0.1 Audio and Heartbeat

The audio and heartbeat system runs concurrently with the rest of the program. On an operating system supporting neither multi-processes nor threads this means using interrupts to stop normal execution and perform tasks on the side.

The idea is to configure the hardware to trigger a hardware interrupt at a regular interval. This interrupt is caught by a system called PIC which transforms it into a software interrupt, or IRQ. The software interrupt ID is used as an offset in a vector to look up a function belonging to the engine. At this point, the CPU is stopped (a.k.a: interrupted) from doing whatever it was doing (likely running the 2D renderer), and it starts running the interrupt handler which is called an ISR[1]. We now have two systems running in parallel.
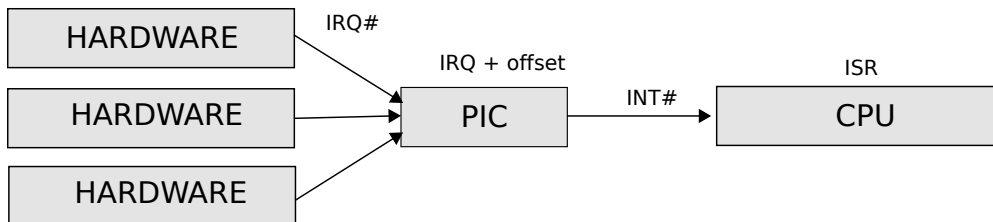

Figure 1: Hardware interrupts are translated to software interrupt via the PIC.

Since interrupts keep triggering constantly from various sources, an ISR must choose what should happen if an IRQ is raised while it is still running. There are two options. The ISR can decide it needs a "long" time to run and disable other IRQs via the IMR [2]. This path introduces the problem of discarding important information such as keyboard or mouse inputs.

Alternately, the ISR can decide not to mask other IRQs and do what it is supposed to do as fast as possible so as to not delay the firing of other important interrupts that may lose data if they aren't serviced quickly enough. Keen Dreams uses the latter approach and keeps tasks in its ISR very small and short.

0.1.1 IRQs and ISRs

The IRQ and ISR system relies on two chips: the Intel 8254 which is a PIT[3] and the Intel 8259 which is a PIC[4]. The PIT features a crystal oscillating in square waves. The PIT

---

[1]Interrupt Service Routine
[2]Interrupt Mask Register
[3]Programmable Interval Timer
[4]Programmable Interrupt Controller

contains three channels, each connected with a counter. On each period, it decrements its three counters. Counter #2 is connected to the buzzer and generates sounds. Counter #1 is connected to the RAM in order to automatically perform something called "memory refresh"[5]. Counter #0 is connected to the PIC. When counter #0 hits zero it generates an IRQ[6] and sends it to the PIC.
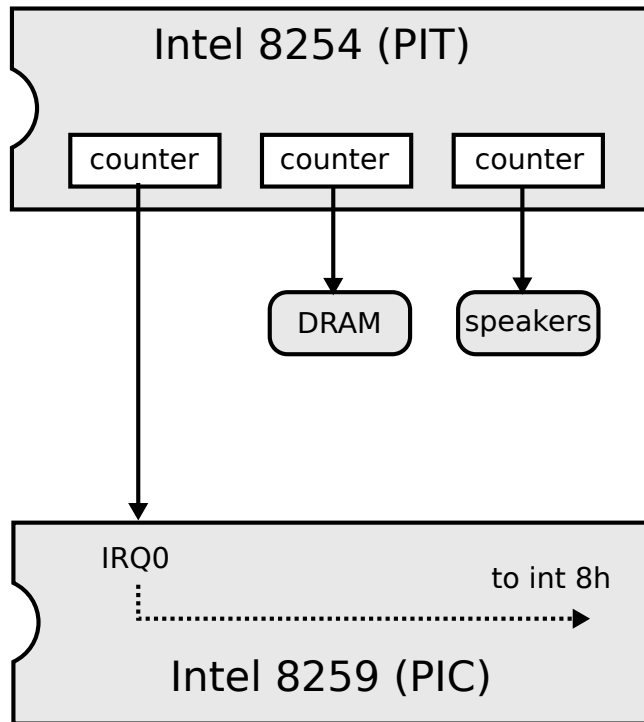

Figure 2: Interactions between PIT and PIC.

The PIC's hardware IRQ-0 to IRQ-8 are mapped to the Interrupt Vector starting at Offset 8 (resulting in mapping to software interrupts INT08 to INT0F).

---

[5]Without frequent refresh, DRAM will lose its content. This is one of the reasons it is slower and SRAM is preferred in the caching system.

[6]Interrupt Request Line: Hardware lines over which devices can send interrupt signals to the CPU.

I.V.T Entry #	Type
00h	CPU divide by zero
01h	Debug single step
02h	Non Maskable Interrupt
03h	Debug breakpoints
04h	Arithmetic overflow
05h	BIOS provided Print Screen routine
06h	Invalid opcode
07h	No math chip
08h	IRQ0, System timer
09h	IRQ1, Keyboard controller
0Ah	IRQ2, Bus cascade services for second 8259
0Bh	IRQ3, Serial port COM2
0Ch	IRQ4, Serial port COM1
0Dh	IRQ5, LPT2, Parallel port (HDD on XT)
0Eh	IRQ6, Floppy Disk Controller
0Fh	IRQ7, LPT1, Parallel port
10h	Video services (VGA)
11h	Equipment check
12h	Memory size determination

Figure 3: The Interrupt Vector Table (entries 0 to 18).

Notice #8 which is associated with the System timer and usually updates the operating system clock at 18.2 ticks per second. Because IVT #8 was hijacked, the operating system clock is not updated while Commander Keen runs. Upon exiting the game, DOS will run late by the amount of time played.

Using these two chips and placing its own function at Interrupt Vector Table (IVT) #8, the engine can stop its runtime at a regular interval, effectively implementing a subsystem running concurrently with everything else.

0.1.2 PIT and PIC

The PIT chip runs at 1.193182 MHz. This initially seems like an odd choice from the hardware designers, but has a logical origin. In 1980 when the first IBM PC 5150 was designed, the common oscillator used in television circuitry was running at 14.31818 MHz. As it was mass produced, the TV oscillator was very cheap so utilizing it in the PC drove down cost. Engineers built the PC timer around it, dividing the frequency by 3 for the CPU (which is why the Intel ran at 4.7MHz), and dividing by 4 to 3.57MHz for the CGA video card. By logically ANDing these signals together, a frequency equivalent to the base frequency divided

by 12 was created. This frequency is 1.1931816666 MHz. By 1990, oscillators were much cheaper and could have used any frequency but backward compatibility prevented this.

0.1.3 Interrupt Frequency

Each counter on the PIT chip is 16-bit, which is decremented after each period. An IRQ is generated and send to the PIC whenever the counter wraps around after 2^{16} = 65,536 decrements. So at default, the interrupts are generated at a frequency of 1.19318MHz / 65,536 = 18.2Hz. Some programs require a faster period than the 18.2 interrupts/second standard rate (for example, execution profilers). So they reprogram the timer by changing the counter value.

```
// Set the number of interrupts generated
// by system timer 0 per second
static void SDL_SetIntsPerSec(word ints)
{
  SDL_SetTimer0(1192755 / ints);
}

// Sets system timer 0 to the specified speed
static void SDL_SetTimer0(word speed)
{
  outportb(0x43,0x36);        // Change PIT counter 0
  outportb(0x40,speed);       // Speed is counter decrements
  outportb(0x40,speed >> 8);  // to send interrupt
}
```

Trivia : Note that SDL_SetTimer0 is using a frequency of 1.192755MHz, instead of the PIT documented 1.193182MHz. Most likely the value is based on 18.2 interrupts per second * 65,536 = 1192755Hz.

So the engine can decide at what frequency to be interrupted, depending on the type of sound/music it needs to play and what devices will be used. As a result, two frequencies are defined:

1. Running at 140Hz to play sound effects and music on the PC beeper, AdLib and SoundBlaster.

2. Running at 700Hz to play sound effects and music on Disney Sound Source.

```
#define TickBase   70

typedef enum {
  sdm_Off ,
  sdm_PC ,
  sdm_AdLib ,
  sdm_SoundBlaster
  sdm_SoundSource
} SDMode ;

static   word   t0CountTable[] = {2 ,2 ,2 ,2 ,10 ,10};

boolean SD_SetSoundMode ( SDMode mode )
{
  word   rate ;

  if ( result && ( mode != SoundMode ))
  {
    SDL_ShutDevice ();
    SoundMode = mode ;
    SDL_StartDevice ();
  }

  // Interrupt refresh to either 140 Hz or 700 Hz
  rate = TickBase * t0CountTable [ SoundMode ];
  SDL_SetIntsPerSec ( rate );
}
```

0.1.4 Heartbeats

Each time the interrupt system triggers, it runs another small (yet paramount) system before taking care of audio requests. The sole goal of this heartbeat system is to maintain a 32-bit variable: `TimeCount`.

```
longword TimeCount;

static void interrupt SDL_t0Service(void)
{
  static word count = 1,

  if (!(--count))
  {
    // Set count to match 70Hz update
    count = t0CountTable[SoundMode];
    TimeCount++;
  }

  outportb(0x20,0x20);  // Acknowledge the interrupt
}
```

It is updated at a rate of 70 units per seconds, to match the VGA update[7] rate of 70Hz. These units are called "ticks". Depending on how fast the audio system runs (from 140Hz to 700Hz), it adjusts how frequent it should increase `TimeCount` to keep the game rate at 70Hz.

Every system in the engine uses this variable to pace itself. The renderer will not start rendering a frame until at least one tick has passed. The AI system expresses action duration in tick units. The input sampler checks for how long a key was pressed, and the list goes on. Everything interacting with human players uses `TimeCount`.

0.1.5   Audio System

The audio system is complex because of the fragmentation of audio devices it can deal with. The early 90's was a time before Windows 95 harnessed all audio cards under the DirectSound common API. Each development studio had to write their own abstraction layer and id Software was no exception. At a high level, the Sound Manager offers a lean API divided in two categories: one for sounds and one for music.

```
void    SD_Startup(void);
void    SD_Shutdown(void);

[...]
```

---

[7]EGA was updated at a rate of 60Hz. Some games, like Keen Dreams, are developed with VGA already in mind.

```
[...]

void     SD_Default(boolean gotit,SDMode sd,SMMode sm);
void     SD_PlaySound(word sound);
void     SD_StopSound(void);
void     SD_WaitSoundDone(void);

void     SD_StartMusic(Ptr music);
void     SD_FadeOutMusic(void);
boolean  SD_MusicPlaying(void);
boolean  SD_SetSoundMode(SDMode mode);
boolean  SD_SetMusicMode(SMMode mode);
word     SD_SoundPlaying(void);
```

But in the implementation lies a maze of functions directly accessing the I/O port of four sound outputs: AdLib, SoundBlaster, Buzzer, and Disney Sound Source. All belong to one of the three supported families of sound generators: FM Synthesizer (Frequency Modulation), PCM (Pulse Code Modulation) or Square Waves (PC speaker).

Sounds effects are stored in three formats.

1. PC Speaker.

2. AdLib.

3. SoundBlaster/Disney Sound Source.

They are all packaged in the AudioT archive created by Muse. Sounds are segregated by format but always stored in the same order. This way a sound can be accessed in three formats by using STARTPCSOUNDS + sound_ID or STARTADLIBSOUNDS + sound_ID.

Despite being part of the source code, support for digital effects for the Sound Blaster & Sound Source devices was cut prior to release of Commander Keen Dreams. Therefore I won't explain digital effects (PCM) in this book[8].

---

[8]For further details on digital sound and PCM, there is an excellent read in the book "Game engine blackbook - Wolfenstein 3D" by Fabien Sanglard.

```
/////////////////////////////////////////////////
//
//  MUSE  Header  for  . KDR
//  Created  Mon  Jul  01  18:21:23  1991
//
/////////////////////////////////////////////////


# define  NUMSOUNDS            28
# define  NUMSNDCHUNKS         84


//
//  Sound  names  &  indexes
//
# define  KEENWALK1SND         0
# define  KEENWALK2SND         1
# define  JUMPSND              2
# define  LANDSND              3
# define  THROWSND             4
# define  DIVESND              5
# define  GETPOWERSND          6
# define  GETPOINTSSND         7
# define  GETBOMBSND           8
# define  FLOWERPOWERSND       9
# define  UNFLOWERPOWERSND     10
[...]
# define  OPENDOORSND          19
# define  THROWBOMBSND         20
# define  BOMBBOOMSND          21
# define  BOOBUSGONESND        22
# define  GETKEYSND            23
# define  GRAPESCREAMSND       24
# define  PLUMMETSND           25
# define  CLICKSND        26
# define  TICKSND         27


//
//  Base  offsets
//
# define  STARTPCSOUNDS    0
# define  STARTADLIBSOUNDS   28
# define  STARTDIGISOUNDS    56
# define  STARTMUSIC       84
```

FM Synthesizer: OPL2/YM3812 Programming

Programming the OPL2 output is esoteric to say the least. AdLib and Creative did publish SDKs but they were expensive. Documentation was sparse and often cryptic. Today, they are very difficult to find.

The OPL2 is made of 9 channels capable of emulating instruments. Each channel is made of two oscillators: a Modulator whose outputs are fed into a Carrier's input. Each channel has individual settings including frequency and envelope (composed of attack rate, decay rate, sustain level, release rate, and vibrato). Each oscillator can also pick a waveform (these characteristic forms are what gave the YM3812 its recognizable sound).

To control all of these channels, a developer must configure the OPL2's 244 internal registers. These are all accessed via two external I/O ports. One port is for selecting the card's internal register and the other is to read/write data to it.

```
0x388 - Address/Status port (R/W)
0x389 - Data port (W/O)
```

When the Adlib was first conceived in 1986, it was tested on IBM XTs and ATs, none of which exceeded a speed of 6 MHz. They wrote their specification based on this, writing that while the Adlib required a certain amount of "wait time" between commands, it was okay to send them as fast as possible because no PC was faster than the minimum wait time. They later found out that a Intel 386 was fast enough to send commands faster than the Adlib was expecting them, and they changed their specification to mention a minimum 35 microseconds wait time between commands.

The Programming Guide was amended with reliable specs to wait 3.3 microseconds after a register select write, and 23 microseconds after a data write. Within the source code it is implemented as a 10 microseconds and 25 microseconds respectively.

```c
//////////////////////////////////////////////////
//
//  alOut(n,b) - Puts b in AdLib card register n
//
//////////////////////////////////////////////////
void
alOut(byte n,byte b)
{
  asm pushf
  asm cli

  asm mov   dx,0x388
  asm mov   al,[n]
  asm out   dx,al
  SDL_Delay(TimerDelay10);    //wait 10ms

  asm mov   dx,0x389
  asm mov   al,[b]
  asm out   dx,al

  asm popf

  SDL_Delay(TimerDelay25);    //wait 25ms
}
```

Every time the audio system wakes up via the timer interrupt, it checks if a sound effects should be sent, and plays the next sample out through the AdLib card.

0.1.6 PC Speaker: Square Waves

The hardware chapter described a problem for sound effects: the default PC speaker could only generate square waves, resulting in long beeps which are not acceptable for gaming.

The solution was to approximate a tune by placing the PC Speaker in repeat mode and make it change frequency every 1/140th of a second. It is simpler to understand when the signal is a simple sinusoid:
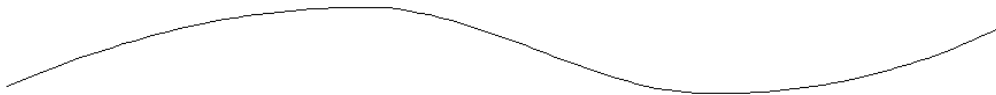
Figure 4: The original sound.

Figure 5: The same sound approximated with square wave and frequency changes.

To do this, the audio system once again relies on the PIT chipset. Channel 0 is used to trigger the audio system. Channel 1 is used to refresh the RAM periodically. Channel 2, however, is directly connected to the PC Speaker.

PICTURE

Figure XXX is a diagram of the hardware for driving the built-in speaker. OUT2 is the output of Channel 2 of the PIT, GATE2 (= bit 1 of port `0x61`) is the enable/trigger control for the Channel 2 counter, and SPEAKER (= bit 0 of port `0x61`) to control the speaker volume. The trick is to set OUT2 to square wave mode so it will repeat after it triggers and program the desired square wave frequency. This can be done by setting MODE (port `0x43`) to Mode 3.

Mode	Type
0	Interrupt on Terminal Count
1	Hardware Re-triggerable One-shot
2	Rate Generator
3	Square Wave Generator
4	Software Triggered Strobe
5	Hardware Triggered Strobe

Figure 6: Available modes of a PIT counter.

When instructed to play a PC Speaker sound effect, the audio system sets itself to run at 140Hz via PIT Counter 0. Every times it wakes up, it reads the frequency to maintain for the next 1/140th of a second and writes it to Counter 2. The frequencies to use are encoded as a stream of bytes, the value of which is decoded as follows:

```
frequency = 1193181 / (value * 60)
```

In the assembly, three I/O ports are accessed:

```
#define pcTimer    0x42
#define pcTAccess  0x43
#define pcSpeaker  0x61
```

While the end result was not great, it was better than a beep.

```
MACRO DOFX
  les di ,[pcSound]        ; PC sound effects
  mov ax , es
  or ax , di
  jz @@nopc                ; no PC sound effect going

  mov bl ,[es:di]          ; Get the byte
  inc [WORD PTR pcSound]   ; Increment pointer
  cmp [pcLastSample],bl    ; Is this sample same as last?
  jz @@pcsame              ; Yep - don't do anything
  mov [pcLastSample],bl    ; No, save it for next time

  or bl , bl
  jz @@pcoff               ; If 0, turn sounds off
  xor bh , bh
  shl bx ,1
  mov bx ,[pcSoundLookup + bx]  ; Use byte as lookup

  mov al ,0b6h             ; Select channel 2 (speaker) timer
  out pcTAccess , al
  mov al , bl
  out pcTimer , al         ; Low byte
  mov al , bh
  out pcTimer , al         ; High byte

  in al , pcSpeaker        ; Turn the speaker & gate on
  or al  ,3
  out pcSpeaker , al
```

Notice how the ∗ 60 is not calculated but looked up. Once again the engine tries to save as much CPU time as possible by using a bit of RAM. The frequency is read from a lookup table pcSoundLookup.

```
word        pcSoundLookup [255];
```

Notice how b6 (10110110) is sent to the PIC Command register:

- 10 = Target Counter 2.

- 11 = High & low byte of counter updated.

- 011 = Square Wave Generator.

- 0 = 16-bit mode.