Windows API による MIDI プログラミング

大阪工業大学 音声音響情報処理研究室
井上謙次

2005 年 9 月 11 日

目次

第 1 章 MIDI の概要 1
1.1 音 1
1.2 音楽 2
1.3 MIDI 4
1.3.1 演奏情報の伝達 4
1.3.2 楽譜情報の交換 5
1.3.3 音源 5

第 2 章 MIDI プログラミングの準備 7
2.1 本テキストにおける開発環境 7
2.2 Visual C++ のインストール 8
2.2.1 Microsoft Visual C++ Toolkit 2003 のインストール 8
2.2.2 Microsoft Platform SDK のインストール 8
2.2.3 環境変数の設定 9
2.3 Visual C++ が既にインストールされている場合 9
2.4 その他の開発環境 10
2.4.1 Borland C++ 10
2.5 サンプルプログラム 10
■ プログラミングメモ 1 — コンパイルとリンク 12
■ プログラミングメモ 2 — 環境変数とパス 13

第 3 章 MIDI の出力 14
3.1 出力デバイスの一覧を得る 14
■ プログラミングメモ 3 — #include 文 16
■ プログラミングメモ 4 — Windows API プログラミング特有の識別子 . 17
3.2 ノートオンとノートオフ 20
3.2.1 ノートオンメッセージ 21

3.2.2 ノートオフメッセージ 22
3.2.3 出力デバイスのオープンとクローズ 22
3.2.4 MIDI メッセージの送信 25
■ プログラミングメモ 5 — バイトオーダー（エンディアン） 27
3.3 チャンネルメッセージ 28
3.3.1 MIDI メッセージの分類 28
3.3.2 チャンネルメッセージ 28
3.3.3 プログラムチェンジの送信 30
3.3.4 複数チャンネルの使用 33
■ プログラミングメモ 6 — 引数の型と型キャスト 34
■ プログラミングメモ 7 — 値渡しと参照渡し 35

第 4 章 MIDI の入力 38
4.1 入力デバイスの一覧を得る 38
■ プログラミングメモ 8 — ジェネリックテキストマッピング 39
4.2 入力デバイスのオープンとクローズ 42
■ プログラミングメモ 9 — main 関数の戻り値 45
4.3 MIDI メッセージの受信 46
4.3.1 受信の開始と停止 46
4.3.2 コールバック関数 47
4.3.3 メッセージの受信 48
■ プログラミングメモ 10 — イベントドリブン型プログラミング 49
4.4 MIDI メッセージの処理 50
4.4.1 コールバック関数に渡されるメッセージ 50
4.4.2 システムエクスクルーシブメッセージ 52
■ プログラミングメモ 11 — 変数の宣言とメモリ領域の確保 58

第 5 章 SMF 61
5.1 SMF の基本構造 61
5.1.1 バイナリエディタの活用 61
5.1.2 ヘッダチャンク 62
5.1.3 トラックチャンク 64
5.1.4 SMF のバイトオーダー 64
■ プログラミングメモ 12 — ファイルの読み書き (1) 65
■ プログラミングメモ 13 — ファイルの読み書き (2) 69

5.2 トラックイベント . . . 73
5.2.1 デルタタイム . . . 74
5.2.2 可変長データ表現 . . . 75
5.2.3 MIDI イベント . . . 76
5.2.4 SysEx イベント . . . 77
5.2.5 メタイベント . . . 78
5.3 SMF ファイルの読み込み . . . 80
5.3.1 DLL の作成 . . . 80
5.3.2 ライブラリの基本設計 . . . 82
5.3.3 ファイルのオープンとクローズ . . . 83
5.3.4 make の利用 . . . 92
5.3.5 トラックの処理 . . . 94
5.3.6 トラックイベントの読み込み . . . 95

付録 A 課題の解答例 96
A.1 第 2 章 . . . 96
A.2 第 3 章 . . . 97
A.3 第 4 章 . . . 99

付録 B 参考文献・Web サイト 105
B.1 MIDI 関連の情報源 . . . 105
B.1.1 書籍 . . . 105
B.1.2 Web サイト . . . 105
B.2 その他 . . . 106
B.2.1 書籍 . . . 106
B.2.2 ソフトウェア . . . 106

索引 108

第 1 章

MIDI の概要

本章では，MIDI の概要を説明する。MIDI は音楽を表すための規格であるが，まず最初に一般の音や音楽について簡単に説明し，次に MIDI の基本的な説明を行う。

1.1　音

音は物質（空気や固体など）の振動が耳に届いたものである。この振動を記録して，目に見えるように描いたものが波形である（図 1.1）。

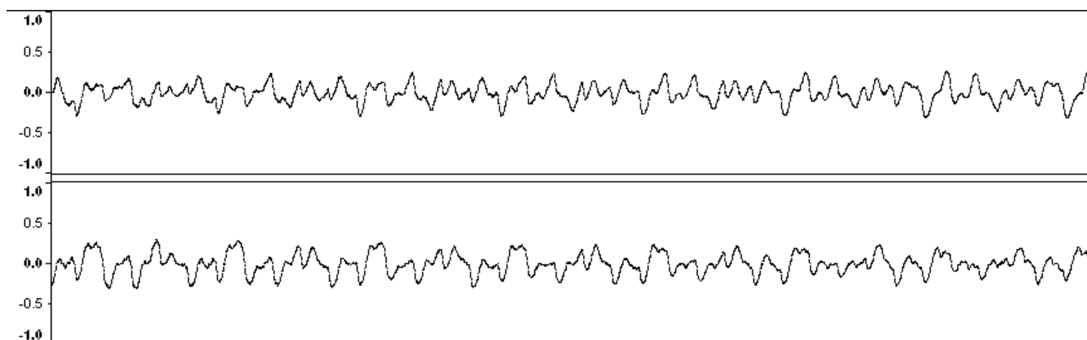


図 1.1　音の波形 (0.12 秒間)

音をコンピュータで扱うには，まず音を AD 変換（サンプリング＋量子化）する。それによって得られる情報は，各サンプリング点の「間隔」と「音の大きさ」であり，単純にこの情報を記録する方式を PCM (Pulse Code Modulation) と呼ぶ（図 1.2）。また，必要に応じて圧縮して記録することも行われる。

音波形のフォーマット例としては，Microsoft Waveform Audio (.wav) や MPEG1 Layer-3 (.mp3) がある。

音声や音楽は，特定の目的に対して音という聴覚領域を利用したものであり，一般的な

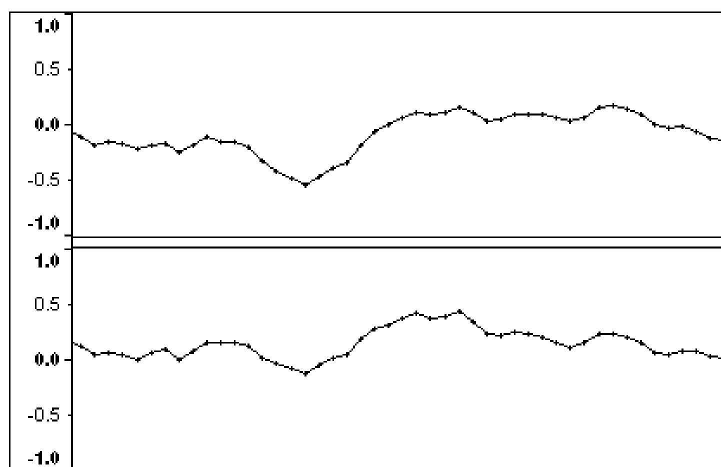


図 1.2　サンプリング点 (サンプリング間隔は約 0.02ms)

音に比べて特徴的な性質を持っている（特徴的な波形になる）。また，音には色々な使われ方や分類がある。例えば：

- 音声（声）
- 音楽・音響
- 信号音
- 環境音

などが挙げられる[*1]。

音の波形を扱う場合は波形処理や信号処理の知識が必要であるが，それらの技術については本テキストでは扱わない。

1.2　音楽

音楽は，基本的に「12 段階 × 数オクターブ」の高さの音の組み合わせで表される。これを紙に記録したものが楽譜である。楽譜に記録される情報は，各音符の「間隔（テンポと各音符の長さ）」と「高さ」である（図 1.3）。また，楽譜に書かれるその他の情報としては,

- 抑揚（音の大きさ），装飾音などの細かな表現
- 各トラック（パート）の楽器名など

などが挙げられる。

[*1] 音響は別に音楽に限ったものではないが，「音がどのように響くか」ということについては音楽が最も興味のある対象であろうから，一緒に並べたものである。

図 1.3　Power Tab Editor

楽譜のフォーマット例としては，Standard MIDI File (.mid) や Power Tab (.ptb) がある。

楽譜に書かれる情報は，波形で表される音空間から音楽に必要な要素（=音符）のみを抜き出して抽象化しているといえる。ただし，楽譜を実際に演奏する場合には，人間の耳に聞こえるのは音波形となる（楽譜データを音波形に変換する必要がある）。

また，音楽には，ある音楽がどのような音符の並びで表されるかということ（楽譜）以外にも，その情報がどのような音波形で再生されるかということ（演奏）と，その情報がどのように聞こえるか（音響）という部分があるだろう。

それぞれ重要な部分であるが，本テキストではこのうち楽譜的な情報を主体として扱う。そして，そのような情報を扱ったものが MIDI である。MIDI は演奏情報を扱う規格と言われるが，これは上に挙げた「楽譜」の情報を主に，「演奏」（音の再生）の情報を含めたデータを扱えるフォーマットである。

本テキストでは，具体的にはプログラミングを通して MIDI の理解と実際に MIDI をソフトウェアで利用する技術を学べるように，チュートリアル形式で説明していく。

次節以降ではこの MIDI についての基本的な説明を行う。

1.3　MIDI

1.3.1　演奏情報の伝達

MIDI は Musical Instrument Digital Interface の略で，音楽の演奏情報を伝達するための世界共通規格である。その名が示すように，元々は音楽用機器間のインタフェースとして開発された。

典型的な例を図 1.4 に示す。図 1.4 ではキーボードからの出力が PC への入力になっている。ここで送られるデータは，押した音の高さ（図では 4 オクターブ目の C=ド）と音の強さ（鍵盤を押す速度）である（ノートオン）。また，指を離したときには「指を離した」という意味の情報（ノートオフ）も送られる。ここでは，データ自身には音の長さに関する情報は含まれておらず，音のオンとオフが実際に送られた時間で決定されることに注意したい。

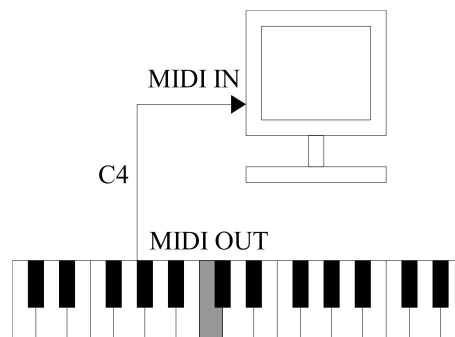


図 1.4　MIDI キーボード

ノートオン/ノートオフ以外には，音色の切り替えを表すプログラムチェンジや，ギターや管楽器などで音程を微妙に変化させるベンド[*2] 奏法を表すピッチベンド，ビブラートなどの各種エフェクトをかけるコントロールチェンジなどがある。これらについては今は詳しく知る必要はない。

ここで伝達されるのは楽譜の情報全体ではなく，単に「一つ一つの音」が，それらが発生したときに逐次送られており（そのように時折発生する事象をイベントという），実時間の演奏データが扱われていたに過ぎず，あくまでケーブル上で送受信される信号用の規格であった。従って，MIDI は当初は楽譜データを表すために作られたものではないが，後に楽譜データ全体を表すことができるようにも拡張された。

[*2] ギターではよくチョーキングという。

1.3.2　楽譜情報の交換

楽譜情報を表現可能な MIDI は，通常 SMF (Standard MIDI File) という標準形式で保存される。これは演奏情報を実時間でやりとりするわけではなく，楽譜全体を記録するためのものなので，名前の通り単一のファイルとして保存される。

SMF は基本的に PC で処理するためのフォーマットであり，SMF を直接 MIDI 機器に送信しても通常は正常に処理されない。しかし，SMF から演奏イベントを抜き出したものであればそのまま送信すれば機能するので，まったく別物ではない。機器間で通信する MIDI 情報に，イベントの発生時間などの情報を付加したものが SMF 形式であるといえる。

SMF で拡張されて対応されるようになったのは，イベントの発生時間（タイムベースとデルタタイム）以外にも複数トラックのサポートや，調やテンポなどの楽譜表記，作者や歌詞，コメントなどの楽譜のメタデータ，マーカやキューポイントなどの作曲ソフト[*3]で利用されるメタデータなどがある。ただし，これらを今覚える必要はない。

SMF 形式の MIDI ファイルの拡張子には .mid が使われることが多い（たまに .smf や .midi も使われる）。

1.3.3　音源

MIDI はあくまで楽譜的な情報を伝達・保存するためのものであるので，それを人間が聞こえるようにするには，紙の楽譜を見て楽器を使って演奏するのと同じように，何らかの手段で音を鳴らす必要がある。この部分は音波形の場合とは異なっている（音波形は実際の音に直接対応する）。

音を鳴らすための装置やシステムを音源と呼ぶ。通常のキーボードでは音源が付いているものと付いていないものがある。Clavinova のような電子ピアノでは内部に音源があって，押したキーボード（鍵盤）のキーに対応する音が再生される。

音源の方式は様々であるが，基本的な PCM 音源について述べる。これは実際の楽器の音などを録音し PCM で保持しておき，再生時に音程を変えるなどの加工を施して出力する。現在はこの PCM 音源が主流である。

MIDI ではプログラムチェンジで音色を変更することができるが，各番号に割り当てられている実際の音の種類がそれぞれの環境で異なってしまってはまともに演奏できない。例えば，ある環境では 1 番の音源を指定すればグランドピアノが鳴るが，違う環境ではサックスが鳴る，というのでは MIDI に互換性があるとは言えない。そこで，プログラム

[*3] シーケンスソフトともいう。

チェンジの番号と音色の種類との最低限必要な対応が GM (General MIDI) という規格で規定されている。GM 規格に沿った音源を GM 音源と呼ぶ。

GM 音源の拡張に GS 音源や XG 音源などがあり，Windows では標準で GS 音源が搭載されている。

第 2 章

MIDI プログラミングの準備

本章では，MIDI プログラミングを行うための開発環境の整備とサンプルプログラムの実行を行い，次章以降における本格的な MIDI プログラミングの準備を整える。

2.1 本テキストにおける開発環境

本テキストでは Windows を対象とし，C 言語を用いて Windows API を使って MIDI のプログラミングを行う。従って，Windows 上で C 言語のコンパイラおよび Windows API を使える環境を構築する必要がある。

もし既に慣れている Windows プログラミングの環境が整っていれば，その環境で良いだろう。ただし，本テキストではコンパイラとして Visual C++（以下 VC++）を採用しているので，動作検証も VC++ のコンパイラでのみ行っていることに注意されたい（ただし VC++ .NET で同様に動作するかは分からない）。

また，本テキストでは VC++ のコンパイラをコマンドラインから実行する環境を想定している。そのような環境を採用しているのは，VC++ が広く一般に普及していることに加えて，インターネット上からダウンロードすれば無料でその環境が構築可能なことが理由である。VC++ の IDE[*1] を使用している場合や，他のコンパイラ —— あるいは他のプログラミング言語 —— を利用したい場合は，その都度各自の環境に読み替えて進んで行っていただきたい。

また，VC++ を既に持っているならば，IDE を使わずに（本テキストで採用している）コマンドラインで利用する方法もあり，その場合は 2.3 節に説明がある。

[*1] 統合開発環境 (Integrated Development Environment)。エディタ，コンパイラ，デバッガなどが統一的なインタフェース（主に GUI）で利用できるソフトウェア。

2.2　Visual C++ のインストール

以下の方法では，ダウンロード容量は最大 400MB，必要なハードディスクの空き容量はおよそ 1GB である。もしそのような空き容量が確保できない場合は，代わりに Borland C++ を利用することもできるので，2.4 節を参照して欲しい。

2.2.1　Microsoft Visual C++ Toolkit 2003 のインストール

まず，Microsoft Visual C++ Toolkit 2003 をインストールする。ダウンロードは http://msdn.microsoft.com/visualc/vctoolkit2003/ 内にある Download the Visual C++ Toolkit 2003 リンクからできるだろう[*2]。これにはコンパイラやリンカなどが含まれている。以下では，このツールキットを標準のフォルダである `C:\Program Files\Microsoft Visual C++ Toolkit 2003` 内にインストールしたものとして進めていく。

2.2.2　Microsoft Platform SDK のインストール

次に，Microsoft Platform SDK をインストールする。これは時折更新されるが，2005 年 6 月現在では Windows Server 2003 SP1 Platform SDK[*3] が配布されている。もし 2003 年 2 月にリリースされた旧バージョンが欲しい場合は http://www.microsoft.com/msdownload/platformsdk/sdkupdate/psdk-full.htm のサイトからダウンロード可能なようである。

Microsoft Platform SDK をインストールするには，まず Windows Server 2003 SP1 Platform SDK Web Install のサイト http://www.microsoft.com/downloads/details.aspx?familyid=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5&displaylang=en から PSDK-x86.exe（他の CPU を使用している場合はその CPU 用のファイル）をダウンロードする。インストーラを実行し，指示に従ってインストールを行う（インストールするコンポーネントは標準で良い）。ダウンロードとインストールが自動的に行われる。

もしダウンロードに失敗するようであれば，同サイトから ISO ディスクイメージをダウンロードし，DaemonUI などで直接実行するか，CD-R に焼くなどして実行すればよい。

[*2] 登録 (pre-download registration) は不要である。

[*3] Windows Server 2003 SP1, Windows XP SP2, Windows XP x64 Pro Edition, Windows 2000 に対応していると書かれているが，おそらくそれ以外の環境でも動くと思われる。もちろん動作保証はないが。

2.2.3　環境変数の設定

MIDI プログラミング用に適当な作業用フォルダを作成しておく。そして，そのフォルダ内に次のようなバッチファイル[*4] を作成して配置する。ファイル名は拡張子が .bat であれば何でもかまわないが，ここでは setenv.bat としておく。このバッチファイルは本テキストを通じて使用する。

リスト 2.1 環境変数設定用バッチファイル

```
rem ----- Microsoft Platform SDK -----
set PATH=C:\Program Files\Microsoft Platform SDK\Bin;%PATH%
set PATH=C:\Program Files\Microsoft Platform SDK\Bin\Win64;%PATH%
set INCLUDE=C:\Program Files\Microsoft Platform SDK\Include
set LIB=C:\Program Files\Microsoft Platform SDK\Lib

rem ----- Microsoft Visual C++ Toolkit -----
set PATH=C:\Program Files\Microsoft Visual C++ Toolkit 2003\bin;%PATH%
set INCLUDE=C:\Program Files\Microsoft Visual C++ Toolkit 2003\include;%INCLUDE%
set LIB=C:\Program Files\Microsoft Visual C++ Toolkit 2003\lib;%LIB%
```

コマンドプロンプト上で setenv.bat を実行すると，環境変数が設定され，cl.exe などのコンパイラや各種ライブラリへのパスが通るようになる（この設定を怠ったり誤ったりするとコンパイルに失敗するので注意)。実行したコマンドプロンプト上でのみ効果があり，一度コマンドプロンプトを閉じると設定は失われる。従って，コンパイル作業を行うコマンドプロンプト上で一度だけ実行すると良い。

2.3　Visual C++ が既にインストールされている場合

VC++，あるいは Visual Studio が既にインストールされている場合，その IDE を利用してプログラミングを行えばよい。しかし，もし本テキストと同じコマンドラインの環境でプログラミングを行いたければ，VC++ がインストールされているフォルダ内にある `bin\vcvars32.bat` をコマンドプロンプト上で実行すると，一時的に（そのコマンドプロンプト上でのみ）VC++ のコンパイラやリンカなどにパスが通るようになる。後は，本テキストの指示通りに進めれば問題無いだろう。

ただし動作検証はしていないため，もしかすると SDK のライブラリへのパスは自分で設定する必要があるかもしれない。その場合は 2.2.3 節を参考にして欲しい。

[*4] ファイル中に記述された複数のコマンドを上から順に実行するスクリプト形式のプログラム。

2.4　その他の開発環境

2.4.1　Borland C++

VC++ コンパイラと Platform SDK の容量が大きすぎると感じるのであれば，代わりに無料の Borland C++ のコンパイラ (BCC) と BCC 対応の開発環境を利用することもできる。BCC は http://www.borland.co.jp/cppbuilder/freecompiler/ からダウンロードすることができる。BCC 用の IDE には，例えば BCC Developer というソフトウェアがあり，http://www.hi-ho.ne.jp/jun_miura/bccdev.htm から入手可能である。これらのインストールに必要な空き容量は 60MB 足らずである。インストールや設定に関しては http://ja2yka.homeip.net/aki/freedev.htm などのサイトを参照するとよい。

2.5　サンプルプログラム

実際にサンプルプログラムを打ち込み，コンパイルして動作させてみよう。まずリスト 2.2 のプログラム (midiout.c) を入力し，

```
cl midiout.c winmm.lib
```

でコンパイルを行う[*5]。もし setenv.bat を実行しておらず環境変数の設定がされていない場合は，cl.exe が見つからない旨のエラーが出るだろう（図 2.1）。その場合は次のようにまず setenv.bat を実行した後にコンパイルを行う。

```
setenv.bat
cl midiout.c winmm.lib
```

一度 setenv.bat を実行すれば，コマンドプロンプトを閉じない限り再度実行する必要はない。プログラムが間違っている場合はもちろん，環境変数の設定が間違っている場合にもエラーが出るので注意されたい（プログラムが間違っている場合はコンパイルエラー，環境変数の設定ミスの場合はリンクの時点でエラーが発生するだろう）。

リスト 2.2　サンプルプログラム (midiout.c)

```
#include <windows.h>
#include <mmsystem.h>
```

[*5] BCC Developer を使用している場合はメニューの「プロジェクト」→「メイク」でコンパイルし，「実行」→「実行」で実行すればよい。

```
int main(void) {
  HMIDIOUT hMidiOut;                          /* MIDI 出力デバイスのハンドル */

  midiOutOpen(&hMidiOut, MIDI_MAPPER, 0, 0, 0);  /* MIDI 出力デバイスを開く */

  midiOutShortMsg(hMidiOut, 0x007f3c90);  /* 中央の C（C4）を強さ 127 で鳴らす */
  Sleep(1000);                            /* 1000ms 間処理を中断 */
  midiOutShortMsg(hMidiOut, 0x007f3c80);  /* 中央の C（C4）の再生を止める */

  midiOutReset(hMidiOut);                 /* 全チャンネルをノートオフ */
  midiOutClose(hMidiOut);                 /* MIDI 出力デバイスを閉じる */

  return 0;
}
```

図 2.1　cl.exe にパスが通っていない場合

上記コマンドでコンパイルに成功すると，midiout.exe という実行ファイルが生成されているはずである[*6]。以下のコマンドでプログラムを実行できる。

```
midiout.exe
```

C4（ピアノの中央のド）の音が 1 秒間鳴れば正常な動作である。

ここではプログラムの解説は行わないが，おおよその処理はコメントから理解できるだろう（詳しい解説は 3.2 節にある）。midiOutShortMsg() 関数の引数中の，0x3c = 60 がノートオンイベントの音の高さ（ノート番号）を表している（以下，16 進数は数字の先頭に 0x を付けて表記する）。試しにこの部分の値を変更してコンパイルし，実行してみて欲しい。

[*6] もし出力される実行ファイルの名前を変更したい場合は cl コマンドの /Fe オプションを用い，`cl /Femyfavoritename.exe midiout.c winmm.lib` というように指定すればよい。

課題 1

リスト 2.2 のプログラムを改造し，ドミソの和音を 1 秒間鳴らすように変更せよ。ベース音のドは中央のド (C4 = 60) とする。ノート番号を 1 大きくすると半音高くなることに注意。

プログラミングメモ 1 — コンパイルとリンク

コンパイルのコマンドについて少し説明しよう。今回の場合，コマンドは `cl midiout.c winmm.lib` である。ここで cl は cl.exe のことであるが，Windows の場合，コマンド実行の指定に拡張子は要らない。midiout.c はコンパイルするソースファイルである。

では，winmm.lib は何か？簡単に言えば，これはオブジェクトプログラムがリンクするライブラリのファイル名を指定している。上記プログラムでは midiOutOpen() や midiOutShortMsg() など MIDI 関連のライブラリ関数を利用しているため，そのライブラリにリンクする必要がある。winmm.lib はそのために利用される。

リンクという作業を知らないか忘れていると困るので，簡潔に説明しておこう。ソースコードから実行可能形式（Windows では .exe 拡張子が付けられる）を生成する作業（通常コンパイルと呼ばれる）には，2 つの工程がある。まず最初に（狭義の）コンパイル（これは上述のコンパイルとは違う意味であるが，混同しやすいので注意）を行ってソースファイルからオブジェクトプログラム (拡張子には .obj がよく使われる) を生成し，次にリンクを行って複数のオブジェクトファイルやライブラリを統合して実行可能ファイルを生成する。（狭義の）コンパイルは C などで書かれたプログラムを機械語に変換する作業であり，リンクはその機械語のファイルを相互に利用できるように接続する作業である。

コンパイルとリンクは別々の作業であるので，実際に 2 つのコマンドに分けて実行することができる。例えば `cl /c midiout.c` と指定してコンパイルすると midiout.obj が生成される（/c はコンパイルのみを行うオプション）ので，`link midiout.obj winmm.lib` でリンクすればよい。これは結局最初のコマンドと同じことを行っている。

ところで，実際に midiout.exe がリンクされているのは Windows のシステムディレクトリにある winmm.dll であり，winmm.lib そのものではない。これは動的リンク（ダイナミックリンク; Dynamic Link）という方式でリンクしているためである。別に静的リンク（スタティックリンク; Static Link）という方式があり，これはリンク時に実行ファイルにライブラリを埋め込んでしまう方法である。その結果，静的リンクはファイルサ

イズは大きくなる。動的リンクでは，プログラムの実行時にその場で DLL[*7] ファイルとのリンクが行われる。winmm.lib は実行時にその DLL ファイルと動的にリンクするための情報を保持している（少し高度なテキストエディタか，適当なバイナリエディタで winmm.lib を開いてみれば，そこにはプログラム自体はないことが分かるだろう）。

なお，cl コマンドや link コマンドの簡単な説明は `cl /?` や `link /?` で見れるのでオプションを調べたい場合は活用しよう。これを見ると，コンパイルのコマンドは `cl midiout.c winmm.lib` よりも `cl midiout.c /link winmm.lib` の方が行儀がよいのかもしれない。

プログラミングメモ 2 — 環境変数とパス

winmm.lib はプログラミングメモ 1 で述べたような役割をする，winmm.lib という名前のファイルなのであるが，カレントフォルダにはそのようなファイルは存在しない。また，cl.exe というファイルも実は存在していない。これらのファイルはどこに存在し，どこから探してきているのであろうか？その情報を与えているのが環境変数である。

Windows のシェルプログラム（コマンドプロンプト）は，環境変数 PATH に登録されているフォルダに cl.exe というファイルが存在するかを調べて，もし見つかれば実行する（無ければエラーが表示される）。また，cl.exe は，環境変数 LIB に登録されているフォルダから winmm.lib を探す。setenv.bat を書いて実行するのは，cl.exe や winmm.lib などのファイルを環境変数に登録するためである。現在の環境変数の内容は，コマンドプロンプト上で `echo %PATH%` や `echo %LIB%` などと打てば表示される。

もし環境変数で指定していない場合は，`"C:\Program Files\Microsoft Visual C++ Toolkit 2003\bin\cl" midiout.c "C:\Program Files\Microsoft Platform SDK\Lib\winmm.lib"` というようにフルパスを記述すれば問題ないが，非常に面倒なためお勧めしない（実際にはヘッダファイルの存在するフォルダ名も指定する必要があるため，この倍程度の長さの記述が必要となる）。

[*7] 動的リンク (Dynamic Link) を利用して使うライブラリを Dynamic Link Library，略して DLL という。

第 3 章

MIDI の出力

MIDI は元々音楽機器間の通信のために開発されたものであるので，MIDI データの送受信が基本である。3 章と 4 章では，PC と MIDI 機器間での MIDI データの送受信を扱えるようになることを目的とする。

本章では PC から MIDI 機器への出力を扱う。まず最初に PC から出力可能な MIDI 機器（出力デバイスという）を調べる方法を説明し，次に調べた出力デバイスに MIDI データ（MIDI メッセージという）を送信し，MIDI 機器を制御する方法を説明する。

3.1　出力デバイスの一覧を得る

MIDI の出力デバイスの一覧を得るには midiOutGetNumDevs() 関数と midiOutGetDevCaps() 関数を使う。

midiOutGetNumDevs() は MIDI 出力デバイス数を返す関数で，プロトタイプ宣言は次の通りである。

```
UINT midiOutGetNumDevs(void);
```

この関数でデバイス数が得られるので，それら（0 〜 デバイス数-1）に対して midiOutGetDevCaps() を呼び出せばよい。midiOutGetDevCaps() は与えられたデバイス ID に対してそのデバイスの情報を返す（構造体に格納する）関数で，そのプロトタイプ宣言は次の通りである。

```
MMRESULT midiOutGetDevCaps(
  UINT_PTR      uDeviceID,
  LPMIDIOUTCAPS lpMidiOutCaps,
  UINT          cbMidiOutCaps
);
```

戻り値は，成功すれば（エラーがなければ）MMSYSERR_NOERROR が返る。エラー時には他の値が返されるが，詳細は省く。MIDIOUTCAPS 構造体は多くのメンバを持つが，一覧を表示するにはデバイス名を保持する szPname メンバの値を出力するだけでよい。その他，詳しくは MSDN などのリファレンスを参照して欲しい（Google で関数名をキーワードに検索すれば引っかかるだろう）。

Windows API プログラミングに慣れていなければ，上記の説明では分かりにくいだろう。実際に MIDI 出力デバイスの一覧を出力するプログラムをリスト 3.1 に示すので，打ち込んでコンパイルし，実行結果を確認して欲しい。

リスト 3.1 出力デバイス一覧の表示 (list-outdevs.c)

```
#include <stdio.h>
#include <windows.h>
#include <mmsystem.h>

int main(void) {
  MIDIOUTCAPS outCaps;
  MMRESULT res;
  UINT num, devid;

  /* デバイス数を取得 */
  num = midiOutGetNumDevs();
  printf("Number of output devices: %d\n", num);
  /* 各デバイス ID に対して for ループ */
  for (devid=0; devid<num; devid++) {
    /* デバイスの情報を outCaps に格納 */
    res = midiOutGetDevCaps(devid, &outCaps, sizeof(outCaps));
    /* midiOutGetDevCaps の戻り値が成功でない（=失敗）なら次のループへ */
    if (res != MMSYSERR_NOERROR) { continue; }
    /* デバイス ID とそのデバイス名を表示 */
    printf("ID=%d: %s\n", devid, outCaps.szPname);
  }

  return 0;
}
```

コンパイルは

```
cl list-outdevs.c winmm.lib
```

で行う。もちろん setenv.bat で環境変数の設定をしておく必要がある。また，

```
list-outdevs.exe
```

で実行する。これ以降はコンパイルと実行の説明は省略する。

参考に筆者の環境での実行結果を図 3.1 に載せる。この Microsoft GS Wavetable SW Synth は Windows 付属の GS 音源である。

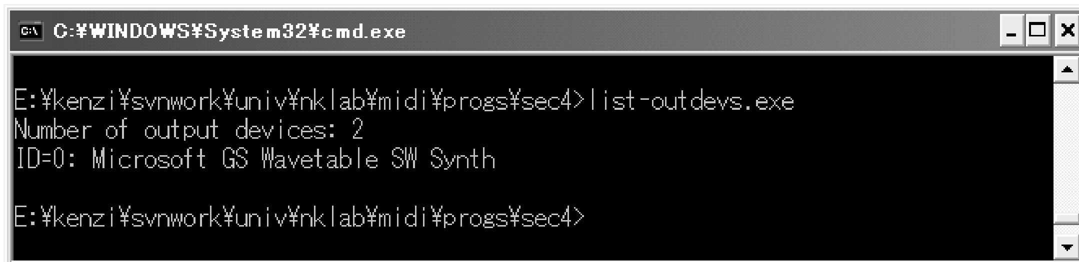


図 3.1　list-outdevs.exe の実行結果

可能であれば，外部 MIDI 機器を接続していない状態と接続している状態の 2 通りで上記プログラムを実行して欲しい。

課題 2

独自音源を内蔵した外部 MIDI 機器を接続した状態[*1] で list-outdevs.exe を実行し，外部 MIDI 機器のデバイス ID を調べる。2.5 節の課題 1 で作成した和音を鳴らすプログラムで，出力デバイス ID を MIDI_MAPPER から外部 MIDI 機器の ID に書き換え実行し，外部 MIDI 機器で和音が再生されることを確認せよ。

プログラミングメモ 3 — #include 文

ソースコード先頭の #include 文について説明する。これらの #include 文は当然ヘッダファイル[*2] の読み込みである。ヘッダファイルには構造体の定義や，関数のプロトタイプ宣言などが記述されている。ある関数を呼ぶため[*3] には，その関数のプロトタイプ（関数名，引数の数と型，戻り値の型の 3 つの情報）が分かっている必要があるので，ライブラリ関数を呼ぶためにはそのライブラリ用のヘッダファイルを読み込まなければならない。

stdio.h[*4] は printf() 関数のためにインクルードしている。これは MIDI プログラミングには関係ない。

[*1] Clavinova であれば，AB タイプの USB ケーブルで USB 端子に接続すればよいだろう。MIDI 機器側に USB 端子が無い場合は，USB-MIDI ケーブルなどを用いて（MIDI 機器側の）MIDI IN 端子に接続する。

[*2] ヘッダファイルにはよく .h という拡張子が使われる。

[*3] より正確には，コンパイル時に関数呼び出しの型が正しいかを判断するため。C 言語は型制約の強い言語である。

[*4] STanDard Input/Output の略であろう。

MIDI プログラミングに関係するのは，mmsystem.h[*5] である。これは Microsoft が提供しているマルチメディア関連の Windows API[*6] を使うためのヘッダファイルである（実体は winmm.dll[*7]）。MIDI 関連の API の関数や構造体などもこの mmsystem.h で定義されている。

windows.h は全ての Windows API で使われる基本的なデータ型などを定義している。mmsystem.h 内にはそれらのデータ型などを利用しているので，windows.h もインクルードする必要がある。

プログラミングメモ 4 — Windows API プログラミング特有の識別子

リスト 3.1 のプログラムには，MIDIOUTCAPS，MMRESULT，UINT，MMSYSERR_NOERROR といった見慣れないデータ型や定数が大量に出現する。これは Windows API プログラミングを始めて最も戸惑うことがらの一つであろう。実は C 言語が理解できていれば（面倒ではあるが）それほど難しくはないのだが，基本的にはこれはもう Microsoft の書き方の流儀に慣れるしかない。

Windows API プログラミングに特有のこれら「全て大文字の識別子」は，次の 3 つのうちのいずれかである[*8*9]：

1. 基本型の別名：上記では MMRESULT と UINT
2. 構造体：上記では MIDIOUTCAPS
3. 定数：上記では MMSYSERR_NOERROR

順に説明していこう。

基本型の別名

基本型の別名は，C 言語の型定義に使う typedef で基本型（言語に標準で定義されている型）の別名を作っている。MMRESULT[*10] と UINT[*11] は実際には次のように定義されている。

```
typedef UINT          MMRESULT;
```

[*5] MultiMedia System の略であろう。
[*6] API (Application Programming Interface) はあるソフトウェア（Windows API の場合は Windows）を動作させるためのライブラリ。処理を依頼するといってもよい。
[*7] WINdows MultiMedia の略であろう。
[*8] おそらく無いとは思うが，もしかすると関数形式のマクロにも全て大文字の識別子のものが使われている可能性もある。その場合は関数呼び出しの形式で書くので，他のものとの区別はしやすいだろう。
[*9] ちなみに最近のプログラミングスタイルでは，全て大文字の識別子は定数にのみ使われる傾向にある。
[*10] MultiMedia RESULT の略であろう。
[*11] Unsigned INTeger の略であろう。符号無し整数（マイナスが付かない整数=非負整数）である。

```
typedef unsigned int  UINT;
```

こうしておくと，型名で MMRESULT と書いた場合は全て UINT 型と見なされる。しかし UINT 型も typedef で作られているので，MMRESULT 型は実際は unsigned int 型である。

ではこれらの定義がどこに書かれているかというと，MMRESULT の型定義は mmsystem.h 内に記述されており，UINT の型定義は windef.h 内に記述されている。windef.h はリスト 3.1 のソースコードからは直接インクルードしていないが，windows.h が windef.h をインクルードしているために UINT が利用できる。

このようにヘッダファイルを調べれば実際にはそれがどういう型であるのかを見つけることができる。Windows API でインクルードされるヘッダファイルは `C:\Program Files\Microsoft Platform SDK\Include` フォルダ内にあるので，そのフォルダ内の .h ファイルに対して grep[*12] をかけるなどの検索を行えばよい。ヘッダファイルの包含関係は複雑であるので，grep を行わないと型定義がなされているファイルを見つけるのは非常に面倒である。

以下に，Windows API プログラミングでよく使われる基本型の別名を挙げる。これ以外が出てきた場合は，自分で windef.h などを調べたり Web を検索したりして自分で探して欲しい。

リスト 3.2 基本型の別名一覧

```
typedef int                 BOOL;
typedef unsigned char       BYTE;
typedef unsigned long       DWORD;
typedef float               FLOAT;
typedef int                 INT;
typedef unsigned int        UINT;
typedef unsigned short      WORD;
```

ついでに，これらの型名によく使われる接頭辞を挙げておこう。ほとんどが英語の略語であるが，1〜2 文字しか使われないために類推しにくい。

- `D : Double -` **サイズが 2 倍**
- `L : Long -` **サイズが大きい（具体的なサイズは処理系などに依存する）**
- `LP : Long Pointer - 32` **ビット長のポインタ**
- `P : Pointer -` **ポインタ**
- `U : Unsigned -` **符号無し（非負の数；符号とは数字の前のプラスやマイナスのこと）**
- `W : Wide -` **ワイド文字（**`Unicode` **関連）**

[*12] 指定ファイルから行単位で検索し，マッチした行のみを出力するプログラムまたは機能。

なお，基本型の別名が定義されているのは可搬性（移植性）のためであるが，実際はほとんど役に立っていない。

構造体

構造体を作る場合は，通常 typedef を用いて型定義を併用して大文字の型名を付ける。例えば MIDIOUTCAPS 構造体の場合は mmsystem.h 内で次のように定義されている[*13]。

リスト 3.3 MIDIOUTCAPS 構造体の定義

```
typedef struct tagMIDIOUTCAPSA {
    WORD    wMid;                  /* manufacturer ID */
    WORD    wPid;                  /* product ID */
    MMVERSION vDriverVersion;      /* version of the driver */
    CHAR    szPname[MAXPNAMELEN];  /* product name (NULL terminated string) */
    WORD    wTechnology;           /* type of device */
    WORD    wVoices;               /* # of voices (internal synth only) */
    WORD    wNotes;                /* max # of notes (internal synth only) */
    WORD    wChannelMask;          /* channels used (internal synth only) */
    DWORD   dwSupport;             /* functionality supported by driver */
} MIDIOUTCAPSA, *PMIDIOUTCAPSA, *NPMIDIOUTCAPSA, *LPMIDIOUTCAPSA;
typedef MIDIOUTCAPSA MIDIOUTCAPS;
typedef PMIDIOUTCAPSA PMIDIOUTCAPS;
typedef NPMIDIOUTCAPSA NPMIDIOUTCAPS;
typedef LPMIDIOUTCAPSA LPMIDIOUTCAPS;
```

もし型定義を行わないと，毎回 struct tagMIDIOUTCAPSA のように長いキーワードを打たなければならず面倒であるので，構造体の定義においては（全て大文字で）別名が付けられるのが通例である。

この定義には，MIDIOUTCAPS の前に P，NP，LP[*14] という接頭辞が付いた型名も宣言されているが，これらはポインタ用の型名を表している（従って，これらを使わずに *MIDIOUCAPS と書いてもよい）。Win32（32 ビット環境用の Windows のことであり，Windows 95 を含めてそれ以降は全て 32 ビットである）であれば 3 つとも同じであり，従って Win32 環境ではどれを使用しても変わらないが，後方互換性のために LP を使うのが通例である。

[*13] 実際には，#ifdef による条件分岐が相当数あり，90 行近くに及ぶ長い定義がされている。具体的には，OS が 32 ビット Windows であるかということ（他の OS は考えていなさそうだが，どうやらこのファイル Macintosh のことだけは考えているようだ。詳しいことは分からないが…）と，Unicode フラグが立っているかどうかで場合分けしている。Unicode 処理に関してはまだ詳しく知らなくてよいだろう。

[*14] それぞれ Pointer，Near Pointer，Long Pointer の略であろう。

定数

定数は #define 文で定義される。例えば MMSYSERR_NOERROR 関連の定数は mmsystem.h 内で次のように定義されている。

リスト 3.4 MMSYSERR 系の定数の定義

```
#define MMSYSERR_BASE          0

/* general error return values */
#define MMSYSERR_NOERROR      0                    /* no error */
#define MMSYSERR_ERROR        (MMSYSERR_BASE + 1) /* unspecified error */
#define MMSYSERR_BADDEVICEID  (MMSYSERR_BASE + 2) /* device ID out of range */
#define MMSYSERR_NOTENABLED   (MMSYSERR_BASE + 3) /* driver failed enable */
#define MMSYSERR_ALLOCATED    (MMSYSERR_BASE + 4) /* device already allocated */
#define MMSYSERR_INVALHANDLE  (MMSYSERR_BASE + 5) /* device handle is invalid */
#define MMSYSERR_NODRIVER     (MMSYSERR_BASE + 6) /* no device driver present */
...以下略...
```

これらは名前の通り，マルチメディア関連のシステムエラーを表しているようだ（ただし，MMSYSERR_NOERROR[*15]だけはエラーが起こらなかったことを示している。）。これらの定数の実際の値が変わることはまず考えられないが，プログラムの分かりやすさも兼ねて定数名で記述するのがよいだろう。例えば，リスト 3.1 のように，条件判定は `res` や `res != 0` とは書かずに，`res != MMSYSERR_NOERROR` のように記述する[*16]。

3.2　ノートオンとノートオフ

ノートオンとノートオフは最も代表的な MIDI メッセージ（あるいは MIDI イベントともいう）である。ノートオン[*17] は音を鳴らすメッセージで，その音を止めるメッセージがノートオフである。あるいは，ノートオンとノートオフは鍵盤の押し離しに直接対応しているともいえる。従ってノートオンとノートオフは必ず対になって出現する（さもないと，音が鳴りっぱなしになる —— もしそうなった場合は，当然のことであるが，電源を落とせば確実に音を消せる）。

まず，実際に送信されるノートオンやノートオフのメッセージの形式を見てみよう。

[*15] MultiMedia SYStem ERRor, NO ERROR の略であろう。
[*16] なお，res は result の略である。
[*17] ノート (note) とは音符のこと。

3.2.1　ノートオンメッセージ

1 つのノートオンメッセージは 3 バイトで構成される（図 3.2）。最初の 1 バイトはステータスバイトと呼ばれ，ノートオンイベントの場合は先頭の 4 ビットが 0x9 になる（この値で他の MIDI イベントと区別される）。1 バイト目の後半 4 ビットはチャンネル番号を表すが，詳細は後ほど 3.3 節で説明するので，ここでは単に 0x0（チャンネル 1）でよい。

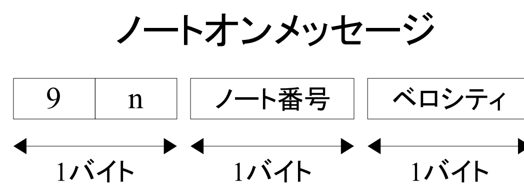


図 3.2　ノートオンメッセージ

後半の 2 バイトはデータバイトと呼ばれ，実際のノートオンのデータを表している。そのうち 2 バイト目はノート番号で，音の高さを表す。ノート番号は，ピアノの中央のドを 60 (=0x3C) として，半音上がれば 1 大きく，半音下がれば 1 小さくなる。ノート番号には先頭の 1 ビットを除く 7 ビット[*18]，すなわち 0〜127 (0x0〜0x7F) の値を指定できるため，およそ $128/12 = 10$ オクターブ半 の音域が利用できる。これは，中央のドを C4（4 オクターブ目の C）とすると[*19]，C-1 から G9 までを表すことができる[*20]。

最後の 3 バイト目はベロシティ[*21] といい，音の大きさを表す。より正確には，鍵盤を押したときの速さ，あるいは楽器を弾いた強さであるが，単純に音の大きさだと考えて差し支えないだろう。ベロシティには 0〜127 (0x0〜0x7F) の値を指定でき，1 が最も小さい音で 127 が最も大きな音になる（ベロシティ 0 はノートオフイベントを表す。詳しくは次の 3.2.2 節を参照）。

例えば，C4 (60=0x3C) の音を中ぐらいの強さ (64=0x40) として，チャンネル 1 (0=0x0) で音を鳴らす場合は次のバイト列を送ればよい。

90 3C 40

実際には 2 進数のバイナリデータがやりとりされるので，上記の 16 進数表記のバイト

[*18] データバイトの 1 ビット目は 0 と決められているため，利用可能なのは 7 ビットとなる。詳しくは 3.3 節を参照。

[*19] ピアノの最低音 C を C0 とした場合。オルガンの最低音 C を C0 として，中央のドを C3 と表記する場合もある。本テキストでは中央のドの表記を C4 に統一する。

[*20] 英語の音名表記は，C=ド，D=レ，E=ミ，F=ファ，G=ソ，A=ラ，B=シ，である。

[*21] ベロシティ (velocity) とは速度のこと。

列を 2 進数で表記すれば,

$$10010000\ 00111100\ 01000000$$

になる。

3.2.2　ノートオフメッセージ

ノートオンメッセージと同様に，ノートオフメッセージは 3 バイトで構成される（図 3.3)。

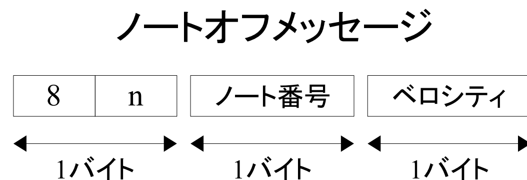


図 3.3　ノートオフメッセージ

形式的にはノートオンメッセージと全く同じであるが，1 バイト目（ステータスバイト）のイベントタイプ（先頭 4 ビット）が 9 から 8 に変わっていることに注意しよう。

ここで指定するチャンネル番号とノート番号は，音を止めたいノートオンで指定した値と同じ値を指定する。

ベロシティは音が消えていく速さを表し，鍵盤から手を離す速度に対応している（リリースタイムを調節できる)。ただし，対応していない鍵盤も多く，SMF などの MIDI データファイルではあまり利用されていない。

また，ノートオンメッセージでベロシティを 0 に指定した場合も，音が鳴らないということから，ノートオフイベントとして扱われる。実際にはこちらの方が多用されているようだ。

次節では，これらのメッセージを実際に送信するプログラムを作る。

3.2.3　出力デバイスのオープンとクローズ

参考のため，2.5 節の midiout.c（リスト 2.2）を行番号付きでもう一度示す。

リスト 3.5 サンプルプログラム再掲 (midiout.c)

```
1    #include <windows.h>
2    #include <mmsystem.h>
3
4    int main(void) {
```

```
 5      HMIDIOUT hMidiOut;
 6
 7      midiOutOpen(&hMidiOut, MIDI_MAPPER, 0, 0, 0);
 8
 9      midiOutShortMsg(hMidiOut, 0x007f3c90);
10      Sleep(1000);
11      midiOutShortMsg(hMidiOut, 0x007f3c80);
12
13      midiOutReset(hMidiOut);
14      midiOutClose(hMidiOut);
15
16      return 0;
17    }
```

以下では各処理の説明を行っているが，このプログラムを見ながら読むと全体の流れが理解しやすいだろう。

midiOutOpen()

メッセージを送信する前に，まず送信したいデバイスを開いておく必要がある。そのために，出力デバイスを開く API として midiOutOpen() が用意されている。midiOutOpen() のプロトタイプ宣言は次の通りである。

```
MMRESULT midiOutOpen(
  LPHMIDIOUT lphmo,
  UINT       uDeviceID,
  DWORD_PTR  dwCallback,
  DWORD_PTR  dwCallbackInstance,
  DWORD      dwFlags
);
```

この関数は，開くデバイスを uDeviceID で指定し，デバイスハンドル[*22]を lphmo で受け取るようになっている。エラーの有無が関数の戻り値として返される。

では，引数を一つずつ説明しよう。最初の引数 lphmo には，HMIDIOUT ハンドルのアドレスを渡す[*23]。もしデバイスのオープンに成功すれば，この変数に開いたデバイスのハンドルが格納される。

2 番目の引数 uDeviceID には，開きたい出力デバイスの ID 番号を指定する。ここには 3.1 節で調べた ID を指定すればよい。もしコントロールパネルで設定している標準の出力デバイスを使用したい場合には，定数 MIDI_MAPPER を指定する。

[*22] ハンドル (handle) は ID と同じような意味で，操作対象を一意に特定可能な識別子である。

[*23] HMIDIOUT は Handle, MIDI OUTput の略であろう。LPHMIDIOUT はその HMIDIOUT 型の Long Pointer である。

3 番目から 5 番目の引数は，コールバック機構を使用する場合に利用する。今はコールバックを利用しないので，とりあえず全て 0 を指定しておけばよいだろう。

もし関数が成功すると（出力デバイスを開くことができると），戻り値として MMSYSERR_NOERROR が返される。エラーがあった場合の値はリファレンスを参照して欲しい。

midiOutClose()

一度開いたデバイスは必要な通信が終われば閉じなければならない。出力デバイスを閉じるには midiOutClose() 関数を用いる。midiOutClose() のプロトタイプ宣言は以下のようになっている。

```
MMRESULT midiOutClose(
  HMIDIOUT hmo
);
```

閉じたい出力デバイスのハンドルを hmo で指定する。エラーが無ければ MMSYSERR_NOERROR が，エラーが発生すれば他の値が返る。

midiOutReset()

ノートオンに対するノートオフが無いままデバイスを閉じたり（音が鳴り続ける），未処理のバッファがあって midiOutClose() に失敗したりすると困る[*24] ので，midiOutClose() を呼び出す前に一度 midiOutReset() を呼び出しておくとよい。これは全ての音を止める[*25]。

midiOutReset() のプロトタイプ宣言は次の通りである。

```
MMRESULT midiOutReset(
  HMIDIOUT hmo
);
```

hmo には出力デバイスのハンドルを指定する。関数が成功すれば MMSYSERR_NOERROR が，関数が失敗すれば他の値が返る。

従って，MIDI 出力デバイスを開いて閉じるだけのプログラムはリスト 3.6 のようになる。

リスト 3.6 MIDI 出力デバイスのオープン/クローズ (midiout-openclose.c)

[*24] 4.4.2 節で詳説するシステムエクスクルーシブメッセージを送信している場合に発生する。詳しくはリファレンスを参照のこと。

[*25] 正確には，全チャンネルの全ノートをオフにし，全チャンネルのサステインコントローラをオフにする。

```
#include <windows.h>
#include <mmsystem.h>

int main(void) {
  HMIDIOUT hMidiOut;

  if ( midiOutOpen(&hMidiOut, MIDI_MAPPER, 0, 0, 0) != MMSYSERR_NOERROR ) {
    printf("Cannot open MIDI output device.\n");
    return 1;
  }

  midiOutReset(hMidiOut);
  midiOutClose(hMidiOut);

  return 0;
}
```

これはリスト 3.5 の 9-11 行目を除いたプログラムに相当する。ただし，デバイスのオープン時にエラー処理を加えてある。

3.2.4 MIDI メッセージの送信

以上で MIDI メッセージを送信するための準備が整った。MIDI メッセージには色々あるが，ノートオンやノートオフはチャンネルメッセージ*[26] の一つであるため，ここでは midiOutShortMsg() 関数を使用する。

midiOutShortMsg() のプロトタイプ宣言は次の通りである。

```
MMRESULT midiOutShortMsg(
  HMIDIOUT hmo,
  DWORD    dwMsg
);
```

hmo には送信先の出力デバイスのハンドルを指定する。dwMsg には送信するメッセージを与えるが，後述する。成功時には MMSYSERR_NOERROR が，失敗すればエラーが返る。

dwMsg は DWORD 型であるから，4 バイトの符号無し整数 (unsigned long int) である（windef.h 内にそう書かれている）。それに対して，ノートオンやノートオフなどの MIDI メッセージは 3 バイトで構成されるので，バイト数が異なっている。残りの 1 バイトは使われず，与えても無視されるので，0x00 を指定すればよい。例えばノート番号 60 (=0x3C) をベロシティ 64 (=0x40) で発音させる場合，バイト列は 0x903C40 となるが，

*[26] チャンネルメッセージについては 3.3 節で詳しく説明する。

最後に 1 バイト分の 0x00 を充填して 0x903C4000 とすればよいだろう。

ただし，関数に渡すときにバイトオーダーの問題が生じる。上記の 0x903C4000 という並び順はビッグエンディアンであるが，この midiOutShortMsg() 関数にはリトルエンディアンでデータを渡さなければならない（midiOutShortMsg() がそういう風に作られているからで，リファレンスにそのように渡すようにと書いてある)。従って，各バイトごとに逆順にして，0x00403C90 を渡せばよい。バイトオーダーやエンディアンに関しては，詳しくはプログラミングメモ 5 を読んで欲しい。

従って，上記データでノートオンしてノートオフするには，次のように書けばよい（オフベロシティ=0x7F とする)。

```
midiOutShortMsg(hMidiOut, 0x00403c90);
midiOutShortMsg(hMidiOut, 0x007f3c80);
```

あるいは，ノートオンメッセージのベロシティ 0 を使ってノートオフする場合は，

```
midiOutShortMsg(hMidiOut, 0x00403c90);
midiOutShortMsg(hMidiOut, 0x00003c90);
```

と書く。ノートオフのオフベロシティを気にしない場合は，こちらを使う方が良いようだ[*27]。

上記コードをそのまま書いても，ノートオン直後にノートオフが来るために音が鳴らないだろう。Sleep() を呼び出してプログラム[*28] を一時中断（サスペンド）すればよい。Sleep() のプロトタイプ宣言は次の通りである。引数にはサスペンドする時間をミリ秒で指定する。

```
void Sleep(
  DWORD dwMilliseconds
);
```

C4 を 1 秒間鳴らし，続いて D4 を 1 秒間鳴らすプログラムをリスト 3.7 に示す。

リスト 3.7 MIDI メッセージの送信 (sendmidimsg.c)

```
#include <windows.h>
#include <mmsystem.h>

int main(void) {
  HMIDIOUT hMidiOut;
```

[*27] ランニングステータス（5.2.3 節参照）を利用できるため。

[*28] Sleep() は正確にはスレッド単位で実行をサスペンドするが，ここで作るプログラムにはスレッドが 1 つしかないので結局は同じことである。

```
    midiOutOpen(&hMidiOut, MIDI_MAPPER, 0, 0, 0);

    /* C4 (60=0x3C) を一秒間鳴らして止める */
    midiOutShortMsg(hMidiOut, 0x00403c90);
    Sleep(1000);
    midiOutShortMsg(hMidiOut, 0x00003c90);

    /* D4 (62=0x3E) を一秒間鳴らして止める */
    midiOutShortMsg(hMidiOut, 0x00403e90);
    Sleep(1000);
    midiOutShortMsg(hMidiOut, 0x00003e90);

    midiOutReset(hMidiOut);
    midiOutClose(hMidiOut);

    return 0;
  }
```

課題 3

メジャースケール（ドレミファソラシド）の音階を，各音を 1 秒間ずつ鳴らすプログラムを作成せよ。

プログラミングメモ 5 — バイトオーダー（エンディアン）

複数バイトのデータをメモリに格納する場合，各バイトを格納する順序が 2 通りある。すなわち，元データを 4 バイトの 0x12345678 とすると，それをそのままの順序で格納すると 0x12, 0x34, 0x56, 0x78 となり（ビッグエンディアン），逆の順序で格納すると 0x78, 0x56, 0x34, 0x12 となる（リトルエンディアン）。また，このような，各バイトの格納順序のことをバイトオーダー，またはエンディアンという。

C 言語でプログラミングを行うとき，通常はバイトオーダーを気にする必要はない。Intel 系の CPU ではリトルエンディアンが採用されており，例えば 0x12345678 はメモリの格納順序は 0x78, 0x56, 0x34, 0x12 であるが，シフト演算子で左に 4 ビットシフトした場合は通常通り 0x23456780（メモリの格納順序は 0x80, 0x67, 0x45, 0x23）になる。

しかし，ファイルに書き込んだり通信でデータを送受信する場合，アプリケーション間や OS 間でバイトオーダーを統一する必要がある。そのために，MIDI では全てビッグエンディアンを使うように規定されており，通信データや SMF ファイル内のデータは全てビッグエンディアンの順序で転送/格納される。

midiOutShortMsg() 関数では，送る 4 バイトのデータをリトルエンディアンで指定す

る。このように指定すると，Windows ではリトルエンディアンを採用している[29] ので，そのデータをもう一度リトルエンディアンでメモリに格納したさいにメモリ上ではビッグエンディアンの並び順になっている。そして，ライブラリの実装がおそらくこのデータを先頭から 3 バイト分だけ 1 バイトずつ読んで送信しているのだろう。少しややこしいが，このようにすると MIDI の規格に沿った形，すなわちビッグエンディアンでデータが送信される。

3.3　チャンネルメッセージ

前 3.2 節ではノートオンメッセージとノートオフメッセージを扱った。MIDI メッセージには他にも様々なメッセージが存在する。本 3.3 節では，MIDI メッセージにどのようなメッセージがあるのかを概説し，実際にプログラムチェンジメッセージの送信プログラムを作成する。

3.3.1　MIDI メッセージの分類

図 3.4 に MIDI メッセージの基本的な分類を示す。

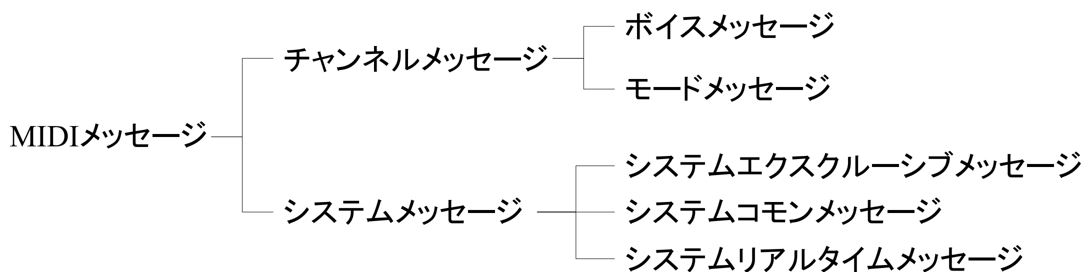


図 3.4　MIDI メッセージの分類

MIDI メッセージにはチャンネルメッセージとシステムメッセージがある。チャンネルメッセージは演奏情報を扱い，システムメッセージは MIDI システム全体に関する情報を扱う。

本節ではチャンネルメッセージについてのみ解説する。

3.3.2　チャンネルメッセージ

チャンネルメッセージは最初にステータスバイトが来て，次にデータバイトが続く（図 3.5）。ステータスバイトにはメッセージの種類を指定し，その種類によってデータのバイ

[29] Windows が動作する CPU の主流がリトルエンディアンの Intel 製 CPU であるため。

ト数が A）2 バイトか B）1 バイトかが決まっている。

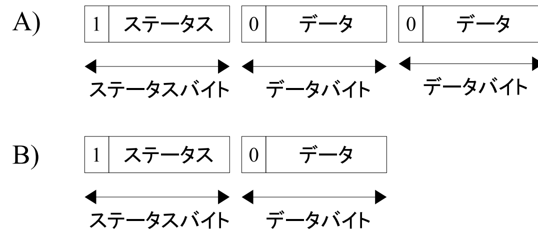


図 3.5　チャンネルメッセージ

ステータスバイトの最上位ビット (MSB) は必ず 1 になっており，データバイトの最上位ビットは必ず 0 になっている。これにより，最上位ビットを見ればそのバイトがステータスバイトであるかデータバイトであるかを容易に判断できるようになっている（その分，表現可能な値が 8 ビット分 (0〜255) から 7 ビット分 (0〜127) に減少している）。

ステータスバイトの残りの 7 ビットは，メッセージの種類に 3 ビット，チャンネル番号に 4 ビットが割り当てられている（図 3.6）。メッセージの種類は 3 ビットであるから，2^3 で 8 種類のメッセージを指定できる（先頭の 1 ビットと合わせて，0x8〜0xF と記述する）。そのうち 0xF はシステムメッセージに割り当てられており（その場合，後半 4 ビットはチャンネル番号ではない），残りの 7 種類がチャンネルメッセージである。

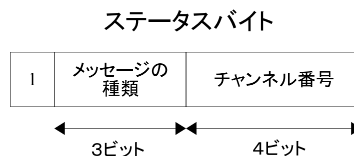


図 3.6　ステータスバイトの構成

チャンネルメッセージではチャンネルごとの演奏情報を指定し，またチャンネル番号を必ず書く必要があり，そのためにチャンネルメッセージと呼ばれる。チャンネルは一つの楽器パートに相当すると考えればよいだろう[*30]。ステータスバイト後半のチャンネル番号は 4 ビットであるから，2^4 で 16 個のチャンネルを指定できる。

チャンネルメッセージはさらにボイスメッセージとモードメッセージに分けられる。ボイスメッセージは，例えばノートオンやノートオフのように，演奏情報や演奏設定という「どのように再生するか」という情報を表すのに，モードメッセージは MIDI 機器の受信設定のように「どのようにボイスメッセージを送受信するか」という情報を表すのに用いられる。

[*30] MIDI 機器は MIDI THRU 端子などを用いて複数つなげることができる。その場合に，処理するチャンネルを各 MIDI 機器に設定することで，チャンネルごとに演奏機器を分けることができる。

以下にボイスメッセージの一覧を載せる（表 3.1）。ステータスバイトの n は，任意のチャンネル番号（4 ビット）を指定可能という意味である。また，プログラムチェンジとチャンネルプレッシャーではデータバイトの 2 バイト目はない。ピッチベンド値の LSB は最下位バイト，MSB は最上位バイトである[*31]。

表 3.1　ボイスメッセージ

メッセージの種類	ステータスバイト	第 1 データバイト	第 2 データバイト
ノートオフ	0x8n	ノート番号	オフベロシティ
ノートオン	0x9n	ノート番号	ベロシティ
ポリフォニックキープレッシャー	0xAn	ノート番号	プレッシャー値
コントロールチェンジ	0xBn	コントロール番号	コントロール値
プログラムチェンジ	0xCn	プログラム番号	——
チャンネルプレッシャー	0xDn	プレッシャー値	——
ピッチベンドチェンジ	0xEn	ピッチベンド値 (LSB)	ピッチベンド値 (MSB)

また，以下にモードメッセージの一覧を載せる（表 3.2）。これらのステータスバイトは全て 0xBn であり，コントロールチェンジと同じになっているが，モードメッセージはコントロールチェンジの一部ではないので注意が必要である（あくまでコントロールチェンジ領域の一部を間借りしているだけである）。第 2 データバイトの 0x00 は，何の意味もないダミーバイトである（単に無視される）。

表 3.2　モードメッセージ

メッセージの種類	ステータスバイト	第 1 データバイト	第 2 データバイト
オールサウンドオフ	0xBn	0x78 (120)	0x00
リセットオールコントローラ	0xBn	0x79 (121)	0x00
ローカルコントローラ	0xBn	0x7A (122)	オフまたはオン
オールノートオフ	0xBn	0x7B (123)	0x00
オムニオフ	0xBn	0x7C (124)	0x00
オムニオン	0xBn	0x7D (125)	0x00
モノモードオン	0xBn	0x7E (126)	チャンネル数
ポリモードオン	0xBn	0x7F (127)	0x00

3.3.3　プログラムチェンジの送信

それでは，ボイスメッセージの一つであるプログラムチェンジを実際に使用してみよう。プログラムチェンジは発音される音色を切り替えるメッセージである。ステータスバ

[*31] MSB (Most Significant Byte/Bit) や LSB (Least Significant Byte/Bit) の B はバイトを表すときとビットを表すときがあるので注意。

イトは 0xCn であり，データバイトは 1 バイトだけで，プログラム番号を指定する（図 3.7)。

図 3.7　プログラムチェンジメッセージ

プログラム番号は 1 バイトのデータバイトであるから，7 ビットで 128 種類の音色を切り替えることができる[*32]。プログラム番号と音色の種類の対応関係は，原則としては音源に依存して互換性はないが，基本的な音色に限っては GM 規格に対応していれば一致している[*33]。GM 規格の音色マッピングについては，MIDI 関連の本の他，http://www.b-sharp.com/midi/spec1/gm002.html や http://www.pluto.dti.ne.jp/~daiki/Midi/IL_ProgramNum.html といった Web ページにも載っているので参照して欲しい。

チェロ (cello) とチューブラーベル (tublar bells) の音色で C4 を 1 秒間ずつ発音させるプログラムをリスト 3.8 に示す（GM 音源を仮定しているが，他の音色であっても音色が変わればよい)。チェロのプログラム番号は 43 であるが，実際に送信するデータはそれから 1 を引いた数 (42=0x2a) になることに注意しよう。

リスト 3.8 プログラムチェンジの送信 (progchange.c)

```
#include <windows.h>
#include <mmsystem.h>

int main(void) {
  HMIDIOUT hMidiOut;

  midiOutOpen(&hMidiOut, MIDI_MAPPER, 0, 0, 0);

  /* プログラムチェンジ: Cello (42=0x2a) */
  midiOutShortMsg(hMidiOut, 0x00002ac0);
  midiOutShortMsg(hMidiOut, 0x00403c90);
  Sleep(1000);
  midiOutShortMsg(hMidiOut, 0x00003c90);

  /* プログラムチェンジ: Tublar Bells (14=0x0e) */
```

[*32] 現在の MIDI 音源では 128 種類以上の音源を積んでいることが多い。その場合はバンクセレクトメッセージを併用して音色を指定する。

[*33] バンクセレクトを行わない場合のメロディ用 128 音色とリズム用 47 音色が統一されている。ただし全く同じ音色で発音されるわけではなく，音源の特徴として細かな差異がある。

```
  midiOutShortMsg(hMidiOut, 0x00000ec0);
  midiOutShortMsg(hMidiOut, 0x00403c90);
  Sleep(1000);
  midiOutShortMsg(hMidiOut, 0x00003c90);

  midiOutReset(hMidiOut);
  midiOutClose(hMidiOut);

  return 0;
}
```

もし GM 音源に対応しているのにも関わらずチェロやチューブラーベルの音色で鳴らない場合は，リスト 3.9 のように GM システムオンメッセージを最初に送信すればよいだろう（詳しい説明は省く）。

リスト 3.9 GM システムオンの送信 (gmsystemon.c)

```
#include <string.h>
#include <windows.h>
#include <mmsystem.h>

int main(void) {
  HMIDIOUT hMidiOut;
  MIDIHDR header;
  BYTE GMSystemOn[] = {0xf0, 0x7e, 0x7f, 0x09, 0x01, 0xf7};

  midiOutOpen(&hMidiOut, MIDI_MAPPER, 0, 0, 0);

  /* ヘッダに GM システムオンメッセージの情報を格納 */
  ZeroMemory(&header, sizeof(MIDIHDR));
  header.lpData = (LPSTR)GMSystemOn;
  header.dwBufferLength = sizeof(GMSystemOn);
  header.dwFlags = 0;

  /* システムエクスクルーシブメッセージの送信 */
  midiOutPrepareHeader(hMidiOut, &header, sizeof(MIDIHDR));
  midiOutLongMsg(hMidiOut, &header, sizeof(MIDIHDR));
  Sleep(100);
  midiOutUnprepareHeader(hMidiOut, &header, sizeof(MIDIHDR));

  /* プログラムチェンジ: Cello (42=0x2a) */
  midiOutShortMsg(hMidiOut, 0x00002ac0);
  midiOutShortMsg(hMidiOut, 0x00403c90);
  Sleep(1000);
  midiOutShortMsg(hMidiOut, 0x00003c90);

  /* プログラムチェンジ: Tublar Bells (14=0x0e) */
  midiOutShortMsg(hMidiOut, 0x00000ec0);
```

```
    midiOutShortMsg(hMidiOut, 0x00403c90);
    Sleep(1000);
    midiOutShortMsg(hMidiOut, 0x00003c90);

    midiOutReset(hMidiOut);
    midiOutClose(hMidiOut);

    return 0;
}
```

3.3.4 複数チャンネルの使用

複数の楽器（音色）を同時に使用する場合，複数のチャンネルを使用するとよい。例えば次のようにプログラムチェンジで各チャンネルに音色を設定する。

```
/* 0x18: Acoustic Guitar (nylon) */
midiOutShortMsg(hMidiOut, 0x000018c0);
/* 0x00: Acoustic Grand Piano */
midiOutShortMsg(hMidiOut, 0x000000c1);
```

ここではチャンネル 1 にプログラム番号 25 の Acoustic Guitar (nylon) を，チャンネル 2 にプログラム番号 0 の Acoustic Grand Piano を指定している[*34]。

また，GM 音源のチャンネル 10 はリズム楽器用の音色が鳴るようになっている（チャンネル 10 以外は全て通常のメロディ楽器用である）。リズム用の楽器には基本的に音程が無いため，チャンネル 10 ではプログラムチェンジで楽器（音色）を指定するのではなく，音を鳴らすノートオンメッセージのノート番号で音色を指定するようになっている。GM 規格では 35〜81 の範囲のノート番号にリズム用の音色が割り当てられている（具体的な表は 3.3.3 節で挙げた Web サイトなどを参照）。なお，チャンネル 10 ではノートオフメッセージは無視される。

課題 4

メロディ用のチャンネル（1 つか 2 つ）とリズム用のチャンネルを使用し，2 または 4 小節分の簡単な音楽を鳴らすプログラムを作成せよ。ただし，テンポは 60 とする（8 分音符が 500ms の長さになる）。

[*34] チャンネルは 1〜16 の 16 チャンネルあるが，実際に送信するデータは 0〜15 で指定するために値が 1 ずれることに注意。

プログラミングメモ 6 — 引数の型と型キャスト

今までのソースコードの例では，関数に渡す引数の型を簡略化して扱ってきた。例えば，midiOutOpen() のプロトタイプ宣言は次のようになっている。

```
MMRESULT midiOutOpen(
  LPHMIDIOUT lphmo,
  UINT       uDeviceID,
  DWORD_PTR  dwCallback,
  DWORD_PTR  dwCallbackInstance,
  DWORD      dwFlags
);
```

そして，midiOutOpen() 関数の引数は今まで次のように指定してきた。

```
midiOutOpen(&hMidiOut, MIDI_MAPPER, 0, 0, 0);
```

最初の引数を見てみよう。hMidiOut は HMIDIOUT 型であるから，&hMidiOut の型は HMIDIOUT*である。しかしプロトタイプ宣言では LPHMIDIOUT 型であるので，両者の型は異なっているといえる。ただし，LPHMIDIOUT 型はそもそも HMIDIOUT*型であると宣言されており[*35]，また HMIDIOUT 型へのポインタという意味から考えても両者の型は結局は同じになる。それゆえにコンパイル時にエラーも警告も出ないのであるが，Windows API が本来提供しようとしている移植性を考えると，LPHMIDIOUT 型に対して HMIDIOUT*型を指定するのは良い作法だとは言えないだろう。従って，ここは型キャストを用いて，`(LPHMIDIOUT)&hMidiOut` として LPHMIDIOUT 型にキャストして渡すのが行儀が良い。

2 番目の引数は UINT 型である。UINT 型であるから，マイナスにならない数字を指定している限りキャストしなくても問題ない。もしキャストしたければ `(UINT)MIDI_MAPPER` とする（MIDI_MAPPER の場合は API が提供している定数であるから，キャストは必要ない)。また，ここに直接数値を記述する場合は，定数サフィックス U を付けて 0U や 1U などと書けばよいだろう。

3 番目の引数は DWORD_PTR 型である。midiOutOpen() のドキュメンテーションを読むと，コールバックを必要としない場合には NULL を指定するように書いてあるので，ここに直接 0 を指定しているのは良くないだろう。NULL は 0 と定義されているのでエラーや誤動作は起こらないし，NULL が 0 以外の値に変更されることはまずないだろうか

[*35] LP (Long Pointer) なので，正確には `typedef HMIDIOUT FAR *LPHMIDIOUT;` と FAR 付きで宣言されている。

ら，現実的に問題が起こることはないだろうが，行儀が良いとは言えない。そこで，ここには 0 ではなく NULL を指定する。しかし NULL は DWORD_PTR 型ではない[36] ので，コンパイラや環境によっては警告が表示される場合がある。そこで，(DWORD_PTR)NULL とキャストして渡せばよいだろう。

4 番目の引数も DWORD_PTR 型である。ドキュメンテーションには，この値を指定する必要がない場合に渡すデータは明記されていないので，ここは 0 でよいだろう。もし型が気になる場合は (DWORD_PTR)0 とキャストしてもよい。

最後の引数は DWORD 型である。ドキュメンテーションを読むと，コールバック機構がない場合は CALLBACK_NULL を指定するように書いてある。CALLBACK_NULL は mmsystem.h 内で 0x00000000L と定義されている[37] ため，単に 0 と指定しても問題はなかったが，ここは仕様書通りに CALLBACK_NULL を指定するのがよいだろう。

以上をまとめると，midiOutOpen() は次のように呼び出すのが行儀が良いといえるだろう。

```
midiOutOpen( (LPHMIDIOUT)&hMidiOut, MIDI_MAPPER,
             (DWORD_PTR)NULL, (DWORD_PTR)0, CALLBACK_NULL );
```

ただし，必ずこのように指定すべきというわけではない。現実的には，少なくとも Win32 環境であればこれらの値が変わることはまずないであろうから，Win32 環境でのみ動作させればよいというのであればコンパイルが通って動作さえすればよいだろう。しかし，変数や引数の型については，常に注意を払ってプログラミングをした方がよい。それがバグの減少にもつながるであろう。

プログラミングメモ 7 — 値渡しと参照渡し

関数に対する引数の渡し方には，主なものとして参照渡し/参照呼び出し (call-by-reference/pass-by-reference) と値渡し/値呼び出し (call-by-value/pass-by-value) の 2 種類がある[38]。

参照渡しの場合，関数の呼び出し時に渡す引数（実引数）と，関数側で渡された引数（仮引数）は同じ実体（オブジェクト）を指す。すなわち，呼び出された関数内で値を変更すれば，呼び出し元の値も変更される。この例を C++ 言語で記述すると次のようになる（C 言語では参照渡し用の構文は存在せず，従って参照渡しを行うことはできない）。

[36] NULL は windef.h 内で `#define NULL 0` または `#define NULL ((void *)0)` として定義されている。

[37] L は long int 型または long double 型を意味する定数サフィックスである。

[38] 他に名前渡し/名前呼び出し (call-by-name/pass-by-name) もある。

```
void func(int &b) {
  b += 10;
  printf("func: %d\n", b);  // -> 20
}

int main(void) {
  int a = 10;

  printf("main: %d\n", a);  // -> 10
  func(a);
  printf("main: %d\n", a);  // -> 20

  return 0;
}
```

それに対して，値渡しの場合，実引数と仮引数は異なる実体を指す。すなわち，呼び出された関数内で値を変更しても，呼び出し元の値は変わらない。C 言語における例を次に示す。

```
void func(int b) {
  b += 10;
  printf("func: %d\n", b);  // -> 20
}

int main(void) {
  int a = 10;

  printf("main: %d\n", a);  // -> 10
  func(a);
  printf("main: %d\n", a);  // -> 10

  return 0;
}
```

値渡しの場合，データは一度コピーされて渡されることになる[*39]。なぜなら，これは具体的には変数 a と変数 b が指しているメモリのアドレスが異なるということであり，新しく作成される変数 b が指すメモリには（最初はゴミデータが入っているので）渡されたデータを格納する必要があるからである。

さて，先ほど述べたように C 言語には参照渡しの機能は用意されておらず，参照渡しが使えない場合は呼び出された側で呼び出し元の変数の値を変更することができない。変更

[*39] アセンブリ言語で扱うような低レベルの実際の動作では，スタックに積むという表現も使う。

したい値が 1 つだけであれば戻り値を使って代入すればよいが，値が複数あったり戻り値を他の用途に使ったりする場合は困ってしまう。そのため，C 言語ではデータのアドレス（変数のポインタ）を値渡しすることによって参照渡しを代替するのが通例である。その例を次に示す。

```
void func(int *b) {
  *b += 10;
  printf("func: %d\n", *b);  // -> 20
}

int main(void) {
  int a = 10;

  printf("main: %d\n", a);  // -> 10
  func(&a);
  printf("main: %d\n", a);  // -> 20

  return 0;
}
```

この場合，値渡しされるのはデータのアドレスであるから，「データのアドレス」というデータ自体はコピーされる。しかし，アドレスが同じであれば結局参照される実体は一緒であるから，呼び出し側の値も変更することができる。

以上のように，呼び出される関数側で呼び出し元の値を変更したい場合は参照渡し（あるいはアドレスの値渡し）をし，その必要がない（あるいは元の値が変更されては困る）変数は値渡しをするように関数を設計すればよい。なお，値渡しでは必ずデータのコピーが行われるため，大量のデータ（例えば 10MB のバイト列）を値渡しすることは（値を変更しない場合でも）通常避けられる。

第 4 章

MIDI の入力

本章では MIDI 機器から PC への入力を扱う。

4.1 入力デバイスの一覧を得る

MIDI の入力デバイスの一覧を得るには midiInGetNumDevs() 関数と midiInGetDevCaps() 関数を使う。midiInGetNumDevs() は MIDI 入力デバイス数を返す関数であり，midiInGetDevCaps() は与えられたデバイス ID に対してそのデバイスの情報を返す関数である。これらの関数の使い方は，出力デバイスを調べるための midiInGetNumDevs() や midiInGetDevCaps() と同様である。

midiInGetNumDevs() および midiInGetDevCaps() のプロトタイプ宣言はそれぞれ次の通りである。

```
UINT midiInGetNumDevs(void);

MMRESULT midiInGetDevCaps(
  UINT_PTR     uDeviceID,
  LPMIDIINCAPS lpMidiInCaps,
  UINT         cbMidiInCaps
);
```

課題 5

利用可能な MIDI 入力デバイス名を全て表示するプログラムを作成せよ。必要であれば 3.1 節のリスト 3.1 を参考にしてもよい。

プログラミングメモ 8 — ジェネリックテキストマッピング

各入力デバイスの情報を得るために MIDIINCAPS 構造体が使われる。この MIDIINCAPS 構造体の定義を見てみよう。少し長くなるが，以下に mmsystem.h に記述されている定義をそのまま掲載する（リスト 4.1）。

リスト 4.1 MIDIINCAPS 構造体の定義

```
 1   /* MIDI input device capabilities structure */
 2   #ifdef _WIN32
 3
 4   typedef struct tagMIDIINCAPSA {
 5       WORD        wMid;                  /* manufacturer ID */
 6       WORD        wPid;                  /* product ID */
 7       MMVERSION   vDriverVersion;        /* version of the driver */
 8       CHAR        szPname[MAXPNAMELEN]; /* product name (NULL terminated string) */
 9   #if (WINVER >= 0x0400)
10       DWORD   dwSupport;                 /* functionality supported by driver */
11   #endif
12   } MIDIINCAPSA, *PMIDIINCAPSA, *NPMIDIINCAPSA, *LPMIDIINCAPSA;
13   typedef struct tagMIDIINCAPSW {
14       WORD        wMid;                  /* manufacturer ID */
15       WORD        wPid;                  /* product ID */
16       MMVERSION   vDriverVersion;        /* version of the driver */
17       WCHAR       szPname[MAXPNAMELEN]; /* product name (NULL terminated string) */
18   #if (WINVER >= 0x0400)
19       DWORD   dwSupport;                 /* functionality supported by driver */
20   #endif
21   } MIDIINCAPSW, *PMIDIINCAPSW, *NPMIDIINCAPSW, *LPMIDIINCAPSW;
22   #ifdef UNICODE
23   typedef MIDIINCAPSW MIDIINCAPS;
24   typedef PMIDIINCAPSW PMIDIINCAPS;
25   typedef NPMIDIINCAPSW NPMIDIINCAPS;
26   typedef LPMIDIINCAPSW LPMIDIINCAPS;
27   #else
28   typedef MIDIINCAPSA MIDIINCAPS;
29   typedef PMIDIINCAPSA PMIDIINCAPS;
30   typedef NPMIDIINCAPSA NPMIDIINCAPS;
31   typedef LPMIDIINCAPSA LPMIDIINCAPS;
32   #endif // UNICODE
33   typedef struct tagMIDIINCAPS2A {
34       WORD        wMid;                  /* manufacturer ID */
35       WORD        wPid;                  /* product ID */
36       MMVERSION   vDriverVersion;        /* version of the driver */
37       CHAR        szPname[MAXPNAMELEN]; /* product name (NULL terminated string) */
38   #if (WINVER >= 0x0400)
39       DWORD       dwSupport;             /* functionality supported by driver */
```

```
40    #endif
41        GUID        ManufacturerGuid;     /* for extensible MID mapping */
42        GUID        ProductGuid;          /* for extensible PID mapping */
43        GUID        NameGuid;             /* for name lookup in registry */
44    } MIDIINCAPS2A, *PMIDIINCAPS2A, *NPMIDIINCAPS2A, *LPMIDIINCAPS2A;
45    typedef struct tagMIDIINCAPS2W {
46        WORD        wMid;                 /* manufacturer ID */
47        WORD        wPid;                 /* product ID */
48        MMVERSION   vDriverVersion;       /* version of the driver */
49        WCHAR       szPname[MAXPNAMELEN]; /* product name (NULL terminated string) */
50    #if (WINVER >= 0x0400)
51        DWORD       dwSupport;            /* functionality supported by driver */
52    #endif
53        GUID        ManufacturerGuid;     /* for extensible MID mapping */
54        GUID        ProductGuid;          /* for extensible PID mapping */
55        GUID        NameGuid;             /* for name lookup in registry */
56    } MIDIINCAPS2W, *PMIDIINCAPS2W, *NPMIDIINCAPS2W, *LPMIDIINCAPS2W;
57    #ifdef UNICODE
58    typedef MIDIINCAPS2W MIDIINCAPS2;
59    typedef PMIDIINCAPS2W PMIDIINCAPS2;
60    typedef NPMIDIINCAPS2W NPMIDIINCAPS2;
61    typedef LPMIDIINCAPS2W LPMIDIINCAPS2;
62    #else
63    typedef MIDIINCAPS2A MIDIINCAPS2;
64    typedef PMIDIINCAPS2A PMIDIINCAPS2;
65    typedef NPMIDIINCAPS2A NPMIDIINCAPS2;
66    typedef LPMIDIINCAPS2A LPMIDIINCAPS2;
67    #endif // UNICODE
68
69    #else
70    typedef struct midiincaps_tag {
71        WORD    wMid;                  /* manufacturer ID */
72        WORD    wPid;                  /* product ID */
73        VERSION vDriverVersion;        /* version of the driver */
74        char    szPname[MAXPNAMELEN];  /* product name (NULL terminated string) */
75    #if (WINVER >= 0x0400)
76        DWORD   dwSupport;             /* functionality supported by driver */
77    #endif
78    } MIDIINCAPS, *PMIDIINCAPS, NEAR *NPMIDIINCAPS, FAR *LPMIDIINCAPS;
79    #endif
```

#ifdef, #if, #else, #endif といった各種プリプロセッサディレクティブが多いために理解しにくくなっているので，整理しながら順に見ていく。

まず 2 行目に `#ifdef _WIN32` がある。これにより，Win32 アプリケーションであれば 4-68 行目が，そうでなければ 70-78 行目がコンパイルされる。ここでは Win32 アプリケーション以外は考えないので，4-68 行目を見ていこう。

4-68 行目を見てみると，4-32 行目と 33-67 行目がほぼ同じ構造を有していることが分かる。詳しく読むと，4-32 行目が MIDIINCAPS 構造体とそのポインタの宣言になっており，33-67 行目が MIDIINCAPS2 構造体とそのポインタの宣言になっていることが分かる。MIDIINCAPS2 構造体は，詳しく調べてみるとどうやら MIDI 入力デバイスに関するハードウェア固有の情報を得るために使われるようである[*1]。ここでは MIDIINCAPS 構造体しか使用しないので，33-67 行目は無視し，4-32 行目を次に見ていくことにする。

4-32 行目を上から見ていこう。まず 4-12 行目で MIDIINCAPSA 構造体とそのポインタが，13-21 行目で MIDIINCAPSW 構造体とそのポインタが宣言されている。これらの構造体はほぼ同じメンバを持っており，唯一の異なる点は szPname メンバの型が MIDIINCAPSA 構造体では CHAR 型に，MIDIINCAPSW 構造体では WCHAR 型になっていることである。

CHAR 型は実際は単なる char 型であり[*2]，1 文字 1 バイトの単純な文字列である。それに対して WCHAR 型は実際は wchar_t 型であり[*3]，1 文字 2 バイトの Unicode 文字列である。現在の文字列処理では両者が混交して使われているため，このような形になっている。

Windows SDK では，接尾辞 -A （ANSI の意味）は char 型文字列用の関数に付けられている。Shift_JIS のように，1 文字を 1 バイトまたは 2 バイト以上の可変長で表す場合を MBCS (MultiByte Character Set) 文字列と呼ぶが，1 文字 1 バイトの ASCII 文字列（SBCS (Single Byte Character Set) 文字列と呼ぶ）が char 型配列を使うのはもちろん，MBCS 文字列でも char 型配列を用いて文字ごとに可変長要素で扱うので，-A 接頭辞の付いた関数はそのような SBCS や MBCS を扱うために用意されている。

それに対して，接尾辞 -W （Wide Character の意味）は wchar_t 型文字列用の関数に付けられている。Unicode のように，1 文字を 2 バイト以上の固定長で表す場合をワイド文字 (Wide Character) と呼び，Unicode では wchar_t 型配列を用いて 1 文字 1 要素で扱うので，-W 接頭辞の付いた関数は 16-bit Unicode を扱うために用意されている[*4]。

もし関数名に -A と -W の付いた 2 つのバージョンが存在している場合には，このような違いがあるということを示している。

次に 22-32 行目を見てみよう。#ifdef により，UNICODE マクロが定義されている場合は 23-26 行目が，そうでない場合は 28-31 行目がコンパイルされる。そして，UNICODE マクロが定義されている場合，MIDIINCAPS 構造体は Unicode 文字列を保持する

[*1] Windows XP 以降でのみサポートされているので，古い SDK を使用している場合は定義されていないかもしれない。

[*2] winnt.h 内で `typedef char CHAR;` と定義されている。

[*3] winnt.h 内で `typedef wchar_t WCHAR;` と定義されている。

[*4] なお，Unicode のエンコーディングである UTF-8 はワイド文字ではなく MBCS である。

MIDIINCAPSW 構造体の別名となり，UNICODE マクロが定義されていない場合は SBCS/MBCS 文字列を保持する MIDIINCAPSA 構造体の別名となっている。従って，UNICODE マクロの定義の有無により，コンパイル時に SBCS/MBCS 版と Unicode 版を切り替えることができるようになっている。

Windows SDK プログラミングにおけるこの機構をジェネリックテキストマッピングと呼んでいる。ジェネリックテキストマッピングを用いると，SBCS/MBCS 版と Unicode 版の両方に対応した汎用的なプログラムを作成することができる[*5]。ただし，そのためにはジェネリックテキストに対応したプログラミングを行う必要がある。例えば,

- ヘッダファイル tchar.h をインクルードする
- UNICODE マクロと _UNICODE マクロを用いる（Unicode 版でコンパイルしたい場合は _UNICODE マクロを定義するだけでよい）
- 汎用的な文字列型（TCHAR や LPTSTR など），文字列定数に対する _TEXT マクロ，汎用的な文字列用関数（printf() の替わりに _tprintf() を用いるなど）を使用する

といったことを行う必要がある。

また，MBCS 用に _MBCS マクロが用意されている。デフォルトでは _UNICODE マクロも _MBCS マクロも定義されておらず，SBCS 用にコンパイルされる（なお，SBCS 用と MBCS 用は，Unicode 用に比べてそれほど違いはない）。

本テキストでは，デフォルトの SBCS 環境でプログラミングを行っている。この場合，文字列は通常の char 型配列や printf() などの関数で扱えばよい（ジェネリックテキスト用の TCHAR 型や _tprintf()，あるいはワイド文字用の wchar_t 型や wprintf() などを使用する必要はない）。また，MIDIINCAPS 構造体は MIDIINCAPSA 構造体の別名となることが分かる。

本テキストの範囲では Unicode やジェネリックテキストマッピングの詳細，およびそれらのプログラミング方法[*6] を知る必要は全くないが，ヘッダファイルの解析などで必要になるので基本的な知識だけ身に着けておくとよいだろう。

4.2　入力デバイスのオープンとクローズ

MIDI メッセージの送信時と同様に，MIDI メッセージを受信する場合にもまず初めに入力デバイスを開き，受信が終わればデバイスを閉じる。入力デバイスのオープンには

[*5] 例えば Windows 9x 系と Windows NT/2000/XP 系との両者に対応できる。

[*6] 一般的に国際化プログラミングや I18N (internationalization) / l10n (localization) などと呼ばれる。

midiInOpen() 関数を，クローズには midiInClose() 関数を使用する。これらの関数の使い方は，出力デバイス用の midiOutOpen() や midiOutClose() と同様である。

midiInOpen() および midiInClose() のプロトタイプ宣言はそれぞれ次の通りである。

```
MMRESULT midiInOpen(
  LPHMIDIIN lphMidiIn,
  UINT      uDeviceID,
  DWORD_PTR dwCallback,
  DWORD_PTR dwCallbackInstance,
  DWORD     dwFlags
);

MMRESULT midiInClose(
  HMIDIIN   hMidiIn
);
```

これらの関数を用いて MIDI 入力デバイスのオープンとクローズを行うプログラムをリスト 4.2 に示す。

リスト 4.2 MIDI 入力デバイスのオープン/クローズ (midiin-openclose.c)

```
#include <stdio.h>
#include <windows.h>
#include <mmsystem.h>

int main(int argc, char **argv) {
  HMIDIIN hMidiIn;             /* MIDI 入力デバイスのハンドル */
  MMRESULT res;                /* 戻り値を保持 */
  UINT devid;                  /* デバイス ID */
  char errmsg[MAXERRORLENGTH]; /* エラーメッセージ格納用 */

  if (argc > 1) {
    /* 引数が指定されていれば unsigned int として読み込んで devid に格納 */
    sscanf(argv[1], "%u", &devid);
  } else {
    /* 引数が指定されていなければ 0 とする */
    devid = 0u;
  }

  /* MIDI 入力デバイスを開く */
  res = midiInOpen(&hMidiIn, devid, (DWORD_PTR)NULL, 0, CALLBACK_NULL);
  if (res != MMSYSERR_NOERROR) {
    /* エラーがあった場合は標準出力にエラーメッセージを表示して終了 */
    printf("Cannot open MIDI input device (ID=%u): ", devid);
    midiInGetErrorText(res, errmsg, sizeof(errmsg));
    printf("%s\n", errmsg);
    return 1;
```

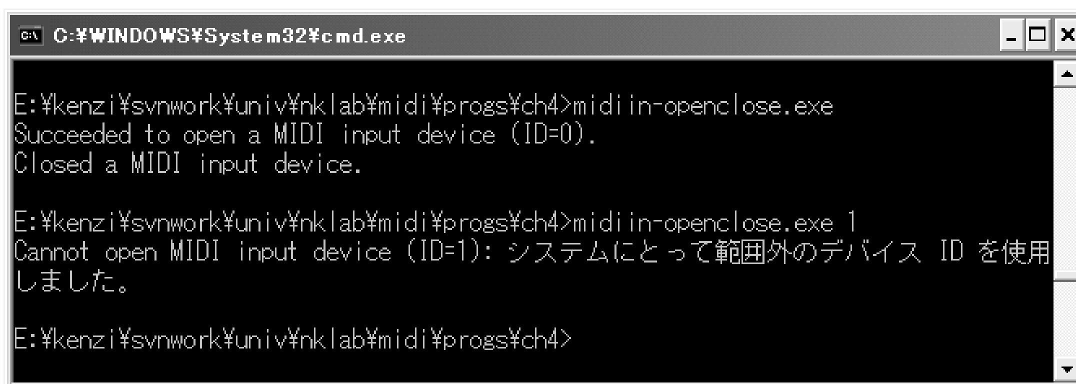
```
    }
    printf("Succeeded to open a MIDI input device (ID=%u).\n", devid);

    /* MIDI 入力デバイスを閉じる */
    midiInClose(hMidiIn);
    printf("Closed a MIDI input device.\n");

    return 0;
}
```

このプログラムの動作は 3.2.3 節のリスト 3.6 と同様，単にデバイスのオープンとクローズを行っているだけであるが，コマンドライン引数の処理とエラー発生時の処理が加えられている。

実際に実行させてみるとプログラムを理解しやすいだろう。筆者の環境では次のような実行結果になった（図 4.1）。


```
C:¥WINDOWS¥System32¥cmd.exe

E:¥kenzi¥svnwork¥univ¥nklab¥midi¥progs¥ch4>midiin-openclose.exe
Succeeded to open a MIDI input device (ID=0).
Closed a MIDI input device.

E:¥kenzi¥svnwork¥univ¥nklab¥midi¥progs¥ch4>midiin-openclose.exe 1
Cannot open MIDI input device (ID=1): システムにとって範囲外のデバイス ID を使用しました。

E:¥kenzi¥svnwork¥univ¥nklab¥midi¥progs¥ch4>
```

図 4.1　midiin-openclose.exe の実行結果

まず，コマンドライン引数は main() 関数の引数として与えられる。第 1 引数の int argc が引数の数，第 2 引数の char **argv が各引数の内容（文字列）である。なお，引数の最初 (argv[0]) にはそのプログラム自身の名前が入っているので，もし引数が指定されていなくても argc は 1 となる。ユーザが指定した引数があれば，argv[1] 以降に順次格納されるので，ここではそれを sscanf() 関数で読み込み，devid に格納している（引数が与えられていない場合はデフォルトとして 0 を格納）。

MIDI 入力デバイスのオープン時にエラーが発生した場合，midiInOpen() 関数の戻り値が MMSYSERR_NOERROR 以外の値になる。その場合，上記プログラムでは midiInGetErrorText() を呼び出してエラーメッセージを取得し，そのエラーメッセージを標準出力に表示して終了するようにしている。

midiInGetErrorText() は midiIn 接頭辞を持つ関数が返すエラーコードに対して，Windows システムが用意しているエラーメッセージを返す関数である。そのプロトタイ

プ関数は以下の通り。

```
MMRESULT midiInGetErrorText(
  MMRESULT wError,
  LPSTR    lpText,
  UINT     cchText
);
```

第 1 引数 wError のエラーコードに対応するエラーメッセージが第 2 引数 lpText に格納される[*7]。それらのエラーメッセージは MAXERRORLENGTH 文字未満であることが保証されているので，上記プログラムでは errmsg を長さ MAXERRORLENGTH で宣言している。

本テキストのように学習を目的とする演習ではプログラム内のエラー処理を省略したり簡潔に済ますことが多いが，実際のプログラム開発ではエラー処理は最も重要なことがらであり，欠かさずに行うようにしたい。

なお，midiInGetErrorText() は入力関連のエラーメッセージを取得する関数であるが，出力関連の同様の機能を持つ関数に midiOutGetErrorText() がある。

プログラミングメモ 9 — main 関数の戻り値

通常の関数は引数と戻り値を持つが，main() 関数も同様に引数と戻り値を持つことができる。main() 関数の場合，引数はプログラムの実行時に渡すコマンドライン引数であり，戻り値はプログラムの戻り値，言い換えればコマンドの終了コードである。

例えば 4.2 節のリスト 4.2 では，成功時に 0 を，デバイスのオープンに失敗した場合（エラー時）には 1 を return で返している[*8*9]。このように，終了コードは正常終了時には 0 を，異常終了時には 0 以外の値を返すのが慣例である。

このような仕組みを利用することによって，スクリプト内やバッチプログラム内でコマンドが成功したかどうかにより処理を分岐することができる。GUI のみを持つアプリケーションや MIDI 入出力のように常に人間が介在して結果を評価する場合はあまり使われないが，一連の処理を自動で行う場合には重宝される仕組みである。

[*7] LPSTR 型は char 型へのポインタと考えてよい。

[*8] C 言語では，return は関数ではなく制御文である。従って関数呼び出しの ( ) は要らない。

[*9] main() 関数外では，return ではなく exit() 関数を使うことでプログラムの終了および終了コードを返すことができる。この場合，exit は関数なので呼び出しには ( ) が要る。

4.3　MIDI メッセージの受信

入力デバイスを開くことができたので，次に実際に MIDI メッセージを受信するプログラムを作る。

4.3.1　受信の開始と停止

MIDI メッセージの受信はデバイスを開いただけでは始まらず，受信を開始するには midiInStart() 関数を呼び出す必要がある。その対となる関数に midiInStop() がある。これらの関数を見てみよう。

midiInStart()

midiInStart() は与えられた MIDI 入力デバイスで入力を開始する API であり，そのプロトタイプ宣言は次の通りである。

```
MMRESULT midiInStart(
  HMIDIIN hMidiIn
);
```

hMidiIn には midiInOpen() で開いておいた入力デバイスのハンドルを指定する。成功時には MMSYSERR_NOERROR が，失敗すればエラーが返る。以下の関数も同様である。

midiInStop()

midiInStop() は与えられた MIDI 入力デバイスで入力を停止する API であり，そのプロトタイプ宣言は次の通りである。

```
MMRESULT midiInStop(
  HMIDIIN hMidiIn
);
```

midiInReset()

上述の 2 つの API に関連して，midiInReset() という関数が用意されている。これはシステムエクスクルーシブメッセージを扱う場合に意味を持つため今のところは気に留めなくてもよいが，そのプロトタイプ宣言だけここで示しておく。

```
MMRESULT midiInReset(
```

```
  HMIDIIN hMidiIn
);
```

4.3.2 コールバック関数

以上の手順を踏むと MIDI メッセージの受信が開始される。入力デバイスとのデータ送受信といった低レベルな動作は Windows の MIDI ライブラリが面倒を見てくれるため，我々のプログラム —— ライブラリの利用者（クライアント）という意味でクライアントプログラムと呼ぶ —— では受信されたメッセージに対する処理だけを書けばよい。

ライブラリが受信したメッセージはコールバック[*10] という仕組みを用いてクライアントプログラムに通知される。ここで利用できるコールバックはウィンドウ呼び出し，スレッド呼び出し，関数呼び出しの 3 種類あるが，ここではコールバック関数を用いた関数呼び出しを利用する。これは，メッセージを受信した際に呼び出して欲しい関数をあらかじめライブラリに知らせておく（登録しておく）方法であり，そのようにして呼び出される関数をコールバック関数と呼ぶ。コールバック関数はメッセージが一つ受信される度に呼び出され，受信メッセージの内容がコールバック関数の引数として与えられるので，その引数を見てメッセージ毎の処理を行えばよい。

MIDI の入力メッセージに対するコールバック関数は次のように定義する。

```
void CALLBACK MidiInProc(
  HMIDIIN hMidiIn,
  UINT wMsg,
  DWORD dwInstance,
  DWORD dwParam1,
  DWORD dwParam2
);
```

5 つのデータが渡されるが，ここで重要なのは wMsg, dwParam1, dwParam2 である。wMsg にはメッセージの種類が格納され，dwParam1 と dwParam2 にメッセージの内容が格納される。これら以外の引数は無視しても構わないだろう。なお，ここでは関数名が MidiInProc となっているが，実際にコールバック関数を定義する場合には好きな名前を付けてよい（MidiInProc でも構わない）。

メッセージ受信時（イベント発生時）に，単に wMsg, dwParam1, dwParam2 の値を出力するだけのコールバック関数は次のようになる。なお，printf() の %X は int 型引数を 16 進数で出力する変換指定子である。その他，詳しい書式は printf() のドキュメン

[*10] コールバックとは元々，電話で呼び出した相手側から折り返し電話を求めることを言う。

テーションを参照して欲しい。

```
void CALLBACK my_callback(HMIDIIN hMidiIn, UINT wMsg, DWORD dwInstance,
                          DWORD dwParam1, DWORD dwParam2) {
  printf("MidiInProc: wMsg=%08X, p1=%08X, p2=%08X\n",
         wMsg, dwParam1, dwParam2);
}
```

このように定義したコールバック関数は MIDI 入力デバイスのオープン時，すなわち midiInOpen() 関数の引数としてコールバック関数を与えることによって，メッセージ受信時に MIDI ライブラリから呼び出されることが可能になる。コールバック関数を割り当てるには，midiInOpen() 関数の第 3 引数 dwCallback にコールバック関数のアドレスを，第 5 引数 dwFlags に定数 CALLBACK_FUNCTION を指定する。my_callback 関数を登録する場合は，例えば次のようにして midiInOpen() を呼び出せばよい。

```
midiInOpen(&hMidiIn, devid,
           (DWORD_PTR)my_callback, 0, CALLBACK_FUNCTION);
```

4.3.3 メッセージの受信

以上で準備は整ったので，コールバック関数を使用して MIDI メッセージを受信し表示するプログラムをリスト 4.3 に示す。

リスト 4.3 MIDI メッセージの受信 (recvmidimsg.c)

```
#include <stdio.h>
#include <windows.h>
#include <mmsystem.h>

/* コールバック関数の定義 */
void CALLBACK my_callback(HMIDIIN hMidiIn, UINT wMsg, DWORD dwInstance,
                          DWORD dwParam1, DWORD dwParam2) {
  printf("MidiInProc: wMsg=%08X, p1=%08X, p2=%08X\n",
         wMsg, dwParam1, dwParam2);
}

int main(int argc, char **argv) {
  HMIDIIN hMidiIn;              /* MIDI 入力デバイスのハンドル */
  MMRESULT res;                 /* 戻り値を保持 */
  UINT devid;                   /* デバイス ID */
  char errmsg[MAXERRORLENGTH];  /* エラーメッセージ格納用 */

  if (argc > 1) {
    sscanf(argv[1], "%u", &devid);
  } else {
```

```
    devid = 0u;
  }

  /* MIDI 入力デバイスを開く */
  res = midiInOpen(&hMidiIn, devid,
                   (DWORD_PTR)my_callback, 0, CALLBACK_FUNCTION);
  if (res != MMSYSERR_NOERROR) {
    printf("Cannot open MIDI input device (ID=%u): ", devid);
    midiInGetErrorText(res, errmsg, sizeof(errmsg));
    printf("%s\n", errmsg);
    return 1;
  }
  printf("Succeeded to open a MIDI input device (ID=%u).\n", devid);

  /* 入力を開始 */
  midiInStart(hMidiIn);

  /* 無限ループ */
  while (1) {
    Sleep(10);
  }

  /* 入力を停止 */
  midiInStop(hMidiIn);
  midiInReset(hMidiIn);

  /* MIDI 入力デバイスを閉じる */
  midiInClose(hMidiIn);
  printf("Closed a MIDI input device.\n");

  return 0;
}
```

外部 MIDI 入力機器を接続した状態で実行し，鍵盤を叩くなどの入力を行って結果を確認して欲しい。なお，このプログラムはループから抜けることができないため，実行したプログラムは Ctrl+C などで強制終了する必要がある。

課題 6

リスト 4.3 で無限ループ以降のプログラムが実行されているかどうかを確認せよ。

プログラミングメモ 10 — イベントドリブン型プログラミング

通常は何も動作せず，イベントが発生するとそれに対応する動作を行うプログラムをイベントドリブン（イベント駆動）型のプログラムという。イベントドリブン型のプログラ

ムは基本的に次のような構造になっている。なお，無限ループを抜けるのはプログラムを終了させるようなイベントが発生した場合に限られる。

```
while ( 無限ループ ) {
  if ( 未処理のイベントがある ) {
    イベントに対応した処理を行う;
  }
}
```

典型的な例として Windows 上で動作する GUI アプリケーションが挙げられる。このような GUI アプリケーションでは，プログラムは（無限ループを作って）ユーザが入力を行うのを待っている。そしてユーザが入力を行う（例えばボタンをクリックする）と，OS がそのイベントをアプリケーションに通知する（イベントキューに登録する)。未処理のイベントを検知したアプリケーションはそのイベントの種類や内容を調べ，そのイベントに対して適切な処理を行う。その後，またイベント待ち状態に入る。この場合，プログラムの動作を最初から最後まで順にプログラミングしていくというよりも，一つ一つのイベントに対する動作をプログラミングすることになる。

MIDI 入力を扱うプログラムも同様にイベントドリブン型のプログラミングが必要になる。この場合，上記の構造における無限ループは MIDI ライブラリの中に組み込まれて（隠蔽されて）おり，イベント発生時に呼び出されるコールバック関数をプログラミングするだけでよくなっている（ただし，プログラムを終了させないための無限ループが別に必要ではある)。

4.4　MIDI メッセージの処理

4.4.1　コールバック関数に渡されるメッセージ

コールバック関数は MIDI メッセージの受信時以外にも，例えば MIDI 入力デバイスのオープン時などにも呼び出される。そして呼び出された理由はコールバック関数の第 2 引数 (UINT wMsg) を見れば分かるようになっており，wMsg には以下の 7 つの定数のいずれかが設定されている[*11]。

MIM_OPEN　MIDI 入力デバイスが開かれたときにこの値が設定されてコールバック関数が呼び出される。

dwParam1: 予約済み; 使用しない。

[*11] 関数呼び出し以外のコールバック機構（ウインドウやスレッドの呼び出し）を利用する場合は違う値を使用することに注意。

dwParam2: 予約済み; 使用しない。

MIM_CLOSE　MIDI 入力デバイスが閉じられたときにこの値が設定されてコールバック関数が呼び出される。

dwParam1: 予約済み; 使用しない。

dwParam2: 予約済み; 使用しない。

MIM_DATA　MIDI メッセージを受信したときにこの値が設定されてコールバック関数が呼び出される。

dwParam1: dwMidiMessage

dwParam2: dwTimestamp

MIM_LONGDATA　システムエクスクルーシブ (SysEx) 用バッファが埋まり，アプリケーションに返されるときにこの値が設定されてコールバック関数が呼び出される。

dwParam1: (DWORD) lpMidiHdr

dwParam2: dwTimestamp

MIM_ERROR　無効な MIDI メッセージを受信したときにこの値が設定されてコールバック関数が呼び出される。

dwParam1: dwMidiMessage

dwParam2: dwTimestamp

MIM_LONGERROR　無効または不完全なシステムエクスクルーシブメッセージを受信したときにこの値が設定されてコールバック関数が呼び出される。

dwParam1: (DWORD) lpMidiHdr

dwParam2: dwTimestamp

MIM_MOREDATA　MIDI メッセージを受信したが，アプリケーションが MIM_DATA を処理しきれていない場合にこの値が設定されてコールバック関数が呼び出される。なお，これは midiInOpen 関数の呼び出し時に MIDI_IO_STATUS が指定された場合にのみ呼び出される。

dwParam1: dwMidiMessage

dwParam2: dwTimestamp

なお，dwMidiMessage は 4 バイトの整数（バイト列）であり，MIDI メッセージがリトルエンディアンで格納される（MSB は使われない）*12。dwTimestamp はメッセージを受信した時間であり，midiInStart() 関数の呼び出し時に 0 にリセットされ，その時点からの経過ミリ秒で表される。lpMidiHdr は MIDIHDR 構造体へのポインタである。MIDIHDR 構造体については 4.4.2 節で説明する。

*12 ランニングステータス（5.2.3 節参照）使用時でも 1 バイト目（リトルエンディアンで格納されるため最下位バイトである）にはステータスバイトが正しく格納されている。

コールバック関数内でそれぞれに対応した処理を場合分けして記述するには，if 文よりも switch 文を使うとよいだろう。例えば次のように書けばよい。

```
void CALLBACK MidiInProc(HMIDIIN hMidiIn, UINT wMsg, DWORD dwInstance,
                         DWORD dwParam1, DWORD dwParam2) {
  switch (wMsg) {
    case MIM_OPEN:
      printf("MIDI device is opened.\n");
      break;
    case MIM_CLOSE:
      printf("MIDI device is closed.\n");
      break;
    case MIM_DATA:
      // 処理...
      break;
    case MIM_LONGDATA:
      // 処理...
      break;
    case MIM_ERROR:
    case MIM_LONGERROR:
    case MIM_MOREDATA:
    default:
      printf("MidiInProc: wMsg=%08X, p1=%08X, p2=%08X\n",
             wMsg, dwParam1, dwParam2);
      break;
  }
}
```

課題 7

ある特定のノート番号のキーを押すと終了するプログラムを作成せよ。ただし，終了時には midiInStop() や midiInClose() などを呼び出して正常に終了させること。

4.4.2 システムエクスクルーシブメッセージ

MIDI システム全体に関する情報を扱うシステムメッセージには，以下の 3 種類のメッセージがあることを 3.3.1 節で述べた。

- システムエクスクルーシブメッセージ（SysEx メッセージと略記する[*13]）

[*13] System Exclusive の略。システムに依存する（システムごとに異なる）メッセージを意味する。

- システムコモンメッセージ
- システムリアルタイムメッセージ

システムメッセージのステータスバイトは上位 4 ビットが 0xF であり，またチャンネルを指定する必要がないため下位 4 ビットでメッセージの種類を指定するようになっている。

SysEx メッセージのステータスバイトは 0xF0 であり，その後にデータバイトが続き（バイト数に制限無し），最後にエンドオブエクスクルーシブ (EOX) メッセージ (0xF7) で閉じられる。任意長のデータを扱わなければならないため，SysEx だけは他の MIDI メッセージと異なる特別な処理が必要となる。

システムコモンメッセージのステータスバイトは 0xF1〜0xF7 で，表 4.1 のようなメッセージがあり，またシステムリアルタイムメッセージの場合は 0xF6〜 0xFF で表 4.2 の通りである。

表 4.1　システムコモンメッセージ

メッセージの種類	ステータスバイト	第 1 データバイト	第 2 データバイト
MTC クォーターフレーム	0xF1	メッセージタイプ / 値	——
ソングポジションポインタ	0xF2	ソングポジション (LSB)	ソングポジション (MSB)
ソングセレクト	0xF3	ソングナンバー	——
チューンリクエスト	0xF6	——	——
エンドオブエクスクルーシブ	0xF7	——	——

表 4.2　システムリアルタイムメッセージ

メッセージの種類	ステータスバイト	第 1 データバイト	第 2 データバイト
タイミングクロック	0xF8	——	——
スタート	0xFA	——	——
コンティニュー	0xFB	——	——
ストップ	0xFC	——	——
アクティブセンシング	0xFE	——	——
システムリセット	0xFF	——	——

システムコモンメッセージやシステムリアルタイムメッセージはチャンネルメッセージと同様に送受信ができる，すなわち midiOutShortMsg() で送信し，コールバック関数では MIM_DATA で受信することが可能である。しかし，SysEx メッセージはバイト長が不定であるために異なった仕組みを用いなければならない。例えば送信の場合は midiOutLongMsg() 関数を用いるし，受信の場合のコールバック関数では MIM_LONGDATA を利用する。

SysEx メッセージを扱うには，送信時受信時ともに MIDIHDR*14 構造体を使用する。まず，MIDIHDR 構造体の定義を見てみよう。なお，各メンバの簡単な説明はコメントとして記述されているが，詳しい内容はドキュメンテーションを参照して欲しい。

リスト 4.4 MIDIHDR 構造体の定義

```
typedef struct {
  LPSTR       lpData;                 /* pointer to locked data block */
  DWORD       dwBufferLength;         /* length of data in data block */
  DWORD       dwBytesRecorded;        /* used for input only */
  DWORD_PTR   dwUser;                 /* for client's use */
  DWORD       dwFlags;                /* assorted flags (see defines) */
  struct midihdr_tag far *lpNext;     /* reserved for driver */
  DWORD_PTR   reserved;               /* reserved for driver */
  DWORD       dwOffset;               /* Callback offset into buffer */
  DWORD_PTR   dwReserved[8];          /* Reserved for MMSYSTEM */
} MIDIHDR;
```

文字列へのポインタを意味する LPSTR 型の lpData に実際のデータ（MIDI メッセージのバイト列）が格納される。SysEx メッセージの受信時には，この MIDIHDR 構造体を扱う関数が 3 つ用意されている。すなわち，midiInAddBuffer()，midiInPrepareHeader()，midiInUnprepareHeader() である。

順に見ていこう。midiInAddBuffer() は SysEx メッセージ用の入力バッファを MIDI 入力デバイスに対して登録する関数であり，プロトタイプ宣言は次の通りである。

```
MMRESULT midiInAddBuffer(
  HMIDIIN hMidiIn,
  LPMIDIHDR lpMidiInHdr,
  UINT cbMidiInHdr
);
```

midiInPrepareHeader() は入力バッファを準備する関数であり，プロトタイプ宣言は次の通りである。

```
MMRESULT midiInPrepareHeader(
  HMIDIIN hMidiIn,
  LPMIDIHDR lpMidiInHdr,
  UINT cbMidiInHdr
);
```

*14 MIDI HeaDeR の略であろう。

midiInUnprepareHeader() は入力バッファに行った準備をクリーンアップする関数であり，プロトタイプ宣言は次の通りである。

```
MMRESULT midiInUnprepareHeader(
  HMIDIIN hMidiIn,
  LPMIDIHDR lpMidiInHdr,
  UINT cbMidiInHdr
);
```

バッファの「準備」といってもよく分からないだろうが，バッファを使用する前には準備 (prepare) し，使用した後には準備を解除 (unprepare) する必要がある。

これらは，MIDI メッセージの受信の場合，例えば次のような手順で関数呼び出しなどを行うように設計されている（末尾に * を付けているものが SysEx 関連の要素である）。

1. MIDIHDR 構造体の変数用のメモリ領域を確保 *
2. midiInOpen()
3. midiInPrepareHeader() *
4. midiInAddBuffer() *
5. midiInStart()
6. 処理
7. midiInStop()
8. midiInReset()
9. midiInUnprepareHeader() *
10. midiInClose()
11. MIDIHDR 構造体の変数用のメモリ領域を解放 *

以上の手順を実装し，SysEx メッセージを受信して表示するプログラムをリスト 4.5 に示す。

リスト 4.5 SysEx メッセージの受信 (recvsysexmsg.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <mmsystem.h>

/* SysEx 用バッファのサイズ */
#define BUFSIZE 256

int end_of_program;

void CALLBACK MidiInProc(HMIDIIN hMidiIn, UINT wMsg, DWORD dwInstance,
```

```
                        DWORD dwParam1, DWORD dwParam2) {
MIDIHDR *hdr;
int i;

switch (wMsg) {
  case MIM_OPEN:
    printf("MIDI device is opened.\n");
    break;
  case MIM_CLOSE:
    printf("MIDI device is closed.\n");
    break;
  case MIM_DATA:
    /* アクティブセンシングは表示しない */
    if ( (dwParam1 & 0x000000ff) == 0xfe ) {
      break;
    }

    /*
    if ( 課題 7 で作成した条件式 ) {
      end_of_program = 1;
      printf("end_of_program flag is on.\n");
    }
    */

    printf("MIM_DATA: wMsg=%08X, p1=%08X, p2=%08X\n",
           wMsg, dwParam1, dwParam2);
    break;
  case MIM_LONGDATA:
    printf("MIM_LONGDATA: ");
    hdr = (MIDIHDR*)dwParam1;  /* DWORD 型から MIDIHDR*型へのキャスト */
    printf("[%d bytes] ", hdr->dwBytesRecorded);  /* 格納されたバイト数を表示 */
    for (i=0; i<hdr->dwBytesRecorded; i++) {
      /* バイトごとに 16 進数で表示 */
      printf("%02X ", (unsigned char)((hdr->lpData)[i]));
    }
    printf("\n");

    /* 再度バッファを準備して登録 */
    midiInPrepareHeader(hMidiIn, hdr, sizeof(MIDIHDR));
    midiInAddBuffer(hMidiIn, hdr, sizeof(MIDIHDR));

    break;
  case MIM_ERROR:
  case MIM_LONGERROR:
  case MIM_MOREDATA:
  default:
    printf("MidiInProc: wMsg=%08X, p1=%08X, p2=%08X\n",
           wMsg, dwParam1, dwParam2);
    break;
}
```

```
}

int main(int argc, char **argv) {
  HMIDIIN hMidiIn;
  MMRESULT res;
  MIDIHDR header;  /* SysEx メッセージを格納する構造体 */
  UINT devid;
  char errmsg[MAXERRORLENGTH];

  if (argc > 1) {
    sscanf(argv[1], "%u", &devid);
  } else {
    devid = 0u;
  }

  /* MIDIHDR 構造体の初期化 */
  ZeroMemory(&header, sizeof(MIDIHDR));     /* メモリ領域を 0 で埋める */
  header.lpData = (char*)malloc(BUFSIZE);   /* メモリを BUFSIZE バイト分確保 */
  header.dwBufferLength = BUFSIZE;          /* バッファの容量を設定 */
  header.dwFlags = 0;

  res = midiInOpen(&hMidiIn, devid, (DWORD_PTR)MidiInProc, 0, CALLBACK_FUNCTION);
  if (res != MMSYSERR_NOERROR) {
    printf("Cannot open MIDI input device (ID=%u): ", devid);
    midiInGetErrorText(res, errmsg, sizeof(errmsg));
    printf("%s\n", errmsg);
    return 1;
  }
  printf("Succeeded to open a MIDI input device (ID=%u).\n", devid);

  /* SysEx 用バッファを準備し，それを登録する */
  midiInPrepareHeader(hMidiIn, &header, sizeof(MIDIHDR));
  midiInAddBuffer(hMidiIn, &header, sizeof(MIDIHDR));

  midiInStart(hMidiIn);

  end_of_program = 0;
  while ( !end_of_program ) {
    Sleep(10);
  }

  midiInStop(hMidiIn);
  midiInReset(hMidiIn);

  /* SysEx 用バッファの準備をクリーンアップ */
  midiInUnprepareHeader(hMidiIn, &header, sizeof(MIDIHDR));

  midiInClose(hMidiIn);
  printf("Closed a MIDI input device.\n");
```

```
  free(header.lpData);  /* バッファのメモリ領域を解放 */

  return 0;
}
```

課題 8

受信した MIDI メッセージをそのまま他の MIDI 機器に送信する，すなわち MIDI メッセージを中継するプログラムを作成せよ。そして中継先を PC の内蔵音源に設定し，PC 上で音が再生されることを確認せよ。

プログラミングメモ 11 — 変数の宣言とメモリ領域の確保

例えば次のように変数を宣言したとしよう。

```
unsigned int var;
```

unsigned int は 4 バイトの符号無し整数であるから，この宣言によって 4 バイトのメモリ領域が確保され，変数 var 用に割り当てられる。メモリのどこ（何番地）に 4 バイトの領域が確保されるかは，通常コンパイル時にコンパイラが適当に空いている位置を探すことで決定される。

もし変数 var が 100 番地に割り当てられたとすると，そのデータは 100〜103 番地までの 4 バイト長の領域を占めることになる。ここでは以降，データの最初のバイトのアドレス（例では 100 番地）を先頭アドレス，最後のバイトのアドレス（例では 103 番地）を末尾アドレスと便宜上呼ぶことにする。

さて，もし次のように変数を宣言すれば，a 用に 4 バイトのメモリ領域が，b 用に 1×3 バイトのメモリ領域が確保される（配列の場合，連続したメモリ領域が割り当てられる）。

```
unsigned int a;
char b[3];
```

ここで例えば変数 a のデータの先頭アドレスを 100 番地，配列 b の先頭要素のデータ（すなわち b[0]）の先頭アドレスを 104 番地とすると[*15]，図 4.2 のようにメモリ領域が確保される（これは実行時ではなくコンパイル時に決定される）。

また，コンパイラは同時に図 4.3 のような表も生成し，このような表を見てコンパイルを進めていく。通常の式中に変数名を記述した場合，データとして評価される（表の中央

[*15] 先頭アドレスといっても char 型は 1 バイトのデータ型であるので，データの先頭も末尾も同じアドレスである。

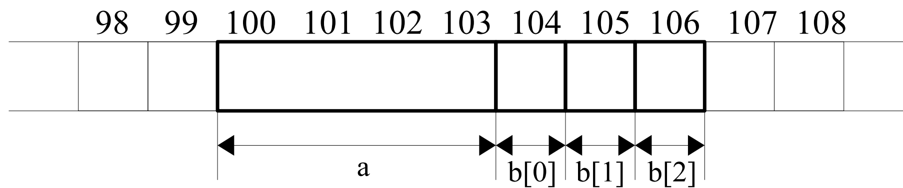


図 4.2　メモリ領域の確保

の列)。表中の *100 という表記は，100 番地のデータ（100 番地から unsigned int として取り出すデータ）を意味する。また，変数名に対して & 演算子を用いた場合（例えば &a）や，あるいは代入文の左辺値として変数名を単独で記述した場合（例えば a = ...）は，アドレスとして評価される（表の右の列)[*16]。なお，配列として宣言した変数名は，配列の型へのポインタとなることに注意する必要がある。

表 4.3　宣言された変数の表

変数名	データとしての評価値（型）	アドレスとしての評価値（型）
a	*100 (unsigned int)	100 (unsigned int *)
b	104 (char *)	104 (char *)

さて，次のように変数を宣言すると，ポインタ型は全て 4 バイト長であるから，変数 c 用に 4 バイトのメモリ領域が確保される（なお，この宣言は `LPUINT c` と等価である)。

```
unsigned int * c;
```

しかしこれだけでは当然ながらポインタ変数用のメモリ領域しか確保されないため，c が指す実データのメモリ領域も確保する必要がある。これは `unsigned int d;` と宣言してメモリ領域を確保し，`c = &d;` などとして利用してもよいが，必要となる実データの個数が実行時にならないと分からない場合など，動的にメモリを確保したい場合は malloc()[*17] 関数などを使うことができる[*18]。また，確保したメモリ領域は必ず解放する必要があり，そのために free() 関数などを使用する[*19]。

malloc() および free() 関数は次のようにして利用できる（これらの関数のプロトタイプ宣言は stdlib.h に書かれているので，`#include <stdlib.h>` を記述しておくこと)。な

[*16] 代入文における左辺値 (l-value) は，右辺値 (r-value) やその他の式中とは異なった扱いを受ける。これは，r-value が単一の値だけを求めるのに対して，l-value は（正確な表現ではないが）最終的には代入先のアドレスを求めることに由来する。
[*17] Memory ALLOCate の略であろう。
[*18] malloc() は標準 C ライブラリの関数である。Windows API では HeapAlloc() 関数などが用意されている。
[*19] free() は標準 C ライブラリの関数である。Windows API では HeapFree() 関数などが用意されている。

お，malloc() の戻り値の型は void* であり，また free() の第 1 引数の型も void* であるが，これらの代入時の型変換（キャスト）は自動的に行われるため，特に明示的にキャストする必要はない。

```
unsigned int * c;

c = malloc( sizeof(unsigned int) );

*c = 30;
printf("c: %p\n", c);
printf("*c: %u\n", *c);

free(c);
```

もしも配列のメモリ領域を動的に確保したい場合は，n 要素の配列であれば上記で `c = malloc( sizeof(unsigned int) * n )` などとして malloc() 関数を呼び出せばよい。そうすれば，c[0] などで配列の要素にアクセス可能である。

第 5 章

SMF

本章では楽譜情報を表すための SMF (Standard MIDI File) フォーマットを扱う。SMF は広く利用されているファイルフォーマットであり，拡張子には通常 .mid が用いられる。

5.1　SMF の基本構造

本 5.1 節では，SMF の大まかな構造を説明する。理解を助けるために，何でもよいので実際に音楽が聴ける SMF ファイルを一つ用意しておくとよいだろう。以下ではそのようなファイルを持っているものとして説明していく。

5.1.1　バイナリエディタの活用

SMF ファイルはバイナリファイルであり，テキストファイルではない[*1]。テキストファイルでないということは，通常のメモ帳などのテキストエディタで開いても文字化けしたように表示されるだけで，そこから実際のデータを読み取ることは難しい。

そのようなファイルはバイナリエディタで開くのがよい。様々なバイナリエディタが開発・公開されているが，インターネットから無料でダウンロード可能なものとしては Stirling – http://www.vector.co.jp/soft/win95/util/se079072.html – やバイナリエディタ Bz – http://www.zob.ne.jp/~c.mos/soft/bz.html – などが挙げられる。

SMF ファイルをバイナリエディタで開くと，0x 4D 54 68 64 (”MThd”) から始まるバイト列が表示されるだろう。以降は，説明を読むと同時に実際の SMF ファイルの内容をバイナリエディタで確認していくとよい。

[*1] テキストファイルは通常の文字列，具体的には ASCII や UTF8，Shift_JIS といった文字エンコーディングのデータだけで表現されているファイルのことを指す。それ以外のデータを含むものは一般的にバイナリファイルと呼ばれる。

5.1.2　ヘッダチャンク

SMF ファイルはまず先頭にヘッダチャンクが 1 つだけ存在し，その後に 1 つ以上のトラックチャンクが続く構造になっている[*2]。現在これら以外のチャンクは存在しない。ここではまず，ファイル全体に関するヘッダ情報を保持するヘッダチャンクについて説明する（図 5.1，表 5.1）。

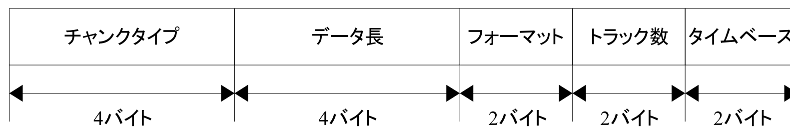


図 5.1　ヘッダチャンク

表 5.1　ヘッダチャンク

オフセット	データ長	データ名	値
0x00	4	チャンクタイプ	”MThd” (0x4D546864)
0x04	4	データ長	6
0x08	2	フォーマットタイプ	0〜2
0x10	2	トラック数	0〜65535
0x12	2	タイムベース	タイムベース値（後述）

ヘッダチャンクの最初の 8 バイトはチャンクタイプとチャンクのデータ長であり，これらは他のチャンクにも共通に存在する。

チャンクタイプ

チャンクタイプはチャンクの種類を表す文字列で，ヘッダチャンクの場合は必ず ”MThd” (0x 4D 54 68 64) になる。

データ長

データ長は（チャンクタイプとデータ長を除く）そのチャンクの大きさをバイト数で表し，ヘッダチャンクの場合は必ず 6 (0x 00 00 00 06) になる。これは，ヘッダチャンクではデータ長の後には必ず 2 バイトのデータが 3 つ続くからである。

データ長の後には，フォーマットタイプ，トラック数，タイムベースが続く。

[*2] チャンク (chunk) はひとかたまりのデータのことを指して使われる。ファイルの読み書きを含むストリーム操作関連でよく使われているように見受けられる。似た用語にブロック (block) があるが，こちらは固定サイズのときに用いられる場合が多い。

フォーマットタイプ

フォーマットタイプには SMF のフォーマットの種類を記述する。フォーマットには 3 種類あり，フォーマット 0 はトラックを 1 つだけ格納でき（シングルトラック），フォーマット 1 は複数トラックを格納でき（マルチトラック），フォーマット 2 は複数シーケンスを格納できる（マルチシーケンス）。

フォーマット 0 では 1 つのトラック（1〜16 の MIDI チャンネルを利用できる）で全ての MIDI 情報を保持するため，曲のタイトルやテンポなどの情報も演奏データと一緒に全て単一のトラックに記録される。それに対してフォーマット 1 では複数のトラックを使うことができるため，通常は曲のタイトルやテンポなどは最初のトラックに記録し，演奏データなどは 2 つ目以降のトラックに記録する。

フォーマット 2 では，同時に演奏される必要がないシーケンス —— 例えばドラムパターンなど —— を保持することができる。これはフォーマット 0 や 1 に比べてサポートされている場合が少ないため，インターネットでの配布など再生プラットフォームを限定したくない場合にはあまり使われない。

トラック数

トラック数にはファイル中にいくつトラックがあるかを記述する。ヘッダチャンクの後にはその数だけトラックチャンクが存在する。

フォーマット 0 では必ずトラック数は 1 になる。

タイムベース

タイムベースにはトラックイベントで使われるデルタタイムを現実時間に直す（変換する）ための値（分解能）を記述する。もし最上位ビットが 0 であれば残りの 15 ビットで 1 拍が何クロック（ティック）であるか (ticks per beat[*3]) を表し，最上位ビットが 1 であれば残りの 15 ビットで 1 秒間が何フレームであるか (frames per second) を表す。多くのシーケンサでは前者の ticks per beat がよく使われるため，ここでは後者の frames per second の詳しい説明は省略する[*4]。

最上位ビットが 0 の場合，時間分解能を ticks per beat で，すなわち四分音符 1 つが何クロックになるかを記述する。一般的には 48〜960，特に 384，480，960 などの 96 の倍数値が使われるようだ。例えばタイムベース値が 0x0180 (384) であるイベントのデルタタイム（5.2.1 節参照）が 192 であれば，そのデルタタイムは八分音符分の時間長を意味

[*3] beat は拍を意味する。

[*4] 最上位ビットが 1 の場合，すなわち frames per second による表現の場合，最上位ビットに続く 7 ビットで 1 秒間あたりの SMPTE フレーム数を，その後の残り 8 ビットで 1 フレームが何クロックかを表す。

することになる（これに加えて 1 拍=四分音符が何秒であるか，すなわちテンポが分かればデルタタイムを現実時間に直すことができる）。

5.1.3　トラックチャンク

ヘッダチャンクの後にはトラックチャンクが続く（図 5.2，表 5.2）。トラックチャンクの構造はフォーマットタイプに関わらず同じである。

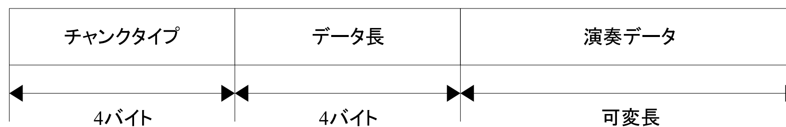


図 5.2　トラックチャンク

表 5.2　トラックチャンク

オフセット	データ長	データ名	値
0x00	4	チャンクタイプ	”MTrk” (0x4D54726B)
0x04	4	データ長	0〜0xffffffff
0x08	可変長	トラックイベントデータ	イベントデータ列（後述）

また，最初の 8 バイトはヘッダチャンクと同様にチャンクタイプとデータ長である。ただし，トラックチャンクの場合，チャンクタイプは必ず ”MTrk” (0x 4D 54 72 6B) になる。データ長は，そのチャンクのそれより後ろにある演奏データの総バイト数である。

トラックイベントデータ

データ長の後には，トラックイベントが続いていく。トラックイベントは MIDI イベントを表すためのもので，5.2 節以降で詳しく説明する。

5.1.4　SMF のバイトオーダー

以上の説明を読んだ上で，手元の SMF ファイルをバイナリエディタで開き，中身と上記の解説とを見比べて理解に努めて欲しい。

また，そうすると値はビッグエンディアンで記録されていることに気が付くであろう（データ長の値 6 は 0x 00 00 00 06 の順番で保存されているのであって，0x 06 00 00 00 の順番ではない）。

Windows OS 上のアプリケーションは通常リトルエンディアン形式でデータを保持するため，ファイルからの読み出し時および書き出し時にはバイトオーダー（エンディア

ン）を変換する必要がある。バイトオーダーを変換する関数は標準では用意されていないようであるから[5]，リトルエンディアンからビッグエンディアンへの変換関数とビッグエンディアンからリトルエンディアンへの変換関数を自前で用意する必要がある。ただし，幸いにもそれら 2 つの関数は実際には同一の処理を行えば済むので，実装するのは片方だけでよいことになる。

4 バイト整数 (long int) のエンディアンを変換する関数の例をリスト 5.1 に示す。

リスト 5.1 エンディアン変換関数

```
long int convert_endian_long(long int obj) {
  long int ret = 0;  /* 結果（戻り値） */
  int i;

  for (i=0; i<4; i++) {
    ret = ret << 8;                     /* 1 バイト分ずらす */
    ret |= ((unsigned char *)&obj)[i];  /* メモリから 1 バイト取り出して加算 */
  }

  return ret;
}
```

なおこの例では入力も出力も整数であるが，実際に SMF ファイルを読み書きする場合には，バイトオーダー変換時に文字列を経由して入出力する関数を用意した方がおそらく便利だろう。

プログラミングメモ 12 — ファイルの読み書き (1)

本テキストは C 言語やその標準ライブラリ自体の解説書ではないため，標準 IO ライブラリを用いたファイルの読み書きについてはこのスペースを用いて簡単に説明する。

まず，ファイルの読み書きはストリームの読み書きの一種である。ストリームとは，ファイルの他，ネットワーク間やプロセス間，あるいは周辺機器との通信データのように，一次元状に並んだ（読み書きあるいは送受信される）データ列を指す。標準 IO ライブラリ（ヘッダファイルは stdio.h を用いる）では様々なストリーム操作用の関数が定義されているが，ファイル操作用にも FILE 構造体というデータ構造を軸に各種のファイル入出力用関数が用意されている。

ファイルのオープンおよびクローズ用には以下の関数が用意されている。

[5] ネットワーク関連では，ホストバイトオーダーとネットワークバイトオーダーを相互に変換する関数群が用意されている。

fopen()

fopen() はファイルを開く関数であり，そのプロトタイプ宣言は次のとおりである。

```
FILE *fopen(const char *path, const char *mode);
```

第 1 引数には開くファイル名を指定し，第 2 引数には開くモードを指定する。主要なモードには，読み込み用に開く"r"，書き込み用に開く"w"，追加書き込み用に開く"a"があり，これらを文字列として記述して指定する。また，その他に，（テキストモードではなく）バイナリモードで開く"b"が用意されており，これはテキストモードを持たない UNIX 系 OS では単に無視されるだけであるが，テキストモードとバイナリモードを区別する Windows OS では有効である。SMF ファイルはバイナリファイルであるから，"b"は必ず指定しておく必要がある[*6]。

関数の戻り値は FILE 構造体へのポインタ (FILE *) であり，もしエラーが発生すれば NULL が返される。

fclose()

fclose() はファイルを閉じる関数であり，そのプロトタイプ宣言は次のとおりである。

```
int fclose(FILE *fp);
```

第 1 引数には（既に開いているファイルの）FILE 構造体へのポインタを指定する。関数が正常に終了すれば 0 が，正常に終了しなければ定数 EOF[*7] が返される。

ファイルからの入力用には以下の関数が用意されている。

fgetc()

fgetc() はファイルから 1 文字（1 バイト）分読み取る関数であり，そのプロトタイプ宣言は次のとおりである。

```
int fgetc(FILE *fp);
```

第 1 引数には FILE 構造体へのポインタを指定する。読み込まれた 1 バイトが戻り値として（int 型で）返される。また，エラーが発生した場合またはファイルの終端に到達した場合には EOF が返される。

[*6] バイナリファイルをデフォルトのテキストモードで開いてしまうと改行コード (0x0D) の扱いで問題が生じる。
[*7] EOF は End Of File の略で，ファイルの終端を意味する。

fread()

fread() はファイルから指定したデータ型を配列で count 個読み取る関数であり，そのプロトタイプ宣言は次のとおりである。

```
size_t fread(void *buf, size_t size, size_t count, FILE *fp);
```

この関数は $size \times count$ バイト分のデータを読み取り，それを buf に格納する（buf は単一の要素を指すポインタでも良いし，配列の先頭を指すポインタでも構わない）。size は 1 要素のバイト数であり，count は読み込む要素数である。戻り値は読み込んだ要素数になる（読み込んだバイト数ではないことに注意）。エラーが発生した場合またはファイルの終端に到達した場合には n より小さい値が返される。

エラーチェック用に以下の関数が用意されている[*8]。

feof()

feof() はファイルポインタの位置がファイルの終端かどうかを調べる関数であり，そのプロトタイプ宣言は次のとおりである。この関数は現在位置がファイルの終端でなければ 0 を，そうでなければ 0 以外の値を返す。

```
int feof(FILE *fp);
```

ferror()

ferror() はエラーが発生しているかどうかを調べる関数であり，そのプロトタイプ宣言は次のとおりである。この関数はエラーが発生していなければ 0 を，エラーが発生していれば 0 以外の値を返す。

```
int ferror(FILE *fp);
```

以上を踏まえて，ファイルの先頭 n バイトを読み込んで，それを各バイトごとに 16 進数で表示するプログラムをリスト 5.2 に示す。このプログラムは引数として先頭から何バイトまでを読み込んで表示するかを指定することができるようにしているが，ファイル名は定数として記述しているので読み込むファイルを変更するためには毎回コンパイルし直さなければならない。なお，このようにデータをそのまま表示することをダンプ (dump) すると言う。

[*8] これらは実際は関数ではなくマクロで実装されるのが普通である。

リスト 5.2 ファイル内容の表示 (dumpfile.c)

```
#include <stdio.h>

int main(int argc, char **argv) {
  char filename[] = "dumpfile.c";  /* 読み込むファイル名 */
  FILE *fp;  /* FILE 構造体へのポインタ */
  int n;     /* 先頭から何バイトまで表示するか */
  int i;     /* ループ制御用変数 */
  int buf;   /* データ読み込み用バッファ */
  int c;     /* 見栄えをよくするための桁数カウンタ */

  /* 表示するバイト数を初期化 */
  if (argc > 1) {
    sscanf(argv[1], "%u", &n);
  } else {
    n = 64;
  }

  /* ファイルを読み取りモード＋バイナリモードで開く */
  fp = fopen(filename, "rb");
  /* エラーがあればその旨を表示して終了 */
  if (fp == NULL) {
    printf("Cannot open file %s.\n", filename);
    return 1;
  }

  c = 0;  /* 桁数カウンタを初期化 */
  for (i=0; i<n; i++) {
    /* ファイルから 1 バイト読み込んでバッファに格納 */
    buf = fgetc(fp);
    /* ファイルの終端とエラーの発生をチェック */
    if (buf == EOF) {
      /* エラーであればその旨を表示 */
      if ( ferror(fp) ) { printf("\nAn error occured.\n"); }
      break;
    }

    /* 読み込んだ内容を表示 */
    printf("%02X ", buf);
    /* 見やすくするために 16 桁目で改行する */
    if (++c == 16) { printf("\n"); c = 0; }
  }

  /* ファイルを閉じる */
  fclose(fp);

  return 0;
}
```

プログラミングメモ 13 — ファイルの読み書き (2)

ここではヘッダチャンクの読み込みを行うプログラムを作成する。

まずヘッダチャンクの内容を格納するデータ構造を設計しよう。ヘッダチャンクは全て決まったデータ型が決まった個数あるだけの静的な内容であるから，ヘッダチャンク用に構造体を定義するのが良いだろう。そうすれば，各値を別々の変数で保持しておくより，構造体の変数 1 つでアクセス可能なために変数名が分かりやすくなる。具体的には次のように定義しておけばよい。なお，short int は 2 バイト，long int は 4 バイトの整数である。

```
struct header {
  char chunktype[4];             /* チャンクタイプ：文字列（4 バイト）*/
  long unsigned int length;      /* データ長：uint（4 バイト）*/
  short unsigned int format;     /* フォーマットタイプ：uint（2 バイト）*/
  short unsigned int numtrack;   /* トラック数：uint（2 バイト）*/
  short unsigned int timebase;   /* タイムベース：uint（2 バイト）*/
};
```

これらの変数（構造体のメンバ）にファイルから読み込んだ値を代入する。ファイル（ストリーム）からの読み込みデータは本質的にバイト列であるから，ファイルからの読み込みそれ自体以外に，読み込んだバイト列を整数値などに変換する作業が必要であるのと，また SMF のビッグエンディアンからローカルマシンの CPU のバイトオーダー（Intel 系 CPU であればリトルエンディアン）に変換する作業が必要である。

これらの 3 つの作業をプログラミングする場合には複数の手法や書き方が考えられるが，(1) ここでは fread() 関数を用いてヘッダチャンク分のデータを全て一度に読み込んで（整数値などへの変換も同時に行われる），(2) その後に一つ一つの値のエンディアンを変換する方法を用いることにする。

まず (1) のプロセスは次のような流れで行う（エラー処理などは省いてあるが本来は必要である）。

```
#define SIZE_OF_SMFHEADER 14

...

struct header hdr;

fp = fopen(filename, "rb");
fread(&hdr, SIZE_OF_SMFHEADER, 1, fp);
```

fread() 関数自体についてはプログラミングメモ 12 で軽く説明してあるので，そちらのテキストも併せて読んで欲しい。ファイルのオープン，すなわち fopen() の呼び出しも同様に既に説明してあるので，ここでは fread() の動作を解説する。この fread() 関数では，ヘッダチャンクのバイト数（構造体のサイズ）分ファイルから読み込んで*9，それを header 構造体の変数 hdr に格納している。第 3 引数は（配列として）読み込む要素数であり，当然構造体は 1 つだけであるから，1 を指定している。

ここで沸き起こる疑問は，なぜ読み込んだバイト列が構造体の各メンバに（順序を指定していないように見えるのに）正しく格納されているのかということと，バイト列をそのまま構造体に代入しては型がどうなるのかということだろう。実は C 言語の構造体は，ソースコード中で宣言した順番でメモリに連続領域で割り当てられるようになっている。例えば上記の struct header 型の hdr 構造体の場合は，次のように割り当てられているはずだ（図 5.3）。

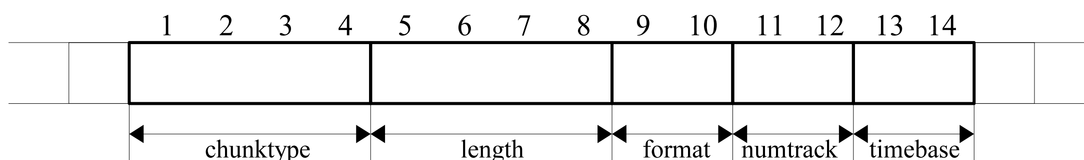


図 5.3　構造体のメモリ割り当て

そして，fread() 関数で読み込まれたバイト列は単純にそのままの順番でメモリに書き込まれる。すると，各変数のバイト数さえ一致していれば，各変数と書き込まれたバイト列の対応関係も一致することになる。型に関しては，C 言語ではメモリ上のバイト列をどのように扱うかということであり，メモリに正しくバイト列が格納されている場合には正常に動作するようになっている。従って，この場合は特別に型変換のような作業を行う必要はない。

ここで一度，以上の動作を行うプログラムを作成し，構造体の各値を出力するなどして上記の振る舞いを各自確認して欲しい（エンディアンはいまだ変換していないことに注意）。また，その際にはバイナリエディタで対象ファイルをダンプ表示し，その結果と見比べると分かりやすいだろう。

次に (2) のプロセス，すなわちエンディアン変換を構造体の各メンバに実施する必要がある。これは 5.1.4 節のリスト 5.1 を流用すればよいだろう。char 型の配列は正しいバイ

*9 本来，読み込むバイト数は sizeof(struct header) で求めたい。しかし，アドレスは 2 の乗数（すなわち 2，4，8，…）区切りで割り当てられるため，構造体のサイズが 2 の乗数でない場合はパディングが最後に挿入されて 2 の乗数のサイズに揃えられてしまう。例えば `struct example { char a; };` という構造体のサイズは 1 ではなく 2 や 4 となる（この値は処理系などによって異なる）。そのために読み込みバイト数を求めるために sizeof() を利用できない。

トオーダーで格納されているから，残りの long unsigned int（32 ビット長であるから uint32 と呼ぼう）と short unsigned int（同様に uint16 と呼ぼう）を変換する関数を用意する。これらはリスト 5.1 を変更して，次のように定義すればよいだろう。

```
long unsigned int convendian_uint32(long unsigned int obj) {
  long unsigned int ret = 0;
  int i;

  for (i=0; i<4; i++) {
    ret = ret << 8;
    ret |= ((unsigned char *)&obj)[i];
  }

  return ret;
}

short unsigned int convendian_uint16(short unsigned int obj) {
  short unsigned int ret = 0;
  int i;

  for (i=0; i<2; i++) {
    ret = ret << 8;
    ret |= ((unsigned char *)&obj)[i];
  }

  return ret;
}
```

この関数は，プログラムを見ての通り，変換されるデータを値渡しで受け取り，変換後のデータを戻り値で返している。従って，次のように使う。

```
hdr.length = convendian_uint32(hdr.length);
```

以上の処理を行って，ヘッダチャンクの値を画面に出力するプログラムをリスト 5.3 に示す。なお，char 型配列の chunktype メンバにはその末尾に NULL 文字 '\0' を置いていないため，printf() の %s 変換指定子を使用することができないことに注意したい（5 バイト目以降も表示されてしまう）。

リスト 5.3 ヘッダチャンクの読み込み (dumpfile.c)

```
#include <stdio.h>

/* SMF ヘッダチャンクのサイズ */
```

```
#define SIZE_OF_SMFHEADER 14

/* SMF ヘッダチャンクを保持する構造体 */
typedef struct header {
  char chunktype[4];             /* チャンクタイプ: "MThd" */
  long unsigned int length;      /* データ長 */
  short unsigned int format;     /* フォーマットタイプ */
  short unsigned int numtrack;   /* トラック数 */
  short unsigned int timebase;   /* タイムベース */
} SMFHEADER;

/* エンディアン変換関数（4 バイトの符号無し整数用）*/
long unsigned int convendian_uint32(long unsigned int obj) {
  long unsigned int ret = 0;
  int i;

  for (i=0; i<4; i++) {
    ret = ret << 8;
    ret |= ((unsigned char *)&obj)[i];
  }

  return ret;
}

/* エンディアン変換関数（2 バイトの符号無し整数用）*/
short unsigned int convendian_uint16(short unsigned int obj) {
  short unsigned int ret = 0;
  int i;

  for (i=0; i<2; i++) {
    ret = ret << 8;
    ret |= ((unsigned char *)&obj)[i];
  }

  return ret;
}

int main(void) {
  char filename[] = "sample.mid";  /* 読み込むファイル名 */
  FILE *fp;         /* ファイルポインタ */
  SMFHEADER hdr;    /* SMF ヘッダチャンクを格納するための構造体 */
  int i;            /* ループ制御用変数 */

  /* ファイルを開く */
  if ( (fp = fopen(filename, "rb")) == NULL) {
    printf("Cannot open file %s.\n", filename);
    return 1;
  }

  /* ヘッダチャンクを読み込んで SMFHEADER 構造体に書き込む */
```

```
    fread(&hdr, SIZE_OF_SMFHEADER, 1, fp);

    /* ファイルを閉じる */
    fclose(fp);

    /* 各メンバのエンディアンを変換 */
    hdr.length = convendian_uint32(hdr.length);
    hdr.format = convendian_uint16(hdr.format);
    hdr.numtrack = convendian_uint16(hdr.numtrack);
    hdr.timebase = convendian_uint16(hdr.timebase);

    /* 各メンバの値を表示 */
    printf("Chunk type: %c%c%c%c\n", hdr.chunktype[0], hdr.chunktype[1],
                                 hdr.chunktype[2], hdr.chunktype[3]);
    printf("Data length: %u\n", hdr.length);
    printf("Format type: %u\n", hdr.format);
    printf("Number of tracks: %u\n", hdr.numtrack);
    printf("Time base: 0x%04X (%u)\n", hdr.timebase, hdr.timebase);

    return 0;
}
```

5.2　トラックイベント

5.1.3 節で述べたように，トラックチャンクは最初にチャンクタイプとデータ長があり，その後に（通常は数多くの）トラックイベントが続く（図 5.4)。各トラックイベントの最初には必ずデルタタイムがあり，次に通常はイベントタイプがあり，最後にそのイベントのデータがある[*10]。

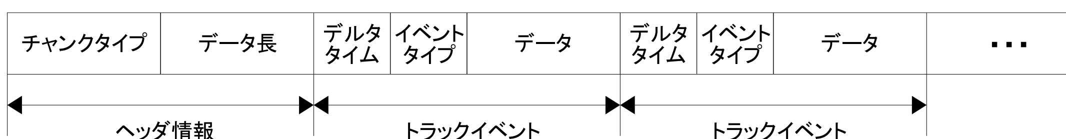


図 5.4　トラックイベント列

トラックイベントには MIDI イベント，システムエクスクルーシブイベント（SysEx イベント)，メタイベントの 3 種類あり，それぞれ別々の扱いをする必要がある。

デルタタイムの後のイベントタイプの値によって，これらは次の表 5.3 のように区別することができる。

それでは，これら 3 種類のトラックイベントについて詳しく説明していく。

[*10] 通常，トラックチャンクの最後には End of Track というイベントを置く。

表 5.3　トラックイベントの種類とイベントタイプ

トラックイベントの種類	イベントタイプの値
MIDI イベント	0x80〜0xEF
MIDI イベント（ランニングステータス利用時）	0x00〜0x7F
SysEx イベント	0xF0 または 0xF7
メタイベント	0xFF

5.2.1　デルタタイム

各イベントの説明に入る前に，各トラックイベントに共通のデルタタイムについて説明しよう。

まず，デルタタイムは物理的には SMF ファイル内では 5.2.2 節で説明される可変長データ表現 (variable-length value) として表されるが，論理的には一つの 4 バイトの符号無し整数と見なすことができる。

デルタタイムは，そのイベントが再生または実行される時間を，直前のイベントからの相対時間で表す（もしそのイベントがそのトラックの最初のイベントであれば，トラックの再生を開始してから待つ時間となる）。もしデルタタイムが 0 であれば，それは前のイベントと同時に再生されることを意味する。

このデルタタイム（5.1.2 節で行ったタイムベースの説明内でのクロックという用語に相当）は，タイムベースやテンポの値と組み合わして現実時間に変換される。特に，タイムベースが ticks per beat で表されている場合（最上位ビットが 0 の場合）と，frames per second で表されている場合（最上位ビットが 1 の場合）とで処理が異なるが，ここでは多く使われている前者の ticks per beat の計算方法のみを詳しく説明しよう。

5.1.2 節で解説したように，例えばタイムベース値が 384 (=0x0180) であれば，その値が 1 拍分（＝四分音符の長さ）に相当するため，もしあるイベントのクロック（デルタタイム値）が 192 であれば前のイベントから四分音符の 1/2 時間，すなわち八分音符分遅れて（八分音符分待ってから）再生または実行されることになる。これだけでは四分音符や八分音符の長さが分からないため，テンポの値を併用して実際の現実時間に直す。なお，テンポは 5.2.5 節で説明するメタイベントで指定されるが，もしテンポの指定がなければデフォルト値の 120 が用いられる。テンポの単位は beats per minute (bpm)，すなわち 1 分間の拍数（1 分間の四分音符数）を表している。もしテンポが 120 であれば，四分音符の長さは 0.5 秒（500 ミリ秒），八分音符の長さはその半分の 250 ミリ秒 (250ms) になる。以上の過程を経ると，例えばデルタタイム値が 192 であれば現実時間で 250ms に相当するということが計算でき，各イベントの時間軸上での位置が分かる。

以上の計算を式で表すと，イベント位置（直前のイベントからの待ち時間）は以下のよ

うになる（ただしメタイベントで指定されるテンポの値は bpm で表されないため，注意が必要である）。

$$
\begin{aligned}
\text{時間}\ [\mathit{second}] &= \frac{\text{デルタタイム}\ [\mathit{ticks}]}{\text{タイムベース値}\ [\mathit{ticks}/\mathit{beat}]} \Big/ \frac{\text{テンポ}\ [\mathit{beats}/\mathit{minute}]}{60} \\
&= \frac{\text{デルタタイム}\ [\mathit{ticks}] \times 60}{\text{タイムベース値}\ [\mathit{ticks}/\mathit{beat}] \times \text{テンポ}\ [\mathit{beats}/\mathit{minute}]}
\end{aligned}
$$

なお，トラックのタイトル名や著作権情報のような時間に関係のないイベントは，トラックの最初にデルタタイム 0 で配置するようにする。

5.2.2　可変長データ表現

デルタタイムやデータ長などの頻出する値は，SMF ファイルで記録する場合には，データの記憶容量を節約するために可変長のバイト数で記録するようになっている。この特別な形式を，ここでは可変長データ表現 (variable-length value) と呼ぶことにする。

可変長データ表現では一つの値が 1〜4 バイトの可変長で表される。各バイトの下位 7 ビットでデータを表し，最上位ビットで次のバイトが可変長データ表現のバイト列の続き（一部）であるかどうかを示す。実際のデータは，各バイトの最上位ビットを除いたビット列をビッグエンディアンの順序そのままで連結したものである。従って，最大 7×4 で 28 ビットのバイト列，すなわち 0x0〜0x0FFFFFFF の値を表すことができる。もし記録されているデータ長が 4 バイト未満であれば，それらに相当する上位ビットは全て 0 と見なされる。

例をいくつか表 5.4 に挙げる。

表 5.4　可変長データ表現の値の例

実際の値	実際の値のビット列	可変長データ表現の値のビット列
0x00000000	00000000 00000000 00000000 00000000	00000000
0x0000007F	00000000 00000000 00000000 01111111	01111111
0x00000080	00000000 00000000 00000000 10000000	10000001 00000000
0x00003FB5	00000000 00000000 00111111 10110101	11111111 00110101
0x00100100	00000000 00010000 00000001 00000000	11000000 10000010 00000000
0x0F0F0F0F	00001111 00001111 00001111 00001111	11111000 10111100 10011110 00001111

可変長データ表現の値をファイルから読み込んで実際の値に変換するプログラム例をリスト 5.4 に示す。なお，この関数は 4 バイトを超える可変長データ表現のデータに対するエラーチェックは行っていないので注意して欲しい。

リスト 5.4 可変長データ表現の値から実際の値への変換

```
long unsigned int read_variable_length_value(FILE *fp) {
  long unsigned int ret = 0;  /* 戻り値（実際の値） */
  int buf = 0x80;             /* 読み込み用バッファ */

  /* 1 バイト目か，前に読み込んだバイトの最上位ビットが 1 であれば */
  while (buf & 0x80) {
    /* ファイルから 1 バイト読み込む */
    buf = fgetc(fp);
    if (buf == EOF) {
      /* ファイルの終端に到達した場合，および読み込みエラーが発生した場合は */
      /* 0x7FFFFFFF より大きい値を返す */
      return ( (long unsigned int)-1 );
    }

    ret = ret << 7;
    ret |= buf & 0x7F;  /* 最上位ビットを除いたビット列を加える */
  }

  /* 変換結果を返す */
  return ret;
}
```

課題 9

実際の値（long unsigned int）から可変長データ表現の値（char 型配列）に変換する関数を作成せよ。

5.2.3　MIDI イベント

MIDI イベントは MIDI チャンネルメッセージ（SysEx メッセージを除く通常の MIDI メッセージ）を表すイベントであり，デルタタイムの後にそのまま（MIDI 出力デバイスに送信するバイト列と同じように）MIDI メッセージを記述する（図 5.5）。

図 5.5　MIDI イベント

従って，MIDI イベントのデータ長 —— イベントタイプ（MIDI メッセージのステータスバイト）とそのデータのデータ長 —— は MIDI メッセージの種類によって異なる。

具体的には，ノートオフ (0x8n)，ノートオン (0x9n)，ポリフォニックキープレッシャー (0xAn)，コントロールチェンジ (0xBn)，ピッチベンドチェンジ (0xEn)，および全モードメッセージ (0xBn) はステータスバイトを含めて 3 バイト，プログラムチェンジ (0xCn) とチャンネルプレッシャー (0xDn) はステータスバイトを含めて 2 バイトである（p.30 の表 3.1 と表 3.2）。

通常の MIDI 入出力と同様に，SMF の MIDI イベントでもランニングステータスを利用することができる。ランニングステータスとは，データ量を節約する仕組みで，前の MIDI メッセージとステータスバイトが同じであればステータスバイトを省略してデータバイトのみを送信する（SMF ファイルに記述する）ことである。これは，MIDI チャンネルメッセージはステータスバイトの最上位ビットが 1，データバイトの最上位ビットが 0 になっているので，各バイトを見ればそのバイトがステータスバイトかデータバイトかを知ることができ，かつ各メッセージにおけるデータ長が決まっているために可能となっている。もちろん直前の MIDI メッセージのステータスバイトを保持しておく必要がある。

なお，曲の途中から再生する場合に問題が生じるなど，ランニングステータスの利用は通常は推奨されない。ただし，SMF ファイルを読み込んだり再生する場合は当然対応する必要がある。

SMF ファイル内におけるランニングステータス利用時の例を図 5.6 に示す。

図 5.6　ランニングステータス

5.2.4　SysEx イベント

SysEx イベントは主にシステムエクスクルーシブメッセージ（SysEx メッセージ）を格納するために用いられる。通常の SysEx メッセージを格納する SysEx イベントは，デルタタイムの後のイベントタイプは 0xF0 （1 バイト）であり，その後にデータ長（可変長データ表現），データ列（可変長）が続く（図 5.7）。また，データ列の最後は 0xF7 で閉じられる。

また，他のシステムメッセージ（システムコモンメッセージ，システムリアルタイムメッセージ）などを送信するために，イベントタイプが 0xF7 で最後に 0xF7 を置かない形式の SysEx イベントがある（図 5.8）。

大きなサイズの SysEx メッセージを一度に送ろうとした場合，そのメッセージの送受

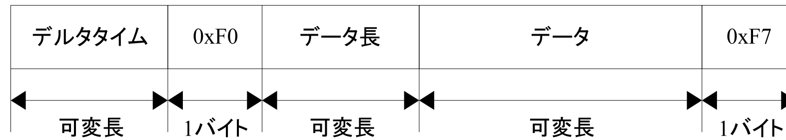


図 5.7　SysEx イベント

図 5.8　イベントタイプ F7 の SysEx イベント

信に時間がかかって他の MIDI メッセージが遅れてしまうことが考えられる（そしてそれは演奏がずれることを意味する）。そのために，一つの SysEx メッセージを複数の SysEx イベントに分割して記録する形式が用意されている。分割された SysEx メッセージの最初のイベントはイベントタイプが 0xF0 でデータ末尾に 0xF7 を置かず，途中のイベントはイベントタイプが 0xF7 でデータ末尾に 0xF7 を置かず，最後のイベントはイベントタイプが 0xF7 でデータ末尾に 0xF7 を置く（図 5.9）。

図 5.9　分割用の SysEx イベント

なお，SysEx イベントの直後にランニングステータスを利用することはできない（MIDI イベントにはステータスバイトを必ず付ける必要がある）。

5.2.5　メタイベント

メタイベントは，その SMF ファイルに関するメタデータを格納するためのイベントであり[*11]，MIDI デバイス間でこれらのイベントのデータが送受信されることはない。メタイベントでは，デルタタイムの後にメタイベントを示すイベントタイプ 0xFF（1 バイト）

[*11] メタデータとは，データに関する（補助的な）データを意味する。例えば，本においては中身の文章をデータとすると，作者名や出版社などの情報がメタデータにあたる。

があり，その後にメタイベントの種類を示すメタイベントタイプ（1 バイト），データ長（可変長データ表現）が続き，最後に実際のデータが来る（図 5.10）。

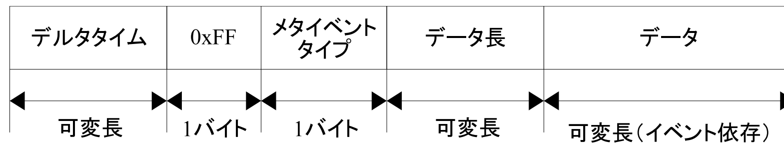


図 5.10　メタイベント

メタイベントの一覧を表 5.5 に載せる。

表 5.5　メタイベントの一覧

イベント名	メタイベントタイプ	データ長
シーケンス番号	0 (0x00)	2
テキスト	1 (0x01)	可変
著作権表示	2 (0x02)	可変
シーケンス名/トラック名	3 (0x03)	可変
楽器名	4 (0x04)	可変
歌詞	5 (0x05)	可変
マーカー	6 (0x06)	可変
キューポイント	7 (0x07)	可変
MIDI チャンネルプレフィクス	32 (0x20)	1
エンドオブトラック	47 (0x2F)	0
セットテンポ	81 (0x51)	3
SMPTE オフセット	84 (0x54)	5
拍子	88 (0x58)	4
調	89 (0x59)	2
シーケンサ固有メタイベント	127 (0x7F)	可変

ここではエンドオブトラックとセットテンポのみを説明する。

エンドオブトラック

エンドオブトラック (End of Track) メタイベントは，トラックの末尾に置かれ，そのトラックチャンクのデータ列がそこで終わることを意味する。

データは持たない。

セットテンポ

セットテンポ (Set Tempo) メタイベントは，テンポを四分音符あたり（1 拍あたり）何マイクロセカンド (μs) であるかという形式で，3 バイトの符号無し整数で表す。従って，例えばテンポが 60 であれば，それは 60 beats per minute (bpm) の意味であるから，1 秒間に 1 拍，すなわちセットテンポの値は 1,000,000 となる。もしセットテンポが無けれ

ば，デフォルト値の 120bpm，すなわち $60,000,000/120 = 500,000$ と見なされる。

セットテンポは通常は 1 番目のトラックチャンクに置かれている。

5.3　SMF ファイルの読み込み

本 5.3 節では，SMF ファイルを読み込むための DLL (Dynamic Link Library) ファイルを作成する。DLL はライブラリであり，複数のプログラムから呼び出して使うことができる。

5.3.1　DLL の作成

SMF ファイルを読み込む DLL を作成する前に，まずは DLL 自体の作成の手順を説明する。ここでは累乗を返す簡単なライブラリを作成し，それを別のプログラムから呼び出すことにする。

次のリスト 5.5 プログラムを打ち込んで，mymath.c という名前で保存する。そして `cl /c mymath.c` でコンパイルだけしておく（mymath.obj のみが生成される）。なお，この mypower() は，第 1 引数（実数）の第 2 引数（整数）乗の値，すなわち d^n を返す関数である。

リスト 5.5 ライブラリのソースコード (mymath.c)

```
/* d の n 乗を返す */
double mypower(double d, int n) {
  double ret = 1.0;

  if (n < 0) {
    n *= -1;
    d = 1.0 / d;
  }

  for ( ; n>0; n--) {
    ret *= d;
  }

  return ret;
}
```

次に，次のリスト 5.6 の内容を持つファイルを作成し，mymath.def という名前で保存する。これはリンク時に必要な .lib ファイルを生成するために必要となる（マルチメディアライブラリの winmm.lib に相当する）。ここでは mypower() 関数をエクスポートしており，外部から mypower() 関数を呼び出せるようにしている。

リスト 5.6 ライブラリの DEF ファイル (mymath.def)

```
LIBRARY mymath
EXPORTS
  mypower
```

そして次のコマンドで .lib ファイルと .dll ファイルを生成する（.exp というファイルも生成される）。他のプログラムからこのライブラリを使用するときは，リンク時にこの .lib ファイルが必要となり，実行時にこの .dll ファイルが必要となる。

```
link mymath.obj /def:mymath.def /dll
```

これで DLL ファイルが作成できたので，次にこのライブラリを使用するプログラムを作成する。しかし，その前にこのライブラリ用のヘッダファイルを作っておくと便利であろう（これはマルチメディアライブラリの mmsystem.h に相当する）。次のリスト 5.7 を打ち込んで，mymath.h というファイルを作る。これは関数のプロトタイプ宣言であり，型の整合性をコンパイル時にチェックするために使われる。

リスト 5.7 ライブラリのヘッダファイル (mymath.h)

```
double mypower(double d, int n);
```

最後にこのライブラリを使うプログラム本体をリスト 5.8 に示すので，このプログラムを testdll.c という名前で作成する。なお，プリプロセッサ命令 #include の <. . .> はシステムが用意しているヘッダファイルを，"..." はユーザが用意したヘッダファイル（カレントフォルダからのパスになる）を意味する。

リスト 5.8 ライブラリを使用するプログラム (testdll.c)

```
#include <stdio.h>
#include "mymath.h"

int main(void) {
  printf("%lf\n", mypower(3.0, 0));
  printf("%lf\n", mypower(3.0, 1));
  printf("%lf\n", mypower(3.0, 10));
  printf("%lf\n", mypower(3.0, -1));
  printf("%lf\n", mypower(3.0, -10));

  return 0;
}
```

コンパイルは `cl testdll.c mymath.lib` で行う。あるいは，リンクオプションを

明示して `cl testdll.c /link mymath.lib` でも構わない。ここで，ヘッダファイル mymath.h はコンパイル時の型の整合性チェックに使われ，mymath.lib はリンク時に DLL ファイルとのリンク情報を得るために使われ，DLL ファイル mymath.dll は実行ファイルの実行時に実際にリンクしてライブラリの関数を呼び出すために使われる。

5.3.2　ライブラリの基本設計

ライブラリを設計するにあたって，まずこのライブラリがどのように利用されるかを考える。ここでは SMF ファイルの内容の表示（ダンプ）や再生を行うプログラムの共通ライブラリとして使うことを想定することにしよう。そのようなプログラムの依存関係は次のようになるだろう（図 5.11）。プログラムの正しさを確認しながらライブラリを作っていくために，ライブラリの作成と同時に SMF ファイルのダンププログラムを作っていくことにする。

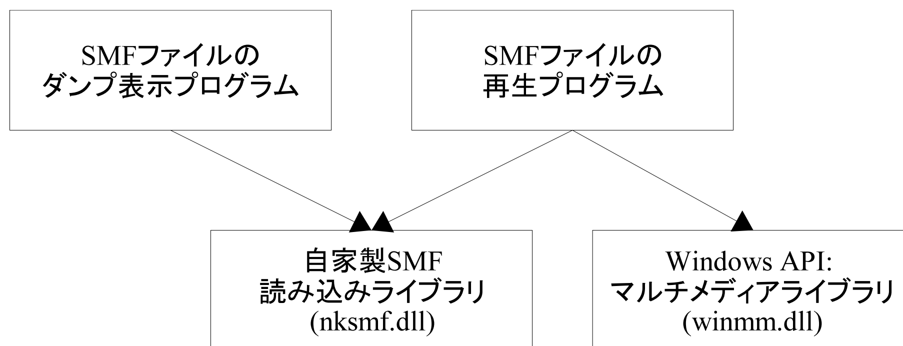


図 5.11　ライブラリの用途例とプログラムの依存関係

ライブラリの名称

ライブラリの名称（ファイル名）は，smf.dll だと平凡過ぎるので，nksmf プレフィクス[*12] を使って nksmf.dll や nksmf.c などの名前を付けることにする[*13]。もちろん，この名称は他の何であっても（的確に事象を表していれば）構わない。

[*12] プレフィクス（prefix; 接頭辞）とは何かの名前や単語の先頭に（共通に）付ける言葉や単語を指す一般的な用語。また，最後に付ける場合はサフィックス（suffix; 接尾辞）と呼ぶ。例えば山田という人を「Mr. 山田」と呼ぶ場合の「Mr.」は語頭に付けるのでプレフィクスであり，「山田さん」と呼ぶ場合の「さん」は語尾に付けるのでサフィックスであると言える。

[*13] 深い意味はなく，筆者の所属研究室に由来する。

関数名のプレフィクス

各関数名も，他の関数との名前の衝突を避けるために，smf というプレフィクスを付けることにする。これは，Windows API の MIDI 関連の関数に midiOut や midiIn といったプレフィクスが付いているのと同様の意味合いである。

対象フォーマット

対象フォーマットは，よく利用されている，マルチトラックが利用可能なフォーマット 1 にする。また，フォーマット 1 はフォーマット 0 に対して上位互換性があるので，フォーマット 1 に対応するということはフォーマット 0 にも対応することになる。

データ操作

SMF ファイルの容量は通常は小さいため（数百キロバイト程度），一度に全てのデータをメモリ（文字列; すなわち char 型配列）に読み込んで，以降はその文字列に対してのみ操作するという方法を考えることができる。これは，その都度ファイルからデータを読み込む方法よりも，実行速度の点でもプログラミングの点でも簡便な方法であろう。しかし，ここではその都度ファイルからデータを読み込む方法を利用する（今までに作成したソースコードがそのまま流用できるため）。

その他

同時に複数のファイルを扱えるようにする。

これら以外にも決めなければならない基本的な事柄は多いが，（特に経験が浅い場合は）実際にライブラリを作りながら考えていく方が分かりやすいため，必要になった時点で順次決定していくことにしよう。

5.3.3　ファイルのオープンとクローズ

自作ライブラリに何をさせるかというと，まず次の内容がすぐに思い浮かぶ。

- ファイルのオープン
- ファイルのクローズ
- ヘッダチャンクのチェック（先頭数バイトをチェックすることで SMF として正しいファイルであるかがおおよそ分かる）
- ヘッダチャンクの読み込み

ヘッダのチェックは，SMF ではないファイルを開いたときに（クライアントプログラムではなくライブラリが）チェックする必要があるために用意する。

これらはそれぞれ独立の関数として実装できるだろうから（ヘッダのチェックはファイルのオープンに含めてもよいだろうが），それぞれ次の関数名を付けることにする。

- smfOpenFile
- smfCloseFile
- smfHasValidHeader
- smfGetHeaderChunk

関数名は英語の語順で付ける，すなわち「動詞-目的語」や「動詞-補語」の順で名付けるのが普通である（主語は自明であるから付けない）。また，HasValidHeader のように，何かをチェックする関数で，かつ戻り値がブーリアン[*14] の場合，関数名やメソッドの最初に has や is を付けることが（最近のプログラミングの傾向として）多い。これらの関数名は，それらの名付け方の慣習に倣ったものであり，以降も同様に名付けていく。

次にこれらの関数のインタフェースを設計しよう。これはプロトタイプ宣言の内容，つまり関数の引数と戻り値を決めることである。このライブラリは複数のファイルを扱えるべきであるから，各関数に対してどのファイルであるかを指定する必要がある。これは，例えばファイル構造体のポインタ (`FILE *fp`，ファイルポインタと呼ぶ) を持ちまわすことで解決でき，例えば次のような引数が考えられる（smfOpenFile() にファイル名を指定する必要があるのは自明であろう）。なお，`FILE **fp` は「ファイル構造体へのポインタ」へのポインタ，すなわちポインタのアドレスを意味する。

- smfOpenFile(FILE **fp, char *filename)
- smfCloseFile(FILE *fp)
- smfHasValidHeader(FILE *fp)
- smfGetHeaderChunk(FILE *fp, SMFHEADER *header)

ここで，smfOpenFile() では fp に値を代入し，それ以外の関数では fp から値を読み出す（参照する）ことに注意して欲しい。

smfOpenFile() では，fopen() 関数のように戻り値としてファイルポインタを戻すように設計することも考えられる。その場合，`FILE *smfOpenFile(char *filename)` となるだろう。また，smfGetHeaderChunk() でもヘッダ情報の構造体を引数で渡すか戻り値で返すかの 2 通りが考えられる。しかし，ここではエラーを戻り値で示すことにし，引

[*14] ブーリアン (boolean) は真か偽をとる型，すなわち真偽値を表す型である。C 言語では 0 が偽，それ以外が真であるため，通常は int 型の 0 とそれ以外の数値（通常は 1）で代用される。

数としてファイルポインタや構造体のポインタを渡すようにする。なお，引数で渡す場合はライブラリを使用するプログラム側でメモリ管理を行う必要があり，戻り値として返す場合はライブラリ側でメモリ管理を行う必要があることに注意する。

この他にも，midiOut や midiIn 系の関数のようにデバイス ID やデバイスハンドルのようなものを持ちまわすこともできるが，この場合はその ID やハンドルとファイルポインタを何らかの仕組みで関連付ける必要があるため，ファイルポインタを直接指定する方が簡潔であろう。

次に戻り値であるが，smfHasValidHeader() はヘッダが正しければ真（0 以外の値）を，ヘッダが正しくなければ偽（値 0）を返すのが自然であろう。それ以外の関数ではエラーがあるかないかを返すことにする。従って，各関数のプロトタイプ宣言は次のようになる。

- int smfOpenFile(FILE **fp, char *filename)
- int smfCloseFile(FILE *fp)
- int smfHasValidHeader(FILE *fp)
- int smfGetHeaderChunk(FILE *fp, SMFHEADER *header)

なお，ヘッダチャンクの構造体 (SMFHEADER) はプログラミングメモ 13 で定義したものをそのまま使うことにする。

今までの設計で, nksmf.h と nksmf.def は次のように書ける（リスト 5.9, リスト 5.10)。

リスト 5.9 nksmf.h

```
/* ヘッダチャンク */
typedef struct smfheader {
  char chunktype[4];           /* チャンクタイプ: "MThd" */
  long unsigned int length;    /* データ長 */
  short unsigned int format;   /* フォーマットタイプ */
  short unsigned int numtrack; /* トラック数 */
  short unsigned int timebase; /* タイムベース */
} SMFHEADER;

int smfOpenFile(FILE **fp, char *filename);
int smfCloseFile(FILE *fp);
int smfHasValidHeader(FILE *fp);
int smfGetHeaderChunk(FILE *fp, SMFHEADER *header);
```

リスト 5.10 nksmf.def

```
LIBRARY nksmf
EXPORTS
  smfOpenFile
```

```
smfCloseFile
smfHasValidHeader
smfGetHeaderChunk
```

次に，ライブラリのプログラム (nksmf.c) を記述する前に，このライブラリを使うプログラムを記述する（実際に開発する場合は，少なくともこの規模のソフトウェアであれば，並行して行う）。前述したようにここではダンププログラムを作成するわけであるから，今のところはヘッダチャンクの表示までを行おう。smfOpenFile() や smfGetHeaderChunk() 関数は，エラーがなければ 0 を，エラーがあれば 0 以外の値を返すように設計すると，ダンププログラムは次のように記述できる（リスト 5.11）。なお，このプログラムは必ずコマンドライン引数を 1 つ取り，それにはダンプするファイル名を指定する。

リスト 5.11 smfdump.c

```
#include <stdio.h>
#include <string.h>
#include "nksmf.h"

int main(int argc, char **argv) {
  FILE *fp;       /* ファイル構造体へのポインタ */
  SMFHEADER hdr;  /* ヘッダチャンクを格納 */
  int ret;        /* 戻り値を保持 */
  int i;

  /* 引数が 1 つ指定されていなければ，Usage を表示して終了 */
  if (argc != 2) {
    fprintf(stderr, "usage: %s filename\n", argv[0]);
    return 1;
  }

  /* ファイルを開く */
  ret = smfOpenFile(&fp, argv[1]);
  if (ret) {
    fprintf(stderr, "smfOpenFile(): %s\n", strerror(ret));
    return 2;
  }

  /* ヘッダをチェックする */
  if ( !smfHasValidHeader(fp) ) {
    fprintf(stderr, "Given file is not a valid SMF.\n");
    return 3;
  }

  /* ヘッダチャンクのダンプ */
  printf("----- Header Chunk -----\n");
```

```
  ret = smfGetHeaderChunk(fp, &hdr);
  if (ret != 0) {
    fprintf(stderr, "Cannot get the header chunk.\n");
    return 4;
  }

  for (i=0;i<4; i++) {
    printf("%c", hdr.chunktype[i]);
  }
  printf("\n");
  printf("hdr.length: %u\n", hdr.length);
  printf("hdr.format: %u\n", hdr.format);
  printf("hdr.numtrack: %u\n", hdr.numtrack);
  printf("hdr.timebase: %u\n", hdr.timebase);

  /* ファイルを閉じる */
  smfCloseFile(fp);

  return 0;
}
```

なお，このプログラムはこの時点でコンパイルできる（リンクは当然できない）。cl /c smfdump.c でコンパイルを試してみて，打ち間違いがないかといったチェックをしておこう。

最後に, ライブラリ本体のソースコード (nksmf.c) を記述しよう。まず, smfOpenFile() と smfCloseFile() は次のように記述できる。関数の成功時には 0 を，失敗時（エラー発生時）には 0 以外の値を返すようにしている。失敗時には，smfOpenFile() の場合は fopen() で設定されるエラー値を，smfCloseFile() の場合は fclose() で設定されるエラー値を返す。これらの戻り値の設計にあたっては，fopen() や fclose() のドキュメンテーションを利用している。一度 fopen() や fclose() や，エラー関連の feof()，ferror()，strerror() 関数などのドキュメンテーションを熟読し，またそれらに関連してヘッダファイルで定義されている定数の値もあわせて調べて欲しい。なお，グローバル変数 errno のために errno.h をインクルードする必要がある。

```
/* SMF ファイルを開く */
int smfOpenFile(FILE **fp, char *filename) {
  *fp = fopen(filename, "rb");
  if ( !(*fp) ) {
    return errno;
  }

  return 0;
}
```

```
/* SMF ファイルを閉じる */
int smfCloseFile(FILE *fp) {
  if ( !fclose(fp) ) {
    return errno;
  }

  return 0;
}
```

次に，smfHasValidHeader() と smfGetHeaderChunk() 関数は次のように記述できる。なお，convendian_uint32() と convendian_uint16() 関数はプログラミングメモ 13 で定義しているものをそのまま利用しており，他の定数と名称を合わせるために SIZE_OF_SMFHEADER を SMF_HEADERSIZE に変更している。

```
/* SMF ヘッダチャンクのサイズ */
#define SMF_HEADERSIZE       14

/* Valid なヘッダのデータ */
#define SMF_VALIDHEADERLEN   8
#define SMF_VALIDHEADERSTR   "MThd\x00\x00\x00\x06"

/* ファイルの先頭数バイト（ヘッダチャンク部分）を調べ，
 * ファイルが SMF フォーマットを持っているかを検証する      */
int smfHasValidHeader(FILE *fp) {
  char buf[SMF_VALIDHEADERLEN];
  size_t ret;
  int i;

  if (!fp) {
    return 0;
  }

  /* ファイルの読み込み位置を先頭に戻す */
  rewind(fp);
  /* ファイルから SMF_VALIDHEADERLEN バイト分読み込む */
  ret = fread(buf, sizeof(char), SMF_VALIDHEADERLEN, fp);

  /* データサイズが本来のサイズよりも小さいか，データが一致しなければ検証に失敗 */
  if ( ret < SMF_VALIDHEADERLEN ||
       memcmp(buf, SMF_VALIDHEADERSTR, SMF_VALIDHEADERLEN) != 0 ) {
    return 0; /* invalid */
  }

  return 1; /* valid */
}

int smfGetHeaderChunk(FILE *fp, SMFHEADER *header) {
  size_t ret;
```

```
    /* ファイルの読み込み位置を先頭に戻す */
    rewind(fp);
    /* ファイルから SMF_HEADERSIZE バイト分読み込む */
    ret = fread(header, SMF_HEADERSIZE, 1, fp);

    /* ファイルの終端に到達するかエラーが発生すれば 0 以外の値を返す */
    if (ret < 1) {
      if ( feof(fp) ) {
        return EOF;
      } else {
        return errno;
      }
    }

    /* エンディアンを変換する */
    header->length = convendian_uint32(header->length);
    header->format = convendian_uint16(header->format);
    header->numtrack = convendian_uint16(header->numtrack);
    header->timebase = convendian_uint16(header->timebase);

    return 0;
  }
```

さて，上記の 3 つの定数やエンディアン変換用の 2 つの関数は（少なくとも今のところは）外部に公開する必要がなく，また実際に外部に公開していないため，これらの定数の定義や関数のプロトタイプ宣言は外部プログラムと共用している nksmf.h に書くべきではないだろう。本来はヘッダファイルを共用すべきではないのかもしれないが，ここでは nksmf.c に直接それらの宣言を書き足すか，別のヘッダファイルを作ることで対処すればいいだろう。本テキストでは nksmf-lib.h という，ライブラリプログラム (nksmf.c) からのみ読み込むヘッダファイルを作成することにし，内容は以下の通りである（リスト 5.12)。

リスト 5.12 nksmf-lib.h

```
/* SMF ヘッダチャンクのサイズ */
#define SMF_HEADERSIZE       14

/* Valid なヘッダのデータ */
#define SMF_VALIDHEADERLEN   8
#define SMF_VALIDHEADERSTR   "MThd\x00\x00\x00\x06"

long unsigned int convendian_uint32(long unsigned int obj);
short unsigned int convendian_uint16(short unsigned int obj);
```

そして今までのところの，ライブラリ nksmf.c の内容は以下の通りである（リスト 5.13)。

リスト 5.13 nksmf.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "nksmf.h"
#include "nksmf-lib.h"

/* SMF ファイルを開く */
int smfOpenFile(FILE **fp, char *filename) {
  *fp = fopen(filename, "rb");
  if ( !(*fp) ) {
    return errno;
  }

  return 0;
}

/* SMF ファイルを閉じる */
int smfCloseFile(FILE *fp) {
  if ( !fclose(fp) ) {
    return errno;
  }

  return 0;
}

/* ファイルの先頭数バイト（ヘッダチャンク部分）を調べ，
 * ファイルが SMF フォーマットを持っているかを検証する     */
int smfHasValidHeader(FILE *fp) {
  char buf[SMF_VALIDHEADERLEN];
  size_t ret;
  int i;

  if (!fp) {
    return 0;
  }

  /* ファイルの読み込み位置を先頭に戻す */
  rewind(fp);
  /* ファイルから SMF_VALIDHEADERLEN バイト分読み込む */
  ret = fread(buf, sizeof(char), SMF_VALIDHEADERLEN, fp);

  /* データサイズが本来のサイズよりも小さいか，データが一致しなければ検証に失敗 */
  if ( ret < SMF_VALIDHEADERLEN ||
       memcmp(buf, SMF_VALIDHEADERSTR, SMF_VALIDHEADERLEN) != 0 ) {
```

```
    return 0; /* invalid */
  }

  return 1; /* valid */
}

int smfGetHeaderChunk(FILE *fp, SMFHEADER *header) {
  size_t ret;

  /* ファイルの読み込み位置を先頭に戻す */
  rewind(fp);
  /* ファイルから SMF_HEADERSIZE バイト分読み込む */
  ret = fread(header, SMF_HEADERSIZE, 1, fp);

  /* ファイルの終端に到達するかエラーが発生すれば 0 以外の値を返す */
  if (ret < 1) {
    if ( feof(fp) ) {
      return EOF;
    } else {
      return errno;
    }
  }

  /* エンディアンを変換する */
  header->length = convendian_uint32(header->length);
  header->format = convendian_uint16(header->format);
  header->numtrack = convendian_uint16(header->numtrack);
  header->timebase = convendian_uint16(header->timebase);

  return 0;
}

/* エンディアン変換関数（4 バイトの符号無し整数用）*/
long unsigned int convendian_uint32(long unsigned int obj) {
  long unsigned int ret = 0;
  int i;

  for (i=0; i<4; i++) {
    ret = ret << 8;
    ret |= ((unsigned char *)&obj)[i];
  }

  return ret;
}

/* エンディアン変換関数（2 バイトの符号無し整数用）*/
short unsigned int convendian_uint16(short unsigned int obj) {
  short unsigned int ret = 0;
  int i;
```

```
    for (i=0; i<2; i++) {
      ret = ret << 8;
      ret |= ((unsigned char *)&obj)[i];
    }

    return ret;
  }
```

コンパイルの手順を再度示しておく。上記 2 行がライブラリのコンパイルであり，最後の 1 行がダンププログラムのコンパイルである。

```
cl /c nksmf.c
link nksmf.obj /def:nksmf.def /dll

cl smfdump.c nksmf.lib
```

ダンププログラムの実行例を図 5.12 に示しておく。

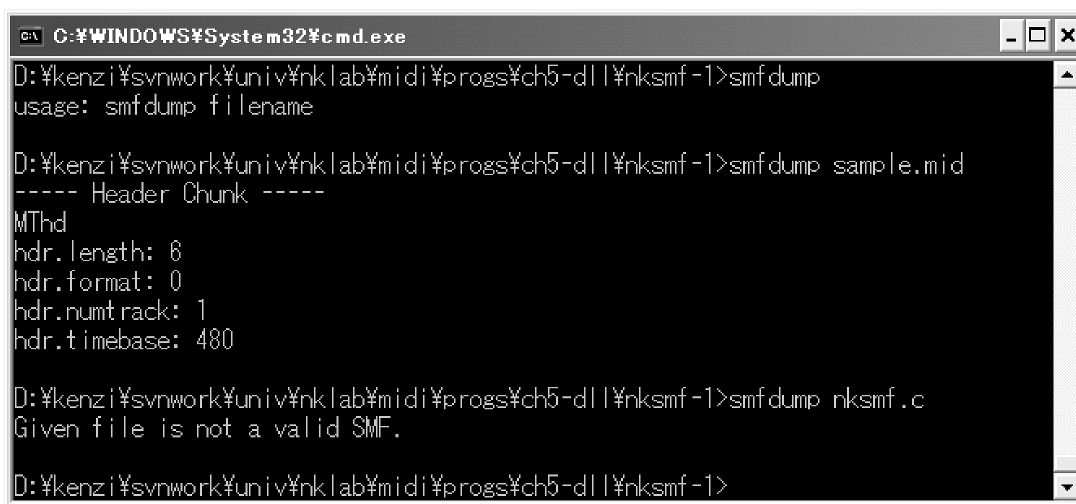


図 5.12　smfdump の実行例

5.3.4　make の利用

コンパイル時に毎回前節のようなコマンドを入力するのは面倒なので，一般的に make と呼ばれるユーティリティプログラムを使ってコンパイルすることにしよう。make は，通常 UNIX 環境では make というファイル名でインストールされているが，他にも gmake (GNU make) や nmake (Microsoft) などのファイル名を持つ make も存在する。Windows Plarform SDK には nmake が入っているため，この nmake を使用することになるが，基本的な記述方法は make によってあまり差はない。

C のソースファイルやヘッダファイルが入っているフォルダに，以下のリスト 5.14 の内容を持つファイルを makefile という名前で作成する[15]。なお，2 行目，4 行目，6 行目の先頭の空白は必ずタブでなければならない。

リスト 5.14 makefile

```
smfdump.exe : smfdump.c nksmf.lib nksmf.h
	cl smfdump.c nksmf.lib

nksmf.lib : nksmf.obj nksmf.def
	link nksmf.obj /def:nksmf.def /dll

nksmf.obj : nksmf.c nksmf.h nksmf-lib.h
	cl /c nksmf.c
```

この makefile を置いておくと，単に `nmake` と入力するだけで必要なコンパイルが自動的に行われる。

この makefile における 1 行目，3 行目，5 行目を依存記述行と呼び，':' (コロン) の左をターゲット，':' の右を依存ファイルと呼ぶ。また，2 行目，4 行目，6 行目をコマンド行と呼ぶ。make は，このターゲットと依存ファイルのタイムスタンプ（各ファイルの最終更新時間）を比較し，もしターゲットよりも依存ファイルの方が新しい場合，コマンド行を実行する。通常，コマンド行には依存ファイルからターゲットを生成する（コンパイルする）コマンドを記述するため，依存ファイル（例えば C 言語のソースファイルやヘッダファイル）を書き換えて更新した場合，make コマンド（ここでは nmake）を実行するだけで必要なファイルのみがコンパイルされる仕組みになっている。

リスト 5.14 の makefile では，（make コマンド実行時に引数で指定しない場合）make はもっとも先頭のターゲット smfdump.exe を更新しようとする。そして make が依存ファイルを調べると，nksmf.lib もターゲットとして指定されていることが分かるため，先にターゲット nksmf.lib を更新しようとする。しかし，さらに nksmf.lib の依存ファイルには，ターゲットとしても指定されている nksmf.obj があるため，nksmf.obj を先に更新しようとする。ターゲット nksmf.obj の依存ファイルにはそれ以上ターゲットに指定されているものがないので，もし nksmf.c，nksmf.h，nksmf-lib.h のいずれかが更新されていれば `cl /c nksmf.c` コマンドが実行されて nksmf.obj が生成される。同様に nksmf.lib が最新の状況に更新され，次いで smfdump.exe が更新される。この動作によって，smfdump.exe が依存している全てのファイルが（更新されたファイルのみ）コンパイルされ，最新の状況に保たれるようになっている。

[15] 全て小文字で makefile である。あるいは，最初が大文字で Makefile としてもよい。

make について詳しくは Web サイトや書籍などを参照して欲しい。コマンドラインベースのコンパイルでは非常に重宝するユーティリティソフトウェアである。

5.3.5　トラックの処理

次にトラックの扱いを考える。対応フォーマットを 1 にしているため，マルチトラックの処理が必要であり，複数のトラックからデータを読み込み可能なインタフェースを設計しなければならない。また，トラック数を事前に（プログラムのコンパイル時に）決定することはできないから，トラック数に依存しないように動的なプログラムを作る必要がある。

さて，5.3.2 節で述べたように，このライブラリではファイル内容をメモリに一度に読み込んで処理せず，随時ファイルから読み出すように設計することにしている。従って，各トラックにおいて，トラックイベントの読み込みは「次のイベントを読み込む」という関数を作るのが妥当であろう。トラック ID のようなものを指定するように設計すると，次のような関数が考えられる（イベントの受け取りはここでは省略する）。また，最初のイベントに戻る関数も付け加えておこう（rewind は巻き戻すという意味である）。

- smfGetNextEvent(トラック ID)
- smfRewindTrack(トラック ID)

次のイベントが分かるということは，すなわち直前に読み込んだイベントを知っている，具体的には直前に読み込んだイベントの位置（ファイル内での位置）を各トラックごとに記憶している必要があるということである。また，rewind するためにも，そのトラックの開始位置も記憶しておくべきであろう。すなわち，ライブラリは各トラックごとに次の情報を保持しなければならない（ここでは直前のイベントではなく次のイベントとしている）。

- トラックの開始位置
- 次に読み込むイベントの位置

このようなデータ構造は構造体で保持するのがいいだろう。トラックチャンクはチャンクタイプとデータ長を持つので，併せてそれらも保持できる次のような構造体を定義することにする。

```
/* トラックチャンク */
typedef struct smftrack {
  char chunktype[4];         /* チャンクタイプ: "MTrk" */
  long unsigned int length;  /* データ長 */
  unsigned int start_pos;    /* 先頭イベントの位置 */
```

```
    unsigned int next_pos;      /* 次のイベントの位置 */
  } SMFTRACK;
```

5.3.6　トラックイベントの読み込み

付録 A

課題の解答例

A.1 第 2 章

課題 1

リスト A.1 ドミソの和音

```
#include <windows.h>
#include <mmsystem.h>

int main(void) {
  HMIDIOUT hMidiOut;

  midiOutOpen(&hMidiOut, MIDI_MAPPER, 0, 0, 0);

  midiOutShortMsg(hMidiOut, 0x007f3c90); /* ノートオン: C4 (60=3C) */
  midiOutShortMsg(hMidiOut, 0x007f4090); /* ノートオン: E4 (64=40) */
  midiOutShortMsg(hMidiOut, 0x007f4390); /* ノートオン: G4 (67=43) */
  Sleep(1000);
  midiOutShortMsg(hMidiOut, 0x007f3c80); /* ノートオフ: C4 (60=3C) */
  midiOutShortMsg(hMidiOut, 0x007f4080); /* ノートオフ: E4 (64=40) */
  midiOutShortMsg(hMidiOut, 0x007f4380); /* ノートオフ: G4 (67=43) */

  midiOutReset(hMidiOut);
  midiOutClose(hMidiOut);

  return 0;
}
```

A.2 第 3 章

課題 3

リスト A.2 C メジャースケール

```
#include <windows.h>
#include <mmsystem.h>

int main(void) {
  HMIDIOUT hMidiOut;
  unsigned int base;
  int i;

  int notes[] = {0x3c, 0x3e, 0x40, 0x41, 0x43, 0x45, 0x47, 0x48}; /* ノート番
号 */
  int sleeptime = 1000; /* 各サスペンド時間 */
  int velocity = 0x40; /* ノートオンベロシティ */

  midiOutOpen(&hMidiOut, MIDI_MAPPER, 0, 0, 0);

  velocity = velocity << 16; /* 2 バイト分ずらして 3 バイト目にセット */
  for (i=0; i<8; i++) {
    base = (notes[i] << 8) + 0x90; /* 1 バイト目と 2 バイト目をセット */
    midiOutShortMsg(hMidiOut, velocity + base); /* ノートオン */
    Sleep(sleeptime);
    midiOutShortMsg(hMidiOut, base); /* ノートオフ（ベロシティ: 0）*/
  }

  midiOutReset(hMidiOut);
  midiOutClose(hMidiOut);

  return 0;
}
```

課題 4

リスト A.3 複数チャンネルの同時発音（メロディとリズム）

```
#include <windows.h>
#include <mmsystem.h>

int main(void) {
  HMIDIOUT hMidiOut;
  unsigned int QUAVER = 500;
```

```
  midiOutOpen(&hMidiOut, MIDI_MAPPER, 0, 0, 0);

  /* プログラムチェンジ: Acoustic Guitar (steel) */
  midiOutShortMsg(hMidiOut, 0x000019c0);

  /* 1 小節目 */
  midiOutShortMsg(hMidiOut, 0x00403c90); /* ノートオン: C4 */
  midiOutShortMsg(hMidiOut, 0x00402999); /* Closed Hi-Hat */
  Sleep(QUAVER);

  midiOutShortMsg(hMidiOut, 0x00402999); /* Closed Hi-Hat */
  Sleep(QUAVER);
  midiOutShortMsg(hMidiOut, 0x00003c90); /* ノートオフ: C4 */

  midiOutShortMsg(hMidiOut, 0x00404090); /* ノートオン: E4 */
  midiOutShortMsg(hMidiOut, 0x00402599); /* Acoustic Snare */
  Sleep(QUAVER);

  midiOutShortMsg(hMidiOut, 0x00402999); /* Closed Hi-Hat */
  Sleep(QUAVER);
  midiOutShortMsg(hMidiOut, 0x00004090); /* ノートオフ: E4 */

  /* 2 小節目 */
  midiOutShortMsg(hMidiOut, 0x00404390); /* ノートオン: G4 */
  midiOutShortMsg(hMidiOut, 0x00402999); /* Closed Hi-Hat */
  Sleep(QUAVER);

  midiOutShortMsg(hMidiOut, 0x00402999); /* Closed Hi-Hat */
  Sleep(QUAVER);
  midiOutShortMsg(hMidiOut, 0x00004390); /* ノートオフ: G4 */

  midiOutShortMsg(hMidiOut, 0x00404890); /* ノートオン: C5 */
  midiOutShortMsg(hMidiOut, 0x00402599); /* Acoustic Snare */
  Sleep(QUAVER);

  midiOutShortMsg(hMidiOut, 0x00402999); /* Closed Hi-Hat */
  Sleep(QUAVER);
  midiOutShortMsg(hMidiOut, 0x00004890); /* ノートオフ: C5 */

  midiOutReset(hMidiOut);
  midiOutClose(hMidiOut);

  return 0;
}
```

A.3　第 4 章

課題 5

リスト A.4 入力デバイス一覧の表示

```
#include <stdio.h>
#include <windows.h>
#include <mmsystem.h>

int main(void) {
  MIDIINCAPS inCaps;
  MMRESULT res;
  UINT num, devid;

  /* デバイス数を取得 */
  num = midiInGetNumDevs();
  printf("Number of input devices: %d\n", num);
  /* 各デバイス ID に対して for ループ */
  for (devid=0; devid<num; devid++) {
    /* デバイスの情報を inCaps に格納 */
    res = midiInGetDevCaps(devid, &inCaps, sizeof(inCaps));
    /* midiInGetDevCaps の戻り値が成功でない（=失敗）なら次のループへ */
    if (res != MMSYSERR_NOERROR) { continue; }
    /* デバイス ID とそのデバイス名を表示 */
    printf("ID=%d: %s\n", devid, inCaps.szPname);
  }

  return 0;
}
```

課題 7

リスト A.5 特定のノート番号を持つノートオンメッセージによるプログラムの終了

```
#include <stdio.h>
#include <windows.h>
#include <mmsystem.h>

/* プログラム終了を判定するためのフラグ */
int end_of_program;

void CALLBACK MidiInProc(HMIDIIN hMidiIn, UINT wMsg, DWORD dwInstance,
                          DWORD dwParam1, DWORD dwParam2) {
  switch (wMsg) {
    case MIM_OPEN:
      printf("MIDI device is opened.\n");
      break;
    case MIM_CLOSE:
      printf("MIDI device is closed.\n");
      break;
    case MIM_DATA:
      /* ノート番号が 0x3C のノートオフメッセージか， */
      /* ノート番号が 0x3C でベロシティが 0 のノートオンメッセージの場合， */
      if ( (dwParam1 & 0x0000fff0) == 0x00003C80 ||
           (dwParam1 & 0x00fffff0) == 0x00003C90    ) {
        /* プログラム終了用のフラグをオンにセットする */
        end_of_program = 1;
        printf("end_of_program flag is on.\n");
      }
    case MIM_LONGDATA:
    case MIM_ERROR:
    case MIM_LONGERROR:
    case MIM_MOREDATA:
    default:
      printf("MidiInProc: wMsg=%08X, p1=%08X, p2=%08X\n",
             wMsg, dwParam1, dwParam2);
      break;
  }
}

int main(int argc, char **argv) {
  HMIDIIN hMidiIn;
  MMRESULT res;
  UINT devid;
  char errmsg[MAXERRORLENGTH];

  if (argc > 1) {
    sscanf(argv[1], "%u", &devid);
  } else {
```

```
    devid = 0u;
  }

  res = midiInOpen(&hMidiIn, devid,
                   (DWORD_PTR)MidiInProc, 0, CALLBACK_FUNCTION);
  if (res != MMSYSERR_NOERROR) {
    printf("Cannot open MIDI input device (ID=%u): ", devid);
    midiInGetErrorText(res, errmsg, sizeof(errmsg));
    printf("%s\n", errmsg);
    return 1;
  }
  printf("Succeeded to open a MIDI input device (ID=%u).\n", devid);

  midiInStart(hMidiIn);

  end_of_program = 0;
  /* プログラム終了用のフラグがオンになるまで待つ */
  while ( !end_of_program ) {
    Sleep(10);
  }

  midiInStop(hMidiIn);
  midiInReset(hMidiIn);

  midiInClose(hMidiIn);
  printf("Closed a MIDI input device.\n");

  return 0;
}
```

課題 8

リスト A.6 MIDI メッセージの中継

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <mmsystem.h>

/* SysEx 受信用バッファのサイズ */
#define BUFSIZE 256

HMIDIOUT hMidiOut;   /* MIDI 出力デバイスのハンドル */
int end_of_program;  /* プログラム終了を判定するためのフラグ */

void CALLBACK MidiInProc(HMIDIIN hMidiIn, UINT wMsg, DWORD dwInstance,
                          DWORD dwParam1, DWORD dwParam2) {
  MIDIHDR *hdr;    /* SysEx メッセージ受信用バッファ */
  MIDIHDR outhdr;  /* SysEx メッセージ送信用バッファ */
  int i;

  switch (wMsg) {
    case MIM_OPEN:
      printf("MIDI device is opened.\n");
      break;
    case MIM_CLOSE:
      printf("MIDI device is closed.\n");
      break;
    case MIM_DATA:
      /* 受信データをそのまま hMidiOut に送信 */
      midiOutShortMsg(hMidiOut, dwParam1);

      if ( (dwParam1 & 0x0000fff0) == 0x00003C80 ||
           (dwParam1 & 0x00fffff0) == 0x00003C90    ) {
        end_of_program = 1;
        printf("end_of_program flag is on.\n");
      }

      /* アクティブセンシング以外であれば表示 */
      if ( (dwParam1 & 0x000000ff) != 0xfe ) {
        printf("MIM_DATA: wMsg=%08X, p1=%08X, p2=%08X\n",
               wMsg, dwParam1, dwParam2);
      }

      break;
    case MIM_LONGDATA:
      /* 受信した SysEx メッセージを表示 */
      printf("MIM_LONGDATA: ");
      hdr = (MIDIHDR*)dwParam1;
```

```
      printf("[%d bytes] ", hdr->dwBytesRecorded);
      for (i=0; i<hdr->dwBytesRecorded; i++) {
        printf("%02X ", (unsigned char)((hdr->lpData)[i]));
      }
      printf("\n");

      /* 出力用に MIDIHDR 構造体を初期化 */
      ZeroMemory(&outhdr, sizeof(MIDIHDR));
      outhdr.lpData = (LPSTR)(hdr->lpData);
      outhdr.dwBufferLength = hdr->dwBytesRecorded;
      outhdr.dwFlags = 0;

      /* MIDIHDR 構造体を準備して送信 */
      midiOutPrepareHeader(hMidiOut, &outhdr, sizeof(MIDIHDR));
      midiOutLongMsg(hMidiOut, &outhdr, sizeof(MIDIHDR));

      /* 再度入力用バッファを準備して登録 */
      midiInPrepareHeader(hMidiIn, hdr, sizeof(MIDIHDR));
      midiInAddBuffer(hMidiIn, hdr, sizeof(MIDIHDR));

      /* 出力用の MIDIHDR 構造体の準備を解除 */
      midiOutUnprepareHeader(hMidiOut, &outhdr, sizeof(MIDIHDR));

      break;
    case MIM_ERROR:
    case MIM_LONGERROR:
    case MIM_MOREDATA:
    default:
      printf("MidiInProc: wMsg=%08X, p1=%08X, p2=%08X\n",
             wMsg, dwParam1, dwParam2);
      break;
  }
}

int main(int argc, char **argv) {
  HMIDIIN hMidiIn;
  MMRESULT res;
  MIDIHDR header;
  UINT devid;
  char errmsg[MAXERRORLENGTH];

  if (argc > 1) {
    sscanf(argv[1], "%u", &devid);
  } else {
    devid = 0u;
  }

  /* MIDI 出力デバイスを開く */
  res = midiOutOpen(&hMidiOut, MIDI_MAPPER, (DWORD_PTR)NULL, 0, CALLBACK_NULL);
  if (res != MMSYSERR_NOERROR) {
```

```
    printf("Cannot open MIDI output device.\n");
    return 1;
  }
  printf("Succeeded to open a MIDI output device.\n");

  ZeroMemory(&header, sizeof(MIDIHDR));
  header.lpData = (char*)malloc(BUFSIZE);
  header.dwBufferLength = BUFSIZE;
  header.dwFlags = 0;

  res = midiInOpen(&hMidiIn, devid, (DWORD_PTR)MidiInProc, 0, CALLBACK_FUNCTION);
  if (res != MMSYSERR_NOERROR) {
    printf("Cannot open MIDI input device (ID=0x%X): ", devid);
    midiInGetErrorText(res, errmsg, sizeof(errmsg));
    printf("%s\n", errmsg);
    return 1;
  }
  printf("Succeeded to open a MIDI input device (ID=%u).\n", devid);

  midiInPrepareHeader(hMidiIn, &header, sizeof(MIDIHDR));
  midiInAddBuffer(hMidiIn, &header, sizeof(MIDIHDR));

  midiInStart(hMidiIn);

  end_of_program = 0;
  while ( !end_of_program ) {
    Sleep(10);
  }

  midiInStop(hMidiIn);
  midiInReset(hMidiIn);

  midiInUnprepareHeader(hMidiIn, &header, sizeof(MIDIHDR));

  midiInClose(hMidiIn);
  printf("Closed a MIDI input device.\n");

  free(header.lpData);

  /* MIDI 出力デバイスをリセットして閉じる */
  midiOutReset(hMidiOut);
  midiOutClose(hMidiOut);
  printf("Closed a MIDI output device.\n");

  return 0;
}
```

付録 B

参考文献・Web サイト

B.1 MIDI 関連の情報源

B.1.1 書籍

MIDI 関連の書籍は数多く出版されているので，ここではそのうち筆者が参考にしたものや，本テキストに沿った内容の書籍を挙げる。

- **MIDI 検定 3 級公式ガイドブック，日本シンセサイザー・プログラマー協会 (JSPA)，ミュージックトレード，2000**
- **MIDI 検定 2 級公式ガイドブック，日本シンセサイザー・プログラマー協会 (JSPA)，ミュージックトレード，2000**
- **SMF リファレンス・ブック，新井 純，エディロール，1996**

B.1.2 Web サイト

- **MIDI Manufacturers Association (MMA)，*http://www.midi.org/***
  MMA は MIDI の標準化団体であり，MIDI の仕様書を購入できる。Web 上で参照できる情報も少しあり，http://www.midi.org/about-midi/specshome.shtml の Online Information にリンクがある。
- **社団法人音楽電子事業協会 (AMEI)，*http://www.amei.or.jp/***
  AMEI は MIDI の業界団体であり，MIDI 検定を主催している団体でもある。Web では GM2 や GM Lite の仕様書が公開されている。
- **B# - MIDI，*http://www.b-sharp.com/midi/***
  GM や SMF を含む MIDI 1.0 の詳しい解説や，MIDI に関する TIPS などがある。

- **The Sonic Spot - Standard MIDI Files,**
  http://www.sonicspot.com/guide/midifiles.html
  SMF のファイルフォーマットについて，適切な文章量で読みやすく記述されている。

B.2　その他

B.2.1　書籍

- **入門 MAKE&RCS —— make & rcs による効率的プログラミング技法，伊藤和人，秀和システム，1998**

B.2.2　ソフトウェア

- **BCC Developer,** ***http://www.hi-ho.ne.jp/jun_miura/bccdev.htm***
  Borland C++ Compiler (BCC) 5.5 に対応した簡易開発環境。
- **Borland C++ Compiler (BCC),**
  http://www.borland.co.jp/cppbuilder/freecompiler/
  無償で配布されている，Borland C++ のコンパイラやリンカを含むコマンドラインベースの開発環境。同 Web サイトでは，Turbo Debugger 5.5 というデバッガも別途配布されている。設定ファイルの作成や環境変数の設定は setbcc というソフトウェアを利用すると簡便である。
- **JGREP,** ***http://www.hi-ho.ne.jp/jun_miura/jgrep.htm***
  Windows で動作する GUI ベースの grep で，正規表現による行単位の検索が可能なソフトウェア。
- **Microsoft Platform SDK,**
  http://www.microsoft.com/msdownload/platformsdk/sdkupdate/
  Windows アプリケーションを開発するための，ドキュメンテーション，サンプル，ヘッダファイル，ライブラリ，ツールを含む SDK（ソフトウェア開発キット）。
- **Microsoft Visual C++ Toolkit 2003,**
  http://msdn.microsoft.com/visualc/vctoolkit2003/
  無償で配布されている，Microsoft Visual C++ のコンパイラやリンカを含む開発キット。
- **Power Tab Editor,** ***http://www.power-tab.net/***
  ギターのタブ譜（.ptb ファイル）を閲覧，編集，MIDI 再生可能なソフトウェア。

- **setbcc，*http://www.power-tab.net/***
  Borland C++ Compiler の設定ファイルの作成や環境変数の設定を簡便に行えるソフトウェア。
- **Stirling，*http://www.vector.co.jp/soft/win95/util/se079072.html***
  高機能バイナリエディタ。
- **バイナリエディタ Bz，*http://www.zob.ne.jp/˜c.mos/soft/bz.html***
  バイナリエディタ。

索引

記号・数字

#define 20
#else 40
#endif 40
#if 40
#ifdef 40
#include 16, 81
-A（関数名接尾辞） 41
-L（定数サフィックス） 35
-U（定数サフィックス） 34
-W（関数名接尾辞） 41
.dll 81
.exp 81
.lib 81
.mid 3, 5, 61
.midi 5
.obj 12
.ptb 3, 106
.smf 5
_MBCS マクロ 42
_tprintf() 42
_UNICODE マクロ 42

A

AMEI 105
ANSI 41
API 17
argc 44
argv 44
ASCII 41

B

beats per minute 74, 79
BOOL 18
boolean 84
Borland C++ 10
bpm 74, 79
BYTE 18

C

call-by-name 35
call-by-reference 35
call-by-value 35
CALLBACK 47
CALLBACK_FUNCTION 48
CHAR 41
cl.exe 9, 13

D

D-（型接頭辞） 18
DLL 13, 80
dwMidiMessage 51
DWORD 18
DWORD_PTR 34
dwTimestamp 51

E

End of Track 73, 79
EOF 66
EOX 53
errno 87
errno.h 87
exit() 45

F

fclose() 66
feof() 67, 87
ferror() 67, 87
fgetc() 66
FILE 65
FLOAT 18
fopen() 66, 87
frames per second 63, 74
fread() 67, 69
free() 59

G

GM 6
GM Lite 105
GM2 105
gmake 92
GM 音源 6, 31, 33
GM 規格 31
GM システムオン 32
grep 18
GS 音源 6, 16

H

has（プレフィクス） 84
HeapAlloc() 59
HeapFree() 59
HMIDIOUT 23

I

I18N 42
INT 18
is（プレフィクス） 84

L

L-（型接頭辞） 18
l-value 59
l10n 42
LIB 13
link.exe 13
long int 69
LP-（型接頭辞） 18, 19
lpMidiHdr 51
LPMIDIOUTCAPS 14, 38
LPSTR 45, 54
LSB 30

M

main() 44, 45
make 92
Makefile 93
makefile 93
malloc() 59
MAXERRORLENGTH 45
MBCS 41–42
Microsoft GS Wavetable SW Synth 16
MIDI_IO_STATUS 51
MIDI_MAPPER 16, 23
MIDIHDR 51, 54
midiInAddBuffer() 54
MIDIINCAPS 39–42
MIDIINCAPS2 41
MIDIINCAPSA 41–42
MIDIINCAPSW 41–42
midiInClose() 43
midiInGetDevCaps() 38
midiInGetErrorText 44
midiInGetNumDevs() 38
midiInOpen() 43–44, 48
midiInPrepareHeader() 54
MidiInProc 47
midiInReset() 46
midiInStart() 46, 51
midiInStop() 46
midiInUnprepareHeader() 54
MIDIOUTCAPS 15, 17–19
midiOutClose() 10, 24
midiOutGetDevCaps() 14–15
midiOutGetErrorText() 45
midiOutGetNumDevs() 14–15
midiOutLongMsg() 53
midiOutOpen() 10, 23
midiOutReset() 10, 24
midiOutShortMsg() 10, 11, 25–27, 53
MIDI イベント 20, 64, 73, 76
MIDI チャンネルプレフィクス 79
MIDI メッセージ 14, 20, 28
MIM_CLOSE 51
MIM_DATA 51, 53
MIM_ERROR 51
MIM_LONGDATA 51, 53
MIM_LONGERROR 51
MIM_MOREDATA 51
MIM_OPEN 50
MMA 105
MMRESULT 14, 17–18, 38
MMSYSERR_NOERROR 17–20
mmsystem.h 17, 35, 81
MSB 29, 30
MTC クォーターフレーム 53
MThd 62
MTrk 64

N

nmake 92
NP-（型接頭辞） 19
NULL 34
NULL 文字 71

P

P-（型接頭辞） 18, 19
PATH 13
PCM 1, 5
PCM 音源 5
Platform SDK 8
printf() 47

R

r-value 59
return 45

S

SBCS 41–42
Set Tempo 79
setenv.bat 9
short int 69
sizeof() 70
Sleep() 10, 26
SMF 5, 61
SMPTE オフセット 79
SMPTE フレーム 63
sscanf() 44
stdio.h 16
strerror() 87
switch 52
SysEx 51–55
SysEx イベント 73, 77
SysEx メッセージ 77
szPname 41

T

TCHAR 42
tchar.h 42
ticks per beat 63, 74

U

U-（型接頭辞） 18
UINT 17, 18
Unicode 41–42
UNICODE マクロ 41
UTF-8 41

V

variable-length value 74, 75
Visual C++ 7, 9
Visual C++ Toolkit 2003 8

W

W-（型接頭辞） 18
WCHAR 41
wchar_t 41–42
Wide Character 41
Win32 19, 40
windef.h 18
Windows API 7, 17
windows.h 17, 18
winmm.dll 12
winmm.lib 12–13, 80
WORD 18

X

XG 音源 6

あ

アクティブセンシング 53
値渡し 35

い

依存記述行 93
依存ファイル 93
イベント 4, 49
イベントタイプ 22, 73
イベントドリブン 49

う

右辺値 59

え

エクスポート 80
エンコーディング 41
エンディアン 27, 65, 69
エンドオブエクスクルーシブ 53
エンドオブトラック 79

お

オールサウンドオフ 30
オールノートオフ 30
オフベロシティ 26

オムニオフ 30
オムニオン 30
音源 5

か

歌詞 79
型キャスト 34
楽器名 79
可変長データ表現 74, 75
仮引数 35
環境変数 9, 13

き

キューポイント 79

く

クロック 63, 74

こ

コールバック 47
コールバック関数 47, 50
国際化プログラミング 42
コマンド行 93
コマンドライン引数 44, 45, 86
コンティニュー 53
コントロールチェンジ 30
コンパイル 12

さ

最下位バイト 30
最上位バイト 30
最上位ビット 29
左辺値 59
参照渡し 35

し

シーケンサ固有メタイベント 79
シーケンス 63
シーケンス番号 79
シーケンス名/トラック名 79
ジェネリックテキストマッピング 42
システムエクスクルーシブイベント 73
システムエクスクルーシブメッセージ 52, 77
システムコモンメッセージ 53, 77
システムメッセージ 28, 29, 52, 77
システムリアルタイムメッセージ 53, 77
システムリセット 53
実引数 35
終了コード 45
出力デバイス 14

す

スクリプト 45
スタート 53
ステータスバイト 21, 28–29
ストップ 53
ストリーム 65

せ

セットテンポ 79

そ

ソングセレクト 53
ソングポジションポインタ 53

た

ターゲット 93
タイミングクロック 53
タイムスタンプ 93
タイムベース 63, 74
ダミーバイト 30
ダンプ 67

ち

チャンク 62
チャンクタイプ 62, 64
チャンネル 29
チャンネル番号 21, 29–30
チャンネルプレッシャー 30
チャンネルメッセージ 25, 28
チューンリクエスト 53
調 79
著作権表示 79

て

定数サフィックス 34, 35
ティック 63
データ長 62
データバイト 21, 28
テキスト（メタイベント） 79
テキストファイル 61
テキストモード 66
デルタタイム 63, 73, 74
テンポ 64, 74, 79

と

トラック 63, 94
トラックイベント 64, 73, 94
トラック数 63
トラックチャンク 62, 64

な

名前渡し 35

ね

音色マッピング 31

の

ノートオフ 4, 20–22, 30, 33
ノートオン 4, 11, 20–22, 30, 33
ノート番号 11, 21, 33

は

バイトオーダー 26, 27, 64
バイナリエディタ 61
バイナリファイル 61
バイナリモード 66
バッチプログラム 45
バンクセレクト 31

ひ

ビッグエンディアン 26–28, 64, 69
ピッチベンドチェンジ 30
拍子 79

ふ

ブーリアン 84
フォーマットタイプ 63
プリプロセッサディレクティブ 40
プログラムチェンジ 5, 30, 33
プログラム番号 31
プロトタイプ宣言 34

へ

ヘッダチャンク 62, 69
ヘッダファイル 81
ベロシティ 21

ほ

ボイスメッセージ 29
ポリフォニックキープレッシャー 30
ポリモードオン 30

ま

マーカー 79
マルチトラック 94

め

メタイベント 73, 78
メタデータ 78
メロディ楽器 33

も

モードメッセージ 29
モノモードオン 30

ら

ランニングステータス 51, 77

り

リズム楽器 33
リセットオールコントローラ 30
リトルエンディアン 27–28, 64, 69
リリースタイム 22
リンク 12

ろ

ローカルコントローラ 30

わ

ワイド文字 41