セガサターン
自作シンセサイザー
読本

— シンセサイザーだって作れる
セガサターンの音源 IC を
エミュレータのコードで解説 —

［著］大神祐真

コミックマーケット 100 新刊
2022 年 8 月 13 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！

本書は、筆者のセガサターン自作シンセサイザーが持つシンセサイザーとしての基本機能の説明を通して、セガサターンの音源 IC にはシンセサイザーを実現できるほどの機能があることを紹介します。

本書の構成

本書は以下の 4 章構成です。

第 1 章「セガサターンのサウンド周り」

セガサターンのサウンド周りのハードウェア構成を紹介します。

第 2 章「オシレータとピッチベンド」

シンセサイザーにおけるオシレータについて説明し、SCSP で波形データを再生する方法を紹介します。また、音の高さ (ピッチ) を変更する機能と、シンセサイザーにおけるピッチベンドを説明します。

第 3 章「EG(エンベロープジェネレータ)」

EG(エンベロープジェネレータ) という機能について解説し、それが SCSP でどのように実現されているのかを説明します。

第 4 章「LFO(低周波オシレータ)」

LFO(低周波オシレータ) という機能について解説し、それが SCSP でどのように実現されているのかを解説します。

対象とするエミュレータについて

これまでのセガサターン関連の既刊[*1]同様、「Yabause」[*2]というエミュレータを対象にコードリーディングを行います。Yabause のソースコードリポジトリは以下の通りです。

- `https://github.com/Yabause/yabause/`

なお、全てではないですが、ドキュメント化されている部分もあり、それらは以下の Wiki に書かれています。こちらの Wiki も適宜参照します。

- `https://wiki.yabause.org/`

[*1] 「エミュレータのコードを読んでわかるセガサターン」・「セガサターン CD システムのうすい本」・「セガサターンと MIDI で通信する本」があります。詳しくは筆者のウェブページをご覧ください。

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

電子版や本書の更新情報について

本書の電子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。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。

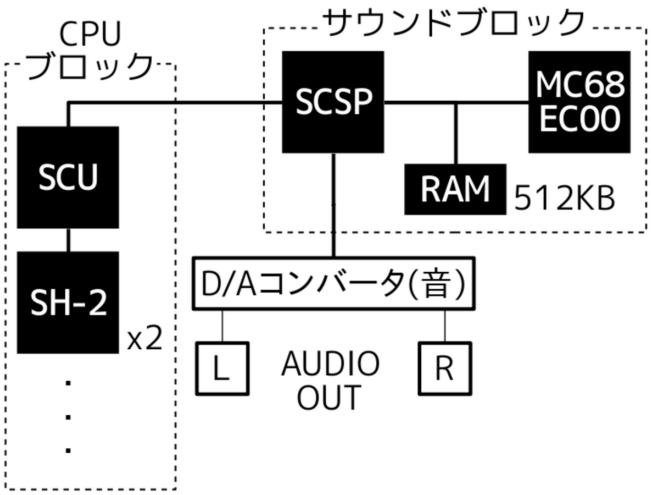
目次

第 1 章
セガサターンのサウンド周り

この章では、セガサターンのサウンド周りのハードウェア構成を紹介します。

1.1 サウンド周りの構成

セガサターンのハードウェア構成は公式で構成図が公開されています[*1]。メイン CPU とサウンド周りを抜き出すと図 1.1 の通りです。



▲図 1.1: セガサターンのサウンド周りのハードウェア構成

[*1] 「【連載】セガハードストーリー第 5 回　家庭用ゲーム機新時代の幕開け『セガサターン』｜セガ」https://sega.jp/history/hard/column/column_05.html

「サウンドブロック」の枠内がサウンドに関するもので、それぞれを簡単に説明すると以下の通りです。

SCSP

セガサターンの音源 IC です。音を作るための様々な機能を備えており、本書で解説する自作シンセサイザーの基本機能は全て SCSP で実現しています。サウンドのミキサーの機能も備えているため、SCSP 内でミキシングされた音声信号は D/A コンバータを通してそのままセガサターンの音声出力から出力されます。専用の DSP も備えており、独自の DSP 命令で信号処理なども可能です。なお、SCSP の各レジスタはメイン CPU(SH-2) とサウンド用 CPU(MC68EC00) のどちらにもマップされており、どちらからもアクセス可能になっています。

MC68EC00

サウンド用の CPU です。この CPU を使うことで、サウンド周りの処理を物理的に独立して実行させることができます。

RAM(512KB)

サウンド用の RAM です。SCSP で使用する波形データや、MC68EC00 用の命令などを置きます。これも、メイン CPU とサウンド用 CPU の両方にマップされています。

メイン CPU(SH-2) 側からは、SCU(System Control Unit) というシステム制御用の IC を通してサウンドブロックへ接続されています。本書で解説する自作シンセサイザーはメイン CPU から SCSP を制御します。SCSP 内の DSP やサウンド用 CPU は使用していません。[*2]

1.2　SCSP の概要

SCSP がどのような IC なのかを Yabause のドキュメントとコードで見ていきます。

♣ SCSP のドキュメントを見てみる

まず、ドキュメント化されている部分を先に見てみることにします。SCSP については、Yabause 公式 Wiki の以下のページにドキュメントがあります。

- 「SCSP - Yabause」
  - `https://wiki.yabause.org/index.php5?title=SCSP`

SCSP の概要とおおよそのレジスタについてはここに書かれています[*3]。色々と書かれていますが、ここでは以下のポイントを紹介しておきます。(具体的な機能は次章から見ていきます。)

[*2] 本書執筆時点 (2022 年 5 月) はまだ使っていない、という話で、将来的には DSP やサウンド用 CPU も活用していきたいと思っています。

[*3] まだ作成中の状況のようで、実際に Yabause の SCSP 実装にあるレジスタでもドキュメントには書かれていないものがあったりしますので、コードも併せて見ていく必要があります。

PCM に対応

波形データフォーマットとして PCM に対応しています。(サンプリング周波数：44.1kHz、量子化ビット数：8 あるいは 16 ビット[*4])

32 チャンネル搭載

最大 32 本の波形を同時に扱うことができます。

チャンネル別のレジスタと共通の制御レジスタ

各チャンネルに**スロットレジスタ**と呼ばれるレジスタがあります。また、チャンネル間で共通の制御レジスタ (**共通制御レジスタ**) もあります。

また、**スロットレジスタは各 32 バイト**あり、**32 チャンネル分で 1024 バイト分の領域**があります。**共通制御レジスタはチャンネル間で共通なので 1 つだけ**で、**48 バイト分の領域**があります。SH-2 からアクセスする際[*5]のメモリマップは表 1.1 の通りです。

▼表 1.1: スロットレジスタ・共通制御レジスタのメモリマップ (SH-2 からのアクセスの場合)

アドレス	レジスタ
0x25B00000	スロットレジスタ (チャンネル 0)
0x25B00020	スロットレジスタ (チャンネル 1)
...	...
0x25B003C0	スロットレジスタ (チャンネル 30)
0x25B003E0	スロットレジスタ (チャンネル 31)
0x25B00400	共通制御レジスタ

♣ SCSP のソースコードを見てみる

続いて SCSP のソースコードを見てみます。SCSP のソースコードは Yabause のリポジトリ直下からみて yabause/src/scsp.c にあるファイルを参照します。

scsp.c の冒頭に書かれているコメントで、共通制御レジスタとスロットレジスタのビットフィールドについて説明されています (リスト 1.1)。

▼リスト 1.1: yabause/src/scsp.c: 冒頭のコメント

```
・・・省略・・・
//----------------------------------------------------------------
//
// Common Control Register (CCR)
//
//      $+00      $+01
```

[*4] スロットレジスタ (後述) の PCM8B ビットで選択します。

[*5] キャッシュを経由しない「キャッシュスルー」の場合のアドレスです。キャッシュを経由する場合のアドレスは、最上位 4 ビットが 0 になります。(例えば、共通制御レジスタの先頭アドレスは 0x05B00400 になります。) ただ、レジスタは直接制御したい都合上、キャッシュを経由しない方が良いので、以降では、SH-2 からのアクセスに関してはキャッシュスルーのアドレスのみ紹介します。この辺りについて詳しくは、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」をご覧ください。

```
// $400 ---- --12 3333 4444 1:MEM4MB memory size 2:DAC18B dac for digital output ›
›3:VER version number 4:MVOL
// $402 ---- ---1 1222 2222 1:RBL ring buffer length 2:RBP lead address
 ・・・省略・・・
//
//-------------------------------------------------------------
//
// Individual Slot Register (ISR)
//
//     $+00      $+01
// $00 ---1 2334 4556 7777 1:KYONEX 2:KYONB 3:SBCTL 4:SSCTL 5:LPCTL 6:PCM8B 7:SA ›
›start address
// $02 1111 1111 1111 1111 1:SA start address
 ・・・省略・・・
```

Yabause Wiki の SCSP のページにも書かれていた内容がコードにもコメントで書かれています。なお、`Common Control Register (CCR)` や `Individual Slot Register (ISR)` と書かれている行から空行の次の行に `$+00　　$+01` と書かれています。これは、**バイトオーダーがビッグエンディアン**であることを示しています。例えば、共通制御レジスタの一番最初の 8 ビット (`MEM4MB` を含む 8 ビット) にバイトアクセスする際のアドレスは `0x25B00400` で、次の 8 ビット (`VER` を含む 8 ビット) は `0x25B00401` となります。

本書で解説する自作シンセサイザーの各機能は主にスロットレジスタで設定します。そのため、次章からのシンセサイザーの各機能の解説では、リスト 1.1 で `Individual Slot Register (ISR)` と説明されているビットフィールドの機能をドキュメントやソースコードから説明していくことになります。

1.3　シンセサイザーの基本機能について

なお、自作のシンセサイザー製作に際して、一般的にシンセサイザーがどの様な機能を持っているか、そしてそれらがどの様な機能であるかは、Yamaha が公開している以下の特集ページを参考にしています。

- ヤマハ | シンセサイザー入門 - スペシャルコンテンツ
  - `https://jp.yamaha.com/products/contents/music_production/guide_to_synth/`

第 2 章
オシレータとピッチベンド

この章では、シンセサイザーにおけるオシレータについて説明し、SCSP で波形データを再生する方法を紹介します。また、音の高さ (ピッチ) を変更する機能と、シンセサイザーにおけるピッチベンドを説明します。

2.1 シンセサイザーにおけるオシレータとは

シンセサイザーにおけるオシレータは、様々な音を作る素となる波形です。多くのシンセサイザーにはプリセットで既に作られた波形データが入っていたりしますが、原始的なシンセサイザーとしてはオシレータとして用意された基本波形を変化させたり合成したりして音を作ります。

セガサターン自作シンセサイザーでは、図 2.1 の波形とホワイトノイズに対応しています。

▲図 2.1: セガサターン自作シンセサイザーで用意している基本波形

シンセサイザーにおけるオシレータと、後述するピッチベンドについて、それがどのようなものであるかは、Yamaha が公開している以下のページを参考にしています。

- ヤマハ | 発振回路（発振器）＝オシレーター - シンセサイザー入門
  - https://jp.yamaha.com/products/contents/music_production/guide_to_synth/002/index.html

2.2　オシレータデモ

各オシレータ波形の再生デモを以下で公開しています。

- セガサターン自作シンセサイザー　オシレータ (ノコギリ波/矩形波/サイン波) デモ - YouTube
  - `https://youtu.be/DZGj1zulsi4`

オシレータ波形として対応している「ノコギリ波」・「矩形波」・「サイン波」それぞれの音を鳴らすデモです。この章では、SCSP で波形を鳴らす方法を紹介します。

2.3　SCSP で波形データを再生するには

SCSP で波形データを再生する方法を Yabause のドキュメントとコードから見ていきます。

♣ 波形データのアドレスを SCSP へ設定

第 1 章で説明の通り、SCSP は波形データとして PCM に対応していて、再生したい波形データはサウンド用の RAM へ配置します。

SCSP でこれを再生するには、まず、この**波形データのアドレスを再生させたいチャンネルのスロットレジスタへ設定**します。Yabause Wiki の SCSP のページ[*1]の「Individual slot registers」を見ると、スロットレジスタに `SA` (start address) というビットフィールドがあります。ここへ波形データのアドレスを設定します。

ただ、 `SA` は 16 ビット刻みで、先頭 16 ビットのビット [3:0] にも、先頭から 2 バイト目のビット [15:0] にもあります。これらはどう使うのでしょうか？ ソースコードを見てみることにします。

まず、先頭 16 ビットのビット [3:0] への書き込みアクセスがどうなるのかを見てみます。チャンネル 0 のスロットレジスタで、バイト単位の書き込みと仮定すると、アドレスは `0x25B00001` です。

セガサターン上のメモリアクセスがどのように処理されるかは、yabause/src/memory.c の `MappedMemoryInit()` から辿ることができます[*2]。アドレス `0x25B00001` についてはリスト 2.1 の通りです。

▼リスト 2.1: yabause/src/memory.c: MappedMemoryInit()

```
・・・省略・・・
void MappedMemoryInit(SH2_struct *msh2, SH2_struct *ssh2, SH2_struct *sh1)
{
   SH2_struct *sh2[2] = { msh2, ssh2 };
   int i;
```

[*1] `https://wiki.yabause.org/index.php5?title=SCSP`

[*2] 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」の「第 3 章 CPU 周り」を参照ください。

```
// MSH2/SSH2
for (i = 0; i < 2; i++)
{
    ・・・省略・・・
   FillMemoryArea(sh2[i], 0x5B0, 0x5BF, &Sh2ScspReadByte,
                                        &Sh2ScspReadWord,
                                        &Sh2ScspReadLong,
                                        &Sh2ScspWriteByte,
                                        &Sh2ScspWriteWord,
                                        &Sh2ScspWriteLong);
    ・・・省略・・・
```

`FillMemoryArea()` が、Yabause 上でのメモリアクセス時にどの関数を呼び出すかを対応付けている関数です。簡単に説明すると、第 2・第 3 引数でアドレスの範囲を指定していて (この場合 `0xX5B00000` から `0xX5BFFFFF`)、指定された範囲のメモリアクセス (読み出し/書き込み) 時に呼び出す関数を第 4 から第 9 引数で指定しています。指定されている関数名から分かるかと思いますが、第 4 引数から順に「1 バイト単位の読み出し」・「2 バイト (ワード) 単位の読み出し」・「4 バイト (ロング) 単位の読み出し」・「1 バイト単位の書き込み」・「2 バイト (ワード) 単位の書き込み」・「4 バイト (ロング) 単位の書き込み」に対応しています。

では、1 バイト単位の書き込みで呼び出される `Sh2ScspWriteByte()` の実装を見てみます (リスト 2.2)。

▼リスト 2.2: yabause/src/scsp.c: Sh2ScspWriteByte()

```
・・・省略・・・
void FASTCALL
Sh2ScspWriteByte(SH2_struct *sh, u32 addr, u8 val)
{
   ScspWriteByte(addr, val);
}
・・・省略・・・
```

`ScspWriteByte()` を呼び出していますので、この実装を見てみます (リスト 2.3)。

▼リスト 2.3: yabause/src/scsp.c: ScspWriteByte()

```
・・・省略・・・
void FASTCALL
ScspWriteByte (u32 addr, u8 val)
{
   scsp_w_b(addr, val);
}
・・・省略・・・
```

今度は `scsp_w_b()` を呼び出していますので、この実装を見てみます (リスト 2.4)。

▼リスト 2.4: yabause/src/scsp.c: scsp_w_b()

```
・・・省略・・・
void FASTCALL
scsp_w_b (u32 a, u8 d)
{
  a &= 0xFFF;

  if (a < 0x400)
    {
       if (use_new_scsp)
          scsp_slot_write_byte(&new_scsp, a, d);
       else
          scsp_slot_set_b(a >> 5, a, d);
      FLUSH_SCSP ();
      return;
    }
  else if (a < 0x600)
   ・・・省略・・・
}
・・・省略・・・
```

アドレスの下位 12 ビットを取り出し、その値で分岐しています。今着目しているのは `0x25B00001` なので、下位 12 ビットは `0x001` で一番最初の if の条件が成立します。

`use_new_scsp` は、Yabause の「Settings」ウィンドウの「Sound」タブの「Enable higher quality sound (may be slower)」のチェックボックスに対応しています[*3]。本書ではこの設定が有効になっている前提でソースコードを読み進めます。

すると、`scsp_slot_write_byte()` を呼び出しています。引数の `new_scsp` は yabause/src/scsp.c に定義されているグローバル変数です (リスト 2.5)。

▼リスト 2.5: yabause/src/scsp.c: new_scsp の定義

```
・・・省略・・・
struct Slot
{
   //registers
   struct SlotRegs regs;

   //internal state
   struct SlotState state;
};

struct Scsp
{
   u16 sound_stack[64];
   struct Slot slots[32];

   int debug_mode;
}new_scsp;
```

[*3] 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」を参照ください。

```
・・・省略・・・
```

new_scsp は struct Scsp のグローバル変数として定義されています。メンバーについては、本書においては struct Slot 型の slots 配列だけ気にしておけば良いです。この構造体配列で全 32 個のスロットレジスタを管理しています。

struct Slot の定義もすぐ上にあります。メンバーは struct SlotRegs 型の regs と struct SlotState 型の state だけです。struct SlotRegs という構造体で、正にスロットレジスタの構造を定義しています (リスト 2.6)。struct SlotState はエミュレータの動作上必要な内部状態の管理用の構造体です。

▼リスト 2.6: yabause/src/scsp.c: struct SlotRegs

```
・・・省略・・・
//unknown stored bits are unknown1-5
struct SlotRegs
{
   u8 kx;
   u8 kb;
   u8 sbctl;
   u8 ssctl;
   u8 lpctl;
   u8 pcm8b;
   u32 sa;
    ・・・省略・・・
```

今着目している SA もちゃんとあります。

今度は scsp_slot_write_byte() を見てみます (リスト 2.7)。

▼リスト 2.7: yabause/src/scsp.c: scsp_slot_write_byte()

```
・・・省略・・・
void scsp_slot_write_byte(struct Scsp *s, u32 addr, u8 data)
{
   int slot_num = (addr >> 5) & 0x1f;
   struct Slot * slot = &s->slots[slot_num];
   u32 offset = (addr - (0x20 * slot_num));

   switch (offset)
   {
   case 0:
       ・・・省略・・・
   case 1:
      slot->regs.ssctl = (slot->regs.ssctl & 2) | ((data >> 7) & 1);
      slot->regs.lpctl = (data >> 5) & 3;
      slot->regs.pcm8b = (data >> 4) & 1;
      slot->regs.sa = (slot->regs.sa & 0xffff) | ((data & 0xf) << 16);
      break;
   case 2:
      slot->regs.sa = (slot->regs.sa & 0xf00ff) | (data << 8);
      break;
```

```
   case 3:
      slot->regs.sa = (slot->regs.sa & 0xfff00) | data;
      break;
   case 4:
    ・・・省略・・・
   }
}
 ・・・省略・・・
```

まず、変数 `slot_num` へ何番目のスロットレジスタであるか (スロット番号) を設定しています。スロットレジスタは `0x20` 毎の間隔で並んでいるので、アドレスを 5 ビット右シフトして抽出した値がスロット番号になります。今回の場合、第 2 引数 `addr` には `0x001` が渡ってくるので、`slot_num` は 0 になります。

次に、`struct Slot *` 型の `slot` に、`slot_num` (この場合 0) 番目のスロットレジスタの `struct Slot` のポインタを設定します。先述したグローバル変数 `new_scsp` の `new_scsp.slots[0]` のポインタが設定されます。

そして、`offset` にスロットレジスタ内のオフセットを設定します。`addr` が `0x001` で `slot_num` が 0 なので、今回の場合、1 が設定されます。

`switch (offset)` の `case 1:` を見ると、`slot->regs.sa` へ `data` の下位 4 ビットを 16 ビット左シフトした値を `slot->regs.sa` の 16 ビット目の位置に設定していることが分かります。`data` はこの関数の第 3 引数で、元を辿ると `0x25B00001` へバイト書き込みした際の 1 バイトです。そのため、**スロットレジスタ先頭 16 ビットのビット [3:0] の `SA` フィールドへの書き込みで `SA` のビット [19:16] へ設定**が行える事が分かります。

同様に、`case 2:` と `case 3:` の実装から、**スロットレジスタ先頭から 2 バイト目のビット [15:0] の `SA` フィールドへの書き込みで `SA` のビット [15:0] へ設定**が行える事が分かります。

以上から、**`SA` は 20 ビットのアドレス幅**であることが分かります。セガサターンのアドレス空間は 32 ビットあり、それより明らかに小さいので、アドレス空間上の任意の場所を `SA` に設定できるわけでは無い事が分かります。`SA` は PCM データのアドレスなので、**SCSP に再生させたい PCM データはどこに置いても良い訳ではない**、という事です。第 1 章で説明した通り、PCM データはサウンドブロックの RAM に置きます。`SA` にはこの RAM の先頭からのオフセットを設定します。

♣ 再生を指示

SCSP へ PCM データのアドレスを設定できたら、再生を指示すると SCSP はその波形を再生します。PCM データのアドレス設定と同様に、再生と停止もスロットレジスタで行います。

再生と停止についてはドキュメントである Yabause Wiki の SCSP のページに十分書かれているのでそちらを参照します。

再生と停止は SCSP ではキーオンとキーオフと呼びます。**キーオン/オフの状態を示すのがスロットレジスタ先頭 16 ビットのビット 11 の位置にある `KYONB` (key on bit)** というビットです。ただ、このビットに 1(キーオン)/0(キーオフ) を設定しても、それだけでは再生/停止は行われま

せん。このビットに該当チャンネルの再生/停止状態を設定した上で、同じく**スロットレジスタ先頭 16 ビットのビット 12 の位置にある `KYONEX` (key on execute) というビットに 1 を設定すると、 `KYONB` に設定した 1(キーオン)/0(キーオフ) が反映**され、再生/停止が行われます。なお、**`KYONEX` はどのチャンネルのスロットレジスタに設定しても、全チャンネルへ反映**されます。

2.4　ループ再生について

キーを押下中はその音を鳴らし続けたいといった場合、鳴らし続ける間の PCM データを全て RAM へ置いておかなければならないのか、というと、そんなことはありません。ループ再生の機能を使うことで、RAM に置いておく PCM データは 1 周期分だけで良くなります。

ループの設定についても、現状の自作シンセサイザーで行っている程度の内容は Yabause Wiki の SCSP のページの内容で十分なので、そちらを参照して説明します。

ループの設定もスロットレジスタにあります。**スロットレジスタ先頭から 4 バイト先にある 16 ビットの `LSA` (loop start address) というビットフィールドに、 `SA` に指定した PCM データのどこからループをスタートするかを設定**します。そして、**続く 16 ビットの `LEA` (loop end address) というビットフィールドに、どこまででループするかを指定**します。 **`LSA` ・ `LEA` 共に設定値はサンプル数**です。

なお、 `LPCTL` (loop control) というビットフィールドでループの仕方を設定できます。これについては次節で登場しますのでそちらで説明します。

以上により、**RAM には 1 周期分の PCM データを置いておき、 `LSA` にその開始位置 (サンプル数設定なので 0) を設定し、 `LEA` にその 1 周期の終了位置 (サンプル数設定なので正に 1 周期分のサンプル数) を設定すれば、その 1 周期をキーオンの間、再生させ続けることができる**訳です。

2.5　音階を表現するには

「音階を表現するには、鳴らしたい音階の数だけ PCM データを RAM に置いておく必要があるのか」というと、それもそんな事はありません。音の高さは波形の周波数ですが、SCSP には `SA` で指定された PCM データの周波数を変化させて再生する機能があります。それを使うことで動的に音の高さを変えることができ、様々な音階を表現できます。

この機能もスロットレジスタにビットフィールドがあります。オクターブ単位で変化させるフィールド `OCT` と、オクターブ内で変化させるフィールド `FNS` があります。共にスロットレジスタ先頭から 16 バイト先の 16 ビットの領域にあります。

`OCT` については Yabause Wiki の SCSP のページに説明があります。 **`OCT` は 16 ビットの領域内のビット [14:11]** にあります。「OCT register notes」の記述によると、**設定値が 0x0 の際は変化無し**で、**0x1 では周波数 2 倍 (+1 オクターブ)**、**0x2 では周波数 4 倍 (+2 オクターブ)**、のよう

に設定値によって**オクターブ単位で音の高さが上がって**行き、**0x7 の +7 オクターブまで上げることができる**ようです。**0x8 から 0xf は負の値**として扱われるようで、**0xf のとき周波数 1/2 倍 (-1 オクターブ)**、**0xe のとき周波数 1/4 倍 (-2 オクターブ)**、のように**オクターブ単位で音の高さを下げる**ことができ、**0x8 で-8 オクターブまで下げる**ことができるようです。すなわち、**`OCT` フィールドの最上位ビットは符号ビット**という訳です。

続いて **`FNS` は 16 ビットの領域内のビット [9:0]** にあります。これは Yabause Wiki に機能についての説明がなく、コードを見てみることにします。

`FNS` を含む 16 ビット領域のアドレスは、チャンネル 0 のスロットレジスタの場合、`0x25B00010` です。このアドレスへワードアクセスで書き込みを行った場合、`MappedMemoryInit()` で登録している `Sh2ScspWriteWord()` が呼び出されます。ここからの関数呼び出しの流れは前述したバイトアクセスで書き込みを行った場合と同様なので省略します。最終的に `scsp_slot_write_word()` が呼び出されます (リスト 2.8)。

▼リスト 2.8: yabause/src/scsp.c: scsp_slot_write_word()

```
・・・省略・・・
void scsp_slot_write_word(struct Scsp *s, u32 addr, u16 data)
{
   int slot_num = (addr >> 5) & 0x1f;
   struct Slot * slot = &s->slots[slot_num];
   u32 offset = (addr - (0x20 * slot_num));

   switch (offset >> 1)
   {
    ・・・省略・・・
   case 8:
      slot->regs.unknown3 = (data >> 15) & 1;
      slot->regs.unknown4 = (data >> 10) & 1;
      slot->regs.oct = (data >> 11) & 0xf;
      slot->regs.fns = data & 0x3ff;
      break;
   ・・・省略・・・
   }
}
・・・省略・・・
```

バイト書き込みの場合と少し異なり、引数 `addr` にはアクセス先アドレスと 0xFFE の AND をとった結果が渡されます[*4]。今回の場合、0x010 です。

ローカル変数 `slot_num` は今回も 0 です。そのため `offset` は 0x010 です。`switch` 文では、`offset` を 1 ビット右シフトしているので 8 になります。

`case 8` の処理を見ると、`slot->regs.fns` (スロットレジスタの `FNS` フィールド) へ、指定されたデータ (`data`) の下位 9 ビットを設定していることが分かります。

では、ここで `slot->regs.fns` へ設定した値はどのように使われるのかを見てみます。“fns”

[*4] 実装については呼び出し元である `scsp_w_w()` を参照してください。

をキーワードに yabause/src/scsp.c 内を検索すると、これを使用している関数として `op1()` がヒットします (リスト 2.9)。

▼リスト 2.9: yabause/src/scsp.c: op1()

```
・・・省略・・・
void op1(struct Slot * slot)
{
   u32 oct = slot->regs.oct ^ 8;
   u32 fns = slot->regs.fns ^ 0x400;
   u32 phase_increment = fns << oct;
    ・・・省略・・・
   slot->state.waveform_phase_value += (phase_increment + plfo_shifted);
}
・・・省略・・・
```

SCSP ではサウンド処理を内部的にパイプライン処理している様で、各段階の処理を `op<番号>()` の関数で実装しています[*5]。 `op1()` はその 1 段階目の処理です。

リスト 2.9 では、ここでの説明に不要な処理は省略しています。省略せずに残した部分を見ていくと、まず変数 `oct` へ、スロットレジスタの `OCT` フィールドの値と 8 を XOR した値を設定しています。 `OCT` フィールドは 4 ビットのビットフィールドで最上位ビットは符号ビットでした。8 との XOR によりこの符号ビットを反転させています。

次に変数 `fns` へ、 `FNS` フィールドの値と 0x400 を XOR した値を設定しています。 `FNS` フィールドは 10 ビットのビットフィールドでした。0x400 との XOR により `FNS` フィールドのビット [9:0] の次のビット 10 に 1 が設定されます。

そして、変数 `phase_increment` へ、変数 `fns` を変数 `oct` の分だけ左ビットシフトした値を設定しています。

最後に、変数 `phase_increment` を `slot->state.waveform_phase_value` へ加算しています。 `plfo_shifted` という変数も加算していますが、ここでは無視してください。

以上の計算にどういう意味があるのかは、 `slot->state.waveform_phase_value` がどういう変数で、そこへの加算にどういう意味があるかが分かれば分かります。

`slot->state.waveform_phase_value` について見ていきます。前述の通り `slot->state` は `struct SlotState` という構造体で、スロットレジスタの内部状態を管理しています (リスト 2.10)。

▼リスト 2.10: yabause/src/scsp.c: struct SlotState

```
・・・省略・・・
struct SlotState
{
   u16 wave;
   int backwards;
   enum EnvelopeStates envelope;
```

[*5] 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」をご覧ください。

```
   s16 output;
   u16 attenuation;
   int step_count;
   u32 sample_counter;
   u32 envelope_steps_taken;
   s32 waveform_phase_value;
   s32 sample_offset;
   u32 address_pointer;
   u32 lfo_counter;
   u32 lfo_pos;

   int num;
   int is_muted;
};
・・・省略・・・
```

`waveform_phase_value` は符号付きの 32 ビット整数型であることが分かります。そして yabause/src/scsp.c 内でこれを使用している箇所を探すとリスト 2.11 が見つかります。

▼リスト 2.11: yabause/src/scsp.c: op2()

```
・・・省略・・・
//address pointer calculation
//modulation data read
void op2(struct Slot * slot, struct Scsp * s)
{
   s32 md_out = 0;
   s32 sample_delta = slot->state.waveform_phase_value >> 18;
    ・・・省略・・・
   //address pointer

   if (slot->regs.lpctl == 0)//no loop
   {
      slot->state.sample_offset += sample_delta;
       ・・・省略・・・
   }
   else if (slot->regs.lpctl == 1)//normal loop
   {
      slot->state.sample_offset += sample_delta;

      if (slot->state.sample_offset >= slot->regs.lea)
         slot->state.sample_offset = slot->regs.lsa;
   }
    ・・・省略・・・
   if (!slot->regs.pcm8b)
      slot->state.address_pointer = (s32)slot->regs.sa + (slot->state.sample_offs›
›et + md_out) * 2;
   else
      slot->state.address_pointer = (s32)slot->regs.sa + (slot->state.sample_offs›
›et + md_out);
}
・・・省略・・・
```

変数 `md_out` へ値を設定する処理もありますがここでは省略しています。

`sample_delta` へ、`slot->state.waveform_phase_value` を 18 ビット右シフトした値を設定しています。`slot->state.waveform_phase_value` の下位 18 ビットを捨てている事が分かります。

`//address pointer` というコメント以降に `slot->regs.lpctl` による if の条件分岐があります。前節で「次節で説明する」としていた `LPCTL` がここで登場しました。`//no loop` や `//normal loop` というコメントが示す通り、**`LPCTL` へ 0 を設定すると「ループ無し」、1 を設定すると「通常のループ」**です。省略していますが、`else if` はまだ続き、`LPCTL` により他にもループの仕方を設定できます。自作シンセサイザーでは「通常のループ」を使用しています。

「通常のループ」の処理に、「`slot->state.sample_offset` が `slot->regs.lea` 以上であるか?」という if 文があり、それが成立する場合、`slot->state.sample_offset` に `slot->regs.lsa` を設定しています。スロットレジスタの `LEA` と比較しているという事で `slot->state.sample_offset` の値はサンプル数であることが分かります。**再生するサンプルが `LEA` 番目以降であれば `LSA` まで戻る**という訳です。

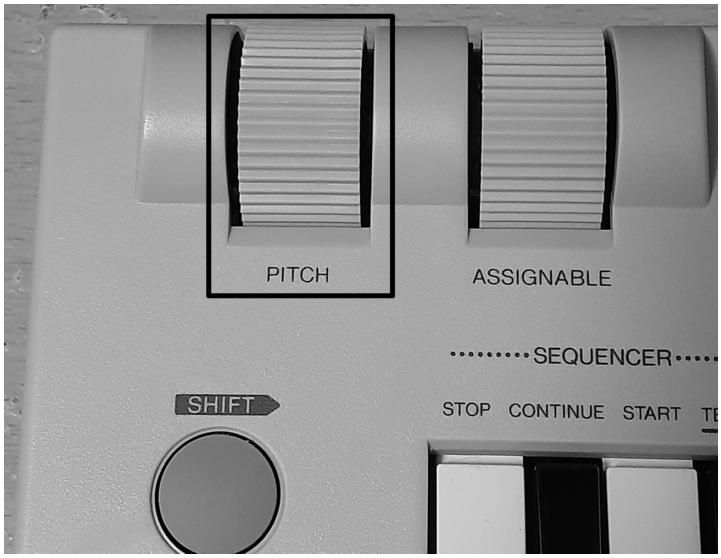
また、「ループ無し」・「通常のループ」共に `sample_delta` を `slot->state.sample_offset` へ加算しています。そして末尾の処理で `slot->regs.sa` (スロットレジスタの `SA`) に `slot->state.sample_offset` と `md_out` (ここでは 0) を足した値を `slot->state.address_pointer` へ設定しています。`slot->regs.pcm8b` すなわちスロットレジスタの `PCM8B` が 0 の場合、量子化ビット数が 16 ビット (2 バイト) という事で、`slot->state.sample_offset` と `md_out` の和を 2 倍しています。

`slot->state.address_pointer` は、`SA` に何らかの値を足した値を設定している事から、次に再生する PCM データのサンプルを示すポインタだと言えます。そして、`SA` からどれだけ進んだサンプルを次に再生するかが、`SA` へ足す値である `slot->state.sample_offset` 等で決まる、という訳です。

以上から、SCSP では**スロットレジスタの `OCT` ・ `FNS` 等を使用して、次に PCM データの何番目のサンプルを再生するかを制御することで周波数を変化させている**事が読み取れます。これにより、基準とする周波数 (音の高さ) の波形データを PCM で設定しておけば、SCSP の機能で音階を作ることができます。

2.6　ピッチベンドとは

この章で紹介するもう一つのシンセサイザー基本機能は「ピッチベンド」です。これは何らかの音を鳴らしている最中に図 2.2 の様なホイールあるいはつまみ等を操作すると動的に音の高さを変えられるというものです。



▲図 2.2: ピッチベンドホイールの例

2.7　ピッチベンドデモ

自作シンセサイザーでのピッチベンドのデモを以下で公開しています。

- セガサターン自作シンセサイザー　ピッチベンドデモ - YouTube
  - `https://youtu.be/0jO5LVnkRSw`

SCSP で動的に音の高さを変更することに関しては前述の通りなので、ここでは特に SCSP の機能に関して追加の説明はありません。

第 3 章
EG(エンベロープジェネレータ)

この章では、EG(エンベロープジェネレータ) という機能について解説し、それが SCSP でどのように実現されているのかを説明します。

3.1 EG(エンベロープジェネレータ) とは

EG(エンベロープジェネレータ) は、ある音を鳴らし始めてからそれを終えるまでの間の音の大きさを制御する機能です。これを何も制御していない様な例としては、キーを押下すると設定された音量で音が鳴り初め、押下中に音量の変化無く、キーを離すとパッと音量が 0 になる、というものです。EG では、例えばピアノの様に、キーを押下した瞬間に音が鳴り、押下し続けていても音が鳴り続ける事は無い、といった音量変化をつけることができます。

SCSP に関わらず、シンセサイザーにおける EG の用語として以下があります。

Attack

キー押下から設定された音量に達するまでの間の事です。線形に音量を変化させるので、この間の長さ (時間) で設定する際は「Attack Time(AT)」というパラメータがあったり、増幅量 (傾き) で設定する際は「Attack Rate(AR)」というパラメータがあったりします。

Decay

設定された音量に達してから Sustain Level(後述) まで減衰し切る間の事です。Attack と同様に、この間の長さ (時間) で設定する際は「Decay Time(DT)」、減衰量 (傾き) で設定する際は「Decay Rate(DR)」というパラメータがあったりします。

Sustain

ある音量まで減衰し切ったらそれを維持する事です。これは、その音量 (Level) の設定が「Sustain Level(SL)」というパラメータで行えたりします。

Release

キーを離してから音量が 0 まで減衰し切る間の事です。Attack 等と同様に、この間の長

さ (時間) で設定する際は「Release Time(RT)」、減衰量 (傾き) で設定する際は「Release Rate(RR)」というパラメータがあったりします。

これらはそれぞれの頭文字をとって「ADSR」と呼ばれたりします。
なお、EG について、Yamaha の以下のページを参考にしています。

- ヤマハ｜音量を変化させる装置＝アンプ - シンセサイザー入門
  - `https://jp.yamaha.com/products/contents/music_production/guide_to_synth/003/index.html`

3.2　EG デモ

自作シンセサイザーでの EG のデモを以下で公開しています。

- セガサターン自作シンセサイザー　EG(エンベロープジェネレータ) デモ - YouTube
  - `https://youtu.be/grs_v0f4Lcc`

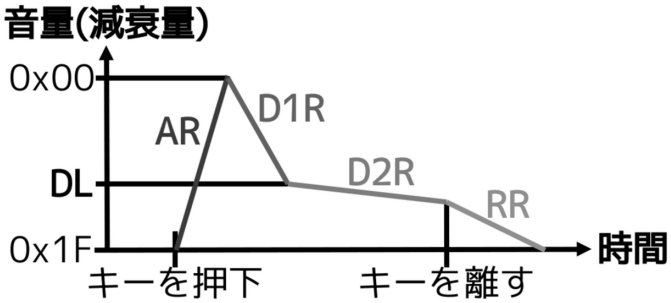
SCSP における EG のパラメータ設定を行い、音を鳴らすデモです。最後の方ではホワイトノイズの音に対して EG で音量変化を付けることで、海岸の波の音のようなものを作っています。

なお、デモで設定している他にもあと 1 つ EG 関連のパラメータが SCSP にはあるのですが、それはまだ自作シンセサイザーの方で対応していないのでデモ画面でも省略しています。このパラメータについても以降で説明します。

3.3　SCSP における EG 設定について

Yabause Wiki の SCSP のページを見ると、**ADSR 関連のフィールドがスロットレジスタの先頭 0x08 バイト以降の 32 ビットの領域にある**事が分かります。 **`AR` (Attack Rate)・ `D1R` (Decay 1 Rate)・ `D2R` (Decay 2 Rate)・ `DL` (Decay Level)・ `RR` (Release Rate) が、前述した ADSR に相当**するフィールドです。

EG デモにも出ていましたが、これらのパラメータをグラフに示すと図 3.1 の通りです。



▲図 3.1: SCSP における EG 設定

SCSP の場合は Decay(減衰) の後は Sustain(持続) ではなくもう一段 Decay(減衰) を設ける事ができます。そのため、**「Sustain Level」に相当するものとして「Decay Level(DL)」**があり、**DL に達する前後の Decay Rate をそれぞれ「Decay 1 Rate(D1R)」・「Decay 2 Rate(D2R)」**と呼んでいます。

なお、縦軸は上にいくほど音量が大きく、下にいくほど音量が小さいです。DL の設定値としては減衰量を設定するため、縦軸の値としては 0x00(減衰量 0) が上に来ています。

3.4　SCSP の EG 機能をコードから見てみる

「Attack」・「Decay 1」・「Decay 2」・「Release」の 4 状態における音量変化がどのように実装されているかをコードで見ていきます。

まず、`AR` (Attack Rate) の機能から「Attack」状態における音量変化の実装を見てみます。

`AR` は、スロットレジスタ先頭から +0x08 バイトの 16 ビット領域内でビット [4:0] にあるビットフィールドです。チャンネル 0 のスロットレジスタでバイトアクセスする際のアドレスは `0x25B00009` になります。

SCSP へバイト単位で書き込みする際の関数 `Sh2ScspWriteByte()` から辿る流れは `SA` で確認した際と同じで、最終的に `scsp_slot_write_byte()` へ行き着きます (リスト 3.1)。

▼リスト 3.1: yabause/src/scsp.c: scsp_slot_write_byte()

```
・・・省略・・・
void scsp_slot_write_byte(struct Scsp *s, u32 addr, u8 data)
{
   int slot_num = (addr >> 5) & 0x1f;
   struct Slot * slot = &s->slots[slot_num];
   u32 offset = (addr - (0x20 * slot_num));

   switch (offset)
   {
    ・・・省略・・・
```

```
   case 9:
      slot->regs.d1r = (slot->regs.d1r & 0x1c) | ((data >> 6) & 3);
      slot->regs.hold = (data >> 5) & 1;
      slot->regs.ar = data & 0x1f;
      break;
    ・・・省略・・・
   }
}
 ・・・省略・・・
```

指定された値 (変数 `data`) の下位 5 ビットを `slot->regs.ar` へ設定しています。

それでは、yabause/src/scsp.c 内で `slot->regs.ar` を使用している箇所を調べてみます。するとリスト 3.2 の処理が見つかりました。

▼リスト 3.2: yabause/src/scsp.c: op4()

```
 ・・・省略・・・
//interpolation
//eg
void op4(struct Slot * slot)
{
   int sample_mod_4 = slot->state.envelope_steps_taken & 3;

   if (slot->state.attenuation >= 0x3bf)
      return;

   if (slot->state.envelope == ATTACK)
   {
      int rate = get_rate(slot, slot->regs.ar);
      int need_step = need_envelope_step(rate, slot->state.sample_counter, slot);

      if (need_step)
      {
         int attack_rate = 0;

         if (rate <= 0x30)
            attack_rate = attack_rate_table[0][sample_mod_4];
         else
            attack_rate = attack_rate_table[rate - 0x30][sample_mod_4];

         slot->state.attenuation -= ((slot->state.attenuation >> attack_rate)) + ›
›1;

         if (slot->state.attenuation == 0)
            change_envelope_state(slot, DECAY1);
      }
   }
   else if (slot->state.envelope == DECAY1)
       ・・・省略・・・
   else if (slot->state.envelope == DECAY2)
       ・・・省略・・・
   else if (slot->state.envelope == RELEASE)
       ・・・省略・・・
```

```
}
・・・省略・・・
```

slot->state.envelope による if 文から、この変数に現在の EG の状態が格納されているのだと思われます。ATTACK と等しい場合の処理を見ていくと、get_rate() に slot->regs.ar を渡して関数呼び出ししています。get_rate() のコードはリスト 3.3 の通りです。

▼リスト 3.3: yabause/src/scsp.c: get_rate()

```
s32 get_rate(struct Slot * slot, int rate)
・・・省略・・・
{
   s32 result = 0;

   if (slot->regs.krs == 0xf)
      result = rate * 2;
   else
   {
      result = (slot->regs.krs * 2) + (rate * 2) + ((slot->regs.fns >> 9) & 1);
      result = (8 ^ slot->regs.oct) + (result - 8);
   }

   if (result <= 0)
      return 0;

   if (result >= 0x3c)
      return 0x3c;

   return result;
}
・・・省略・・・
```

最初に slot->regs.krs による条件分岐があります。これはスロットレジスタの KRS フィールドです。0xf と比較し、条件が成立する場合は KRS を使用せず、不成立の場合は使用している事から、 KRS が 0xf の場合は KRS 自体が無効なのだと思われます。 KRS について詳しくは後述します。

ここでは、 KRS は 0xf であるとして読み進めることにします。その場合、第 2 引数 rate すなわち AR の値を 2 倍した値を変数 result へ設定します。

そして、残る 2 つの if 文で最小値を 0、最大値を 0x3c となるように戻り値を調整しています。

以上から、ここでは get_rate() は「第 2 引数の値を 2 倍し、最小値 0・最大値 0x3c の範囲内で返す」関数であるとしておきます。

リスト 3.2 に戻ります。

get_rate() の戻り値は変数 rate に格納されます。

続いて need_envelope_step() を呼び出します。第 1 引数には変数 rate を指定しています。第 2 引数に slot->state.sample_counter を指定していますが、これが何であるのかを

これを使用しているコードを見て確認します。同様に yabause/src/scsp.c 内を調べると、リスト 3.4 の処理が見つかりました。

▼リスト 3.4: yabause/src/scsp.c: op7(), keyon()

```
・・・省略・・・
//sound stack write
void op7(struct Slot * slot, struct Scsp*s)
{
   u32 previous = s->sound_stack[slot->state.num + 32];
   s->sound_stack[slot->state.num + 32] = slot->state.output;
   s->sound_stack[slot->state.num] = previous;

   slot->state.sample_counter++;
   slot->state.lfo_counter++;
}
・・・省略・・・
void keyon(struct Slot * slot)
{
   if (slot->state.envelope == RELEASE)
   {
      slot->state.envelope = ATTACK;
      slot->state.attenuation = 0x280;
      slot->state.sample_counter = 0;
      slot->state.step_count = 0;
      slot->state.sample_offset = 0;
      slot->state.envelope_steps_taken = 0;

      if (new_scsp.debug_mode)
         scsp_debug_add_instrument(slot->regs.sa);
   }
   //otherwise ignore
}
・・・省略・・・
```

パイプライン処理する `op<番号>()` の関数は `op7()` まであります。なので、 `op7()` はパイプライン処理の最後の関数です。 `op7()` の実装から、 `slot->state.sample_counter` はパイプライン処理を 1 周するとインクリメントされるのだと分かります。また、 `keyon()` は関数名の通り、キーオン時に呼び出されます。この実装内容から `slot->state.sample_counter` はキーオン時に 0 で初期化される事が分かります。

リスト 3.2 に話を戻すと、この `slot->state.sample_counter` と `get_rate()` で得られた `rate` を渡して `need_envelope_step()` を呼び出しています。この関数について詳しい説明は省略しますが、戻り値を格納した変数 `need_step` の変数名はその後の使われ方からも分かる通り、その後の処理を実施するか否かを示す 1 あるいは 0 の値を返します。

そして、 `need_step` が 1 の時、コード内を太字で示す通り、 **`slot->state.attenuation` という変数の減算**が行われます。**“attenuation” は日本語で「減衰」で、音量の減衰量を示しています。これを減算しているという事で、音量を上げています**。そして続く if 文で、**この減衰量が 0 になった時、 `DECAY1` へ遷移**しています。

以降、「Decay 1」・「Decay 2」・「Release」についても `op4()` のコードから見ていきます。「Attack」の説明で引用した際にはこれらの 3 状態の処理は省略していましたが、それらを示すとリスト 3.5 の通りです。

▼リスト 3.5: yabause/src/scsp.c: scsp_slot_write_byte()

```
・・・省略・・・
void op4(struct Slot * slot)
{
   ・・・省略・・・
   if (slot->state.envelope == ATTACK)
      ・・・省略・・・
   else if (slot->state.envelope == DECAY1)
   {
      do_decay(slot,slot->regs.d1r);

      if ((slot->state.attenuation >> 5) >= slot->regs.dl)
         change_envelope_state(slot, DECAY2);
   }
   else if (slot->state.envelope == DECAY2)
      do_decay(slot, slot->regs.d2r);
   else if (slot->state.envelope == RELEASE)
      do_decay(slot, slot->regs.rr);
}
・・・省略・・・
```

`DECAY1` ・ `DECAY2` ・ `RELEASE` のいずれの場合も `do_decay()` を呼び出しています。これは**第 2 引数で指定された値に応じて `slot->state.attenuation` へ加算**します。それぞれの場合の**第 2 引数に、スロットレジスタの `D1R` ・ `D2R` ・ `RR` を与えている事から、それらの値に応じて音量を下げていく**事が分かります。

そして、 `DECAY1` の場合の if ブロックの中に、スロットレジスタの `DL` との比較を行っている if 文があります。**減衰量が `DL` に達したら `DECAY2` へ遷移**するようにしています。なお、`slot->state.attenuation` を 5 ビット右シフトした値と比較していることから、 **`DL` には減衰量を 5 ビット右シフトした値を設定**する事が分かります。

`RELEASE` へ遷移する処理が `op4()` にありません。押下したキーを離したときに遷移するため、 **`RELEASE` への遷移はキーオフの処理の中にあります** (リスト 3.6)。

▼リスト 3.6: yabause/src/scsp.c: keyoff()

```
・・・省略・・・
void keyoff(struct Slot * slot)
{
   change_envelope_state(slot, RELEASE);
}
・・・省略・・・
```

以上のように、**SCSP の EG では「Attack」・「Decay 1」・「Decay 2」・「Release」の 4 状態で音量変化を行います**。

第 4 章

LFO(低周波オシレータ)

この章では、LFO(低周波オシレータ) という機能について解説し、それが SCSP でどのように実現されているのかを解説します。

4.1 LFO(低周波オシレータ) とは

LFO は “Low Frequency Oscillator(低周波オシレータ)” の略です。数 Hz から 100Hz 程度の低い周波数で音量 (振幅) や音の高さ (周波数) を変化させる (揺らす) 効果があります。

LFO についても、Yamaha が公開している以下のページを参考にしています。

- ヤマハ | もう一つの強力な武器＝ LFO - シンセサイザー入門
  - https://jp.yamaha.com/products/contents/music_production/guide_to_synth/006/index.html

4.2 LFO デモ

どういうものかは実際に聞いてもらった方が分かりやすいかと思います。自作シンセサイザーでの LFO のデモを以下で公開していますので、良ければ聞いてみてください。

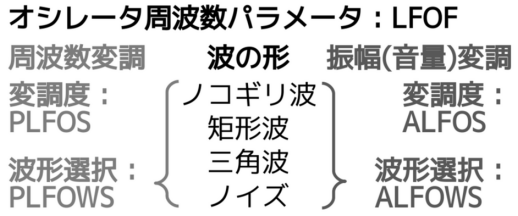
- セガサターン自作シンセサイザー　LFO デモ - YouTube
  - https://youtu.be/5oEV5af744o

SCSP が持つ LFO の機能で周波数 (音の高さ) と振幅 (音量) を、それぞれ数 Hz〜100Hz 程度の周波数で変化させています。変化させる際の低周波の波の形も設定でき、「ノコギリ波」・「矩形波」・「三角波」・「ノイズ」の 4 つを試しています。「ノイズ」は少し分かり難いですが、周波数や振幅を変化にノイズを掛ける、という感じです。

4.3　SCSP における LFO 設定について

LFO の設定もスロットレジスタで行います。例によって Yabause Wiki の SCSP のページを見ると、**スロットレジスタの先頭から +0x12 バイトの位置の 16 ビットの領域に `LFORE` (LFO reset)・ `LFOF` (LFO frequency)・ `PLFOWS` (pitch LFO wave select)・ `PLFOS` (pitch LFO displacement)・ `ALFOWS` (amplitude LFO wave select)・ `ALFOS` (amplitude LFO displacement) という LFO に関する 6 つのフィールドがある**事が分かります。

前節のデモは、これらのフィールドの値を変更するもので、フィールド名との対応を示すと図 4.1 の通りです。



▲図 4.1: SCSP の LFO 設定

4.4　SCSP の LFO 機能をコードから見てみる

LFO に関するスロットレジスタの 6 つのフィールドがどのように機能するのかをコードから見ていきます。

まず、例によって `scsp_slot_write_byte()` を見て、これらのフィールドへの書き込みがどの変数への設定に対応しているかを確認します (リスト 4.1)。

▼リスト 4.1: yabause/src/scsp.c: scsp_slot_write_byte()

```
・・・省略・・・
void scsp_slot_write_byte(struct Scsp *s, u32 addr, u8 data)
{
    ・・・省略・・・
   switch (offset)
   {
    ・・・省略・・・
   case 18:
      slot->regs.re = (data >> 7) & 1;
      slot->regs.lfof = (data >> 2) & 0x1f;
      slot->regs.plfows = data & 3;
      break;
   case 19:
      slot->regs.plfos = (data >> 5) & 7;
```

```
      slot->regs.alfows = (data >> 3) & 3;
      slot->regs.alfos = data & 7;
      break;
   ・・・省略・・・
   }
}
・・・省略・・・
```

LFORE だけは slot->regs.re となっていますが、それ以外は slot->regs.<フィールド名> である事が分かります。

以降では、それぞれの変数を使用している箇所から、それぞれのフィールドがどのように機能するのかを見ていきます。ただ、 slot->regs.re だけはそれを使用している箇所が Yabause 内に無く、どうやら Yabause ではこのフィールドは機能していないようなので飛ばすことにします。

♣ LFOF・PLFOWS・PLFOS

slot->regs.lfof や slot->regs.plfo* を使用している箇所を探すとリスト 4.2 が見つかりました。

▼リスト 4.2: yabause/src/scsp.c: op1()

```
・・・省略・・・
void op1(struct Slot * slot)
{
   ・・・省略・・・
   if (slot->state.lfo_counter % lfo_step_table[slot->regs.lfof] == 0)
   {
      slot->state.lfo_counter = 0;
      slot->state.lfo_pos++;

      if (slot->state.lfo_pos > 0xff)
         slot->state.lfo_pos = 0;
   }

   if (slot->regs.plfows == 0)
      plfo_val = plfo.saw_table[slot->state.lfo_pos];
   else if (slot->regs.plfows == 1)
      plfo_val = plfo.square_table[slot->state.lfo_pos];
   else if (slot->regs.plfows == 2)
      plfo_val = plfo.tri_table[slot->state.lfo_pos];
   else if (slot->regs.plfows == 3)
      plfo_val = plfo.noise_table[slot->state.lfo_pos];

   plfo_shifted = (plfo_val << slot->regs.plfos) >> 2;
   ・・・省略・・・
   slot->state.waveform_phase_value += (phase_increment + plfo_shifted);
}
・・・省略・・・
```

第 2 章で OCT ・ FNS を説明する際にも登場した op1() です。その際はほとんど省略してい

ましたがリスト 4.2 の様になっています。

`slot->regs.lfof` は `lfo_step_table` という配列の添字に使われています。これはリスト 4.3 の様に定義されています。

▼リスト 4.3: yabause/src/scsp.c: lfo_step_table[]

```
・・・省略・・・
//samples per step through a 256 entry lfo table
const int lfo_step_table[0x20] = {
   0x3fc,//0
   0x37c,//1
    ・・・省略・・・
   0x002,//0x1e
   0x001,//0x1f
};
・・・省略・・・
```

コメント行の内容から、**`LFOF` はサンプル数に対応**していて、配列 `lfo_step_table` でその対応付けを行っている事が分かります。また、配列の長さが 0x20 であることから、**`LFOF` は 0x00 から 0x1f** である事も分かります。

リスト 4.2 に戻ると、配列 `lfo_step_table` に `LFOF` を添字に指定して取得した値 (サンプル数) で、`slot->state.lfo_counter` との剰余を計算していました。この剰余が 0 の場合、if のブロック内に入り、`slot->state.lfo_counter` に 0 が設定されます。この変数は何かというと、yabause/src/scsp.c 内を調べてみると、`op7()` でインクリメントしています (リスト 4.4)。

▼リスト 4.4: yabause/src/scsp.c: op7()

```
・・・省略・・・
void op7(struct Slot * slot, struct Scsp*s)
{
    ・・・省略・・・
   slot->state.lfo_counter++;
}
・・・省略・・・
```

すなわち、パイプライン処理しているサウンドのオペレーション 1 周期毎にインクリメントしています。この値が `LFOF` によって決まるサンプル数と一致した時 (剰余が 0 の時)、if ブロックの中に入る、という訳です。

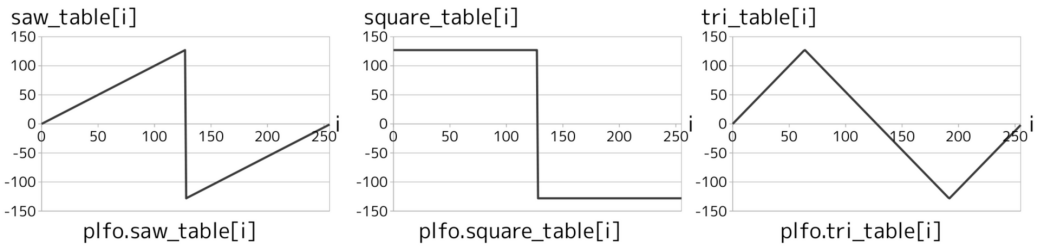
そして、if ブロックの中では `slot->state.lfo_counter` のゼロクリアと `slot->state.lfo_pos` のインクリメントを行っています。直後の if 文で `slot->state.lfo_pos` が 0xff より大きい時、ゼロクリアしていることから、この変数は 0x00 から 0xff である事が分かります。

それに続いて `slot->regs.plfows` すなわちスロットレジスタの `PLFOWS` を用いた 4 つの if 文があります。`PLFOWS` の値に応じて `plfo.saw_table[]` ・ `plfo.square_table[]` ・ `plfo.tri_table[]` ・ `plfo.noise_table[]` の配列のいずれかから取得した値を変数 `plfo_val` に代入しています。これらがどんな配列かはリスト 4.5 を見ると分かります。

▼リスト 4.5: yabause/src/scsp.c: fill_plfo_tables()

```
・・・省略・・・
void fill_plfo_tables()
{
   int i;

   //saw
   for (i = 0; i < 256; i++)
   {
      if (i < 128)
         plfo.saw_table[i] = i;
      else
         plfo.saw_table[i] = -256 + i;
   }

   //square
   for (i = 0; i < 256; i++)
   {
      if (i < 128)
         plfo.square_table[i] = 127;
      else
         plfo.square_table[i] = -128;
   }

   //triangular
   for (i = 0; i < 256; i++)
   {
      if (i < 64)
         plfo.tri_table[i] = i * 2;
      else if (i < 192)
         plfo.tri_table[i] = 255 - (i * 2);
      else
         plfo.tri_table[i] = (i * 2) - 512;
   }

   //noise
   for (i = 0; i < 256; i++)
   {
      plfo.noise_table[i] = rand() & 0xff;
   }
}
・・・省略・・・
```

コメント行からも分かる通り、それぞれの配列には「ノコギリ波」・「矩形波」・「三角波」・「ノイズ」の波形を設定しています。`plfo.saw_table[]`・`plfo.square_table[]`・`plfo.tri_table[]`の波形をグラフにすると図 4.2 の通りです。(`plfo.noise_table[]`は単に乱数を設定しているだけなので省略します。)



▲図 4.2: plfo のノコギリ波・矩形波・三角波

リスト 4.2 の `op1()` に話を戻すと、**スロットレジスタの `PLFOWS` によって周波数の変化をかける LFO の波形が決まる**訳です。**いずれかの配列から取得した値を `plfo_val` へ格納し、それを `slot->regs.plfos` すなわちスロットレジスタの `PLFOS` の値に応じて左シフトし、`plfo_shifted` へ格納します。それを `slot->state.waveform_phase_value` へ加算することで周波数に変化をかける**訳です。

♣ ALFOWS・ALFOS

`slot->regs.alfo*` を使用している箇所を探すと `04_op5` が見つかりました。

▼リスト 4.6: yabause/src/scsp.c: op5()

```
・・・省略・・・
void op5(struct Slot * slot)
{
   if (slot->state.attenuation > 0x3bf)
       ・・・省略・・・
   else
   {
       ・・・省略・・・
      if (slot->regs.alfows == 0)
         alfo_val = alfo.saw_table[slot->state.lfo_pos];
      else if (slot->regs.alfows == 1)
         alfo_val = alfo.square_table[slot->state.lfo_pos];
      else if (slot->regs.alfows == 2)
         alfo_val = alfo.tri_table[slot->state.lfo_pos];
      else if (slot->regs.alfows == 3)
         alfo_val = alfo.noise_table[slot->state.lfo_pos];

      lfo_add = (((alfo_val + 1)) >> (7 - slot->regs.alfos)) << 1;

      sample = apply_volume(slot->regs.tl, slot->state.attenuation + lfo_add, slo›
›t->state.output);
      slot->state.output = sample;
   }
}
・・・省略・・・
```

`ALFOWS` や `ALFOS` も、変化をかける対象が音量だという違いのみで、処理の流れは `PLFOWS` ・

`PLFOS` **と同様**であることが見て分かるかと思います。

`slot->regs.alfows` **(スロットレジスタの** `ALFOWS` **) で LFO の波形を選択し、波形別の配列に従って変化量を取得して、それを** `slot->regs.alfos` **(スロットレジスタの** `ALFOS` **) に応じてシフトし、その値を** `apply_volume()` **で音量へ反映**させています。

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、セガサターンの音源 IC である SCSP がシンセサイザーも作れる程に高機能であることを紹介しました。本書では紹介しきれませんでしたが、SCSP には他にも「音色」を作る機能として FM 変調の機能があったり、第 1 章でも紹介した通り DSP も搭載しているので信号処理としては他にも色々な事ができそうです。それに加えて、MC68EC00 という 16 ビット CPU も搭載しているのですから、サウンド周りだけで相当豪華な作りであったと思います。

ただ、当時のゲームソフトではこのような SCSP の機能がフル活用されることはあまり無かったようです[*1]。というのも、PCM が使えるということは、PCM で BGM や SE を用意しておけば良いため、通常のゲームソフトで SCSP の機能が活用されることはあまり無かったのかと思われます。

このように、セガサターンはある部分ではゲームハードとしてはオーバースペックな所もありますが、それが筆者の「フルスクラッチでハードを直接制御する」プログラミングではやりがいのある部分だと思っています。

[*1] 「サターンミュージックスクール」のような音楽用のソフトは別ですが。

セガサターン自作シンセサイザー読本

シンセサイザーだって作れる
セガサターンの音源 IC を
エミュレータのコードで解説

2022 年 8 月 13 日　ver 1.0 （コミックマーケット 100 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://twitter.com/yohgami)
印刷所　日光企画

(powered by Re:VIEW Starter)