);

    if (sample == 0 && isSoundEnabled) {
        outportb(0x0061, inportb(0x0061) & ~0x03);
    } else if (isSoundEnabled) {
        outportb(0x0043, 0xb6);
        outportb(0x0042, (byte) (sample & 0x00ff));
        outportb(0x0042, (byte) (sample >> 8));
        outportb(0x0061, inportb(0x0061) | 0x03);
    }

    soundCursor++;

```

Just like in the `END_SOUND` check, the word value `*(soundDataPtr[activeSoundIndex] + soundCursor)` is the piece of sound effect data that needs to play next. This value is stored in the local `sample` variable.

Each of the subsequent checks requires `isSoundEnabled` to be true (indicating that the user wants to hear sound). When `isSoundEnabled` is false, the user has disabled sound and hardware output is inhibited.

If `sample` holds a zero value, this is an indication that the sound effect has a region of silence. Use the now-familiar pairing of `inportb()` and `outportb()` to turn off bits 0 and 1 at I/O port 61h. These are the “timer 2 gate to speaker enable” and “speaker data enable” bits on the keyboard controller’s control register. This stops the timer from oscillating, and disables the speaker driver. This belt-and-suspenders approach thoroughly silences the speaker.

Otherwise, a nonzero value exists in `sample` which must be written into the programmable interval timer to produce the desired frequency at the speaker. This operation requires a few calls to `outportb()`.

First, the byte B6h is written to I/O port 43h. This port is the control register for the programmable interval timer, and the data byte has the following interpretation:

Bit Position	Value (= B6h)	Interpretation
7–6 (most significant)	10b	Apply these settings to timer channel 2: Speaker.
5–4	11b	Modify counter bits 0–7 first, then 8–15.
3–1	011b	Choose output mode 3: Square wave generator.
0 (least significant)	0b	Use binary counting mode.

As far as hardware programming goes, this is relatively straightforward. Counter channel 2 is selected, which is the timer circuit that is connected to the speaker. This channel is set to a “little endian” style write mode, where multi-byte values are written with the low bits first, with the counter value interpreted as binary instead of decimal. Mode 3 generates square wave output, where the signal is “on” 50% of the time and “off” for the other 50%.

The next two calls to `outportb()` write the `sample` value into the timer’s counter register. I/O port 42h is the counter register for timer channel 2, and it accepts byte values only. Since the counter requires a 16-bit value, the register is written twice – the low 8 bits are written first, followed by the high 8 bits.

The final bit of hardware programming reads the byte at I/O port 61h via `inportb()`, turns on the bits at positions 0 and 1, and writes the modified byte back to the same port via `outportb()`. This turns on the “timer 2 gate to speaker enable” and “speaker data enable” bits on the keyboard controller’s control register, simultaneously starting timer channel 2’s output, and enabling speaker output. This causes the speaker to emit a tone, whose frequency is fixed to the value in the timer counter register.

Once all hardware programming is done, whether the speaker is playing or silenced, the `soundCursor` value is incremented. This prepares the PC speaker service for its next upcoming run. The function returns, leaving the speaker either silenced or playing a constant tone under asynchronous timer control.

`WaitHard()`

The `WaitHard()` function pauses execution for `delay` game ticks with no provision for the user to skip the wait.

There are 140 game ticks per second.

```
void WaitHard(word delay)
{
    gameTickCount = 0;
    while (gameTickCount < delay) ;
}
```

At first glance, this appears to be an infinite loop: `gameTickCount` is set to zero, and then we enter a `while` loop that repeats until the tick count reaches the (presumably nonzero) `delay`. If nothing increments `gameTickCount` or decrements `delay`, this loop never terminates.

But it does terminate, eventually. The reason is due to the timer hardware, which is constantly firing and generating interrupts. Each time a timer interrupt occurs, execution pauses and the timer interrupt service executes. In the game’s case, the interrupt service causes a call to `PCSpeakerService()`, which increments `gameTickCount` each time it

runs. Once the interrupt service returns, control is passed back to the code that was originally running. Even though this loop is infinite, it is paused at regular intervals while `gameTickCount` is incremented externally.

`gameTickCount` increments at a fixed rate of 140 Hz, so each delay can be converted to/from seconds using the ratio `delay/140`. Once this amount of time has elapsed, the function returns.

`WaitSoft()`

The `WaitSoft()` function pauses execution for `delay` game ticks, returning early if the user presses a key.

```
void WaitSoft(word delay)
{
    gameTickCount = 0;

    do {
        if (gameTickCount >= delay) break;
    } while ((inportb(0x0060) & 0x80) != 0);
}
```

The operation of this function is essentially the same as `WaitHard()`, except this function has a secondary exit condition if a key is pressed.



The `inportb()` call reads one byte from the I/O port at address 60h, which addresses the output buffer of the system's keyboard controller. The byte returned contains a snapshot of the most recent event that occurred at the keyboard.

Very briefly, the keyboard can report "key down" events (called **make** codes) and "key up" events (**break** codes). Key make events store a byte in the range 0h–7Fh, while key break events store a byte in the range 80h–FFh. The make code's value is the key's scancode, and the break code's value is the scancode plus 80h. Due to this arrangement, the most significant bit of the data byte can be used to differentiate between make and break codes, without regard to which key was actually pressed.

Based on the above, we can see that the `do...while` loop continues as long as the last thing that happened on the keyboard was a key release. Once

any key goes down (or if a key is already being held down), this condition no longer succeeds and the loop ends, causing the function to return.

If no key presses occur within `delay` game ticks, the loop also ends and the function returns as `WaitHard()` does.

1. <https://github.com/chocolate-doom/chocolate-doom/issues/795> 
2. 261.626 Hz for middle C is a consequence of the twelve-tone equal temperament³ tuning system, a.k.a. “musical notes that sound normal to Western listeners.” The reference frequency of this system is 440 Hz, which is assigned to the A above middle C (this note is sometimes written as **A440**). The rest of the note frequencies are determined by using a ratio of $\sqrt[12]{2}$ relative to the adjacent note. Middle C is 9 note-steps below A440, so the computation for its frequency is $440 \times (\sqrt[12]{2})^{-9} = 261.626$ Hz. 
3. https://en.wikipedia.org/wiki/12_equal_temperament 