シェルスクリプトで
ゲームボーイプログラミング
入門

大神祐真　著

エアコミケ（2020 年春）新刊
2020 年 5 月 5 日　 ver 1.0

■免責

本書は情報の提供のみを目的としています。
本書の内容を実行・適用・運用したことで何が起きようとも、それは実行・適用・運用した人自身の責任であり、著者や関係者はいかなる責任も負いません。

■商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。
また本書では、™、®、© などのマークは省略しています。

はじめに

本書をお手にとっていただきありがとうございます！

本書では、アセンブラやコンパイラなどを使用せず、echo コマンドのバイナリ出力や dd コマンドなどを駆使して、シェルスクリプトでゲームボーイの ROM ファイルを生成します。

そのためには、ゲームボーイの CPU の各機械語命令について、そのバイナリ列を echo 等で生成する処理をシェル関数で一つ一つ用意して、というような、正にアセンブラを作る作業をすることになります。

ただ、本書では、筆者が個人の開発で使っている自作のライブラリ群を使います。このライブラリ群自身も単なるシェルスクリプトで、source コマンドで読み込むことで、ゲームボーイの CPU の機械語命令やカートリッジヘッダ等のバイナリ列を生成するいくつものシェル関数が使えるようになります。

本書では、このように、予め用意したシェル関数を活用してシェルからバイナリを直接吐くことで ROM ファイルを生成します。

具体的なやり方はこのあと本文で紹介しますが、「シェルスクリプトのみでやる」というのは手軽さもある反面、とても強い縛りでもあるので、少々トリッキーなやり方が登場するのはご了承ください。。

本書の構成

本書は以下の 3 章構成です。

- 第 1 章 開発環境構築と最初のプログラム
  - 開発に使用するスクリプトやエミュレータを紹介しながら、最初のプログラムとして「無限ループするだけ」の ROM ファイルを生成します
- 第 2 章 背景にタイルを配置
  - 画面描画の入門として、ゲームボーイの画面描画方式を紹介し、背景にタイルを配置してみます
- 第 3 章 キー入力を試す
  - 入力取得の方法と、割り込みを使用する方法を紹介し、キー入力取得を試します

PDF/HTML 版や本書の更新情報について

本書の PDF/HTML 版は筆者のウェブページで公開しています。

- `http://yuma.ohgami.jp`

本書の内容について訂正や更新があった場合もこちらのページに記載しますので、何かおかしな点などあった場合はまずはこちらのページをご覧ください。

サンプルについて

本文でも紹介しますが、本書のサンプルは以下のリポジトリで公開しています。

- `https://github.com/cupnes/gb_programming_with_shell-script_samples`

本文中、コードリストを掲載する際は、このリポジトリ内のファイルパスをキャプションに記載していますので、適宜参考にしてみてください。

目次

第1章
開発環境構築と最初のプログラム

この章では、使用するスクリプト群とエミュレータを紹介しながら、本書でのゲームボーイ (以下、GB) プログラミングの流れを紹介します。

1.1 事前準備

Bash が動作する環境を用意しておいてください。Linux や macOS などの場合は問題ないかと思いますが、Windows の場合は WSL などで Bash が動作する環境を用意しておいてください。

1.2 本書のサンプルプログラムをダウンロード

シェルスクリプトでの GB プログラミングを補助するスクリプト群は、本書のサンプルに含まれています。

本書のサンプルは以下の GitHub リポジトリで公開しておりますので、こちらを clone あるいはダウンロードしてください。

- `https://github.com/cupnes/gb_programming_with_shell-script_samples`

zip でダウンロードした場合はお好きな場所へ展開してください。

1.3　簡単なプログラムを作ってみる

プログラミングのための準備はこれで完了です[*1]。さっそく、簡単なプログラムを作ってみましょう。

ここでは、最も簡単なプログラムとして「起動後、無限ループするだけ」というものを作ってみることにします。

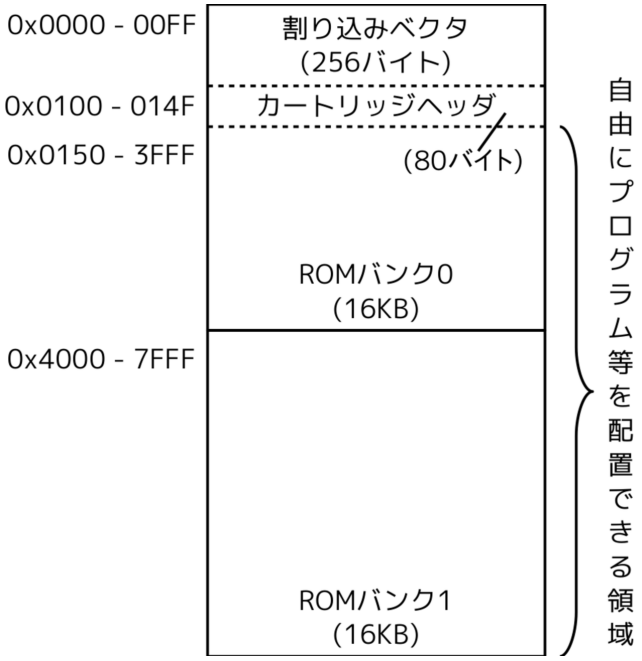
♣ 本書における GB プログラミングの方針

まず、本書での GB プログラミングの方針を説明します。

GB プログラミングのアウトプットは「GB の ROM ファイル」です。本書ではこの ROM ファイルをシェルスクリプトで生成します。

ROM ファイルは GB のメモリ空間の ROM の領域そのものです。GB のアドレス幅は 16 ビットで 0x0000 から 0xFFFF までのアドレス空間があります。その内、ROM の領域は 0x0000 から 0x7FFF までの 32KB で、この部分のメモリマップは図 1.1 の通りです。

[*1] テキストエディタは適宜お好きなものをご用意ください。



▲図 1.1: GB の ROM 領域のメモリマップ

本書ではこの 32KB をシェルスクリプトで生成します。

♣ 割り込みベクタを作る

前節で紹介した ROM 領域のメモリマップを先頭から埋めていきましょう。

まずは、割り込みベクタです。ここは割り込みやリセットが発生した際にジャンプしてくる 256 バイトの領域です。アドレスごとに「どの割り込み/リセットのときにそこへジャンプしてくるか」が決まっています。

ここでは割り込みは使わないので、この領域は全て nop という命令で埋めることにします。nop は「何もしない (No OPeration)」という CPU 命令です。

後述するカートリッジヘッダの先頭にはプログラム本体へのジャンプ命令を配置するため、割り込みベクタを nop で埋めておけば、どのような割り込み/リセットが発生したとしても、何もせずにカートリッジヘッダの先頭まで CPU の命令実行が進み、プログラム本体へジャンプさせることができます。

では、割り込みベクタを作ってみましょう。

まず、作業場所として、サンプルリポジトリを clone あるいはダウンロードして展開したディレクトリへ移動します。

```
$ cd gb_programming_with_shell-script_samples
```

nop は機械語で 0x00 です。ここでは dd コマンドを使って全て 0 の 256 バイトのファイルを生成します。

```
$ dd if=/dev/zero of=vec.dat bs=1 count=256
256+0 レコード入力
256+0 レコード出力
256 bytes copied, 0.00247894 s, 103 kB/s
$
```

これで割り込みベクタを作成できました。

♣ カートリッジヘッダを作る

次にカートリッジヘッダを作ります。

カートリッジヘッダは、先述の通り「GB の起動時にどこへジャンプするか」というプログラムのエントリポイントや、ゲームタイトル文字列、カートリッジタイプ (ゲームボーイかゲームボーイカラーか・バッテリ搭載の有無など) などと末尾にチェックサムが書かれている領域です。

本書では、ゲームボーイ用でバッテリ無しの ROM(ROM only) でゲームタイトル文字列も空で作成することにします。チェックサムはエントリポイントアドレス以降の領域で計算するため、タイトル文字列やカートリッジタイプなどを固定にしておくとチェックサム含めて固定のバイト列で使いまわすことができます。

そのようにしてカートリッジヘッダの生成をシェル関数化したものを、サンプルの `include/gb.sh` に `gb_cart_header_no_title` という名前の関数で用意しています。

その他にも GB の LR35902[*2]という CPU の CPU 命令や、いくつかの処理をシェル関数化しており、 `gb.sh` を source コマンドで読み込むことで使えるようにしています。

`gb_cart_header_no_title` はエントリポイントアドレスを引数にとります。ROM で自由にプログラムを配置できる領域は 0x0150 以降なので、エントリポイントは 0x0150 にします。

それでは、このシェル関数を使ってカートリッジヘッダを生成してみましょう。

```
$ . include/gb.sh
$ gb_cart_header_no_title 0150 >head.dat
```

[*2] 8 ビット CPU で、Z80 のカスタム CPU(シャープ製)

```
$
```

これで head.dat というファイル名でカートリッジヘッダのバイナリを生成できました。

■注意：include ディレクトリと同じ階層で作業すること

なお、 include ディレクトリ内のスクリプトを source コマンド (あるいは簡略表現の".") で読み込む際は、include が存在するディレクトリと同じディレクトリで行ってください。

include 内のスクリプトはその他のスクリプトに定義された処理を使用する際はそのスクリプトを . include/スクリプト名 のように読み込んでいます。実行時のカレントディレクトリが include ディレクトリが存在するディレクトリであることを想定して作られているので、サブディレクトリを作ってその中で作業する際は、その中に include ディレクトリのシンボリックリンクを作成するか include ディレクトリをコピーして配置してください。

♣ 無限ループのプログラムを作成する

続いて、0x0150 以降の領域を作成します。ここからがプログラム本体です。好きなようにプログラムやデータを配置することができます。

カートリッジヘッダに、起動後のエントリアドレスとしてこの領域先頭である 0x0150 を指定しましたので、ゲームボーイの電源を入れて Nintendo ロゴが表示された後、0x0150 へジャンプしてきます。

今回は「無限ループするだけのプログラム」ということで、ここに無限ループのプログラムを配置することにします。

プログラムは CPU が直接解釈する機械語で配置します。ただ、手で機械語の 16 進数を書いていくのは少しつらいので、CPU の機械語命令のバイナリを生成するシェル関数を用意しています[*3]。

ここでは、CPU の相対ジャンプ命令を使って無限ループを実現します。各 CPU 命令は lr35902_ から始まる名前のシェル関数で実装しています。相対ジャンプ命令は lr35902_rel_jump という名前のシェル関数です。引数にジャンプする相対的なバイト数を 2 桁の 16 進数 (1 バイト) で指定します。

試しに、引数に 0x00 を指定して実行してみます。標準出力へ機械語バイナリを出力するので、適当なファイルへリダイレクトします。

[*3] まだ全部ではないと思いますが、GB 上で動く OS もどきを作るのに困らないくらいはあります。

```
$ lr35902_rel_jump 00 >rel.dat
$
```

`lr35902` から始まる CPU 命令をシェル関数化したものを実行すると、カレントディレクトリに `asm.lst` というファイルが生成されます。ここには生成した機械語命令のアセンブラのリストが追記されていきます。特に本書で使ったりはしませんが、デバッグ時などに参考にしてみてください。(コロン区切りの右側の数字はその命令にかかるサイクル数です。)

```
$ cat asm.lst
jr $00  ;12
$
```

リダイレクトしたバイナリを hexdump で見てみると `0x18 00` という 2 バイトが生成されていることがわかります。

```
$ hexdump -C rel.dat
00000000  18 00                                             |..|
00000002
$
```

これが相対ジャンプ命令の機械語バイナリです。`0x18` が「相対ジャンプ命令」であることを示し、続く `0x00` がジャンプする相対的なバイト数を示しています (引数で与えた 0x00 がそのまま入っています)。

相対ジャンプ命令で指定する「相対的なバイト数」は、「自身の次の命令」が基準です。そのため、今回のように 0 を指定した場合、「自身の次の命令」へジャンプすることになります[*4]。

無限ループにするためには自身の命令のバイト数マイナスの位置へジャンプする必要があります。マイナスの表現には 2 の補数を使います。相対ジャンプ命令のバイト数は 2 バイトなので、-2 を表す 0xfe を指定すると無限ループになります。

```
$ lr35902_rel_jump fe >inf.dat
$
```

これで、無限ループのプログラムを作ることができました。

なお、2 の補数も都度計算するのは面倒なので、シェル関数を用意しています。

`two_comp` というシェル関数は、引数で与えた 16 進数 2 桁の数値を 2 の補数によるマイナス表現へ変換します。例えば、-2 の 2 の補数表現を得る際は以下のように使用

[*4] 単に次の命令へ進むだけで、nop と同じ挙動

します。

```
$ two_comp 02
FE
$
```

これは、そのまま相対ジャンプ命令の引数に与えることができます。

```
$ lr35902_rel_jump $(two_comp 02) | hexdump -C
00000000  18 fe                                            |..|
00000002
$
```

`lr35902_rel_jump` の出力をそのまま hexdump に与えて確認すると、`0x18 fe` という 2 バイトが生成できています。`0x18` という相対ジャンプ命令のオペランドとして `0xfe` (-2 の 2 の補数表現) が与えられており、意図通りです。

また、10 進数で指定した数を 2 の補数によるマイナス表現へ変換する関数 `two_comp_d` も用意しています。(適宜使い分けます)

♣ 余白を埋める

ここまでで、割り込みベクタ (256 バイト)・カートリッジヘッダ (80 バイト)・機械語プログラム (2 バイト) を作成しました。ROM ファイルを 32KB とするために、残る ROM の領域を適当なデータで埋めます。

残る ROM の領域は、`(32*1024)-256-80-2=32430` より、32430 バイトです。このサイズ分を全て 0 で埋めたデータを dd コマンドで作成します。

```
$ dd if=/dev/zero of=pad.dat bs=1 count=32430
32430+0 レコード入力
32430+0 レコード出力
32430 bytes (32 kB, 32 KiB) copied, 0.0553519 s, 586 kB/s
$
```

♣ 結合して ROM ファイル完成

ここまでで作成した全てを結合すると ROM ファイルになります。

```
$ cat cat vec.dat head.dat inf.dat pad.dat >inf.gb
$
```

無限ループするだけの ROM ということで、`inf.gb` という ROM ファイル名で作成してみました。

1.4　エミュレータで実行してみる

エミュレータは何でも構いませんが、本書では「BGB」というエミュレータで説明します。

BGB は以下のウェブサイトからダウンロードできます。

- `http://bgb.bircd.org/`

「downloads」というリンクをクリックするとジャンプするページ下部の以下の場所で実行バイナリをダウンロードできます。

- `http://bgb.bircd.org/#downloads`

筆者は 64 ビット版のアーカイブ「bgbw64.zip」をダウンロードして使用しています。

公開されている実行バイナリは Windows 向けのみですが、Linux 上でも Wine[*5]を使用することで問題なく動作します[*6]。

Linux の場合について、Wine のセットアップ方法を詳しく説明はしませんが、Debian/Ubuntu などの APT が使える環境の場合、以下のあたりをインストールすれば問題ないかと思います。

- wine64
- winetricks

`winetricks` は、Wine 用のツールやデータをダウンロードしたり設定したりするパッケージマネージャのようなものです。Wine で BGB を起動した際にアルファベットが文字化けしたため、その対処を行うためにインストールしました。このような問題に陥った場合の対処方法は、筆者が参考にさせていただいたブログ記事を本書末尾の「参考にさせてもらった情報」に記載していますので、参照してみてください。

Wine を使用する場合、以下のように BGB を起動できます。なお、BGB の zip アーカイブ (bgbw64.zip) はホームディレクトリ直下に展開されてるとします。

```
$ wine64 ~/bgbw64/bgb64.exe -nowarn ROMファイル名
```

なお、`-nowarn` のオプションは、カートリッジヘッダの生成時に設定を省いている一部のチェックサムのために付けています。実機では参照されないチェックサムなのですが[*7]、BGB ではそこも確認し、起動時にワーニングのメッセージを表示するため、それをスキップするようにしています。

[*5] Linux 上で Windows アプリケーションを動作させるツール。

[*6] 筆者自身、Linux(Debian) 上で Wine を使用して BGB を動作させています。BGB のために Wine をインストールするくらい、BGB は色々なデバッグができる強力なエミュレータです。

[*7] カートリッジヘッダの「グローバルチェックサム」というものです。詳しくは、本書末尾の「参考に

以降、BGB で ROM ファイルを動作させることを示す際のコマンドは以下のように記載します[*8]。

```
$ bgb ROMファイル名
```

なお、Windows の場合は、上記の表記が出てきた場合も、よしなに BGB の実行ファイルをダブルクリックして ROM ファイルを開くなりしてもらえれば構いません。

それでは、作成した ROM ファイル `inf.gb` を起動してみましょう。

```
$ bgb inf.gb
```

起動すると図 1.2 のような画面が表示されます。

▲図 1.2: inf.gb の実行画面

させてもらった情報」の「Everything You Always Wanted To Know About GAMEBOY」の「The Cartridge Header」の章を参照してください。

[*8] 適宜読み替えても良いですし、シェルへのエイリアス設定追加や、パスの通った場所に"bgb"というファイル名で起動用のスクリプトを配置しても構いません。

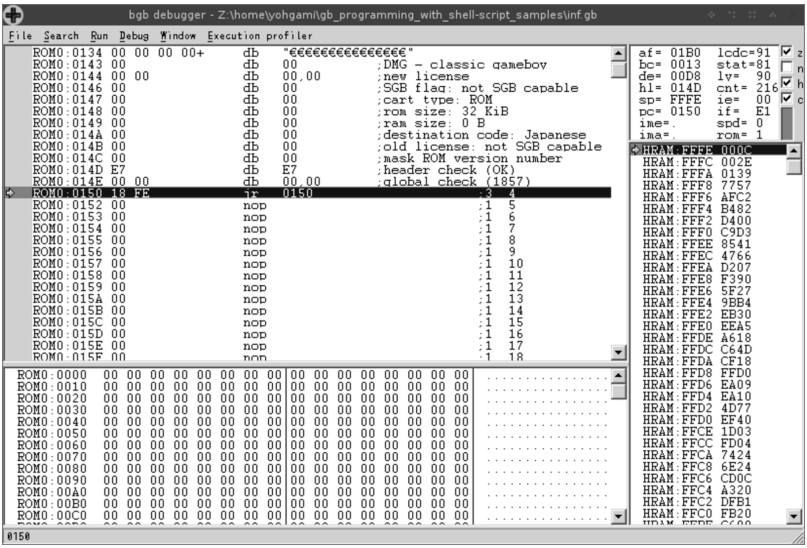
「Nintendo」というロゴが表示されている状態がゲームボーイの起動直後のデフォルト画面です。

無限ループしているだけなので、画面上は何も変化がありません。

ちゃんと意図通りに動いているのか確認するために、デバッグ画面を開いてみましょう。

右クリックで表示されるメニューから [Other] の中の [Debugger] をクリックしてください。

すると、図 1.3 の画面が表示されます。



▲図 1.3: inf.gb のデバッグ画面

色々と表示されていますが、ここでは 4 分割された中の左上の領域を見てください。

ここには、「今 ROM のどこを実行中であるか」が、機械語を逆アセンブルした結果とともに表示されています。

右向き矢印で示された行が今実行中の箇所で、この行のみ抜き出すとリスト 1.1 の通りです。

▼リスト 1.1: 今実行中の行

```
ROM0: 0150 18 FE            jr 0150
```

この行は「ROM のバンク 0 内でアドレスが 0x0150」、「0x18 fe というバイナリが書かれていて逆アセンブルすると jr 0150」ということを表しています。

jr は相対ジャンプ命令のアセンブラ表現で、オペランドが 0150 になっているのは BGB が逆アセンブル時に絶対アドレスへ変換しているためです。(機械語を見ると 18 fe と、意図通りに書かれています。)

「0x0150 へジャンプし続ける無限ループ」が実現できており、意図通りに動いていることを確認できました。

本書ではこのようにして GB の ROM ファイルを作成していきます。

1.5　次章からのための準備

基本的には以上のやり方で GB の ROM ファイルを作っていくことになります。

ただ、都度コマンドを手打ちするのは面倒なので、今後はシェルスクリプト化したものをベースに説明します。

♣ シェルスクリプト化

この章の内容をリスト 1.2 のシェルスクリプトにします。

▼リスト 1.2: 01_inf/01_inf.sh

```
#!/bin/bash

set -uex

. include/gb.sh

PROG_SIZE=2

# 全てnopの割り込みベクタ生成
gb_all_nop_vector_table

# カートリッジヘッダ生成
gb_cart_header_no_title ${GB_ROM_FREE_BASE}

# 無限ループ
lr35902_rel_jump $(two_comp 02)

# 32KBに満たない分を0で埋める
dd if=/dev/zero bs=1 \
   count=$((GB_ROM_SIZE - GB_VECT_SIZE - GB_HEAD_SIZE - PROG_SIZE))
```

シェルスクリプト化にあたり、以下の変更を加えています。

- 実行時のデバッグ出力や未定義変数チェックなどのために、冒頭に `set -uex` を追加しています

- 割り込みベクタ生成処理は `gb_all_nop_vector_table` というシェル関数化しているため、それを使っています
- ROM のカートリッジヘッダ直後の自由に使える領域の先頭アドレスは `GB_ROM_FREE_BASE` という名前で変数を定義済みのため、それを使っています
- 32KB に満たない分を埋める処理のサイズ計算にも、定義済みの変数を使っています

シェルスクリプトでは、標準出力へ ROM バイナリを生成するようにしています。

ROM ファイルはシェルスクリプト実行時にファイルへリダイレクトすることで生成できます。

```
$ 01_inf/01_inf.sh >inf.gb
$
```

♣ halt を使う

次章以降を始める前の準備として、無限ループの箇所を少し変更します。

このような OS レスのプログラミングにおいては、実行終了後の戻り先が無いこともあり、プログラムの最後では無限ループで停止させることがよくあります。

本書今後作っていくいくつかのサンプルもその方針なのですが、CPU の実行を進めないようにする際に、単なるビジーループは CPU に余計な負荷をかけてしまい、優しくないです。

`halt` という CPU 命令を使うと、割り込みなどのイベントがあるまで CPU を休ませることができるので、何もしないビジーループでも、halt を挟むと良いです。

先程シェルスクリプト化したものを halt を使用するように変更するとリスト 1.3 の通りです。

▼リスト 1.3: 01_inf/02_halt.sh

```
#!/bin/bash

set -uex

. include/gb.sh

PROG_SIZE=4     # 変更

# 全てnopの割り込みベクタ生成
gb_all_nop_vector_table
```

```
# カートリッジヘッダ生成
gb_cart_header_no_title ${GB_ROM_FREE_BASE}

# ・・・変更(ここから)・・・
# 無限halt
lr35902_halt
lr35902_rel_jump $(two_comp 04)
# ・・・変更(ここまで)・・・

# 32KBに満たない分を0で埋める
dd if=/dev/zero bs=1 \
   count=$((GB_ROM_SIZE - GB_VECT_SIZE - GB_HEAD_SIZE - PROG_SIZE))
```

halt 命令は `lr35902_halt` というシェル関数で用意しています。

2 バイトの命令にしている[*9]ので、相対ジャンプのジャンプの戻り先も 2 バイト分増やして `$(two_comp 04)` としています。

```
$ lr35902_halt | hexdump -C
00000000  76 00                                             |v.|
00000002
```

halt を使用して無限に待ち続ける、という処理は今後も良く書くと思われるので、`infinite_halt` というシェル関数化してあります。

これを使うように変更するとリスト 1.4 の通りです。

▼リスト 1.4: 01_inf/03_halt_2.sh

```
#!/bin/bash

set -uex

. include/gb.sh

PROG_SIZE=4

# 全てnopの割り込みベクタ生成
gb_all_nop_vector_table

# カートリッジヘッダ生成
gb_cart_header_no_title ${GB_ROM_FREE_BASE}
```

[*9] halt 命令自体は 0x76 のみの 1 バイトの命令なのですが、色々調べてみると halt 命令は実行時、次の命令を一つ実行してしまう挙動があるとの情報もあったので、halt 命令には明示的に nop(0x00) を一つ追加するようにしています。(halt を呼ぶような時は急いでいることも無いので)

```
# ・・・変更(ここから)・・・
# 無限halt
infinite_halt
# ・・・変更(ここまで)・・・

# 32KBに満たない分を0で埋める
dd if=/dev/zero bs=1 \
   count=$((GB_ROM_SIZE - GB_VECT_SIZE - GB_HEAD_SIZE - PROG_SIZE))
```

♣ 割り込みは一旦全て止める

割り込みは、使うまで一旦全て止めます。

割り込み機能自体を有効化/無効化する命令があり、ここで予め割り込みを無効化する命令を実行するようにしておきます。

処理を追加するとリスト 1.5 の通りです。

▼リスト 1.5: 01_inf/04_di.sh

```
#!/bin/bash

set -uex

. include/gb.sh

PROG_SIZE=5     # 変更

# 全てnopの割り込みベクタ生成
gb_all_nop_vector_table

# カートリッジヘッダ生成
gb_cart_header_no_title ${GB_ROM_FREE_BASE}

# ・・・追加(ここから)・・・
# 割り込み無効化
lr35902_disable_interrupts
# ・・・追加(ここまで)・・・

# 無限halt
infinite_halt

# 32KBに満たない分を0で埋める
dd if=/dev/zero bs=1 \
   count=$((GB_ROM_SIZE - GB_VECT_SIZE - GB_HEAD_SIZE - PROG_SIZE))
```

次章からはこのコードをベースに説明します。

第2章
背景にタイルを配置

GB では画像は全て「タイル」という単位で扱います。ここではタイルの定義方法を紹介し、タイルの定義と背景への配置を行ってみます。

2.1 GB の画面描画方式

GB の画面を制御する LCDC(LCD コントローラ) は、「タイル」という単位で画面を描画します。

「タイル」は 8x8 のビットマップで、専用のメモリ領域に定義します。タイルには連番で番号が対応付いており、例えば背景に何かを描画する際は、「N 番のタイルを配置する」というようにタイル番号を指定します。

♣ タイルについて

タイルを定義するメモリ空間は、設定により 2 つあるのですが、本書では 0x8000 から 0x8FFF の領域を使用します[*1]。

ゲームボーイのタイルの各ピクセルは 2 ビットで、2 * 8 * 8 = 128 ビット (16 バイト) で 1 つのタイルを表します。2 ビットで白黒なので、表現できる色は、黒・濃いグレー・薄いグレー・白の 4 色です。

♣ パレットについて

タイルの各 2 ビットを黒・濃いグレー・薄いグレー・白のどれと対応付けるかを決めるのが「パレット」という仕組みです。

パレットは 8 ビットのレジスタで、背景用に 1 つ (BackGround Palette:BGP) と、

[*1] もう片方は 0x8800 から 0x97FF です。タイルを配置する領域としてどちらを使うかで使い方が変わるのですが、本書では 0x8000 から 0x8FFF を使う場合についてのみ紹介します。

本書では扱いませんがオブジェクト用に 2 つ (Object Palette 0,1:OBP0,OBP1) の計 3 つあります。

パレットの各ビットの役割は表 2.1 の通りです。

▼表 2.1: パレットレジスタの各ビットの役割

ビット	役割
7-6	色番号 3 の色指定
5-4	色番号 2 の色指定
3-2	色番号 1 の色指定
1-0	色番号 0 の色指定

色番号 0 から 3 の全 4 つの色番号に何色を対応付けるかを、各 2 ビットずつの色指定で指定します。

そして、色指定の 2 ビットと色との対応が表 2.2 の通りです。

▼表 2.2: 色指定の 2 ビットと色の対応

11	黒
10	濃いグレー
01	薄いグレー
00	白

そのため、BGP へ 0xE4 を設定すれば、タイルを背景として描画する際、タイルの各ピクセル (2 ビット) を、「11 は黒」・「10 は濃いグレー」・「01 は薄いグレー」・「00 は白」と表示できます。

2.2 タイルを作ってみる

では、実際にタイルを定義してみましょう。

と言っても、1 タイル 16 バイトの中に 8x8 の各ピクセル (2 ビット) がどのように並ぶのかというと、それも少し特殊です[*2]。

[*2] 詳しくは本書末尾の「参考にさせてもらった情報」に記載しているリンク先などを参照してみてください。

16 バイトのタイルデータも、自動生成するスクリプトを用意しているので、本書ではそれを使います。

使用するスクリプトは、サンプルの `tools` ディレクトリ内にある `txt22bpp` です。「2bpp」は「2 Bits Per Pixel」の略で、GB のピクセルフォーマットをこのように呼んだりします。

`txt22bpp` は、8x8 のビットマップをテキストで表現したテキストファイルを与えると、2bpp 形式のタイルバイナリデータへ変換します。

例えばリスト 2.1 のようなファイルを引数に与えることができます。

▼リスト 2.1: txt22bpp へ与えるテキストの例

```
####****
####****
####****
####****
    ....
    ....
    ....
    ....
```

`txt22bpp` は、'#' を色番号 3 へ、'*' を色番号 2 へ、'.' を色番号 1 へ、' ' を色番号 0 へ変換します `02_about_txt22bpp_case` 。

リスト 2.1 を `tile.txt` というファイル名で保存したとすると、以下のようにタイルデータを生成できます。

```
$ tools/txt22bpp tile.txt tile.2bpp
$ ls tile.2bpp
tile.2bpp
$
```

生成された `tile.2bpp` が 2bpp 形式のタイルデータバイナリです。

2.3 タイルをロードしてみる

タイルはメモリ上 0x8000 以降のタイルデータ領域へロードすることで使えるようになります。この領域は先頭から 16 バイト毎にタイル番号が割り振られていきます。(0x8000 以降の 16 バイトの領域にロードしたタイルデータはタイル番号 0 番、0x8010 以降の 16 バイトの領域にロードしたタイルデータはタイル番号 1 番、という具合です。)

ここでは、0x8000 以降の 16 バイトの領域に先程生成したタイルデータをロードして

タイル番号 0 番として使えるようにしてみます。

♣ VRAM へアクセスするには

タイルデータ領域や、背景用のタイルマップ領域、オブジェクトを定義する領域などの VRAM の領域は、画面描画を行う LCDC もアクセスします。LCD がアクセスしている間、CPU は VRAM へアクセスしてはいけないことになっています。

そのため、VRAM を読み書きする際は以下のいずれかの対応を行う必要があります。

- LCD を止める
  - 止めている間、VRAM へアクセスし放題
  - ただし、画面が消える (真っ白になる)
  - 止める操作は V ブランク期間に行う必要がある
- LCD がアクセスしていない期間にアクセスするようにする
  - 次に LCD がアクセスし始めるまでは VRAM アクセス可能
  - LCD のステータスを監視する、割り込みを使う等でタイミングを計る必要がある
- DMA を使う
  - オブジェクトデータの転送にのみ使える

DMA は用途が決まっているのでここでは該当しないとして、「LCD を止める」方法と「LCD がアクセスしていない期間にアクセスするようにする」方法は、ゲーム起動直後に前者の方法で VRAM を初期化し、その後に VRAM へアクセスする必要がある際は後者の方法でアクセスする、と使い分けていることが多いようです。

今の実装は正に起動直後なので、「LCD を止める」方法で実装してみることにします。

♣ LCD を止めるためには

LCD が画面描画を行う中ではいくつかの期間があり、その中で VRAM へアクセスしていない期間はいくつかあるのですが、LCD の停止はその中でも「V ブランク」期間限定でハードウェア的に許されている操作です。

ただ、「LCD が V ブランク期間になるのを待つ」処理は `gb_wait_for_vblank_to_start` という関数で実装済みです。今回はこのシェル関数を使えば良いので、LCD の振る舞いについて詳しくは説明しません[*3]。

[*3] ただ、ディスプレイの画面更新処理はこのようなレトロなハードでプログラミングする際の基礎知識だとも思うので、興味があれば「LCD H ブランク V ブランク」などのキーワードで調べてみると面白いです。

♣ LCDC のレジスタについて

そして、V ブランクを待った上で、LCD を止めるにはどうするかというと、LCDC のレジスタを操作します。LCD の有効/無効自体はこのレジスタの最上位ビットなので、このビットだけ操作すれば LCD を止めることができます。

ただ、LCD についてはいくつか設定しなければならない部分もあるので、LCDC のレジスタの各ビットについてここで説明します。

LCDC の 8 ビットレジスタの各ビットをまとめると表 2.3 の通りです。

▼表 2.3: LCDC のレジスタ

ビット	役割
7	LCD の有効 (=1)/無効 (=0)
6	ウィンドウタイルマップアドレス設定 (0=0x9800-9BFF, 1=0x9C00-9FFF)
5	ウィンドウの有効 (=1)/無効 (=0)
4	背景とウィンドウ用のタイルデータ配置アドレス設定 (0=0x8800-97FF, 1=0x8000-8FFF)
3	背景タイルマップアドレス設定 (0=0x9800-9BFF, 1=0x9C00-9FFF)
2	オブジェクトサイズ (0=8x8, 1=8x16)
1	オブジェクトの有効 (=1)/無効 (=0)
0	背景の有効 (=1)/無効 (=0)

「ウィンドウ」や「オブジェクト」は、本書では扱いませんが、簡単に説明すると、GB の画面は 3 層のレイヤー構造のようになっていて、一番手前にあるのが「オブジェクト」、次が「ウィンドウ」、最後が「背景」です。そして、それぞれは単なるレイヤーではなく機能が異なります。

「オブジェクト」は「スプライト」などとも呼ばれるもので、8x8(タイル 1 枚) あるいは 8x16(タイル 2 枚) の画像を座標やその他いくつかの属性情報とともに管理する機能で、オブジェクトに紐付いた座標値のみ更新するだけで、LCD はそのオブジェクトを指定された位置に描画してくれます。

「ウィンドウ」は機能的にはほとんど「背景」と同じで、VRAM 上のタイルマップ領域に「この座標にはこのタイル」と指定することで画面上にタイルを描画します。透過色や、ウィンドウ幅・高さの変更などはできないので、有効にすると、画面サイズ分のスクリーンがそのまま 1 枚、背景を覆い尽くす形で描画されることになります。ただ、ウィンドウの開始座標 (左上座標) は変更できるので、画面下部にメッセージウィンドウを描画する、などに使うことができます。

以上を踏まえ、本書では LCDC には表 2.4 の設定を行うことにします。

▼表 2.4: 本書の LCDC 設定

ビット	設定値
7	有効 (=1)/無効 (=0) は状況に応じて設定
6	背景に使わない側を設定 (1=0x9C00-9FFF)
5	ウィンドウは無効 (=0)
4	タイルデータ配置アドレスは 0x8000-8FFF(=1) にする
3	背景タイルマップアドレスは 0x9800-9BFF(=0) にする
2	オブジェクトは使わないのでどちらでも良い (=0)
1	オブジェクトは無効 (=0)
0	背景は有効 (=1)

表 2.4 を踏まえて、LCD の無効 (ビット 7 を 0) も行うと、LCDC のレジスタ設定値は 2 進数で 0b0101 0001 で 16 進数で 0x51 です。

それでは、この設定を LCDC へ行う処理を追加してみます (リスト 2.2)。

▼リスト 2.2: 02_bg/01_stop_lcd.sh

```
#!/bin/bash

set -uex

. include/gb.sh

# ・・・追加(ここから)・・・
main() {
        # 割り込み無効化
        lr35902_disable_interrupts

        # Vブランク期間を待つ
        gb_wait_for_vblank_to_start

        # LCD設定&LCD止める
        lr35902_set_reg regA 51
        lr35902_copy_to_ioport_from_regA $GB_IO_LCDC

        # 無限halt
        infinite_halt
}
# ・・・追加(ここまで)・・・

# 全てnopの割り込みベクタ生成
```

```
gb_all_nop_vector_table

# カートリッジヘッダ生成
gb_cart_header_no_title ${GB_ROM_FREE_BASE}

# ・・・変更(ここから)・・・
# main処理
main >main.o
cat main.o

# 32KBに満たない分を0で埋める
main_size=$(stat -c '%s' main.o)
dd if=/dev/zero bs=1 \
   count=$((GB_ROM_SIZE - GB_VECT_SIZE - GB_HEAD_SIZE - main_size))
# ・・・変更(ここまで)・・・
```

処理を追加するにあたり、「32KB に満たない分を 0 で埋める」の箇所の計算を都度行うのは面倒なので、プログラム本体部分は `main` というシェル関数に記述し、その部分のみ `main.o` というファイルへ出力して、そのファイルサイズを使うようにしました。

「LCD 設定&LCD 止める」処理は 2 つの命令で行っています。まず 1 つ目の `lr35902_set_reg regA 51` で、レジスタ A へ LCDC へ設定する値 0x51 を格納しています。 `lr35902_set_reg` は「汎用レジスタ」へ値を設定する命令をシェル関数化したものです。GB の CPU である LR35902 の汎用レジスタについて詳しくは後述しますが、ここでは A という 8 ビットのレジスタに 0x51 という値を格納しています。

そして、`lr35902_copy_to_ioport_from_regA $GB_IO_LCDC` でレジスタ A の内容を LCDC のレジスタへ設定しています。 `lr35902_copy_to_ioport_from_regA` は、レジスタ A の内容を指定された IO レジスタへ設定する命令をシェル関数化したものです。

♣ タイルを ROM へ追加

VRAM のタイルデータ領域へロードするには、そもそもロード対象であるタイルデータが ROM 内に存在する必要があります。

タイルデータは 2bpp 形式で作成しましたので (`tile.2bpp`)、それを ROM の中に組み込むようにします (リスト 2.3)。

▼リスト 2.3: 02_bg/02_add_tile_to_rom.sh

```
#!/bin/bash

set -uex

. include/gb.sh

# ・・・追加(ここから)・・・
TILE_SIZE=10    # 16バイト

# エントリアドレスは自由に使えるROM領域の先頭(0x0150)から
# タイルを配置した16バイト先のアドレスになる
ENTRY_ADDR=$(echo "${GB_ROM_FREE_BASE}+${TILE_SIZE}" | bc)
# ・・・追加(ここまで)・・・

main() {
        # ・・・追加(ここから)・・・
        # 自由に使える領域の先頭(0x0150)にタイルデータを追加する
        cat tile.2bpp
        # ・・・追加(ここまで)・・・

        # 割り込み無効化
        lr35902_disable_interrupts

        # Vブランク期間を待つ
        gb_wait_for_vblank_to_start

        # LCD設定&LCD無効化
        lr35902_set_reg regA 51
        lr35902_copy_to_ioport_from_regA $GB_IO_LCDC

        # 無限halt
        infinite_halt
}

# 全てnopの割り込みベクタ生成
gb_all_nop_vector_table

# カートリッジヘッダ生成
gb_cart_header_no_title $ENTRY_ADDR

# main処理
main >main.o
cat main.o

# 32KBに満たない分を0で埋める
main_size=$(stat -c '%s' main.o)
```

```
dd if=/dev/zero bs=1 \
   count=$((GB_ROM_SIZE - GB_VECT_SIZE - GB_HEAD_SIZE - main_size))
```

タイルデータをロードする際にタイルデータ自身の位置がずれると面倒なので、タイルデータは自由に使える ROM 領域の先頭へ配置しました。

それに伴って、実行時のエントリアドレスがタイルサイズ分 (16 バイト) 後ろにずれるので、その調整を行っています。

♣ GB の CPU が持つ汎用レジスタについて

次の項でロード処理を実装するにあたり、ここで、GB の CPU(LR35902) が持つ汎用レジスタについて説明します。

LR35902 が持つ汎用レジスタ (一部そうでないものも混ざっていますが) をまとめると表 2.5 の通りです。

▼表 2.5: LR35902 の汎用レジスタ

16 ビット	8 ビット	8 ビット	備考
AF	A	-	F はフラグレジスタ
BC	B	C	
DE	D	E	
HL	H	L	

8 ビットレジスタが A・B・C・D・E・H・L の計 7 個があります。

そして、それぞれは合体させることで 16 ビットレジスタとして使うことも可能で、AF・BC・DE・HL の 4 つの 16 ビットレジスタとして機能します。合体させる際の上位側・下位側も名前の並びの通りです。(例えば、レジスタ BC は、レジスタ B が上位 8 ビットでレジスタ C が下位 8 ビット)

ただし、A と合体させた F は「フラグレジスタ」という演算結果に応じて変化する専用のレジスタなので、AF については「16 ビットの値を格納」などには使えません。

♣ タイルデータを VRAM へロードする

それでは、ROM に配置したタイルデータを VRAM のタイルデータ領域へロードしてみましょう。

実装の方針は以下の通りです。

- レジスタ BC へ ROM 上のタイルデータのアドレス (0x0150) を設定
- レジスタ HL へ VRAM 上のタイルデータロード先アドレス (0x8000) を設定

- レジスタ C へタイルサイズ (16) を設定
- 以下をレジスタ C が 0 になるまで繰り返す
  - レジスタ BC が指す先 1 バイトをレジスタ HL の指す先へコピー
  - レジスタ BC と HL をそれぞれインクリメント
  - レジスタ C をデクリメント

これを実装するとリスト 2.4 の通りです。

▼リスト 2.4: 02_bg/03_load_tile_to_vram.sh

```
# ・・・省略・・・

VRAM_TILE_DATA_BASE=8000        # 追加

main() {
        # 自由に使える領域の先頭(0x0150)にタイルデータを追加する
        cat tile.2bpp

        # 割り込み無効化
        lr35902_disable_interrupts

        # Vブランク期間を待つ
        gb_wait_for_vblank_to_start

        # LCD設定&LCD無効化
        lr35902_set_reg regA 51
        lr35902_copy_to_ioport_from_regA $GB_IO_LCDC

        # ・・・追加(ここから)・・・
        # タイルデータをVRAMのタイルデータ領域(0x8000)へロードする
        lr35902_set_reg regDE $GB_ROM_FREE_BASE
        lr35902_set_reg regHL $VRAM_TILE_DATA_BASE
        lr35902_set_reg regC $TILE_SIZE
        (
                lr35902_copy_to_from regA ptrDE
                lr35902_copyinc_to_ptrHL_from_regA
                lr35902_inc regDE
                lr35902_dec regC
        ) >main.1.o
        cat main.1.o
        local sz_1=$(stat -c '%s' main.1.o)
        lr35902_rel_jump_with_cond NZ $(two_comp_d $((sz_1+2)))
        # ・・・追加(ここまで)・・・

        # 無限halt
        infinite_halt
}
```

```
# ・・・省略・・・
```

レジスタに数値を設定する命令も `lr35902_set_reg` という名前でシェル関数化しています。 `lr35902_set_reg regDE $GB_ROM_FREE_BASE` というようにレジスタ DE にタイルデータのアドレスを設定していて、HL や C も同様の方法で値を設定しています。

`()` でサブシェル化している箇所がループで繰り返し実行される部分です。ここで ROM から VRAM へのタイルデータのコピーを 1 バイトずつ行っています。各行を説明すると、まず `lr35902_copy_to_from regA ptrDE` で、レジスタ DE が指す先の 1 バイトをレジスタ A へコピーします。 `lr35902_copy_to_from` は第 1 引数のレジスタへ第 2 引数のレジスタをコピーする機械語を生成するシェル関数で、 `ptrDE` のようにレジスタ指定するとそのレジスタをポインタとして使います[*4]。次に `lr35902_copyinc_to_ptrHL_from_regA` でレジスタ HL が指す先へレジスタ A をコピーします。なお、このシェル関数ではレジスタ HL のインクリメントも同時に行います[*5]。これで 1 バイト分の ROM から VRAM へのコピーが完了です。 `lr35902_inc regDE` でレジスタ DE をインクリメントし、 `lr35902_dec regC` でレジスタ C をデクリメントします。

そして、このサブシェル部分は出力を `main.1.o` というファイルへリダイレクトしています。これにより、このサブシェル部分は直ちに標準出力へ出力するのではなく、一旦ファイルへ保存するようにしています。ただこの場合は直ちに cat でその内容を標準出力へ出力しているわけですが、何故このようなことをしているのかというと、この後、この処理をループさせるために相対ジャンプ命令で「何バイト戻れば良いのか」を算出するためです。 `local sz_1=$(stat -c '%s' main.1.o)` でサブシェル化した部分のバイト数を `sz_1` という変数へ格納し、 `lr35902_rel_jump_with_cond NZ $(two_comp_d $((sz_1+2)))` で相対ジャンプする際の戻るバイト数の計算に使っています。2 を足しているのは、相対ジャンプ命令自体のバイト数 (2 バイト) です。

最後に、この相対ジャンプ命令のシェル関数には `_with_cond` が付いています。これは「条件付き相対ジャンプ命令」という「直前の演算結果に応じてジャンプするか否かを変える」命令をシェル関数化したものです。今回の場合、第 1 引数に `NZ` を指定しているので「直前の演算結果がゼロでは無い場合 (Not Zero)」にジャンプすることに

[*4] C 言語のポインタと同じ意味です。そのレジスタの中身ではなく、そのレジスタに格納されている値をアドレスとして使い、そのアドレスが指す先に作用するようになります。

[*5] レジスタ HL に対しては同時にインクリメントも行う命令 (ldi) が用意されているため、それに対応するシェル関数を用意しています。

なります。直前の演算はレジスタ C のデクリメントなので、レジスタ C が 0 ではない間、サブシェル化した部分の先頭へジャンプすることになります。

♣ パレットを設定する

この節の最後に背景用パレット (BGP) を設定しておきます。

パレットについては先述の通りで、0xE4 を設定します。実装はリスト 2.5 の通りです。

▼リスト 2.5: 02_bg/04_set_bgp.sh

```
# ・・・省略・・・

main() {
        # ・・・省略・・・

        # タイルデータをVRAMのタイルデータ領域(0x8000)へロードする
        lr35902_set_reg regDE $GB_ROM_FREE_BASE
        lr35902_set_reg regHL $VRAM_TILE_DATA_BASE
        lr35902_set_reg regC $TILE_SIZE
        (
                lr35902_copy_to_from regA ptrDE
                lr35902_copyinc_to_ptrHL_from_regA
                lr35902_inc regDE
                lr35902_dec regC
        ) >main.1.o
        cat main.1.o
        local sz_1=$(stat -c '%s' main.1.o)
        lr35902_rel_jump_with_cond NZ $(two_comp_d $((sz_1+2)))

        # ・・・追加(ここから)・・・
        # LCDを再開させる
        lr35902_copy_to_regA_from_ioport $GB_IO_LCDC
        lr35902_set_bitN_of_reg 7 regA
        lr35902_copy_to_ioport_from_regA $GB_IO_LCDC

        # BGP設定
        lr35902_set_reg regA $BGP_VAL
        lr35902_copy_to_ioport_from_regA $GB_IO_BGP
        # ・・・追加(ここまで)・・・

        # 無限halt
        infinite_halt
}

# ・・・省略・・・
```

パレット設定は VRAM とは別の単なるレジスタ設定なので、LCD 再開後でも構いません。そこで、LCD 再開の処理も追加して、その後で BGP 設定を行うようにしました。

シェル関数名や変数名からやっていることはわかるかと思いますが、「LCD を再開させる」の箇所では、`lr35902_copy_to_regA_from_ioport $GB_IO_LCDC` で LCDC レジスタの内容をレジスタ A へロードした後、`lr35902_set_bitN_of_reg 7 regA` でビット 7 のみセット (LCD 有効化) し、`lr35902_copy_to_ioport_from_regA $GB_IO_LCDC` で LCDC レジスタへ書き戻しています。

「BGP 設定」の方は設定方法自体は LCDC のときと同様なので、コードの説明は省略します。

♣ 試しに動作確認

この時点で一度、動作確認してみます。

シェルスクリプトをファイルへリダイレクトして、生成した ROM ファイルをエミュレータで実行してみてください。(前項時点のシェルスクリプトが `02_bg/04_set_bgp.sh` に保存されているとします。また、作成した `tile.2bpp` がカレントディレクトリに存在するようにしてください。)

```
$ 02_bg/04_set_bgp.sh >02_bg/04_set_bgp.gb
...
$ bgb 02_bg/04_set_bgp.gb
...
```

実行すると図 2.1 の画面が表示されます。

▲図 2.1: BGP 設定までの実行結果

Nintendo ロゴのみが表示されているデフォルト状態では、タイル番号 0 のタイルデータは真っ白のタイルだったのですが、今回そこへ自作のタイルのタイルデータを上書きしたため、背景のタイルマップでタイル番号 0 が指定されていた部分には自作のタイルが表示されるようになりました。

2.4　画面全体を自作タイルで敷き詰めてみる

では、この章の最後に画面全体を自作のタイル (タイル番号 0) で敷き詰めるように、背景のタイルマップ領域を初期化してみます。

♣ タイルマップ領域の使い方

LCDC レジスタ設定の際に、背景用のタイルマップ領域は 0x9800-9BFF(1024 バイト) のアドレスとしました。この領域をどのように使うかというと、画面左上から順にタイル番号 (1 バイト) を並べるだけです。

GB の画面は、表示領域は 160x144 ピクセルで、8x8 のタイルで換算すると 20x18 タ

イルなのですが、実は非表示の領域もあり、スクリーン全体としては 256x256 ピクセル (32x32 タイル) の大きさがあります。次章で紹介しますが LCD には画面スクロール機能があり、その機能のレジスタを設定すると非表示の領域まで画面をスクロールできます。

0x9800-9BFF(1024 バイト) のタイルマップ領域は、各 1 バイトがスクリーン全体 32x32 タイル (全 1024 タイル) の各タイルと対応づいていて、この各バイトにタイル番号を設定すると、画面の対応する位置のタイルがそのタイル番号のタイルになる、というわけです。

ここでは、非表示の部分も含めて全てタイル番号 0 で初期化したいので、0x9800-9BFF(1024 バイト) の領域を全て 0 クリアします。

♣ 実装

さっそくですが、実装してみるとリスト 2.6 の通りです。

▼リスト 2.6: 02_bg/05_init_bg.sh

```
# ・・・省略・・・

VRAM_TILE_DATA_BASE=8000
VRAM_BG_TILE_MAP_BASE=9800      # 追加
VRAM_BG_TILE_MAP_END=9bff       # 追加

BGP_VAL=e4

main() {
        # ・・・省略・・・

        # タイルデータをVRAMのタイルデータ領域(0x8000)へロードする
        lr35902_set_reg regDE $GB_ROM_FREE_BASE
        lr35902_set_reg regHL $VRAM_TILE_DATA_BASE
        lr35902_set_reg regC $TILE_SIZE
        (
                lr35902_copy_to_from regA ptrDE
                lr35902_copyinc_to_ptrHL_from_regA
                lr35902_inc regDE
                lr35902_dec regC
        ) >main.1.o
        cat main.1.o
        local sz_1=$(stat -c '%s' main.1.o)
        lr35902_rel_jump_with_cond NZ $(two_comp_d $((sz_1+2)))

        # ・・・追加(ここから)・・・
        # 背景用タイルマップ(0x9800-9BFF)をタイル番号0で初期化する
        ## 最終アドレス上位8ビット(0x9b)
```

```
        local end_th=$(echo $VRAM_BG_TILE_MAP_END | cut -c1-2)
        ## 最終アドレス下位8ビット(0xff)
        local end_bh=$(echo $VRAM_BG_TILE_MAP_END | cut -c3-4)
        lr35902_set_reg regHL $VRAM_BG_TILE_MAP_BASE
        (
                (
                        # 0x00をレジスタHLの指す先へ設定し、HLをインクリメント
                        lr35902_clear_reg regA
                        lr35902_copyinc_to_ptrHL_from_regA

                        # レジスタHと最終アドレス上位8ビットを比較
                        lr35902_copy_to_from regA regH
                        lr35902_compare_regA_and $end_th
                ) >main.2.o
                cat main.2.o
                local sz_2=$(stat -c '%s' main.2.o)
                # レジスタH != 0x9b なら
                # main.2.oのサイズ(+相対ジャンプ命令のサイズ)分戻る
                lr35902_rel_jump_with_cond NZ $(two_comp_d $((sz_2+2)))

                # レジスタLと最終アドレス下位8ビットを比較
                lr35902_copy_to_from regA regL
                lr35902_compare_regA_and $end_bh
        ) >main.3.o
        cat main.3.o
        local sz_3=$(stat -c '%s' main.3.o)
        # レジスタL != 0xff なら
        # main.3.oのサイズ(+相対ジャンプ命令のサイズ)分戻る
        lr35902_rel_jump_with_cond NZ $(two_comp_d $((sz_3+2)))
        # ・・・追加(ここまで)・・・

        # LCDを再開させる
        lr35902_copy_to_regA_from_ioport $GB_IO_LCDC
        lr35902_set_bitN_of_reg 7 regA
        lr35902_copy_to_ioport_from_regA $GB_IO_LCDC

        # ・・・省略・・・
}

# ・・・省略・・・
```

追加の行数が少し多めですが、戦略としては「レジスタ HL へタイルマップアドレスを設定し、最終アドレス (0x9bff) への設定完了まで、インクリメントしながら繰り返す」というものです。

ここで新たに登場したシェル関数は 2 つで、まず 1 つ目は `lr35902_clear_reg` です。これは、指定したレジスタをゼロクリアするシェル関数で、A レジスタ以外を指定

した場合、指定されたレジスタへ 0 を設定する命令を出力します。A レジスタについては xor でゼロクリアする命令を出力します[*6]。

そして、2 つ目は `lr35902_compare_regA_and` で、「レジスタ A と比較する」命令を出力するシェル関数です。LR35902 が持つ比較命令はレジスタ A との比較のみなので、このようなシェル関数名になっています。比較をするとそれに応じてフラグレジスタが変化するため、直後の条件付きジャンプ命令と組み合わせて使用します。

なお、比較が 8 ビットずつしか行えないため、「レジスタ H が最終アドレス上位 8 ビット (0x9b) と等しく無い間繰り返す (main.2.o のサブシェル部分)」と、それを囲む「レジスタ L が最終アドレス下位 8 ビット (0xff) と等しくない間繰り返す (main.3.o のサブシェル部分)」の 2 段のループを作り、レジスタ H も L も最終アドレスと等しい場合にループを脱出するようにしています。

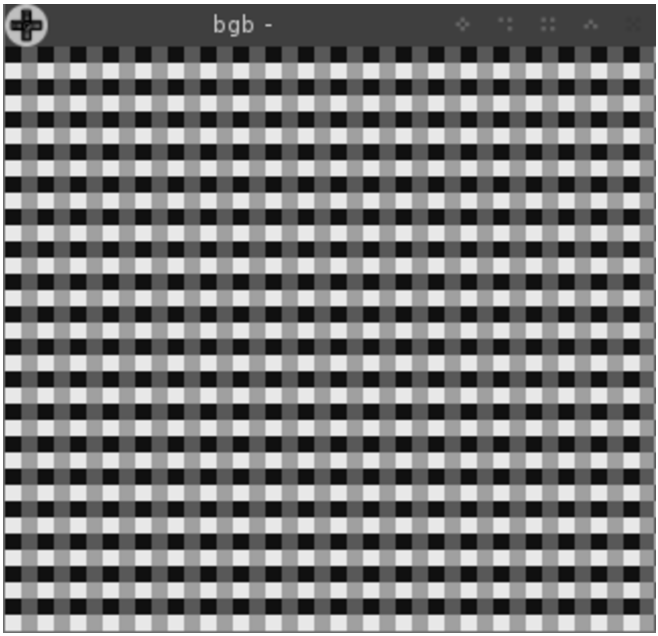
ここではあくまでも、背景用タイルマップ領域を全て 0 で初期化できれば、その実装方法は何でも良いです。(おそらくもっと良い実装もあるかも[*7])

♣ 結果確認

エミュレータで実行すると、今度は図 2.2 のように画面全体が自作のタイルで初期化されました。

[*6] xor 命令の方が 0 を設定する命令より命令バイト数が少ないためそのようにしています。LR35902 が持つ xor 命令は「レジスタ A と何かを xor した結果をレジスタ A へ格納する」というものであるためレジスタ A に関してのみ、xor でクリアするようにしています。

[*7] ただし、16 ビット変数に 1024 を入れてデクリメントしていく方針の場合、注意が必要です。例えばレジスタ BC をカウンタ用として、そこに 1024 を入れて、デクリメントしながらループを回す実装が考えられますが、デクリメント命令は 16 ビット演算の際はフラグを設定しないので、相対ジャンプ命令の条件にできません。結局、カウンタ用のレジスタ B と C を別々に 0 と等しいか比較する必要があります。



▲図 2.2: 背景用タイルマップ初期化後の実行結果

第3章
キー入力を試す

この章ではキー入力を使ってみます。例として十字キーに応じて画面スクロールするプログラムを作ってみます。

3.1 キー入力の取得方法

キー入力も LCDC やパレットと同様に専用のレジスタ (ジョイパッド,JOYP[*1]) から取得できます。8 ビットのレジスタで、各ビットについては表 3.1 の通りです。

▼表 3.1: キー入力のレジスタ (JOYP) について

ビット	役割
7	使用しない
6	使用しない
5	ボタン入力を選択 (0 で選択)
4	方向入力を選択 (0 で選択)
3	下/スタート状態 (0=押下)(読み取り専用)
2	上/セレクト状態 (0=押下)(読み取り専用)
1	左/B 状態 (0=押下)(読み取り専用)
0	右/A 状態 (0=押下)(読み取り専用)

ビット 5 と 4 で、ボタン (A/B/スタート/セレクト) の入力を取得するのか、方向 (上下左右) の入力を取得するのかを選択すると、下位 4 ビットで選択したキーの押下状態を取得できます。

[*1] IO レジスタに対するレジスタ名は資料によりまちまちです。本書では本書末尾「参考にさせてもらった情報」に記載の「Everything You Always Wanted To Know About GAMEBOY」を参考にしています。

3.2　画面スクロールの方法

画面スクロールもレジスタ操作で行います。以下の 2 つのレジスタを使用します。

- SCY - 表示領域原点 (左上) の Y 座標を指定
- SCX - 表示領域原点 (左上) の X 座標を指定

SCX と SCY で指定した座標を原点 (左上) として、そこから 160x144 ピクセルの領域を表示します。

また、それぞれのレジスタはオーバーフロー/アンダーフローした場合、境界をもう片方の端と自動的に合わせてくれるため、「右端より右へスクロールしたら左端が見える」という処理がハードウェア的に自動で行われます。

3.3　V ブランク割り込みを使用する

キーの入力状態の取得とそれに応じてスクロールレジスタを更新すること自体は後は実装するだけなのですが、問題は定期的に処理されるようにするにはどうするかです。

その際に使用するのが割り込みですが、キー入力の割り込みはボタンを押し込んだ時と指を話した時しか割り込みを発生しないので、押しっぱなしの間に継続的に何かの処理を行うことが難しいです。

ここでは、V ブランク割り込み時にキー状態の確認とスクロールレジスタ更新を行うことにします。V ブランクは、ざっくり説明すると、ディスプレイが画面を更新する一連の流れの中で、画面を描画し終わってから次に画面を描画し始める間の期間です。V ブランク割り込みは LCD がこの期間に入った際に発生する割り込みです。V ブランク期間中、LCD は VRAM へアクセスしないので、動作中にタイルデータやタイルマップ領域を更新したい場合にはこの期間で更新したりします。今回の場合、あるレジスタを読んで別のレジスタを更新するだけで、VRAM へアクセスしたりはしないのですが、画面描画に関するレジスタの更新なので、V ブランクと同期して実施すると都合が良いです。

V ブランク割り込みを使用するまでの流れは以下の通りです

- 割り込みベクタに V ブランク割り込み時の処理を記載
- 割り込みイネーブル (IE) レジスタで V ブランク割り込みを有効化する
- 割り込み機能自体を有効化

♣ 割り込みベクタに V ブランク割り込み時の処理を記載

まず、これまで全て nop で埋めていた割り込みベクタに V ブランク割り込み時の処理を記載します。

V ブランクの割り込みベクタアドレスは 0x0040 です。そのため、V ブランクが発生した時、割り込みが有効であれば 0x0040 へジャンプしてきます。

ただ、今回はここには割り込みからリターンする命令だけを書きます。そして、main() の halt 無限ループの箇所で、halt と相対ジャンプの間に「キー入力確認&スクロール更新」の処理を追加します。

なぜかというと、割り込みが発生後、割り込みからリターンするまでの間、その他の全ての割り込みが無効化されるため、一般的に割り込みからはなるべく早くリターンすべきで、今回の場合、必ずしも割り込み期間中に実施しなければならない処理では無いからです。

そこで利用するのが halt 命令の振る舞いです。halt 命令は、CPU の実行がこの命令に到達すると CPU を休ませますが、割り込みが発生すると halt 命令の次の命令へ CPU の実行を進めます。そのため、halt 命令の後ろに処理を実装しておけば、割り込み契機で実行されるが、その他の割り込みは止めていない状態で実行できます。

そんな訳で、ここでは V ブランク割り込みのベクタアドレスの位置には割り込みからリターンする命令だけを書きます。実装するとリスト 3.1 の通りです。

▼リスト 3.1: 03_joypad/01_vblank_vec.sh

```
# ・・・省略・・・

main() {
        # ・・・省略・・・
}

# ・・・変更(ここから)・・・
# 割り込みベクタ生成
## 0x0000 - 0x003F(64バイト)は0x00(nop)で埋める
dd if=/dev/zero bs=1 count=64
## 0x0040(Vブランク)に割り込みからリターンする命令(reti)(1バイト)を配置
lr35902_ei_and_ret
## 0x0041 - 0x00FF(191バイト)は0x00(nop)で埋める
dd if=/dev/zero bs=1 count=191
# ・・・変更(ここまで)・・・

# ・・・省略・・・
```

`lr35902_ei_and_ret` が、割り込みからリターンする命令 (1 バイト) を出力するシェル関数です。0x0000 から 0x00FF の割り込みベクタ領域の中で 0x0040 のみにこ

の命令を配置し、その他の領域は全て 0 で埋めるようにしています。

♣ V ブランク割り込みを有効化する

個別に割り込みを有効化する割り込みイネーブル (IE) レジスタで V ブランク割り込みを有効化します。

IE レジスタの各ビットの役割は表 3.2 の通りです。

▼表 3.2: IE レジスタについて

ビット	役割
7 - 5	使用しない
4	ボタン入力割り込み (1=有効)
3	シリアル割り込み (1=有効)
2	タイマー割り込み (1=有効)
1	LCD ステータス割り込み (1=有効)
0	V ブランク割り込み (1=有効)

シリアル割り込みは、通信ケーブルでのシリアル通信に関する割り込みです。LCD ステータス割り込みは、予め設定した LCD 状態になったら割り込みを発生させてくれるものです。

V ブランク割り込みはビット 0 です。1 をセットすると有効になります。

実装するとリスト 3.2 の通りです。

▼リスト 3.2: 03_joypad/02_vblank_en.sh

```
# ・・・省略・・・

main() {
        # ・・・省略・・・

        # BGP設定
        lr35902_set_reg regA $BGP_VAL
        lr35902_copy_to_ioport_from_regA $GB_IO_BGP

        # ・・・追加(ここから)・・・
        # Vブランク割り込み有効化
        lr35902_copy_to_regA_from_ioport $GB_IO_IE
        lr35902_set_bitN_of_reg 0 regA
        lr35902_copy_to_ioport_from_regA $GB_IO_IE
        # ・・・追加(ここまで)・・・

        # 無限halt
```

```
        infinite_halt
}

# ・・・省略・・・
```

`lr35902_copy_to_regA_from_ioport $GB_IO_IE` でIEレジスタの値をレジスタAに取得し、`lr35902_set_bitN_of_reg 0 regA` でレジスタAのビット0に1をセットした後、逆に書き戻しています。

♣ 割り込み機能自体を有効化

今は割り込み機能自体を無効化しているので、割り込みを有効化します。

割り込み機能の有効化の命令は `lr35902_enable_interrupts` という名前でシェル関数化しています。

haltに入る直前に有効化するようにします(リスト3.3)。

▼リスト3.3: 03_joypad/03_ei.sh

```
# ・・・省略・・・

main() {
        # ・・・省略・・・

        # Vブランク割り込み有効化
        lr35902_copy_to_regA_from_ioport $GB_IO_IE
        lr35902_set_bitN_of_reg 0 regA
        lr35902_copy_to_ioport_from_regA $GB_IO_IE

        # ・・・追加(ここから)・・・
        # 割り込み有効化
        lr35902_enable_interrupts
        # ・・・追加(ここまで)・・・

        # 無限halt
        infinite_halt
}

# ・・・省略・・・
```

これで、Vブランクに入ると割り込みによりhaltを抜けるようになりました。

3.4　十字キーに応じてスクロールする処理を実装する

それでは、本題の処理を実装していきます。

♣ 十字キー入力取得処理

まずは十字キーの入力取得です。キーの状態は先述の通り、ジョイパッドのレジスタから取得できます。

さっそく実装を紹介するとリスト 3.4 の通りです。

▼リスト 3.4: 03_joypad/04_joyp.sh

```
# ・・・省略・・・

# ・・・追加(ここから)・・・
manual_scroll() {
        # ジョイパッド入力取得(十字キー)
        ## 十字キーの入力を取得するように設定
        lr35902_copy_to_regA_from_ioport $GB_IO_JOYP
        lr35902_set_bitN_of_reg 5 regA
        lr35902_res_bitN_of_reg 4 regA
        lr35902_copy_to_ioport_from_regA $GB_IO_JOYP
        ## 入力取得(ノイズ除去のため2回読む)
        lr35902_copy_to_regA_from_ioport $GB_IO_JOYP
        lr35902_copy_to_regA_from_ioport $GB_IO_JOYP
        ## ビット反転(押下中のキーのビットが1になる)
        lr35902_complement_regA
        ## レジスタBへ格納
        lr35902_copy_to_from regB regA
}
# ・・・追加(ここまで)・・・

main() {
        # ・・・省略・・・

        # Vブランク割り込み有効化
        lr35902_copy_to_regA_from_ioport $GB_IO_IE
        lr35902_set_bitN_of_reg 0 regA
        lr35902_copy_to_ioport_from_regA $GB_IO_IE

        # 割り込み有効化
        lr35902_enable_interrupts

        # ・・・変更(ここから)・・・
        (
                # 割り込みがあるまでhalt
```

```
                lr35902_halt

                # 手動画面スクロール
                manual_scroll
        ) >main.4.o
        cat main.4.o
        local sz_4=$(stat -c '%s' main.4.o)
        lr35902_rel_jump $(two_comp_d $((sz_4+2)))
        # ・・・変更(ここまで)・・・
}

# ・・・省略・・・
```

十字キーの入力を取得してそれに応じてスクロールレジスタを更新する処理は、`manual_scroll` というシェル関数へ分けました。

`main` 関数では、「無限 halt」していた箇所を、halt 命令と相対ジャンプ命令へ分け、その間に manual_scroll を入れました。halt と manual_scroll をサブシェルに入れて一旦ファイル出力し、そのサイズを相対ジャンプ命令に使っています。(manual_scroll 後の相対ジャンプで halt 命令の位置まで戻るようにしています。)

manual_scroll には、ここではまず、十字キー入力の取得処理を実装しました。JOYP レジスタのレジスタ値をレジスタ A へ取得し、ビット 5 に 1 を設定 (ボタン側を非選択)、ビット 4 に 0 を設定 (十字キー側を選択) し、レジスタ A を JOYP レジスタへ書き戻すことで設定を反映させています。

その後、JOYP レジスタを入力状態確認のために読むのですが、設定が反映されるまでの時間なのか直後だと正しく読めないため、2 回読むようにしています[*2]。

JOYP レジスタでは、押下状態のキーに対するビットが 0 になるので、今後の処理で扱いやすいように `lr35902_complement_regA` でビット反転しています。

そして、レジスタ A は別の処理でも使うのでレジスタ B へ値をコピーして、十字キー入力取得処理は完了です。

♣ スクロール処理

取得した十字キー状態に応じてスクロールレジスタを更新する処理を追加します。

ここでも実装で説明するため、さっそくコードを紹介します (リスト 3.5)。

[*2] 本書末尾「参考にさせてもらった情報」の「GameBoy CPU Manual」の「1. FF00 (P1)」(P.35) を参考にさせてもらいました。

▼リスト 3.5: 03_joypad/05_joyp_scrl.sh

```
# ・・・省略・・・

manual_scroll() {
        # ・・・省略・・・
        ## レジスタBへ格納
        lr35902_copy_to_from regB regA

        # ・・・追加(ここから)・・・
        # 十字キー押下状態に応じてスクロールレジスタ更新
        ## 下キーチェック
        lr35902_test_bitN_of_reg 3 regB
        (
                # 下キーが押下中の場合

                # SCYをインクリメント
                lr35902_copy_to_regA_from_ioport $GB_IO_SCY
                lr35902_inc regA
                lr35902_copy_to_ioport_from_regA $GB_IO_SCY
        ) >manual_scroll.1.o
        local sz_1=$(stat -c '%s' manual_scroll.1.o)
        ## 下キーが押下中でない場合、押下中の処理を飛ばす
        lr35902_rel_jump_with_cond Z $(two_digits_d $sz_1)
        ## 押下中の処理
        cat manual_scroll.1.o

        ## 上キーチェック
        lr35902_test_bitN_of_reg 2 regB
        (
                # 上キーが押下中の場合

                # SCYをデクリメント
                lr35902_copy_to_regA_from_ioport $GB_IO_SCY
                lr35902_dec regA
                lr35902_copy_to_ioport_from_regA $GB_IO_SCY
        ) >manual_scroll.2.o
        local sz_2=$(stat -c '%s' manual_scroll.2.o)
        ## 上キーが押下中でない場合、押下中の処理を飛ばす
        lr35902_rel_jump_with_cond Z $(two_digits_d $sz_2)
        ## 押下中の処理
        cat manual_scroll.2.o

        ## 左キーチェック
        lr35902_test_bitN_of_reg 1 regB
        (
                # 左キーが押下中の場合
```

```
                # SCXをデクリメント
                lr35902_copy_to_regA_from_ioport $GB_IO_SCX
                lr35902_dec regA
                lr35902_copy_to_ioport_from_regA $GB_IO_SCX
        ) >manual_scroll.3.o
        local sz_3=$(stat -c '%s' manual_scroll.3.o)
        ## 左キーが押下中でない場合、押下中の処理を飛ばす
        lr35902_rel_jump_with_cond Z $(two_digits_d $sz_3)
        ## 押下中の処理
        cat manual_scroll.3.o

        ## 右キーチェック
        lr35902_test_bitN_of_reg 0 regB
        (
                # 右キーが押下中の場合

                # SCXをインクリメント
                lr35902_copy_to_regA_from_ioport $GB_IO_SCX
                lr35902_inc regA
                lr35902_copy_to_ioport_from_regA $GB_IO_SCX
        ) >manual_scroll.4.o
        local sz_4=$(stat -c '%s' manual_scroll.4.o)
        ## 右キーが押下中でない場合、押下中の処理を飛ばす
        lr35902_rel_jump_with_cond Z $(two_digits_d $sz_4)
        ## 押下中の処理
        cat manual_scroll.4.o
        # ・・・追加(ここまで)・・・
}

# ・・・省略・・・
```

追加した行数は多いですが、同じような処理を 4 つ (上下左右) 実施しているだけです。

例えば下キーについて、まず、キー状態を格納したレジスタ B で、ビット 3(下キー) がセットされているか否かをチェックします。

押下中である場合に実施する処理をサブシェル内に入れて一旦ファイルに保存し、そのファイルサイズを相対ジャンプ命令に使っています。(キーが押下中でない場合はそのサイズ分だけ処理をスキップするようにしています。)

押下中の処理としては、下キーに対しては SCY をインクリメント、上キーに対しては SCY をデクリメント、左キーに対しては SCX をデクリメント、右キーに対しては SCX をインクリメントしています。

♣ 実行結果

実行結果の見た目は前章の最後と変わらないため画像は省略します。
十字キーでの上下左右の入力に応じて画面がスクロールすることが確認できます。

おわりに

ここまで本書をお読みいただきありがとうございました！

C98 が中止となり電子/紙問わず本を書くことは中断していたのですが、エアコミケという形で実施されることを知り、急遽、今やっていることを本にしました。そのため、お見苦しいところや誤記など、残っているかもしれません (すみません)。

ただ、本書を通して低レイヤーの面白さ、エミュレータではあるけどハードウェアを直接操作する感動を少しでも感じてもらえれば幸いです。なお、本書で生成した ROM イメージは、もちろん実機でも動きます。本書では紹介しきれませんでしたが興味があればぜひ調べて試してみてください。

本書では、アセンブリ言語さえ使わずに ROM ファイルを生成してきました。結局のところ、欲しいのはバイナリで、バイナリさえ得られればその手段は「ソースコードから変換する」だけでなくても良いんだ、という低レイヤーの自由さ (?) みたいなところも伝わったら嬉しいです。

そして、8 ビットというレトロなコンピュータは、現代の 64 ビットに比べて「マシンを直接動かす機械語バイナリを手で書く」事が楽です。8 ビットというコンパクトさでありながら、画面描画やユーザ入力、サウンド、そして通信 (通信ケーブル) まであるゲームボーイは、色々いじり回すのには面白いコンピュータなんじゃないかなと、最近は思っています。

参考にさせてもらった情報

GB のアーキテクチャや CPU について

- Everything You Always Wanted To Know About GAMEBOY
  - http://bgb.bircd.org/pandocs.htm
  - BGB が公開しているゲームボーイの仕様まとめ
  - 網羅的にまとめてありとても参考になります
- GameBoy CPU Manual
  - http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf
  - ゲームボーイの CPU(LR35902) の仕様について PDF にまとめられています
  - 各 CPU 命令の動作などはこちらが分かりやすいです
- Gameboy CPU (LR35902) instruction set
  - https://www.pastraiser.com/cpu/gameboy/gameboy_opcodes.html
  - ゲームボーイの CPU である LR35902 の全命令の機械語コードが表の形で見やすくまとめられています
- GameBoy Memory Map
  - http://gameboy.mongenel.com/dmg/asmmemmap.html
  - ゲームボーイのメモリマップはこちらが見やすいです
- Gameboy 2BPP Graphics Format
  - https://www.huderlem.com/demos/gameboy2bpp.html
  - GB のピクセルフォーマットに関して説明しているページです
  - 16 バイト分の 16 進数を入力すると、どのような表示になるかを見せてくれるフォームもあります

bgb について

- winetricks による Wine の文字化け (アルファベットが豆腐) を解消 - Symfoware
  - https://symfoware.blog.fc2.com/blog-entry-2228.html
  - Wine で BGB を起動した際にアルファベットが豆腐になってしまった問題をこちらの記事を参考に解決しました

シェルスクリプトでゲームボーイプログラミング入門

2020 年 5 月 5 日　ver 1.0

著　者　大神祐真
印刷所　日光企画

(powered by Re:VIEW Starter)