# セガサターンと
# MIDI で通信する本

— MIDI キーボードや PC と通信、実行バイナリローダの作成 —

［著］大神祐真

技術書典 12 新刊
2022 年 1 月 22 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、セガサターンと MIDI で接続して通信を行ってみます。セガサターンには、当時、本体へ MIDI の接続を行うデバイスがあり、それを使うことで MIDI キーボードの接続が行えました。本書ではその制御方法を確認し、実際に制御を行って、MIDI キーボードや PC と通信してみます。

## 本書の構成

本書は以下の 4 章構成です。

**第 1 章「セガサターンと MIDI について」**

MIDI は電子楽器を接続するインタフェースです。この章ではセガサターンに MIDI 接続する方法を紹介します。また、ソフトウェア側からどのように制御できるのかをエミュレータのソースコードで確認します。

**第 2 章「MIDI キーボードの MIDI メッセージを読む」**

MIDI の通信は MIDI メッセージでやり取りします。この章では、まず MIDI キーボードの MIDI メッセージを読んでみます。最初に PC と繋いで MIDI メッセージを読んでみた後、今度はセガサターンと接続して読んでみます。

**第 3 章「PC と繋ぎ MIDI メッセージを送受信」**

いよいよセガサターンと PC を MIDI で接続し通信してみます。この章では、簡単な MIDI メッセージとして MIDI キーボードのキー押下時に送信されるノート・オンの MIDI メッセージの送受信を試してみます。

**第 4 章「簡単なプロトコルで PC から実行バイナリをロード・実行」**

前章で確認した MIDI メッセージにデータを埋め込むことで PC からセガサターンへデータ送信を行ってみます。この章では、まず簡単なプロトコルを作り、それを用いて PC から受信した実行バイナリをメモリへロードし実行するローダを作ってみます。

## 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
# セガサターンと MIDI について

この章では、セガサターンの MIDI 接続を紹介し、MIDI 通信をソフトウェアでどのように行うのかをエミュレータのソースコードで説明します。

## 1.1 セガサターンの MIDI 接続

### ♣ MIDI とは

「MIDI(Musical Instrument Digital Interface)」は電子楽器を接続するインタフェースです。演奏情報を「MIDI メッセージ」という形式でやり取りする仕組みで、楽譜をやり取りしているようなものです。これを時間の情報と共に記録した「MIDI ファイル」は、「WAVE ファイルよりファイルサイズが小さくて済む音楽ファイル」としてご存知の方も多いかと思います。

### ♣ MIDI 接続の仕方

当時、セガサターン背面の通信拡張端子 (図 1.1)[*1]を通して MIDI を接続できるようにするケーブル (図 1.2) があり、これを使うと MIDI をセガサターンへ接続できます。


▲図 1.1: 本体背面の通信拡張端子

[*1] 主にセガサターン同士で通信するための端子で、専用の通信ケーブルもありました。


▲図 1.2: 通信拡張端子を通して MIDI を接続できるようにするケーブル

MIDI キーボードを接続すると図 1.3 の通りです[*2]。


▲図 1.3: MIDI キーボードを繋いだ様子

USB-MIDI の変換ケーブルを使えば PC との接続も可能です。これにより、MIDI という伝送路でセガサターンと PC の通信が可能という訳です。

なお、セガサターン上で自作のプログラムを動作させる方法について詳しい説明等は省略します。[*3]

## 1.2　エミュレータのコードを読んでわかる MIDI 通信

ハードウェア的な接続は前節の通りですが、ソフトウェア的にセガサターン上で MIDI をどのように扱うのかを見ていきます。

[*2] 写真では MIDI に関するケーブルしか繋いでいないですが、動作のためには電源ケーブルや映像出力用のケーブル、コントローラなども接続する必要があります。以降も MIDI の接続を示す写真では MIDI 以外のケーブルは省いています。

[*3] 参考になるページだけ紹介しておきます。`https://www.mdnomad.com/game/ss/environ/ssvirtual_cd-rom` 筆者はこちらのページでも紹介されている「Satiator」という物を使っています。

## ♣ エミュレータについて

ここでは、これまでのセガサターン関連の同人誌[*4]と同様に、エミュレータ「Yabause[*5]」のソースコードを通して理解していきます。

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。

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

本書は、2022 年 1 月時点の最新である下記のコミット時点を対象にします。

- 7e38821d Fixed gtk port compilation

## ♣ yabause/src/scsp.c

セガサターンには「SCSP(Saturn Custom Sound Processor)」というサウンド用 IC が搭載されており、MIDI の機能もこの IC が持っています。その SCSP を実装しているのが yabause/src/scsp.c です。

このソースファイルを見ると、冒頭にリスト 1.1 のコメントがあります。

▼リスト 1.1: yabause/src/scsp.c: 40 行目辺り

```
 ・・・省略・・・
// Common Control Register (CCR)
//
//      $+00      $+01
// $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
// $404 ---1 2345 6666 6666 1:MOFULL out fifo full 2:MOEMP empty 3:MIOVF overflow›
› 4:MIFULL in 5:MIEMP 6:MIBUF
// $406 ---- ---- 1111 1111 1:MOBUF midi output data buffer
 ・・・省略・・・
```

リスト 1.1 には SCSP の制御レジスタのメモリマップが書かれていて、太字で示したレジスタあるいはビットが MIDI の通信で使うものです。

リスト 1.1 は見方が少し難しいのですが、各行の `1:` の左側の部分がアドレスとビットフィールドを、 `1:` を含む右側の部分がビットフィールドで示した各ビットの内容を示しています。例えば `$406 ---- ---- 1111 1111 1:MOBUF midi output data buffer` の行は以下を示しています。

**アドレス 0x406+00(すなわち 0x406)**

　　未使用

**アドレス 0x406+01(すなわち 0x407)**

　　「MOBUF」という 8 ビットのレジスタがある

[*4] 2022 年 1 月現在、「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」があります。詳しくは筆者ウェブサイト (`http://yuma.ohgami.jp`) をご覧ください。

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

そして、リスト 1.1 で太字で示したレジスタ/ビットの意味について、MIDI の通信はシリアル通信なので、シリアル通信で必要となるレジスタ/ビットが用意されており、以下の通りです。

**MIBUF - 受信レジスタ**

このレジスタを読み出すと受信バッファ (FIFO) からデータを取り出す事ができる。

**MIEMP - 受信バッファのステータスビット**

受信バッファが空だと 1、そうでなければ 0。

**MIFULL - 受信バッファのステータスビット**

受信バッファに空きが無ければ 1、空きがあれば 0。

**MIOVF - 受信バッファのステータスビット**

受信バッファがオーバーフローすると 1、そうでなければ 0。

**MOBUF - 送信レジスタ**

このレジスタへ書き込むと送信バッファ (FIFO) にデータを追加する事ができる。

**MOEMP - 送信バッファのステータスビット**

送信バッファが空だと 1、そうでなければ 0。

**MOFULL - 送信バッファのステータスビット**

送信バッファに空きが無ければ 1、空きがあれば 0。

なお、受信バッファ ( `MIBUF` )・送信バッファ ( `MOBUF` ) のサイズ (段数) はそれぞれ 4 バイト (4 段) であることがリスト 1.2 を見ると分かります。

▼リスト 1.2: yabause/src/scsp.c: struct scsp_t

```
typedef struct scsp_t
{
   ・・・省略・・・
  u8 midinbuf[4];       // midi in buffer
  u8 midoutbuf[4];      // midi out buffer
   ・・・省略・・・
```

加えて、SCSP のレジスタはキャッシュスルー[*6]の場合 `0x25B00000` にマップされます[*7]。そのため、アドレスの上位 20 ビットを補うと、メモリマップとしては表 1.1 の通りです。

▼表 1.1: MIDI 関連レジスタの 32 ビットメモリマップ

| アドレス | +00 | +01 |
|---|---|---|
| 0x25B00404 | ステータスビット群 | MIBUF |
| 0x25B00406 | 未使用 | MOBUF |

以上のレジスタとステータスビットを使用して、シリアル通信で MIDI のメッセージの送受信を行います。

[*6] キャッシュを経由しないアクセスで、制御レジスタへのアクセスの際はこの方が良いです。なおこれは SH7604 の機能で、マップするアドレスに応じてキャッシュを経由するかしないかが変わる、というものです。

[*7] 詳しくは「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。

**ステータスビットは MIBUF とまとめて読み出す事**

MIBUF から読み出す際、事前にステータスビットを読んでバッファにデータが存在するかを確認すると思いますが、その際、ステータスビット群だけをバイトアクセスで読み出さない方が良いです。

ステータスビット群だけを取得するために `0x25B00404` をバイトアクセスで 1 バイト読み出しても、内部的に MIBUF も読み出されてしまうようで、MIBUF が 1 バイト失われてしまう事を実機で確認しています。

そのため、`0x25B00404` をワードアクセスで MIBUF 含む 2 バイトを読み出し、ステータスビットを確認して受信バッファが空で無ければ読み出した MIBUF の値を使う、とすると良いです。

# 第 2 章
# MIDI キーボードの MIDI メッセージを読む

この章では、MIDI キーボードが出力する MIDI メッセージを読んでみます。まずは PC で試し、次にセガサターン上で同じことを試します。

## 2.1 PC から読んでみる

PC とセガサターンを繋ぐ前に、そもそも MIDI でやり取りするメッセージはどんなものなのかを知るために、MIDI キーボードが送信する簡単な MIDI メッセージを読んでみます。

それではまず PC と接続して読んでみます。接続した様子は図 2.1 の通りです。


▲図 2.1: PC と MIDI キーボードを繋いだ様子

なお、本書では PC の OS は Linux[*1]を想定します。また、USB-MIDI 変換[*2]を使って MIDI キーボードを PC と接続しています。これはセガサターンと PC を MIDI で接続する際も使います。

接続すると、/dev/snd/に midi で始まるファイル名のデバイスファイルが生成されます (リスト 2.1)。

▼リスト 2.1: デバイスファイル例

```
$ ls /dev/snd/midi*
/dev/snd/midiC1D0
```

本書では/dev/snd/midiC1D0 というデバイスファイルが生成されたものとして説明します。

そして、接続した MIDI キーボードの電源を入れ、PC 側でリスト 2.2 のようにデバイスファイルを読み出します。

▼リスト 2.2: デバイスファイル読み出し例

```
$ hexdump -Cv /dev/snd/midiC1D0
00000000  f8 f8 f8 f8 f8 f8 f8 f8  f8 f8 f8 f8 f8 f8 f8 f8  |................|
00000010  f8 f8 f8 f8 f8 f8 f8 f8  f8 f8 f8 f8 f8 f8 f8 f8  |................|
...
```

YAMAHA の CBX-K1 で試すと、何も操作していない時、上記のように 0xf8 が繰り返し出力されていました。これは「タイミング・クロック」と呼ばれる MIDI メッセージで、同期をとることができるように送られています[*3](表 2.1)。MIDI の規格書に「送信側は演奏時でなくても，タイミング・クロック（F8H）を送信し続けることはさしつかえないが」という説明がされている[*4]ため、これを送信することは義務付けられているものでは無いようです。そのため、MIDI のデバイスによってはこのタイミング・クロックは出ていないものもあるかもしれません。

▼表 2.1: MIDI メッセージ:タイミング・クロック

| フォーマット | 意味 |
|---|---|
| 0xf8 | 同期をとることができるように 4 分音符あたり 24 の割合で送られる。 |

それでは、キーボードの適当なキーで何度か押下を繰り返してみます。

▼リスト 2.3: 最も左の白鍵で押下を繰り返した例

```
...
000000b0  f8 f8 f8 f8 f8 f8 f8 f8  f8 f8 f8 90 30 51 f8 f8  |...........0Q..|
000000c0  f8 f8 90 30 00 f8 f8 f8  f8 90 30 39 f8 f8 f8 f8  |...0......09....|
000000d0  90 30 00 f8 f8 f8 f8 f8  90 30 3b f8 f8 f8 f8 90  |.0.......0;.....|
```

*1 筆者は「Debian GNU/Linux 11 (bullseye)」を使用しています。

*2 何でも良いかと思いますが、参考として、筆者は Roland の「UM-ONE mk2」というものを使っています。

*3 MIDI に関する説明は「MIDI 1.0 規格書」を参照しています。2022 年 1 月現在は MIDI 2.0 も公開されていますが、セガサターンの 1994 年当時は MIDI 1.0 なのでこれを参照しています。MIDI 1.0 規格書：http://amei.or.jp/midistandardcommittee/MIDIspcj.html

*4 「MIDI 1.0 規格書（日本語版 98.1)」P.2-30

```
...
```

筆者の環境ではリスト 2.3 の出力が得られました。

`0x90 30 XX` という出力が繰り返し登場しています[*5]。これらはキーを押下した/離した際に送信される「ノート・オン」/「ノート・オフ」と呼ばれる MIDI メッセージです (表 2.2)。

▼表 2.2: MIDI メッセージ:ノート・オン/オフ

| フォーマット | 意味 |
|---|---|
| `0x90 kk vv` | ノート・オン。kk のキーが vv のベロシティ[*6]で押下された事を示す。 |
| `0x90 kk 00` | ノート・オフ。kk のキーが離された事を示す。 |

なお、ここまで説明してきませんでしたが、「MIDI メッセージ」は「ステータス・バイト (1 バイト) とデータ・バイト (0 あるいは数バイト)」という構成になっています。タイミング・クロックの場合、`0xf8` というステータスバイトだけの MIDI メッセージで、ノート・オン/オフの場合、`0x90` のステータス・バイトと `kk vv` あるいは `kk 00` のデータ・バイトという構成になっています。ここで、「ステータス・バイトは最上位ビットが 1」、「データ・バイトは最上位ビットが 0」と決められており、そのため、kk と vv は共に最上位ビットが 0 です。

表 2.2 を踏まえてリスト 2.3 の太字箇所を見ると、以下のキー操作が行われていたと読み取ることができます。

1. `90 30 51` : `0x30` のキー (最も左の白鍵) を `0x51` のベロシティで押下した。
2. `90 30 00` : `0x30` のキーを離した。
3. `90 30 39` : `0x30` のキーを `0x39` のベロシティで押下した。
4. `90 30 00` : `0x30` のキーを離した。
5. `90 30 3b` : `0x30` のキーを `0x3b` のベロシティで押下した。

なお、これは「キー押下にのみベロシティがあるキーボード」の場合です。「ベロシティが無いキーボード」や「キー押下時/離した時の両方にベロシティがあるキーボード」の場合、MIDI メッセージのフォーマットが少し変わります。詳しくは MIDI の規格書を参照してください。

以上で、PC で MIDI キーボードが出力する簡単な MIDI メッセージとして「ノート・オン」・「ノート・オフ」のメッセージを読んでみることができました。

## 2.2　セガサターンから読んでみる

それでは同じことを、今度はセガサターンに MIDI キーボードを繋いで行ってみます。セガサターンと MIDI キーボードの接続については第 1 章で紹介した通りです。

[*5] 最後の 1 バイトは `90` から先が見切れていますが、同様に `0x90 30 XX` という出力になっています。
[*6] どのくらい強くそのキーが押されたかを示す値です。

## ♣ MIBUF をダンプするプログラム

セガサターン上で動作させるプログラムは「echo や dd を駆使してマシン語を吐くシェルスクリプト」という独特な方法で作っているため、C 言語ライクな表現へ置き換えて紹介します。

セガサターン上で動作する筆者の実験プログラムは以下の GitHub リポジトリで公開しています。もし興味があれば見てみてください。

- https://github.com/cupnes/sh_ss_test/
  - 各実験プログラムがディレクトリで分かれています。
  - この節のプログラムは「080_dump_mibuf」ディレクトリです。

それでは、セガサターン側で動作させるプログラムを紹介します (リスト 2.4)。

▼リスト 2.4: MIBUF をダンプするプログラム

```
#define MIOSTAT_MIBUF   *((unsigned short *)0x25B00404)
#define MIOSTAT_BIT_MIEMP       0x01

void main(void)
{
        /* 各種デバイスの初期化(説明省略) */

        /* MIBUFの取得と半角スペース区切りの出力を繰り返す */
        while (1) {
                unsigned char miostat;
                unsigned char mibuf;
                unsigned short tmp;

                /* MIEMP == 0 になるのを待つ
                 * (MIEMP == 1の間繰り返す) */
                do {
                        /* MIOSTATとMIBUFを同時に取得 */
                        tmp = MIOSTAT_MIBUF;
                        /* MIOSTATのみ抽出 */
                        miostat = tmp >> 8;
                } while (miostat & MIOSTAT_BIT_MIEMP);

                /* MIOSTATと同時に取得したMIBUFを変数に代入し出力 */
                mibuf = (unsigned char)tmp;
                printf("%02X ", mibuf);
        }
}
```

第 1 章で紹介したアドレスから MIOSTAT と MIBUF を取得し、MIEMP のビットが 0 なら (MIBUF が空でないなら)、取得した MIBUF を出力しています。

## ♣ 実行結果

前項のプログラムをセガサターン実機で実行し、PC に繋いだ時と同様に最も左の白鍵で押下を繰り返しました。

TV 画面に出力された様子は図 2.2 の通りです。


▲図 2.2: セガサターン実機で動作させた様子 (TV 画面の写真の一部分)

ノート・オンとノート・オフの MIDI メッセージは四角の枠を付けて示しています[*7]。これを見ると、`0x30` のキー (最も左の白鍵) を `0x4e` のベロシティで押下した後、`0x30` のキーを離した事が読み取れます。これで、セガサターンで MIDI メッセージを受信することができました。

なお、`0xfe` というものも出ていますが、これは「アクティブ・センシング」という MIDI メッセージです (表 2.3)。

▼表 2.3: MIDI メッセージ:アクティブ・センシング

| フォーマット | 意味 |
|---|---|
| `0xfe` | 主に何もデータ送信が無い時に MIDI ケーブルに何らかの問題が無い事を確認するために送信されるメッセージ。受信側はこれを受け取ると、その後 300ms 以内に新たに何らかのメッセージが送られて来なければ MIDI ケーブルに問題が起きたと判断し、各種の発音処理を止める。 |

PC と接続した時にこのメッセージが現れていなかったのは、どうやら USB-MIDI 側で処理されていたようです[*8]。

[*7] 四角の枠は画像編集ソフトで付けたものです。

[*8] セガサターンと PC を USB-MIDI を通して MIDI で繋ぎ、セガサターン側から任意のバイト列を MIDI で送信するプログラムを動作させて確認しました。

# 第 3 章
# PC と繋ぎ MIDI メッセージを送受信

この章では、セガサターンを PC と繋ぎ、相互に MIDI メッセージの送受信を試してみます。

## 3.1 PC から送信した MIDI メッセージをセガサターンで受信

PC から簡単な MIDI メッセージを送信し、セガサターンで受信してみます。なお、セガサターン側は第 2 章で MIDI キーボードからの MIDI メッセージの受信に使用したプログラムを再度使用します。

### ♣ PC から MIDI メッセージを送信

まず、PC とセガサターンを USB-MIDI を使用して MIDI 接続します (図 3.1)。


▲図 3.1: PC とセガサターンを USB-MIDI を通して繋いだ様子

そして、PC からはリスト 3.1 のように MIDI メッセージを送信します。

▼リスト 3.1: PC から MIDI メッセージを送信する

```
$ echo -en '\x90\x01\x40' >/dev/snd/midiC1D0
```

送信する MIDI メッセージはノート・オンですが、第 2 章とはキーの番号とベロシティを変えてみます。

### ♣ 実行結果

実行結果を図 3.2 に示します。


▲図 3.2: セガサターン実機で動作させた様子 (TV 画面の写真の一部分)

図 3.2 は、前項で紹介した `echo` コマンドを何度か繰り返した際の一部分です。

ちゃんと `0x90 01 40` が受信できている事が確認できます。

また、`0xfe`(アクティブ・センシング) を繰り返し受信しています。これは `echo` で送信していなかった事から、やはりこの USB-MIDI はアクティブ・センシングを自ら処理していると考えられます。

## 3.2　セガサターンから送信した MIDI メッセージを PC で受信

次に、セガサターンから簡単な MIDI メッセージを送信し、PC で受信してみます。

なお、PC 側は第 2 章で MIDI キーボードから MIDI メッセージの受信を確認した際と同様に、デバイスファイルを `hexdump` コマンドで読み出す事で、受信した MIDI メッセージの表示を行います。

### ♣ ノート・オンを送信し続けるプログラム

確認のためにノート・オンを送信し続けるプログラムをセガサターン上で動作させます。動作させるプログラムは前章でも紹介した GitHub リポジトリの「082_send_note_on」のものです。例によって C 言語ライクな表現で書いたもので説明します (リスト 3.2)。

▼リスト 3.2: 0x90 01 40 を送信し続けるプログラム

```
#define MIOSTAT *((unsigned char *)0x25B00404)
#define MIOSTAT_BIT_MOFULL      0x10
#define MOBUF   *((unsigned char *)0x25B00407)
```

```
void main(void)
{
        /* 各種デバイスの初期化(説明省略) */

        /* 0x90 01 40を繰り返し送信し続ける */
        while (1) {
                # MOFULL == 0 になるのを待つ
                while (MIOSTAT & MIOSTAT_BIT_MOFULL);

                # 0x90を送信
                MOBUF = 0x90;

                # MOFULL == 0 になるのを待つ
                while (MIOSTAT & MIOSTAT_BIT_MOFULL);

                # 0x01を送信
                MOBUF = 0x01;

                # MOFULL == 0 になるのを待つ
                while (MIOSTAT & MIOSTAT_BIT_MOFULL);

                # 0x40を送信
                MOBUF = 0x40;
        }
}
```

「MOBUF に空きが有る事を確認して 1 バイト送信」を繰り返しているだけです。なお、今回は受信バッファのデータを使わないので、ステータスビットの読み出し時に同時に読み出されてしまっても構わない事から、ステータスビット群がある 1 バイトだけを読み出しています。

### ♣ 実行結果

セガサターン側で前項のプログラムを動作させ、PC 側で `hexdump` による MIDI の読み出しを行った所、リスト 3.3 の出力が得られました。

▼リスト 3.3: セガサターンから PC への MIDI メッセージ受信結果

```
$ hexdump -Cv /dev/snd/midiC1D0
00000000  90 01 40 90 01 40 90 01  40 90 01 40 90 01 40 90  |..@..@..@..@..@.|
00000010  01 40 90 01 40 90 01 40  90 01 40 90 01 40 90 01  |.@..@..@..@..@..|
...
```

セガサターン側で送信したノート・オンの MIDI メッセージが、PC 側で正しく受信できていることを確認できました。

# 第 4 章

# 簡単なプロトコルで PC から実行バイナリをロード・実行

この章では、簡単なプロトコルを用意し、セガサターン用の実行バイナリを PC から MIDI 通信でセガサターンへロードし実行してみます。

## 4.1 PC と通信してできること

前章までででセガサターンと PC で相互に MIDI による通信が行える事を確認しました。セガサターンと PC が通信できる事で色々とできる事がありそうです。

初めての通信プログラムなので小さいデータサイズから始めたい所です。そこで、本書ではセガサターン用の実行バイナリをメモリへロードし実行するローダを作ってみます。セガサターンのメイン CPU である SH2 は命令サイズが 2 バイトの固定長なので数十バイト程度でもちょっとしたプログラムが作れます。また、受信したデータをメモリに並べ、そこへジャンプするだけで良いので最初に作ってみる例として良さそうです。

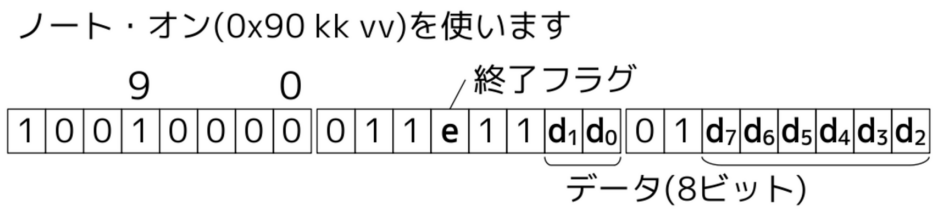
## 4.2 MIDI でデータ通信するプロトコル

前章までで言及していませんでしたが、通信は 100% 成功しているのではなく、セガサターンで MIBUF を読んだ時、PC から送信したデータが `0x00` へ化けてしまう事がありました[*1]。MIEMP ビットで MIBUF が空でない事は確認しており読み方に何らかの問題は思い当たりませんでした[*2]。

[*1] 次に MIBUF を読んだ時、PC から次に送ったデータが読めるので、「まだ準備中だった」とかではなく、本来読めるべきデータが化けてしまっているようです。

[*2] Yabause のコードを見る限り、MIBUF に関するステータスビットは他に MIFULL と MIOVF がありますが、

そこで、簡単なプロトコルを用意してみることにします。まず、データを含むパケット (以降、「データパケット[*3]」と呼ぶ) を、ノート・オンの MIDI メッセージで表すことにします (図 4.1)。




▲図 4.1: データパケットのフォーマット

1 バイト (8 ビット) のデータをノート・オンの MIDI メッセージに埋め込んでいます。先述の通り、MIDI メッセージの規格上、kk と vv の最上位ビットは 0 でなければならないので、データは kk のバイトと vv のバイトに分散して配置しています。「終了フラグ」は一連のデータ送信の終了を受信側へ伝える際のフラグです。余ったビットを 1 にしているのは、プロトコル上「0x00」が発生しないようにするためです。これにより、セガサターン側で受信したデータパケットの中に「0x00」というバイトがあった場合は、「正しく受信できなかった」と判断できます。

加えて、「正しく受信できた/できなかった」をセガサターンから PC へ伝えるパケットとして「ACK パケット」と「NAK パケット」を用意します。「ACK パケット」は「スタート (`0xfa`)」で、「NAK パケット」は「コンティニュー (`0xfb`)」の MIDI メッセージ (表 4.1) で表すことにします。

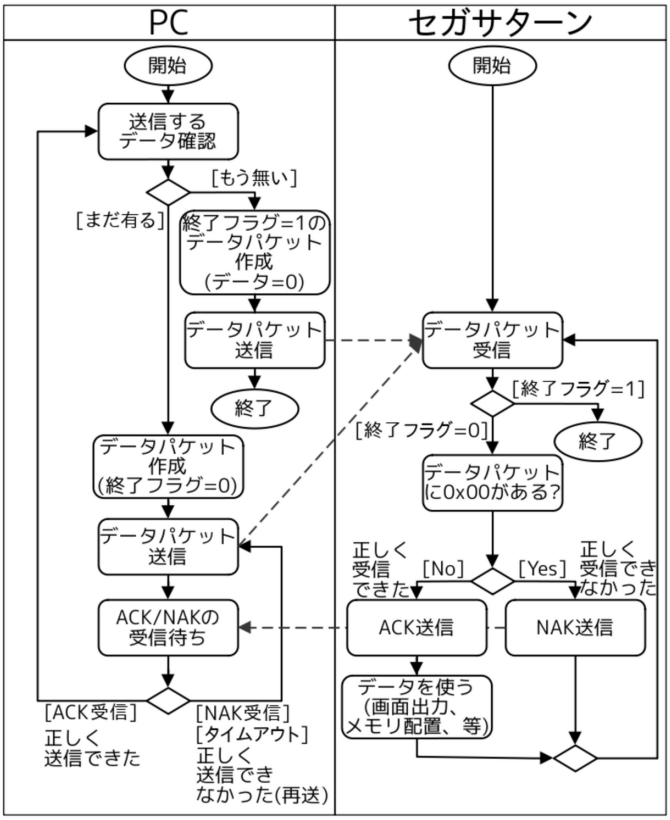
▼表 4.1: MIDI メッセージ:スタート・コンティニュー

| フォーマット | 意味 |
|---|---|
| `0xfa` | シーケンサーなどの PLAY ボタンで送信される。受信側は次のタイミング・クロック (0xf8) で演奏を始める。 |
| `0xfb` | CONTINUE ボタンで送信される。受信側は次のタイミング・クロックで演奏を再開する。 |

以上のパケットを使用し、図 4.2 の流れでセガサターン・PC のやり取りを行います。

MIFULL は後一歩でオーバーフローするのでこれを待ってから MIBUF を読むのは危険そうで、MIOVF は既にオーバーフローしているのでこれを待つのはダメそうです。MIEMP が 0 になってから少しウェイトを入れる、とかも考えましたが、先述の通りステータスビットと MIBUF は同時に読む必要があるためそれも難しいです。

[*3] MIDI 規格にも「データ・パケット」という名前のものがありますが、それとは別物です。あくまでも今回のプロトコルで「データを含むパケット」をここでは「データパケット」と呼んでいます。




▲図 4.2: プロトコルの流れ

「セガサターンで受信したデータに 0x00 が含まれていたら再送する」という簡単なものです。今回のローダで使うには今のところ問題ないという程度のもので、プロトコル自体の完成度は高くありません[*4]。

このプロトコルを実装し、受信したデータを画面に出力するようにしたプログラムを先述の GitHub リポジトリの 083_data_trans_over_midi ディレクトリに置いています。PC 側で「指定された 1 バイトを送信する」という簡単なプログラム (これもシェルスクリプト) も、この中の sample ディレクトリに置いています (send_byte.sh というスクリプト)。興味があれば見てみてください。

実行結果については、これを使って実際に簡単なプログラムをロード・実行してみた次節のプログラムで紹介します。

[*4] 想定していないユースケースにはプロトコル上のバグが残っています。例えば、セガサターン側で ACK を送信した後、何らかの事情でそれが PC 側へ到達せず、PC 側の「ACK/NAK 受信待ち」がタイムアウトすると、PC 側がデータパケットを再送するため、セガサターン側は次のデータだと思ってダブったデータを受信する事になりますが、その様なケースは想定していません。PC 側の「ACK/NAK 受信待ち」のタイムアウトは、PC から送ったデータパケットがセガサターン側の「データパケット受信」のタイミングで受信されないことがあった問題の対処で、それ以外は想定していません。

## 4.3 HELLO WORLD プログラムをロード・実行してみる

### ♣ ローダについて

前節で紹介したプロトコルを使って、PC から実行バイナリを受信しメモリへ配置 (ロード)、実行してみます。「実行」は、具体的にはロード先のアドレスを `bsr` (Branch to SubRoutine) 命令[*5]で関数呼び出しするだけです。なので、実行されるプログラム側で `rts` (ReTurn from SubRoutine) 命令を実行すれば戻ってくる事ができます。

ローダがやることはこれくらいです。ちなみにこのローダプログラムは GitHub リポジトリの 084_midi_loader ディレクトリです。実装について興味があれば見てみてください。PC 側のプログラムはその中の sample ディレクトリに置いています。

### ♣ sample ディレクトリのスクリプトについて

セガサターンと PC を MIDI で接続し、セガサターン側で前述のローダを起動した状態で、PC 側では sample ディレクトリ内の 3 つのスクリプトを使います。

まず send_byte.sh と send_file.sh です。send_byte.sh は前節でも少し紹介しましたが、コマンドライン引数で指定された 16 進数 2 桁の値 (1 バイト) をこの章で紹介したプロトコルに従って MIDI 送信するスクリプトです。そして、send_file.sh は、コマンドライン引数で指定されたファイルを MIDI 送信するスクリプトで、send_byte.sh を使っています。send_file.sh は、カレントディレクトリに send_byte.sh がある想定で動作します。send_byte.sh と send_file.sh は同じディレクトリに置くようにしてください。

最後に hello_world.sh は、HELLO WORLD プログラムの実行バイナリを標準出力へ出力するシェルスクリプトです。こちらは `include` しているスクリプトの場所の関係上、084_midi_loader ディレクトリをカレントディレクトリにした状態で実行してください (リスト 4.1)。

▼リスト 4.1: hello_world.sh の実行例

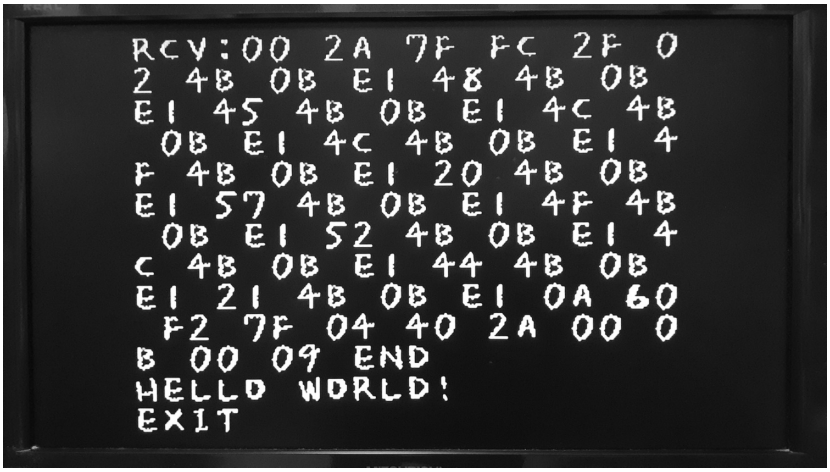
```
$ cd /path/to/084_midi_loader      ← 084_midi_loaderディレクトリをカレントディレクトリにする
$ sample/hello_world.sh >sample/hello_world.exe    ← HELLO WORLD実行バイナリを出力
```

hello_world.sh の中身については本節の最終項で補足として紹介します。

### ♣ 実行結果

実行結果は図 4.3 の通りです。

[*5] 他の CPU で言う所の `call` 命令です。




▲図 4.3: 実行したセガサターンの TV 画面の写真

`RCV:` から `END` までの内容が、セガサターン側で受信しメモリへロードした各バイトです。今回の場合、この部分に HELLO WORLD バイナリの全 68 バイトが表示されています。

ローダは `END` を出力した後、ロード先のアドレスを `bsr` 命令で関数呼び出しします。それにより HELLO WORLD プログラムが実行されます。 `HELLO WORLD!` の行は HELLO WORLD プログラムで出力したものです。

HELLO WORLD プログラムは最後に `rts` 命令で呼び出し元へ戻ります。これによりローダへ戻ってきます。 `EXIT` の部分は処理が戻ってきた後、ローダ側で出力しているものです。

写真では分からないですが、この後、セガサターン側でコントローラのスタートボタンを押すと、再度受信待ちになります。この状態でまた PC 側から実行バイナリを送信し、セガサターン側でロード・実行させることができます。

### ♣ 考察

ここまでで作ったプロトコルとローダはあくまでも PoC です。プロトコルもローダも機能的には当初の要件を満たし、今の所、安定してロード・実行を行えています。しかし、68 バイトの HELLO WORLD プログラムのロードに 1 分程かかり、速度としては 1 バイト/秒程と、性能的には実用に難があります。

MIDI 通信の転送レートは 31.25K ビット/秒 (3.9KB/秒) です。再送が発生しないなら、1 バイトを送信し終えるのにデータパケット (3 バイト) 一つと ACK パケット (1 バイト) 一つなので、MIDI 通信の転送レートの問題ではありません。(体感でも MIDI 通信自体に遅さはありませんでした。)

ボトルネックはセガサターンと PC の ACK/NAK のやり取りの部分で、PC 側で ACK/NAK の受信タイムアウトが多発していました。それにより、遅いときには 1 つのデータパケットを送信し終えるのに体感で 2 秒ほどかかる場合もあり、平均すると 1 バイト/秒という性能になっています。

PC 側で ACK/NAK の受信タイムアウトが発生するのは、PC から送信したデータパケットがセガサターン側の「データパケット受信」のタイミングで受信できない事があるためで、その原因も目星はついているのですが、そもそものプロトコルの方針に問題がありそうです。

そもそも、プロトコルを求めたのは「セガサターンで MIBUF を読んだ時、PC から送信したデータが `0x00` へ化けてしまう事がある」のが理由でした。まさか ACK/NAK やり取りのオーバーヘッドがこんなに大きくなるとは思わなかったので、できる限りやり取りするバイト数を減らす方針で今回のプロトコルにしていましたが、今思えば「送信するデータを一度に複数個送る」という方針でも良かったのかもしれません。例えば、「一度に 10 個送る」プロトコルにしておいて、PC 側は 1 バイトのデータにつきデータパケットを 10 個送ります (全て同じ内容)。受信したセガサターン側は、その内のいくつかが化けたとしても多数決で何が正しいデータかが分かります。この方針なら PC 側は常に送信するだけで、セガサターン側も常に受信するだけで済むので、ACK/NAK のやり取りが発生しません。通信量が一度に送る個数分だけ増えますが、10 個ずつ送るとしてもデータパケット (3 バイト) を 10 個分で 30 バイトなので 31.25K ビット/秒 (3.9KB/秒) の MIDI の転送レートから見れば問題にはならないかと思われます。

もっと言うと、実は MIDI には「ファイル・ダンプ」という機能があります。ただ、これは扱う MIDI メッセージがノート・オン等に比べて長く複雑なので、初めて MIDI メッセージをやり取りするプログラムを作る例としては相応しくありませんでした。

## ♣ 補足：hello_world.sh の中身について

筆者独自のビルドシステムの紹介にもなるので、HELLO WORLD バイナリを生成する hello_world.sh をここで紹介します (リスト 4.2)。

▼ リスト 4.2: sample/hello_world.sh

```
#!/bin/bash

# set -uex
set -ue

. include/common.sh
. include/sh2.sh
. include/charcode.sh

main() {
        # 変更が発生するレジスタを退避
        ## pr
        sh2_copy_to_reg_from_pr r0
        sh2_add_to_reg_from_val_byte r15 $(two_comp_d 4)
        sh2_copy_to_ptr_from_reg_long r15 r0

        # "HELLO WORLD!\n"を出力
        sh2_abs_call_to_reg_after_next_inst r11
        sh2_set_reg r1 $CHARCODE_H
        sh2_abs_call_to_reg_after_next_inst r11
        sh2_set_reg r1 $CHARCODE_E
```

```
        ・・・省略・・・
        sh2_abs_call_to_reg_after_next_inst r11
        sh2_set_reg r1 $CHARCODE_LF

        # 退避したレジスタを復帰しreturn
        ## pr
        sh2_copy_to_reg_from_ptr_long r0 r15
        sh2_add_to_reg_from_val_byte r15 04
        sh2_copy_to_pr_from_reg r0
        ## return
        sh2_return_after_next_inst
        sh2_nop
}

main
```

`sh2_` で始まるコマンドはそれぞれが SH2 の命令に対応していて、include/sh2.sh にそれぞれのマシン語を標準出力へ出力するシェル関数として実装しています。例えば一番最初に実行している `sh2_copy_to_reg_from_pr` は、引数で指定されたレジスタへ PR(プロシージャレジスタ)というレジスタから値をコピーする命令を標準出力へ出力します[*6]。PR は関数呼び出し時に呼び出し元のアドレスが自動的に格納されるレジスタです。ローダから関数呼び出しでここへ来ているためこの時点で PR にローダへ戻る際のアドレスが入っています。この後の処理で文字を出力する関数を呼び出すため、その際に PR が上書きされて失われることが無いよう、予めスタックへ退避しておきます。一番最初の命令では PR を一旦 r0 レジスタ[*7]へコピーしています。続く命令で r15(SP、スタックポインタ) に-4 を加算しています。 `two_comp_d` は引数で指定された値を 2 の補数へ変換するシェル関数で、このようなよく使う計算等は include/common.sh に定義しています。SH2 のアドレス幅は 4 バイト (32 ビット) なので、スタックポインタを-4 して PR を退避しておく場所を作ったわけです。そして 3 つ目の命令でそこへ r0 にコピーしておいた PR を退避しています。このように、シェルスクリプトで SH2 のプログラミングが行えるようになっています。

そして、HELLO WORLD の出力を行っている部分では、`sh2_abs_call_to_reg_after_next_inst` というシェル関数を使っています。これは前述の `bsr` 命令のマシン語を出力するもので、引数で指定されたアドレスの関数呼び出しを行います。今回の場合 r11 を指定していますが、実はローダが実行バイナリを実行する際、r11 に「r1 で指定された文字を出力する関数」のアドレスが設定されています。また、シェル関数に `after_next_inst` とある通り、次の命令を一つ実行してから関数呼び出しを行います[*8]。次の命令は `sh2_set_reg` で、引数で指定されたレジスタへ引数で指定された値を設定します。ここで r1 に文字コードを設定しています[*9]。この様にして 1 文字ずつ出力を行っています。

[*6] アセンブラで言う所の mov 命令ですが、関数名を読むだけで機能や引数に渡すべき内容が分かるようにしています。

[*7] r に番号が続くレジスタは r15 を除いて「汎用レジスタ」と呼ばれるレジスタです。計算や値の一時的な保存場所等、自由に使うことができます。r15 だけは「SP(スタックポインタ)」と用途が決まっています。

[*8] 遅延分岐と呼ばれる SH2 の機能です。

[*9] `CHARCODE_` で始まる定数で定義している文字コードは独自のものです。include/charcode.sh に定義しています。

ここで自作 OS を旨とする当サークルとして重要なのは、実行バイナリがハードウェア資源へ直接アクセスせず、ローダ側で用意された機能を使って目的を果たせている、という所です。セガサターンの場合、画面描画を行うには「VDP」と呼ばれる IC を制御する必要があります。今回の場合、ローダ側にそのような機能が実装されており、文字出力等の関数として用意されています。実行バイナリはその関数を使うだけで文字出力ができていました。正に OS とその上で動くアプリケーションの関係です。そんな訳で、今回のローダもセガサターンにおける一つの自作 OS(の第一歩) なのかなと思っています。

# 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、セガサターンと MIDI インタフェースを通して MIDI キーボードや PC と通信を行ってみました。特にセガサターン上で自作のソフトを動かす所が容易ではないため、試してみるのは難しいかもしれませんが、読み物としてでも楽しんでもらえれば幸いです。

本書で紹介した PC とのデータ通信は、主にプロトコルの設計の問題で速度が出ない結果となりましたが、本文でも説明した通り MIDI の転送レートとしてはもっと出ますので、プロトコル設計を見直してもっと使えるものにしたいなと思います。数百から数 KB/秒くらい出れば、テキストや画像のやり取りはそれほどストレス無くできそうです。

セガサターンと PC の間の通信がスピードの問題も改善されてくれば、できることはぐっと広がりそうです。PC を通してインターネットへアクセスできますので、セガサターンからツイート、なんてできたら面白そうです。

また、今回ローダを作りましたので、PC 側のクライアントアプリの作りを工夫すれば、PC からセガサターン実機の各種レジスタを動的に変更する、ということもできそうです。何でもできそうですが、セガサターンのサウンド周りをまだいじったことが無いので、FM 音源を使って音を鳴らす実験をしてみる、とか面白そうです[*10]。

[*10] 流石に PC からリアルタイムに演奏する、というのは難しそうな気がしますが。

# セガサターンと MIDI で通信する本

MIDI キーボードや PC と通信、実行バイナリローダの作成

---

2022 年 1 月 22 日　ver 1.0 （技術書典 12 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

---

© 2022 へにゃぺんて

*(powered by Re:VIEW Starter)*