フルスクラッチで作る!
UEFI ベアメタルプログラミング
パート 2

大神祐真　著

2017-10-22 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。

本書は「UEFI」で「ベアメタルプログラミング」を行う本のパート 2 です。UEFI の仕様には前著 (パート 1) で紹介した他にも色々な機能があります。本書はパート 1 で紹介できなかった機能を紹介する TIPS 本です。

なお、本書ではパート 1 の内容をベースに UEFI の機能の呼び出し方を紹介します。そのため、パート 1 で作成した関数は特に説明もなく登場します。パート 1 を先に読んでいただくと良いですが、各節で単体で実行できるサンプルコードを用意しているので、特に意味が分からずとも、ソースコードをいじって色々と試してみるのも良いと思います。

本書に関する情報の公開場所

前著や本書の 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の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/uefi_bare_metal_programming_part2_samples`

目次

第 1 章

コンソール出力

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL 関係の TIPS を紹介します。

1.1 文字色と文字の背景色を設定する

SetAttribute() を使うことで文字色と文字の背景色を設定できます (リスト 1.1)。
サンプルのディレクトリは"010_simple_text_output_set_attribute"です。

▼**リスト** 1.1　SetAttribute() **の定義** (efi.h **より**)

```
//**************************************************
// Attributes
//**************************************************
#define EFI_BLACK      0x00
#define EFI_BLUE       0x01
#define EFI_GREEN      0x02
#define EFI_CYAN       0x03
#define EFI_RED        0x04
#define EFI_MAGENTA    0x05
#define EFI_BROWN      0x06
#define EFI_LIGHTGRAY  0x07
#define EFI_BRIGHT     0x08
#define EFI_DARKGRAY   0x08
#define EFI_LIGHTBLUE  0x09
#define EFI_LIGHTGREEN 0x0A
#define EFI_LIGHTCYAN  0x0B
#define EFI_LIGHTRED   0x0C
#define EFI_LIGHTMAGENTA       0x0D
#define EFI_YELLOW     0x0E
#define EFI_WHITE      0x0F

#define EFI_BACKGROUND_BLACK   0x00
#define EFI_BACKGROUND_BLUE    0x10
#define EFI_BACKGROUND_GREEN   0x20
#define EFI_BACKGROUND_CYAN    0x30
#define EFI_BACKGROUND_RED     0x40
```

```
#define EFI_BACKGROUND_MAGENTA 0x50
#define EFI_BACKGROUND_BROWN   0x60
#define EFI_BACKGROUND_LIGHTGRAY       0x70

struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {
        ・・・
        unsigned long long (*SetAttribute)(
                struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
                unsigned long long Attribute
                        /* 文字色と文字の背景色を設定
                         * 設定できる色は上記 define の通り */
                );
        ・・・
};
```

第 2 引数 Attribute に文字色と背景色を 1 バイトの値で設定します。上位 4 ビットが背景色で下位 4 ビットが文字色です。

SetAttribute() を使用して表紙画像を画面表示するサンプルがリスト 1.2 です。

▼**リスト** 1.2　SetAttribute() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle __attribute__ ((unused)),
 5:           struct EFI_SYSTEM_TABLE *SystemTable)
 6: {
 7:     efi_init(SystemTable);
 8:
 9:     puts(L" ");
10:
11:     ST->ConOut->SetAttribute(ST->ConOut,
12:                              EFI_LIGHTGREEN | EFI_BACKGROUND_LIGHTGRAY);
13:     ST->ConOut->ClearScreen(ST->ConOut);
14:
15:     puts(L"\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n");
16:     puts(L"                                  ");
17:     puts(L"フルスクラッチで作る!\r\n");
18:
19:     ST->ConOut->SetAttribute(ST->ConOut,
20:                              EFI_LIGHTRED | EFI_BACKGROUND_LIGHTGRAY);
21:     puts(L"                               ");
22:     puts(L"UEFI ベアメタルプログラミング\r\n");
23:
24:     ST->ConOut->SetAttribute(ST->ConOut,
25:                              EFI_LIGHTMAGENTA | EFI_BACKGROUND_LIGHTGRAY);
26:     puts(L"                                    ");
27:     puts(L"パート 2\r\n");
28:
29:     ST->ConOut->SetAttribute(ST->ConOut,
30:                              EFI_LIGHTBLUE | EFI_BACKGROUND_LIGHTGRAY);
31:     puts(L"\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n");
32:     puts(L"                                     ");
33:     puts(L"大ネ申　ネ右真 著\r\n");
```

```
34:
35:     ST->ConOut->SetAttribute(ST->ConOut,
36:                              EFI_WHITE | EFI_BACKGROUND_LIGHTGRAY);
37:     puts(L"\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n");
38:     puts(L"                          ");
39:     puts(L"2017-10-22 版  ");
40:
41:     ST->ConOut->SetAttribute(ST->ConOut,
42:                              EFI_LIGHTCYAN | EFI_BACKGROUND_LIGHTGRAY);
43:     puts(L"henyapente ");
44:
45:     ST->ConOut->SetAttribute(ST->ConOut,
46:                              EFI_MAGENTA | EFI_BACKGROUND_LIGHTGRAY);
47:     puts(L"発 行\r\n");
48:
49:     while (TRUE);
50: }
```

リスト 1.2 では、efi_init() の直後に半角スペースを 1 つだけ出力しています。これは、実機 (筆者の場合 Lenovo 製) の UEFI ファームウェアの挙動に合わせるためで、筆者の実機の場合、起動直後の何も画面出力していない状態の ClearScreen() は無視されるようで、たとえその直前で SetAttribute() で属性を設定していたとしても画面を指定した背景色でクリアすることができませんでした。そこで、一度、何らかの画面表示を行う必要があり、SetAttribute() と ClearScreen() の直前に半角スペースの出力を入れています[*1]。

1.2 出力可能な文字であるか否かを判定する

UEFI では文字は Unicode で扱います。Unicode の範囲としては日本語なども含みますが、UEFI のファームウェアの実装としてすべての文字をサポートしているとは限りません。

TestString() を使うことで、文字 (列) が出力可能か否かを判定できます (リスト 1.3)。

サンプルのディレクトリは"011_simple_text_output_test_string"です。

▼**リスト** 1.3　TestString() **の定義** (efi.h **より**)

```
struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {
        ・・・
        unsigned long long (*TestString)(
                struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
                unsigned short *String
                        /* 判定したい文字列の先頭アドレスを指定 */
```

[*1] 「何も画面出力を行っていないのだから ClearScreen() は無視する」というファームウェア側の実装は分からなくは無いのですが、SetAttribute() で背景色を変更している場合は新しい背景色で描画しなおして欲しいですね。

```
                );
        ...
};
```

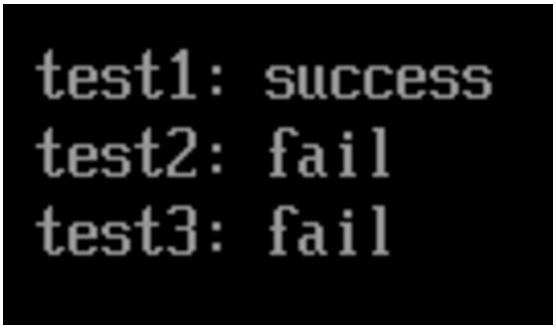
TestString() は引数に文字列の先頭アドレスを指定し、判定結果は戻り値で受け取ります。出力可能な場合は EFI_SUCCESS(=0) を、出力できない文字が含まれる場合は EFI_UNSUPPORTED(=0x80000000 00000003) を返します。

使用例はリスト 1.4 の通りです。

▼**リスト** 1.4　TestString() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle __attribute__ ((unused)),
 5:           struct EFI_SYSTEM_TABLE *SystemTable)
 6: {
 7:     efi_init(SystemTable);
 8:     ST->ConOut->ClearScreen(ST->ConOut);
 9:
10:     /* test1 */
11:     if (!ST->ConOut->TestString(ST->ConOut, L"Hello"))
12:             puts(L"test1: success\r\n");
13:     else
14:             puts(L"test1: fail\r\n");
15:
16:     /* test2 */
17:     if (!ST->ConOut->TestString(ST->ConOut, L"こんにちは"))
18:             puts(L"test2: success\r\n");
19:     else
20:             puts(L"test2: fail\r\n");
21:
22:     /* test3 */
23:     if (!ST->ConOut->TestString(ST->ConOut, L"Hello, こんにちは"))
24:             puts(L"test3: success\r\n");
25:     else
26:             puts(L"test3: fail\r\n");
27:
28:     while (TRUE);
29: }
```

QEMU 上で実行してみると、OVMF のファームウェアは日本語に対応していない為、図 1.1 の通り、日本語を含む文字列を指定している test2 と test3 は失敗する結果となります。なお、QEMU 上で実行する際には Makefile を一部修正する必要があります。詳しくは後述のコラムを参照してください。



▲図 1.1　QEMU での TestString() の実行例

日本語表示が可能な実機で実行してみると図 1.2 の通り、test1〜test3 の全てが成功します。

▲図 1.2　実機での TestString() の実行例

QEMU(OVMF) では実行バイナリサイズに制限 (?)

QEMU 用の UEFI ファームウェアである OVMF には実行バイナリサイズに制限がある様子で、コード量が増え、実行バイナリサイズが大きくなると、同じ実行バイナリが実機では実行できるが QEMU 上では実行できない (UEFI ファームウェアが実行バイナリを見つけてくれない) 事態に陥ります。
当シリーズの UEFI ベアメタルプログラミングの対象は主に実機 (Lenovo 製で動作確認) であるため[a]、本書では QEMU 上で動作させる際は、Makefile を修正し、不要なソースコードをコンパイル・リンクの対象に含めないようにしています。
例えば、リスト 1.4 を QEMU 上で動作させる際は、Makefile をリスト 1.5 のように修正し、make を行って下さい。

▼**リスト** 1.5　QEMU(OVMF) **上で実行させる際の** Makefile

```
・・・
# fs/EFI/BOOT/BOOTX64.EFI: efi.c common.c file.c graphics.c shell.c gui.c
# ↑コメントアウトか削除
fs/EFI/BOOT/BOOTX64.EFI: efi.c common.c main.c  <= 追加
        mkdir -p fs/EFI/BOOT
        x86_64-w64-mingw32-gcc -Wall -Wextra -e efi_main -nostdinc \
        -nostdlib -fno-builtin -Wl,--subsystem,10 -o $@ $+
・・・
```

[a] 言い訳ですが。。

1.3　テキストモードの情報を取得する

コンソールへのテキスト出力に関して、これまでは UEFI のファームウェアが認識するデフォルトのモード (テキストモード) で使用してきました。ただし、コンソールデバイスによってはその他のテキストモードへ切り替えることで、画面あたりの列数・行数を変更できます。

サンプルのディレクトリは"012_simple_text_output_query_mode"です。

QueryMode() でテキストモード番号に対する画面あたりの列数・行数を取得できます (リスト 1.6)。

▼**リスト** 1.6　QueryMode() **の定義** (efi.h **より**)

```
struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {
        ・・・
        unsigned long long (*QueryMode)(
                struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
                unsigned long long ModeNumber,
                        /* 列数・行数を確認したいモード番号を指定 */
                unsigned long long *Columns,
                        /* 列数を格納する変数へのポインタ */
                unsigned long long *Rows);
                        /* 行数を格納する変数へのポインタ */
        ・・・
};
```

QueryMode() は、第 2 引数で指定されたモード番号がコンソールデバイスでサポート外の場合、EFI_UNSUPPORTED(0x80000000 00000003) を返します。

なお、コンソールデバイスで使用できるモード番号の範囲は EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL->Mode->MaxMode で確認できます。EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL->Mode は、SIMPLE_TEXT_OUTPUT_MODE という構造体で、定義はリスト 1.7 の通りです。

▼**リスト** 1.7　SIMPLE_TEXT_OUTPUT_MODE **構造体の定義** (efi.h **より**)

```
#define EFI_SUCCESS     0
#define EFI_ERROR       0x8000000000000000
#define EFI_UNSUPPORTED         (EFI_ERROR | 3)

struct SIMPLE_TEXT_OUTPUT_MODE {
        int MaxMode;  /* コンソールデバイスで使用できるモード番号の最大値 */
        int Mode;  /* 現在のモード番号 */
        int Attribute;  /* 現在設定されている属性値（文字色・背景色）*/
        int CursorColumn;  /* カーソル位置（列）*/
        int CursorRow;  /* カーソル位置（行）*/
        unsigned char CursorVisible;  /* 現在のカーソル表示設定（表示=1/非表示=0）*/
};
```

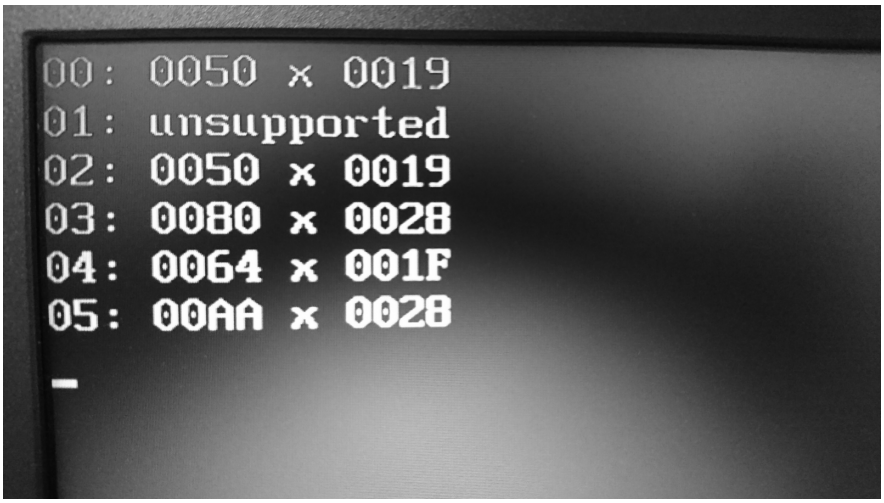
以上を踏まえ、QueryMode() の使用例はリスト 1.8 の通りです。

▼**リスト** 1.8　QueryMode() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle __attribute__ ((unused)),
 5:           struct EFI_SYSTEM_TABLE *SystemTable)
 6: {
 7:     int mode;
 8:     unsigned long long status;
 9:     unsigned long long col, row;
10:
```

```
11:     efi_init(SystemTable);
12:     ST->ConOut->ClearScreen(ST->ConOut);
13:
14:     for (mode = 0; mode < ST->ConOut->Mode->MaxMode; mode++) {
15:             status = ST->ConOut->QueryMode(ST->ConOut, mode, &col, &row);
16:             switch (status) {
17:             case EFI_SUCCESS:
18:                     puth(mode, 2);
19:                     puts(L": ");
20:                     puth(col, 4);
21:                     puts(L" x ");
22:                     puth(row, 4);
23:                     puts(L"\r\n");
24:                     break;
25:
26:             case EFI_UNSUPPORTED:
27:                     puth(mode, 2);
28:                     puts(L": unsupported\r\n");
29:                     break;
30:
31:             default:
32:                     assert(status, L"QueryMode");
33:                     break;
34:             }
35:     }
36:
37:     while (TRUE);
38: }
```

実行例は図 1.3 の通りです。



▲図 1.3　QueryMode() の実行例

1.4 テキストモードを変更する

SetMode() でテキストモードを変更できます (リスト 1.9)。
サンプルのディレクトリは"013_simple_text_output_set_mode"です。

▼**リスト** 1.9　SetMode() **の定義** (efi.h **より**)

```
struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {
        ・・・
        unsigned long long (*SetMode)(
                struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
                unsigned long long ModeNumber  /* テキストモード番号 */
                );
        ・・・
```

使用例はリスト 1.10 の通りです。

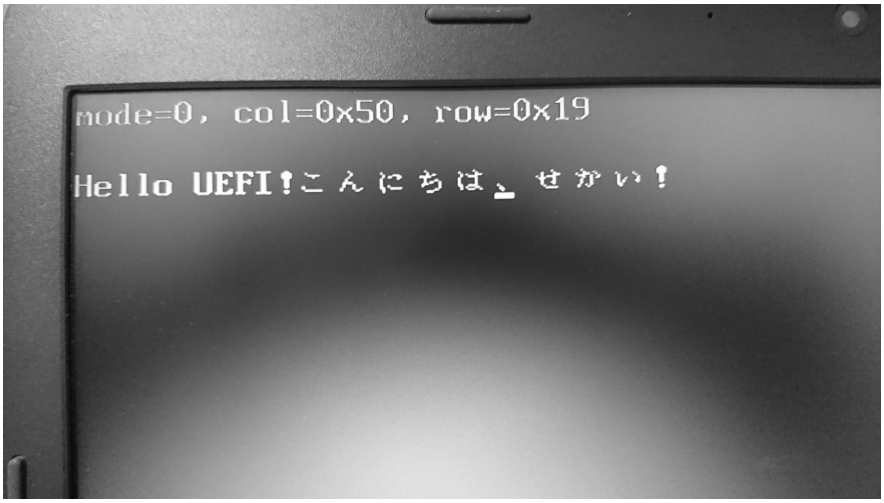
▼**リスト** 1.10　SetMode() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle __attribute__ ((unused)),
 5:           struct EFI_SYSTEM_TABLE *SystemTable)
 6: {
 7:     int mode;
 8:     unsigned long long status;
 9:     unsigned long long col, row;
10:
11:     efi_init(SystemTable);
12:     ST->ConOut->ClearScreen(ST->ConOut);
13:
14:     while (TRUE) {
15:             for (mode = 0; mode < ST->ConOut->Mode->MaxMode; mode++) {
16:                     status = ST->ConOut->QueryMode(ST->ConOut, mode, &col,
17:                                                    &row);
18:                     if (status)
19:                             continue;
20:
21:                     ST->ConOut->SetMode(ST->ConOut, mode);
22:                     ST->ConOut->ClearScreen(SystemTable->ConOut);
23:
24:                     puts(L"mode=");
25:                     puth(mode, 1);
26:                     puts(L", col=0x");
27:                     puth(col, 2);
28:                     puts(L", row=0x");
29:                     puth(row, 2);
30:                     puts(L"\r\n");
31:                     puts(L"\r\n");
32:                     puts(L"Hello UEFI!こんにちは、せかい!");
```

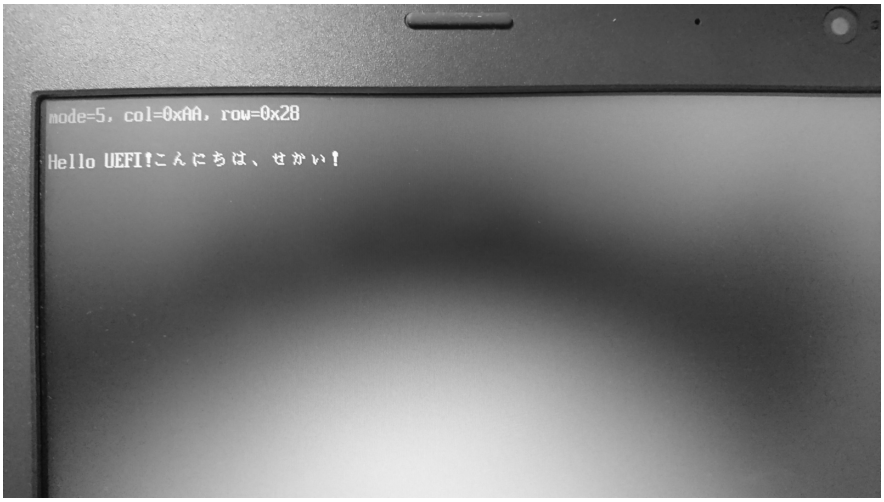
```
33:
34:                     getc();
35:             }
36:     }
37: }
```

リスト 1.10 では、コンソールデバイスが対応しているテキストモードを順に切り替え、テキストモードの情報と簡単なテキスト ("Hello UEFI!こんにちは、世界！ ") を表示します。表示後、getc() で待ちますので、何らかのキーを入力してください (すると、次のテキストモードへ切り替わります)。

実行例は図 1.4 と図 1.5 の通りです。テキストモードによって見た目がどう変わるのかは、ぜひご自身の実機で試してみてください。



▲図 1.4　SetMode の実行例:モード 0(列数 80(0x50), 行数 25(0x19)) の場合



▲図 1.5　SetMode の実行例:モード 5(列数 170(0xAA), 行数 40(0x28)) の場合

グラフィックモードの情報を確認し、変更するには

EFI_GRAPHICS_OUTPUT_PROTOCOL も QueryMode() と SetMode() を持っています。そのため、フレームバッファのグラフィックモードについても、QueryMode() でモード番号に対する情報 (解像度) を取得し、SetMode() で指定したモード番号へ変更することができます。
参考までに、EFI_GRAPHICS_OUTPUT_PROTOCOL の QueryMode() と SetMode() の定義をリスト 1.11 に示します。

▼**リスト** 1.11　QueryMode() **と** SetMode() **の定義**

```
#define EFI_SUCCESS    0
#define EFI_ERROR      0x8000000000000000
#define EFI_INVALID_PARAMETER  (EFI_ERROR | 2)

struct EFI_GRAPHICS_OUTPUT_PROTOCOL {
        unsigned long long (*QueryMode)(
                struct EFI_GRAPHICS_OUTPUT_PROTOCOL *This,
                unsigned int ModeNumber,
                        /* 解像度等を確認したいモード番号を指定 */
                unsigned long long *SizeOfInfo,
                        /* 第 3 引数"info"のサイズが設定される
                         * 変数のポインタを指定 */
                struct EFI_GRAPHICS_OUTPUT_MODE_INFORMATION **Info
                        /* グラフィックモードの情報を格納する構造体
                         * EFI_GRAPHICS_OUTPUT_MODE_INFORMATION への
                         * ポインタのポインタを指定 */
                );

        unsigned long long (*SetMode)(
                struct EFI_GRAPHICS_OUTPUT_PROTOCOL *This,
                unsigned int ModeNumber
                        /* グラフィックモード番号 */
                );
        ...
};
```

なお、EFI_GRAPHICS_OUTPUT_PROTOCOL の QueryMode() は、第 2 引数で無効なモード番号を指定した時、EFI_INVALID_PARAMETER を返します。

第 2 章

キーボード入力

EFI_SIMPLE_TEXT_INPUT_PROTOCOL よりも少し高機能な EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL があります。この章では EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL 関係の TIPS を紹介します。

EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL の GUID と定義はリスト 2.1 の通りです。

▼**リスト** 2.1　EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL **の** GUID **と定義**

```
struct EFI_GUID stiep_guid = {0xdd9e7534, 0x7762, 0x4698, \
                              {0x8c, 0x14, 0xf5, 0x85,   \
                               0x17, 0xa6, 0x25, 0xaa}};

struct EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL {
        /* テキスト入力デバイスをリセットする */
        unsigned long long (*Reset)(
                struct EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
                unsigned char ExtendedVerification);

        /* キー入力データを取得 */
        unsigned long long (*ReadKeyStrokeEx)(
                struct EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
                struct EFI_KEY_DATA *KeyData);

        /* キー入力を待つ EFI_EVENT */
        void *WaitForKeyEx;

        /* キーのトグル状態を設定 (ScrollLock,NumLock,CapsLock) */
        unsigned long long (*SetState)(
                struct EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
                unsigned char *KeyToggleState);

        /* 特定のキー入力を通知する関数を設定 */
        unsigned long long (*RegisterKeyNotify)(
                struct EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
                struct EFI_KEY_DATA *KeyData,
                unsigned long long (*KeyNotificationFunction)(
```

```
                        struct EFI_KEY_DATA *KeyData),
                void **NotifyHandle);

        /* 特定のキー入力を通知する関数を解除 */
        unsigned long long (*UnregisterKeyNotify)(
                struct EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
                void *NotificationHandle);
};
```

この章では RegisterKeyNotify() を紹介します。

2.1 特定のキー入力で呼び出される関数を登録する

RegisterKeyNotify() を使用すると、特定のキー入力で呼び出される関数を登録できます (リスト 2.2)。

サンプルのディレクトリは"020_simple_text_input_ex_register_key_notify"です。

▼**リスト** 2.2　RegisterKeyNotify() **の定義** (efi.h **より**)

```
struct EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL {
        ・・・
        unsigned long long (*RegisterKeyNotify)(
                struct EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
                struct EFI_KEY_DATA *KeyData,
                        /* イベントとして扱うキー入力を指定 */
                unsigned long long (*KeyNotificationFunction)(
                        struct EFI_KEY_DATA *KeyData),
                        /* 通知関数 */
                void **NotifyHandle
                        /* ユニークなハンドルを返す
                         * 登録解除時に使用 */
                );
        ・・・
};
```

RegisterKeyNotify() は、第 2 引数へイベントとして扱うキー入力を指定し、第 3 引数へイベント発生を通知する関数を設定します。第 2 引数で指定する struct EFI_KEY_DATA の定義はリスト 2.3 の通りです。

▼**リスト** 2.3　struct EFI_KEY_DATA **の定義** (efi.h **より**)

```
struct EFI_KEY_DATA {
        /* 入力されたキーに関する指定 */
        struct EFI_INPUT_KEY {
                unsigned short ScanCode;
                        /* Unicode 外のキー入力時に使用。スキャンコード */
```

```
                unsigned short UnicodeChar;
                        /* Unicode 内のキー入力時に使用。ユニコード値 */
        } Key;
        /* キー入力時の状態に関する指定 */
        struct EFI_KEY_STATE {
                unsigned int KeyShiftState;
                        /* Shift キーの押下状態 */
                unsigned char KeyToggleState;
                        /* キーボードのトグル状態
                         * (ScrollLock,NumLock,CapsLock) を指定 */
        } KeyState;
};
```

「どういう状態で何のキーが押されたか」を指定するために EFI_KEY_DATA にはキーボードのトグル状態なども含まれますが、今回は EFI_INPUT_KEY のみ使用し、その他は 0 を設定しておきます。EFI_KEY_STATE の定義について詳しくは、仕様書の"Protocols - Console Support"の"Simple Text Input Ex Protocol"の"EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.ReadKeyStrokeEx()"を見てみてください。

RegisterKeyNotify() の使用例はリスト 2.4 の通りです。

▼**リスト** 2.4　RegisterKeyNotify() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: unsigned char is_exit = FALSE;
 5:
 6: unsigned long long key_notice(
 7:     struct EFI_KEY_DATA *KeyData __attribute__ ((unused)))
 8: {
 9:     is_exit = TRUE;
10:
11:     return EFI_SUCCESS;
12: }
13:
14: void efi_main(void *ImageHandle __attribute__ ((unused)),
15:           struct EFI_SYSTEM_TABLE *SystemTable)
16: {
17:     unsigned long long status;
18:     struct EFI_KEY_DATA key_data = {{0, L'q'}, {0, 0}};
19:     void *notify_handle;
20:
21:     efi_init(SystemTable);
22:     ST->ConOut->ClearScreen(ST->ConOut);
23:
24:     puts(L"Waiting for the 'q' key input...\r\n");
25:
26:     status = STIEP->RegisterKeyNotify(STIEP, &key_data, key_notice,
27:                                       &notify_handle);
28:     assert(status, L"RegisterKeyNotify");
```

```
29:
30:     while (!is_exit);
31:
32:     puts(L"exit.\r\n");
33:
34:     while (TRUE);
35: }
```

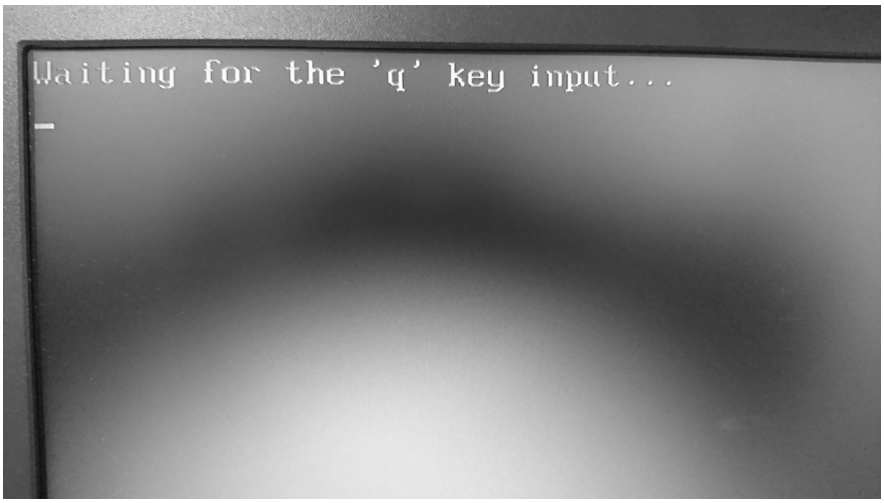
リスト 2.4 は、'q' キーの入力で efi_main() 内の特定の処理を抜けるサンプルになっています。

まず、'q' キーをイベント通知対象とするため、struct EFI_KEY_DATA key_data へは struct EFI_INPUT_KEY の UnicodeChar へ'q' を登録し、その他は 0 を設定しています。

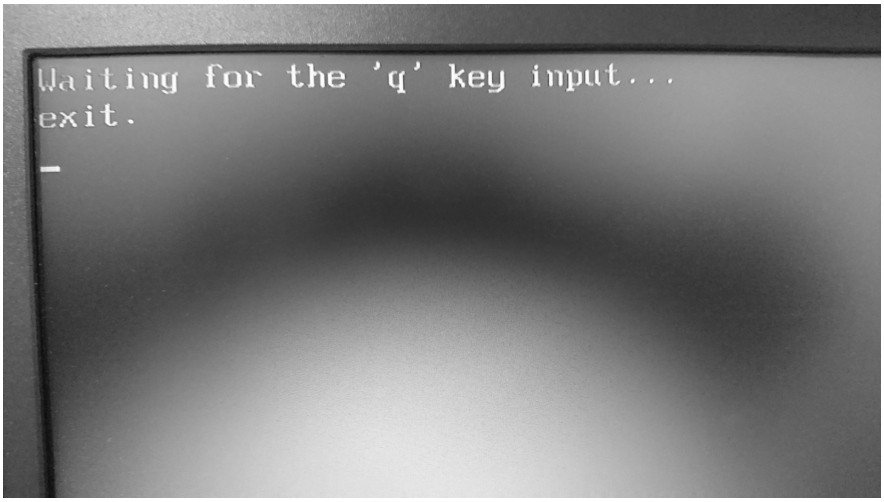
そして、RegisterKeyNotify() を使用し、'q' キー入力で key_notice() が呼び出される様に登録しています。

key_notice() では、グローバル変数 is_exit へ TRUE を設定するため、'q' キー入力で efi_main() 内の"while (!is_exit);"を抜けます。

実行例は図 2.1 と図 2.2 の通りです。



▲**図 2.1　RegisterKeyNotify() 実行例 1('q' キー入力待ち)**



▲図 2.2　RegisterKeyNotify() 実行例 2('q' キー入力後)

補足: Unocode 外のキー入力を検出するには

Unicode 外のキー入力を検出するには、struct EFI_KEY_DATA の ScanCode を使用します。仕様書"12.1.2 ConsoleIn Definition"からの引用ですが、指定できるスキャンコードは表 2.1 の通りです。

表 2.1 から、Escape キーを検出したい場合は、struct EFI_KEY_DATA の ScanCode へ 0x17 を設定しておけば良いことが分かります。

そのため、リスト 2.4 を、「Escape キーで while ループを抜ける」へ変更する場合、18 行目の変数"key_data"への代入処理をリスト 2.5 へ変更すれば良いです。

▼リスト 2.5　"ESC"キーで while ループを抜けるよう変更

```
struct EFI_KEY_DATA esc_key_data = {{0x17, 0}, {0, 0}};
```

▼表 2.1　スキャンコード

スキャンコード	説明	スキャンコード	説明
0x00	Null スキャンコード	0x15	F11
0x01	上	0x16	F12
0x02	下	0x68	F13
0x03	右	0x69	F14
0x04	左	0x6A	F15
0x05	Home	0x6B	F16
0x06	End	0x6C	F17
0x07	Insert	0x6D	F18
0x08	Delete	0x6E	F19
0x09	Page Up	0x6F	F20
0x0a	Page Down	0x70	F21
0x0b	F1	0x71	F22
0x0c	F2	0x72	F23
0x0d	F3	0x73	F24
0x0e	F4	0x7F	ミュート
0x0f	F5	0x80	音量を上げる
0x10	F6	0x81	音量を下げる
0x11	F7	0x100	画面の明るさを上げる
0x12	F8	0x101	画面の明るさを下げる
0x13	F9	0x102	サスペンド
0x14	F10	0x103	ハイバネート
0x17	Escape	0x104	ディスプレイトグル
		0x105	リカバリー
		0x106	イジェクト
		0x8000-0xFFFF	OEM 予約領域

第 3 章

UEFI アプリケーションのロード・実行

EFI_BOOT_SERVICES 内の LoadImage() と StartImage() を使用することで、UEFI アプリケーションをロード・実行できます。ただし、そのためには UEFI の「デバイスパス」という概念でパスを作成する必要があります。

この章ではデバイスパスを見てみる、作ってみるところから、順を追って説明し、UEFI アプリケーションをロード・実行する方法を紹介します。

なお実は、今の Linux カーネルは UEFI アプリケーションとしてカーネルイメージを生成する機能があります。そこで、この章の最後では、Linux カーネルの起動を行ってみます。

3.1 自分自身のパスを表示してみる

LoadImage() で UEFI アプリケーションの実行バイナリをロードするには、実行バイナリへのパスを「デバイスパス」というもので作成する必要があります。

実は、自分自身 (EFI/BOOT/BOOTX64.EFI) のデバイスパスを取得する方法がありますので、この章では、既存のデバイスパスを改造して、起動したい UEFI アプリケーションのデバイスパスを作ることにします。

この節では、まず、自分自身のデバイスパスを画面へ表示し、どんなものか見てみます。

サンプルのディレクトリは"030_loaded_image_protocol_file_path"です。

3.1.1 EFI_LOADED_IMAGE_PROTOCOL

ロード済みのイメージ (UEFI アプリケーション) の情報を取得するには、EFI_LOADED_IMAGE_PROTOCOL を使用します (リスト 3.1)。

▼**リスト** 3.1　EFI_LOADED_IMAGE_PROTOCOL **の定義** (efi.h **より**)

```
struct EFI_LOADED_IMAGE_PROTOCOL {
        unsigned int Revision;
        void *ParentHandle;
        struct EFI_SYSTEM_TABLE *SystemTable;
        // Source location of the image
        void *DeviceHandle;
        struct EFI_DEVICE_PATH_PROTOCOL *FilePath;
        void *Reserved;
        // Image's load options
        unsigned int LoadOptionsSize;
        void *LoadOptions;
        // Location where image was loaded
        void *ImageBase;
        unsigned long long ImageSize;
        enum EFI_MEMORY_TYPE ImageCodeType;
        enum EFI_MEMORY_TYPE ImageDataType;
        unsigned long long (*Unload)(void *ImageHandle);
};
```

EFI_LOADED_IMAGE_PROTOCOL は、これまで見てきたプロトコルとは異なり、メンバのほとんどが変数や構造体のポインタで、これらのメンバにロード済みイメージの各種情報が格納されています。リスト 3.1 を見てみると、"FilePath"というメンバがあります。しかも型が"EFI_DEVICE_PATH_PROTOCOL"という名前なので、ここにロード済みイメージのデバイスパスが格納されていそうです。

3.1.2 OpenProtocol() で EFI_LOADED_IMAGE_PROTOCOL を取得

そして、ロード済みイメージの EFI_LOADED_IMAGE_PROTOCOL を取得するために EFI_BOOT_SERVICES の OpenProtocol() を使用します (リスト 3.2)。

▼**リスト** 3.2　OpenProtocol() **の定義** (efi.h **より**)

```
struct EFI_SYSTEM_TABLE {
        ・・・
        struct EFI_BOOT_SERVICES {
                ・・・
                unsigned long long (*OpenProtocol)(
```

```
                        void *Handle,
                                /* オープンするプロトコルで
                                 * 扱う対象のハンドルを指定 */
                        struct EFI_GUID *Protocol,      /* プロトコルの GUID */
                        void **Interface,
                                /* プロトコル構造体のポインタを格納 */
                        void *AgentHandle,
                                /* OpenProtocol() を実行している
                                 * UEFI アプリケーションのイメージハンドル
                                 * すなわち、自分自身のイメージハンドル */
                        void *ControllerHandle,
                                /* OpenProtocol() を実行しているのが
                                 * 「UEFI ドライバー」である場合に指定する
                                 * コントローラハンドル
                                 * そうでない UEFI アプリケーションの場合、
                                 * NULL を指定 */
                        unsigned int Attributes
                                /* プロトコルインタフェースを開くモードを指定 */
                        );
                . . .
        } *BootServices;
};
```

EFI_LOADED_IMAGE_PROTOCOL を開く場合、OpenProtocol() の第 1 引数へは EFI_LOADED_IMAGE_PROTOCOL で情報を見たい対象のイメージハンドルを指定し、第 4 引数へは OpenProtocol() を実行している UEFI アプリケーションのイメージハンドルを指定します。今回の場合はどちらも、起動時から実行している UEFI アプリケーションです。

なお通常、UEFI アプリケーションのイメージハンドルは後述する LoadImage() で UEFI アプリケーションをロードする際に取得しますが、起動時から実行している UEFI アプリケーションについては、エントリ関数の第 1 引数"ImageHandle"が自分自身のイメージハンドルです。そのため、OpenProtocol() の第 1 引数と第 4 引数へは ImageHandle を指定します。

また、OpenProtocol() の第 6 引数"Attributes"へ指定できるモード定数はリスト 3.3 の通りです。今回の場合、"EFI_OPEN_PROTOCOL_GET_PROTOCOL"を指定します。

▼**リスト** 3.3　OpenProtocol() **第 4 引数**"Atributes"**へ指定できる定数** (efi.h **より**)

```
#define EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL   0x00000001
#define EFI_OPEN_PROTOCOL_GET_PROTOCOL 0x00000002
#define EFI_OPEN_PROTOCOL_TEST_PROTOCOL        0x00000004
#define EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER  0x00000008
#define EFI_OPEN_PROTOCOL_BY_DRIVER    0x00000010
#define EFI_OPEN_PROTOCOL_EXCLUSIVE    0x00000020
```

ここまでをまとめると、OpenProtocol() はリスト 3.4 の様に使用します。なお、

EFI_LOADED_IMAGE_PROTOCOL の GUID"lip_guid"は、efi.h(リスト 3.5) と efi.c(リスト 3.6) へ定義を追加しています。

▼**リスト** 3.4　OpenProtocol() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
 5: {
 6:     unsigned long long status;
 7:     struct EFI_LOADED_IMAGE_PROTOCOL *lip;
 8:
 9:     efi_init(SystemTable);
10:     ST->ConOut->ClearScreen(ST->ConOut);
11:
12:     status = ST->BootServices->OpenProtocol(
13:             ImageHandle, &lip_guid, (void **)&lip, ImageHandle, NULL,
14:             EFI_OPEN_PROTOCOL_GET_PROTOCOL);
15:     assert(status, L"OpenProtocol");
16:
17:     while (TRUE);
18: }
```

▼**リスト** 3.5　lip_guid **の** extern(efi.h **より**)

```
1: extern struct EFI_GUID lip_guid;
```

▼**リスト** 3.6　lip_guid **の定義** (efi.c **より**)

```
1: struct EFI_GUID lip_guid = {0x5b1b31a1, 0x9562, 0x11d2,
2:                         {0x8e, 0x3f, 0x00, 0xa0,
3:                          0xc9, 0x69, 0x72, 0x3b}};
```

3.1.3　EFI_DEVICE_PATH_TO_TEXT_PROTOCOL

EFI_DEVICE_PATH_TO_TEXT_PROTOCOL を使用することで、デバイスパスを画面に表示できるようテキストへ変換できます (リスト 3.7)。

▼**リスト** 3.7　EFI_DEVICE_PATH_TO_TEXT_PROTOCOL **の定義**

```
struct EFI_DEVICE_PATH_TO_TEXT_PROTOCOL {
        unsigned long long _buf;
        unsigned short *(*ConvertDevicePathToText)(
```

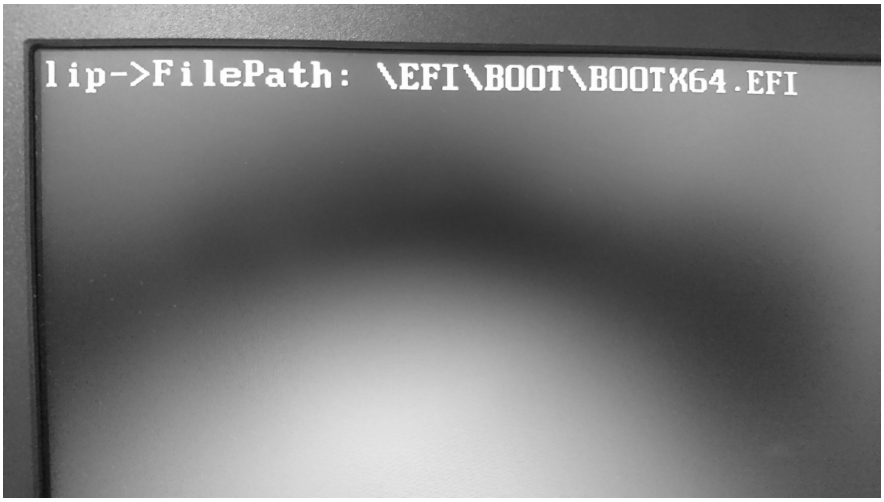
```
                const struct EFI_DEVICE_PATH_PROTOCOL* DeviceNode,
                unsigned char DisplayOnly,
                unsigned char AllowShortcuts);
};
```

前節で取得した EFI_LOADED_IMAGE_PROTOCOL の struct EFI_DEVICE_PATH_PROTOCOL *FilePath の内容をテキストへ変換して表示してみます (リスト 3.8)。

▼**リスト** 3.8　ConvertDevicePathToText() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
 5: {
 6:     unsigned long long status;
 7:     struct EFI_LOADED_IMAGE_PROTOCOL *lip;
 8:
 9:     efi_init(SystemTable);
10:     ST->ConOut->ClearScreen(ST->ConOut);
11:
12:     status = ST->BootServices->OpenProtocol(
13:             ImageHandle, &lip_guid, (void **)&lip, ImageHandle, NULL,
14:             EFI_OPEN_PROTOCOL_GET_PROTOCOL);
15:     assert(status, L"OpenProtocol");
16:
17:     /* 追加（ここから） */
18:     puts(L"lip->FilePath: ");
19:     puts(DPTTP->ConvertDevicePathToText(lip->FilePath, FALSE, FALSE));
20:     puts(L"\r\n");
21:     /* 追加（ここまで） */
22:
23:     while (TRUE);
24: }
```

実行すると図 3.1 の様に表示されます。



▲図 3.1　デバイスパスの表示例

図 3.1 を見ると、起動時から実行している自分自身のデバイスパスとしては、実行バイナリを配置した通り、"\EFI\BOOT\BOOTX64.EFI"というデバイスパスが格納されていることが分かります。

3.2　デバイスパスを作成してみる (その 1)

前節では確認のためにデバイスパスをテキストへ変換する関数 ConvertDevicePathToText() を紹介しました。

実はその逆に、テキストをデバイスパスへ変換する関数もあります。この節ではその関数を使用してみます。

サンプルのディレクトリは"031_create_devpath_1"です。

テキストをデバイスパスへ変換する関数は、EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL の ConvertTextToDevicePath() です (リスト 3.9)。

▼リスト 3.9　EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL の定義 (efi.h より)

```
struct EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL {
        unsigned long long _buf;
        struct EFI_DEVICE_PATH_PROTOCOL *(*ConvertTextToDevicePath) (
                const unsigned short *TextDevicePath);
};
```

EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL の取得は LocateProtocol() で行います (リスト 3.10)。

▼**リスト** 3.10　EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL **の取得** (efi.h **より**)

```
struct EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL *DPFTP;  /* => efi.h で extern */

void efi_init(struct EFI_SYSTEM_TABLE *SystemTable)
{
        ・・・
        struct EFI_GUID dpftp_guid = {0x5c99a21, 0xc70f, 0x4ad2,
                                      {0x8a, 0x5f, 0x35, 0xdf,
                                       0x33, 0x43, 0xf5, 0x1e}};
        ・・・
        ST->BootServices->LocateProtocol(&dpftp_guid, NULL, (void **)&DPFTP);
}
```

それでは、ConvertTextToDevicePath() を使用してデバイスパスを作成してみたいと思います。前節で、起動時から実行している自分自身のデバイスパスは"\EFI\BOOT\BOOTX64.EFI"というテキストでした。"\test.efi"というテキストをデバイスパスへ変換すれば、「USB フラッシュメモリ直下の test.efi という UEFI アプリケーション」を表せそうです。ConvertTextToDevicePath() の使用例はリスト 3.11 の通りです。

▼**リスト** 3.11　ConvertTextToDevicePath() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle __attribute__ ((unused)),
 5:           struct EFI_SYSTEM_TABLE *SystemTable)
 6: {
 7:     struct EFI_DEVICE_PATH_PROTOCOL *dev_path;
 8:
 9:     efi_init(SystemTable);
10:     ST->ConOut->ClearScreen(ST->ConOut);
11:
12:     dev_path = DPFTP->ConvertTextToDevicePath(L"\\test.efi");
13:     puts(L"dev_path: ");
14:     puts(DPTTP->ConvertDevicePathToText(dev_path, FALSE, FALSE));
15:     puts(L"\r\n");
16:
17:     while (TRUE);
18: }
```

リスト 3.11 では、確認のために、作成したデバイスパスを ConvertDevicePathToText() を使用してテキストへ戻し、画面表示しています。

実行すると図 3.2 の様に表示されます。

▲図 3.2　ConvertTextToDevicePath() の実行例

意図した通りのデバイスパスが作成できました。

3.3　デバイスパスをロードしてみる (その 1)

デバイスパスを作成できたので、LoadImage() でロードしてみます。
サンプルのディレクトリは"032_load_devpath_1"です。
LoadImage() は EFI_BOOT_SERVICES 内で定義されています (リスト 3.12)。

▼リスト 3.12　LoadImage() の定義 (efi.h より)

```
struct EFI_SYSTEM_TABLE {
        ・・・
        struct EFI_BOOT_SERVICES {
                ・・・
                //
                // Image Services
                //
                unsigned long long (*LoadImage)(
                        unsigned char BootPolicy,
                                /* ブートマネージャーからのブートか否かの指定
                                 * 今回はブートマネージャー関係ないので FALSE */
                        void *ParentImageHandle,
                                /* 呼び出し元のイメージハンドル
                                 * 今回の場合、エントリ関数第 1 引数の
```

```
                                * ImageHandle を指定 */
                        struct EFI_DEVICE_PATH_PROTOCOL *DevicePath,
                                /* ロードするイメージへのパス */
                        void *SourceBuffer,
                                /* NULL で無ければ、ロードされるイメージの
                                 * コピーを持つアドレスを示す
                                 * 今回は特に使用しないため NULL を指定 */
                        unsigned long long SourceSize,
                                /* SourceBuffer のサイズを指定（単位:バイト）
                                 * SourceBuffer を使用しないため 0 を指定 */
                        void **ImageHandle
                                /* 生成されたイメージハンドルを格納する
                                 * ポインタ */
                        );
                . . .
        } *BootServices;
};
```

使用例はリスト 3.13 の通りです。

▼**リスト** 3.13　LoadImage() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle __attribute__ ((unused)),
 5:           struct EFI_SYSTEM_TABLE *SystemTable)
 6: {
 7:     struct EFI_DEVICE_PATH_PROTOCOL *dev_path;      /* 追加 */
 8:     unsigned long long status;      /* 追加 */
 9:     void *image;
10:
11:     efi_init(SystemTable);
12:     ST->ConOut->ClearScreen(ST->ConOut);
13:
14:     dev_path = DPFTP->ConvertTextToDevicePath(L"\\test.efi");
15:     puts(L"dev_path: ");
16:     puts(DPTTP->ConvertDevicePathToText(dev_path, FALSE, FALSE));
17:     puts(L"\r\n");
18:
19:     /* 追加（ここから）*/
20:     status = ST->BootServices->LoadImage(FALSE, ImageHandle, dev_path, NULL,
21:                                          0, &image);
22:     assert(status, L"LoadImage");
23:     puts(L"LoadImage: Success!\r\n");
24:     /* 追加（ここまで）*/
25:
26:     while (TRUE);
27: }
```

デバイスパスの生成までは前節のままで、生成したデバイスパス"dev_path"を LoadImage() に渡しています。LoadImage() に失敗した場合は assert() でログを出して止まり、成功した場合は puts() で"LoadImage: Success!"が表示されます。

リスト 3.13 を実行してみます。なお、ロードされる側の"test.efi"は、適当な UEFI アプリケーションをビルドして用意しておけばよいのですが、1 点だけビルド時の注意点がありますので、後述のコラムをご覧ください。ここでは、前著"フルスクラッチで作る!UEFI ベアメタルプログラミング (パート 1)"のサンプルプログラム"sample1_1_hello_uefi"を使用します[*1]。そしてビルドした efi バイナリを"test.efi"とリネームし、起動ディスクとして使用する USB フラッシュメモリのルート直下へ配置したこととします。

実行してみると、結果は図 3.3 の通りです。

▲図 3.3　LoadImage() の実行例

失敗しています。EFI_STATUS の"0x80000000 0000000E"は、最上位の 0x8 が"エラーである"事を示し、下位の"0xE"が"EFI_NOT_FOUND"を示します[*2]。

LoadImage() がイメージを見つけられなかった理由はデバイスパスが完全では無かったからです。実は、デバイスパスにはもう少し付け加えなければならない要素があり、次節から説明します。

[*1] https://github.com/cupnes/c92_uefi_bare_metal_programming_samples で公開しています

[*2] 詳しくは仕様書の"Appendix D Status Codes"を見てみてください

ロードされる UEFI アプリケーションは"-shared"オプションを付けてビルド

LoadImage() でロードされる UEFI アプリケーションの実行バイナリはリロケータブルなバイナリである必要があります。
そのため、Makefile の"fs/EFI/BOOT/BOOTX64.EFI"ターゲットで"x86_64-w64-mingw32-gcc"のオプションを指定している箇所へ"-shared"オプションを追加してください (リスト 3.14)。

▼**リスト** 3.14　"-shared"**の追加例**

```
all: fs/EFI/BOOT/BOOTX64.EFI

fs/EFI/BOOT/BOOTX64.EFI: main.c
        mkdir -p fs/EFI/BOOT
        x86_64-w64-mingw32-gcc -Wall -Wextra -e efi_main -nostdinc \
        -nostdlib -fno-builtin -Wl,--subsystem,10 -shared -o $@ $<
        #                                          ^^^^^^^
        #                                           追加
...
```

3.4　デバイスを指定するパス指定

これまで、"\EFI\BOOT\BOOTX64.EFI"や"\test.efi"のようにパスを指定してみました。ただし、考えてみれば、これでは「どのデバイスであるか」を指定できておらず、例えば、ノート PC 本体のハードディスクと USB フラッシュメモリのどちらを指しているのかが分かりません。そもそも、デバイスをパスの形で指定できるからこそ"デバイスパス"であるにも関わらず、"デバイス"の箇所を指定していませんでした。

それでは、デバイスの部分はどのように指定するのかを確認するために、例として、起動時から実行している UEFI アプリケーション自身のデバイス部分のパスを見てみます。

サンプルのディレクトリは"033_loaded_image_protocol_device_handle"です。

実は、デバイス部分のパスを得るための情報も EFI_LOADED_IMAGE_PROTOCOL にあります (リスト 3.15)。

▼**リスト** 3.15　(**再掲**)EFI_LOADED_IMAGE_PROTOCOL **の定義** (efi.h **より**)

```
struct EFI_LOADED_IMAGE_PROTOCOL {
        unsigned int Revision;
        void *ParentHandle;
        struct EFI_SYSTEM_TABLE *SystemTable;
        // Source location of the image
        void *DeviceHandle;
        struct EFI_DEVICE_PATH_PROTOCOL *FilePath;
        void *Reserved;
        // Image's load options
        unsigned int LoadOptionsSize;
        void *LoadOptions;
        // Location where image was loaded
        void *ImageBase;
        unsigned long long ImageSize;
        enum EFI_MEMORY_TYPE ImageCodeType;
        enum EFI_MEMORY_TYPE ImageDataType;
        unsigned long long (*Unload)(void *ImageHandle);
};
```

"Source location of the image"のコメントが書かれている箇所には"FilePath"の他に"DeviceHandle"があります。この"DeviceHandle"を使用してデバイス部分のパスを得ることができます。

デバイスパス (EFI_DEVICE_PATH_PROTOCOL) を得る方法は、EFI_LOADED_IMAGE_PROTOCOL を取得する時と同じく、OpenProtocol() を使用します。そして、OpenProtocol() の第 1 引数にこの"DeviceHandle"を指定することで、"DeviceHandle"の"EFI_DEVICE_PATH_PROTOCOL"を得ることができます (リスト 3.16)。

▼**リスト** 3.16　DeviceHandle **のデバイスパスを取得する例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
 5: {
 6:     struct EFI_LOADED_IMAGE_PROTOCOL *lip;
 7:     struct EFI_DEVICE_PATH_PROTOCOL *dev_path;
 8:     unsigned long long status;
 9:
10:     efi_init(SystemTable);
11:     ST->ConOut->ClearScreen(ST->ConOut);
12:
13:     /* ImageHandle の EFI_LOADED_IMAGE_PROTOCOL(lip) を取得 */
14:     status = ST->BootServices->OpenProtocol(
15:             ImageHandle, &lip_guid, (void **)&lip, ImageHandle, NULL,
16:             EFI_OPEN_PROTOCOL_GET_PROTOCOL);
17:     assert(status, L"OpenProtocol(lip)");
18:
19:     /* lip->DeviceHandle の EFI_DEVICE_PATH_PROTOCOL(dev_path) を取得 */
20:     status = ST->BootServices->OpenProtocol(
```

```
21:             lip->DeviceHandle, &dpp_guid, (void **)&dev_path, ImageHandle,
22:             NULL, EFI_OPEN_PROTOCOL_GET_PROTOCOL);
23:     assert(status, L"OpenProtocol(dpp)");
24:
25:     /* dev_path をテキストへ変換し表示 */
26:     puts(L"dev_path: ");
27:     puts(DPTTP->ConvertDevicePathToText(dev_path, FALSE, FALSE));
28:     puts(L"\r\n");
29:
30:     while (TRUE);
31: }
```

なお、上記の実装に合わせて、efi.h と efi.c へ EFI_DEVICE_PATH_PROTOCOL の GUID を定義します (リスト 3.17、リスト 3.18)。

▼**リスト** 3.17　EFI_DEVICE_PATH_PROTOCOL **の** GUID **の** extern(efi.h **より**)

```
extern struct EFI_GUID dpp_guid;
```

▼**リスト** 3.18　EFI_DEVICE_PATH_PROTOCOL **の** GUID **の定義** (efi.c **より**)

```
struct EFI_GUID dpp_guid = {0x09576e91, 0x6d3f, 0x11d2,
                            {0x8e, 0x39, 0x00, 0xa0,
                             0xc9, 0x69, 0x72, 0x3b}};
```

実行すると、図 3.4 の様に表示されます。

▲**図** 3.4　DeviceHandel **のデバイスパスの表示例**

"PciRoot"から始まるテキストが表示されており、「デバイス」の「パス」っぽいですね。本書で扱う範囲では立ち入らずに済むためあまり説明しませんが、UEFI の概念ではこの様に、デバイスをパスの形で指定して扱います。ストレージデバイスだけでなく、PC に接続されるマウス等のデバイスもこの様にパスの形で指定します。

3.5　デバイスパスを作成してみる (その 2)

ここまでで、"PciRoot"から始まる正に"デバイス"を指定しているパスと、"\test.efi"といったよく見るファイルのパスの 2 つのパスを確認しました。実は、この 2 つを連結する

ことでフルパスとなります。

サンプルのディレクトリは"034_create_devpath_2"です。

パス連結等のパスを操作する関数群を持つのが EFI_DEVICE_PATH_UTILITIES_PROTOCOL です。ここでは、AppendDeviceNode() を使用します (リスト 3.19)。

▼**リスト** 3.19 EFI_DEVICE_PATH_UTILITIES_PROTOCOL.AppendDeviceNode() **の定義** (efi.h **より**)

```
struct EFI_DEVICE_PATH_UTILITIES_PROTOCOL {
        unsigned long long _buf[3];
        /* デバイスパスとノードを連結し、連結結果のパスを返す */
        struct EFI_DEVICE_PATH_PROTOCOL *(*AppendDeviceNode)(
                const struct EFI_DEVICE_PATH_PROTOCOL *DevicePath,
                        /* 連結するデバイスパスを指定 */
                const struct EFI_DEVICE_PATH_PROTOCOL *DeviceNode
                        /* 連結するデバイスノードを指定 */
                );
};
```

"デバイスノード"は、デバイスパスの部分で、"\"で区切られた 1 要素のことです。例えば、"\EFI\BOOT\BOOTX64.EFI"の場合、"EFI"や"BOOT"がデバイスノードです。そのため、AppendDeviceNode() は、デバイスパスの末尾に 1 つのノードを追加する関数、となります[*3]。

今回の場合、ファイルパスに相当する部分は"test.efi"という単一のデバイスノードで済むため、AppendDeviceNode() を使用します。それに併せて、テキストからデバイスパスを生成する関数群を持つ EFI_DEVICE_PATH_UTILITIES_PROTOCOL には ConvertTextToDeviceNode() という関数もあるので、ここではこちらの関数を使用します (リスト 3.20)。

▼**リスト** 3.20 ConvertTextToDeviceNode() **の定義** (efi.h **より**)

```
struct EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL {
        /* 指定されたテキストをデバイスノードへ変換、変換後のデバイスノードを返す */
        struct EFI_DEVICE_PATH_PROTOCOL *(*ConvertTextToDeviceNode) (
                const unsigned short *TextDeviceNode
                        /* デバイスノードへ変換するテキストを指定 */
                );
        struct EFI_DEVICE_PATH_PROTOCOL *(*ConvertTextToDevicePath) (
                const unsigned short *TextDevicePath);
};
```

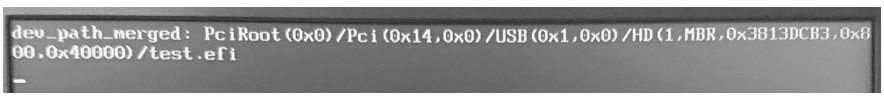
[*3] EFI_DEVICE_PATH_UTILITIES_PROTOCOL の中にはデバイスパスにデバイスパスを連結する関数もあるのですが、使用しないため省略します。

以上を踏まえ、デバイスパスとデバイスノードを連結する例はリスト 3.21 の通りです。

▼**リスト** 3.21　AppendDeviceNode() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
 5: {
 6:     struct EFI_LOADED_IMAGE_PROTOCOL *lip;
 7:     struct EFI_DEVICE_PATH_PROTOCOL *dev_path;
 8:     struct EFI_DEVICE_PATH_PROTOCOL *dev_node;              /* 追加 */
 9:     struct EFI_DEVICE_PATH_PROTOCOL *dev_path_merged;       /* 追加 */
10:     unsigned long long status;
11:
12:     efi_init(SystemTable);
13:     ST->ConOut->ClearScreen(ST->ConOut);
14:
15:     /* ImageHandle の EFI_LOADED_IMAGE_PROTOCOL(lip) を取得 */
16:     status = ST->BootServices->OpenProtocol(
17:             ImageHandle, &lip_guid, (void **)&lip, ImageHandle, NULL,
18:             EFI_OPEN_PROTOCOL_GET_PROTOCOL);
19:     assert(status, L"OpenProtocol(lip)");
20:
21:     /* lip->DeviceHandle の EFI_DEVICE_PATH_PROTOCOL(dev_path) を取得 */
22:     status = ST->BootServices->OpenProtocol(
23:             lip->DeviceHandle, &dpp_guid, (void **)&dev_path, ImageHandle,
24:             NULL, EFI_OPEN_PROTOCOL_GET_PROTOCOL);
25:     assert(status, L"OpenProtocol(dpp)");
26:
27:     /* 追加・変更（ここから） */
28:     /* "test.efi"のデバイスノードを作成 */
29:     dev_node = DPFTP->ConvertTextToDeviceNode(L"test.efi");
30:
31:     /* dev_path と dev_node を連結 */
32:     dev_path_merged = DPUP->AppendDeviceNode(dev_path, dev_node);
33:
34:     /* dev_path_merged をテキストへ変換し表示 */
35:     puts(L"dev_path_merged: ");
36:     puts(DPTTP->ConvertDevicePathToText(dev_path_merged, FALSE, FALSE));
37:     puts(L"\r\n");
38:     /* 追加・変更（ここまで） */
39:
40:     while (TRUE);
41: }
```

実行すると図 3.5 の様に表示されます。



▲**図** 3.5　AppendDeviceNode() **の実行例**

デバイスパスとデバイスノードを表す EFI_DEVICE_PATH_PROTOCOL について

デバイスパスとデバイスノードは型は分かれておらず、共に EFI_DEVICE_PATH_PROTOCOL です。
どういうことかというと、EFI_DEVICE_PATH_PROTOCOL がデバイスノードで、EFI_DEVICE_PATH_PROTOCOL が連結することでデバイスパスとなります。
それでは、EFI_DEVICE_PATH_PROTOCOL はどういう定義になっているかというと、リスト 3.22 の様になっています。

▼**リスト** 3.22　EFI_DEVICE_PATH_PROTOCOL **の定義** (efi.h **より**)

```
struct EFI_DEVICE_PATH_PROTOCOL {
        unsigned char Type;
        unsigned char SubType;
        unsigned char Length[2];
};
```

リスト 3.22 を見ると、リンクリストを構成するようなメンバはありません。EFI_DEVICE_PATH_PROTOCOL はメモリ上に連続に並ぶことでパスを構成します。
また、リスト 3.22 を見ると、ファイルパスで必要となる"ファイル名"といった要素を格納するようなメンバがありません。実は、EFI_DEVICE_PATH_PROTOCOL は各種デバイスノードのヘッダ部分のみで、ボディに当たる部分は EFI_DEVICE_PATH_PROTOCOL に続くメモリ領域へ配置します。そのため、デバイスノードのタイプを"Type"・"SubType"メンバで指定し、ヘッダ・ボディ含めたサイズを"Length"で指定します。
デバイスパス内のノード数を示す要素はどこにもありません。ノード終端を示すデバイスノードを配置することでデバイスノードの終わりを示します。
UEFI では、デバイスパスやノードの構造やメモリ上の配置は意識せずに使用できるよう、EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL や EFI_DEVICE_PATH_UTILITIES_PROTOCOL といったプロトコルを用意しています。AppendDeviceNode() でデバイスパスへデバイスノードを追加する際も、ノード終端の要素は自動で配置してくれます。そのため、UEFI のファームウェアを仕様書通りに叩く上では特に意識する必要はありません。

3.6 デバイスパスをロードしてみる (その 2)

それでは、再びデバイスパスのロードを試してみます。

サンプルのディレクトリは"035_load_devpath_2"です。

前節のサンプル (リスト 3.21) へリスト 3.13 で紹介した LoadImage() の処理を追加するだけです (リスト 3.23)。

▼**リスト** 3.23　**フルパスのデバイスパスをロードする例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
 5: {
 6:     struct EFI_LOADED_IMAGE_PROTOCOL *lip;
 7:     struct EFI_DEVICE_PATH_PROTOCOL *dev_path;
 8:     struct EFI_DEVICE_PATH_PROTOCOL *dev_node;
 9:     struct EFI_DEVICE_PATH_PROTOCOL *dev_path_merged;
10:     unsigned long long status;
11:     void *image;    /* 追加 */
12:
13:     efi_init(SystemTable);
14:     ST->ConOut->ClearScreen(ST->ConOut);
15:
16:     /* ImageHandle の EFI_LOADED_IMAGE_PROTOCOL(lip) を取得 */
17:     status = ST->BootServices->OpenProtocol(
18:             ImageHandle, &lip_guid, (void **)&lip, ImageHandle, NULL,
19:             EFI_OPEN_PROTOCOL_GET_PROTOCOL);
20:     assert(status, L"OpenProtocol(lip)");
21:
22:     /* lip->DeviceHandle の EFI_DEVICE_PATH_PROTOCOL(dev_path) を取得 */
23:     status = ST->BootServices->OpenProtocol(
24:             lip->DeviceHandle, &dpp_guid, (void **)&dev_path, ImageHandle,
25:             NULL, EFI_OPEN_PROTOCOL_GET_PROTOCOL);
26:     assert(status, L"OpenProtocol(dpp)");
27:
28:     /* "test.efi"のデバイスノードを作成 */
29:     dev_node = DPFTP->ConvertTextToDeviceNode(L"test.efi");
30:
31:     /* dev_path と dev_node を連結 */
32:     dev_path_merged = DPUP->AppendDeviceNode(dev_path, dev_node);
33:
34:     /* dev_path_merged をテキストへ変換し表示 */
35:     puts(L"dev_path_merged: ");
36:     puts(DPTTP->ConvertDevicePathToText(dev_path_merged, FALSE, FALSE));
37:     puts(L"\r\n");
38:
39:     /* 追加（ここから） */
40:     /* dev_path_merged をロード */
41:     status = ST->BootServices->LoadImage(FALSE, ImageHandle,
42:                                          dev_path_merged, NULL, 0, &image);
43:     assert(status, L"LoadImage");
44:     puts(L"LoadImage: Success!\r\n");
```

```
45:     /* 追加（ここまで） */
46:
47:     while (TRUE);
48: }
```

実行してみると、今度はロードに成功しました (図 3.6)。

▲図 3.6　フルパスのデバイスパスをロードする実行例

3.7　ロードしたイメージを実行してみる

ロードが成功しましたので、いよいよ実行してみます。

サンプルのディレクトリは"036_start_devpath"です。

ロードしたイメージを実行するには EFI_BOOT_SERVICES の StartImage() を使用します (リスト 3.24)。

▼リスト 3.24　StartImage() の定義 (efi.h より)

```
struct EFI_SYSTEM_TABLE {
        ・・・
        struct EFI_BOOT_SERVICES {
                ・・・
                //
                // Image Services
                //
                unsigned long long (*LoadImage)(
                        unsigned char BootPolicy,
                        void *ParentImageHandle,
                        struct EFI_DEVICE_PATH_PROTOCOL *DevicePath,
                        void *SourceBuffer,
                        unsigned long long SourceSize,
                        void **ImageHandle);
                unsigned long long (*StartImage)(
                        void *ImageHandle,
                                /* 実行するイメージハンドル */
                        unsigned long long *ExitDataSize,
                                /* 第 3 引数 ExitData のサイズを指定
                                 * ExitData が NULL の場合、こちらも NULL を指定 */
                        unsigned short **ExitData
                                /* 呼びだされたイメージが
                                 * EFI_BOOT_SERVICES.Exit() 関数で終了した場合に
```

```
                                        * 呼び出し元へ返されるデータのポインタのポインタ
                                        * 使用しないため今回は NULL */
                        );
                . . .
        } *BootServices;
};
```

StartImage() の使用例はリスト 3.25 の通りです。

▼**リスト** 3.25　StartImage() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
 5: {
 6:     struct EFI_LOADED_IMAGE_PROTOCOL *lip;
 7:     struct EFI_DEVICE_PATH_PROTOCOL *dev_path;
 8:     struct EFI_DEVICE_PATH_PROTOCOL *dev_node;
 9:     struct EFI_DEVICE_PATH_PROTOCOL *dev_path_merged;
10:     unsigned long long status;
11:     void *image;
12:
13:     efi_init(SystemTable);
14:     ST->ConOut->ClearScreen(ST->ConOut);
15:
16:     /* ImageHandle の EFI_LOADED_IMAGE_PROTOCOL(lip) を取得 */
17:     status = ST->BootServices->OpenProtocol(
18:             ImageHandle, &lip_guid, (void **)&lip, ImageHandle, NULL,
19:             EFI_OPEN_PROTOCOL_GET_PROTOCOL);
20:     assert(status, L"OpenProtocol(lip)");
21:
22:     /* lip->DeviceHandle の EFI_DEVICE_PATH_PROTOCOL(dev_path) を取得 */
23:     status = ST->BootServices->OpenProtocol(
24:             lip->DeviceHandle, &dpp_guid, (void **)&dev_path, ImageHandle,
25:             NULL, EFI_OPEN_PROTOCOL_GET_PROTOCOL);
26:     assert(status, L"OpenProtocol(dpp)");
27:
28:     /* "test.efi"のデバイスノードを作成 */
29:     dev_node = DPFTP->ConvertTextToDeviceNode(L"test.efi");
30:
31:     /* dev_path と dev_node を連結 */
32:     dev_path_merged = DPUP->AppendDeviceNode(dev_path, dev_node);
33:
34:     /* dev_path_merged をテキストへ変換し表示 */
35:     puts(L"dev_path_merged: ");
36:     puts(DPTTP->ConvertDevicePathToText(dev_path_merged, FALSE, FALSE));
37:     puts(L"\r\n");
38:
39:     /* dev_path_merged をロード */
40:     status = ST->BootServices->LoadImage(FALSE, ImageHandle,
41:                                          dev_path_merged, NULL, 0, &image);
42:     assert(status, L"LoadImage");
43:     puts(L"LoadImage: Success!\r\n");
44:
```

```
45:     /* 追加（ここから） */
46:     /* image の実行を開始する */
47:     status = ST->BootServices->StartImage(image, NULL, NULL);
48:     assert(status, L"StartImage");
49:     puts(L"StartImage: Success!\r\n");
50:     /* 追加（ここまで） */
51:
52:     while (TRUE);
53: }
```

実行例は図 3.7 の通りです。

▲図 3.7　StartImage() の実行例

無事に実行できました！ これで、今後は UEFI アプリケーションをバイナリ単位で分割し、呼び出す事ができます。

そう言えば、前著 (パート 1) の sample1_1_hello_uefi では、CR('\r') が入っていないので、今回のように、別の UEFI アプリケーションから呼び出された場合、改行はしますが、行頭は戻ってないですね。

3.8　Linux を起動してみる: カーネルビルド

現在の Linux カーネルはビルド時の設定 (menuconfig) で UEFI バイナリを生成するように設定できます。Linux カーネルを UEFI バイナリとしてビルドすることで、Linux のイメージ (bzImage) をこれまで説明した UEFI アプリケーションの実行方法で実行させ

ることができます。

サンプルのディレクトリは"037_start_bzImage"です。

3.8.1 ビルド環境を準備

Debian、Ubuntu 等のパッケージ管理システムで APT が使用できる事を前提に説明します。

APT では"apt-get build-dep <**パッケージ名**>"というコマンドで、指定したパッケージ名をビルドするために必要な環境をインストールすることができます。

カーネルイメージは"linux-image-<バージョン>-<アーキテクチャ>"というパッケージ名です。現在使用しているカーネルバージョンとアーキテクチャは"uname -r"コマンドで取得できるので、結果としては、以下のコマンドで Linux カーネルをビルドする環境をインストールできます。

```
$ sudo apt-get build-dep linux-image-$(uname -r)
```

なお、menuconfig を使用するための"libncurses5-dev"パッケージが build-dep でインストールされないので、別途インストールする必要があります。

```
$ sudo apt install libncurses5-dev
```

3.8.2 Linux カーネルのソースコードを準備

ソースコードも APT では"apt-get source <**パッケージ名**>"というコマンドで取得できます。しかし、せっかくなのでここでは本家である kernel.org から最新の安定版をダウンロードして使用してみることにします。

https://www.kernel.org/ へアクセスし、"Latest Stable Kernel:"からダウンロードしてください (図 3.8)。

▲図 3.8　kernel.org のウェブサイト

ダウンロード後は、展開しておいてください。

```
$ tar Jxf linux-<バージョン>.tar.xz
```

3.8.3　ビルド設定を行う

Linux カーネルのビルドの設定を行います。

まず、展開した Linux カーネルのソースコードディレクトリへ移動し、x86_64 向けのデフォルト設定を反映させます。

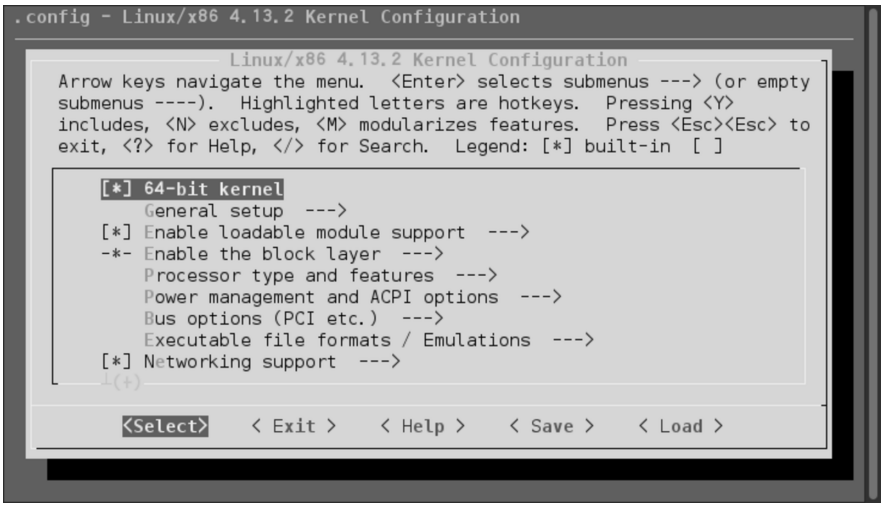
```
$ cd linux-<バージョン>
$ make x86_64_defconfig
```

そして、menuconfig で、UEFI バイナリを生成する設定 (コンフィグシンボル名"CONFIG_EFI_STUB") を有効化します。

menuconfig を起動させます。

```
$ make menuconfig
```

すると、図 3.9 の画面になります。



▲図 3.9　menuconfig 画面

リスト 3.26 の場所にある CONFIG_EFI_STUB を有効化します。

▼リスト 3.26　CONFIG_EFI_STUB の場所

```
Processor type and features  --->
  [*] EFI runtime service support
  [*]   EFI stub support  <== 有効化
```

3.8.4　ビルドする

make コマンドでビルドします。-j オプションでビルドのスレッド数を指定できます。ここでは nproc コマンドで取得した CPU コア数を-j オプションに渡しています。

```
$ make -j $(nproc)
```

ビルドには時間がかかりますので、のんびりと待ちましょう。
ビルドが完了すると、"bzImage"というイメージファイルができあがっています。

```
$ ls arch/x86/boot/bzImage
arch/x86/boot/bzImage
```

3.8.5 起動してみる

arch/x86/boot/bzImage を USB フラッシュメモリ等のこれまで"test.efi"を配置していた場所に"bzImage.efi"という名前で配置し、リスト 3.25 の"test.efi"を"bzImage.efi"へ変更するだけで、Linux カーネルを起動できます図 3.10。

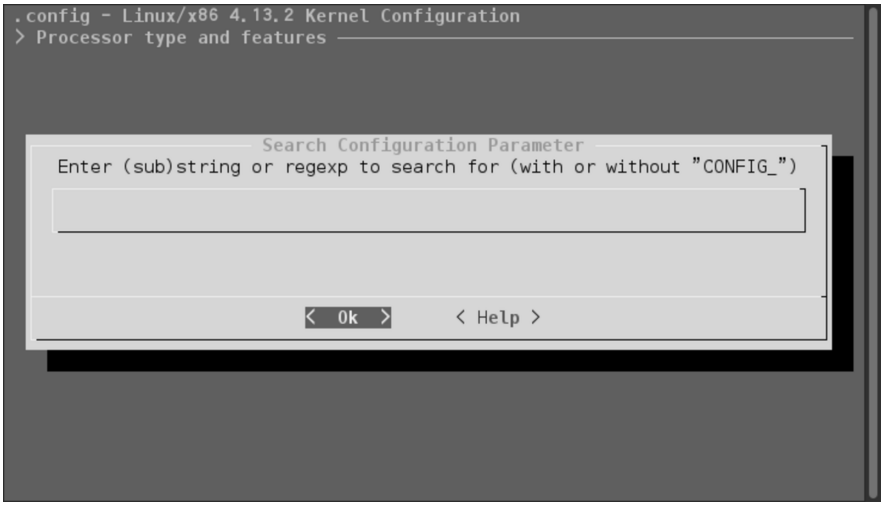
▲図 3.10　Linux 起動。。ただし失敗

Linux カーネルの起動は始まりますが、ルートファイルシステムを何も準備していないのでカーネルパニックで止まります。

補足: menuconfig では検索が使えます

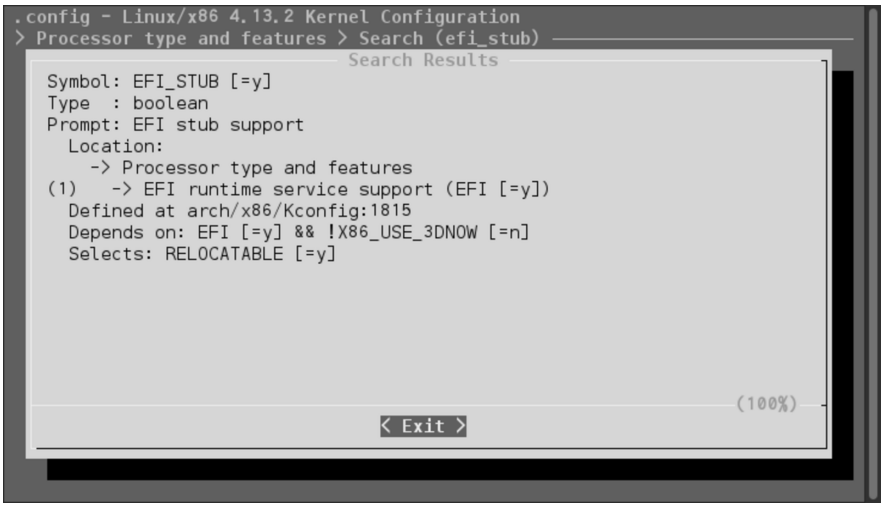
もし、前述の場所に設定項目が無い場合は、menuconfig の検索機能を使ってみてください。

menuconfig 画面内で"/(スラッシュ)"キーを押下すると図 3.11 の画面になります。



▲図 3.11　menuconfig 検索画面

この画面で"efi_stub"の様に検索したいキーワードを入力[*4]し、Enter を押下すると、コンフィギュレーションの依存関係や設定項目の場所等の情報が表示されます (図 3.12)。なお、検索結果の画面ではコンフィグシンボル名の"CONFIG_"は省略されています。



▲図 3.12　menuconfig 検索結果

[*4] 大文字/小文字はどちらでも構いません

図 3.12 では、"Location"の欄にコンフィグの場所が、"Prompt"の欄に設定項目名が記載されています。

"Location"と"Prompt"が示す場所にも設定項目が無い場合は、特にコンフィグの依存関係 ("Depends on") を見てみると良いです。図 3.12 の場合、"Depends on"は、"CONFIG_EFI が有効で、かつ CONFIG_X86_USE_3DNOW が無効であること"を示しており、"[]"の中は現在の設定値です。現在の設定値が要求している依存関係と一致していない場合、その設定項目は menuconfig 画面内に現れませんので、先に依存するコンフィグを変更する必要があります。

3.9 Linux を起動してみる: カーネル起動オプション指定

前節では、Linux カーネルが起動時に参照するルートファイルシステム ("root=") 等のオプションを指定していなかったため、カーネルパニックに陥ってしまっていました。そこで、カーネル起動時のオプションを設定してみます。

サンプルのディレクトリは"038_start_bzImage_options"です。

UEFI アプリケーション実行時のオプション (引数) は、EFI_LOADED_IMAGE_PROTOCOL の `unsigned int LoadOptionsSize`メンバと `void *LoadOptions`メンバへ行います (リスト 3.27)。

▼**リスト** 3.27　(**再掲**)EFI_LOADED_IMAGE_PROTOCOL **の定義** (efi.h **より**)

```
struct EFI_LOADED_IMAGE_PROTOCOL {
        unsigned int Revision;
        void *ParentHandle;
        struct EFI_SYSTEM_TABLE *SystemTable;
        // Source location of the image
        void *DeviceHandle;
        struct EFI_DEVICE_PATH_PROTOCOL *FilePath;
        void *Reserved;
        // Image's load options
        unsigned int LoadOptionsSize;
                /* LoadOptions メンバのサイズを指定（バイト） */
        void *LoadOptions;
                /* イメージバイナリのロードオプションへのポインタを指定 */
        // Location where image was loaded
        void *ImageBase;
        unsigned long long ImageSize;
        enum EFI_MEMORY_TYPE ImageCodeType;
        enum EFI_MEMORY_TYPE ImageDataType;
        unsigned long long (*Unload)(void *ImageHandle);
};
```

使用例はリスト 3.28 の通りです。

▼**リスト** 3.28　LoadOptionsSize と LoadOptions の使用例 (main.c より)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
 5: {
 6:     struct EFI_LOADED_IMAGE_PROTOCOL *lip;
 7:     struct EFI_LOADED_IMAGE_PROTOCOL *lip_bzimage;  /* 追加 */
 8:     struct EFI_DEVICE_PATH_PROTOCOL *dev_path;
 9:     struct EFI_DEVICE_PATH_PROTOCOL *dev_node;
10:     struct EFI_DEVICE_PATH_PROTOCOL *dev_path_merged;
11:     unsigned long long status;
12:     void *image;
13:     unsigned short options[] = L"root=/dev/sdb2 init=/bin/sh rootwait";
14:                                                              /* 追加 */
15:
16:     efi_init(SystemTable);
17:     ST->ConOut->ClearScreen(ST->ConOut);
18:
19:     /* ImageHandle の EFI_LOADED_IMAGE_PROTOCOL(lip) を取得 */
20:     status = ST->BootServices->OpenProtocol(
21:             ImageHandle, &lip_guid, (void **)&lip, ImageHandle, NULL,
22:             EFI_OPEN_PROTOCOL_GET_PROTOCOL);
23:     assert(status, L"OpenProtocol(lip)");
24:
25:     /* lip->DeviceHandle の EFI_DEVICE_PATH_PROTOCOL(dev_path) を取得 */
26:     status = ST->BootServices->OpenProtocol(
27:             lip->DeviceHandle, &dpp_guid, (void **)&dev_path, ImageHandle,
28:             NULL, EFI_OPEN_PROTOCOL_GET_PROTOCOL);
29:     assert(status, L"OpenProtocol(dpp)");
30:
31:     /* "bzImage.efi"のデバイスノードを作成 */
32:     dev_node = DPFTP->ConvertTextToDeviceNode(L"bzImage.efi");
33:
34:     /* dev_path と dev_node を連結 */
35:     dev_path_merged = DPUP->AppendDeviceNode(dev_path, dev_node);
36:
37:     /* dev_path_merged をテキストへ変換し表示 */
38:     puts(L"dev_path_merged: ");
39:     puts(DPTTP->ConvertDevicePathToText(dev_path_merged, FALSE, FALSE));
40:     puts(L"\r\n");
41:
42:     /* dev_path_merged をロード */
43:     status = ST->BootServices->LoadImage(FALSE, ImageHandle,
44:                                          dev_path_merged, NULL, 0, &image);
45:     assert(status, L"LoadImage");
46:     puts(L"LoadImage: Success!\r\n");
47:
48:     /* 追加（ここから） */
49:     /* カーネル起動オプションを設定 */
50:     status = ST->BootServices->OpenProtocol(
51:             image, &lip_guid, (void **)&lip_bzimage, ImageHandle, NULL,
52:             EFI_OPEN_PROTOCOL_GET_PROTOCOL);
53:     assert(status, L"OpenProtocol(lip_bzimage)");
54:     lip_bzimage->LoadOptions = options;
55:     lip_bzimage->LoadOptionsSize =
56:             (strlen(options) + 1) * sizeof(unsigned short);
```

```
57:     /* 追加（ここまで） */
58:
59:     /* image の実行を開始する */
60:     status = ST->BootServices->StartImage(image, NULL, NULL);
61:     assert(status, L"StartImage");
62:     puts(L"StartImage: Success!\r\n");
63:
64:     while (TRUE);
65: }
```

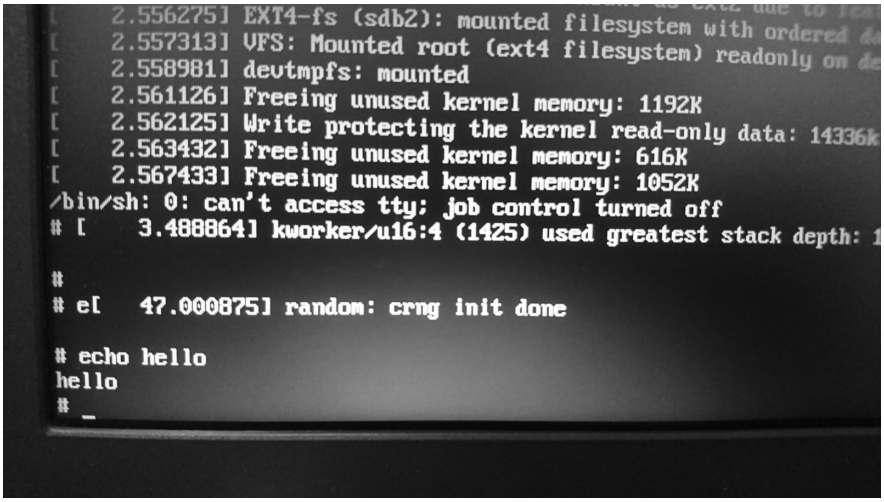
リスト 3.28 では、カーネル起動オプションとして"root=/dev/sdb2 init=/bin/sh rootwait"を指定しています。

"root=/dev/sdb2"は、ルートファイルシステムを配置しているパーティションへのデバイスファイルの指定です。筆者が実験で使用している PC の場合、内蔵の HDD を"sda"として認識し、接続した USB フラッシュメモリを"sdb"として認識します。そのため、USB フラッシュメモリの第 2 パーティションを指定するため、"/dev/sdb2"としています。

"init=/bin/sh"は起動時にカーネルが最初に実行する実行バイナリの指定です。何も指定しない場合、"/sbin/init"が実行されますが、起動直後にシェルを立ち上げてしまおうと、"/bin/sh"を指定しています。そのため、後述しますが、USB フラッシュメモリの第 2 パーティションへ/bin/sh を配置しておく必要があります。

"rootwait"は、Linux カーネルがルートファイルシステムを検出するタイミングを遅らせるオプションです。USB フラッシュメモリ等の場合、デバイスが検出されるタイミングは非同期です。デバドラの検出より先に Linux カーネルのルートファイルシステム検出を行おうとするとルートファイルシステムの検出に失敗してしまうので、ルートファイルシステムの検出を遅延させるためのオプションです。

実行例は図 3.13 の通りです。



▲図 3.13　起動オプションを指定した実行例

晴れて、シェルを起動させることができました。

補足: ルートファイルシステムの作り方

今回の場合の様に「シェルさえ動けば良い」のであれば、ルートファイルシステムの構築には"BusyBox"が便利です。BusyBox は"組み込み Linux のスイスアーミーナイフ"と呼ばれるもので、一つの"busybox"という実行バイナリへシンボリックリンクを張ることで、色々なコマンドが使用できるようになるというものです。

Debian、あるいは Ubuntu 等の APT を使用できる環境では、"busybox-static"というパッケージをインストールすることで、必要なライブラリが静的に埋め込まれた busybox バイナリを取得できます。

```
$ sudo apt install busybox-static
・・・
$ ls /bin/busybox
/bin/busybox
```

インストールすると、/bin/busybox へ busybox の実行バイナリが配置されます。この実行バイナリを USB フラッシュメモリ第 2 パーティションへ以下の様に配置すれば良いです。("sh"は"busybox"へのシンボリックリンクです。)

▼ busybox バイナリの配置

```
USB フラッシュメモリ第 2 パーティション
└── bin/
    ├── busybox
    └── sh -> busybox
```

なお、基本的なコマンドは BusyBox で事足りますが、「apt を使用したい」等の場合はルートファイルシステムの構築が面倒になってきます。

Debian から公開されている debootstrap というコマンドを使用すると、apt 等が使用できる最低限の Debian のルートファイルシステムをコマンド一つで構築できます。

```
$ sudo debootstrap <debian のバージョン> <作成先のディレクトリ>
```

例えば、USB フラッシュメモリの第 2 パーティションを/mnt/storage へマウントしている時、sid(unstable) の Debian ルートファイルシステムを/mnt/storage へ構築するコマンドは以下の通りです。

```
$ sudo debootstrap sid /mnt/storage
```

第 4 章

タイマーイベント

タイマーイベントを扱うことで、ある処理の中で時間経過を待ったり、イベント通知関数を使用することで時間経過後に非同期に処理を行わせたりすることができます。

この章ではそれぞれのやり方を紹介します。

4.1 時間経過を待つ

EFI_BOOT_SERVICES の CreateEvent() と SetTimer() と WaitForEvent() で時間経過を待つことができます (リスト 4.1)。なお、WaitForEvent() は前著 (パート 1) で紹介したので説明は省略します。

サンプルのディレクトリは"040_evt_timer_blocking"です。

▼**リスト** 4.1　CreateEvent() **と** SetTimer() **と** WaitForEvent() **の定義** (efi.h **より**)

```
#define EVT_TIMER                               0x80000000
#define EVT_RUNTIME                             0x40000000
#define EVT_NOTIFY_WAIT                                 0x00000100
#define EVT_NOTIFY_SIGNAL                       0x00000200
#define EVT_SIGNAL_EXIT_BOOT_SERVICES           0x00000201
#define EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE       0x60000202

enum EFI_TIMER_DELAY {
        TimerCancel,    /* 設定されているトリガー時間をキャンセル */
        TimerPeriodic,  /* 現時刻からの周期的なトリガー時間を設定 */
        TimerRelative   /* 現時刻から 1 回のみのトリガー時間を設定 */
};

struct EFI_SYSTEM_TABLE {
        ・・・
        struct EFI_BOOT_SERVICES {
                ・・・
                //
                // Event & Timer Services
```

```
                //
                /* 第 1 引数 Type で指定したイベントを生成し、
                 * 第 5 引数 Event へ格納する */
                unsigned long long (*CreateEvent)(
                        unsigned int Type,
                                /* イベントタイプを指定
                                 * 使用できる定数は上記の通り
                                 * この節では EVT_TIMER を扱う */
                        unsigned long long NotifyTpl,
                                /* イベントの通知関数実行時のタスク優先度レベル
                                 * 詳しくは次節で説明
                                 * この節では通知関数を使わない為、0 指定 */
                        void (*NotifyFunction)(void *Event, void *Context),
                                /* イベント発生時に実行する関数（通知関数）
                                 * この節では使わない為、NULL 指定*/
                        void *NotifyContext,
                                /* 通知関数へ渡す引数を指定
                                 * この節では通知関数を使わない為、NULL 指定 */
                        void *Event
                                /* 生成されたイベントを格納するポインタ */
                        );
                /* イベント発生のトリガー時間を設定する */
                unsigned long long (*SetTimer)(
                        void *Event,
                                /* タイマーイベントを発生させるイベントを指定
                                 * CreateEvent() で作成したイベントを指定する */
                        enum EFI_TIMER_DELAY Type,
                                /* タイマーイベントのタイプ指定
                                 * 指定できる定数は上記の通り
                                 * この節では TimerRelative を使用 */
                        unsigned long long TriggerTime
                                /* 100ns 単位でトリガー時間を設定
                                 * 0 を指定した場合、
                                 * - TimerPeriodic: 毎 tick[a]毎にイベントを発生
                                 * - TimerRelative: 次 tick でイベントを発生
                                 * この節では 1 秒（10000000）を指定 */
                        );
                unsigned long long (*WaitForEvent)(
                        unsigned long long NumberOfEvents,
                        void **Event,
                        unsigned long long *Index);
                ...
        } *BootServices;
};
```

[a] "tick"とはシステムで扱う時間の単位で、語源は時計の"チクタク"。例えば 1ms 毎のタイマー割り込みで時間を管理しているシステムなら tick は 1ms。UEFI の場合どうなのかはまだ分かってないです。ごめんなさい。

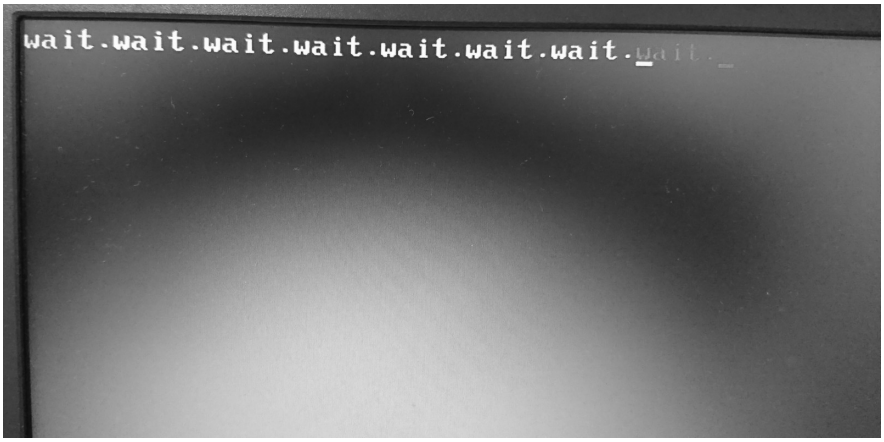
使用例はリスト 4.2 の通りです。

▼**リスト** 4.2　CreateEvent() **と** SetTimer() **と** WaitForEvent() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle __attribute__ ((unused)),
 5:           struct EFI_SYSTEM_TABLE *SystemTable)
 6: {
 7:     unsigned long long status;
 8:     void *tevent;
 9:     void *wait_list[1];
10:     unsigned long long idx;
11:
12:     efi_init(SystemTable);
13:     ST->ConOut->ClearScreen(ST->ConOut);
14:
15:     /* タイマーイベントを作成し、tevent へ格納 */
16:     status = ST->BootServices->CreateEvent(EVT_TIMER, 0, NULL, NULL,
17:                                            &tevent);
18:     assert(status, L"CreateEvent");
19:
20:     /* WaitForEvent() へ渡す為にイベントリストを作成 */
21:     wait_list[0] = tevent;
22:
23:     while (TRUE) {
24:             /* tevent へ 1 秒のトリガー時間を設定 */
25:             status = ST->BootServices->SetTimer(tevent, TimerRelative,
26:                                                 10000000);
27:             assert(status, L"SetTimer");
28:
29:             /* イベント発生を待つ */
30:             status = ST->BootServices->WaitForEvent(1, wait_list, &idx);
31:             assert(status, L"WaitForEvent");
32:
33:             /* 画面へ"wait."を出力 */
34:             puts(L"wait.");
35:     }
36: }
```

リスト 4.2 では、SetTimer() でトリガー時間を設定し、WaitForEvent() でイベント発生を待つ事で、1 秒毎に"wait."が画面出力されます。

実行例は図 4.1 の通りです。



▲図 4.1　CreateEvent() と SetTimer() と WaitForEvent() の実行例

4.2　イベント発生時に呼び出される関数を登録する

CreateEvent() では、イベント発生時に指定した関数を呼び出すように設定できます。
サンプルのディレクトリは"041_evt_timer_nonblocking"です。
前節の CreateEvent() 定義の説明箇所のみを再掲しますリスト 4.3。

▼リスト 4.3　(再掲)CreateEvent() の定義 (efi.h より)

```
/* 第 1 引数 Type で指定したイベントを生成し、
 * 第 5 引数 Event へ格納する */
unsigned long long (*CreateEvent)(
        unsigned int Type,
                /* イベントタイプを指定
                 * 使用できる定数は上記の通り
                 * この節では EVT_TIMER を扱う */
        unsigned long long NotifyTpl,
                /* イベントの通知関数実行時のタスク優先度レベル
                 * 詳しくは次節で説明
                 * この節では通知関数を使わない為、0 指定 */
        void (*NotifyFunction)(void *Event, void *Context),
                /* イベント発生時に実行する関数（通知関数）
                 * この節では使わない為、NULL 指定*/
        void *NotifyContext,
                /* 通知関数へ渡す引数を指定
                 * この節では通知関数を使わない為、NULL 指定 */
        void *Event
                /* 生成されたイベントを格納するポインタ */
        );
```

使用例をリスト 4.4 に示します。

▼**リスト** 4.4　CreateEvent() **の通知関数設定例**

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void timer_callback(void *event __attribute__ ((unused)),
 5:                 void *context __attribute__ ((unused)))
 6: {
 7:     puts(L"wait.");
 8: }
 9:
10: void efi_main(void *ImageHandle __attribute__ ((unused)),
11:           struct EFI_SYSTEM_TABLE *SystemTable)
12: {
13:     unsigned long long status;
14:     void *tevent;
15:
16:     efi_init(SystemTable);
17:     ST->ConOut->ClearScreen(ST->ConOut);
18:
19:     /* タイマーイベントを作成し、tevent へ格納 */
20:     status = ST->BootServices->CreateEvent(EVT_TIMER | EVT_NOTIFY_SIGNAL,
21:                                            TPL_CALLBACK, timer_callback,
22:                                            NULL, &tevent);
23:     assert(status, L"CreateEvent");
24:
25:     /* tevent へ 1 秒の周期トリガー時間を設定 */
26:     status = ST->BootServices->SetTimer(tevent, TimerPeriodic,
27:                                         10000000);
28:     assert(status, L"SetTimer");
29:
30:     while (TRUE);
31: }
```

リスト 4.4 では、SetTimer() へ周期タイマー (TimerPeriodic) を設定してみました。1 秒周期で timer_callback() が呼び出されます。

実行例は図 4.2 の通りです。

▲図 4.2　CreateEvent() の通知関数実行例

第 5 章

BootServices や RuntimeServices のその他の機能

EFI_BOOT_SERVICES や、まだ機能の紹介はしていなかった EFI_RUNTIME_SERVICES には、他にも色々な機能があります。この章ではそれらの一部を紹介します。

5.1 メモリアロケータを使う

UEFI のファームウェアはメモリアロケータを持っていて、EFI_BOOT_SERVICES の AllocatePool() と FreePool() で利用できます (リスト 5.1)。

サンプルのディレクトリは"050_bs_malloc"です。

▼**リスト** 5.1　AllocatePool() **と** FreePool() **の定義** (efi.h **より**)

```
enum EFI_MEMORY_TYPE {
        ・・・
        EfiLoaderData,
                /* ロードされた UEFI アプリケーションのデータと
                 * UEFI アプリケーションのデフォルトのデータアロケーション領域 */
        ・・・
};

struct EFI_SYSTEM_TABLE {
        ・・・
        struct EFI_BOOT_SERVICES {
                ・・・
                //
                // Memory Services
                //
                unsigned long long _buf3[3];
                unsigned long long (*AllocatePool)(
```

```
                        enum EFI_MEMORY_TYPE PoolType,
                                /* どこのメモリプールから確保するかを指定
                                 * この節では"EfiLoaderData"を指定 */
                        unsigned long long Size,
                                /* 確保するサイズをバイト単位で指定 */
                        void **Buffer
                                /* 確保した領域の先頭アドレスを格納するポインタ
                                 * のポインタを指定 */
                        );
                unsigned long long (*FreePool)(
                        void *Buffer
                                /* 開放したい領域のポインタを指定 */
                        );
                ・・・
        } *BootServices;
};
```

AllocatePool() 第 1 引数の"PoolType"について、"enum EFI_MEMORY_TYPE"には他にもメモリタイプがありますが、仕様書を読む限り、UEFI アプリケーションのデータを配置する領域は"EfiLoaderData"であるようで、この節では"EfiLoaderData"を使用しています。その他のメモリタイプについては仕様書の"6.2 Memory Allocation Services"を見てみてください。

使用例はリスト 5.2 の通りです。

▼**リスト** 5.2　AllocatePool() **と** FreePool() **の使用例** (main.c **より**)

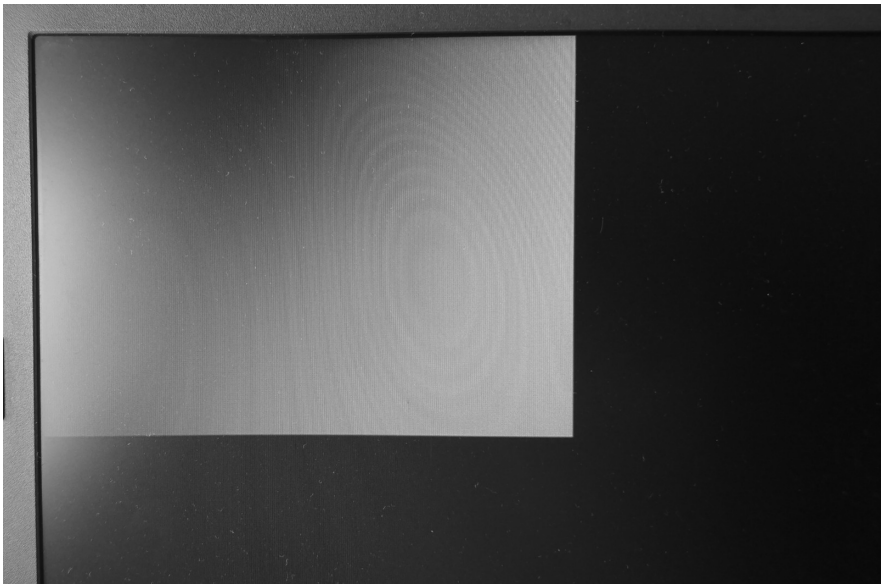
```
 1: #include "efi.h"
 2: #include "common.h"
 3: #include "graphics.h"
 4:
 5: #define IMG_WIDTH  256
 6: #define IMG_HEIGHT 256
 7:
 8: void efi_main(void *ImageHandle __attribute__ ((unused)),
 9:           struct EFI_SYSTEM_TABLE *SystemTable)
10: {
11:     unsigned long long status;
12:     struct EFI_GRAPHICS_OUTPUT_BLT_PIXEL *img_buf, *t;
13:     unsigned int i, j;
14:
15:     efi_init(SystemTable);
16:     ST->ConOut->ClearScreen(ST->ConOut);
17:
18:     /* 画像バッファ用のメモリを確保 */
19:     status = ST->BootServices->AllocatePool(
20:             EfiLoaderData,
21:             IMG_WIDTH * IMG_HEIGHT *
22:             sizeof(struct EFI_GRAPHICS_OUTPUT_BLT_PIXEL),
23:             (void **)&img_buf);
24:     assert(status, L"AllocatePool");
25:
26:     /* 画像を生成 */
```

```
27:     t = img_buf;
28:     for (i = 0; i < IMG_HEIGHT; i++) {
29:             for (j = 0; j < IMG_WIDTH; j++) {
30:                     t->Blue = i;
31:                     t->Green = j;
32:                     t->Red = 0;
33:                     t->Reserved = 255;
34:                     t++;
35:             }
36:     }
37:
38:     /* 画像描画（フレームバッファへ書き込み）*/
39:     blt((unsigned char *)img_buf, IMG_WIDTH, IMG_HEIGHT);
40:
41:     /* 確保したメモリを解放 */
42:     status = ST->BootServices->FreePool((void *)img_buf);
43:     assert(status, L"FreePool");
44:
45:     while (TRUE);
46: }
```

リスト 5.2 では、255x255 の画像用バッファ (img_buf) を確保してピクセルデータを配置し、blt() でフレームバッファへの書き込みを行っています。

生成している画像は X 軸に青色を、Y 軸に緑色を、それぞれ 255 階調で表示するものです。

試してみると図 5.1 のような感じです。印刷は白黒なので、ぜひご自身で試してみてください。



▲図 5.1　AllocatePool() と FreePool() の実行例

5.2　シャットダウンする

メモリ上で動作するだけの UEFI アプリケーションならば電源ボタンで終了しても良いのですが、EFI_RUNTIME_SERVICES の ResetSystem() を使用すると PC をシャットダウンしたり、再起動したりできます。

サンプルのディレクトリは"051_rs_resetsystem"です。

EFI_RUNTIME_SERVICES も EFI_BOOT_SERVICES 同様に EFI_SYSTEM_TABLE のメンバです。EFI_BOOT_SERVICES はブートローダーに対して機能を提供しているのに対し、EFI_RUNTIME_SERVICES は OS 起動後も使用できるという違いがあります。

具体的な契機は EFI_BOOT_SERVICES の ExitBootServices() で、この関数を呼ぶと、それ以降 EFI_BOOT_SERVICES の機能は無効になりますが、EFI_RUNTIME_SERVICES の機能は引き続き使用可能です。

そして、ResetSystem() は EFI_RUNTIME_SERVICES 内でリスト 5.3 の様に定義されています。

▼リスト 5.3　ResetSystem() の定義 (efi.h より)

```
enum EFI_RESET_TYPE {
        /* 説明は仕様書 ("7.5.1 Reset System") の意訳です。 */
        EfiResetCold,
                /* システム全体のリセット。
                 * システム内の全ての回路を初期状態へリセットする。
                 * (追記: いわゆる再起動で、CPU 以外の回路も
                 * 電気的に遮断するコールドリセット)
                 * このリセットタイプはシステム操作に対し非同期で、
                 * システムの周期的動作に関係なく行われる。 */
        EfiResetWarm,
                /* システム全体の初期化。
                 * プロセッサは初期状態へリセットされ、
                 * 保留中のサイクルは破壊されない (?)。
                 * (追記: いわゆる再起動で、CPU のみリセットするウォームリセット)
                 * もしシステムがこのリセットタイプをサポートしていないならば、
                 * EfiResetCold が実施されなければならない。 */
        EfiResetShutdown,
                /* システムが ACPI G2/S5 あるいは G3 状態に相当する電源状態へ
                 * 遷移する (追記: いわゆるシャットダウンの状態)。
                 * もしシステムがこのリセットタイプをサポートしておらず、
                 * システムが再起動した時、EfiResetCold の振る舞いを示すべき。 */
        EfiResetPlatformSpecific
                /* システム全体のリセット。
                 * 厳密なリセットタイプは引数"ResetData"に従う EFI_GUID
                 * により定義される。
                 * プラットフォームが ResetData 内の EFI_GUID を認識できないならば、
                 * サポートできるリセットタイプを選択しなければならない。
                 * プラットフォームは発生した非正常なりセットから
                 * パラメータを記録することができる (?)。 */
};

struct EFI_SYSTEM_TABLE {
         ・・・
        struct EFI_RUNTIME_SERVICES {
                ・・・
                //
                // Miscellaneous Services
                //
                unsigned long long _buf_rs5;
                void (*ResetSystem)(
                        enum EFI_RESET_TYPE ResetType,
                                /* 実施されるリセットタイプ
                                 * この節では EfiResetShutdown を使用 */
                        unsigned long long ResetStatus,
                                /* リセットのステータスコードを指定
                                 * システムのリセットが
                                 * 正常なものならば EFI_SUCCESS、
                                 * 異常によるものならばエラーコードを指定
                                 * この節では EFI_SUCCESS(=0) を指定 */
                        unsigned long long DataSize,
                                /* ResetData のデータサイズをバイト単位で指定
                                 * この節では ResetData は使用しないので 0 を指定 */
                        void *ResetData
                                /* ResetType が EfiResetCold・EfiResetWarn
                                 * ・EfiResetShutdown の時、
                                 * ResetStatus が EFI_SUCCESS で無いならば、
                                 * ResetData へ NULL 終端された文字列を指定することで、
```

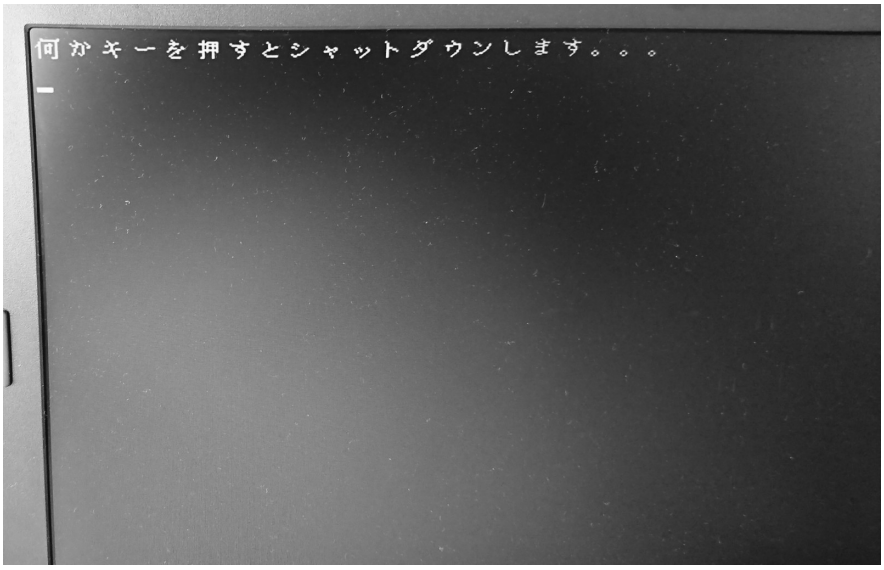
```
                                        * 後々、呼び出し元へリセット事由を通知できる
                                        *（ようである）
                                        * ResetType が EfiResetPlatformSpecific の時、
                                        * EFI_GUID を NULL 終端文字列として指定する事で、
                                        * プラットフォーム依存のリセットを行える
                                        * この節では ResetStatus は EFI_SUCCESS なので、
                                        * 使用しない（NULL を指定する）*/
                        );
        } *RuntimeServices;
};
```

使用例はリスト 5.4 の通りです。

▼**リスト** 5.4　ResetSystem() **の使用例** (main.c **より**)

```
 1: #include "efi.h"
 2: #include "common.h"
 3:
 4: void efi_main(void *ImageHandle __attribute__ ((unused)),
 5:           struct EFI_SYSTEM_TABLE *SystemTable)
 6: {
 7:     efi_init(SystemTable);
 8:     ST->ConOut->ClearScreen(ST->ConOut);
 9:
10:     /* キー入力待ち */
11:     puts(L"何かキーを押すとシャットダウンします。。。\r\n");
12:     getc();
13:
14:     /* シャットダウン */
15:     ST->RuntimeServices->ResetSystem(EfiResetShutdown, EFI_SUCCESS, 0,
16:                                      NULL);
17:
18:     while (TRUE);
19: }
```

リスト 5.4 は何かキーを入力するとシャットダウンします。
実行例は図 5.2 の通りです。



▲図 5.2　ResetSystem() の実行例

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！

前著 (パート 1) では「OS っぽいものを作る」という方針でしたが、本書に至り、やっと UEFI ファームウェアが提供するブートローダーとしての機能を実現することができました。

本書内で紹介した通り、今の Linux カーネルは UEFI を使うことで簡単にブートすることができます。そのため、自分でビルドした Linux カーネルをベースに小さな Linux システムを USB フラッシュメモリ上に構築していくのも面白いのではないかと思います。

また、コンソール出力やキー入力に関してもいくつかの TIPS を紹介しました。解像度変更や、特定のキー入力を割り込みで受ける等、少し地味ですが、良ければ何かに活用してみてください。

最後になりますが、本書に限らずその他の情報等をきっかけに PC をベアメタルでいじろうとする人が少しでも増えると良いなと思います[*1]。今の所、ケータイやスマホ等と違い、PC はハードやソフトをどうイジろうが、人に迷惑でも掛けない限り、怒られたりしないです。せっかくなので変なことをして遊び倒すのが面白いと思います。

[*1] 居る所には結構居たりしてびっくりしますが

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。
- ブートローダーは 4 行で実装される - 技術者見習いの独り言
  - http://orumin.blogspot.jp/2014/12/4.html
  - Linux ブートをやってみるきっかけであり、参考にさせていただいた情報です。
  - 本書はこちらの記事を「gnu-efi を使わずフルスクラッチでやってみる」内容です。

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

フルスクラッチで作る!UEFI ベアメタルプログラミング パート 2

2017 年 10 月 22 日　技術書典 3 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