フルスクラッチで作る!
UEFI ベアメタルプログラミング

大神祐真　著

2017-08-11 版　　へにゃぺんて 発行

はじめに

本書をお手にとっていただき、ありがとうございます。

本書では「UEFI」で「ベアメタルプログラミング」を行います。

ベアメタルプログラミングは、OS 無しでハードウェアを直接制御するプログラムを書くプログラミング方法です。

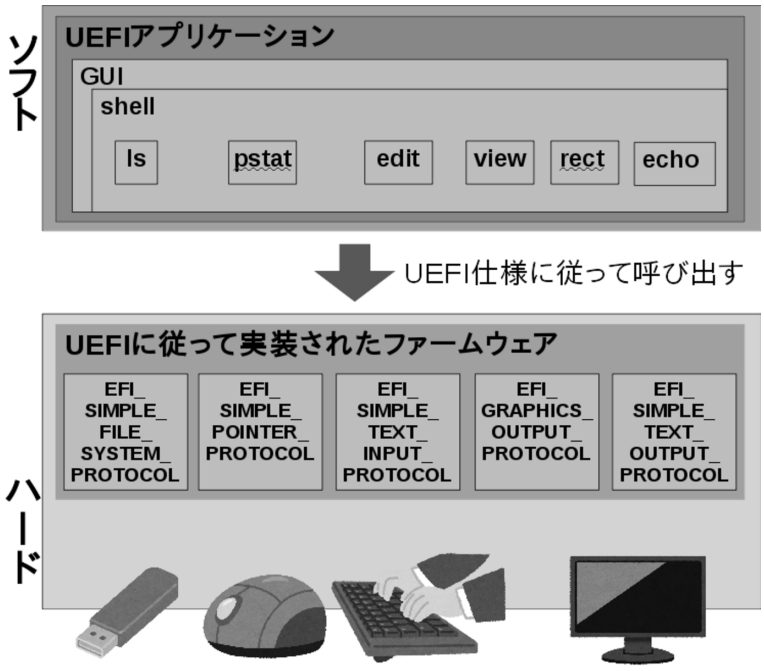
「ハードウェアを直接制御」といっても、信号線の一本一本を制御するようなプログラムを書くわけではありません。たいていのハードウェアには「ファームウェア」というソフトウェアが ROM に書き込まれています。そのため、ファームウェアと適切にやり取りすることでハードウェアを制御できます。

PC の代表的なファームウェアは「BIOS」です。近年は、実装が規格化された「UEFI」に置き換わっています。そのため、PC では、電源を入れると BIOS あるいは UEFI のファームウェアが動作し、起動ディスクから OS をブートします。

本書では近年の流れに沿って、UEFI でベアメタルプログラミングを行います。そのために、本書では主に「UEFI のファームウェアの機能をどうやって呼び出すのか」について説明していきます。

poiOS について

本書では UEFI の機能を呼び出すことで、OS っぽいものを作ってみます。本書では"poiOS"と名づけており、構造は図 .1 の通りです。



▲図 1　poiOS の構造

本書に関する情報の公開場所

本書の PDF データ版、サンプルのソースコードやコンパイル済バイナリ等、本書に関する情報は以下のページ (あるいはそこから辿れるリンク先) にまとめています。

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

また、上記のページにも書いていますが、サンプルコードは以下の GitHub のリポジトリで公開しています。サンプルコードは各節毎にディレクトリを分けており、各節の冒頭で該当するサンプルコードのディレクトリ名を説明します。文中ではコードの一部分しか引用しないこともありますので、コード全体を見たいときは下記 URL 先のものを参照してください。

- `https://github.com/cupnes/c92_uefi_bare_metal_programming_samples`

目次

第 1 章

Hello UEFI!

この章では、UEFI でベアメタルプログラミングを行う流れとして、環境構築、"Hello UEFI!"を画面出力するプログラムの作成・実行までを説明します。

1.1 ベアメタルプログラミングの流れ

UEFI ファームウェアがロード・実行するプログラムを「UEFI アプリケーション」と呼びます。本書のベアメタルプログラミングでは UEFI アプリケーションを作成し、PC へ実行させます。流れとしては以下の通りです。

1. UEFI の仕様に沿ったプログラムを書く
2. UEFI ファームウェアが実行できる形式へクロスコンパイル
3. UEFI ファームウェアが見つけられるように起動ディスクを作成

次から、この流れに沿って、"Hello UEFI!"と画面表示するプログラムを作ってみます。

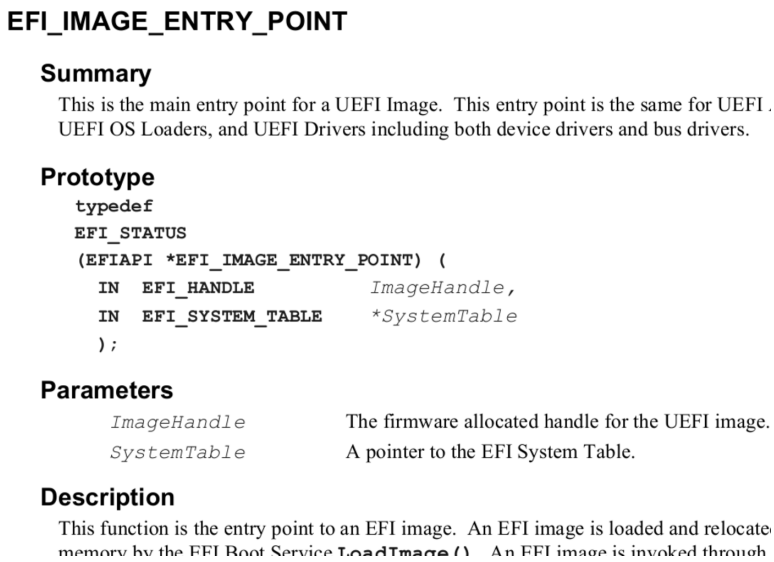
1.2 UEFI の仕様に沿ったプログラムを書く

UEFI 仕様書の読み方

UEFI の仕様書は `http://www.uefi.org/specifications` で公開されています ("UEFI Specification"の箇所から PDF をダウンロードできます)。なお、本書では UEFI バージョン 2.3.1 の仕様書を参照します[*1]。ただし、本書で扱うような範囲は、実機の UEFI バージョン・参照する仕様書のバージョン共に、多少バージョンが前後しても問題は無いと思います。

[*1] 私の動作確認している実機環境 (Lenovo ThinkPad E450) の都合上です。

UEFI の仕様書には C 言語のコード片なども含まれており、UEFI ファームウェア上で動作するプログラムは基本的に C 言語で作ります。例えば、プログラムの実行開始場所であるエントリポイント[*2]は、仕様書の"4.1 UEFI Image Entry Point(P.75)"に図 1.1 の記載があります。



EFI_IMAGE_ENTRY_POINT

Summary

This is the main entry point for a UEFI Image. This entry point is the same for UEFI
UEFI OS Loaders, and UEFI Drivers including both device drivers and bus drivers.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IMAGE_ENTRY_POINT) (
  IN  EFI_HANDLE          ImageHandle,
  IN  EFI_SYSTEM_TABLE    *SystemTable
  );
```

Parameters

ImageHandle	The firmware allocated handle for the UEFI image.
SystemTable	A pointer to the EFI System Table.

Description

This function is the entry point to an EFI image. An EFI image is loaded and relocate
memory by the EFI Boot Service LoadImage(). An EFI image is invoked through

▲図 1.1　仕様書のエントリポイントの説明箇所

図 1.1 を見ると、引数と戻り値が決められていることが分かります。関数名はコンパイル時のオプションで指定するため何でも良いです。引数や戻り値は独自の型で定義されていますが、これらの型も仕様書内で定義されています。仕様書内を"EFI_STATUS"で検索してみると、"2.3.1 Data Types(P.23)"がヒットします (図 1.2)。

[*2] C 言語の一般的なプログラムにおける main 関数に相当するものです。

UINTN	Unsigned value of native width. (4 bytes on supported 32-bit processor instructions, 8 bytes on supported 64-bit processor instructions)
INT8	1-byte signed value.
UINT8	1-byte unsigned value.
INT16	2-byte signed value.
UINT16	2-byte unsigned value.
INT32	4-byte signed value.
UINT32	4-byte unsigned value.
INT64	8-byte signed value.
UINT64	8-byte unsigned value.
CHAR8	1-byte character. Unless otherwise specified, all 1-byte or ASCII characters and strings are stored in 8-bit ASCII encoding format, using the ISO-Latin-1 character set.
CHAR16	2-byte Character. Unless otherwise specified all characters and strings are stored in the UCS-2 encoding format as defined by Unicode 2.1 and ISO/IEC 10646 standards.
VOID	Undeclared type.
EFI_GUID	128-bit buffer containing a unique identifier value. Unless otherwise specified, aligned on a 64-bit boundary.
EFI_STATUS	Status code. Type UINTN.

▲**図** 1.2　"2.3.1 Data Types"**の**"Table 6. Common UEFI Data Types"(**抜粋**)

図 1.2 には仕様書内で良く使われる型の定義が書かれています。ここを見ると、"EFI_STATUS"はステータスコードを示し、"UINTN"という型であること、また"UINTN"は 32 ビット CPU では unsigned の 4 バイト (unsigned int)、64 ビット CPU では unsigned の 8 バイト (unsigned long long) であることが分かります。

なお、"IN"・"OUT"・"OPTIONAL"・"EFIAPI"等は引数や関数の説明の為の記載です。EDK2 や gnu-efi といった既存の開発環境やツールチェイン[*3]では空文字列として定義されています。

"EFI_SYSTEM_TABLE"は、"2.3.1 Data Types"の"Table 6."に説明がありません。これは構造体を typedef したもので、構造体については章を設けて説明されています。"EFI_SYSTEM_TABLE"で仕様書内を検索してみると、"4.3 EFI System Table(P.78)"がヒットし、こちらも C 言語のコードで定義が記載されています (図 1.3)。

[*3] 本書はベアメタルプログラミングのため、どちらも使用しませんが、これらのソースコードは UEFI の機能を呼び出す実装方法の参考になります。

```
typedef struct {
  EFI_TABLE_HEADER                  Hdr;
  CHAR16                            *FirmwareVendor;
  UINT32                            FirmwareRevision;
  EFI_HANDLE                        ConsoleInHandle;
  EFI_SIMPLE_TEXT_INPUT_PROTOCOL    *ConIn;
  EFI_HANDLE                        ConsoleOutHandle;
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL   *ConOut;
```

▲図 1.3　EFI_SYSTEM_TABLE の定義 (一部)

図 1.3 には様々なメンバが並んでいます。UEFI の仕様を見ていくコツとして、「プロトコル」という概念があります。UEFI では「プロトコル」という単位で機能を分けており、"〜_PROTOCOL"という名前の構造体がいくつもあります。"〜_PROTOCOL"という名前の構造体は関数ポインタをメンバに持っており、その関数ポインタから UEFI ファームウェアの機能を呼び出せます。

図 1.3 では、"EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL"が画面へ文字を出力するためのプロトコルです。では、"EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL"がどのようなメンバを持っているのかというと、これまで同様、仕様書を検索してみます。すると、"11.4 Simple Text Output Protocol(P.424)"に構造体の定義が書かれていることが分かります (図 1.4)。

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL

Summary

This protocol is used to control text-based output devices.

GUID

```
#define EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL_GUID \
  {0x387477c2,0x69c7,0x11d2,0x8e,0x39,0x00,0xa0,\
  0xc9,0x69,0x72,0x3b}
```

Protocol Interface Structure

```
typedef struct _EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {
    EFI_TEXT_RESET                Reset;
    EFI_TEXT_STRING               OutputString;
    EFI_TEXT_TEST_STRING          TestString;
    EFI_TEXT_QUERY_MODE           QueryMode;
    EFI_TEXT_SET_MODE             SetMode;
    EFI_TEXT_SET_ATTRIBUTE        SetAttribute;
    EFI_TEXT_CLEAR_SCREEN         ClearScreen;
    EFI_TEXT_SET_CURSOR_POSITION  SetCursorPosition;
    EFI_TEXT_ENABLE_CURSOR        EnableCursor;
    SIMPLE_TEXT_OUTPUT_MODE       *Mode;
} EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL;
```

Parameters

Reset	Reset the *ConsoleOut* device. See **Reset()**.
OutputString	Displays the string on the device at the current cursor location. See **OutputString()**.

▲**図** 1.4　EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL **の定義**

図 1.4 の"Protocol Interface Structure"をみると、"OutputString"というメンバがあり、これを使うと画面へ文字を表示できそうだと分かります。図 1.4 の"Parameters"に構造体の各メンバの説明があり、"OutputString"の説明には定義の説明を行っているページへのリンクがあります (下線で示されている"OutputString()"の箇所)。OutputString は仕様書で図 1.5 の様に定義されています。

```
typedef
EFI_STATUS
(EFIAPI *EFI_TEXT_STRING) (
  IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL   *This,
  IN CHAR16                            *String
  );
```

▲**図** 1.5　OutputString **の定義**

引数の意味は以下の通りです。

IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL 自体のポインタ。

IN CHAR16 *String

画面へ表示する文字列のポインタ。UEFI では文字は Unicode(2 バイト)。

戻り値は EFI_STATUS(unsigned long long) 型で、実行結果のステータスが格納されています。成功時に 0、エラー/ワーニング時に 0 以外の値が格納されます。本書では「0 であるか否か」でしか使用しませんが、ステータス値の詳細は仕様書の"Appendix D Status Codes(P.1873)"と、各プロトコルの説明箇所の"Status Codes Returned"を参照してください。

なお、プロトコル内の関数はどれも「第 1 引数にプロトコル自体のポインタをとる」、「EFI_STATUS 型のステータスコードを返す」であるため、以降、この 2 つの説明は省略します。

また、UEFI のファームウェアによっては起動時に画面にメッセージが出力されます。ここでは画面をクリアしてから OutputString で画面へ文字列を表示することにします。

画面をクリアするには EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL の ClearScreen を使用します (仕様書"11.4 Simple Text Output Protocol(P.437)")。図 1.4 では見切れていますが ClearScreen も"Parameters"に説明があり、定義を説明しているページへのリンクがあります。ClearScreen の定義は図 1.6 の通りです。引数はプロトコル自体を指す"This"のみです。

```
typedef
EFI_STATUS
(EFIAPI *EFI_TEXT_CLEAR_SCREEN) (
  IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL    *This
  );
```

▲**図 1.6　ClearScreen の定義**

なお、以降の章ではプロトコル構造体とメンバである関数の定義は、サンプルコードをベースに説明します。

ソースコードを書く

エントリポイントの仕様も、画面へ文字列を出力するための関数の呼び出し方も分かったのでソースコードを書いてみます。サンプルソースコードのディレクトリは"sample1_1_hello_uefi"です。

なお、本書では開発環境は Debian GNU/Linux を想定しています。ただし、これは筆

者の作業環境がそうであるというだけで、クロスコンパイルと USB フラッシュメモリのフォーマットができれば何でもよいです (詳細は後述します)。もちろん、エディタも何でも良いです。

"Hello UEFI!"を出力するソースコードをリスト 1.1 に示します。

▼**リスト** 1.1　sample1_1_hello_uefi/main.c

```
 1: struct EFI_SYSTEM_TABLE {
 2:     char _buf[60];
 3:     struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {
 4:             unsigned long long _buf;
 5:             unsigned long long (*OutputString)(
 6:                     struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
 7:                     unsigned short *String);
 8:             unsigned long long _buf2[4];
 9:             unsigned long long (*ClearScreen)(
10:                     struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This);
11:     } *ConOut;
12: };
13:
14: void efi_main(void *ImageHandle __attribute__ ((unused)),
15:            struct EFI_SYSTEM_TABLE *SystemTable)
16: {
17:     SystemTable->ConOut->ClearScreen(SystemTable->ConOut);
18:     SystemTable->ConOut->OutputString(SystemTable->ConOut,
19:                                       L"Hello UEFI!\n");
20:     while (1);
21: }
```

リスト 1.1 について、EFI_SYSTEM_TABLE の構造体定義箇所では使用するメンバのみ定義し、その他はアドレスが合うようにバッファを入れています。

また、"efi_main"という名前で定義しているエントリポイントについて、OutputString 関数の前に、ClearScreen 関数で画面クリアを行っています。そして、末尾には、while の無限ループを設置しています (戻り先が無いため)。

なお、"EFI_STATUS"等の UEFI 固有の型は全て"unsigned long long"等へ書き下しています。これは単純に個人の趣味です。移植性が下がるため、UEFI の仕様書通りの書き方をしたい場合は適宜 typedef を追加してください。

以降は、リスト 1.1 が"main.c"というファイルで保存されているものとして説明します。

1.3 UEFI ファームウェアが実行できる形式へクロスコンパイル

ソースコードを用意できたので、コンパイルを行います。UEFI ファームウェアが認識する実行ファイル形式は PE32+ という形式です[*4]。Linux の実行ファイル形式は ELF なので、PE32+ へクロスコンパイルする必要があります。

クロスコンパイラを用意するために、gcc-mingw-w64-x86-64 パッケージをインストールします。

```
$ sudo apt install gcc-mingw-w64-x86-64
```

インストール後、以下のコマンドでクロスコンパイルできます。

```
$ x86_64-w64-mingw32-gcc -Wall -Wextra -e efi_main -nostdinc -nostdlib \
        -fno-builtin -Wl,--subsystem,10 -o main.efi main.c
```

"-e"オプションがエントリポイントを指定するオプションです。ここで"efi_main"を指定しているため、efi_main 関数がエントリポイントとして扱われます。また、"--subsystem,10"では、作成する実行ファイルが UEFI アプリケーションであることを指定しています。生成される"main.efi"が UEFI 向け PE32+ 実行ファイルです。

なお、PE32+ で UEFI アプリケーション向けにコンパイルが行えれば、他の方法でも構いません。例えば、Windows 版の x86_64-w64-mingw32-gcc は `https://sourceforge.net/projects/mingw-w64/` からダウンロードできるようです[*5]。

[*4] 主に Windows で使用される実行ファイル形式です。

[*5] 参考:"Windows(64 ビット環境) で vimproc をコンパイルしてみよう":http://qiita.com/akase244/items/ce5e2e18ad5883e98a77

Makefile で自動化

https://github.com/cupnes/c92_uefi_bare_metal_programming_samples で公開しているサンプルプログラムには Makefile も入っています。
サンプルのディレクトリへ移動し、makeを実行することでコンパイルできます。

```
$ cd c92_uefi_bare_metal_programming_samples/<各サンプルのディレクトリ>
$ make
```

コンパイルが完了すると、fs/EFI/BOOT/BOOTX64.EFI に efi 実行ファイルが生成されます。

1.4 UEFI ファームウェアが見つけられるように起動ディスクを作成

UEFI アプリケーションの実行ファイルを生成できたので、このファイルを UEFI ファームウェアが見つけられるようにストレージへ配置し、UEFI アプリケーションの起動ディスクを作成します。ストレージとしては USB フラッシュメモリが簡単です。本書では USB フラッシュメモリを想定して説明します。

UEFI は FAT ファイルシステムを認識できます。そのため、まずは USB フラッシュメモリを FAT32 あたりでフォーマットします。

単に FAT32 でフォーマットできれば何でもよいです。操作例としては以下の通りです(USB フラッシュメモリは/dev/sdb として認識されているものとします)。

```
$ sudo fdisk /dev/sdb
Welcome to fdisk (util-linux 2.25.2).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


Command (m for help): d    <= 既存のパーティションを削除
Selected partition 1
Partition 1 has been deleted.
```

```
Command (m for help): o

Created a new DOS disklabel with disk identifier 0xde746309.

Command (m for help): n
Partition type
   p   primary (0 primary, 0 extended, 4 free)
   e   extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-15228927, default 2048):
Last sector, +sectors or +size{K,M,G,T,P} (2048-15228927, default 15228927):

Created a new partition 1 of type 'Linux' and of size 7.3 GiB.

Command (m for help): t
Selected partition 1
Hex code (type L to list all codes): b
If you have created or modified any DOS 6.x partitions, please see the fdisk \\
documentation for additional information.
Changed type of partition 'Linux' to 'W95 FAT32'.

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
$ sudo mkfs.vfat -F 32 /dev/sdb1
mkfs.fat 3.0.27 (2014-11-12)
$
```

フォーマット完了後、main.efi を"BOOTX64.EFI"へリネームし、USB フラッシュメモリ内に"EFI/BOOT/BOOTX64.EFI"というパスで配置してください。

操作例は以下の通りです。

```
$ sudo mount /dev/sdb1 /mnt
$ sudo mkdir -p /mnt/EFI/BOOT
$ sudo cp main.efi /mnt/EFI/BOOT/BOOTX64.EFI
$ sudo umount /mnt
```

作成した USB フラッシュメモリからブートすると、"Hello UEFI!"と画面へ出力されます (図 1.7)。

▲図 1.7　"Hello UEFI!"と画面出力される様子

シャットダウンの機能は無いので、終了させる際は電源ボタンで終了させてください。

QEMU で実行する

QEMU 上で実行することも可能です。QEMU には UEFI のファームウェアが含まれていないので、ovmf(Open Virtual Machine Firmware) パッケージもインストールしてください。

```
$ sudo apt install qemu-system-x86
$ sudo apt install ovmf
```

QEMU には、指定したディレクトリをハードディスクと見なして実行してくれる機能があります。例えば、fs というディレクトリを作成し、fs/EFI/BOOT/-BOOTX64.EFI に efi ファイルを配置した上で、"-hda fat:fs"というオプションを指定して QEMU を実行すると、QEMU が fs ディレクトリ以下を FAT フォーマットされたハードディスクと見なして実行してくれます。

```
$ qemu-system-x86_64 -bios /usr/share/ovmf/OVMF.fd -hda fat:fs
```

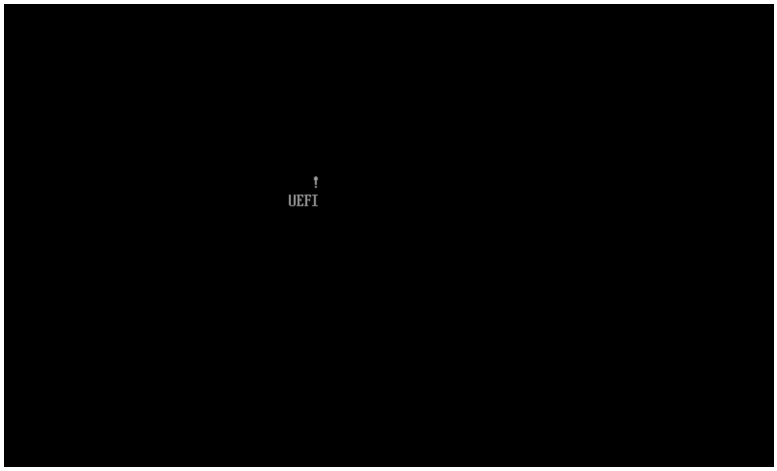
なお、`https://github.com/cupnes/c92_uefi_bare_metal_programming_samples` で公開しているサンプルの Makefile には QEMU で実行するルールも記載済みです。`make run`で実行できます。

```
$ cd c92_uefi_bare_metal_programming_samples/<各サンプルのディレクトリ>
$ make run
```

ただし、OVMF では実装されていないのか、動作しない機能もあります。本書の内容だと、マウス入力を取得する"EFI_SIMPLE_POINTER_PROTOCOL"が動作しませんでした。

日本語表示

Unicode としては日本語も扱えます。ただし、UEFI ファームウェアによってサポートされている文字はまちまちな様です。表紙の写真の通り、筆者の Lenovo 製 ThinkPad E450 の UEFI ファームウェア (バージョン 2.3.1) では、ひらがなや漢字等は一部の文字が表示できませんでした。
なお、QEMU の OVMF で試してみると、日本語は一切表示できないようです (図 1.8)。



▲図 1.8　QEMU(OVMF) での日本語表示

第 2 章

キー入力を取得する

第 1 章では、UEFI 仕様書から機能を調べ、UEFI アプリケーションの作成・実行までの流れを説明しました。この章では、キー入力の取得方法を紹介し、簡単なシェルもどきを作ってみます。

2.1 EFI_SIMPLE_TEXT_INPUT_PROTOCOL

キー入力を取得する関数は"EFI_SIMPLE_TEXT_INPUT_PROTOCOL"の中にあります。EFI_SIMPLE_TEXT_INPUT_PROTOCOL も、前の章でテキスト出力のために使用した EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL と同じく、SystemTable のメンバです (図 2.1)。

```
typedef struct {
  EFI_TABLE_HEADER                  Hdr;
  CHAR16                            *FirmwareVendor;
  UINT32                            FirmwareRevision;
  EFI_HANDLE                        ConsoleInHandle;
  EFI_SIMPLE_TEXT_INPUT_PROTOCOL    *ConIn;
  EFI_HANDLE                        ConsoleOutHandle;
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL   *ConOut;
```

▲図 2.1　EFI_SYSTEM_TABLE の定義 (一部)(再掲)

EFI_SIMPLE_TEXT_INPUT_PROTOCOL の定義はリスト 2.1 の通りです。リスト 2.1 では使用する関数のみ定義しています。EFI_SIMPLE_TEXT_INPUT_PROTOCOL の全体は、仕様書の"11.3 Simple Text Input Protocol(P.420)"を参照してください。

▼**リスト** 2.1　EFI_SIMPLE_TEXT_INPUT_PROTOCOL **の定義**

```
struct EFI_SIMPLE_TEXT_INPUT_PROTOCOL {
        unsigned long long _buf;
        unsigned long long (*ReadKeyStroke)(
                struct EFI_SIMPLE_TEXT_INPUT_PROTOCOL *This,
                struct EFI_INPUT_KEY *Key);
};
```

キー入力は、EFI_SIMPLE_TEXT_INPUT_PROTOCOL の"ReadKeyStroke"関数で取得できます (仕様書"11.3 Simple Text Input Protocol(P.423)"参照)。ReadKeyStroke 関数はノンブロッキングの関数で、関数実行時にキー入力が無ければエラーのステータスを返します。

引数の意味は以下の通りです (第 1 引数は省略)。

struct EFI_INPUT_KEY *Key

取得した文字を格納するポインタ。

なお、EFI_INPUT_KEY 構造体の定義はリスト 2.2 の通りです。

▼**リスト** 2.2　EFI_INPUT_KEY **の定義**

```
struct EFI_INPUT_KEY {
        unsigned short ScanCode;
        unsigned short UnicodeChar;
};
```

また、リスト 2.2 のメンバの意味は以下の通りです。

unsigned short ScanCode

Unicode 範囲外のキー (ESC、上下左右、Fn 等) の値を表すスキャンコード。スキャンコードの一覧は仕様書 P.410 の"Table 88"を参照。本書では ESC キー (ScanCode=0x17) のみ使用する。Unicode 範囲内のキー (英数字、Enter) が入力された時、ScanCode へは 0 が格納される。

unsigned short UnicodeChar

Unicode 範囲内のキー入力時、入力文字に対応する Unicode 値が格納される。Unicode 範囲外のキー入力時、0 が格納される。

2.2 エコーバックプログラムを作ってみる

ReadKeyStroke 関数を使用して、取得した文字を画面へ出力する「エコーバック」のサンプルをリスト 2.3 に示します。サンプルのディレクトリは"sample2_1_echoback"です。

▼**リスト** 2.3　sample2_1_echoback/main.c

```
 1: struct EFI_INPUT_KEY {
 2:     unsigned short ScanCode;
 3:     unsigned short UnicodeChar;
 4: };
 5:
 6: struct EFI_SYSTEM_TABLE {
 7:     char _buf1[44];
 8:     struct EFI_SIMPLE_TEXT_INPUT_PROTOCOL {
 9:             unsigned long long _buf;
10:             unsigned long long (*ReadKeyStroke)(
11:                     struct EFI_SIMPLE_TEXT_INPUT_PROTOCOL *This,
12:                     struct EFI_INPUT_KEY *Key);
13:     } *ConIn;
14:     unsigned long long _buf2;
15:     struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {
16:             unsigned long long _buf;
17:             unsigned long long (*OutputString)(
18:                     struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
19:                     unsigned short *String);
20:             unsigned long long _buf2[4];
21:             unsigned long long (*ClearScreen)(
22:                     struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This);
23:     } *ConOut;
24: };
25:
26: void efi_main(void *ImageHandle __attribute__ ((unused)),
27:           struct EFI_SYSTEM_TABLE *SystemTable)
28: {
29:     struct EFI_INPUT_KEY key;
30:     unsigned short str[3];
31:     SystemTable->ConOut->ClearScreen(SystemTable->ConOut);
32:     while (1) {
33:             if (!SystemTable->ConIn->ReadKeyStroke(SystemTable->ConIn,
34:                                                    &key)) {
35:                     if (key.UnicodeChar != L'\r') {
36:                             str[0] = key.UnicodeChar;
37:                             str[1] = L'\0';
38:                     } else {
39:                             str[0] = L'\r';
40:                             str[1] = L'\n';
41:                             str[2] = L'\0';
42:                     }
43:                     SystemTable->ConOut->OutputString(SystemTable->ConOut,
44:                                                       str);
45:             }
```

```
46:     }
47: }
```

リスト 2.3 では、EFI_SIMPLE_TEXT_INPUT_PROTOCOL の定義を EFI_SYSTEM_TABLE に追加しています。

efi_main 関数内について、ClearScreen 後の while の無限ループがエコーバックの処理です。ReadKeyStroke でキー入力を取得できたら、str 配列へヌル文字 (L'\0') を付加した文字列として格納し、OutputString で画面へ表示します。なお、Enter キーの入力時は CR('\r') を取得するため、取得した文字が CR のときは LF('\n') も出力するようにしています。

サンプルを実行すると、入力した文字がそのまま表示されるエコーバック動作を確認できます (図 2.2)。

▲図 2.2　エコーバックサンプル実行の様子

補足: キー入力を待つ (WaitForKey)

リスト 2.3 では、while() 内で ReadKeyStroke 関数が成功するまで、ReadKeyStroke 関数を何度も呼び出しています。しかし、キー入力が得られるまで CPU を休ませてあげた方が CPU に優しいです。

そのために EFI_SIMPLE_TEXT_INPUT_PROTOCOL には"WaitForKey"というメンバ変数があります (リスト 2.4、仕様書"11.3 Simple Text Input Protocol(P.421)")。

▼リスト 2.4　WaitForKey の定義

```
struct EFI_SIMPLE_TEXT_INPUT_PROTOCOL {
        unsigned long long _buf;
        unsigned long long (*ReadKeyStroke)(
                struct EFI_SIMPLE_TEXT_INPUT_PROTOCOL *This,
                struct EFI_INPUT_KEY *Key);
        void *WaitForKey;
};
```

"void *"は、UEFI でイベントを指す"EFI_EVENT"型の実体で、仕様書上は EFI_EVENT WaitForKeyと定義されています。仕様書 P.421 の WaitForKey の説明にも記載の通り、WaitForKey は WaitForEvent 関数で使用できます。

WaitForEvent は指定したイベントの発生を待つ関数です。SystemTable->BootServices 内で定義されています。BootServices は EFI_BOOT_SERVICES という構造体で、主にブートローダー向けに UEFI が提供している関数 (サービス) を持ちます (詳細は次の章で説明します)。WaitForEvent の定義はリスト 2.5 の通りです。

▼リスト 2.5　WaitForEvent の定義

```
unsigned long long (*WaitForEvent)(
        unsigned long long NumberOfEvents,
        void **Event,
        unsigned long long *Index);
```

引数の意味は以下の通りです。

unsigned long long NumberOfEvents

第 2 引数 Event に指定するイベントの数。

void **Event

イベントリスト。

unsigned long long *Index

発生したイベントのイベントリスト内のインデックスを設定する変数のポインタ。

WaitForKey と WaitForEvent を使用して、リスト 2.6 の様にキー入力を待つことができます。

▼リスト 2.6　WaitForKey と WaitForEvent を使用する例

```
1: struct EFI_INPUT_KEY key;
2: unsigned long long waitidx;
3:
```

```
4: /* キー入力取得まで待機 */
5: SystemTable->BootServices->WaitForEvent(1,
6:     &(SystemTable->ConIn->WaitForKey), &waitidx);
7:
8: /* キー入力取得 */
9: SystemTable->ConIn->ReadKeyStroke(SystemTable->ConIn, &key);
```

2.3 シェルっぽいものを作ってみる

ここまでで、コンソール画面上での文字の入出力ができるようになりました。OS っぽいものを作る上で、まずはシェルっぽいものを作ってみます。サンプルのディレクトリは"sample2_2_shell"です。

このサンプルでは、これから OS っぽいものを作っていく上で土台となるソースコード構成を用意します。ここでは、関数化を適宜行い、ソースコードを以下の様に分けます。

main.c
: エントリポイント (efi_main) を配置。

efi.h,efi.c
: UEFI 仕様上の定義や、初期化処理を配置。

common.h,common.c
: 汎用的に使用される定義や関数を配置。

shell.h,shell.c
: シェルの処理を配置。

エントリポイントの引数である SystemTable は、何をするにしても必要になるため、グローバル変数へ格納しておくことにします。その処理を行うのが efi.c の初期化処理 (efi_init 関数) です (リスト 2.7)[*1]。efi_init では SetWatchdogTimer という関数を呼び出していますが、これについては後述のコラムで説明します。

▼**リスト** 2.7　sample2_2_shell/efi.c

```
1: #include "efi.h"
2: #include "common.h"
3:
4: struct EFI_SYSTEM_TABLE *ST;
5:
6: void efi_init(struct EFI_SYSTEM_TABLE *SystemTable)
```

[*1] EDK2 や gnu-efi といった開発環境やツールチェインでも同様に、SystemTable 等をグローバル変数へ格納する枠組みになっています。

```
 7: {
 8:     ST = SystemTable;
 9:     ST->BootServices->SetWatchdogTimer(0, 0, 0, NULL);
10: }
```

そして、シェルのソースコードはリスト 2.8、エントリポイントのソースコードはリスト 2.9 の通りです。

▼**リスト** 2.8　sample2_2_shell/shell.c

```
 1: #include "common.h"
 2: #include "shell.h"
 3:
 4: #define MAX_COMMAND_LEN    100
 5:
 6: void shell(void)
 7: {
 8:     unsigned short com[MAX_COMMAND_LEN];
 9:
10:     while (TRUE) {
11:             puts(L"poiOS> ");
12:             if (gets(com, MAX_COMMAND_LEN) <= 0)
13:                     continue;
14:
15:             if (!strcmp(L"hello", com))
16:                     puts(L"Hello UEFI!\r\n");
17:             else
18:                     puts(L"Command not found.\r\n");
19:     }
20: }
```

▼**リスト** 2.9　sample2_2_shell/main.c

```
 1: #include "efi.h"
 2: #include "shell.h"
 3:
 4: void efi_main(void *ImageHandle __attribute__ ((unused)),
 5:           struct EFI_SYSTEM_TABLE *SystemTable)
 6: {
 7:     SystemTable->ConOut->ClearScreen(SystemTable->ConOut);
 8:     efi_init(SystemTable);
 9:
10:     shell();
11: }
```

リスト 2.8 では、"OS っぽいもの"ということで、プロンプトに"poiOS"と付けてみました[*2]。

[*2] "mockOS"とか"OSmodoki"とかも考えていたのですが、Google 検索してみると、どちらも既に世の中に存在するようです。

リスト 2.8 で登場した各種の定数や、関数 puts・gets・strcmp は、common.h と common.c で定義しています。これまで説明した UEFI の機能の呼び出し方を関数化しただけなので、紙面上では特に紹介しません (独特な実装方法をしているわけでもないので)。気になる方は、GitHub のサンプルコードをダウンロードして見てみてください。

また、リスト 2.9 では、efi_init 関数で UEFI の初期化処理を行い、shell 関数を実行することでシェルを起動しています。以降、main.c は書き換えません。

そして、サンプル実行の様子は図 2.3 の通りです。

▲図 2.3　シェルもどき実行の様子

5 分のウォッチドッグタイマを解除する

実は UEFI アプリケーション起動時、ウォッチドッグタイマがセットされています。その時間は 5 分ですので、何もしないでいると、UEFI アプリケーションが起動してから 5 分後に再起動することになります。ウォッチドッグタイマは SystemTable->BootServices->SetWatchdogTimer 関数で解除できます。SetWatchdogTimer 関数の定義はリスト 2.10 の通りです (仕様書"6.5 Miscellaneous Boot Services(P.201)")。

▼**リスト** 2.10　SetWatchdogTimer **の定義**

```
unsigned long long (*SetWatchdogTimer)(
        unsigned long long Timeout,
        unsigned long long WatchdogCode,
        unsigned long long DataSize,
        unsigned short *WatchdogData);
```

また、引数の意味は以下の通りです。

unsigned long long Timeout
: ウォッチドッグのタイムアウト時間。0 を指定するとウォッチドッグタイマー無効化。無効化の際、その他の引数は 0 あるいは NULL で良い。

unsigned long long WatchdogCode
: タイムアウト時のウォッチドッグコード (イベント番号) を指定。本書では使用しない。

unsigned long long DataSize
: WatchdogData のデータサイズ (バイト指定)。

unsigned short *WatchdogData
: オプショナルであり、本書では使用しない。加えて、何なのか良く分かっていないです (ウォッチドッグイベント発生時にログに記録される追加の説明?)。

ウォッチドッグタイマー無効化のコード例はリスト 2.11 の通りです。

▼**リスト** 2.11　**ウォッチドッグタイマ無効化**

```
ST->BootServices->SetWatchdogTimer(0, 0, 0, NULL);
```

第 3 章

画面に絵を描く

シェルっぽいものが一応できたので、次は GUI っぽいものを作ってみます。まずは画面へ絵を描く方法を紹介し、アイコン代わりに矩形を描いてみます。サンプルプログラムは"sample3_1_draw_rect"ディレクトリです。

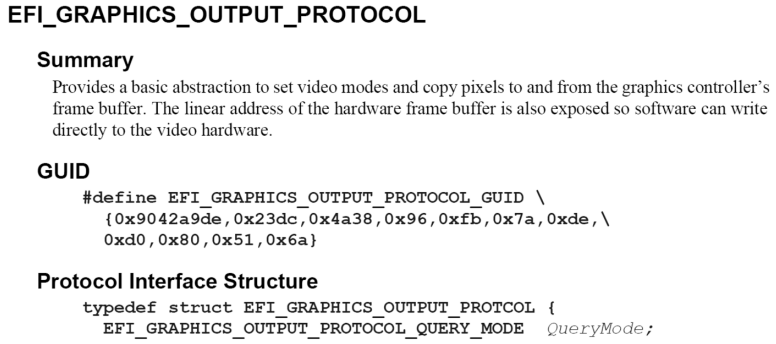
3.1 EFI_GRAPHICS_OUTPUT_PROTOCOL

グラフィック描画には"EFI_GRAPHICS_OUTPUT_PROTOCOL"を使用します (仕様書"11.9 Graphics Output Protocol(P.464)"参照)。ただし、このプロトコルは SystemTable のメンバにはありません。

実はほとんどのプロトコルは SystemTable->BootServices の中の関数を使用してプロトコルの先頭アドレスを取得する必要があります (仕様書"4.4 EFI Boot Services Table(P.80)"参照)。BootServices は EFI_BOOT_SERVICES という構造体で、主にブートローダー向けに UEFI が提供している関数 (サービス) を持ちます。[*1]

SystemTable のメンバに無いほとんどのプロトコルは、SystemTable->BootServices->LocateProtocol 関数 (仕様書"6.3 Protocol Handler Services(P.184)"参照) でプロトコル構造体の先頭アドレスを取得できます。LocateProtocol 関数は、プロトコル毎に一意に決められている"GUID"からプロトコルの先頭アドレスを取得する関数です。"GUID"も仕様書に記載されています。例えば EFI_GRAPHICS_OUTPUT_PROTOCOL の場合、図 3.1 のように記載されています。

[*1] "～_SERVICES"には他に"EFI_RUNTIME_SERVICES"があり、こちらも SystemTable->RuntimeServices という形で SystemTable から参照できます。



EFI_GRAPHICS_OUTPUT_PROTOCOL

Summary

Provides a basic abstraction to set video modes and copy pixels to and from the graphics controller's frame buffer. The linear address of the hardware frame buffer is also exposed so software can write directly to the video hardware.

GUID

```
#define EFI_GRAPHICS_OUTPUT_PROTOCOL_GUID \
  {0x9042a9de,0x23dc,0x4a38,0x96,0xfb,0x7a,0xde,\
  0xd0,0x80,0x51,0x6a}
```

Protocol Interface Structure

```
typedef struct EFI_GRAPHICS_OUTPUT_PROTOCOL {
  EFI_GRAPHICS_OUTPUT_PROTOCOL_QUERY_MODE  QueryMode;
```

▲図 3.1　EFI_GRAPHICS_OUTPUT_PROTOCOL の GUID

LocateProtocol の定義はリスト 3.1 の通りです。

▼リスト 3.1　LocateProtocol の定義

```
unsigned long long (*LocateProtocol)(
        struct EFI_GUID *Protocol,
        void *Registration,
        void **Interface);
```

引数の意味は以下の通りです。

struct EFI_GUID *Protocol

取得したいプロトコルの GUID を指定。

void *Registration

オプショナル。必要に応じてレジストレーションキーというものを指定するらしい。本書では使用しない (NULL 指定)。

void **Interface

プロトコル構造体の先頭アドレスを格納するポインタを指定。

SystemTable と同じく、EFI_GRAPHICS_OUTPUT_PROTOCOL も、efi_init 関数でグローバル変数へ格納することにします。LocateProtocol 処理を追加した efi_init はリスト 3.2 の通りです。

▼リスト 3.2　sample3_1_draw_rect/efi.c

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: struct EFI_SYSTEM_TABLE *ST;
 5: struct EFI_GRAPHICS_OUTPUT_PROTOCOL *GOP;
 6:
 7: void efi_init(struct EFI_SYSTEM_TABLE *SystemTable)
 8: {
 9:     struct EFI_GUID gop_guid = {0x9042a9de, 0x23dc, 0x4a38, \
10:                                 {0x96, 0xfb, 0x7a, 0xde,   \
11:                                  0xd0, 0x80, 0x51, 0x6a}};
12:
13:     ST = SystemTable;
14:     ST->BootServices->SetWatchdogTimer(0, 0, 0, NULL);
15:     ST->BootServices->LocateProtocol(&gop_guid, NULL, (void **)&GOP);
16: }
```

これで、EFI_GRAPHICS_OUTPUT_PROTOCOL を取得できました。EFI_GRAPHICS_OUTPUT_PROTOCOL の定義はリスト 3.3 の通りです (仕様書"11.9.1 Blt Buffer(P.466)")。なお、例によって、定義の内容は本書で使用するメンバのみに限定しています。定義の全体は仕様書を確認してください。

▼**リスト** 3.3　EFI_GRAPHICS_OUTPUT_PROTOCOL **の定義**

```
struct EFI_GRAPHICS_OUTPUT_PROTOCOL {
        unsigned long long _buf[3];
        struct EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE {
                unsigned int MaxMode;
                unsigned int Mode;
                struct EFI_GRAPHICS_OUTPUT_MODE_INFORMATION {
                        unsigned int Version;
                        unsigned int HorizontalResolution;
                        unsigned int VerticalResolution;
                        enum EFI_GRAPHICS_PIXEL_FORMAT {
                                PixelRedGreenBlueReserved8BitPerColor,
                                PixelBlueGreenRedReserved8BitPerColor,
                                PixelBitMask,
                                PixelBltOnly,
                                PixelFormatMax
                        } PixelFormat;
                } *Info;
                unsigned long long SizeOfInfo;
                unsigned long long FrameBufferBase;
        } *Mode;
};
```

画面描画はフレームバッファへピクセルデータを書き込むことで行います。GOP->Mode->FrameBufferBase 変数からフレームバッファの先頭アドレスを取得できます。ピクセルフォーマットは GOP->Mode->Info->PixelFormat 変数から確認できます。PixelFormat は"enum EFI_GRAPHICS_PIXEL_FORMAT"という enum 定数で

す。この enum 定数により、ピクセルフォーマットが何であるかを確認できます (仕様書"11.9.1 Blt Buffer(P.467)")。

EFI_GRAPHICS_PIXEL_FORMAT の定義はリスト 3.4 の通りです。筆者が動作確認に使用できる環境 (ThinkPad E450 と QEMU の OVMF) では全て、ピクセルフォーマットは PixelBlueGreenRedReserved8BitPerColor であったため、本書ではピクセルフォーマットは「BGR+Reserved 各 8 ビット」を想定します。もし異なる場合は、適宜読み替えてください。

▼**リスト** 3.4　EFI_GRAPHICS_PIXEL_FORMAT **の定義**

```
enum EFI_GRAPHICS_PIXEL_FORMAT {
        PixelRedGreenBlueReserved8BitPerColor,
        PixelBlueGreenRedReserved8BitPerColor,
        PixelBitMask,
        PixelBltOnly,
        PixelFormatMax
};
```

なお、GOP->Mode->Info->PixelFormat の値を確認するには画面に数値を出力できる必要があります。次の章のサンプル (sample4_1_get_pointer_state ディレクトリ) で、16 進数で数値を出力する関数 puth を common.c に追加しますので、参考にしてみてください。

加えて、ピクセルフォーマットのバイト列を定義する EFI_GRAPHICS_OUTPUT_BLT_PIXEL 構造体もリスト 3.5 に示します (仕様書"11.9.1 Blt Buffer(P.474)")。

▼**リスト** 3.5　EFI_GRAPHICS_OUTPUT_BLT_PIXEL **の定義**

```
struct EFI_GRAPHICS_OUTPUT_BLT_PIXEL {
        unsigned char Blue;
        unsigned char Green;
        unsigned char Red;
        unsigned char Reserved;
};
```

以上で、画面に矩形を描く為の UEFI の定義の追加は終了です。

3.2　矩形を描くサンプルを実装

フレームバッファの先頭アドレスもピクセルフォーマットも分かったので、後はピクセルフォーマットに従ったバイト列をフレームバッファの領域へ書き込むだけです。まずは 1 ピクセルを指定の座標に描く関数"draw_pixel"を作成します (リスト 3.6)。グラフィッ

クス関係の処理は"graphics.c"というソースファイルを作成し、そこへ追加します。

▼**リスト** 3.6　sample3_1_draw_rect/graphics.c(draw_pixel **関数**)

```
 1: void draw_pixel(unsigned int x, unsigned int y,
 2:              struct EFI_GRAPHICS_OUTPUT_BLT_PIXEL color)
 3: {
 4:     unsigned int hr = GOP->Mode->Info->HorizontalResolution;
 5:     struct EFI_GRAPHICS_OUTPUT_BLT_PIXEL *base =
 6:             (struct EFI_GRAPHICS_OUTPUT_BLT_PIXEL *)GOP->Mode->FrameBufferBase;
 7:     struct EFI_GRAPHICS_OUTPUT_BLT_PIXEL *p = base + (hr * y) + x;
 8:
 9:     p->Blue = color.Blue;
10:     p->Green = color.Green;
11:     p->Red = color.Red;
12:     p->Reserved = color.Reserved;
13: }
```

リスト 3.6 では、以下を行っています。

1. GOP->Mode->Info->HorizontalResolutionで水平解像度を取得
2. 水平解像度の値と与えられた X、Y 座標値から 1 ピクセルを描くアドレスを計算
3. 2. へ color 変数の値を書き込み

なお、ここで取得している水平解像度は UEFI ファームウェアがデフォルトで認識している画面モードの解像度で、筆者の ThinkPad E450 の場合 640 ピクセルでした。画面モードは EFI_GRAPHICS_OUTPUT_PROTOCOL の SetMode 関数で変更できますので興味があれば試してみてください (仕様書"11.9.1 Blt Buffer(P.473)")。SetMode 関数ではモード番号を指定しますが、この番号の最大値は同じく EFI_GRAPHICS_OUTPUT_PROTOCOL の Mode->MaxMode から確認できます (仕様書"11.9.1 Blt Buffer(P.466)")。

次に、draw_pixel を使用して、draw_rect をリスト 3.7 のように実装できます。

▼**リスト** 3.7　sample3_1_draw_rect/graphics.c(draw_rect **関数**)

```
 1: void draw_rect(struct RECT r, struct EFI_GRAPHICS_OUTPUT_BLT_PIXEL c)
 2: {
 3:     unsigned int i;
 4:
 5:     for (i = r.x; i < (r.x + r.w); i++)
 6:             draw_pixel(i, r.y, c);
 7:     for (i = r.x; i < (r.x + r.w); i++)
 8:             draw_pixel(i, r.y + r.h - 1, c);
 9:
10:     for (i = r.y; i < (r.y + r.h); i++)
11:             draw_pixel(r.x, i, c);
12:     for (i = r.y; i < (r.y + r.h); i++)
```

```
13:             draw_pixel(r.x + r.w - 1, i, c);
14: }
```

なお、RECT 構造体はリスト 3.8 のように graphics.h で定義します。

▼**リスト** 3.8　RECT **構造体の定義**

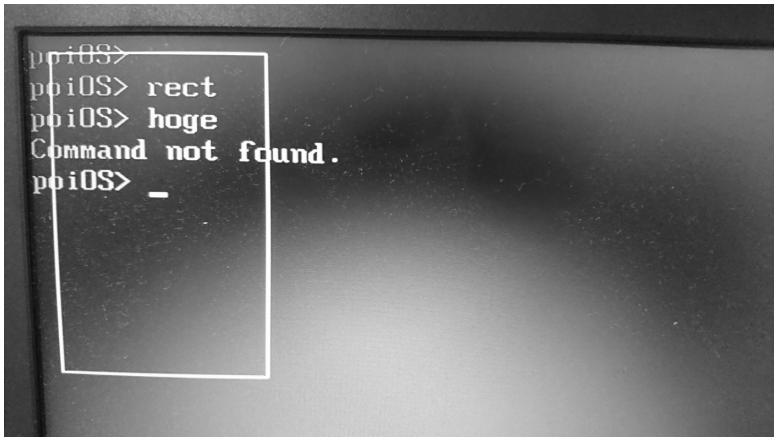
```
struct RECT {
        unsigned int x, y;
        unsigned int w, h;
};
```

以上を使用して、画面に矩形を描く"rect"コマンドを追加してみます。追加後の shell.c はリスト 3.9 の通りです。

▼**リスト** 3.9　sample3_1_draw_rect/shell.c

```
 1: #include "common.h"
 2: #include "graphics.h"
 3: #include "shell.h"
 4:
 5: #define MAX_COMMAND_LEN    100
 6:
 7: void shell(void)
 8: {
 9:     unsigned short com[MAX_COMMAND_LEN];
10:     struct RECT r = {10, 10, 100, 200};   /* 追加 */
11:
12:     while (TRUE) {
13:             puts(L"poiOS> ");
14:             if (gets(com, MAX_COMMAND_LEN) <= 0)
15:                     continue;
16:
17:             if (!strcmp(L"hello", com))
18:                     puts(L"Hello UEFI!\r\n");
19:             else if (!strcmp(L"rect", com)) /* 追加 */
20:                     draw_rect(r, white);    /* 追加 */
21:             else
22:                     puts(L"Command not found.\r\n");
23:     }
24: }
```

実機で実行した様子は図 3.2 の通りです。



▲図 3.2　rect コマンド実行の様子

3.3　GUI モードを追加する

GUI のモードへ切り替える"gui"コマンドを追加してみます。サンプルのディレクトリは"sample3_2_add_gui_mode"です。

GUI モードとしては、まずはアイコンに見立てた矩形を 1 つ配置するだけです。GUI モードの実装は、新たに gui.c を作成し、そこへ記述することにします。そして、シェルからは、gui コマンドが実行されたときに、gui.c に実装されている gui 関数を呼び出すこととします。

gui.c の実装をリスト 3.10 に示します。

▼リスト 3.10　sample3_2_add_gui_mode/gui.c

```
 1: #include "efi.h"
 2: #include "common.h"
 3: #include "graphics.h"
 4: #include "gui.h"
 5:
 6: void gui(void)
 7: {
 8:     struct RECT r = {10, 10, 20, 20};
 9:
10:     ST->ConOut->ClearScreen(ST->ConOut);
11:
12:     /* ファイルアイコンに見立てた矩形を描画 */
13:     draw_rect(r, white);
14:
```

```
15:     while (TRUE);
16: }
```

gui 関数を呼び出す gui コマンドを追加した shell.c をリスト 3.11 に示します。

▼**リスト** 3.11　sample3_2_add_gui_mode/shell.c

```
 1: #include "common.h"
 2: #include "graphics.h"
 3: #include "shell.h"
 4: #include "gui.h"   /* 追加 */
 5:
 6: #define MAX_COMMAND_LEN    100
 7:
 8: void shell(void)
 9: {
10:     unsigned short com[MAX_COMMAND_LEN];
11:     struct RECT r = {10, 10, 100, 200};
12:
13:     while (TRUE) {
14:             puts(L"poiOS> ");
15:             if (gets(com, MAX_COMMAND_LEN) <= 0)
16:                     continue;
17:
18:             if (!strcmp(L"hello", com))
19:                     puts(L"Hello UEFI!\r\n");
20:             else if (!strcmp(L"rect", com))
21:                     draw_rect(r, white);
22:             else if (!strcmp(L"gui", com))  /* 追加 */
23:                     gui();  /* 追加 */
24:             else
25:                     puts(L"Command not found.\r\n");
26:     }
27: }
```

図 3.3 のように gui コマンドを実行すると、アイコンに見立てた矩形をひとつだけ描いた状態の画面 (図 3.4) へ遷移します。

▲図 3.3　gui コマンド実行の様子

▲図 3.4　アイコン一つ表示するだけの GUI モード

第 4 章

マウス入力を取得する

次はマウスを使ってみます。この章では、マウス入力を取得し、マウスの動きに合わせてマウスカーソルを描画してみます。

4.1 EFI_SIMPLE_POINTER_PROTOCOL

マウスの入力値を取得するには EFI_SIMPLE_POINTER_PROTOCOL を使用します (仕様書"11.5 Simple Pointer Protocol(P.439)")。本書で使用する箇所の定義はリスト 4.1 の通りです。

▼**リスト** 4.1　EFI_SIMPLE_POINTER_PROTOCOL **の定義**

```
struct EFI_SIMPLE_POINTER_PROTOCOL {
        unsigned long long (*Reset)(
                struct EFI_SIMPLE_POINTER_PROTOCOL *This,
                unsigned char ExtendedVerification);
        unsigned long long (*GetState)(
                struct EFI_SIMPLE_POINTER_PROTOCOL *This,
                struct EFI_SIMPLE_POINTER_STATE *State);
        void *WaitForInput;
};
```

EFI_SIMPLE_POINTER_PROTOCOL では、Reset 関数でポインティングデバイス (マウス) をリセットし、GetState 関数で状態を取得します。

Reset 関数の引数の意味は以下の通りです。

unsigned char ExtendedVerification

拡張の検査 (ExtendedVerification) を行うか否かのフラグ。TRUE が設定されていると拡張検査が行われる。どのような拡張検査が行われるかはファームウェア依存。本書では特にそのような検査は不要であるため FALSE を指定。

また、GetState 関数の引数については以下の通りです。

struct EFI_SIMPLE_POINTER_STATE *State

ポインティングデバイスの状態を指定されたポインタ変数へ格納。

なお、状態を表す EFI_SIMPLE_POINTER_STATE 構造体の定義はリスト 4.2 の通りです。

▼**リスト** 4.2　EFI_SIMPLE_POINTER_STATE **の定義**

```
struct EFI_SIMPLE_POINTER_STATE {
        int RelativeMovementX;  /* X 軸方向の相対移動量 */
        int RelativeMovementY;  /* Y 軸方向の相対移動量 */
        int RelativeMovementZ;  /* Z 軸方向の相対移動量 */
        unsigned char LeftButton;       /* 左ボタン押下状態 (TRUE=1,FALSE=0) */
        unsigned char RightButton;      /* 右ボタン押下状態 (TRUE=1,FALSE=0) */
};
```

リスト 4.2 に関し、各メンバの意味はコメントの通りです。なお、"RelativeMovementZ" は普通のマウスを使う限り常に 0 でした (特殊なマウスを持っていないので、筆者が試す限り 0 以外見たことがありません)。

EFI_SIMPLE_POINTER_PROTOCOL(リスト 4.1) の"WaitForInput"は、マウス入力があるまで待機するためのイベントです。EFI_SIMPLE_TEXT_INPUT_PROTOCOL の WaitForKey と同様に、SystemTable->BootServices->WaitForEvent 関数へ渡して使うことができます。

マウスの状態を見てみる (pstat コマンド)

EFI_SIMPLE_POINTER_PROTOCOL を使うことでマウスの状態を取得できることが分かったので、試しにマウスの状態をコンソール上へダンプしてみます。ここでは pstat というコマンドとしてマウス状態ダンプの機能を追加してみます。サンプルのディレクトリは"sample4_1_get_pointer_state"です。

まず、LocateProtocol 関数で EFI_SIMPLE_POINTER_PROTOCOL の構造体の先頭アドレスを取得します。そのため、efi.c の efi_init 関数へ LocateProtocol の処理を追加し、グローバル変数としてアクセスできるようにします (リスト 4.3)。

▼**リスト** 4.3　sample4_1_get_pointer_state/efi.c

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: struct EFI_SYSTEM_TABLE *ST;
 5: struct EFI_GRAPHICS_OUTPUT_PROTOCOL *GOP;
 6: struct EFI_SIMPLE_POINTER_PROTOCOL *SPP;    /* 追加 */
 7:
 8: void efi_init(struct EFI_SYSTEM_TABLE *SystemTable)
 9: {
10:     struct EFI_GUID gop_guid = {0x9042a9de, 0x23dc, 0x4a38, \
11:                                 {0x96, 0xfb, 0x7a, 0xde,   \
12:                                  0xd0, 0x80, 0x51, 0x6a}};
13:     /* 追加（ここから） */
14:     struct EFI_GUID spp_guid = {0x31878c87, 0xb75, 0x11d5, \
15:                                 {0x9a, 0x4f, 0x0, 0x90,    \
16:                                  0x27, 0x3f, 0xc1, 0x4d}};
17:             /* 追加（ここまで） */
18:
19:     ST = SystemTable;
20:     ST->BootServices->SetWatchdogTimer(0, 0, 0, NULL);
21:     ST->BootServices->LocateProtocol(&gop_guid, NULL, (void **)&GOP);
22:     /* 追加 */
23:     ST->BootServices->LocateProtocol(&spp_guid, NULL, (void **)&SPP);
24: }
```

pstat コマンドを追加した shell.c はリスト 4.4 の通りです。

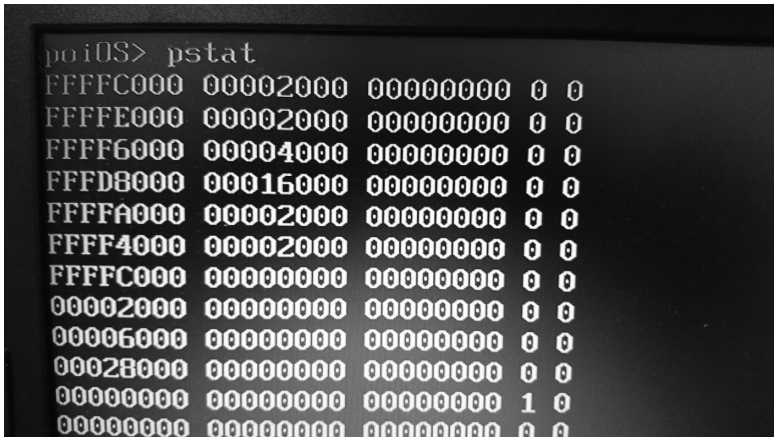
▼**リスト** 4.4　sample4_1_get_pointer_state/shell.c

```
 1: #include "efi.h"   /* 追加 */
 2: #include "common.h"
 3: #include "graphics.h"
 4: #include "shell.h"
 5: #include "gui.h"
 6:
 7: #define MAX_COMMAND_LEN    100
 8:
 9: /* 追加（ここから） */
10: void pstat(void)
11: {
12:     unsigned long long status;
13:     struct EFI_SIMPLE_POINTER_STATE s;
14:     unsigned long long waitidx;
15:
16:     SPP->Reset(SPP, FALSE);
17:
18:     while (1) {
19:             ST->BootServices->WaitForEvent(1, &(SPP->WaitForInput),
20:                                            &waitidx);
21:             status = SPP->GetState(SPP, &s);
22:             if (!status) {
23:                     puth(s.RelativeMovementX, 8);
24:                     puts(L" ");
25:                     puth(s.RelativeMovementY, 8);
26:                     puts(L" ");
```

```
27:                     puth(s.RelativeMovementZ, 8);
28:                     puts(L" ");
29:                     puth(s.LeftButton, 1);
30:                     puts(L" ");
31:                     puth(s.RightButton, 1);
32:                     puts(L"\r\n");
33:             }
34:     }
35: }
36: /* 追加（ここまで） */
37:
38: void shell(void)
39: {
40:     unsigned short com[MAX_COMMAND_LEN];
41:     struct RECT r = {10, 10, 100, 200};
42:
43:     while (TRUE) {
44:             puts(L"poiOS> ");
45:             if (gets(com, MAX_COMMAND_LEN) <= 0)
46:                     continue;
47:
48:             if (!strcmp(L"hello", com))
49:                     puts(L"Hello UEFI!\r\n");
50:             /* ・・・省略・・・ */
51:             else if (!strcmp(L"pstat", com))        /* 追加 */
52:                     pstat();        /* 追加 */
53:             else
54:                     puts(L"Command not found.\r\n");
55:     }
56: }
```

リスト 4.4 に関して、pstat 関数では puth 関数を使用して画面へ数値を表示しています。puth 関数はこのサンプルから common.c へ追加した関数で、引数で指定した数値を 16 進数で表示します。第 1 引数が表示する数値で、第 2 引数が表示する際の桁数です。

最後にリスト 4.4 のサンプルの実行の様子を図 4.1 に示します。



▲**図** 4.1　pstat **コマンド実行の様子**

4.2　マウスカーソルを追加する

グラフィックを描画する方法が分かり、マウスの入力値の取得もできたので、アイコンもどきの矩形を表示するだけだった GUI モードへマウスカーソルを表示する機能を追加してみます。サンプルのディレクトリは"sample4_2_add_cursor"です。

実装の仕方としては、ここではとにかく簡単にするため、「マウスカーソルは 1 ドット」で作ってみます。マウスカーソルの機能を追加した gui.c をリスト 4.5 に示します。

▼**リスト** 4.5　sample4_2_add_cursor/gui.c

```
 1: #include "efi.h"
 2: #include "common.h"
 3: #include "graphics.h"
 4: #include "shell.h"
 5: #include "gui.h"
 6:
 7: /* 追加（ここから）*/
 8: struct EFI_GRAPHICS_OUTPUT_BLT_PIXEL cursor_tmp = {0, 0, 0, 0};
 9: int cursor_old_x;
10: int cursor_old_y;
11:
12: void draw_cursor(int x, int y)
13: {
14:     draw_pixel(x, y, white);
15: }
16:
17: void save_cursor_area(int x, int y)
18: {
```

```
19:     cursor_tmp = get_pixel(x, y);
20:     cursor_tmp.Reserved = 0xff;
21: }
22:
23: void load_cursor_area(int x, int y)
24: {
25:     draw_pixel(x, y, cursor_tmp);
26: }
27:
28: void put_cursor(int x, int y)
29: {
30:     if (cursor_tmp.Reserved)
31:             load_cursor_area(cursor_old_x, cursor_old_y);
32:
33:     save_cursor_area(x, y);
34:
35:     draw_cursor(x, y);
36:
37:     cursor_old_x = x;
38:     cursor_old_y = y;
39: }
40: /* 追加（ここまで） */
41:
42: void gui(void)
43: {
44:     struct RECT r = {10, 10, 20, 20};
45:     /* 追加・変更（ここから） */
46:     unsigned long long status;
47:     struct EFI_SIMPLE_POINTER_STATE s;
48:     int px = 0, py = 0;
49:     unsigned long long waitidx;
50:     unsigned char is_highlight = FALSE;
51:
52:     ST->ConOut->ClearScreen(ST->ConOut);
53:     SPP->Reset(SPP, FALSE);
54:
55:     /* ファイルアイコンに見立てた矩形を描画 */
56:     draw_rect(r, white);
57:
58:     while (TRUE) {
59:             ST->BootServices->WaitForEvent(1, &(SPP->WaitForInput), &waitidx);
60:             status = SPP->GetState(SPP, &s);
61:             if (!status) {
62:                     /* マウスカーソル座標更新 */
63:                     px += s.RelativeMovementX >> 13;
64:                     if (px < 0)
65:                             px = 0;
66:                     else if (GOP->Mode->Info->HorizontalResolution <=
67:                              (unsigned int)px)
68:                             px = GOP->Mode->Info->HorizontalResolution - 1;
69:                     py += s.RelativeMovementY >> 13;
70:                     if (py < 0)
71:                             py = 0;
72:                     else if (GOP->Mode->Info->VerticalResolution <=
73:                              (unsigned int)py)
74:                             py = GOP->Mode->Info->VerticalResolution - 1;
75:
76:                     /* マウスカーソル描画 */
```

```
77:                     put_cursor(px, py);
78:
79:                     /* ファイルアイコン処理 */
80:                     if (is_in_rect(px, py, r)) {
81:                             if (is_highlight) {
82:                                     draw_rect(r, yellow);
83:                                     is_highlight = TRUE;
84:                             }
85:                     } else {
86:                             if (is_highlight) {
87:                                     draw_rect(r, white);
88:                                     is_highlight = FALSE;
89:                             }
90:                     }
91:             }
92:     }
93:     /* 追加・変更（ここまで） */
94: }
```

gui 関数へ追加した処理の流れとしては、ClearScreen 関数から draw_rect 関数までで、画面とマウスの初期化を行い、アイコン (矩形) の描画を行っています。`while(TRUE)`内の定常処理では以下の流れで処理を行っています。

1. WaitForEvent で待機し、GetState 関数でマウス入力値を取得
2. マウスカーソル座標更新
3. マウスカーソル描画 (put_cursor 関数)
4. ファイルアイコン処理

なお、put_cursor 関数は引数で指定した座標へマウスカーソルを移動させる関数です。put_cursor では以下の流れで処理を行います。

1. 退避していたピクセルデータを復帰 (load_cursor_area)
2. 描画するカーソル位置のピクセルデータを退避 (save_cursor_area)
3. カーソル描画 (draw_cursor)
4. カーソル位置を退避 (cursor_old_x,cursor_old_y)

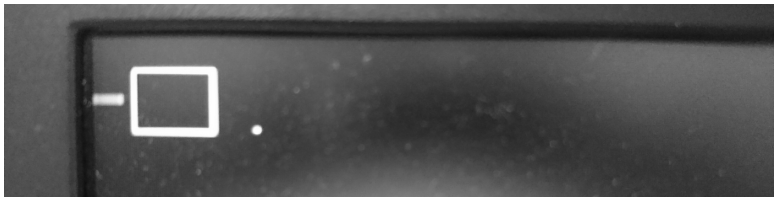
save_cursor_area 関数内に関して、get_pixel 関数は graphics.c へ新たに追加した関数で、フレームバッファからピクセルデータを取得します。ピクセルデータは EFI_GRAPHICS_OUTPUT_BLT_PIXEL 構造体の形式です。フレームバッファから読み出した際の Reserved メンバの値は 0 なのですが、フレームバッファへ書き込むときに 0 を指定していると何も表示されないので、save_cursor_area 関数内で Reserved は 0xff で上書きしています。これを利用して、put_cursor 関数内では、退避済みのピクセルデータがあるか否かを Reserved メンバの値で確認しています。

gui 関数内の `while(TRUE)`内の処理に戻り、"2. マウスカーソル座標更新"のマウスの

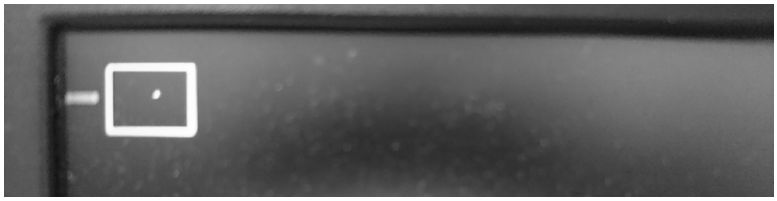
移動量計算は、pstat コマンドで確認した結果からおおよそ算出したものです。下位 12 ビットが全て 0 であり、12 ビットシフトするだけでは移動量としては大きすぎたため、13 ビット右シフトしています。もし pstat コマンドで確認した結果が異なるようであれば、適宜修正してください。

"4. ファイルアイコン処理"のアイコン更新処理では、現在のマウスカーソル座標がアイコン (矩形) の範囲内であればアイコンをハイライト色で上書きし、そうでなければ元の色で上書きしています。

サンプルを実行し、シェル上で gui コマンドを実行することで GUI モードへ遷移すると、図 4.2 の画面になります。アイコン (矩形) の右側にある点がマウスカーソルです[*1]。マウスカーソルがアイコンの上にのると、枠が黄色にハイライト表示されます (図 4.3)[*2]。



▲図 4.2　マウスカーソルが表示される様子



▲図 4.3　マウスカーソルがアイコンにのるとハイライト表示

[*1] ゴミのように見えますがマウスカーソルなのです。。。

[*2] 白黒の印刷なので印刷上は分からないかも知れませんが。。

第 5 章

ファイル読み書き

マウスカーソルを用意できたので、何かクリックしてみたいです。UEFI のファームウェアは FAT ファイルシステム上のファイルを操作するプロトコルを持っていますので、それを使ってファイル操作機能を追加してみます。

5.1 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL と EFI_FILE_PROTOCOL

UEFI では FAT ファイルシステムを扱えます。ファイルシステムを操作するためのプロトコルが"EFI_SIMPLE_FILE_SYSTEM_PROTOCOL"と"EFI_FILE_PROTOCOL"です (仕様書"12.4 Simple File System Protocol(P.494)"と"12.5 EFI File Protocol(P.497)")。

EFI_SIMPLE_FILE_SYSTEM_PROTOCOL の定義はリスト 5.1 の通りです。

▼**リスト** 5.1　EFI_SIMPLE_FILE_SYSTEM_PROTOCOL **の定義**

```
struct EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
        unsigned long long Revision;
        unsigned long long (*OpenVolume)(
                struct EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *This,
                struct EFI_FILE_PROTOCOL **Root);
};
```

リスト 5.1 は定義の全体で、関数は OpenVolume のみを持っています。OpenVolume は名前の通りボリュームを開く関数です。"ボリューム"はディスクのパーティションに当たるもので、Windows OS で言うところの"ドライブ"に相当するものです。OpenVolume 関数の引数の意味は以下の通りです。

struct EFI_FILE_PROTOCOL **Root

ボリュームを開いて得られた最上位階層のディレクトリ (ルートディレクトリ)。

UEFI ではファイル/ディレクトリの各エントリを EFI_FILE_PROTOCOL という構造体で扱います。OpenVolume で得られるルートディレクトリエントリも EFI_FILE_PROTOCOL です。EFI_FILE_PROTOCOL の定義をリスト 5.2 に示します (本書で使用するもののみ定義しています)。

▼**リスト** 5.2　EFI_FILE_PROTOCOL **の定義**

```
struct EFI_FILE_PROTOCOL {
        unsigned long long _buf;
        unsigned long long (*Open)(struct EFI_FILE_PROTOCOL *This,
                                   struct EFI_FILE_PROTOCOL **NewHandle,
                                   unsigned short *FileName,
                                   unsigned long long OpenMode,
                                   unsigned long long Attributes);
        unsigned long long (*Close)(struct EFI_FILE_PROTOCOL *This);
        unsigned long long _buf2;
        unsigned long long (*Read)(struct EFI_FILE_PROTOCOL *This,
                                   unsigned long long *BufferSize,
                                   void *Buffer);
        unsigned long long (*Write)(struct EFI_FILE_PROTOCOL *This,
                                    unsigned long long *BufferSize,
                                    void *Buffer);
        unsigned long long _buf3[4];
        unsigned long long (*Flush)(struct EFI_FILE_PROTOCOL *This);
};
```

リスト 5.2 の各関数の使い方は、以降の節で説明します。

5.2　ルートディレクトリ直下のファイル/ディレクトリを一覧表示 (ls)

ルートディレクトリ直下のファイル/ディレクトリを一覧表示する"ls"コマンドを作成してみます。サンプルのディレクトリは"sample5_1_ls"です。

まずは、EFI_SIMPLE_FILE_SYSTEM_PROTOCOL を使えるように、efi.c の efi_init 関数へ LocateProtocol 関数の処理を追加します (リスト 5.3)。

▼**リスト** 5.3　sample5_1_ls/efi.c

```
 1: /* ・・・省略・・・ */
 2: struct EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *SFSP;       /* 追加 */
 3:
 4: void efi_init(struct EFI_SYSTEM_TABLE *SystemTable)
 5: {
 6:     /* ・・・省略・・・ */
```

```
 7:     /* 追加（ここから） */
 8:     struct EFI_GUID sfsp_guid = {0x0964e5b22, 0x6459, 0x11d2, \
 9:                                  {0x8e, 0x39, 0x00, 0xa0,    \
10:                                   0xc9, 0x69, 0x72, 0x3b}};
11:             /* 追加（ここまで） */
12:     /* ・・・省略・・・ */
13:     /* 追加 */
14:     ST->BootServices->LocateProtocol(&sfsp_guid, NULL, (void **)&SFSP);
15: }
```

これで、グローバル変数"SFSP"を通して EFI_SIMPLE_FILE_SYSTEM_PROTOCOL を使用できるようになりました。次は、SFSP->OpenVolume 関数を呼び出すことでルートディレクトリを開きます。コード例はリスト 5.4 の通りです。

▼**リスト** 5.4　OpenVolume **関数の使用例**

```
1: struct EFI_FILE_PROTOCOL *root;
2: SFSP->OpenVolume(SFSP, &root);
```

EFI_FILE_PROTOCOL がディレクトリである場合、Read 関数を呼び出す事で、ディレクトリ内に存在するファイル/ディレクトリ名を取得できます。EFI_FILE_PROTOCOL の Read 関数の定義はリスト 5.5 の通りです。

▼**リスト** 5.5　EFI_FILE_PROTOCOL **の** Read **関数の定義**

```
unsigned long long (*Read)(struct EFI_FILE_PROTOCOL *This,
                           unsigned long long *BufferSize,
                           void *Buffer);
```

引数の意味は以下の通りです。

unsigned long long *BufferSize

第 3 引数"Buffer"のサイズ (バイト指定) を格納した変数のポインタを指定。Read 関数実行後、このポインタで参照された変数には read したデータサイズが格納される。EFI_FILE_PROTOCOL がディレクトリの場合、全てのファイル/ディレクトリ名を取得し終えると、0 が格納される。

void *Buffer

read したデータを格納するバッファの先頭アドレス。EFI_FILE_PROTOCOL がディレクトリの場合、1 回の read につき 1 つのファイル/ディレクトリ名を格納する。

ファイル/ディレクトリに対する一通りの処理を終えたら、EFI_FILE_PROTOCOL

の Close 関数を実行します。Close 関数の定義はリスト 5.6 の通りです。

▼**リスト** 5.6　EFI_FILE_PROTOCOL **の** Close **関数の定義**

```
unsigned long long (*Close)(struct EFI_FILE_PROTOCOL *This);
```

以上を踏まえて、起動ディスク直下のファイル/ディレクトリを一覧表示するコマンドを追加してみます。

まずは、ファイルの付帯情報を管理する構造体の配列"struct FILE file_list[]"を作成します。管理する付帯情報としては、今はまだ「ファイル名」だけです。行数は少ないのですが、file.h と file.c というファイル名でリスト 5.7 とリスト 5.8 のソースコードを作成します。

▼**リスト** 5.7　sample_5_1_ls/file.h

```
 1: #ifndef _FILE_H_
 2: #define _FILE_H_
 3:
 4: #include "graphics.h"
 5:
 6: #define MAX_FILE_NAME_LEN  4
 7: #define MAX_FILE_NUM       10
 8: #define MAX_FILE_BUF       1024
 9:
10: struct FILE {
11:     unsigned short name[MAX_FILE_NAME_LEN];
12: };
13:
14: extern struct FILE file_list[MAX_FILE_NUM];
15:
16: #endif
```

▼**リスト** 5.8　sample_5_1_ls/file.c

```
1: #include "file.h"
2:
3: struct FILE file_list[MAX_FILE_NUM];
```

そして、"ls"コマンドを追加した shell.c の内容はリスト 5.9 の通りです。

▼**リスト** 5.9　sample5_1_ls/shell.c

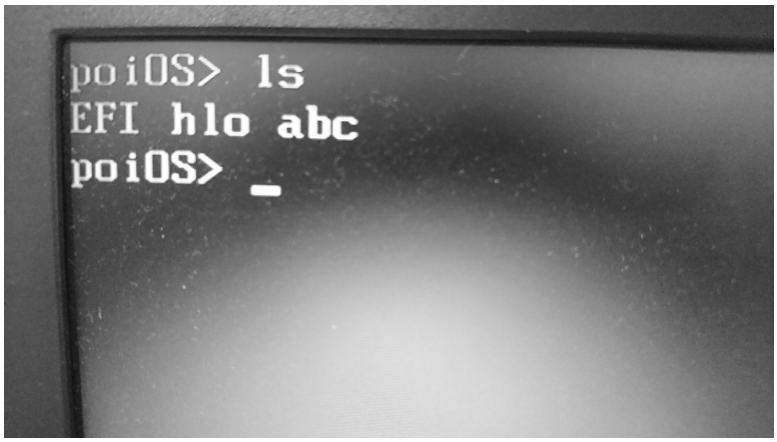
```
 1: /* ・・・省略・・・ */
 2:
 3: /* 追加（ここから） */
 4: int ls(void)
 5: {
 6:     unsigned long long status;
 7:     struct EFI_FILE_PROTOCOL *root;
 8:     unsigned long long buf_size;
 9:     unsigned char file_buf[MAX_FILE_BUF];
10:     struct EFI_FILE_INFO *file_info;
11:     int idx = 0;
12:     int file_num;
13:
14:     status = SFSP->OpenVolume(SFSP, &root);
15:     assert(status, L"SFSP->OpenVolume");
16:
17:     while (1) {
18:             buf_size = MAX_FILE_BUF;
19:             status = root->Read(root, &buf_size, (void *)file_buf);
20:             assert(status, L"root->Read");
21:             if (!buf_size) break;
22:
23:             file_info = (struct EFI_FILE_INFO *)file_buf;
24:             strncpy(file_list[idx].name, file_info->FileName,
25:                     MAX_FILE_NAME_LEN - 1);
26:             file_list[idx].name[MAX_FILE_NAME_LEN - 1] = L'\0';
27:             puts(file_list[idx].name);
28:             puts(L" ");
29:
30:             idx++;
31:     }
32:     puts(L"\r\n");
33:     file_num = idx;
34:
35:     root->Close(root);
36:
37:     return file_num;
38: }
39: /* 追加（ここまで） */
40:
41: void shell(void)
42: {
43:     unsigned short com[MAX_COMMAND_LEN];
44:     struct RECT r = {10, 10, 100, 200};
45:
46:     while (TRUE) {
47:             puts(L"poiOS> ");
48:             if (gets(com, MAX_COMMAND_LEN) <= 0)
49:                     continue;
50:
51:             if (!strcmp(L"hello", com))
52:                     puts(L"Hello UEFI!\r\n");
53:             /* ・・・省略・・・ */
54:             else if (!strcmp(L"ls", com))   /* 追加 */
55:                     ls();   /* 追加 */
56:             else
57:                     puts(L"Command not found.\r\n");
58:     }
```

```
59: }
```

リスト 5.9 では、ls 関数実行の都度 OpenVolume 関数でルートディレクトリを開いています。これは、Read で一通りファイル/ディレクトリエントリを取得しきってしまうと、再度取得するには一度 Close し、再度 OpenVolume する必要があるためです。取得したファイル/ディレクトリエントリをキャッシュしておくという考え方もありますが、ここでは新鮮な結果を返すために (また、簡単のために)、都度 OpenVolume し、Read しています。

また、リスト 5.9 の ls 関数では assert 関数を呼び出しています。assert 関数は引数で渡されたステータス値をチェックし、ステータス値が成功 (=0) 以外であれば、同じく引数で指定されたメッセージを出力して"`while(1);`"で固めます。なお、内部的にはステータスチェックとメッセージ表示は check_warn_error という関数に分かれていて、assert 関数は check_warn_error の戻り値に応じて"`while(1);`"で固めるだけです。詳しくは common.c のソースコードを見てみてください。

サンプルを実行する際は USB フラッシュメモリ直下にファイルをいくつか配置してみてください。例えば、"abc"と"hlo"の 2 つのファイルを置いてみると、図 5.1 の様に表示されます。



▲図 5.1　ファイル/ディレクトリー覧を表示している様子

5.3 GUI モードでファイル/ディレクトリ一覧を表示する

これまで GUI モードではファイルのアイコンに見立てた矩形が表示されるだけでした。ここでは、矩形内にファイル名を配置してみます。サンプルプログラムは"sample5_2_gui_ls"のディレクトリです。

処理の流れとしては、ファイル名のリストを表示した後、各ファイル名を矩形で囲みます。なお、簡単のためにファイル名は 3 文字とします。

まず、ファイルの付帯情報として矩形の位置・大きさとハイライト状態を追加します。変更箇所は file.h で、リスト 5.10 の通りです。

▼**リスト** 5.10　sample5_2_gui_ls/file.h

```
 1: /* ・・・省略・・・ */
 2: struct FILE {
 3:     struct RECT rect;       /* 追加 */
 4:     unsigned char is_highlight;     /* 追加 */
 5:     unsigned short name[MAX_FILE_NAME_LEN];
 6: };
 7: /* ・・・省略・・・ */
```

以上を踏まえて gui.c へ機能を追加するとリスト 5.11 の通りです。

▼**リスト** 5.11　sample5_2_gui_ls/gui.c

```
 1: #include "efi.h"
 2: #include "common.h"
 3: #include "file.h"
 4: #include "graphics.h"
 5: #include "shell.h"
 6: #include "gui.h"
 7:
 8: #define WIDTH_PER_CH       8       /* 追加 */
 9: #define HEIGHT_PER_CH      20      /* 追加 */
10:
11: /* ・・・省略・・・ */
12:
13: /* 追加（ここから） */
14: int ls_gui(void)
15: {
16:     int file_num;
17:     struct RECT t;
18:     int idx;
19:
20:     ST->ConOut->ClearScreen(ST->ConOut);
21:
22:     file_num = ls();
23:
```

```
24:     t.x = 0;
25:     t.y = 0;
26:     t.w = (MAX_FILE_NAME_LEN - 1) * WIDTH_PER_CH;
27:     t.h = HEIGHT_PER_CH;
28:     for (idx = 0; idx < file_num; idx++) {
29:             file_list[idx].rect.x = t.x;
30:             file_list[idx].rect.y = t.y;
31:             file_list[idx].rect.w = t.w;
32:             file_list[idx].rect.h = t.h;
33:             draw_rect(file_list[idx].rect, white);
34:             t.x += file_list[idx].rect.w + WIDTH_PER_CH;
35:
36:             file_list[idx].is_highlight = FALSE;
37:     }
38:
39:     return file_num;
40: }
41: /* 追加（ここまで） */
42:
43: void gui(void)
44: {
45:     unsigned long long status;
46:     struct EFI_SIMPLE_POINTER_STATE s;
47:     int px = 0, py = 0;
48:     unsigned long long waitidx;
49:     int file_num;   /* 追加 */
50:     int idx;        /* 追加 */
51:
52:     SPP->Reset(SPP, FALSE);
53:     file_num = ls_gui();    /* 追加 */
54:
55:     while (TRUE) {
56:             ST->BootServices->WaitForEvent(1, &(SPP->WaitForInput), &waitidx);
57:             status = SPP->GetState(SPP, &s);
58:             if (!status) {
59:                     /* マウスカーソル座標更新 */
60:                     /* ・・・省略・・・ */
61:
62:                     /* マウスカーソル描画 */
63:                     put_cursor(px, py);
64:
65:                     /* 追加（ここから） */
66:                     /* ファイルアイコン処理ループ */
67:                     for (idx = 0; idx < file_num; idx++) {
68:                             if (is_in_rect(px, py, file_list[idx].rect)) {
69:                                     if (!file_list[idx].is_highlight) {
70:                                             draw_rect(file_list[idx].rect,
71:                                                     yellow);
72:                                             file_list[idx].is_highlight = TRUE;
73:                                     }
74:                             } else {
75:                                     if (file_list[idx].is_highlight) {
76:                                             draw_rect(file_list[idx].rect,
77:                                                     white);
78:                                             file_list[idx].is_highlight =
79:                                                     FALSE;
80:                                     }
81:                             }
```

```
82:                     }
83:                     /* 追加（ここまで）*/
84:             }
85:     }
86: }
```

リスト 5.11 で追加した ls_gui 関数は、shell.c の ls 関数をラップした関数です。画面クリアした後、ls 関数を実行してファイル/ディレクトリのリストを画面に表示し、その後、矩形を描きます。また、ls_gui 関数は、描画する矩形の位置と大きさ、ハイライト状態を file_list 配列のメンバへ設定します。

リスト 5.11 の gui 関数へ追加した"ファイルアイコン処理ループ"では、マウス操作に応じた各ファイルの処理を記述しています。ここでは、マウスカーソルがアイコンの上に乗ったらアイコンの枠をハイライト色へ変更する事を行っています。

サンプルの実行結果は図 5.2 の通りです。

▲**図 5.2　ファイルリスト対応版 GUI モード実行の様子**

5.4　ファイルを読んでみる (cat)

ファイルのリストを取得できたので、次はファイルの中身を読んでみます。まずはシェルコマンドとして"cat(もどき)"を作ってみます。サンプルのディレクトリは"sample5_3_cat"です。

なお、ここでは UEFI ファームウェアをどのように呼び出せばファイルを読めるのかが分かればよいので、簡単のため、cat コマンドが読み出すファイル名は固定とします。

ファイルリスト取得の際にはディレクトリの EFI_FILE_PROTOCOL に対して Read 関数を呼び出すことでファイル名を取得していました。ファイルの EFI_FILE_PROTOCOL に対して Read 関数を呼び出すとファイルの内容を読むことができます (仕様書"12.5 EFI File Protocol(P.504)")。

では、ファイルの EFI_FILE_PROTOCOL はどうやって取得すればよいのかというと、ディレクトリの EFI_FILE_PROTOCOL に対してファイル名を指定して Open 関

数を呼び出すことで取得できます (仕様書"12.5 EFI File Protocol(P.499)")。Open 関数の定義はリスト 5.12 の通りです。

▼**リスト** 5.12　EFI_FILE_PROTOCOL **の** Open **関数の定義**

```
unsigned long long (*Open)(struct EFI_FILE_PROTOCOL *This,
                           struct EFI_FILE_PROTOCOL **NewHandle,
                           unsigned short *FileName,
                           unsigned long long OpenMode,
                           unsigned long long Attributes);
```

引数の意味は以下の通りです。

struct EFI_FILE_PROTOCOL **NewHandle

新しく開いた EFI_FILE_PROTOCOL を格納するポインタへのアドレス。

unsigned short *FileName

ファイル名。

unsigned long long OpenMode

ファイルを開くモード (後述)。

unsigned long long Attributes

属性ビット。ファイル作成時にファイルの属性ビットへ設定する値を指定。本書では使用しない。

OpenMode へ指定できる定数の定義はリスト 5.13 の通りです。

▼**リスト** 5.13　Open **関数の** OpenMode **へ指定できる定数**

```
#define EFI_FILE_MODE_READ     0x0000000000000001
#define EFI_FILE_MODE_WRITE    0x0000000000000002
#define EFI_FILE_MODE_CREATE   0x8000000000000000
```

なお、OpenMode へ指定できる組み合わせは以下のいずれかです。

- READ
- READ | WRITE
- READ | WRITE | CREATE

以上を踏まえてまとめると、"abc"というファイル名のファイルを読む際の処理の流れは以下の通りです。

1. EFI_SIMPLE_FILE_SYSTEM_PROTOCOL の OpenVolume 関数でボリュームを開く (ルートディレクトリの EFI_FILE_PROTOCOL を取得)

2. 1. の EFI_FILE_PROTOCOL の Open 関数をファイル名に"abc"を指定して実行 ("abc"ファイルの EFI_FILE_PROTOCOL を取得)
3. 2. の EFI_FILE_PROTOCOL の Read 関数を呼び出す (ファイルの内容を取得)

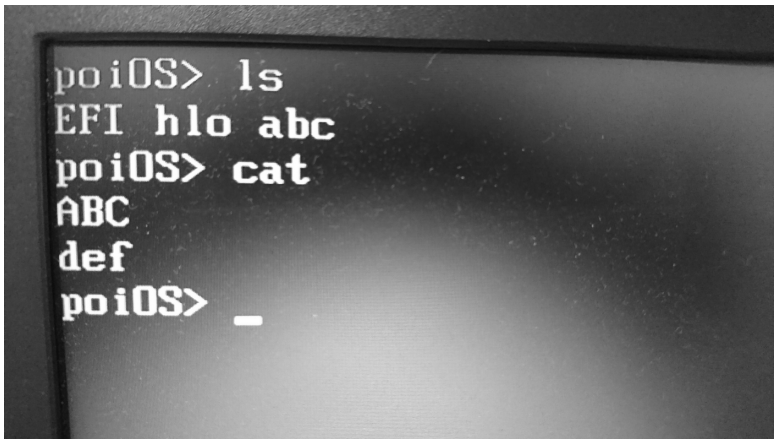
cat コマンドの実装はリスト 5.14 の通りです。処理の流れとしては上記の 1.〜3. の通りで、最後に Close 処理を行っています。

▼**リスト** 5.14　sample5_3_cat/shell.c

```
 1: /* ・・・省略・・・ */
 2:
 3: /* 追加（ここから） */
 4: void cat(unsigned short *file_name)
 5: {
 6:     unsigned long long status;
 7:     struct EFI_FILE_PROTOCOL *root;
 8:     struct EFI_FILE_PROTOCOL *file;
 9:     unsigned long long buf_size = MAX_FILE_BUF;
10:     unsigned short file_buf[MAX_FILE_BUF / 2];
11:
12:     status = SFSP->OpenVolume(SFSP, &root);
13:     assert(status, L"SFSP->OpenVolume");
14:
15:     status = root->Open(root, &file, file_name, EFI_FILE_MODE_READ, 0);
16:     assert(status, L"root->Open");
17:
18:     status = file->Read(file, &buf_size, (void *)file_buf);
19:     assert(status, L"file->Read");
20:
21:     puts(file_buf);
22:
23:     file->Close(file);
24:     root->Close(root);
25: }
26: /* 追加（ここまで） */
27:
28: void shell(void)
29: {
30:     unsigned short com[MAX_COMMAND_LEN];
31:     struct RECT r = {10, 10, 100, 200};
32:
33:     while (TRUE) {
34:             puts(L"poiOS> ");
35:             if (gets(com, MAX_COMMAND_LEN) <= 0)
36:                     continue;
37:
38:             if (!strcmp(L"hello", com))
39:                     puts(L"Hello UEFI!\r\n");
40:             /* ・・・省略・・・ */
41:             else if (!strcmp(L"cat", com))  /* 追加 */
42:                     cat(L"abc");    /* 追加 */
43:             else
44:                     puts(L"Command not found.\r\n");
45:     }
46: }
```

サンプルの実行の様子を図 5.3 に示します。なお、UEFI ファームウェアの機能で表示できる Unicode ファイルは以下の nkf コマンドで作成可能です。

```
$ nkf -w16L0 orig.txt > unicode.txt
```



▲**図** 5.3　cat **コマンド実行の様子**

5.5　GUI モードへテキストファイル閲覧機能追加

ファイルの内容を取得する方法が分かったので、GUI モードを拡張してみます。ここでは以下の仕様とします。サンプルのディレクトリは"sample5_4_gui_cat"です。

1. ファイルのアイコン (矩形) 内を左クリックすると閲覧モード開始
2. 閲覧モードでは、「(1) 画面クリア」、「(2) ファイル内容を Unicode で表示」を行う
3. ESC キー押下で閲覧モード終了

主に追加・変更するソースコードは gui.c です。gui.c のソースコードをリスト 5.15 に示します。

▼**リスト** 5.15　sample5_4_gui_cat/gui.c

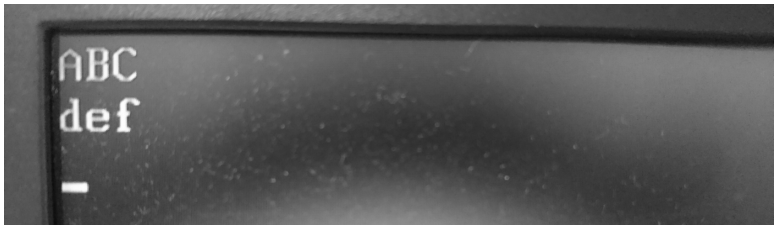
```
 1: /* ・・・省略・・・ */
 2:
 3: /* 追加（ここから） */
 4: void cat_gui(unsigned short *file_name)
 5: {
 6:     ST->ConOut->ClearScreen(ST->ConOut);
 7:
 8:     cat(file_name);
 9:
10:     while (getc() != SC_ESC);
11: }
12: /* 追加（ここまで） */
13:
14: void gui(void)
15: {
16:     unsigned long long status;
17:     struct EFI_SIMPLE_POINTER_STATE s;
18:     int px = 0, py = 0;
19:     unsigned long long waitidx;
20:     int file_num;
21:     int idx;
22:     unsigned char prev_lb = FALSE;  /* 追加 */
23:
24:     SPP->Reset(SPP, FALSE);
25:     file_num = ls_gui();
26:
27:     while (TRUE) {
28:             ST->BootServices->WaitForEvent(1, &(SPP->WaitForInput), &waitidx);
29:             status = SPP->GetState(SPP, &s);
30:             if (!status) {
31:                     /* マウスカーソル座標更新 */
32:                     /* ・・・省略・・・ */
33:
34:                     /* マウスカーソル描画 */
35:                     put_cursor(px, py);
36:
37:                     /* ファイルアイコン処理ループ */
38:                     for (idx = 0; idx < file_num; idx++) {
39:                             if (is_in_rect(px, py, file_list[idx].rect)) {
40:                                     if (!file_list[idx].is_highlight) {
41:                                             draw_rect(file_list[idx].rect,
42:                                                       yellow);
43:                                             file_list[idx].is_highlight = TRUE;
44:                                     }
45:                                     /* 追加（ここから） */
46:                                     if (prev_lb && !s.LeftButton) {
47:                                             cat_gui(file_list[idx].name);
48:                                             file_num = ls_gui();
49:                                     }
50:                                     /* 追加（ここまで） */
51:                             } else {
52:                                     if (file_list[idx].is_highlight) {
53:                                             draw_rect(file_list[idx].rect,
54:                                                       white);
55:                                             file_list[idx].is_highlight =
56:                                                     FALSE;
57:                                     }
58:                             }
```

```
59:                     }
60:
61:                     /* 追加（ここから）*/
62:                     /* マウスの左ボタンの前回の状態を更新 */
63:                     prev_lb = s.LeftButton;
64:                     /* 追加（ここまで）*/
65:             }
66:     }
67: }
```

リスト 5.15 に関して、gui 関数へはマウスクリック時の処理を追加しています。マウスのボタンから指を離した瞬間をマウスクリックとするために、マウスボタンの前回状態を prev_lb という変数へ格納しています。そして、ファイルクリック時にファイル名を指定して cat_gui 関数を呼び出す処理を、"ファイルアイコン処理ループ"内へ追加しています。cat_gui 関数は cat 関数をラップしている関数で、上述の閲覧モードを実現しています。

サンプルを実行し、gui コマンドで gui モードを起動すると図 5.4 のように表示されます。そして、ファイルをクリックすると図 5.5 のようにファイルの内容が画面に表示されます。

▲図 5.4　GUI モードでファイル一覧が表示される様子



▲図 5.5　ファイルの内容が表示される様子

getc でスキャンコードも返せるよう修正 (オガム文字の領域を使う)

リスト 5.15 の cat_gui 関数では、ESC キーで閲覧モードを終了できるように、getc 関数が"SC_ESC(ESC キーのスキャンコード)"を返すまで待機しています。実はこのサンプルコード時点で、common.c の getc 関数をスキャンコード「も」返すことができるように修正しています (リスト 5.16)。

▼**リスト** 5.16　sample_5_4_gui_cat/common_c(getc **関数**)

```
 1: unsigned short getc(void)
 2: {
 3:     struct EFI_INPUT_KEY key;
 4:     unsigned long long waitidx;
 5:
 6:     ST->BootServices->WaitForEvent(1, &(ST->ConIn->WaitForKey),
 7:                                    &waitidx);
 8:     while (ST->ConIn->ReadKeyStroke(ST->ConIn, &key));
 9:
10:     /* 変更 */
11:     return (key.UnicodeChar) ? key.UnicodeChar
12:             : (key.ScanCode + SC_OFS);
13: }
```

リスト 5.16 では、key.UnicodeChar が 0(押下されたキーが Unicode 範囲外) である時、key.ScanCode に下駄 (SC_OFS) を履かせた値を返しています。これは、スキャンコードの値の範囲 (0x00〜0x17) が 2 バイト Unicode の値の範囲 (0x0000〜0xffff) と被っているためで、Unicode の使わなそうな範囲をスキャンコードとして使っています。
ここでは 0x1680〜0x1697 をスキャンコードとしています。Unicode のこの範囲は「オガム文字」という文字を扱う範囲らしく、「アイルランドを中心に発見された古アイルランド語の碑文に記されている文字で、中世初期に碑文用に用いられたといわれています。」とのことです ([a])。
そのため、SC_OFS と SC_ESC は common.h でリスト 5.17 の様に定義しています。

▼**リスト** 5.17　SC_OFS **と** SC_ESC **の定義**

```
#define SC_OFS 0x1680
#define SC_ESC (SC_OFS + 0x0017)
```

[a] http://www.asahi-net.or.jp/~ax2s-kmtn/ref/unicode/european.html

5.6 ファイルへ書き込んでみる (edit)

ファイルを読むことができたので、次はファイルへの書き込みをしてみます。ここでは「画面上に記述した内容でファイルを上書きする」コマンドとして edit コマンドを追加します。サンプルのディレクトリは"sample5_5_edit"です。

EFI_FILE_PROTOCOL の Write 関数でファイルへの書き込みが行えます。Write 関数の定義はリスト 5.18 の通りです。

▼**リスト** 5.18　EFI_FILE_PROTOCOL **の** Write **関数の定義**

```
unsigned long long (*Write)(struct EFI_FILE_PROTOCOL *This,
                            unsigned long long *BufferSize,
                            void *Buffer);
```

引数の意味は以下の通りです。

unsigned long long *BufferSize
: 第 3 引数"Buffer"のサイズ (バイト指定) を格納した変数のポインタを指定する。Write 関数実行後、このポインタで参照された変数には書き込んだデータサイズが格納される。

void *Buffer
: 書き込むデータを格納したバッファ。

なお、Write 関数を実行しても、ただちにディスクへ反映されるわけではありません。キャッシュされている Write 結果をディスクへ反映させるには Flush 関数を実行する必要があります。Flush 関数の定義はリスト 5.19 の通りです。

▼**リスト** 5.19　EFI_FILE_PROTOCOL **の** Flush **関数の定義**

```
unsigned long long (*Flush)(struct EFI_FILE_PROTOCOL *This);
```

以上を踏まえて edit コマンドを追加します。変更後の shell.c をリスト 5.20 に示します。

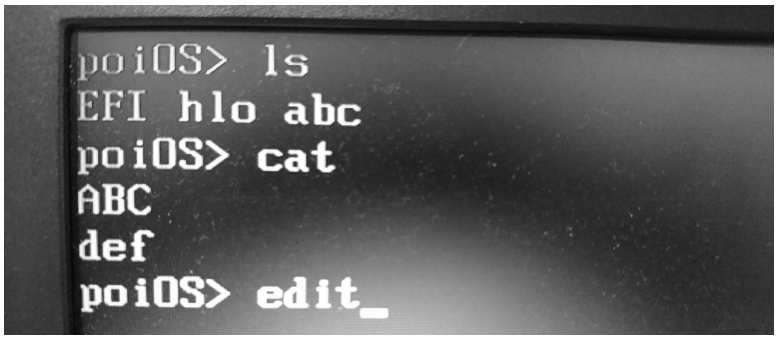
▼**リスト** 5.20　sample5_5_edit/shell.c

```
 1: /* ・・・省略・・・ */
 2:
 3: /* 追加（ここから） */
 4: void edit(unsigned short *file_name)
 5: {
 6:     unsigned long long status;
 7:     struct EFI_FILE_PROTOCOL *root;
 8:     struct EFI_FILE_PROTOCOL *file;
 9:     unsigned long long buf_size = MAX_FILE_BUF;
10:     unsigned short file_buf[MAX_FILE_BUF / 2];
11:     int i = 0;
12:     unsigned short ch;
13:
14:     ST->ConOut->ClearScreen(ST->ConOut);
15:
16:     while (TRUE) {
17:             ch = getc();
18:
19:             if (ch == SC_ESC)
20:                     break;
21:
22:             putc(ch);
23:             file_buf[i++] = ch;
24:
25:             if (ch == L'\r') {
26:                     putc(L'\n');
27:                     file_buf[i++] = L'\n';
28:             }
29:     }
30:     file_buf[i] = L'\0';
31:
32:     status = SFSP->OpenVolume(SFSP, &root);
33:     assert(status, L"SFSP->OpenVolume");
34:
35:     status = root->Open(root, &file, file_name,
36:                         EFI_FILE_MODE_READ | EFI_FILE_MODE_WRITE, 0);
37:     assert(status, L"root->Open");
38:
39:     status = file->Write(file, &buf_size, (void *)file_buf);
40:     assert(status, L"file->Write");
41:
42:     file->Flush(file);
43:
44:     file->Close(file);
45:     root->Close(root);
46: }
47: /* 追加（ここまで） */
48:
49: void shell(void)
50: {
51:     /* ・・・省略・・・ */
52:     while (TRUE) {
53:             /* ・・・省略・・・ */
54:             if (!strcmp(L"hello", com))
55:                     puts(L"Hello UEFI!\r\n");
56:             /* ・・・省略・・・ */
57:             else if (!strcmp(L"edit", com)) /* 追加 */
58:                     edit(L"abc");   /* 追加 */
```

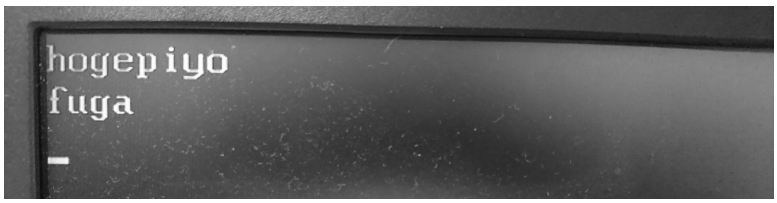
```
59:             else
60:                     puts(L"Command not found.\r\n");
61:     }
62: }
```

リスト 5.20 の edit 関数について、"while (TRUE)"のコードブロックが上書き用のデータを作成している処理で、ESC キーでループを抜けます。その後、ファイルへの書き込み処理を行います。また、shell 関数内に関して、edit コマンドも cat コマンドと同様に対象のファイル名は固定で、"abc"というファイル名とします。

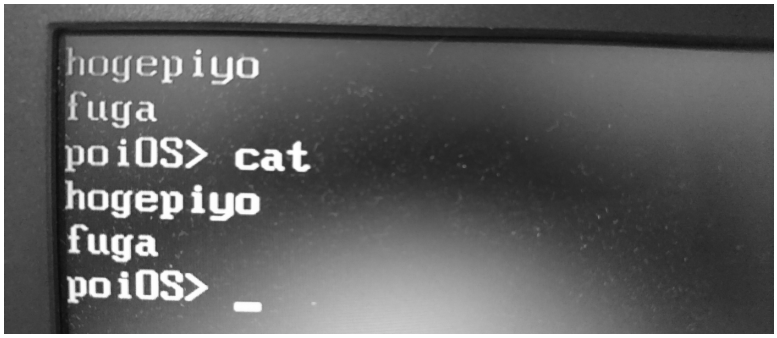
サンプルの実行結果は図 5.6、図 5.7、図 5.8 の通りです。



▲図 5.6　edit コマンドを実行する様子



▲図 5.7　上書き用データを作成している様子



▲図 5.8　上書きが反映されていることを確認する様子

5.7　GUI モードへテキストファイル上書き機能追加

shell.c へ追加した edit 関数を使うように以下の仕様で GUI モードを拡張します。サンプルのディレクトリは"sample5_6_gui_edit"です。

1. ファイルアイコン、あるいは何もない箇所を右クリックで上書きモード (edit 関数) 開始
2. ESC キー押下で上書きモード終了

なお、何もない箇所を右クリックした場合はファイルを新規に作成するようにしてみます。ファイルを新規に作成するには、Open 関数実行時に第 3 引数の OpenMode へ EFI_FILE_MODE_CREATE を追加で指定するだけです。

主な変更箇所は gui.c と shell.c です。まず gui.c をリスト 5.21 に示します。

▼リスト 5.21　sample5_6_gui_edit/gui.c

```
 1: /* ・・・省略・・・ */
 2:
 3: void gui(void)
 4: {
 5:     unsigned long long status;
 6:     struct EFI_SIMPLE_POINTER_STATE s;
 7:     int px = 0, py = 0;
 8:     unsigned long long waitidx;
 9:     int file_num;
10:     int idx;
11:     unsigned char prev_lb = FALSE;
12:     unsigned char prev_rb = FALSE, executed_rb;     /* 追加 */
13:
```

```
14:     SPP->Reset(SPP, FALSE);
15:     file_num = ls_gui();
16:
17:     while (TRUE) {
18:             ST->BootServices->WaitForEvent(1, &(SPP->WaitForInput), &waitidx);
19:             status = SPP->GetState(SPP, &s);
20:             if (!status) {
21:                     /* マウスカーソル座標更新 */
22:                     /* ・・・省略・・・ */
23:
24:                     /* マウスカーソル描画 */
25:                     put_cursor(px, py);
26:
27:                     /* 右クリックの実行済フラグをクリア */  /* 追加 */
28:                     executed_rb = FALSE;    /* 追加 */
29:
30:                     /* ファイルアイコン処理ループ */
31:                     for (idx = 0; idx < file_num; idx++) {
32:                             if (is_in_rect(px, py, file_list[idx].rect)) {
33:                                     /* ・・・省略・・・ */
34:                                     if (prev_lb && !s.LeftButton) {
35:                                             cat_gui(file_list[idx].name);
36:                                             file_num = ls_gui();
37:                                     }
38:                                     /* 追加（ここから） */
39:                                     if (prev_rb && !s.RightButton) {
40:                                             edit(file_list[idx].name);
41:                                             file_num = ls_gui();
42:                                             executed_rb = TRUE;
43:                                     }
44:                                     /* 追加（ここまで） */
45:                             } else {
46:                                     /* ・・・省略・・・ */
47:                             }
48:                     }
49:
50:                     /* 追加（ここから） */
51:                     /* ファイル新規作成・編集 */
52:                     if ((prev_rb && !s.RightButton) && !executed_rb) {
53:                             /* アイコン外を右クリックした場合 */
54:                             dialogue_get_filename(file_num);
55:                             edit(file_list[file_num].name);
56:                             ST->ConOut->ClearScreen(ST->ConOut);
57:                             file_num = ls_gui();
58:                     }
59:                     /* 追加（ここまで） */
60:
61:                     /* マウスの左右ボタンの前回の状態を更新 */
62:                     prev_lb = s.LeftButton;
63:                     prev_rb = s.RightButton;        /* 追加 */
64:             }
65:     }
66: }
```

リスト 5.21 では右クリック時の処理を追加しています。"ファイルアイコン処理ループ"内に追加しているのがファイルを右クリックした場合の処理で、"ファイル新規作成・

編集"のコードブロックがアイコン外を右クリックした場合の処理です。"ファイルアイコン処理ループ"内で右クリックを処理済みであることを示すために"executed_rb"変数を用意しています。

"ファイル新規作成・編集"時の dialogue_get_filename 関数は shell.c へ追加しています。shell.c へは他に、edit 関数実行時の Open へ EFI_FILE_MODE_CREATE を追加しています。変更後の shell.c をリスト 5.22 に示します。

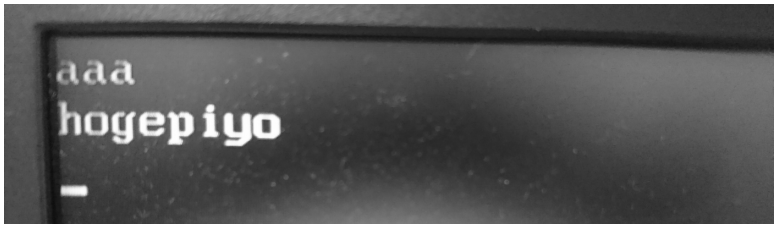
▼**リスト** 5.22　sample5_6_gui_edit/shell.c

```
 1: /* ・・・省略・・・ */
 2:
 3: /* 追加（ここから） */
 4: void dialogue_get_filename(int idx)
 5: {
 6:     int i;
 7:
 8:     ST->ConOut->ClearScreen(ST->ConOut);
 9:
10:     puts(L"New File Name: ");
11:     for (i = 0; i < MAX_FILE_NAME_LEN; i++) {
12:             file_list[idx].name[i] = getc();
13:             if (file_list[idx].name[i] != L'\r')
14:                     putc(file_list[idx].name[i]);
15:             else
16:                     break;
17:     }
18:     file_list[idx].name[i] = L'\0';
19: }
20: /* 追加（ここまで） */
21:
22: /* ・・・省略・・・ */
23:
24: void edit(unsigned short *file_name)
25: {
26:     /* ・・・省略・・・ */
27:     status = root->Open(root, &file, file_name,
28:                         EFI_FILE_MODE_READ | EFI_FILE_MODE_WRITE | \
29:                         EFI_FILE_MODE_CREATE, 0);   /* 追加 */
30:     assert(status, L"root->Open");
31:     /* ・・・省略・・・ */
32: }
33:
34: /* ・・・省略・・・ */
```

サンプルの実行結果は図 5.9、図 5.10、図 5.11 の通りです。ここではファイル新規作成を試しています。

▲図 5.9　アイコン外を右クリック

▲図 5.10　ファイル名入力画面



▲図 5.11　ファイル内容入力

第 6 章

poiOS の機能拡張例

ここまで OS っぽいものを作る事を目指して UEFI ファームウェアの機能の呼び出し方を説明してきました。題材として作ってきた poiOS はまだまだ不完全なものなので、ぜひご自身で機能拡張を行ってみてください (あるいはフルスクラッチで作ってみてください)。

ここでは機能拡張の例を紹介します。この章で説明する全ての拡張を行ったサンプルを"sample_poios"のディレクトリに格納しています。

6.1 画像を表示してみる

フレームバッファのアドレスとピクセルフォーマットが分かっているので、フレームバッファへ書き込むことで画像を表示することもできます。

blt 関数を追加

まず、指定されたバッファに格納されているピクセルデータをフレームバッファへ書き込む関数"blt(BLock Transfer)"を graphics.c へ追加します (リスト 6.1)。

▼**リスト** 6.1　sample_poios/graphics.c(blt **関数**)

```
 1: void blt(unsigned char img[], unsigned int img_width, unsigned int img_height)
 2: {
 3:     unsigned char *fb;
 4:     unsigned int i, j, k, vr, hr, ofs = 0;
 5:
 6:     fb = (unsigned char *)GOP->Mode->FrameBufferBase;
 7:     vr = GOP->Mode->Info->VerticalResolution;
 8:     hr = GOP->Mode->Info->HorizontalResolution;
 9:
10:     for (i = 0; i < vr; i++) {
```

```
11:             if (i >= img_height)
12:                     break;
13:             for (j = 0; j < hr; j++) {
14:                     if (j >= img_width) {
15:                             fb += (hr - img_width) * 4;
16:                             break;
17:                     }
18:                     for (k = 0; k < 4; k++)
19:                             *fb++ = img[ofs++];
20:             }
21:     }
22: }
```

リスト 6.1 は単に引数 img のバイト列をピクセルデータとしてフレームバッファへ書き込んでいるだけです。なお、画像は常に原点 (0,0) から描画されます。

シェルへ画像閲覧コマンド追加 (view)

次に、blt 関数を使用して画像ファイルを画面へ描画するコマンド"view"を shell.c へ追加します (リスト 6.2)。なお、「画像ファイル」は UEFI に対応したピクセルフォーマットの生バイナリとします。

▼**リスト** 6.2　sample_poios/shell.c

```
 1: /* ・・・省略・・・ */
 2: #define MAX_COMMAND_LEN    100
 3:
 4: #define MAX_IMG_BUF        4194304 /* 4MB */       /* 追加 */
 5:
 6: unsigned char img_buf[MAX_IMG_BUF]; /* 追加 */
 7: /* ・・・省略・・・ */
 8: void view(unsigned short *img_name)
 9: {
10:     unsigned long long buf_size = MAX_IMG_BUF;
11:     unsigned long long status;
12:     struct EFI_FILE_PROTOCOL *root;
13:     struct EFI_FILE_PROTOCOL *file;
14:     unsigned int vr = GOP->Mode->Info->VerticalResolution;
15:     unsigned int hr = GOP->Mode->Info->HorizontalResolution;
16:
17:     status = SFSP->OpenVolume(SFSP, &root);
18:     assert(status, L"error: SFSP->OpenVolume");
19:
20:     status = root->Open(root, &file, img_name, EFI_FILE_MODE_READ,
21:                         EFI_FILE_READ_ONLY);
22:     assert(status, L"error: root->Open");
23:
24:     status = file->Read(file, &buf_size, (void *)img_buf);
25:     if (check_warn_error(status, L"warning:file->Read"))
26:             blt(img_buf, hr, vr);
27:
```

```
28:     while (getc() != SC_ESC);
29:
30:     status = file->Close(file);
31:     status = root->Close(root);
32: }
33:
34: void shell(void)
35: {
36:     /* ・・・省略・・・ */
37:
38:     while (!is_exit) {
39:             /* ・・・省略・・・ */
40:             else if (!strcmp(L"view", com)) {      /* 追加 */
41:                     view(L"img");   /* 追加 */
42:                     ST->ConOut->ClearScreen(ST->ConOut);       /* 追加 */
43:             } else
44:                     puts(L"Command not found.\r\n");
45:     }
46: }
```

リスト 6.2 について、view コマンド実行時、"img"というファイル名を指定して view 関数を実行します。view 関数は引数で指定されたファイル名のファイルを開き、読み出したバイナリデータをバッファ"img_buf"へ格納し blt 関数へ渡します。その後、ESC キー押下まで待機し、ESC キーが押下されると view 関数を抜けます。cat 関数や edit 関数もそうですが、shell の各コマンドの関数は「画面を散らかしたまま返す」が仕様なので、view 関数も画面をクリアせずに呼び出し元へ戻ります。そのため、shell 関数側で view 関数実行後、ClearScreen 関数を呼び出しています。

BGRA32 ビットへの画像変換方法

BGRA32 ビット (各 8 ビット) の生の画像データ (ヘッダ等が無いバイナリ) への変換は、ImageMagick の convert コマンドで行えます。コマンド例は以下の通りです。

```
$ convert hoge.png -depth 8 yux.bgra
```

なお、現状の view コマンドは画像のスクロールやリサイズは行うことができません。そのため、筆者は、自身の PC の UEFI ファームウェアがデフォルトで認識している解像度 (Width=640px) に合わせるように画像をリサイズしています。コマンド例は以下の通りです。

```
$ convert hoge.png -resize 640x -depth 8 yux.bgra
```

GUI モードへ画像ファイル表示機能追加

最後に、gui.c を拡張し、画像ファイルをクリックすると view 関数を使用して画面へ画像を描画するようにします (リスト 6.3)。なお、poiOS においては「"i"で始まるファイル名のファイル」を画像ファイルとすることにします。

▼**リスト** 6.3　sample_poios/gui.c(gui **関数内**)

```
 1: /* ファイルアイコン処理ループ */
 2: for (idx = 0; idx < file_num; idx++) {
 3:     if (is_in_rect(px, py, file_list[idx].rect)) {
 4:             /* ・・・省略・・・ */
 5:             if (prev_lb && !s.LeftButton) {
 6:                     if (file_list[idx].name[0] != L'i')     /* 追加 */
 7:                             cat_gui(file_list[idx].name);
 8:                     else    /* 追加 */
 9:                             view(file_list[idx].name);      /* 追加 */
10:                     file_num = ls_gui();
11:             }
12:             /* ・・・省略・・・ */
13:     }
14: }
```

リスト 6.3 に示す通り、gui.c の変更箇所は gui 関数内の"ファイルアイコン処理ループ"内だけです。左クリック時に cat_gui 関数を呼び出していたところを、ファイル名が"i"で始まる場合には view 関数を呼び出すようにしています。

blt 関数は UEFI の仕様に存在します

実は仕様上、EFI_GRAPHICS_OUTPUT_PROTOCOL 内に Blt という関数が存在します (仕様書"11.9.1 Blt Buffer(P.474)")。UEFI の Blt 関数はバッファのデータをフレームバッファへ書き込む (EfiBltBufferToVideo) だけでなく、フレームバッファのデータをバッファへ格納する (EfiBltVideoToBltBuffer)、フレームバッファのある矩形領域内のデータを別の座標へコピーする (EfiBltVideoToVideo) 等ができます。
ただし、筆者が動作確認に使用している Lenovo ThinkPad E450(UEFI バージョン 2.3.1) ではこの機能を使うことができず[a]、今回は自前で実装しました。なお、QEMU(OVMF、UEFI バージョン 2.3.1) では動作しました。

[a] ステータスは Success を返すが、画面には何も描画されていないという状況でした。

6.2 GUI モードとシェルの終了機能を追加する

シェル終了機能追加

exit コマンドで終了できるようにしてみます。
変更箇所は shell.c の shell 関数内だけです (リスト 6.4)。

▼**リスト** 6.4　sample_poios/shell.c(shell **関数へ** exit **機能追加**)

```
1: void shell(void)
2: {
3:     unsigned short com[MAX_COMMAND_LEN];
4:     struct RECT r = {10, 10, 100, 200};
5:     unsigned char is_exit = FALSE;  /* 追加 */
6:
7:     while (!is_exit) {     /* 変更 */
8:             /* ・・・省略・・・ */
9:             } else if (!strcmp(L"exit", com))      /* 追加 */
```

```
10:                     is_exit = TRUE; /* 追加 */
11:             else
12:                     puts(L"Command not found.\r\n");
13:     }
14: }
```

GUI モード終了機能追加

右上に [X] ボタンを配置し、このボタンをクリックすることで GUI モードを終了できるようにしてみます。

変更するソースコードは gui.c のみです。まず、[X] ボタンを配置する put_exit_button 関数と、マウスカーソルに対して再描画やクリック判定を行う update_exit_button 関数を追加します (リスト 6.5)。

▼**リスト** 6.5　sample_poios/gui.c(put_exit_button と update_exit_button)

```
 1: /* ・・・省略・・・ */
 2: #define EXIT_BUTTON_WIDTH  20
 3: #define EXIT_BUTTON_HEIGHT 20
 4: /* ・・・省略・・・ */
 5: struct FILE rect_exit_button;
 6: /* ・・・省略・・・ */
 7: void put_exit_button(void)
 8: {
 9:     unsigned int hr = GOP->Mode->Info->HorizontalResolution;
10:     unsigned int x;
11:
12:     rect_exit_button.rect.x = hr - EXIT_BUTTON_WIDTH;
13:     rect_exit_button.rect.y = 0;
14:     rect_exit_button.rect.w = EXIT_BUTTON_WIDTH;
15:     rect_exit_button.rect.h = EXIT_BUTTON_HEIGHT;
16:     rect_exit_button.is_highlight = FALSE;
17:     draw_rect(rect_exit_button.rect, white);
18:
19:     /* EXIT ボタン内の各ピクセルを走査（バッテンを描く） */
20:     for (x = 3; x < rect_exit_button.rect.w - 3; x++) {
21:             draw_pixel(x + rect_exit_button.rect.x, x, white);
22:             draw_pixel(x + rect_exit_button.rect.x,
23:                     rect_exit_button.rect.w - x, white);
24:     }
25: }
26:
27: unsigned char update_exit_button(int px, int py, unsigned char is_clicked)
28: {
29:     unsigned char is_exit = FALSE;
30:
31:     if (is_in_rect(px, py, rect_exit_button.rect)) {
32:             if (!rect_exit_button.is_highlight) {
33:                     draw_rect(rect_exit_button.rect, yellow);
34:                     rect_exit_button.is_highlight = TRUE;
```

```
35:             }
36:             if (is_clicked)
37:                     is_exit = TRUE;
38:     } else {
39:             if (rect_exit_button.is_highlight) {
40:                     draw_rect(rect_exit_button.rect, white);
41:                     rect_exit_button.is_highlight = FALSE;
42:             }
43:     }
44:
45:     return is_exit;
46: }
```

リスト 6.5 について、put_exit_button 関数では [X] ボタンを画面に描画すると同時に、グローバル変数"struct FILE rect_exit_button"へボタンの座標・大きさ・ハイライト状態を格納しています。また、update_exit_button 関数は、ハイライト状態を更新するのに加え、ボタンクリック判定結果を TRUE/FALSE で返します。

次に、これらの関数を使用するように gui 関数へ機能追加を行います (リスト 6.6)。

▼**リスト** 6.6　sample_poios/gui.c(gui **関数へ終了機能追加**)

```
 1: void gui(void)
 2: {
 3:     /* ・・・省略・・・ */
 4:     unsigned char is_exit = FALSE;  /* 追加 */
 5:
 6:     SPP->Reset(SPP, FALSE);
 7:     file_num = ls_gui();
 8:     put_exit_button();      /* 追加 */
 9:
10:     while (!is_exit) {     /* 変更 */
11:             ST->BootServices->WaitForEvent(1, &(SPP->WaitForInput), &waitidx);
12:             status = SPP->GetState(SPP, &s);
13:             if (!status) {
14:                     /* ・・・省略・・・ */
15:
16:                     /* ファイルアイコン処理ループ */
17:                     for (idx = 0; idx < file_num; idx++) {
18:                             if (is_in_rect(px, py, file_list[idx].rect)) {
19:                                     /* ・・・省略・・・ */
20:                                     if (prev_lb && !s.LeftButton) {
21:                                             /* ・・・省略・・・ */
22:                                             file_num = ls_gui();
23:                                             put_exit_button();  /* 追加 */
24:                                     }
25:                                     if (prev_rb && !s.RightButton) {
26:                                             edit(file_list[idx].name);
27:                                             file_num = ls_gui();
28:                                             put_exit_button();  /* 追加 */
29:                                             executed_rb = TRUE;
30:                                     }
31:                             } else {
```

```
32:                                     /* ・・・省略・・・ */
33:                             }
34:                     }
35:
36:                     /* ファイル新規作成・編集 */
37:                     if ((prev_rb && !s.RightButton) && !executed_rb) {
38:                             /* アイコン外を右クリックした場合 */
39:                             dialogue_get_filename(file_num);
40:                             edit(file_list[file_num].name);
41:                             ST->ConOut->ClearScreen(ST->ConOut);
42:                             file_num = ls_gui();
43:                             put_exit_button();      /* 追加 */
44:                     }
45:
46:                     /* 追加（ここから） */
47:                     /* 終了ボタン更新 */
48:                     is_exit = update_exit_button(px, py,
49:                                                  prev_lb && !s.LeftButton);
50:                     /* 追加（ここまで） */
51:
52:                     /* ・・・省略・・・ */
53:             }
54:     }
55: }
```

リスト 6.6 では、最初の画面描画時と cat コマンド実行時等の画面再描画時に put_exit_button 関数を実行するようにしています。また、マウス処理のループの最後に"終了ボタン更新"の処理を追加し、update_exit_button の戻り値に応じて gui 関数のメインループを抜けるようにしています。

6.3 マウスを少し大きくする

1px のマウスカーソルは、使えなくは無いのですが、あまりにも小さいので、少し大きく 4x4px にしてみます。

変更箇所は gui.c のみです (リスト 6.7)。

▼**リスト** 6.7　sample_poios/gui.c(**マウスカーソルを大きくする**)

```
 1: #define CURSOR_WIDTH       4       /* 追加 */
 2: #define CURSOR_HEIGHT      4       /* 追加 */
 3: /* ・・・省略・・・ */
 4: /* 追加（ここから） */
 5: struct EFI_GRAPHICS_OUTPUT_BLT_PIXEL cursor_tmp[CURSOR_HEIGHT][CURSOR_WIDTH] =
 6: {
 7:     {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}},
 8:     {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}},
 9:     {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}},
10:     {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}
11: };
```

```
12: int cursor_old_x;
13: int cursor_old_y;
14: /* 追加（ここまで） */
15: /* ・・・省略・・・ */
16: void draw_cursor(int x, int y)
17: {
18:     /* 変更（ここから） */
19:     int i, j;
20:     for (i = 0; i < CURSOR_HEIGHT; i++)
21:             for (j = 0; j < CURSOR_WIDTH; j++)
22:                     if ((i * j) < CURSOR_WIDTH)
23:                             draw_pixel(x + j, y + i, white);
24:     /* 変更（ここまで） */
25: }
26:
27: void save_cursor_area(int x, int y)
28: {
29:     /* 変更（ここから） */
30:     int i, j;
31:     for (i = 0; i < CURSOR_HEIGHT; i++) {
32:             for (j = 0; j < CURSOR_WIDTH; j++) {
33:                     if ((i * j) < CURSOR_WIDTH) {
34:                             cursor_tmp[i][j] = get_pixel(x + j, y + i);
35:                             cursor_tmp[i][j].Reserved = 0xff;
36:                     }
37:             }
38:     }
39:     /* 変更（ここまで） */
40: }
41:
42: void load_cursor_area(int x, int y)
43: {
44:     /* 変更（ここから） */
45:     int i, j;
46:     for (i = 0; i < CURSOR_HEIGHT; i++)
47:             for (j = 0; j < CURSOR_WIDTH; j++)
48:                     if ((i * j) < CURSOR_WIDTH)
49:                             draw_pixel(x + j, y + i, cursor_tmp[i][j]);
50:     /* 変更（ここまで） */
51: }
52:
53: void put_cursor(int x, int y)
54: {
55:     if (cursor_tmp[0][0].Reserved)  /* 変更 */
56:             load_cursor_area(cursor_old_x, cursor_old_y);
57:
58:     save_cursor_area(x, y);
59:
60:     draw_cursor(x, y);
61:
62:     cursor_old_x = x;
63:     cursor_old_y = y;
64: }
65: /* ・・・省略・・・ */
```

リスト 6.7 については、これまでの関数の枠組みは変わらず、各関数の処理の規模が少し大きくなっただけです。なお、マウスカーソルの形は、"(x * y) < CURSOR_WIDTH"

が真となる座標 (x,y) にのみドットを置く事で、"┌"を描いています。

6.4 機能拡張版 poiOS 実行の様子

機能拡張後の poiOS を画面写真で紹介します (図 6.1、図 6.2、図 6.3、図 6.4)。

▲図 6.1　GUI モードの状態

▲図 6.2　画像ファイルをクリックすると、画像を表示できます

▲図 6.3　[X] ボタンクリックで GUI モード終了し、シェルモードへ戻ります

▲**図** 6.4　exit **コマンド実行でシェルモードと** poiOS **終了し、ブートメニュー画面へ**

おわりに

ここまで読んでいただき、本当にありがとうございます！

ここまで、UEFI でのベアメタルプログラミングの方法について説明しました。「やればできそうだ」と思ってもらえたら嬉しいです。

UEFI はかしこいです。色々な機能を持っていて、色々な事をやってくれます。仕様書の目次を眺めていると、「こんなこともファームウェア側でやってくれるのか」と驚きます[*1]。また、ソフトウェアの中で最も下のレイヤー[*2]を扱っているにも関わらず、メモリマップを説明せずに済んでいる事はすごいことです。

"はじめに"でも書きましたが、本書のコンセプトには、「UEFI ファームウェアの機能をラップしていくだけで OS っぽいものが作れるんじゃ?」という思いがあります。中でも、ファイルシステムの機能をファームウェアが持っているのは強力です。フルスクラッチで OS 自作を始めようと考えている人が居れば、UEFI で始めるのも良いんじゃないかなと思います。

最後に、バイナリ短歌で終わりたいと思います (コードリスト .1、バイナリリスト .2)。サンプルコードのディレクトリは"sample_tanka"です。

▼リスト.1　"msg"ラベルのアドレスにある文字列を出力する

```
 1: #define CONOUT_ADDR        0xa3819590
 2: #define OUTPUTSTRING_ADDR  0xa387e1b8
 3:
 4:     .text
 5:     .globl          efi_main
 6: efi_main:
 7:     /* OutputString 第 1 引数 (ConOut) を RCX へ格納 */
 8:     movq            $CONOUT_ADDR, %rcx
 9:     /* OutputString のアドレスを RAX へ格納 */
10:     movq            $OUTPUTSTRING_ADDR, %rax
11:     /* ".ascii"のアドレスを RDX へ格納 */
12:     leaq            msg, %rdx
13:     /* OutputString を呼び出す */
```

[*1] メモリアロケータを持っているのは個人的に驚きでした。

[*2] もちろん、厳密にはより下にファームウェアがありますが。

```
14:     callq           *%rax
15:
16:     /* 無限ループ */
17:     jmp             .
18:
19:     .data
20: msg:
21:     /* "ABCD" */
22:     .ascii          "A\0B\0C\0D\0\0\0"
23:     /* "すごーい" */
24:     /*.ascii                "Y0T0\3740D0\0\0"*/
25:     /* ボス */
26:     /*.ascii                "\326\0\0\0"*/
```

▼**リスト**.2　**実行バイナリ逆アセンブル結果** (text **セクション** 32 **バイト、**1 **バイト字余り**)

```
 1: 0000000000401000 <efi_main>:
 2:   401000:       48 b9 90 95 81 a3 00    movabs $0xa3819590,%rcx
 3:   401007:       00 00 00
 4:   40100a:       48 b8 b8 e1 87 a3 00    movabs $0xa387e1b8,%rax
 5:   401011:       00 00 00
 6:   401014:       48 8d 14 25 00 20 40    lea    0x402000,%rdx
 7:   40101b:       00
 8:   40101c:       ff d0                   callq  *%rax
 9:   40101e:       eb fe                   jmp    40101e <efi_main+0x1e>
```

ここまで読んでいただき、本当にありがとうございました。

参考情報

参考にさせてもらった情報

- UEFI Specification
  - http://www.uefi.org/specifications
  - 一次情報源であり、もっとも参考にさせてもらいました。
- ツールキットを使わずに UEFI アプリケーションの Hello World! を作る - 品川高廣（東京大学）のブログ
  - http://d.hatena.ne.jp/shina_ecc/20140819/1408434995
  - UEFI でベアメタルプログラミングを始めるきっかけとなった記事です。
- EDK2 ソースコード
  - https://github.com/tianocore/edk2
  - 仕様書を見ても実装のイメージがわかない時に参考にさせてもらいました。
- gnu-efi ソースコード
  - https://sourceforge.net/p/gnu-efi/code/ci/master/tree/
  - 同じく実装の参考にさせてもらいました。

本書の他に UEFI ベアメタルプログラミングについて公開している情報

- ブログ記事 (へにゃぺんて@日々勉強のまとめ)
  - UEFI ベアメタルプログラミング - Hello UEFI!(ベアメタルプログラミングの流れについて)
    - http://d.hatena.ne.jp/cupnes/20170408/1491654807
  - UEFI ベアメタルプログラミング - マルチコアを制御する
    - http://d.hatena.ne.jp/cupnes/20170503/1493787477
- サンプルコード

- https://github.com/cupnes/bare_metal_uefi
- 本書のサンプルコードより書き殴りに近いですが、本書で紹介していないような使い方もあります
- タイマーイベントを扱うサンプル
  * https://github.com/cupnes/bare_metal_uefi/tree/master/080_timer_wait_for_event

フルスクラッチで作る!UEFI ベアメタルプログラミング

2017 年 8 月 11 日　コミックマーケット 92 版 v1.0
著　者　大神祐真
発行所　へにゃぺんて
連絡先　yuma@ohgami.jp
印刷所　日光企画