

从零开始实现GameBoy模拟器 #11 MBC2、MBC3卡带



銀葉吉祥
浙江大学 软件工程硕士

已关注

14 人赞同了该文章

目录

收起

- MBC2卡带
 - 启用/禁用RAM
 - 选择ROM1区域的映射分块
 - 实现MBC2
- MBC3卡带
 - 启用/禁用RAM和RTC寄存器读
 - 设置ROM1映射分块序号
 - 设置RAM映射分块序号，或者
 - 锁住/解锁RTC时间寄存器
 - 实现实时时钟
 - 实现MBC3卡带读写



欢迎来到从零开始实现GameBoy模拟器第十一章。在本章中，我们接着上文的内容，继续扩展卡带读写逻辑，以支持MBC2和MBC3卡带。那么，让我们开始吧！

本章节代码已经上传至项目仓库，名称为LunaGB-11，已经下载项目代码的读者可以更新代码库来查阅最新代码。同时，欢迎大家加入GameBoy模拟器开发交流群：33854734
2024.3.3更新：修复了卡带非法地址读写时错误日志的格式问题。

MBC2卡带

MBC2是MBC系列卡带的第二种规格，支持最多256KB的ROM和512x4位的RAM。MBC2对于内存区域的映射与MBC1类似，如下所示：

- 0x0000~0x3FFF：ROM0区域（16KB）
- 0x4000~0x7FFF：ROM1区域（16KB）
- 0xA000~0xA1FF：RAM区域（512x4位）
- 0xA200~0xBFFF：ECHO RAM区域

其中，ROM0区域固定映射到ROM bank 0，即第一个ROM分块，ROM1区域可以由程序控制映射到ROM bank 1至ROM bank 15。MBC2的RAM区域比较特殊，MBC2控制芯片并不支持外接RAM芯片，而是在MBC控制器中内置了512个半字节的RAM存储区域。所谓的“半字节”指的是RAM区域中的每一个地址只能够存储4个bit的数据，而不是完整的8个bit，因此在读取这部分数据时，只有低四位的数据为有效数据，高四位的数据是无效的，应当由程序丢弃。同样，在向MBC2卡带的RAM地址区域写入数据时，只有低四位的数据会被实际写入RAM中，高四位的数据则会被直接丢弃。

由于MBC2卡带的RAM尺寸固定为512个半字节，MBC2内部只使用了9个地址选择引脚（即9个bit）来指定RAM中的地址偏移，因此只有0xA000~0xA1FF区域为有效区域。当程序读写0xA200~0xBFFF区域时，由于高位地址选择引脚的数据被丢弃，其表现类似于将0xA000~0xA1FF地址区间在0xA000~0xBFFF范围内重复映射16次，因此读写0xA200、0xA400、0xA600等地址的值完全等同于读写0xA000地址的值，以此类推。部分文档会将0xA200~0xBFFF区域称为ECHO RAM区域，这部分区域实际上仅仅是0xA000~0xA1FF区域的重复映射，在实现MBC2卡带的RAM读写时，我们可以将用户传入的地址先减去0xA000，然后与512进行mod操作来模拟这种重复映射的特性。

需要注意的是，由于MBC2卡带的RAM并不是单独的RAM芯片，因此在所有MBC2卡带中，卡带元信息中的ram_size字段都为0。我们需要单独判断加载的卡带是否为MBC2卡带，并在加载MBC2卡带时分配固定尺寸的CRAM内存。同时，MBC2卡带也分为带电池（06）和不带电池

(05) 的两个芯片版本，对于带电池的MBC2卡带，我们也需要使用与其余卡带相同的逻辑保存和加载RAM中的数据。

MBC2支持的操作一共有两个：启用/禁用RAM，以及选择RAM1区域的映射分块。两个操作都通过向0x0000~0x3FFF地址区间中的任意地址写入特定值来执行，其通过判断写入地址的第8位（从0开始算，也就是高8位中的最低位）的值来确定具体要执行的操作。

启用/禁用RAM

当写入地址的第8位为0时，该操作用于控制是否启动RAM。此时，如果程序写入的值等于0x0A，则RAM会被启用，否则RAM会被禁用。

选择ROM1区域的映射分块

当写入地址的第8位为1时，该操作用于选择RAM1区域的映射分块。此时，程序向该地址写入的值会作为选择分块的序号使用。由于MBC2卡带只支持最多16个分块的ROM，因此只有写入值的低4位会用于实际控制ROM1区域加载的分块。和MBC1一样，当写入值的低四位为0时，MBC2芯片会等同于程序写入1，并将ROM bank 1映射到ROM1区域，因此程序无法将ROM bank 0映射到ROM1区域。

实现MBC2

在了解了MBC2的工作原理和控制方法后，是时候编写代码实现MBC2卡带了！首先我们在Cartridge.hpp中添加用于检测当前卡带是否为MBC2卡带的辅助函数：

```
inline bool is_cart_mbc2(u8 cartridge_type)
{
    return cartridge_type >= 5 && cartridge_type <= 6;
}
```

由于MBC2卡带所支持的两个操作与MBC1卡带完全重合，我们可以直接复用MBC1的变量来表示MBC2控制器的当前状态，只需要修改一下注释即可：

```
struct Emulator
{
    /*...*/
    //! MBC1/MBC2: The cartridge RAM is enabled for reading / writing.
    bool cram_enable = false;
    //! MBC1/MBC2: The ROM bank number controlling which rom bank is mapped to
    u8 rom_bank_number = 1;
    /*...*/
};
```

接着修改Emulator::init函数，由于MBC2卡带中的ram_size字段均为0，因此当检测到卡带类型为MBC2后，我们直接指定分配512字节的CRAM内存，并且只用每个字节的低4位来存储数据：

```
RV Emulator::init(Path cartridge_path, const void* cartridge_data, size_t cartridge_size)
{
    /*...*/
    switch(header->ram_size)
    {
        case 2: cram_size = 8_kb; break;
        case 3: cram_size = 32_kb; break;
        case 4: cram_size = 128_kb; break;
        case 5: cram_size = 64_kb; break;
        default: break;
    }
    if(is_cart_mbc2(header->cartridge_type))
    {
        // MBC2 cartridges have fixed 512x4 bits of RAM, which is not shown in
        cram_size = 512;
    }
    if(cram_size)
    {
        cram = (byte_t*)memalloc(cram_size);
        memzero(cram, cram_size);
        if(is_cart_battery(header->cartridge_type))
        {
            load_cartridge_ram_data();
        }
    }
    return ok;
}
```

最后便是修改Cartridge.cpp，加上卡带的读写功能了。我们定义两个新函数mbc2_read和mbc2_write来表示MBC2卡带的读写功能。mbc2_read的实现如下：

```
u8 mbc2_read(Emulator* emu, u16 addr)
{
    if(addr <= 0x3FFF)
    {
        return emu->rom_data[addr];
    }
    if(addr >= 0x4000 && addr <= 0x7FFF)
    {
        // Cartridge ROM bank 01-0F.
        usize bank_index = emu->rom_bank_number;
        usize bank_offset = bank_index * 16_kb;
        return emu->rom_data[bank_offset + (addr - 0x4000)];
    }
    if(addr >= 0xA000 && addr <= 0xBFFF)
    {
        if(!emu->cram_enable) return 0xFF;
        u16 data_offset = addr - 0xA000;
        data_offset %= 512;
        return (emu->cram[data_offset] & 0x0F) | 0xF0;
    }
    log_error("LunaGB", "Unsupported MBC2 cartridge read address: 0x%04X", (u32)addr);
    return 0xFF;
}
```

MBC2中读取ROM部分的代码与MBC1基本相同，此处不再赘述。在读取RAM的时候，我们会首先计算出需要读取的地址在RAM地址区间中的偏移，然后将偏移值mod 512，以模拟MBC2中重复映射同一个RAM16次的特性。在读取数据时，我们会将实际存储的数据与0x0F做与操作，从而清空高4位的数据，然后再与0xF0做或操作，从而将高四位的数据全部设置为1。实际上，对于高四位的返回数据，GameBoy的文档并无规定，我们在这里采用“空引脚默认高电平”的惯例将高四位的值设置为1。

mbc2_write的实现如下：

```
void mbc2_write(Emulator* emu, u16 addr, u8 data)
{
    if(addr <= 0x3FFF)
    {
        if(addr & 0x100) // bit 8 is set.
        {
            // Set ROM bank number.
            emu->rom_bank_number = data & 0x0F;
            if(emu->rom_bank_number == 0)
            {
                emu->rom_bank_number = 1;
            }
            if(emu->num_rom_banks <= 2)
            {
                emu->rom_bank_number = emu->rom_bank_number & 0x01;
            }
            else if(emu->num_rom_banks <= 4)
            {
                emu->rom_bank_number = emu->rom_bank_number & 0x03;
            }
            else if(emu->num_rom_banks <= 8)
            {
                emu->rom_bank_number = emu->rom_bank_number & 0x07;
            }
            return;
        }
    }
    else
    {
        // Enable/disable cartridge RAM.
        if(emu->cram)
        {
            if(data == 0x0A)
            {
                emu->cram_enable = true;
            }
            else
            {
                emu->cram_enable = false;
            }
        }
    }
}
```

```
        return;
    }
}
else if(addr >= 0xA000 && addr <= 0xBFFF)
{
    if(!emu->cram_enable) return;
    u16 data_offset = addr - 0xA000;
    data_offset %= 512;
    emu->cram[data_offset] = data & 0x0F;
    return;
}
log_error("LunaGB", "Unsupported MBC2 cartridge write address: 0x%04X", (u16)addr);
}
```

在mbc2_write中，当写入地址区间在0x0000~0x3FFF时，我们首先将地址与0x0100做与操作，从而取出地址的第8位，并根据其是否为0来决定需要执行的操作。当写入地址的区间在0xA000~0xBFFF时，我们使用与mbc2_read相同的方式处理地址，并在写入数据前将数据与0x0F进行与操作，从而丢弃数据的高4位。

最后，我们修改cartridge_read和cartridge_write函数，加入mbc2_read和mbc2_write：

```
u8 cartridge_read(Emulator* emu, u16 addr)
{
    u8 cartridge_type = get_cartridge_header(emu->rom_data)->cartridge_type;
    if(is_cart_mbc1(cartridge_type))
    {
        return mbc1_read(emu, addr);
    }
    else if(is_cart_mbc2(cartridge_type))
    {
        return mbc2_read(emu, addr);
    }
    else
        /*...*/
}
void cartridge_write(Emulator* emu, u16 addr, u8 data)
{
    u8 cartridge_type = get_cartridge_header(emu->rom_data)->cartridge_type;
    if(is_cart_mbc1(cartridge_type))
    {
        mbc1_write(emu, addr, data);
        return;
    }
    else if(is_cart_mbc2(cartridge_type))
    {
        mbc2_write(emu, addr, data);
        return;
    }
    else
        /*...*/
}
```

此时编译并运行模拟器，加载一个MBC2格式的游戏ROM，例如《超级机器人大战》，可以看到游戏已经能够正常运行了：



MBC3卡带

MBC3是MBC系列卡带的第三种规格，支持最多2MB的ROM和最多32KB的RAM。MBC3卡带可以看成是MBC1卡带的升级版，其大部分操作与MBC1卡带类似，除了以下不同：

1. MBC3卡带具有完整的7+2条分块选择引脚，因此可以同时支持2MB的ROM和32KB的RAM，不再出现MBC1卡带中被迫在大尺寸ROM和大尺寸RAM之间“二选一”的问题。
2. MBC3卡带的ROM1现在支持映射ROM bank 0x20，0x40和0x60，同时ROM0固定映射至ROM bank 0x00，不再跟随分块切换而改变。
3. MBC3卡带加入了实时时钟（TIMER）组件。实时时钟组件由卡带内的纽扣电池供电，可以在GameBoy关闭，甚至卡带被拔出的情况下持续记录时间流逝，从而实现一些与当前时间有关的游戏功能。一个典型的例子就是宝可梦在第二世代中引入的时间系统，从而实现了“部分宝可梦只在特定时间出现”的游戏玩法。

MBC3对于内存区域的映射与MBC1基本相同，如下所示：

1. 0x0000~0x3FFF：ROM0区域（16KB）
2. 0x4000~0x7FFF：ROM1区域（16KB）
3. 0xA000~0xBFFF：RAM区域/RTC寄存器

其中，ROM0固定映射至ROM bank 0，ROM1可以根据程序指令映射至ROM bank 0x01至0x7F的任意分块。RAM区域则可以根据程序指令映射至RAM bank 0x00至0x03的任意分块，或者映射至RTC（real time clock）寄存器，供程序读写实时时钟的时间和设置。

MBC3支持的操作包括以下几种：

地址区间	操作	默认值
0x0000~0x1FFF	启用/禁用RAM和RTC寄存器读写	禁用
0x2000~0x3FFF	设置ROM1映射分块序号	1
0x4000~0x5FFF	设置RAM映射分块序号，或者RTC寄存器	0
0x6000~0x7FFF	锁住/解锁RTC时间寄存器	未锁住

启用/禁用RAM和RTC寄存器读写

与MBC1类似，该操作用于启用/禁用程序对RAM的访问。由于RTC寄存器也使用RAM地址区间读写，因此当RAM区域映射为RTC寄存器时，该操作也同时用于启用/禁用程序对RTC寄存器的访问。对该区域中任意地址写入0x0A可以启用RAM/RTC寄存器读写，写入其余值则禁用RAM/RTC寄存器读写。

需要注意的是，该操作仅用于控制程序对RAM地址区间的访问权限，在禁用RAM/RTC寄存器读写的时候并不会停止时钟运行。时钟的运行和停止需要使用下文会介绍的RTC寄存器来实现。

设置ROM1映射分块序号

该操作与MBC1类似，使用写入值作为序号来更改ROM1区域的映射分块。与MBC1不同的是，现在0x20、0x40、0x60分块也可以被映射到ROM1区域，而ROM0区域则固定映射为0x00分块，不受该值影响。如果写入值为0x00，则其会被视为0x01，并映射ROM bank 1至ROM1区域。

设置RAM映射分块序号，或者RTC寄存器

该操作用于切换RAM区域的映射，当写入值的范围为0x00~0x03时，该操作会将RAM bank 0~3映射至RAM区域；当写入值范围为0x08~0x0C时，该操作会将RTC的各个寄存器映射至RAM区域。当MBC3将RTC寄存器映射到RAM区域时，对于RAM区域任何地址的读写操作均被视为对相应寄存器的读写操作。在以上两个写入值范围外的任何值均为非法写入值，游戏程序应当避免写入这些非法值。

当写入值的范围为0x08~0x0C时，该操作会按照下表所示映射RTC的对应寄存器：

写入值	寄存器名称	描述
0x08	RTC S	存储当前时间的秒，范围为0~59
0x09	RTC M	存储当前时间的分，范围为0~59
0x0A	RTC H	存储当前时间的时，范围为0~23
0x0B	RTC LD	存储当前时间的天的低8位
0x0C	RTC HD	存储当前时间的天的高1位，以及时钟控制位

0x0C RTC HD的各个位的含义如下表所示：

7	6	5	4	3	2	1	0
溢出位	暂停时钟	-	-	-	-	-	天数

- 天数：存储当前时间天数的最高位，因此实时时钟的天数寄存器一共有9位，最多可以计时511天。
- 暂停时钟：向该位写入1会暂停时钟的计时，写入0则恢复时钟计时。在向RTC的任何寄存器写入值前，程序需要首先通过向该位写入0来暂停时钟运行。
- 溢出位：当实时时钟记录的天数超过511天23小时59分59秒后，实时时钟会将该位设置为1，同时将当前时间重置为全部为0。游戏在启用后可以通过读取该位来判断实时时钟是否发生了溢出，并需要在处理溢出后手动将该位清零。

由于判断溢出需要启动游戏后才能进行，因此实时时钟要求玩家至少需要每1022天内启动一次游戏，才能正确进行计时，否则其中的至少511天会被丢弃。当游戏程序读取当前的时间后，其可以将游戏在关闭期间已经运行的天数写入RAM中，然后将RTC的天数寄存器清零。通过这种方式，游戏可以连续记录至少10000年的时间流逝。

锁住/解锁RTC时间寄存器

在默认情况下，RTC寄存器中的时间会随着时钟运行而同步更新。向该地址区域先写入0x00，再写入0x01会锁住所有的时间寄存器的值，以便程序在读取时间的时候这些寄存器的值不会变化，同时也不需要暂停时钟。在时间寄存器锁住的状态下，再次写入0x00 0x01可以解锁这些寄存器，让他们重新与当前时钟的时间同步。

实现实时时钟

在了解了MBC3的工作原理和控制方法后，是时候编写代码实现MBC3卡带了！首先我们在Cartridge.hpp中添加用于检测当前卡带是否为MBC3卡带的辅助函数：

```
inline bool is_cart_mbc3(u8 cartridge_type)
{
    return cartridge_type >= 15 && cartridge_type <= 19;
}
```

由于并不是所有的MBC3卡带都搭载实时时钟，我们还需要一个函数来判断当前卡带是否搭载时钟：

```
inline bool is_cart_timer(u8 cartridge_type)
```

```
{
    return cartridge_type == 15 || cartridge_type == 16;
}
```

然后我们需要在模拟器中加入时钟组件。我们新建两个文件RTC.hpp和RTC.cpp来保存时钟组件的代码。RTC.hpp代码如下：

```
#pragma once
#include <Luna/Runtime/MemoryUtils.hpp>
using namespace Luna;

struct RTC
{
    u8 s; // Seconds.
    u8 m; // Minutes.
    u8 h; // Hours.
    u8 dl;// Lower 8 bits of Day Counter.
    u8 dh;// Upper 1 bit of Day Counter, Carry Bit, Halt Flag.

    // Internal state.
    f64 time;
    bool time_latched;
    // Set to `true` when writing 0x00 to 0x6000~0x7FFF.
    bool time_latching;

    void init();
    void update(f64 delta_time);
    void update_time_registers();
    void update_timestamp();
    void latch();
    u16 days() const { return (u16)dl + (((u16)(dh & 0x01)) << 8); }
    bool halted() const { return bit_test(&dh, 6); }
    bool day_overflow() const { return bit_test(&dh, 7); }
};
```

和之前我们实现的PPU一样，我们首先在RTC的头部定义了RTC的五个寄存器，这样我们就可以通过字节偏移直接读写每一个寄存器的值。同时，由于时间寄存器的值可能会被锁住，所以我们使用time变量来记录实际的时间，并使用update_time_registers和update_timestamp在寄存器的时间值和实际时间之间进行同步。RTC的状态通过调用update函数来实现更新，每一帧更新一次。latch方法用于锁住/解锁寄存器时间。最后，我们定义了三个辅助函数：days、halted和day_overflow，用于方便地从寄存器中读取各个状态值。

RTC.cpp的实现如下：

```
#include "RTC.hpp"

void RTC::init()
{
    memzero(this);
}
void RTC::update(f64 delta_time)
{
    if(!halted())
    {
        time += delta_time;
        if(!time_latched)
        {
            update_time_registers();
        }
    }
}
void RTC::update_time_registers()
{
    s = ((u64)time) % 60;
    m = (((u64)time) / 60) % 60;
    h = (((u64)time) / 3600) % 24;
    u16 days = (u16)(((u64)time) / 86400);
    dl = (u8)(days & 0xFF);
    if(days & 0x100) bit_set(&dh, 0);
    else bit_reset(&dh, 0);
    if(days >= 512) bit_set(&dh, 7);
    else bit_reset(&dh, 7);
}
void RTC::update_timestamp()
{
    time = s + ((u64)m) * 60 + ((u64)h) * 3600 + ((u64)days()) * 86400;
    if(day_overflow())
```

```
        {
            time += 86400 * 512;
        }
    }
    void RTC::latch()
    {
        if(!time_latched)
        {
            time_latched = true;
        }
        else
        {
            time_latched = false;
            update_time_registers();
        }
    }
}
```

上述代码的要点有以下几个：

1. 在调用RTC::update时，我们需要传入当前帧的帧时间（以秒为单位）。当RTC没有被停止时，RTC会将该时间累加到time变量中，并在时间寄存器没有锁住的情况下调用update_time_registers来更新寄存器的值。
2. update_time_registers和update_timestamp分别用于根据当前时间更新时间寄存器以及根据当前寄存器的值更新时间。前者在时间被更新，且时间寄存器没有被锁住的时候调用，后者则需要在任意时间寄存器被程序修改后调用，以同步时间寄存器中的时间。两个函数的实现只是简单地在秒和日期之间转换，并根据是否溢出来设置对应的溢出标志位。
3. latch函数同时用于锁住和解锁时间寄存器，其行为取决于当前时间寄存器是否被锁住。当我们解锁时间寄存器时，我们需要立即调用一次update_time_registers，从而让最新的时间能够同步到寄存器中。

接着我们将RTC加入到Emulator类中：

```
/*...*/
#include "RTC.hpp"
using namespace Luna;

/*...*/

struct Emulator
{
    /*...*/
    Joypad joypad;
    RTC rtc;

    /*...*/
};
```

然后修改Emulator的init和update函数，添加对RTC的支持：

```
RV Emulator::init(Path cartridge_path, const void* cartridge_data, usize cartridge_size)
{
    /*...*/
    joypad.init();
    rtc.init();
    /*...*/
}
void Emulator::update(f64 delta_time)
{
    joypad.update(this);
    if(is_cart_timer(get_cartridge_header(rom_data)->cartridge_type))
    {
        rtc.update(delta_time);
    }
    /*...*/
}
```

在update中，我们需要判断当前的卡带类型是否含有实时时钟，并仅在卡带支持时钟的情况下才更新RTC组件。

在实现了上述功能以后，我们的RTC已经能够在游戏运行时实现精确计时功能了。然而，实时时钟的一个很大的优点在于其可以在游戏机（即我们的模拟器）关闭的情况下持续计时，而在我们当前的实现下，一旦模拟器关闭，时钟的数据就会丢失，并在下一次模拟器启动的时候重置！这显然是我们不希望的结果，因此我们需要一个方式令我们在下次打开同一个卡带的时候，模拟器能够表现出时钟在持续工作的特性。

根据上一章的实现RAM时的知识可知，为了在下一次卡带启动时保留上一次的运行数据，我们可以在关闭卡带时将需要保存的数据写入一个.sav存档文件中，并在下一次打开游戏的时候加载这个存档文件，以恢复卡带RAM中的数据。我们可以使用同样的方法对RTC进行处理，将RTC中的所有状态写入存档文件中，这样就可以在下次打开游戏的时候恢复数据了。然而，该方法也只是让我们能够恢复上一次关闭游戏时的RTC时钟时间，对于两次打开游戏之间的时间流逝，该方法无法追踪，我们还需要想其它办法来解决该问题。

实际上，所有的主流操作系统都提供了一个全局的time函数，通过该函数可以获取一个使用整数表示的系统时间。在大部分的系统上，该时间被定义为从1970年1月1日00:00到当前时间点所经过的秒数（可以为负，从而表示1970年之前的时间），以这种形式定义的时间被称为UNIX时间戳。在有了UNIX时间戳的帮助后，我们可以在存档的时候同时将当前时间以UNIX时间戳的形式保存在存档文件中，然后在下次加载存档文件时读取该时间戳，并与当前的时间相减，就可以求得两次运行游戏之间流逝的时间差了！

按照这个思路，我们需要修改Emulator::save_cartridge_ram_data和Emulator::load_cartridge_ram_data函数，以添加对于RTC的状态保存和加载操作。首先我们为Emulator.cpp文件添加一个头文件：

```
#include <Luna/Runtime/Time.hpp>
```

该头文件声明了Luna SDK提供的系统时间相关API，其中就包括了get_utc_timestamp函数，我们需要调用这个函数返回UNIX时间戳形式表示的当前系统时间。需要注意的是，LunaSDK同时提供了get_local_timestamp函数和get_utc_timestamp函数，两者的区别在于前者会收到系统的时区设置的影响，因此在同一时刻，不同时区设置下，返回的时间戳会有所不同，而get_utc_timestamp会返回UTC时间，因此避免了不同时区设置对返回值的影响。

我们首先修改Emulator::save_cartridge_ram_data函数，在识别到具有RTC的卡带类型后，将RTC的值以及当前时间保存在存档文件中：

```
void Emulator::save_cartridge_ram_data()
{
    lutry
    {
        /*...*/
        luexp(f->write(cram, cram_size));
        if(is_cart_timer(get_cartridge_header(rom_data)->cartridge_type))
        {
            // Save RTC state.
            luexp(f->write(&rtc, sizeof(RTC)));
            // Save current timestamp.
            i64 timestamp = get_utc_timestamp();
            luexp(f->write(&timestamp, sizeof(i64)));
        }
        log_info("LunaGB", "Save cartridge RAM data to %s.", path.encode().c_str());
    }
    lucatch
    {
        log_error("LunaGB", "Failed to save cartridge RAM data. Game progress is %d%%.", progress);
    }
}
```

可以看到，我们直接将RTC结构体和当前时间戳的数据以二进制的形式写入了文件中。细心的读者也许会察觉到，由于大端CPU和小端CPU对多字节变量的内存排布顺序不同，如果读写文件时的平台字节序不同，则读取到的数据会有错误。这里我们出于实现方便的考虑并没有考虑平台字节序的影响。

接着修改Emulator::load_cartridge_ram_data函数，在游戏启动的时候恢复RTC的数据，并在RTC没有停止的情况下将游戏关闭期间流逝的时间添加到time变量中，并更新寄存器：

```
void Emulator::load_cartridge_ram_data()
{
    lutry
    {
        /*...*/
        luexp(f->read(cram, cram_size));
        if(is_cart_timer(get_cartridge_header(rom_data)->cartridge_type))
        {
            // Restore RTC.
            luexp(f->read(&rtc, sizeof(RTC)));
            // Read timestamp.
            i64 save_timestamp;
            luexp(f->read(&save_timestamp, sizeof(i64)));
            if(!rtc.halted())
            {
                time += save_timestamp;
```

```
        // Apply delta time between last save time and current time.
        i64 current_timestamp = get_utc_timestamp();
        i64 delta_time = current_timestamp - save_timestamp;
        if(delta_time < 0) delta_time = 0;
        rtc.time += delta_time;
        rtc.update_time_registers();
    }
}
log_info("LunaGB", "cartridge RAM data loaded: %s", path.encode().c_str());
}
lucatch {}
return;
}
```

这样一来，我们就实现了RTC在游戏停止状态下依然能够跟踪时间流逝的功能。此处我们对delta_time是否小于0做了一个特殊判断，这种情况通常不会发生，但是如果第二次运行时系统的时间设置有误（例如，你可以手动将当前系统时间调到上世纪90年代），则第二次运行的时间可能会早于第一次运行的时间。在这种情况下，我们简单将流逝时间设置为0，从而避免系统时间设置错误导致的负面影响。

实现MBC3卡带读写

由于MBC3卡带所使用的寄存器与MBC1卡带完全一致，我们可以复用MBC1的卡带控制变量，只需要修改对应的注释即可：

```
struct Emulator
{
    /*...*/

    ///! The number of ROM banks. 16KB per bank.
    usize num_rom_banks = 0;
    ///! MBC1/MBC2: The cartridge RAM is enabled for reading / writing.
    ///! MBC3: The cartridge RAM and cartridge timer enabled.
    bool cram_enable = false;
    ///! MBC1/MBC2/MBC3: The ROM bank number controlling which rom bank is mapped.
    u8 rom_bank_number = 1;
    ///! MBC1: The RAM bank number register controlling which ram bank is mapped.
    ///! If the cartridge ROM size is larger than 512KB (32 banks), this is used for
    ///! high 2 bits of rom bank number, enabling the game to use at most 2MB of ROM.
    ///! MBC3: The RAM bank number register controlling which ram bank/RTC registers are mapped.
    ///! 0-3: RAM banks.
    ///! 8-12: RTC registers.
    u8 ram_bank_number = 0;

    /*...*/
};
```

其中，当卡带是MBC3时，我们使用ram_bank_number的8~12范围来表示对RTC寄存器的映射。接着在Cartridge.cpp中添加mbc3_read和mbc3_write函数，mbc3_read函数的实现如下：

```
u8 mbc3_read(Emulator* emu, u16 addr)
{
    if(addr <= 0x3FFF)
    {
        return emu->rom_data[addr];
    }
    if(addr >= 0x4000 && addr <= 0x7FFF)
    {
        // Cartridge ROM bank 01-7F.
        usize bank_index = emu->rom_bank_number;
        usize bank_offset = bank_index * 16_kb;
        return emu->rom_data[bank_offset + (addr - 0x4000)];
    }
    if(addr >= 0xA000 && addr <= 0xBFFF)
    {
        if(emu->ram_bank_number <= 0x03)
        {
            if(emu->cram)
            {
                if(!emu->cram_enable) return 0xFF;
                usize bank_offset = emu->ram_bank_number * 8_kb;
                luassert(bank_offset + (addr - 0xA000) <= emu->cram_size);
                return emu->cram[bank_offset + (addr - 0xA000)];
            }
        }
    }
}
```

```
        if(is_cart_timer(get_cartridge_header(emu->rom_data)->cartridge_type) {
            emu->ram_bank_number >= 0x08 && emu->ram_bank_number <= 0x0C)
        {
            return ((u8*)&emu->rtc.s)[emu->ram_bank_number - 0x08];
        }
    }
    log_error("LunaGB", "Unsupported MBC3 cartridge read address: 0x%04X", (u32)addr);
    return 0xFF;
}
```

mbc3_read的实现与mbc1_read类似，但是在访问RAM地址区间（0xA000~0xBFFF）时，在当前卡带包含实时时钟时，我们需要检查当前的RAM映射值是否在0x08~0x0C之间，并根据结果返回不同的RTC寄存器的值。mbc3_write函数的实现如下：

```
void mbc3_write(Emulator* emu, u16 addr, u8 data)
{
    if(addr <= 0x1FFF)
    {
        // Enable/disable cartridge RAM.
        if(data == 0x0A)
        {
            emu->cram_enable = true;
        }
        else
        {
            emu->cram_enable = false;
        }
        return;
    }
    if(addr >= 0x2000 && addr <= 0x3FFF)
    {
        // Set ROM bank number.
        emu->rom_bank_number = data & 0x7F;
        if(emu->rom_bank_number == 0)
        {
            emu->rom_bank_number = 1;
        }
        return;
    }
    if(addr >= 0x4000 && addr <= 0x5FFF)
    {
        // Set RAM bank number, or map RTC registers.
        emu->ram_bank_number = data;
        return;
    }
    if(addr >= 0x6000 && addr <= 0x7FFF)
    {
        if(is_cart_timer(get_cartridge_header(emu->rom_data)->cartridge_type))
        {
            if(data == 0x01 && emu->rtc.time_latching)
            {
                emu->rtc.latch();
            }
            if(data == 0x00)
            {
                emu->rtc.time_latching = true;
            }
            else
            {
                emu->rtc.time_latching = false;
            }
            return;
        }
    }
    if(addr >= 0xA000 && addr <= 0xBFFF)
    {
        if(emu->ram_bank_number <= 0x03)
        {
            if(emu->cram)
            {
                if(!emu->cram_enable) return;
                usize bank_offset = emu->ram_bank_number * 8_kb;
                luassert(bank_offset + (addr - 0xA000) <= emu->cram_size);
                emu->cram[bank_offset + (addr - 0xA000)] = data;
                return;
            }
        }
    }
}
```

```
        if(is_cart_timer(get_cartridge_header(emu->rom_data)->cartridge_type) {
            emu->ram_bank_number >= 0x08 && emu->ram_bank_number <= 0x0C)
        {
            ((u8*)(&emu->rtc.s))[emu->ram_bank_number - 0x08] = data;
            emu->rtc.update_timestamp();
            return;
        }
    }
    log_error("LunaGB", "Unsupported MBC3 cartridge write address: 0x%04X", (u16)addr);
}
```

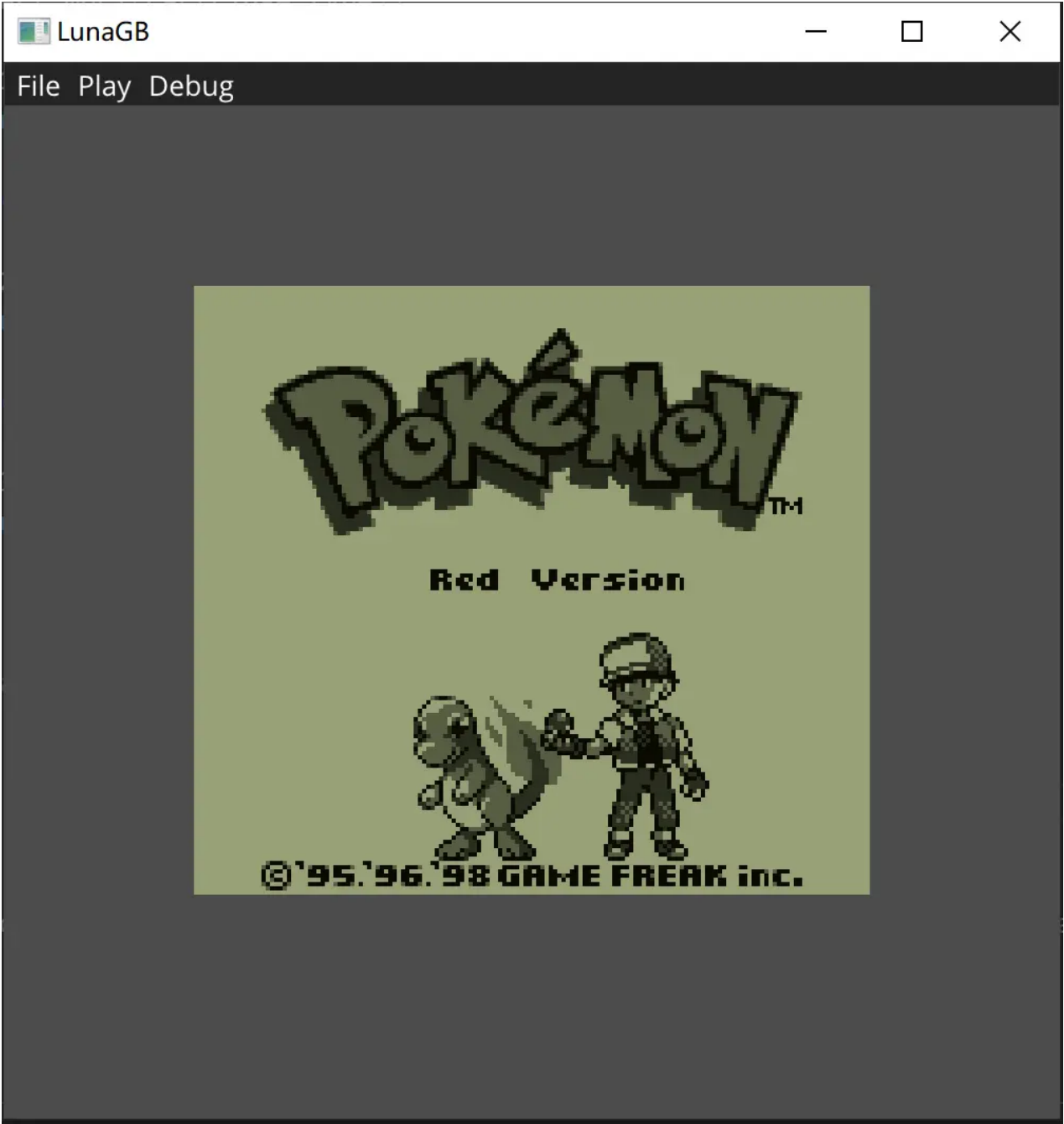
mbc3_write的实现与mbc1_write类似，区别在于mbc3不再需要考虑banking_mode的设置，可以直接映射任意RAM分块并实现读写。同时，针对有RTC的卡带，当我们发现程序向RTC寄存器写入数据后，我们需要调用update_timestamp，从而将RTC的当前时间与寄存器中的时间同步。MBC3卡带的0x6000~0x7FFF区域用于控制RTC时间寄存器的加锁和解锁，由于我们需要向内存中写入连续的两个数（00 01）才能实现加锁和解锁，因此我们在代码中使用RTC::time_latching来储存我们是否已经写入了一个00，并在time_latching为true，同时写入值又为01的情况下调用RTC::latch来执行加锁/解锁操作。

最后，我们修改cartridge_read和cartridge_write函数，加入MBC3卡带的支持：

```
u8 cartridge_read(Emulator* emu, u16 addr)
{
    /***/
    else if(is_cart_mbc2(cartridge_type))
    {
        return mbc2_read(emu, addr);
    }
    else if(is_cart_mbc3(cartridge_type))
    {
        return mbc3_read(emu, addr);
    }
    else
    /***/
}

void cartridge_write(Emulator* emu, u16 addr, u8 data)
{
    /***/
    else if(is_cart_mbc2(cartridge_type))
    {
        mbc2_write(emu, addr, data);
        return;
    }
    else if(is_cart_mbc3(cartridge_type))
    {
        mbc3_write(emu, addr, data);
        return;
    }
    else
    /***/
}
```

编译并运行模拟器，加载任意MBC3类型的卡带，例如《宝可梦：红》，可以看到我们的模拟器已经能够正常运行MBC3卡带了：



恭喜！我们的模拟器已经能够正确运行大部分单色GameBoy所支持的游戏卡带了。至此，我们的GameBoy模拟器专题只剩下最后一个组件——APU没有实现了。在下一章中，我们将开始APU部分的介绍，并开始搭建一个最基本的实时音频信号处理框架。那么，下一章见！

銀葉吉祥：从零开始实现GameBoy模拟器 #12 APU
10 赞同 · 2 评论 [文章](#)

编辑于 2024-03-19 22:30 · IP 属地上海

- 任天堂 Switch
- Game Boy (GB)
- 游戏机模拟器


362 赞同

欢迎参与讨论




还没有评论，发表第一个评论吧

文章被以下专栏收录

-  吉祥的游戏制作笔记
游戏制作中的技术、艺术和设计灵感记录

推荐阅读



关于苹果手机怎么玩模拟器游戏：打开这些站点一键在线玩

老男孩游戏盒

教你用macbook玩模拟器游戏


前言相信大家从小就接触过老任的主机游戏，以及一些其他的街机游戏，都非常的经典。我本人小时候家里爸妈不给买游戏机，所以都是跑到亲戚家里去蹭堂哥的玩，暑假和堂哥，两个人抱着半个西瓜...

Wende... 发表于有关生活的...

国内安卓模拟器测评，哪款安卓模拟器64位安卓版本最完

国内安卓模拟器测评，哪款安卓模拟器64位安卓版本最完善？64位系统的运算能力远超32位系统，随着安卓64位操作系统的普及，越来越多的重度手游比如《天谕》《天涯明月刀》，只能在安卓64位的...

奇怪所所长



RA替代计划（下）—PSV全能模拟器太难用？推荐几个替代

leon

▲ 赞同 14

▼

● 添加评论

📌 分享

♥ 喜欢

★ 收藏

📄 申请转载

...

↑