セガサターン
タイル描画の
うすい本

［著］大神祐真

技術書典 17 新刊
2024 年 11 月 2 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

本書を手に取っていただき、ありがとうございます！

本書では、セガサターンに搭載された画面描画用の IC が持つ「タイル」という描画方式について、エミュレータで動作する小さなサンプルを参考にしつつ、エミュレータのソースコード[*1]を読んで理解します。

セガサターンが持つ画面描画機能は「タイル」という方式一つをとっても高機能ゆえに複雑です。本書ではこの方式について、全貌をというよりは、基本的な動作ケースにおいて IC がどのような流れで画面上にピクセルを描画するのかを解説します。

本書の性質上、ソースコードと文章が多くなってしまっていますが[*2]、エミュレータのコードや、そこからわかるセガサターンの動きについて楽しんでいただければ幸いです。

本書の更新情報等について

本書を含め、当サークルの同人誌・同人作品の情報は下記の筆者ウェブページにまとめています。

- `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。何かおかしな点があった場合等は、まずこちらのページをご覧ください。

[*1] エミュレータのコードは C 言語です。

[*2] 一部は図で説明していたりもしますが。

目次

第 1 章
VDP2 の紹介とエミュレータ上でサンプルを試す

この章では、本書で解説するタイル描画の機能も持つ「VDP2」という IC を簡単に紹介します。そして OSS のセガサターンエミュレータ「Yabause」とゲームエンジン「Jo Engine」を用いて VDP2 を使用するサンプルを試し、次章以降に向けて改造してみます。

1.1 VDP2 について

「VDP」とは「Video Display Processor」の略で、画面描画周りを担う、ざっくりとは GPU 的な IC です。セガサターンにはそれが 2 つ搭載されていて、それぞれ「VDP1」・「VDP2」と呼びます。「VDP1」は「スプライト」と呼ばれる描画機能を持つ IC です。主に画面上で動くキャラを描画するのに使われます。対して「VDP2」は主に背景を描画するのに使われる IC で、「ビットマップ」の形式や本書で解説する「タイル」の形式で描画する機能を持ちます。[*1]

1.2 実行環境について

VDP2 やそのタイル描画がどのようなものかを知るために、また、次章以降でコードを読む際の参考にもするために、これからセガサターンのエミュレータと、その上で動作させるサンプルを作るためのゲームエンジンを用意します。

筆者の環境が Linux 環境（Debian[*2]）である都合上、Debian 系の Linux 環境について紹介し

[*1] タイル描画機能については解説していませんが、セガサターンを構成する IC について全般的には、既刊「エミュレータのコードを読んでわかるセガサターン」で解説しています。もしご興味あれば「はじめに」に記載の筆者ウェブサイトまでお越しください！

[*2] Debian GNU/Linux 11 (bullseye) を使っています。

ます。

なお、この後紹介するエミュレータとゲームエンジンはいずれも Windows でも動作するものであり、Windows についても補足しますが、未検証である点はご容赦ください。[*3]

macOS については、後述しますが、エミュレータは対応しているようなのですが、ゲームエンジンの方がサポート外なようです。

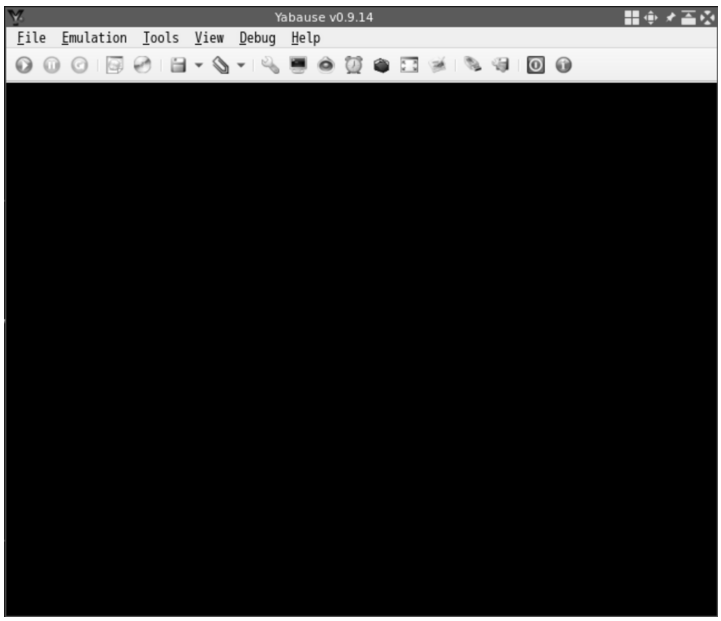
1.3　エミュレータ「Yabause」のインストール

この節ではまず OSS のセガサターンエミュレータ「Yabause[*4]」をインストールします。

Debian 系 Linux 環境の場合、`apt` コマンドでインストールできます。

```
$ sudo apt install yabause
```

インストールが完了すると、`yabause` コマンドでエミュレータを起動できます（図 1.1）。



▲図 1.1: Yabause の起動後の画面

なお、Windows に関しては、次節以降で紹介するゲームエンジンの中に Yabause も入っているので、ここでインストールする必要はありません。もし必要であれば、Windows や macOS については、公式サイトのダウンロードページ（下記）にビルド済みバイナリが公開されています。

[*3] いずれもむしろ主に Windows での動作を想定しているもので、ドキュメントも主に Windows 向けに書かれていたりしますので、Windows であればむしろ問題ないかと思います。

[*4] `https://yabause.org/`

- Yabause >> Download
  - `https://yabause.org/download/`

1.4　ゲームエンジン「Jo Engine」の取得

「Jo Engine[*5]」は C 言語で書かれたセガサターン用の 2D/3D ゲームエンジンです。

Linux/Windows に関しては、ライブラリ本体に加えてそれぞれの OS 用のビルド済みのコンパイラやサンプルコードなども含んだ一式が以下で公開されています。

- Jo Sega Saturn Engine, Download Jo Engine with samples
  - `https://www.jo-engine.org/download/`

上記のページには zip アーカイブへのリンクと GitHub へのリンクがあります。どちらから取得しても構いません。GitHub リポジトリの方にもビルド済みのコンパイラは入っていますし、そもそも zip アーカイブは単に GitHub の master ブランチを zip アーカイブにしているだけのようです。zip アーカイブの方で取得した場合は展開しておいてください。なお、本書の内容は GitHub リポジトリを clone し、以下のコミット時点で作業したものです。

- 7e13996 Merge pull request #78 from ReyeMe/master

補足として、ビルド済みのコンパイラは取得した Jo Engine のディレクトリ直下の Compiler ディレクトリ以下に OS 別にディレクトリを切って置いてあります。そこには MACOS というディレクトリもあるのですが、そこに置いてあるのは TODO.txt というテキストファイルだけで、まだ macOS 向けのコンパイラは用意されていません。

1.5　printf サンプルをビルド・実行

♣ ビルド

ビルドに必要なツールも基本的に先程取得した一式の中に入っていますが、Linux 版の make が入っていないので、Linux を使用の際は予め make をインストールしておいてください。Windows に関しては Windows 版の make が入っているので別途インストールする必要はありません。

それでは、試しにサンプルを一つビルドし実行してみましょう。ここでは、Jo Engine のライブラリ内の `printf` 関数（ `jo_printf` 関数）のデモである Samples ディレクトリ内の demo - printf を試してみることにします。ターミナル上でこのディレクトリへ移動し、用意されているシェルスクリプトを以下のように実行するとビルドできます。

[*5] `https://www.jo-engine.org/`

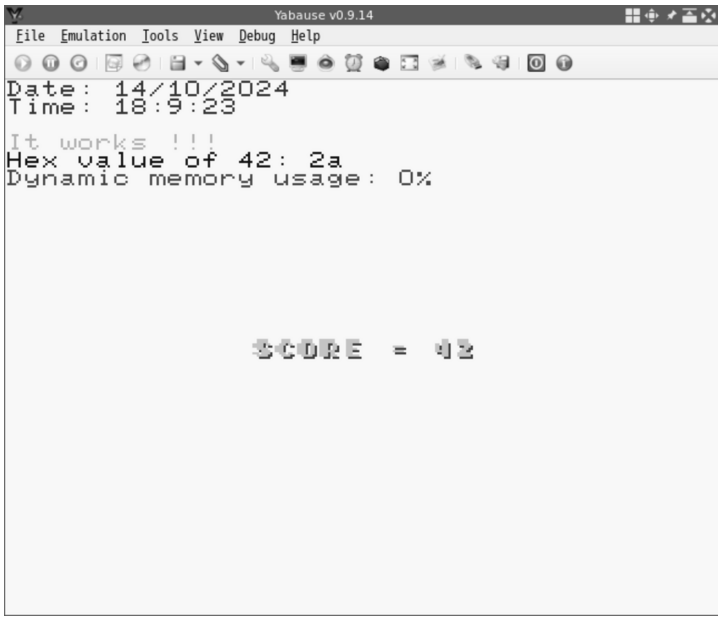
```
$ ./compile.sh
 ・・・省略（ビルドログ）・・・
$ ls game.iso game.cue
game.cue  game.iso      ← CDイメージができあがる
```

セガサターンのゲームメディアは CD なので、ビルドすると CD イメージができあがります。

なお、Windows の場合は同じディレクトリ内に compile.bat というバッチファイルがありますので、これを実行してください。

♣ 実行

実行についてもシェルスクリプトが用意されています。コンパイルした時と同じディレクトリ内で run_with_yabause.sh というシェルスクリプトを実行すると、Yabause が起動し、`printf` のデモプログラムが動作します（図 1.2）。



▲図 1.2: Yabause 上で printf のデモプログラムが動作する様子

なお、Windows の場合は同じディレクトリ内にある run_with_yabause.bat というバッチファイルを実行してください。

♣ うまく行かなかった場合は

Jo Engine の取得からサンプルの実行までについては、Jo Engine 公式のチュートリアルで説明されています。もし何かうまく行かなければこちらを参考にしてみてください。

- Jo Sega Saturn Engine, Jo Engine tutorial
  - `https://www.jo-engine.org/tutorial/`

1.6 'A' を表示するだけのプログラムへ変更

♣ main.c を変更

先程の `printf` のデモの「SCORE = 42」以外の文字は VDP2 のタイルの機能で描画しています[*6]。ただ、次章以降で参考にするにあたり極力シンプルにしたいので、VDP2 のタイル描画を用いる最小のサンプルとして、'A' を表示するだけのプログラムへ変更します。demo - printf ディレクトリ内の main.c が本体で、この中の `my_draw` 関数をリスト 1.1 のように変更します。

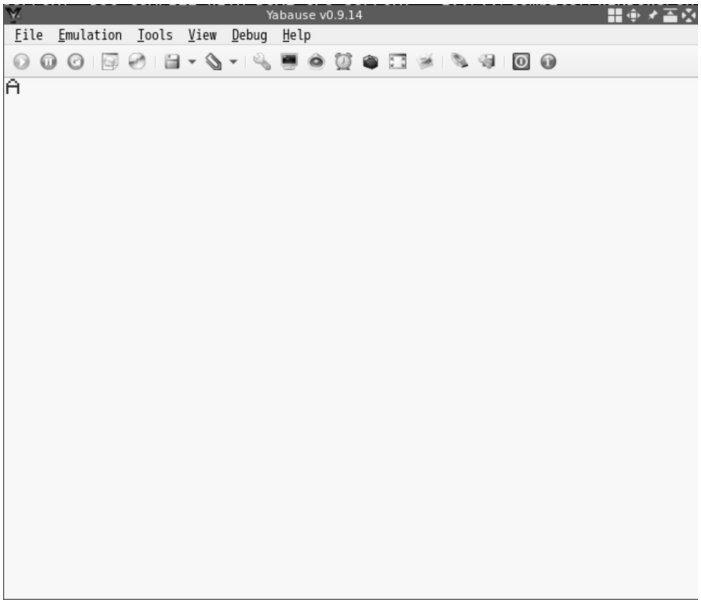
▼リスト 1.1: 'A' のみを描画するように変更した my_draw 関数

```
void                        my_draw(void)
{
    /* See jo/colors.h to know palette color indexes */
    jo_set_printf_color_index(JO_COLOR_INDEX_Red);
    jo_printf(0, 0, "A");
}
```

`my_draw` 関数は、直後の `jo_main` 関数内にて、周期的な描画処理で呼び出されるように登録している関数です。このように変更することで、'A' だけを画面に表示できます。

♣ ビルド・実行

ビルドし実行すると図 1.3 のように画面に'A' だけを表示します。



▲図 1.3: 'A' だけを表示 ※ グレースケールのため文字色が黒に見えるが実際は赤

[*6] 「SCORE = 42」の部分は VDP1 のスプライトで描画しています。

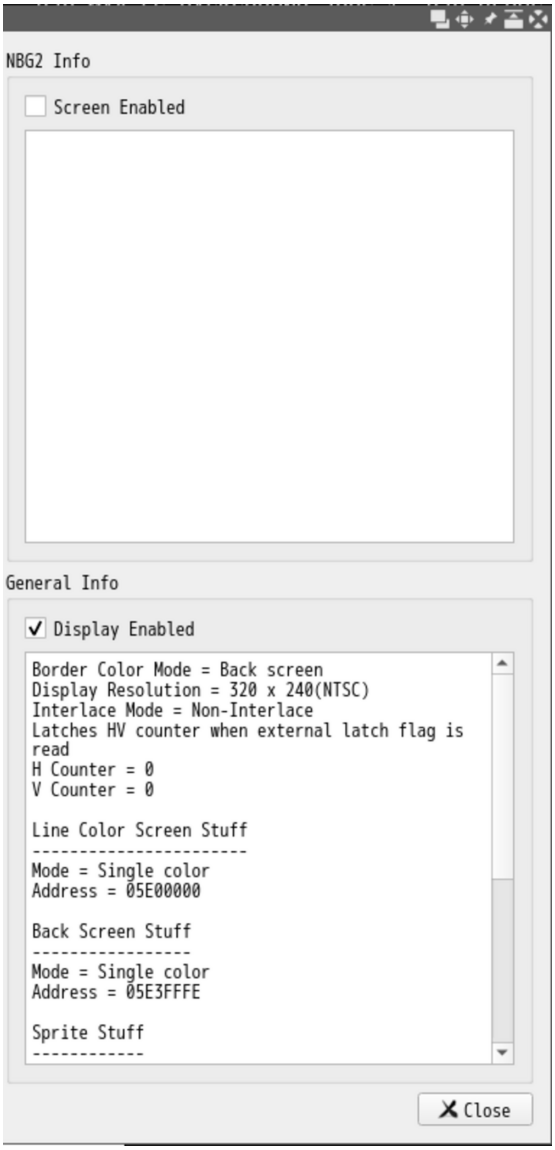
1.7　デバッグ画面を眺める

Yabause にはデバッグ機能があり、そこに表示されるものを見るだけでもわかることがあります。ここでは少しデバッグ画面を眺めてみましょう。先程作成したサンプルを Yabause で実行し、メニューバーの [Debug] を選択し、その中の [VDP2] を選択すると VDP2 のデバッグ画面が開きます。画面の様子は図 1.4 と図 1.5 の通りです。

▲図 1.4: VDP2 のデバッグ画面（左側と真ん中）

デバッグ画面には 6 つのテキスト欄があります。各欄の上部に「NBG0/RBG1 Info」、「NBG1

Info」等と書かれています。「NBG<数字>」や「RBG<数字>」はそれぞれレイヤーです。「NBG」というのは通常の背景レイヤーで縦横のスクロール等が行え、「NBG0」から「NBG3」までの 4 枚あります。「RBG」は回転が使える特殊なレイヤーで、「RBG0」と「RBG1」の 2 枚あります。なお、「NBG0」と「RBG1」が同じ欄なのはこれらは同時に使用できないということで、この欄の 1 行目に「NBG0 mode」とあるので現在は NBG0 が動作しています。また、NBG0 の方には 3 行目に「Tile」とあるのに対し、NBG1 の方には 2 行目に「Bitmap」とあります。NBG0 はタイルのモード、NBG1 はビットマップのモードである事がわかります。



▲図 1.5: VDP2 のデバッグ画面（右側）

「General Info」という欄は特定のレイヤーではなく、VDP2 全般の設定が表示されています。

例えば「Display Resolution = 320 x 240(NTSC)」から画面解像度の設定がわかります。

ちなみに、各レイヤーの表示/非表示はメニューバーの [View] - [Layer] に並ぶ各レイヤーのチェックボックスで行えます。試しにこれで各レイヤーの表示/非表示を切り替えてみると、'A' という文字は NBG0 にあることが分かります。

「Screen Enabled」というチェックボックスについて

VDP2 デバッグ画面のそれぞれの欄の上部に「Screen Enabled」というチェックボックスがあり、このチェックを付けたり外したりできますが、このチェックを付け外ししても表示上は何も変わりません。恐らくここは現在の各レイヤーの表示設定を出しているだけで、このチェックボックスから各レイヤーの表示/非表示を切り替えられるものでは無いのかと思います。

まとめると、このサンプルの'A' は NBG0 のレイヤーにあって、このレイヤーはタイルのモードで動作しています。期待通り、タイル描画において最小の動作ができていそうです。なお、NBG0 の欄には以下のように「プレーンアドレス」と呼ばれるアドレスも表示されていました。次章ではこれをキーワードにタイル描画の実装を見ていきます。

▼次章のキーワード

```
Plane A Address = 00076000
Plane B Address = 00076000
Plane C Address = 00076000
Plane D Address = 00076000
```

第 2 章

1 ピクセルずつ描画処理を行っている所まで

この章では、まず、Yabause のソースコードの在り処を紹介し、前章の最後にデバッグ画面で確認した「プレーンアドレス」をきっかけにソースコードを見始め、画面を 1 ピクセルずつ描画しているコードを見つける所までを解説します。

2.1 Yabause のソースコードの在り処

Yabause のソースコードは以下の GitHub リポジトリで公開されています。

- GitHub - Yabause/yabause: Yabause is a Sega Saturn emulator.
  - `https://github.com/Yabause/yabause`

`git clone` で取得しても、ブラウザで眺めても構いません。なお、本書では以下のコミット時点を対象とします。

- 7e38821d Fixed gtk port compilation

2.2 デバッグ画面の実装をきっかけにコードを見始める

それでは、'A' の画面表示をセガサターンのハード的にはどのように行っているのかを Yabause のコードで見ていきましょう。コードを読み始める取っ掛かりを考えるにあたり、一般的な話として、画面表示を行う IC は RAM の決まったアドレスに決まったバイナリ形式で配置されたデータを読み、それに従って画面表示を行います。「画面出力を行う IC」は今回の場合 VDP2 です。VDP2 にも画面表示を行うためにデータを読みに行くアドレスがあるはずで、ちょうど VDP2 の

デバッグ画面にも「プレーンアドレス」というものがありました。まずは、このプレーンアドレスを VDP2 がどのように扱うのかを見てみることにしましょう。

VDP2 のデバッグ画面ではプレーンアドレスを `Plane <A/B/C/D> Address = <アドレス>` といった形式で表示していました。デバッグ表示の際にどうやってプレーンアドレスを表示していたのかが分かれば今後コードを読んでいく助けになりそうです。 `Plane .* Address` といった正規表現等で Yabause のソースコード内を探すと、Yabause のリポジトリ直下から見て yabause/src/の vdp2debug.c 内にリスト 2.1 の処理を見つけることができます。なお、Yabause の動きを確認する上で参照するようなソースファイルは基本的に全て yabause/src/にあります。以降ではソースファイルを示す際は yabause/src/というパスは省略して、単に「vidsoft.c」や「vidshared.c」といった形でファイル名だけで記載します。

▼リスト 2.1: vdp2debug.c: AddMapInfo 関数後半部

```
// Map Planes A-D
for (i = 0; i < numplanes; i++)
{
   tmp = mapoffset | map[i];

   if (patterndatasize == 1)
   {
      if (patternwh == 1)
         addr = ((tmp & 0x3F) >> deca) * (multi * 0x2000);
      else
         addr = (tmp >> deca) * (multi * 0x800);
   }
   else
   {
      if (patternwh == 1)
         addr = ((tmp & 0x1F) >> deca) * (multi * 0x4000);
      else
         addr = ((tmp & 0x7F) >> deca) * (multi * 0x1000);
   }

   AddString(outstring, "Plane %C Address = %08X\r\n",
             0x41+i, (unsigned int)addr);
}
```

Yabause の画面表示周りのソースファイルについて

Yabause で画面表示に関する実装は主に vidsoft.h と vidsoft.c、vidshared.h と vidshared.c にあります。VDP2 固有の実装は vdp2.h と vdp2.c にあります。[*1]

`AddString` という関数で `Plane <1 文字> Address = <8 桁の 16 進数>` という文字列を作

[*1] Yabause のソースコードのディレクトリ構造やファイルについてもより詳しくは既刊「エミュレータのコードを読んでわかるセガサターン」で紹介しています。

っています。そして、 <8 桁の 16 進数> (%08X) に対応する引数として addr を渡していることから、プレーンアドレスとして表示している値は addr に入っていることが分かります。

リスト 2.1 の全体を見てみると、まず、 numplanes の回数だけ実行されるループになっています。for 文のすぐ上のコメントにもある通り、これはプレーン A から D までの 4 回分ループします。そして、for のコードブロック内を見ると、 patterndatasize と patternwh がそれぞれ 1 であるか否かに応じて合計 4 つの分岐があり、それぞれで少し異なった計算式でアドレスを算出し addr へ代入しています。

実際に使われる計算式がどれであるかはともかく、プレーンアドレスを算出する計算式を見つけることができました。試しに 4 つある内の一番上の ((tmp & 0x3F) >> deca) * (multi * 0x2000) [*2]をキーワードにソースコード内を検索してみると、vidshared.h の CalcPlaneAddr 関数内に同じ計算式があります (リスト 2.2)。

▼リスト 2.2: vidshared.h: CalcPlaneAddr 関数（コメントアウトされている処理を削除済み）

```
static INLINE void CalcPlaneAddr(vdp2draw_struct *info, u32 tmp)
{
   int deca = info->planeh + info->planew - 2;
   int multi = info->planeh * info->planew;

   if (info->patterndatasize == 1)
   {
      if (info->patternwh == 1)
         info->addr = ((tmp & 0x3F) >> deca) * (multi * 0x2000);
      else
         info->addr = (tmp >> deca) * (multi * 0x800);
   }
   else
   {
      if (info->patternwh == 1)
         info->addr = ((tmp & 0x1F) >> deca) * (multi * 0x4000);
      else
         info->addr = ((tmp & 0x7F) >> deca) * (multi * 0x1000);
   }
}
```

これで、描画するまでの流れの中で、プレーンアドレスは CalcPlaneAddr という関数内で、第 1 引数で与えた vdp2draw_struct 構造体のポインタ変数 info が指す addr というメンバー変数へ設定される事が分かります。

そして、この関数を呼び出している所を探すと、vidshared.c 内で Vdp2NBG0PlaneAddr や Vdp2NBG1PlaneAddr 等の関数から呼ばれていることが分かります。今回は NBG0 で描画を行っているので Vdp2NBG0PlaneAddr 関数を見てみます（リスト 2.3)。

[*2] 実はこれが実際に使用される計算式です。

▼リスト 2.3: vidshared.c: Vdp2NBG0PlaneAddr 関数

```
void FASTCALL Vdp2NBG0PlaneAddr(vdp2draw_struct *info, int i, Vdp2* regs)
{
    ・・・省略・・・

   CalcPlaneAddr(info, tmp);
}
```

Vdp2NBG0PlaneAddr という正にプレーンアドレスを算出していそうな名前の関数の最後で CalcPlaneAddr 関数が呼び出されており、その際に第 1 引数に与えている info というポインタ変数は Vdp2NBG0PlaneAddr の第 1 引数でもあるため、算出し info->addr に設定したプレーンアドレスはこの呼び出し元で使っていると思われます。

Vdp2NBG0PlaneAddr 関数の呼び出し元を探すと、vidsoft.c の Vdp2DrawNBG0 関数内で使用しています（リスト 2.4)。

▼リスト 2.4: vidsoft.c: Vdp2DrawNBG0 関数前半部（処理は色々と省略）

```
static void Vdp2DrawNBG0(Vdp2* lines, Vdp2* regs, u8* ram, u8* color_ram, struct ›
›CellScrollData * cell_data)
{
   vdp2draw_struct info = { 0 };
    ・・・省略・・・

   if (regs->BGON & 0x20)
   {
      // RBG1 mode
       ・・・省略・・・
   }
   else if (regs->BGON & 0x1)
   {
      // NBG0 mode
       ・・・省略・・・
      info.PlaneAddr = (void FASTCALL(*)(void *, int, Vdp2*))&Vdp2NBG0PlaneAddr;
   }

    ・・・省略・・・
```

リスト 2.4 から、NBG0 の場合、プレーンアドレス算出用の関数として Vdp2NBG0PlaneAddr を使うようにしていることが分かります。コードは省略しますが、関数ポインタ info.PlaneAddr は同じく vidsoft.c 内の LoadLineParamsNBG0 関数内で GeneratePlaneAddrTable 関数を呼び出す際に引数として渡されます。GeneratePlaneAddrTable 関数はリスト 2.5 の通りです。

▼リスト 2.5: vidsoft.c: GeneratePlaneAddrTable 関数

```
static INLINE void GeneratePlaneAddrTable(
   vdp2draw_struct *info, u32 *planetbl,
   void FASTCALL (* PlaneAddr)(void *, int, Vdp2* ), Vdp2* regs)
{
```

```
   int i;

   for (i = 0; i < (info->mapwh*info->mapwh); i++)
   {
      PlaneAddr(info, i, regs);
      planetbl[i] = info->addr;    ← planetblへプレーンアドレスを保存
   }
}
```

注目してもらいたいのは、 PlaneAddr 関数の実行後、 info->addr に入っているプレーンアドレスを planetbl 配列へ保存している部分です。 GeneratePlaneAddrTable 関数の実行時、 info->mapwh は 2 です。そのためこの for ループは 4 回実行されます。それはプレーン A から D に対応しており、ここではプレーン A から D のそれぞれのアドレスを planetbl 配列へ保存しています。この planetbl 配列が、後の描画処理で使用されます。

2.3　描画処理の実装箇所を見つける

GeneratePlaneAddrTable 関数を呼び出す LoadLineParamsNBG0 関数は、前節で見た Vdp2DrawNBG0 関数の末尾であるポインタ変数へ設定しています（リスト 2.6）。

▼リスト 2.6: vidsoft.c: Vdp2DrawNBG0 関数末尾（処理は色々と省略）

```
    ・・・省略・・・

   info.LoadLineParams =
      (void (*)(void *, void *,int ,Vdp2*)) LoadLineParamsNBG0;

   if (info.enable == 1)
   {
      // NBG0 draw
      Vdp2DrawScroll(&info, lines, regs, ram, color_ram, cell_data);
   }
    ・・・省略・・・
}
```

ここで、 Vdp2DrawNBG0 関数についてざっくり説明すると、この if ブロックより上の部分でレイヤー個別（この関数の場合 NBG0）の設定を行い、if ブロック内で Vdp2DrawScroll という縦横スクロールする NBG 系のレイヤー共通の描画処理を呼び出します。描画処理の中でプレーンアドレスがどのように使われるかを見たいので、 Vdp2DrawScroll 関数の実装を見てみましょう。 Vdp2DrawScroll 関数も vidsoft.c にあります（リスト 2.7）。

▼リスト 2.7: vidsoft.c: Vdp2DrawScroll 関数（処理は色々と省略）

```
static void FASTCALL Vdp2DrawScroll(
   vdp2draw_struct *info, Vdp2* lines, Vdp2* regs, u8* ram, u8* color_ram,
```

```
   struct CellScrollData * cell_data)
{
    ・・・省略・・・
   int output_y = 0;
    ・・・省略・・・

   for (j = start_line; j < vdp2height; j += line_increment)
   {
       ・・・省略・・・
      for (i = 0; i < vdp2width; i++)
      {
          ・・・省略・・・
         // Fetch Pixel, if it isn't transparent, continue
         if (!info->isbitmap)
         {
            // Tile
             ・・・省略・・・
            Vdp2MapCalcXY(info, &x, &y, &sinfo, regs, ram, bad_cycle);
               ←タイルアドレスとタイル内座標を算出
         }

         if (!bad_cycle)
         {
            charaddr = info->charaddr;
            paladdr = info->paladdr;
         }
          ・・・省略・・・

         if (!Vdp2FetchPixel(info, x, y, &color, &dot, ram, charaddr, paladdr,
                             color_ram))   ←タイル内座標のピクセルを取得
         {
            continue;
         }
          ・・・省略・・・

         {
             ・・・省略・・・
            TitanPutPixel(priority, i, output_y,
               info->PostPixelFetchCalc(info, COLSAT2YAB32(alpha, color)),
               info->linescreen, info);
               ← 1ピクセル描画
         }
      }
      output_y++;
   }
}
```

リスト 2.7 ではざっくり処理を省略しています。残っているのは関数の後半部分にある for の二重ループです。まず外側の for ループについて説明します。カウンタ変数 `j` の初期値である `start_line` は、今回の'A' を表示するサンプルにおいては 0 です。（以降、変数の値に言及する際、それは基本的に'A' を表示するサンプルに関するものであるとします。）そして、`vdp2height` は画面解像度の高さである 240、 `line_increment` は 1 です。すなわち、外側の

for ループは画面の高さ（240）の分だけ 1 ピクセルずつ処理するループであり、カウンタ変数 `j` は 0 から 239 の間となります。そして、内側の for ループも同様に、 `vdp2width` は画面解像度の幅である 320 で、カウンタ変数 `i` が 0 から 319 まで 1 ピクセルずつ処理するループになっています。すなわち、この二重ループとしては、画面の全ピクセルを 1 ピクセルずつ処理するループになっています。

そして、二重ループの内側で呼び出しているリスト 2.7 で太字にしている 3 つの関数もここで簡単に紹介しておきます。まず `Vdp2MapCalcXY` は、8x8 のピクセル[*3]（付近のコメントにもある通り、これを「タイル（Tile)」と呼びます）に関するアドレスと、タイル内で次に描画するピクセルの座標を算出する関数です。次に、 `Vdp2FetchPixel` は、タイルに関するアドレスとタイル内の座標から、該当のピクセルの色情報を取得する関数です。最後に、 `TitanPutPixel` は、取得した色情報に従い画面へ 1 ピクセルを描画する関数です。以降では、 `Vdp2MapCalcXY` 関数と `Vdp2FetchPixel` を通して VDP2 のタイル機能を解説します。 `TitanPutPixel` 関数内は PC の GUI アプリとしての話になりセガサターンとは関係ないため、こちらの関数については特に詳しく解説は行いません。

「ピクセル」と呼ぶか「ドット」と呼ぶか

リスト 2.7 で引用したコード部分だけでも変数名や関数名で「ピクセル（Pixel)」表記と「ドット（Dot)」表記が共に存在しています。これは何らかのルールで使い分けがされているのかもしれませんが、それがまだよく分かっていないため、本書の説明では基本的に「ピクセル」表記で統一します。

2.4　異なる意味を持つ 2 つの座標

これで、正に 1 ピクセルずつ処理している関数まで見つけられたため、この章ではここまででも良いのですが、ここでもう少し説明しておきたいことがあります。それは「描画する先の画面上の座標」と「描画する元のピクセルデータの座標」についてです。なお、予めコメントしておくと、今回の'A' を表示するサンプルにおいては、これらの座標は同じ値になります。なので、気にしなくても良いといえばそうなのですが、エミュレータの実装として興味深い部分なので解説します。

まず、「描画する先の画面上の座標」を説明します。これは具体的には `TitanPutPixel` 関数で指定する座標のことです。 `TitanPutPixel` 関数では描画する先である画面上の座標を第 2 引数（X 座標）と第 3 引数（Y 座標）で指定します。実際に `TitanPutPixel` 関数に指定されている引数を見てみると、第 2 引数には内側の for ループ（X 軸方向のループ）のカウンタ変数 `i` が指定されています。第 3 引数には外側の for ループ（Y 軸方向のループ）のカウンタ変数 `j` では

[*3] 少なくとも'A' を表示するサンプルの実行時は 8x8px のタイルサイズで動作しています。この関数内にはそれ以外のタイルサイズ用のものと思われる処理もありますが、詳しくはまだ分かっていません。

なく、`output_y`という変数が指定されています。これは、関数の冒頭の変数宣言時に 0 で初期化され、その後は外側の for ループの最後でインクリメントされる他は変更されない変数です。`start_line`が 0 ではない場合、カウンタ変数`j`が 0 始まりではなくなるため、このような変数を用意しているのかと思います。[*4]

次に、「描画する元のピクセルデータの座標」を説明します。これは具体的には`Vdp2MapCalcXY`関数で指定する座標のことです。`Vdp2MapCalcXY`は「タイルアドレスとタイル内座標を算出する関数」と説明しました。これは何を元に算出するのかというと、指定された座標を元に算出します。ここで、例えば座標（X, Y）＝（`i`，`output_y`）を指定したとすると、これは`TitanPutPixel`関数で描画する先の画面上の座標でもあるので、座標（`i`，`output_y`）のピクセルデータを座標（`i`，`output_y`）に描画することになります。今度は座標（`i + 5`，`output_y`）を`Vdp2MapCalcXY`関数へ指定し、`TitanPutPixel`関数には変わらず座標（`i`，`output_y`）を指定したとします。これは座標（`i + 5`，`output_y`）のピクセルデータを座標（`i`，`output_y`）へ描画することになります。画面左上が原点なので、元よりも 5 ピクセル左側にずれて描画されることになります。正にこれは画面スクロールで、Yabause ではこのようにしてセガサターンの VDP2 の画面スクロールを実装しています。実はリスト 2.7 では省略していましたが、外側の Y 軸方向のループの中に`y += scroll_value;`という処理が、内側の X 軸方向の for ループの中に`x += linescrollx;`という処理がありました。これらがスクロールを行っている部分です。省略している行では他にも画面効果の処理を行っており、それらを処理した結果の X 座標と Y 座標がそれぞれ`x`と`y`に入っています。それらを`Vdp2MapCalcXY`関数の引数に指定し、前処理をした後の X 座標・Y 座標を元にタイルアドレスとタイル内座標を算出しています。

なお、今回のサンプルではスクロールなどの画面効果を使っていないため、前述の通り、「描画する先の画面上の座標」と「描画する元のピクセルデータの座標」は同じ値になります。

[*4] `j`から`start_line`を引けばそれで良さそうな気もします。`output_y`を用意している理由について、「0 始まりのカウンタ変数を用意している」以上のことは分かっていません。

第 3 章
タイルアドレスとタイル内座標を算出する処理

この章では、前章で確認した 1 ピクセルずつ描画処理を行っている `Vdp2DrawScroll` 関数内で、タイルアドレスとタイル内座標を算出するために呼び出している `Vdp2MapCalcXY` 関数を解説します。

3.1 if ブロックに入る条件

それでは、for の二重ループ内で呼び出している 3 つの関数の 1 つ目である `Vdp2MapCalcXY` 関数の実装を見てみましょう。まずは冒頭箇所です（リスト 3.1）。

▼リスト 3.1: vidsoft.c: Vdp2MapCalcXY 関数（冒頭）

```
static INLINE void FASTCALL Vdp2MapCalcXY(
   vdp2draw_struct *info, int *x, int *y, screeninfo_struct *sinfo, Vdp2* regs,
   u8 * ram, int bad_cycle)
{
   ・・・省略・・・
   const int check = ((y[0] >> cellwh) << 16) | (x[0] >> cellwh);
   ・・・省略・・・
   if(check != sinfo->oldcellcheck)
   {
      sinfo->oldcellx = x[0] >> cellwh;
      sinfo->oldcelly = y[0] >> cellwh;
      sinfo->oldcellcheck = (sinfo->oldcelly << 16) | sinfo->oldcellx;
```

まず `check` へ代入する値を算出している式について説明します。`y[0]` と `x[0]` は、この関数の第 2・第 3 引数で渡された描画する元のピクセルデータの座標です。ポインタ

で渡されているため、値の参照を行っています。[*1]そして、 `cellwh` には 3 が入っています。これはタイルの幅と高さである「8」による割り算をビットシフトで行うための値です。 `y[0] >> cellwh` と `x[0] >> cellwh` では、指定された Y 座標と X 座標がそれぞれタイルだと何番目（0 始まり）であるかを算出しています。（以降、これを「タイル座標」と呼ぶことにします。）最後に、タイル Y 座標を 16 ビット左シフトした値と、タイル X 座標の論理和を算出し、 `check` へ代入しています。すなわち、 `check` には指定された座標のタイル座標が入っています。

そして、if の条件式は一旦飛ばし、if ブロック内の冒頭処理について説明します。先程も説明した `cellwh` 分のビットシフトにより X と Y それぞれのタイル座標を算出し、引数でポインタで渡されたある構造体のメンバー変数（ `oldcellx` と `oldcelly` ）へ保存しています。また、それらのメンバー変数を用いて、 `check` で行ったものと同様に、上位 16 ビットにタイル Y 座標、下位 16 ビットにタイル X 座標をまた別のメンバー変数（ `oldcellcheck` ）へ保存しています。

以上を踏まえて if の条件式に戻ると、この時、 `oldcellcheck` には前回この関数が呼ばれた時に指定された座標のタイル座標が入っています。振り返ると、この関数は二重ループで画面解像度の分だけ 1 ピクセルずつ処理する中で呼ばれていました。そのため、 `oldcellcheck` と `check` の比較は「前回から 1 ピクセル進んだ結果、タイル単位で見ると次のタイルへ進んだか？」という意味になります。そして、この if ブロック内は次のタイルへ進んでいた場合の処理という訳です。

「タイル」と呼ぶか「セル」と呼ぶか

前述の通り、8 × 8 などのピクセルデータの集まりをコード内では「セル」とも呼んでいます。コードを読んでいると「タイル」はもうちょっと広い意味でも使っているようで、小さなピクセルデータの集まりを並べていく描画方式自体についても「タイル」と呼んでいるようです。（対して、1 枚のビットマップデータを表示することもできてその描画方式は「ビットマップ」と呼んでいます。）本書では、描画方式自体についても、小さなピクセルデータの集まりについても「タイル」と呼ぶことにします。

3.2　planenum 配列からプレーンアドレスを取得

そして、この if ブロック内の続きの処理はリスト 3.2 の通りです。なお、インデントは削除しています。

▼リスト 3.2: vidsoft.c: Vdp2MapCalcXY（if ブロック内（1））

```
 ・・・省略・・・

// Calculate which plane we're dealing with
planenum = ((y[0] >> sinfo->planepixelheight_bits) * info->mapwh)
```

[*1] 配列等ではない実体に対して `*y` の代わりに `y[0]` と書くのはあまり見ない気がしますが。

```
          + (x[0] >> sinfo->planepixelwidth_bits);
x[0] = (x[0] & sinfo->planepixelwidth_mask);
y[0] = (y[0] & sinfo->planepixelheight_mask);

// Fetch and decode pattern name data
info->addr = sinfo->planetbl[planenum];

・・・省略・・・
```

if ブロック内にはまだ処理がありますが、一旦ここまでを説明します。まず、`// Calculate which plane we're dealing with` というコメントが付いている部分についてです。コメントの通り、この部分では指定された座標（`y[0]` と `x[0]`）がどのプレーン（プレーン A からプレーン D）のものであるかを算出し `planenum` へ 0 から 3 のいずれかの数字で設定します。`planenum` へ代入する値を算出している計算式について、この関数の実行時、`sinfo->planepixelheight_bits` と `sinfo->planepixelwidth_bits` は共に 9 です。そのため、これらの変数を用いて `y[0]` と `x[0]` を右シフトしている箇所は、与えられた X・Y 座標をそれぞれ 9 ビット右シフトしていることになります。9 ビット右シフトは 512 による除算と等しいです。[*2]そしてこの 512 というのは、「プレーン」のサイズが 512x512px であることを示しています。そのため、X 座標・Y 座標それぞれを 512 で割るということは、指定された X 座標と Y 座標がそれぞれプレーンだと何番目（0 始まり）であるかを算出しています。今回の場合、そもそも画面解像度（320x240px）がプレーンサイズ（512x512px）未満でスクロールもしないので、X 座標・Y 座標をそれぞれ 9 ビット右シフト結果は常に共に 0 になるため、`planenum` は常に 0（プレーン A）となります。`planenum` を算出した後の 2 行は X・Y 座標をそれぞれプレーン内の座標になるようにマスクしていますが、同様の理由からマスクしても座標に変化はありません。

そして、`// Fetch and decode pattern name data` というコメントが付いている部分で前述した `planetbl` 配列に入っているプレーンアドレスを `info->addr` へ設定します。`planenum` は常に 0 なので、ここでは常にプレーン A のアドレスが設定されます。

3.3　取得したプレーンアドレスへ加算する計算式

引き続き、この if ブロック内を見ていきます（リスト 3.3）。

▼リスト 3.3: vidsoft.c: Vdp2MapCalcXY（if ブロック内（2））※ 改行やインデントは変更

```
・・・省略・・・

// Figure out which page it's on(if plane size is not 1x1)
info->addr +=
(
  ( ( (y[0] >> sinfo->pagepixelwh_bits) << pagesize_bits) << info->planew_bits) +
```

[*2] もちろん小数を考えない話です。

```
  (   (x[0] >> sinfo->pagepixelwh_bits) << pagesize_bits) +
  ( ( (y[0] & sinfo->pagepixelwh_mask) >> cellwh) << info->pagewh_bits) +
  (   (x[0] & sinfo->pagepixelwh_mask) >> cellwh)
) << (info->patterndatasize_bits + 1)

・・・省略・・・
```

複雑な式が出てきました。式の中の変数を見ていると、新たに「ページ（Page)」というものが登場しています。そして実はこの後、「キャラクタ（Character)」というものも登場します。タイル描画におけるピクセルの集まりはタイル（8x8）を最小単位としてプレーンに至るまで、「タイル⊂キャラクタ⊂ページ⊂プレーン」という包含関係になっています。ただ、今回の場合、キャラクタがタイルと同サイズ、プレーンがページと同サイズに設定されています。そのため、新たに登場した（今後登場する)「ページ」や「キャラクタ」については、「プレーン」や「タイル」と同じと考えて構いません。

それでは、この式を見ていきます。まず、リスト 3.4 の部分についてです。

▼リスト 3.4: 前段

```
  ( ( (y[0] >> sinfo->pagepixelwh_bits) << pagesize_bits) << info->planew_bits) +
  (   (x[0] >> sinfo->pagepixelwh_bits) << pagesize_bits) +
```

`y[0]` と `x[0]` をそれぞれ `sinfo->pagepixelwh_bits` の分だけ右シフトしている箇所があります。これらはこれまでタイルやプレーンにあったものと同様です。ページのサイズはプレーンと等しいので `sinfo->pagepixelwh_bits` は 9 で、これらのビットシフトは 512 による除算と等しいです。 `x[0]` と `y[0]` は共に 512 を超えることは無いことから、これらのビットシフトは共に 0 になります。これらのビットシフト後、さらに `<< pagesize_bits` というビットシフトがありますが、0 はいくらビットシフトしても 0 なので、その結果も 0 になります。そして `y[0]` に関してはさらに `<< info->planew_bits` というビットシフトがありますが、これも同様に 0 です。そのため、リスト 3.4 の部分は 0 になります。

次はリスト 3.5 の部分についてです。

▼リスト 3.5: 中段

```
  ( ( (y[0] & sinfo->pagepixelwh_mask) >> cellwh) << info->pagewh_bits) +
  (   (x[0] & sinfo->pagepixelwh_mask) >> cellwh)
```

`y[0] & sinfo->pagepixelwh_mask` と `x[0] & sinfo->pagepixelwh_mask` について、これらは X・Y 座標をページ内の座標にするためにマスクをしていますが、X・Y 座標が 512 以上の値になることは無いのでこのマスクも効果はありません。

以上を踏まえると、リスト 3.3 の式はリスト 3.6 のように整理できます。

▼リスト 3.6: 整理した式

```
info->addr +=
(
```

```
  ( (y[0] >> cellwh) << info->pagewh_bits) +
    (x[0] >> cellwh)
) << (info->patterndatasize_bits + 1);
```

リスト 3.6 を見ていきます。`y[0] >> cellwh` と `x[0] >> cellwh` は、前述の通り、それぞれ Y と X のタイル座標を算出します。そして、Y 座標に関してはさらに `info->pagewh_bits` の分だけ左シフトしています。`info->pagewh_bits` は 6 で、6 ビット左シフトは 64 を掛けることと等しいです。すなわち、タイル Y 座標へ 64 を掛けています。64 というのはページ内の 1 行のタイル数です。すなわち、タイル Y 座標が示す行数分のタイル数を求めています。そしてそれにタイル X 座標を足すことで、プレーンの先頭（左上）から指定された座標（`x[0]`, `y[0]`）の直前のタイルまでの総タイル数になります。（言い換えるとこれは、「指定された座標のタイルはプレーン先頭から 0 始まりで何番目か？」という値です。）

そして最後に `info->patterndatasize_bits + 1` の分だけ左シフトしています。`info->patterndatasize_bits` についてはややこしいので後述しますが、値としては 0 です。そのため、ここでは求めた総タイル数を 1 ビット左シフトしています。1 ビット左シフトは 2 を掛けることと等しいです。2 というのはコード内で「パターンネームデータ」や「パターンデータ」と呼んでいるデータ[*3]のバイト数[*4]です。パターンネームデータがどのようなものかは後述しますが、実は「プレーンアドレス」は、「パターンネームデータが並ぶテーブルの先頭アドレス」です。パターンネームデータは「キャラクタ」（＝「タイル」）に対応しており、画面左上の原点（0, 0）の位置のタイルに対応するパターンネームデータがパターンネームデータのテーブルの先頭に存在します。そのため、プレーンアドレスが指す先には（0, 0）のタイルのパターンネームデータがあります。

繰り返しになりますが、整理すると、リスト 3.6 の式の

```
  ( (y[0] >> cellwh) << info->pagewh_bits) +
    (x[0] >> cellwh)
```

の部分は、`x[0]` と `y[0]` で指定された座標の直前のタイルまでの総タイル数を求めています。そして、

```
) << (info->patterndatasize_bits + 1);
```

の部分では、求めた総タイル数へ 1 つのパターンネームデータのバイト数である 2 を掛けています。それにより、指定された座標のタイルに対応するパターンネームデータまでの総バイト数が得られます。それを `info->addr`（プレーンアドレス）へ加算しているので、`info->addr` は「指定された座標のタイルに対応するパターンネームデータのアドレス」になります。なお、今回の場合、描画する'A' のタイルは原点位置に存在するため、対応するパターンネームデータはテー

[*3] 本書では「パターンネームデータ」と呼ぶことにします。

[*4] パターンネームデータのサイズについては実は VDP2 のデバッグ画面の「NBG0/RBG1 Info」欄内に「Pattern Name data size = 1 word」と出ていました。（1 ワード = 2 バイト）

ブルの先頭に存在します。 y[0] も x[0] も 0 であるためリスト 3.6 の式で info->addr へ加算する値としても 0 であり、そのままのプレーンアドレスが info->addr に入っています。

info->patterndatasize_bits は何なのか？

info->patterndatasize_bits は「1 をその数だけ左シフトするとパターンネームデータのワード数になる値」です。今回の場合これは 0 です。1 を 0 ビット左シフトすると 1 なので「パターンネームデータは 1 ワード」となります。

3.4　パターンネームデータを取得し 2 つのアドレスを算出

if ブロック内では次に Vdp2PatternAddr 関数を呼び出します（リスト 3.7）。なお、if ブロック内の以降の処理はサンプルでは効果のないものなので、if ブロック内についてはこれで最後です。

▼リスト 3.7: vidsoft.c: Vdp2MapCalcXY（if ブロック内（3））

```
・・・省略・・・

Vdp2PatternAddr(info, regs, ram); // Heh, this could be optimized

・・・省略（サンプルでは効果のないコード）・・・
```

この関数の実装はリスト 3.8 の通りです。

▼リスト 3.8: vidsoft.c: Vdp2PatternAddr 関数（冒頭）

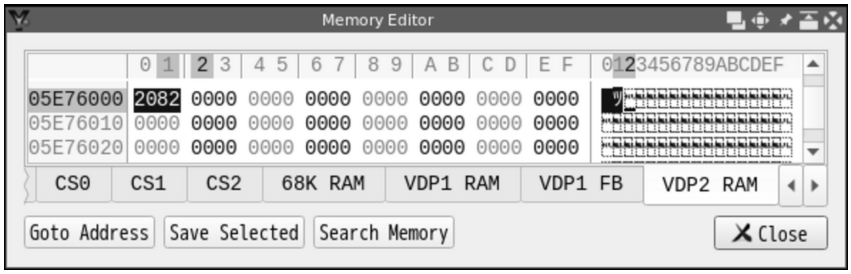
```
static INLINE void Vdp2PatternAddr(vdp2draw_struct *info, Vdp2* regs, u8* ram)
{
   switch(info->patterndatasize)
   {
      case 1:
      {
         u16 tmp = T1ReadWord(ram, info->addr);

          ・・・省略・・・
```

リスト 3.8 では冒頭部分のみ示しています。コードは省略しますが、 T1ReadWord 関数は第 1 引数で指定されたポインタ変数へ第 2 引数の値をオフセットとして足したアドレスから 1 ワード（2 バイト）を読み出して返す関数です。ここで、 T1ReadWord に第 1 引数で指定している ram は Vdp2PatternAddr 関数の引数で渡されてきたポインタ変数です。どこに実体があるのかと呼び出し元をたどっていくと、これは vdp2.c の u8 * Vdp2Ram というポインタ変数から来ている事が分かります。そして、このポインタ変数へは、同じく vdp2.c にある Vdp2Init 関数内で動的確保した 512KB のメモリーのアドレスを設定しています。すなわち、これは VDP2 の RAM をエミュレータとして用意しているものです。

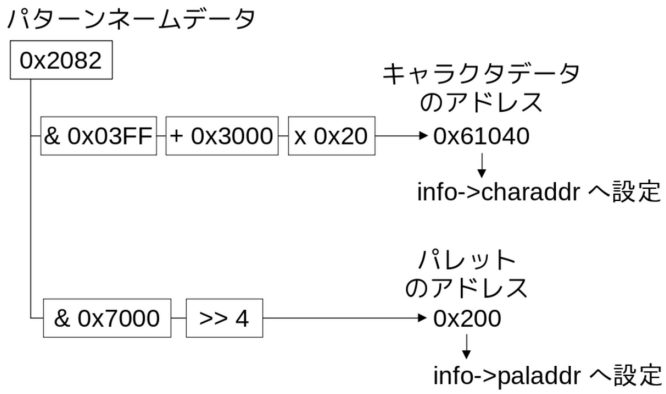
そして、 `info->addr` には先程算出したパターンネームデータ（2 バイト）を指すアドレスが入っています。このアドレスは VDP2 RAM 内のアドレス、すなわち VDP2 RAM の先頭からのオフセットです。そのため、ポインタ変数 `ram` へオフセットとして `info->addr` を足し、そのアドレスから 1 ワードを読み出すことで、VDP2 RAM からパターンネームデータを読み出す操作が行えます。なお、前述の通り、'A' のタイルのパターンネームデータ読み出し時、 `info->addr` にはプレーンアドレス（VDP2 デバッグ画面で確認した `0x76000` ）が入っています。

そして、改めて Yabause でサンプルを実行し、メニューバーから [Debug] - [Memory Editor] を開き、[VDP2 RAM] のタブからこのアドレスに何が書かれているのかを見てみると、 `0x2082` という値が書かれていることが分かります（図 3.1）。([Goto Address] ボタンからアドレスを入力してジャンプすると楽です。ただ、[VDP2 RAM] のタブを開くと `0x05E00000` というアドレスから始まっていることからもわかる通り、VDP2 RAM は `0x05E00000` にマップされているため、ここで見る際のアドレスとしては `0x05E76000` になることに注意してください。）



▲図 3.1: パターンネームデータとして何が書かれているのかを見てみる

そして、 `Vdp2PatternAddr` 関数の以降の処理ではこの `0x2082` という値（パターンネームデータ）を処理しています。これについては処理内容を図示したもので説明します（図 3.2）。



▲図 3.2: Vdp2PatternAddr 関数のパターンネームデータ取得後の処理

まずキャラクタデータのアドレスを得るまでの流れについてです。最初に `0x03FF` との AND を算出します。これはパターンネームデータの下位 10 ビットがキャラクタデータに関するもの

であることを示しています。そして、 `0x3000` という値を足しています。これはパターンネームデータが 1 ワードの場合の処理で、 `PNCN0` （NBG0 の場合）というレジスタの下位 5 ビットを 10 ビット左シフトした値です。パターンネームデータが 1 ワードの場合、キャラクタデータのアドレスを作るのに足りないため、レジスタに設定されている情報も使うようです。最後に `0x20` という値を掛けています。これはコード上でも `0x20` の乗算がベタ書きされているだけで、それをする理由などは読み取れませんでしたが、VDP2 RAM へのアドレスを得るための係数のようなものなのかと思います。以上の計算を行うと、今回の場合、 `0x61040` というアドレスが得られます。そしてそれを `info->charaddr` に設定します。

続いて、パレットのアドレスを得るまでの流れを説明します。まず「パレット」とは、使用するそれぞれの色の情報が定義されているテーブルです。サンプルでは'A' という文字を赤色で表示していましたが、この赤もパレット内に定義されています。そして最初に行う計算は `0x7000` との AND です。これはパターンネームデータのビット 14 からビット 12 を抽出しており、そこにパレットに関する情報があることを示しています。続いてそれを 4 ビット右シフトします。今回の場合、 `0x200` という値が得られます。最後にそれを `info->paladdr` へ設定します。

3.5　if ブロック後の処理について

そして、if ブロックの後の処理はリスト 3.9 の通りです。

▼リスト 3.9: vidsoft.c: Vdp2MapCalcXY 関数（末尾）

```
   if(check != sinfo->oldcellcheck)
   {
      ・・・省略・・・
   }

    ・・・省略・・・

   // Figure out which pixel in the tile we want
   if (info->patternwh == 1)
   {
      x[0] &= 8-1;
      y[0] &= 8-1;
       ・・・省略（サンプルでは効果のないコード）・・・
   }
    ・・・省略（サンプルでは効果のないコード）・・・
}
```

リスト 3.9 では、サンプルの場合に実行される行のみ残しています。 `x[0]` と `y[0]` のそれぞれを下位 3 ビットのみ抽出した値で上書きしています。これは、描画する元のピクセルデータの座標を 8x8px のタイル内の座標へ変換しています。

3.6　Vdp2MapCalcXY 関数のまとめ

`Vdp2MapCalcXY` 関数は以上です。少々複雑だったかと思いますので整理しておくと、この関数では、引数で指定された「描画する元のピクセルデータの座標」を用いて、まずタイル座標を算出します。そしてそのタイル座標が前回 `Vdp2MapCalcXY` 関数が呼ばれた時と異なる場合、キャラクタデータのアドレス（ `info->charaddr` ）とパレットのアドレス（ `info->paladdr` ）を更新します。最後に、タイル座標が前回と異なるか否かに関わらず、指定された X・Y それぞれの座標の下位 3 ビットより上位のビットを全て 0 にすることで、指定された座標をタイル内座標へ変換します。

第 4 章
取得したアドレスを用いた描画処理

この章では、前章で最終的に得られた `info->charaddr` と `info->paladdr` を用いてどのように描画を行うのかを解説します。

4.1 Vdp2MapCalcXY 関数から戻った後の処理

それでは、前章に引き続き、 `Vdp2DrawScroll` 関数の `Vdp2MapCalcXY` 関数から戻った後の処理を見ていきましょう（リスト 4.1)。

▼リスト 4.1: vidsoft.c: Vdp2DrawScroll 関数（Vdp2MapCalcXY 関数から戻った後）※ インデントは削除

```
・・・省略・・・
if (!bad_cycle)
{
   charaddr = info->charaddr;      ←算出したキャラクタアドレスをローカル変数へ設定
   paladdr = info->paladdr;        ←算出したパレットアドレスをローカル変数へ設定
}
・・・省略・・・
if (!Vdp2FetchPixel(info, x, y, &color, &dot, ram, charaddr, paladdr,color_ram))
{
   continue;
}
・・・省略・・・
```

`Vdp2MapCalcXY` 関数で得られたキャラクタデータのアドレスとパレットのアドレスをそれぞれローカル変数 `charaddr` と `paladdr` へ設定しています。そして、それらに加えて、タイル内座標へ変換された `x` と `y` も引数に与える形で `Vdp2FetchPixel` 関数を呼び出しています。

4.2 Vdp2FetchPixel 関数について

`Vdp2FetchPixel` 関数の実装はリスト 4.2 の通りです。

▼リスト 4.2: vidsoft.c: Vdp2FetchPixel 関数

```
static INLINE int Vdp2FetchPixel(
   vdp2draw_struct *info, int x, int y, u32 *color, u32 *dot, u8 * ram,
   int charaddr, int paladdr, u8* vdp2_color_ram)
{
   switch(info->colornumber)
   {
       ・・・省略（サンプルでは効果のないコード）・・・
      case 1: // 8 BPP
         *dot = T1ReadByte(ram, ((charaddr + (y * info->cellw) + x) & 0x7FFFF));
         if (!(*dot & 0xFF) && info->transparencyenable) return 0;
         else
         {
            *color = Vdp2ColorRamGetColor(
                        info->coloroffset + (paladdr | (*dot & 0xFF)),
                        vdp2_color_ram);
            return 1;
         }
       ・・・省略（サンプルでは効果のないコード）・・・
   }
}
```

まず、`T1ReadByte` 関数を呼び出している箇所についてです。第 1 引数として与えている `ram` は、エミュレータとして VDP2 の RAM を表現するために動的確保したメモリ領域の先頭アドレスです。そして、第 2 引数には計算式が書かれています。この計算式について、まず末尾で `0x7FFFF` と AND を取っているのは、そこより手前の計算結果が VDP2 RAM のサイズである 512（0x80000）KB を超えないようにするためのマスクです。このマスクを行っている箇所より左側では、`charaddr`（キャラクタデータのアドレス）へ、`y`（タイル内 Y 座標）に `info->cellw`（タイル幅 [px] = 8）を掛けたものと、`x`（タイル内 X 座標）を足しています。実はキャラクタデータに並ぶピクセルデータは 1px=1 バイトです。そのため、ここではキャラクタデータのアドレスへ「(Y 座標 * タイル幅 [px]) + X 座標」を足すことで、与えられた座標に対応するピクセルデータのアドレスを算出しています。そして算出したアドレスを `T1ReadByte` 関数へ与えて、そこから 1 バイトを読み出し、読み出した結果を `*dot`（呼び出し元からポインタで渡された変数）へ設定しています。

続いて if の行を見てみると、「`*dot` が 0」かつ「`info->transparencyenable` が 0 以外」の場合、戻り値 0 でこの関数から return します。これは透過機能に関するものです。VDP2 では NBG0 や NBG1 といったレイヤーごとに透過機能の有効/無効を切り替えられます。`info->transparencyenable` には現在処理対象にしているレイヤーの透過処理が有効な場合 1 が、無効な場合 0 が入っています。今回の場合、処理対象のレイヤー（NBG0）は透過処理が有効なので 1 が入っています。以上を踏まえると、この if 文は「取得したピクセルデータが 0

でかつ透過処理が有効な場合、戻り値 0 でこの関数から return」という意味になります。そして、`Vdp2DrawScroll` 関数を改めて見てみると（リスト 4.3)、`Vdp2FetchPixel` 関数が 0 を返した場合 `continue` するので、それ以降にある `TitanPutPixel` 関数によるピクセルの描画が行われないため、透過ができていることが分かります。

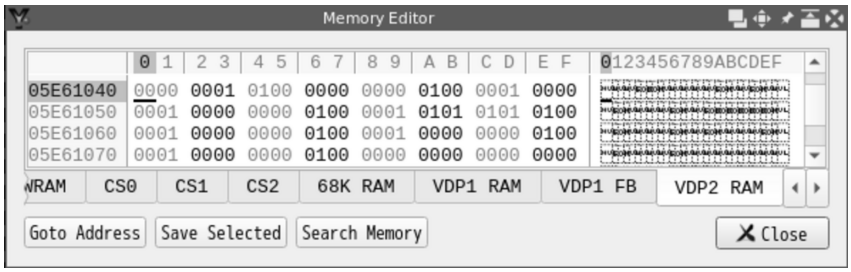
▼リスト 4.3: vidsoft.c: Vdp2DrawScroll 関数（Vdp2FetchPixel 関数の戻り値の扱いについて）

```
        ・・・省略・・・

       if (!Vdp2FetchPixel(info, x, y, &color, &dot, ram, charaddr, paladdr,
                           color_ram))    ←タイル内座標のピクセルを取得
       {
          continue;
       }
        ・・・省略・・・

       {
           ・・・省略・・・
          TitanPutPixel(priority, i, output_y,
             info->PostPixelFetchCalc(info, COLSAT2YAB32(alpha, color)),
             info->linescreen, info);
             ← 1ピクセル描画
       }
    }
     ・・・省略・・・
```

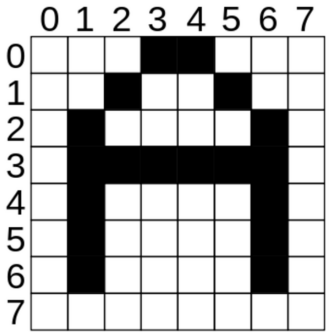
ここで、エミュレータ上で VDP2 RAM のキャラクタデータがどうなっているのかを見てみましょう。Yabause でサンプルを実行し、メニューバーから [Debug] - [Memory Editor] を開き、[VDP2 RAM] のタブからキャラクタデータのアドレス（VDP2 RAM 内のアドレスとしては `0x61040` ）に何が書かれているのかを見てみます。なお、例によってここで確認する際のアドレスとしては上位 12 ビットを `0x05E` とした `0x05E61040` というアドレスになることに注意してください。各ピクセルデータは 1 バイトで、タイルサイズが 8x8px、キャラクタサイズはタイルサイズと等しい設定なので、キャラクタ 1 つは 64 バイトです。そこで、`0x05E61040` から 64 バイト分の領域を見てみると、図 4.1 という状態でした。



▲図 4.1: サンプルのキャラクタデータの状態

各バイトに書かれている値は 0 か 1 のいずれかです。「0」が「透過」という意味であることが分かっているので、「1」については少なくとも「透過しない」ことは分かります。図 4.1 に表示され

ている 64 個分のバイトの並びを 8x8 で左上から並べると図 4.2 のようになります。



▲図 4.2: キャラクタデータの各バイトを 0=白、1=黒で並べたもの

'A' という文字が現れました。これで、タイルを構成する各ピクセルが VDP2 の RAM 上にどのように並んでいるのかを確認することができました。ただ、「赤」という色がどのように設定され、このタイルの描画時にどのように選択されているのかがまだ分かっていません。

4.3　パレットの色情報も見てみる

`Vdp2FetchPixel` 関数に戻って、if の else 節を見ると、`Vdp2ColorRamGetColor` という関数を呼び出しています（リスト 4.4）。

▼リスト 4.4: vidsoft.c: Vdp2FetchPixel 関数

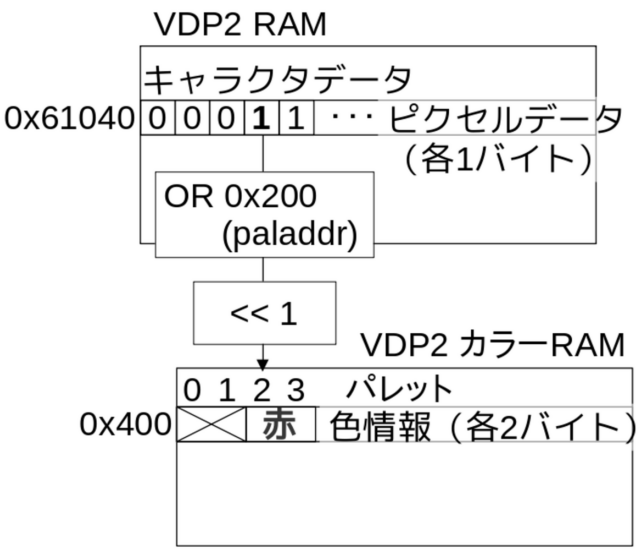
```
static INLINE int Vdp2FetchPixel(vdp2draw_struct *info, int x, int y, u32 *color,
 u32 *dot, u8 * ram, int charaddr, int paladdr, u8* vdp2_color_ram)
{
   switch(info->colornumber)
   {
       ・・・省略（サンプルでは効果のないコード）・・・
      case 1: // 8 BPP
         *dot = T1ReadByte(ram, ((charaddr + (y * info->cellw) + x) & 0x7FFFF));
         if (!(*dot & 0xFF) && info->transparencyenable) return 0;
         else
         {
            *color = Vdp2ColorRamGetColor(
                        info->coloroffset + (paladdr | (*dot & 0xFF)),
                        vdp2_color_ram);
            return 1;
         }
      ・・・省略（サンプルでは効果のないコード）・・・
   }
}
```

この関数は第 2 引数で指定されたアドレスへ第 1 引数で指定された値をオフセットとして足し

たアドレスから色情報を取得して返す関数です。第 2 引数に `vdp2_color_ram` というポインタ変数が指定されています。「カラー RAM（Color RAM）」というのは VDP2 の RAM とは別に存在する色情報を扱う専用の RAM でパレットもここに書かれています。 `vdp2_color_ram` には、VDP2 RAM の時と同様に、エミュレータ上でこの RAM を表現するために動的確保したメモリのアドレスが入っています。なお、カラー RAM に関してもメモリの動的確保は vdp2.c の `Vdp2Init` 関数で行っていて、カラー RAM 用には 4MB を確保しています。

また、 `Vdp2ColorRamGetColor` 関数の第 1 引数では少し計算を行っています。内側の括弧から見ていくと、まず `*dot` を `0xFF` でマスクしています。 `*dot` に取得したピクセルデータは 1 バイトなので、この場合このマスクは特に効果はありません。次に、マスクした結果の `*dot` と `paladdr` （パレットのアドレス）の OR を行っています。パレットのアドレスは `0x200` で、非透過のピクセルでは `*dot` は `1` だったので、OR の結果は `0x201` です。最後に `info->coloroffset` との和を取っていますがサンプルの実行時 `info->coloroffset` は 0 でしたので効果はありません。注目は「パレットのアドレスにピクセルデータを OR で結合している」ということです。すなわち、ピクセルデータ（1 バイト）はパレット内の色を指定するために使用されます。

そして、 `Vdp2ColorRamGetColor` 関数内の処理についてですが、それについては図で説明します（図 4.3）。

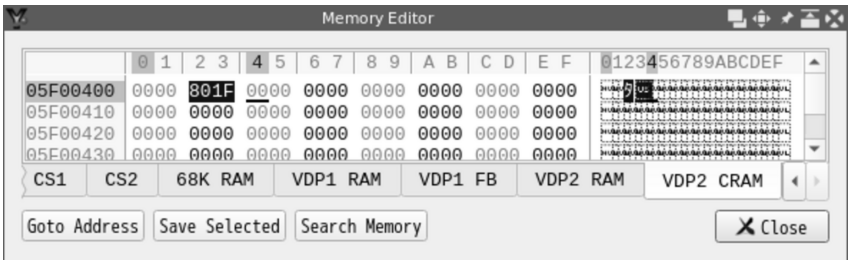


▲図 4.3: Vdp2ColorRamGetColor 関数の処理

まず、キャラクタデータから下向きに伸びている矢印について、これは、キャラクタデータ内のあるピクセルデータからどのようにパレットの色情報が指定されるかを示しています。まず、これは引数でやっていたことですが、ピクセルデータに `paladdr` （ `0x200` ）を OR 演算します。そして実は `Vdp2ColorRamGetColor` 関数内ではその後、1 ビット左シフトします。すなわち 2 倍するので `0x402` という値になります。これを VDP2 のカラー RAM 内のアドレスとして使うた

め、パレットのアドレスとしては `0x400` で、パレット内の色情報はパレットの先頭から +2 バイトの位置ということになります。なお、パレットの 0 バイト目はピクセルデータが 0 のとき透過の扱いなので色情報として使うことはできません。

実際にパレットの色情報をエミュレータ上で確認してみます。カラー RAM は `0x05F00000` にマップされているため、Memory Editor で確認する際のアドレスとしては `0x05F00402` です。見てみると図 4.4 の通りで、 `0x801F` という値が色情報として設定されていました。



▲図 4.4: ピクセルデータから参照される色情報は 0x801F

そして、色情報のフォーマットについては図 4.5 の通りです。図 4.5 と照らし合わせると、`0x801F` という色情報は「赤」であることがわかります。

色情報（赤の場合）

ビット	15	14 - 10	9 - 5	4 - 0
内容	不明	青	緑	赤
今回	1	00000	00000	11111

3ビット左シフトした
8ビットの値が
色情報として使用される

▲図 4.5: 色情報のフォーマット

なお、実は色情報のフォーマットについては `Vdp2ColorRamGetColor` 関数の実装からは分かりません。サンプルの方で文字色を変えてパレット内の色情報を確認したり、Memory Editor で書き換えてみたりすることで確認しました。「不明」としているビットについては、色を変えて確認する中で常に 1 が設定されていましたが、Memory Editor 上でこのビットを 0 に変えてみても特に表示に影響はありませんでした。

これで、VDP2 のタイル描画機能について、タイル情報を VDP2 の RAM にどのように設定するとどのように表示されるのかを、簡単な例だけではありますが、確認することができました。

おわりに

ここまで読んでいただきありがとうございます！ 本書ではセガサターンの VDP2 という画面描画 IC が持つ「タイル」という描画方式について Yabause というエミュレータのコードを通して解説しました。セガサターンが持つ画面描画機能は高機能ゆえに複雑で、「タイル」という描画方式ひとつでも本書ではまだまだ解説し足りないくらいではありますが、「ハードではこんな風に RAM を読んで、こんな風に描画しているのか」と何となく分かってもらえれば幸いです。

「タイル」は 2D のゲームにおける表現方法です。このような所からも、やはりセガサターンは 2D のゲームハードとして作られていた痕跡が感じられます。せっかくこんなにすごいものを持っているなら、それを使ったソフトを作ってみたいと思い、最近はタイル描画について調べていてこのような本を作るに至ったという経緯があったりします。どのように VDP2 の RAM が使われるのかは何となく分かったので、実験用のプログラムをいくつか作ってハードの制御方法を確認しつつ、何かソフトを作れたら良いなと考えています。

セガサターンタイル描画のうすい本

2024 年 11 月 2 日　ver 1.0 （技術書典 17 新刊）

著　者　大神祐真
発行者　大神祐真
連絡先　yuma@ohgami.jp
http://yuma.ohgami.jp
@yohgami (https://x.com/yohgami)
印刷所　日光企画

(powered by Re:VIEW Starter)