0.1 Keyboard

Before Microsoft introduced the DirectInput API in Windows 95, developers needed to write custom drivers for each input device. This required direct communication with hardware using the vendor's protocol over physical ports.

0.1.1 Keyboard scancodes

Each key on a PC keyboard is linked to a scancode. When a key is pressed, circuitry in the keyboard generates the scancode and sends it serially to the keyboard controller. This key-down event, also known as a "make code", signals the key press. When a key is released, its scancode high-bit (`80h`) is set, producing a "break code" that is sent by the same mechanism. For example, pressing "**A**" generates the make code `1Eh`, while releasing it sends the break code `9Eh`. Holding down a key will type a repeating sequence of that character after a short delay, resulting in retransmitting the make code at regular intervals for as long as the key is being held.

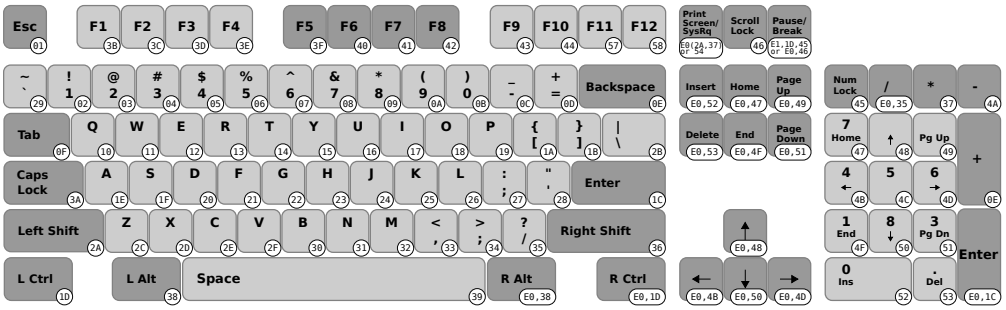

Figure 1: IBM Keyboard Model M scancode layout (scancodes in hexadecimal).

The Model M includes 14 keys, such as "**Right Ctrl**", "**Right Alt**", and the arrow keys, that duplicate functions from the 84-key AT layout. To save encoding space, these keys are sent as two-byte scancodes: the first byte is always `E0h`, followed by the make or break code of the equivalent key from the original AT layout. For instance, the standard "/"-key produces the scancode `35h`, while the numeric keypad's "**Num** /"-key sends `E0h` followed by `35h`. This allows software to differentiate between the keys without expanding the scancode table.

This encoding also has the additional benefit that, if the underlying software only understands the 84-key layout, it will (presumably) discard the `E0h` byte in this example and only process the `35h` byte. This would result in a "/" being correctly typed from the numeric keypad, even though the software doesn't understand that this key exists.

Trivia : The `E0h` scancode introduced some unexpected behaviors. For example, typing "**Shift**" + "**Num** /" should produce a "/", as indicated on the keycap. However, older software may interpret this sequence incorrectly: Shift is down, I don't understand `E0h`, and here comes `35h`. That means the user wanted to type "**?**".

0.1.2 Keyboard input

Once a key is pressed or released, the scancode is decoded and processed by the keyboard controller. On the IBM XT, this is an Intel 8255 PPI[1] chip. Its main purpose is to listen for serial data from the keyboard, verify that it arrived intact, store each incoming scancode in a buffer, and request an interrupt so it can be read by the software.

With the advent of the IBM AT and the need for bidirectional communication with the keyboard for features like keyboard status LEDs, the Intel 8042 UPI[2] chip was introduced. The UPI retained backward compatibility with the PPI chip, handling all scancodes from the keyboard. When a key is pressed, the interrupt is routed to ISR #9 in the Vector Interrupt Table. The engine installs its own ISR there.

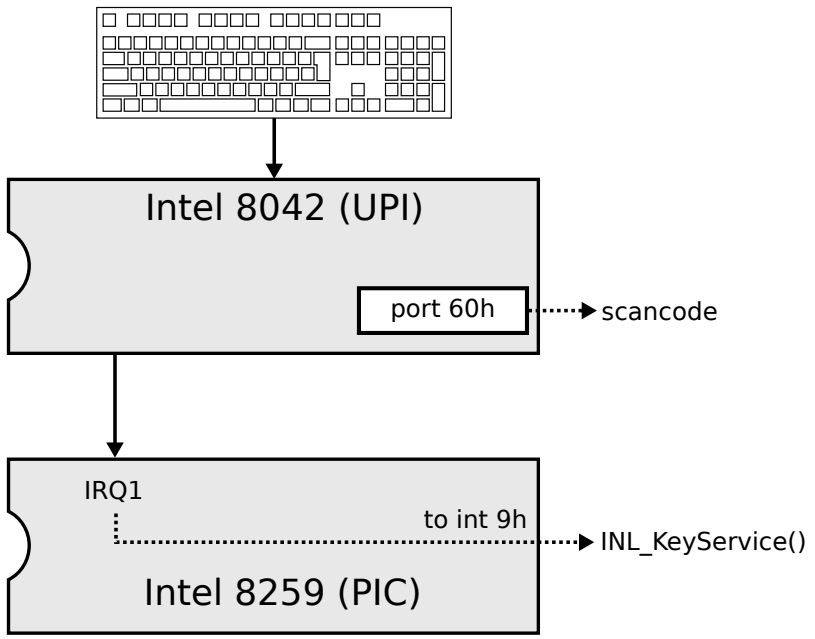

Figure 2: Keyboard hardware diagram.

[1] Programmable Peripheral Interface
[2] Universal Peripheral Interface.

```
#define KeyInt    9 // The keyboard ISR number

static void INL_StartKbd(void) {

  IN_ClearKeysDown();

  OldKeyVect = getvect(KeyInt);
  setvect(KeyInt,INL_KeyService);
}

static void interrupt INL_KeyService(void) {
  [...]
}
```

The state of the keyboard is maintained in a global array `Keyboard`, available for the entire engine to lookup.

```
#define NumCodes  128
boolean Keyboard[NumCodes];
```

When a scancode byte arrives from the keyboard, it is copied to a buffer in the keyboard controller and an IRQ #1 is raised. Scancodes arrive one byte at a time, and one interrupt at a time, even if it is a multi-byte code. The interrupt handler can read a byte from I/O port `60h` to capture the scancode for further processing. Once that is complete, the interrupt handler must explicitly acknowledge the IRQ at the interrupt controller to re-arm it for the next keyboard event. This is accomplished by writing an "end of interrupt" signal byte to I/O port `20h`.

There is a slight difference between the PPI and the UPI in terms of how the keyboard input buffer is managed. The PPI will hold a byte in its input buffer indefinitely until the software acknowledges that it has completed the read. The acknowledgment procedure is to briefly strobe the high bit of I/O port `61h` on, then off. When this occurs, the keyboard controller resets its buffer status and resumes reading from the keyboard. The UPI simplifies this process. Reading from I/O port `60h` automatically resets the buffer, eliminating the need for a separate acknowledgment. However, to maintain backward compatibility, most programs still toggle the high bit of I/O port `61h`, even though this has no effect on 286-based systems.

```
static void interrupt INL_KeyService ( void ) {
  byte k;
  k = inportb (0 x60 ); // Get the scan code

  // Tell the XT keyboard controller to clear the key
  outportb(0x61,(temp = inportb(0x61)) | 0x80);
  outportb(0x61,temp);

  if (k == 0xe0)     // Special key prefix
    special = true;
  else if (k == 0xe1) // Handle Pause key
    Paused = true;
  else
  {
    if (k & 0x80) // Break code
    {
      k &= 0x7f;
      Keyboard[k] = false;
    }
    else           // Make code
    {
      LastCode = CurCode;
      CurCode = LastScan = k;
      Keyboard[k] = true;
      [...]        //Process the key
    }
  }
  outportb (0 x20 ,0 x20 ); // ACK interrupt to interrupt
   system
}
```