This is part 10 of an article series on emulation development in JavaScript; ten parts are currently available, and others are expected to follow.

- *Part 1: The CPU*
- *Part 2: Memory*
- *Part 3: GPU Timings*
- *Part 4: Graphics*
- *Part 5: Integration*
- *Part 6: Input*
- *Part 7: Sprites*
- *Part 8: Interrupts*
- *Part 9: Memory Banking*
- *Part 10: Timers*

Since the first computers were put together, one of their basic functions has been to keep time: to coordinate actions according to timers. Even the simplest of games has an element of time to it: Pong, for example, needs to move the ball across the screen at a particular rate. In order to handle these timing issues, every games console has some form of timer to allow for things to happen at a given moment, or at a specific rate.

The GameBoy is no exception to this rule, and contains a set of registers which automatically increment based on a programmable schedule. In this part of the series, I'll be investigating the structure and operation of the timer, and how it can be used to seed pseudo-random number generators, such as the one contained in Tetris and its various clones. One example of a Tetris clone which uses the timer, to pick random pieces for the game, is demonstrated below.

Reset | Run

Figure 1: jsGB implementation with timer

Timer structure

The GameBoy's CPU, as described in the first part of this series, runs on a 4,194,304Hz clock, with two internal measures of the time taken to execute each instruction: the T-clock, which increments with each clock step, and the M-clock, which increments at a quarter of the speed (1,048,576Hz). These clocks are used as the source of the timer, which counts up, in turn, at a

quarter of the rate of the M-clock: 262,144Hz. In this article, I'll refer to this final value as the timer's "base speed".

The GameBoy's timer hardware offers two separate timer registers: the system works by incrementing the value in each of these registers at a pre-determined rate. The "divider" timer is permanently set to increment at 16384Hz, one sixteenth of the base speed; since it's only an eight-bit register, its value will go back to zero after it reaches 255. The "counter" timer is more programmable: it can be set to one of four speeds (the base divided by 1, 4, 16 or 64), and it can be set to go back to a value that isn't zero when it overflows past 255. In addition, the timer hardware will send an interrupt to the CPU, as described in part 8, whenever the "counter" timer does overflow.

There are four registers used by the timer; these are made available for use by the system as part of the I/O page, just like the graphics and interrupt registers:

<table>
<tr><th>Address</th><th>Register</th><th>Details</th></tr>
<tr><td>0xFF04</td><td>Divider</td><td>Counts up at a fixed 16384Hz;
reset to 0 whenever written to</td></tr>
<tr><td>0xFF05</td><td>Counter</td><td>Counts up at the specified rate

Triggers INT 0x50 when going 255->0</td></tr>
<tr><td>0xFF06</td><td>Modulo</td><td>When Counter overflows to 0,
it's reset to start at Modulo</td></tr>
<tr><td>0xFF07</td><td>Control</td><td>
<table>
<tr><th>Bits</th><th>Function</th><th>Details</th></tr>
<tr><td>0-1</td><td>Speed</td><td>00: 4096Hz
01: 262144Hz
10: 65536Hz
11: 16384Hz</td></tr>
<tr><td>2</td><td>Running</td><td>1 to run timer, 0 to stop</td></tr>
<tr><td>3-7</td><td colspan="2">Unused</td></tr>
</table>
</td></tr>
</table>

Table 1: Timer registers

Since the "counter" timer triggers an interrupt when it overflows, it can be especially useful if a game requires something to happen at a regular interval. However, a Gameboy game can generally use the vertical blank to much the same effect, since it occurs at a regular pace of almost 60Hz; the vertical blanking handler can be used not only to refresh the screen contents, but to check the keypad and update the game state. Therefore, there's little call for use of the timer in traditional Gameboy games, though it can be used to greater effect in graphic demos.

Implementing the timer emulation

The emulation developed in this article series uses the CPU's clock as the basic unit of time. For that reason, it's simplest to maintain a clock for the timer that runs in step with the CPU clock, and is updated by the dispatch function. It's convenient at this stage to keep the DIV register as a separate entity to the controllable timer, incremented at 1/16th the rate again of the fastest timer step:

Timer.js: Clock increment

```
TIMER = {
    _clock: {
        main: 0,
        sub:  0,
```

```
        div:  0
    },

    _reg: {
        div:  0,
        tima: 0,
        tma:  0,
        tac:  0
    },

    inc: function()
    {
        // Increment by the last opcode's time
        TIMER._clock.sub += Z80._r.m;

        // No opcode takes longer than 4 M-times,
        // so we need only check for overflow once
        if(TIMER._clock.sub >= 4)
        {
            TIMER._clock.main++;
            TIMER._clock.sub -= 4;

            // The DIV register increments at 1/16th
            // the rate, so keep a count of this
            TIMER._clock.div++;
            if(TIMER._clock.div == 16)
            {
                TIMER._reg.div = (TIMER._reg.div+1) & 255;
                TIMER._clock.div = 0;
            }
        }

        // Check whether a step needs to be made in the timer
        TIMER.check();
    }
};
```

Z80.js: Dispatcher

```
while(true)
{
    // Run execute for this instruction
    var op = MMU.rc(Z80._r.pc++);
    Z80._map[op]();
    Z80._r.pc &= 65535;
    Z80._clock.m += Z80._r.m;
    Z80._clock.t += Z80._r.t;

    // Update the timer
    TIMER.inc();

    Z80._r.m = 0;
    Z80._r.t = 0;

    // If IME is on, and some interrupts are enabled in IE, and
    // an interrupt flag is set, handle the interrupt
    if(Z80._r.ime && MMU._ie && MMU._if)
    {
```

```
        // Mask off ints that aren't enabled
        var ifired = MMU._ie & MMU._if;

        if(ifired & 0x01)
        {
            MMU._if &= (255 - 0x01);
            Z80._ops.RST40();
        }
    }

    Z80._clock.m += Z80._r.m;
    Z80._clock.t += Z80._r.t;

    // Update timer again, in case a RST occurred
    TIMER.inc();
}
```

From here, the controllable timer is made up of varying divisions of the base speed, making it relatively simple to check whether the timer values need to be stepped up, and to provide the registers as part of the memory I/O page. The interface between the following section of code and the MMU I/O page handler, is left as an exercise for the reader.

Timer.js: Register check and update

```
    check: function()
    {
        if(TIMER._reg.tac & 4)
        {
            switch(TIMER._reg.tac & 3)
            {
                case 0: threshold = 64; break;          // 4K
                case 1: threshold =  1; break;          // 256K
                case 2: threshold =  4; break;          // 64K
                case 3: threshold = 16; break;          // 16K
            }

            if(TIMER._clock.main >= threshold) TIMER.step();
        }
    },

    step: function()
    {
        // Step the timer up by one
        TIMER._clock.main = 0;
        TIMER._reg.tima++;

        if(TIMER._reg.tima > 255)
        {
            // At overflow, refill with the Modulo
            TIMER._reg.tima = TIMER._reg.tma;

            // Flag a timer interrupt to the dispatcher
            MMU._if |= 4;
        }
    },

    rb: function(addr)
```

```
    {
        switch(addr)
        {
            case 0xFF04: return TIMER._reg.div;
            case 0xFF05: return TIMER._reg.tima;
            case 0xFF06: return TIMER._reg.tma;
            case 0xFF07: return TIMER._reg.tac;
        }
    },

    wb: function(addr, val)
    {
        switch(addr)
        {
            case 0xFF04: TIMER._reg.div = 0; break;
            case 0xFF05: TIMER._reg.tima = val; break;
            case 0xFF06: TIMER._reg.tma = val; break;
            case 0xFF07: TIMER._reg.tac = val & 7; break;
        }
    }
```

Seeding a pseudo-random number generator

A major component of many games is unpredictability: Tetris, for instance, will throw an unknown pattern of pieces down the well, and the game consists of building rows using these pieces. Ideally, a computer provides unpredictability by generating random numbers, but this runs contrary to the methodical nature of a computer; it's not possible for a computer to provide a truly random pattern of numbers. Various algorithms exist to produce sequences of numbers that look superficially like they're random, and these are called *pseudo*-random number generation (PRNG) algorithms.

A PRNG is generally implemented as a formula that, given a particular input number, will produce another number with almost no relation to the input. For Tetris, nothing so complicated is required; instead, the following code is used to produce a seemingly random block.

Tetris.asm: Select new block

```
BLK_NEXT = 0xC203
BLK_CURR = 0xC213
REG_DIV  = 0x04

NBLOCK: ld hl, BLK_CURR         ; Bring the next block
        ld a, (BLK_NEXT)        ; forward to current
        ld (hl),a
        and 0xFC                ; Clear out any rotations
        ld c,a                  ; and hold onto previous

        ld h,3                  ; Try the following 3 times

.seed:  ldh a, (REG_DIV)        ; Get a "random" seed
        ld b,a
.loop:  xor a                   ; Step down in sevens
.seven: dec b                   ; until zero is reached
        jr z, .next             ; This loop is equivalent
        inc a                   ; to (a%7)*4
        inc a
        inc a
        .
```

```
        inc a
        cp 28
        jr z, .loop
        jr .seven

.next:  ld e,a                  ; Copy the new value
        dec h                   ; If this is the
        jr z, .end              ; last try, just use this
        or c                    ; Otherwise check
        and 0xFC                ; against the previous block
        cp c                    ; If it's the same again,
        jr z, .seed             ; try another random number
.end:   ld a,e                  ; Get the copy back
        ld (BLK_NEXT), a        ; This is our next block
```

The basis of the Tetris block selector is the DIV register: since the selection routine is only run once every few seconds, the register will have an unknown value on any given run, and it thus makes a fair approximation of a random number source. With the timer system having been emulated, Tetris and its clones can be emulated to full functionality, as shown in Figure 1.

Coming up: Sound

One aspect of game emulation which has been overlooked until now is the generation of sound, and the synchronisation of sound to the speed of the emulation. Over and above the aspect of sound generation by the emulator, is the method by which sound is output to the browser; the next part of this series will investigate the issues surrounding sound output mechanisms, and whether a coherent strategy can be put together for sound production in JavaScript.

Imran Nazar <tf@imrannazar.com>, Feb 2011.

Article dated: 25th Feb 2011