This is part 9 of an article series on emulation development in JavaScript; ten parts are currently available, and others are expected to follow.

Thus far in this series, we've been dealing with the loading and emulation of a simple memory map for the GameBoy, with the entirety of the game ROM fitting into the lower half of memory. There aren't many games that fit into memory in full (Tetris is one of the few); most games are larger than this, and have to employ an independent mechanism to swap "banks" of game ROM into the GameBoy CPU's view.

Some of the first games in the GameBoy library were built with a Memory Bank Controller inside the cartridge, which did this job of swapping banks of ROM into view; over the years, various versions of the cartridge MBC were built for increasingly large games. In the particular example of the demo associated with this part, the first version of the MBC is used to handle the loading of a 64kB ROM.

Reset | Run

Figure 1: jsGB implementation with MBC1 support

Banking and memory expansion

Through the years, many computer systems have had to deal with the problem of having too much program to fit into memory. Traditionally, there have been two ways to deal with this problem.

- **Increase the address space**: Build a new CPU with more address lines, allowing it to see and understand a larger amount of memory. This is the preferred solution, but requires significant time to redevelop the computer system in question, and may need more changes to be made in the supporting chipset for the CPU.
- **Virtual memory**: This can either refer to the holding of chunks of RAM on disk, and their swapping in when required; or the swapping in of chunks of pre-written ROM when required. In both cases, the system hardware needs little extension but any software for the system has to be aware of the paging/banking system, in order to use it.

Since the GameBoy is a fixed hardware platform with wide distribution, there's no way to increase the address space when larger games are produced; instead, the Memory Bank Controller built into the cartridge offers a way to switch 16kB banks of ROM into view. In addition to this, the MBC1 supports up to 32kB of "external RAM", which is writable memory in the cartridge; this can be banked into the `[A000-BFFF]` space in the memory map, if it's available.

In order to facilitate software that uses the MBC1, the first 16kB bank of ROM (bank 0) is fixed at address `0000`; the second half of the ROM space can be made into a window on any ROM bank between 1 and 127, for a maximum ROM size of 2048kB. One of the oddities of the MBC1 is that it deals internally in 32's: banks #32, #64 and #96 are inaccessible, since they're treated within the banking system as bank #0. This means that 125 banks apart from the fixed bank #0 are usable.

There are four registers within the MBC1 chip, that allow for switching of banks for the ROM and RAM; these can be changed by writing to the (normally read-only) ROM space anywhere within a certain range. The details are given in the below table.

Locations	Register	Details
0000-1FFF	Enable external RAM	4 bits wide; value of 0x0A enables RAM, any other value disables
2000-3FFF	ROM bank (low 5 bits)	Switch between banks 1-31 (value 0 is seen as 1)
4000-5FFF	ROM bank (high 2 bits) RAM bank	ROM mode: switch ROM bank "set" {1-31}-{97-127} RAM mode: switch RAM bank 0-3
6000-7FFF	Mode	0: ROM mode (no RAM banks, up to 2MB ROM) 1: RAM mode (4 RAM banks, up to 512kB ROM)

Table 1: MBC1 register set

MBCs and the cartridge header

Since there are multiple kinds of controller for banking, any given game must state which MBC is used, in the cartridge header data. This is the first chunk of data in the cartridge ROM, and follows a specific format.

Location(s)	Value	Size (bytes)	Details
0100-0103h	Entry point	4	Where the game starts Usually "NOP; JP 0150h"
0104-0133h	Nintendo logo	48	Used by the BIOS to verify checksum
0134-0143h	Title	16	Uppercase, padded with 0

0144-0145h	Publisher	2	Used by newer GameBoy games
0146h	Super GameBoy flag	1	Value of 3 indicates SGB support
0147h	Cartridge type	1	MBC type/extras
0148h	ROM size	1	Usually between 0 and 7 Size = 32kB << [0148h]
0149h	RAM size	1	Size of external RAM
014Ah	Destination	1	0 for Japan market, 1 otherwise
014Bh	Publisher	1	Used by older GameBoy games
014Ch	ROM version	1	Version of the game, usually 0
014Dh	Header checksum	1	Checked by BIOS before loading
014E-014Fh	Global checksum	2	Simple summation, not checked
0150h	Start of game		

Table 2: Cartridge header format

In this particular case, we're interested in the value of `0147h`, the cartridge type. The cartridge type can be one of the following values, if an MBC1 is fitted to the cartridge:

Value	**Definition**
00h	No MBC
01h	MBC1
02h	MBC1 with external RAM
03h	MBC1 with battery-backed external RAM

Table 3: Cartridge type values pertaining to MBC1

For the purposes of this article, a system of battery backing will not be implemented for the external RAM; this feature is often used by games to save their state for later use, and will be looked at in more detail in a later part.

Implementation of MBC1

The memory bank controllers are an obvious manipulation of memory, and thus fit neatly into the MMU. Since the first ROM bank (bank #0) is fixed, an offset need only be maintained for the MBC to indicate where it's reading for the second bank. In order to allow for more MBC handling to be added later, an array of data can be used to hold the state of a given controller:

MMU.js: MBC state and reset

```
MMU = {
    // MBC states
    _mbc: [],

    // Offset for second ROM bank
    _romoffs: 0x4000,

    // Offset for RAM bank
    _ramoffs: 0x0000,

    // Copy of the ROM's cartridge-type value
    _carttype: 0,

    reset: function()
    {
        ...

        // In addition to previous reset code,
        // initialise MBC internal data
        MMU._mbc[0] = {};
        MMU._mbc[1] = {
            rombank: 0,         // Selected ROM bank
            rambank: 0,         // Selected RAM bank
            ramon: 0,           // RAM enable switch
            mode: 0             // ROM/RAM expansion mode
        };

        MMU._romoffs = 0x4000;
        MMU._ramoffs = 0x0000;
    },

    load: function(file)
    {
        ...
        MMU._carttype = MMU._rom.charCodeAt(0x0147);
    }
}
```

As can be seen in the above code, the internal state of the MBC1's four registers is represented by an object within the MMU, associated with MBC type 1. When these are changed, the ROM and RAM offsets can be modified to point into the appropriate bank of memory; once the pointers are set, access to the memory can proceed almost as normal.

MMU.js: MBC1-based access

```
MMU = {
    rb: function(addr)
    {
        switch(addr & 0xF000)
        {
            ...

            // ROM (switched bank)
            case 0x4000:
            case 0x5000:
            case 0x6000:
            case 0x7000:
                return MMU._rom.charCodeAt(MMU._romoffs +
                                           (addr & 0x3FFF));

            // External RAM
            case 0xA000:
            case 0xB000:
                return MMU._eram[MMU._ramoffs +
                                 (addr & 0x1FFF)];
        }
    }
};
```

The calculation of these pointer offsets is performed when the MBC registers are written, as shown below.

MMU.js: MBC1 control

```
    wb: function(addr, val)
    {
        switch(addr & 0xF000)
        {
            // MBC1: External RAM switch
            case 0x0000:
            case 0x1000:
                switch(MMU._carttype)
                {
                    case 2:
                    case 3:
                        MMU._mbc[1].ramon =
                            ((val & 0x0F) == 0x0A) ? 1 : 0;
                        break;
                }
                break;

            // MBC1: ROM bank
            case 0x2000:
            case 0x3000:
                switch(MMU._carttype)
                {
                    case 1:
                    case 2:
                    case 3:
                        // Set lower 5 bits of ROM bank (skipping #0)
                        val &= 0x1F;
                        if(!val) val = 1;
                        MMU._mbc[1].rombank =
                            (MMU._mbc[1].rombank & 0x60) + val;

                        // Calculate ROM offset from bank
                        MMU._romoffs = MMU._mbc[1].rombank * 0x4000;
                        break;
                }
                break;

            // MBC1: RAM bank
            case 0x4000:
            case 0x5000:
                switch(MMU._carttype)
                {
                    case 1:
                    case 2:
                    case 3:
                        if(MMU._mbc[1].mode)
                        {
                            // RAM mode: Set bank
                            MMU._mbc[1].rambank = val & 3;
                            MMU._ramoffs = MMU._mbc[1].rambank * 0x2000;
                        }
                        else
                        {
                            // ROM mode: Set high bits of bank
                            MMU._mbc[1].rombank =
                                (MMU._mbc[1].rombank & 0x1F) +
                                ((val & 3) << 5);

                            MMU._romoffs = MMU._mbc[1].rombank * 0x4000;
                        }
                        break;
                }
                break;

            // MBC1: Mode switch
            case 0x6000:
            case 0x7000:
                switch(MMU._carttype)
                {
                    case 2:
                    case 3:
                        MMU._mbc[1].mode = val & 1;
                        break;
                }
                break;

            ...

            // External RAM
            case 0xA000:
            case 0xB000:
                MMU._eram[MMU._ramoffs + (addr & 0x1FFF)] = val;
                break;
        }
    }
```

In the above control code, instances of MBC1 that are stated as having external RAM attached are the ones which have RAM banking. With this code in place, the demo shown in Figure 1 loads and runs properly; without the MBC1 handler, the code would crash while attempting to access sprite and background data for the display.

Coming up

Aside from being able to fit larger games into memory, one of the more important aspects of a game is the ability to keep time: a clock-based game, for example, is useless without some kind of timing mechanism on which to base its clock. As mentioned previously, many games use the vertical blanking interrupt for this timing, but some require a finer-grained time structure; this is provided in the GameBoy by a hardware timer, tied into the CPU clock.

The timer also provides a method of examining the CPU clock, which makes it useful as a seed for random number generators; Tetris, for example, picks its blocks using this functionality of the hardware timer. In the next part, I'll look at the details of how the timer works, and how it can be implemented.

Imran Nazar <tf@imrannazar.com>, Dec 2010.

Article dated: 3rd Dec 2010