フルスクラッチで作る!
x86_64 自作 OS
パート 2

ACPI で HPET 取得して
スケジューラを作る本

大神祐真　著

2018-08-10 版　　へにゃぺんて 発行

はじめに

本書をお手に取っていただきありがとうございます。

本書は 64 ビットの x86 CPU(x86_64) を対象にフルスクラッチで OS をステップバイステップで作ってみる本のパート 2 です。

本書でやること

前著では、ハードウェアを抽象化してアプリケーションへ提供する簡単な OS を作りました。

ただ、前著はシングルタスクであったため、本書ではマルチタスクを実現するために簡単なスケジューラを実装してみます。

スケジューラは短い時間 (本書ではミリ秒単位) で複数のアプリを切り替えることで、同時に動いているように見せます。そのため、スケジューラを作るには OS にタイマーの機能が必要です。

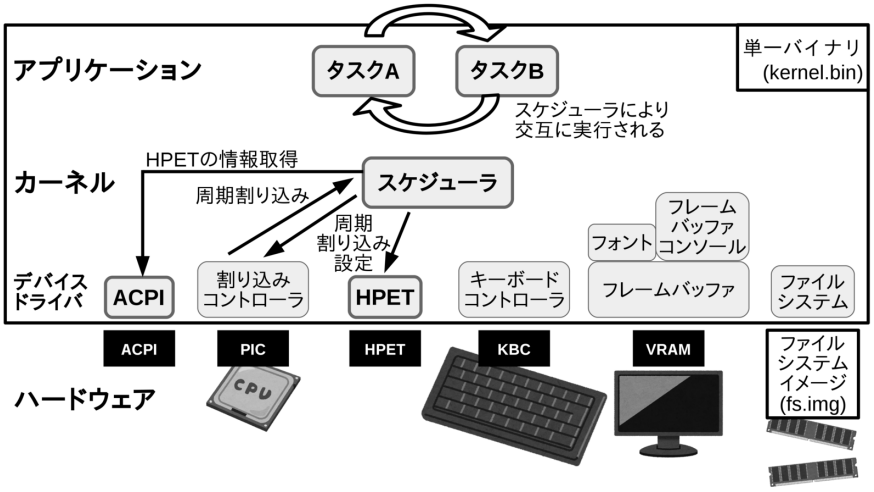
そのため、本書では、2005 年頃から PC に組み込まれるようになった比較的新しいハードウェアタイマーである HPET(High Precision Event Timer、高精度イベントタイマー) を扱う機能を OS へ追加します。

また、HPET の情報は ACPI(Advanced Configuration and Power Interface) という規格に従ってハードウェアから取得します。そのため、ACPI を扱う機能も追加します。

まとめると、本書では以下の流れでスケジューラを作ります。

1. ACPI から HPET の情報を取得
2. 取得した情報を使って HPET を制御する機能追加
3. 追加した HPET の機能を使ってスケジューラ実装

前著の「はじめに」で紹介したアーキテクチャ図に、本書で作る機能を追加すると図 .1 の通りです。



▲図 1　本書まででできあがる OS のアーキテクチャ図

太い枠線・太字の箇所が本書で追加する内容です。

開発環境・動作確認環境

本書の内容は、これまで通り、開発環境として Debian 上で開発し、QEMU と ThinkPad 上で動作確認したもので、詳しくは以下の通りです。

開発環境

- OS: Debian GNU/Linux 8.11 (コードネーム: Jessie)
- コンパイラ: GCC 4.9.2
  - build-essential パッケージでインストールされるものです

動作確認環境

- エミュレータ: QEMU 2.1
  - qemu-system-x86 パッケージでインストールされるものです
  - QEMU には UEFI のファームウェアが含まれていないため、別途"ovmf"パッケージもインストールします
- 実機: Lenovo ThinkPad E450

環境構築と動作確認方法

前著を参照してください。

本書含め、サークル「へにゃぺんて」の発行物はすべて PDF 版/HTML 版は無料で公開しています。

本書の PDF 版/HTML 版やサンプルコードの公開場所について

これまでの当サークルの同人誌同様、本書も PDF 版/HTML 版は以下のページで無料で公開しています。

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

また、本書で扱うサンプルコードとコンパイル済みバイナリは以下の GitHub で公開しています。

- ソースコード
  - `https://github.com/cupnes/x86_64_jisaku_os_2_samples`
- コンパイル済みバイナリ
  - `https://github.com/cupnes/x86_64_jisaku_os_2_samples/releases`

前著からの差分について

前著までで作ったブートローダーとカーネルに対して変更や追加を行う形で説明していきますが、ベースとなるソースコードは一部修正等で前著から少し進んでいます。

主な差分は以下の通りです。一部の差分は本文でも言及します。

■ ブートローダー

- Makefile: run ターゲット追加
  - make run で単体実行できます
- poiboot.c: リファクタリング
  - 変数や定数名の"apps"を"fs"へリネーム
  - コンフィグ・カーネル・ファイルシステムのロード処理をそれぞれの関数へ分割

■ カーネル

- common.h/common.c: strncmp 関数追加
  - 文字数指定の文字列比較
- fbcon.h/fbcon.c: puth 関数追加
  - 16 進数で数値をダンプ
- 各所の構造体定義箇所
  - ___attribute___((packed)) 追加
- main.c: 画像ビューア起動処理削除
  - カーネルはアプリを何も起動させない状態を本書ではベースとします

本書で始める基準となるソースコードは、サンプルコードのリポジトリ内に、ブートローダーは「A00_boot_base」というディレクトリで、カーネル・アプリは「A01_kern_app_base」というディレクトリで置いています。本書の説明通りに手を動かしてみる際はこちらを使ってみてください。

目次

はじめに **2**
本書でやること 2
開発環境・動作確認環境 3
本書の PDF 版/HTML 版やサンプルコードの公開場所について 4
前著からの差分について 4

第 1 章 ACPI **9**
1.1 共通: この章でやること 9
1.2 boot: EFI Configuration Table を見てみる 11
1.2.1 やること 11
1.2.2 実装 13
1.2.3 動作確認 15
1.3 boot: ACPI Table を見つける処理を実装する 16
1.3.1 実装 16
1.3.2 動作確認 18
1.4 boot: RSDP をカーネルへ渡すようにする 19
1.4.1 やること 19
1.4.2 実装 20
1.5 kernel: RSDP をブートローダーから受け取る 22
1.5.1 実装: RSDP を受け取り、シグネチャを表示 22
1.5.2 動作確認 24
1.5.3 実装: RSDP の定義と ACPI 初期化関数追加 24
1.6 kernel: XSDT を見てみる 27
1.6.1 やること 27
1.6.2 実装 28
1.6.3 動作確認 30

1.7 kernel: OVMF のバージョンを上げる 31
1.7.1 やること 31
1.7.2 新しい OVMF を入手 (deb ファイルをダウンロード) 32
1.7.3 新しい OVMF を入手 (APT で入手) 34
1.7.4 実装: Makefile を修正 34
1.7.5 動作確認 35
1.8 kernel: HPET テーブルを見つける処理を追加する 35
1.8.1 やること 36
1.8.2 実装 36
1.8.3 動作確認 38

第 2 章 HPET 39
2.1 HPET の情報を表示してみる 39
2.1.1 やること 39
2.1.2 実装: レジスタ郡の先頭アドレス取得 41
2.1.3 実装: GCIDR を見てみる 43
2.1.4 動作確認 46
2.1.5 実装: その他のレジスタも見てみる 47
2.1.6 動作確認 50
2.1.7 実装: HPET 初期化時に HPET を明示的に無効化 52
2.2 sleep を実装してみる 52
2.2.1 やること 52
2.2.2 実装 53
2.2.3 動作確認 55
2.3 alert を実装してみる 55
2.3.1 やること 56
2.3.2 実装 57
2.3.3 動作確認 65
2.4 周期タイマー (ptimer) を実装してみる 65
2.4.1 やること 66
2.4.2 実装 66
2.4.3 動作確認 70

第 3 章 スケジューラ 71
3.1 やること 71
3.2 実装: sched.c を用意 71

3.3 コンテキストスイッチの流れ 73
3.4 これから実装すること 74
3.5 実装: コンテキストスイッチ 74
3.6 実装: 実験用のタスクを定義 77
3.7 実装: タスク B のスタックの初期化 78
3.8 動作確認 79

おわりに **81**
表紙について 81

参考情報 **82**
参考にさせてもらった情報 82
本シリーズの過去の著作 82

第 1 章

ACPI

HPET を使用するに当たり、この章で、ACPI から HPET の情報の引き出し方を説明します。

なお、この章は前著[*1]より前の著作[*2]で扱ったブートローダーに関する説明も含みます。ブートローダーに関する項は「boot:」を、カーネルに関する項は「kernel:」を、共通の項には「共通:」を項のタイトルに付けていますので、「ブートローダーは道具として使うだけ」等の場合は適宜読み飛ばしてください。ブートローダーとカーネルはバイナリレベルで分かれているため、カーネルを作る上でブートローダーの中身まで知らなくても大丈夫です。

1.1 共通: この章でやること

HPET はアドレス空間上に配置された HPET のレジスタを使用して制御します。詳しくは後述しますが、タイマーの設定を行うレジスタや、タイマーのカウンタ値を取得したり設定したりするレジスタ等があり、これらのレジスタへ値をセットしたり、取得した値からタイマーの状態を確認しながら HPET を制御します。

そのため、HPET を制御するために HPET のレジスタの先頭アドレスを取得する必要があるのですが、それには段階があります。

まず、PC に搭載されている各種デバイスの情報は ACPI(Advanced Configuration and Power Interface) で管理されています。ACPI では、個々のデバイス等の情報はそれぞれを管理するテーブルがあり、HPET にも「HPET Table」というテーブルがあります。このテーブルに HPET レジスタの先頭アドレスも格納されています。

そして、HPET Table 含む個々のテーブルの先頭アドレス一覧を持つ「Extended

[*1] フルスクラッチで作る!x86_64 自作 OS (パート 1)

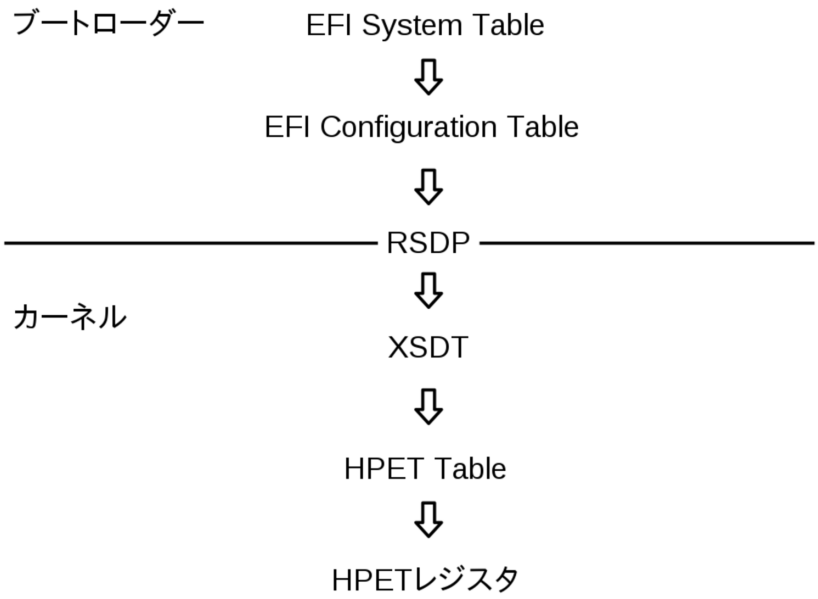
[*2] フルスクラッチで作る!UEFI ベアメタルプログラミング パート 1/パート 2

System Description Table(XSDT)」[*3]というテーブルがあり、HPET Table の先頭アドレスを知るには XSDT の先頭アドレスを知る必要があります。

XSDT の先頭アドレスは、「Root System Description Pointer(RSDP)」というテーブルに持っていて、RSDP の先頭アドレスは、UEFI 使用時は UEFI ファームウェアが「EFI Configuration Table」というテーブルに持っています。

そして EFI Configuration Table のアドレスは、UEFI ファームウェアがブートローダーを起動させるときに渡してくれる「EFI System Table」の中にあります。

EFI System Table から HPET のレジスタのアドレスを取得するまでの流れを図にまとめると図 1.1 の通りです。



▲図 1.1　HPET レジスタのアドレスを知るまでの流れ

本書では「UEFI の各種テーブルから RSDP を見つけてカーネルへ渡す」事をブートローダーの役割とし、「RSDP から HPET レジスタの先頭アドレスを見つけ、HPET を制御する」のをカーネルの役割とします。

[*3] XSDT は 64 ビット用のテーブルです。名前の「Extended」というのは、元々 32 ビットの頃から「Root System Description Table(RSDT)」というテーブルがあり、それを 64 ビットへ拡張した、という意味です。

次の項からは、それらの役割をブートローダーとカーネルへ実装していきます。

1.2 boot: EFI Configuration Table を見てみる

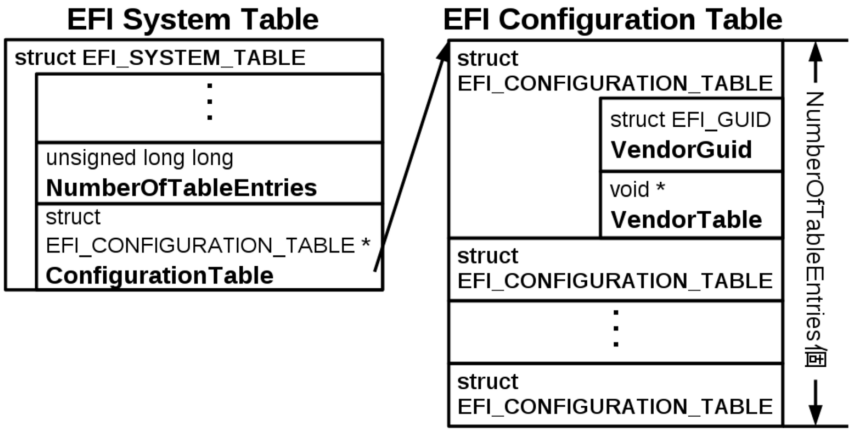
まずは EFI System Table から EFI Configuration Table を見つけ、EFI Configuration Table の内容を見てみるようにブートローダーへ実装してみます。

この項のサンプルディレクトリは「010_dump_config」です。

1.2.1 やること

EFI System Table は、PC 起動時に UEFI ファームウェアがブートローダーを呼び出す際、「struct EFI_SYSTEM_TABLE」という構造体のポインタとしてエントリ関数の引数に渡してくれます[*4]。

そして、EFI System Table と EFI Configuration Table の関係は図 1.2 の通りです。



▲図 1.2　EFI System Table と EFI Configuration Table の関係

[*4] 詳しくは拙著「フルスクラッチで作る!UEFI ベアメタルプログラミング」をご覧ください。

EFI Configuration Table の実体は EFI_CONFIGURATION_TABLE 構造体の配列です。EFI_CONFIGURATION_TABLE は、参照しているテーブルが何のテーブルであるのかを一意に決める「VendorGuid(struct EFI_GUID 型)」という GUID[*5]と、そのテーブルへのポインタ「VendorTable(void *型)」のペアです。

XSDT にも GUID が決められており、その値は UEFI 仕様に書かれています[*6]。なお、「XSDT」という呼び方は ACPI の世界での呼び方で、UEFI 仕様上は単に「ACPI Table」と呼んでいます[*7]。「EFI_ACPI_TABLE_GUID」という名前で値が書かれており、その内容はリスト 1.1 の通りです。

▼**リスト** 1.1　EFI_ACPI_TABLE_GUID

```
//
// ACPI 2.0 or newer tables should use EFI_ACPI_TABLE_GUID
//
#define EFI_ACPI_TABLE_GUID \
  {0x8868e871,0xe4f1,0x11d3,\
   {0xbc,0x22,0x00,0x80,0xc7,0x3c,0x88,0x81}}
```

リスト 1.1 の GUID を持つ EFI_CONFIGURATION_TABLE の VendorTable が、RSDP です。

なお、リスト 1.1 のコメントに「ACPI 2.0 かそれ以降では EFI_ACPI_TABLE_GUID を使うべきである」旨が書かれている通り、EFI_ACPI_TABLE_GUID は ACPI 2.0 以降向けの GUID です。本書では ACPI のバージョンは 2.0 以降を想定して説明します。

そして、EFI_SYSTEM_TABLE 構造体の最後のメンバである「ConfigurationTable」変数が EFI_CONFIGURATION_TABLE の配列の先頭を指していて、EFI_CONFIGURATION_TABLE が何個並んでいるかは NumberOfTableEntries に格納されています。

まとめると、以下を行えば RSDP を取得できます。

1. EFI_SYSTEM_TABLE の ConfigurationTable から EFI_CONFIGURATION_TABLE の配列の先頭アドレスを取得
2. EFI_CONFIGURATION_TABLE の配列の中から VendorGuid が EFI_ACPI_TABLE_GUID と一致するものを探す

[*5] Guaranteed Unique IDentifier。一意であることが保証された ID。

[*6] 4.6 EFI Configuration Table & Properties Table

[*7] XSDT さえ得られれば、そこから先は ACPI の世界の話なので、UEFI から見れば ACPI のテーブルは XSDT のみです。

1.2.2 実装

それでは、実装していきます。

まずは、efi.h の struct EFI_SYSTEM_TABLE 定義箇所へ NumberOfTableEntries と ConfigurationTable メンバを追加します (リスト 1.2)。

また、これから実装する EFI Configuration Table をダンプする関数「dump_efi_configuration_table」のプロトタイプ宣言も追加しておきます (リスト 1.2)。

▼**リスト** 1.2　010_dump_config/include/efi.h(310 **行目辺り**)

```
/* ・・・省略・・・ */

struct __attribute__((packed)) EFI_SYSTEM_TABLE {
        char _buf1[44];
        /* ・・・省略・・・ */
        } *BootServices;
        /* 追加（ここから） */
        unsigned long long NumberOfTableEntries;
        struct EFI_CONFIGURATION_TABLE {
                struct EFI_GUID VendorGuid;
                void *VendorTable;
        } *ConfigurationTable;
        /* 追加（ここまで） */
};

/* ・・・省略・・・ */

void efi_init(struct EFI_SYSTEM_TABLE *SystemTable);
void dump_efi_configuration_table(void);        /* 追加 */

#endif
```

次に、EFI Configuration Table をダンプする関数「dump_efi_configuration_table」を libuefi/efi.c へ追加します (リスト 1.3)。

▼**リスト** 1.3　010_dump_config/libuefi/efi.c(60 **行目辺り**)

```
/* ・・・省略・・・ */

void efi_init(struct EFI_SYSTEM_TABLE *SystemTable)
{
        /* ・・・省略・・・ */
        ST = SystemTable;
        /* ・・・省略・・・ */
}

/* 追加（ここから） */
void dump_efi_configuration_table(void)
```

```
{
        unsigned long long i;
        for (i = 0; i < ST->NumberOfTableEntries; i++) {
                puth(i, 1);
                putc(L':');
                puth((unsigned long long)&ST->ConfigurationTable[i], 16);
                putc(L':');
                puth(ST->ConfigurationTable[i].VendorGuid.Data1, 8);
                putc(L' ');
                puth(ST->ConfigurationTable[i].VendorGuid.Data2, 4);
                putc(L' ');
                puth(ST->ConfigurationTable[i].VendorGuid.Data3, 4);
                putc(L' ');
                unsigned char j;
                for (j = 0; j < 8; j++)
                        puth(ST->ConfigurationTable[i].VendorGuid.Data4[j], 2);
                putc(L':');
                puth((unsigned long long)ST->ConfigurationTable[i].VendorTable,
                     16);
                puts(L"\r\n");
        }
}
/* 追加（ここまで） */
```

poiboot は起動時に実行する efi_init 関数で EFI_SYSTEM_TABLE のポインタをグローバル変数 ST へ格納しているので、dump_efi_configuration_table 関数ではこれを使っています。

やっていることは単に、NumberOfTableEntries の数だけループを回して ConfigurationTable の VendorGuid と VendorTable(ポインタ) をダンプしているだけです。

最後に、poiboot.c へ dump_efi_configuration_table 関数を呼び出す処理を追加します (リスト 1.4)。

▼**リスト** 1.4　010_dump_config/poiboot.c(35 **行目辺り**)

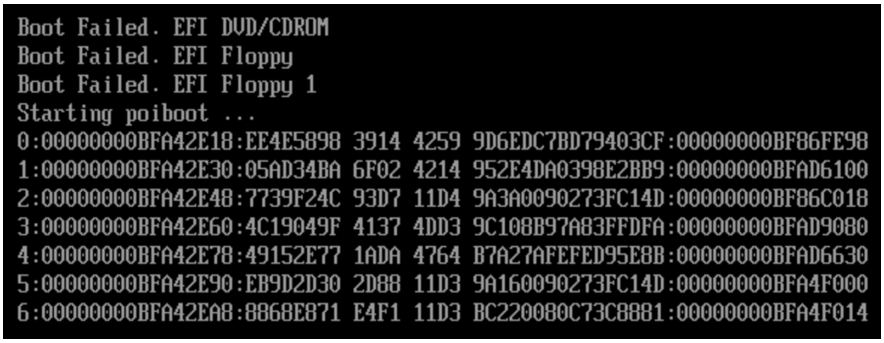
```
/* ・・・省略・・・ */

void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
{
        efi_init(SystemTable);

        puts(L"Starting poiboot ...\r\n");

        /* 追加（ここから） */
        /* ConfigurationTable の内容を表示して停止する */
        dump_efi_configuration_table();
        while (TRUE);
        /* 追加（ここまで） */

        /* ・・・省略・・・ */
}
```

```
/* ・・・省略・・・ */
```

1.2.3 動作確認

poiboot.conf のロードなどの直前に処理を追加しているので、ビルドして生成された poiboot.efi 単体で実行してみることができます。

実行してみると、EFI_ACPI_TABLE_GUID と一致するものがあることがわかります (図 1.3)。



▲図 1.3　010_dump_config の実行結果

struct 定義時の「__attribute__((packed))」について

リスト 1.2 で struct EFI_SYSTEM_TABLE へ struct EFI_CONFIGURATION_TABLE の追加を説明するために引用していた EFI_SYSTEM_TABLE の定義箇所には「__attribute__((packed))」が付いていました。
前著までは付けていなかったので、この attribute について簡単に補足しておくと、これは「構造体の各メンバーはメモリ空間上に隙間なく並べてくれ」というコンパイラへの指定です。
これが付いていないと、例えば char 型のメンバー変数 A と int 型のメンバー変数 B がこの順に構造体のメンバーとして指定されていた時、「B は 4 バイトアラインされた場所に配置しよう」として A と B の間に 3 バイトの隙間を開ける可能性があります。
「単に char 型と int 型の変数をセットで使えれば良い」という構造体ならば隙間は良しなに空けてもらっても良いのですが、EFI_SYSTEM_TABLE のような場合は「先頭から N バイト目には〜が並ぶ」という仕様に基づくものなので、メンバー間で隙間が空いてしまうと目的の場所へ正しくアクセスできなくなってしまいます。
これまで筆者の手元で確認する限りメンバー間に隙間が空けられる事は無かったのですが、本書のサンプルを書く中で新たに定義した構造体でそのような事があったため、過去の構造体定義にも「__attribute__((packed))」を付けるようにしています。

1.3 boot: ACPI Table を見つける処理を実装する

前項では、EFI Configuration Table をダンプして ACPI Table が存在することを確認しました。この項では ACPI Table を見つける処理を実装します。
この項のサンプルディレクトリは「011_find_acpi」です。

1.3.1 実装

新たに説明することは特に無いので、さっそく実装してみます。
まず、libuefi/efi.c へ ACPI Table を見つけてその先頭アドレスを返す

「find_efi_acpi_table」関数を追加します (リスト 1.5)。

▼**リスト** 1.5　011_find_acpi/libuefi/efi.c(80 **行目辺り**)

```
/* ・・・省略・・・ */

void dump_efi_configuration_table(void)
{
        /* ・・・省略・・・ */
}

/* 追加（ここから） */
void *find_efi_acpi_table(void)
{
        const struct EFI_GUID efi_acpi_table = {
                0x8868e871, 0xe4f1,0x11d3,
                {0xbc, 0x22, 0x00, 0x80, 0xc7, 0x3c, 0x88, 0x81}};

        unsigned long long i;
        for (i = 0; i < ST->NumberOfTableEntries; i++) {
                struct EFI_GUID *guid = &ST->ConfigurationTable[i].VendorGuid;
                if ((guid->Data1 == efi_acpi_table.Data1)
                    && (guid->Data2 == efi_acpi_table.Data2)
                    && (guid->Data3 == efi_acpi_table.Data3)) {
                        unsigned char is_equal = TRUE;
                        unsigned char j;
                        for (j = 0; j < 8; j++) {
                                if (guid->Data4[j] != efi_acpi_table.Data4[j])
                                        is_equal = FALSE;
                        }
                        if (is_equal == TRUE)
                                return ST->ConfigurationTable[i].VendorTable;
                }
        }
        return NULL;
}
/* 追加（ここまで） */
```

const 定義している efi_acpi_table の GUID に一致するものを for ループで一つずつチェックしながら見つけているだけです。

次に、この関数を poiboot.c から呼び出せるように include/efi.h へプロトタイプ宣言を追加します (リスト 1.6)。

▼**リスト** 1.6　011_find_acpi/include/efi.h(510 **行目辺り**)

```
/* ・・・省略・・・ */

void efi_init(struct EFI_SYSTEM_TABLE *SystemTable);
void dump_efi_configuration_table(void);
void *find_efi_acpi_table(void);        /* 追加 */

#endif
```

最後に find_efi_acpi_table 関数を poiboot.c から呼び出してみます (リスト 1.7)。

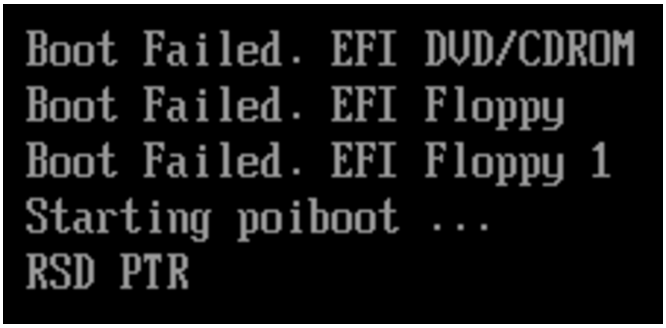
▼**リスト** 1.7　011_find_acpi/poiboot.c(40 **行目辺り**)

```
/* ・・・省略・・・ */

void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
{
        efi_init(SystemTable);

        puts(L"Starting poiboot ...\r\n");

        /* 変更（ここから） */
        /* RSDP のシグネチャを表示して停止する */
        char *s = find_efi_acpi_table();
        putc(*s++);     /* 'R' */
        putc(*s++);     /* 'S' */
        putc(*s++);     /* 'D' */
        putc(*s++);     /* ' ' */
        putc(*s++);     /* 'P' */
        putc(*s++);     /* 'T' */
        putc(*s++);     /* 'R' */
        putc(*s);       /* ' ' */
        while (TRUE);
        /* 変更（ここまで） */

        /* ・・・省略・・・ */
```

ACPI Table の構成について詳しくは後述しますが、ACPI Table の先頭には ASCII のシグネチャが並んでいます。RSDP には 8 バイト (8 文字) のシグネチャがあり、find_efi_acpi_table 関数でとってきた ACPI Table(RSDP) 先頭アドレスを利用してシグネチャを表示してみています。コメントにも書いてある通りですが、XSDT のシグネチャは"RSD PTR "の 8 文字 (最後にスペース有) なので、それが表示されるはずです。

1.3.2　動作確認

試しに実行してみると、確かに"RSD PTR "というシグネチャが表示されることを確認できます (図 1.4)。



▲図 1.4　011_find_acpi の実行結果

1.4　boot: RSDP をカーネルへ渡すようにする

ブートローダー側の変更の最後として、見つけた ACPI Table 先頭アドレス (RSDP) をカーネルへ渡すように変更します。

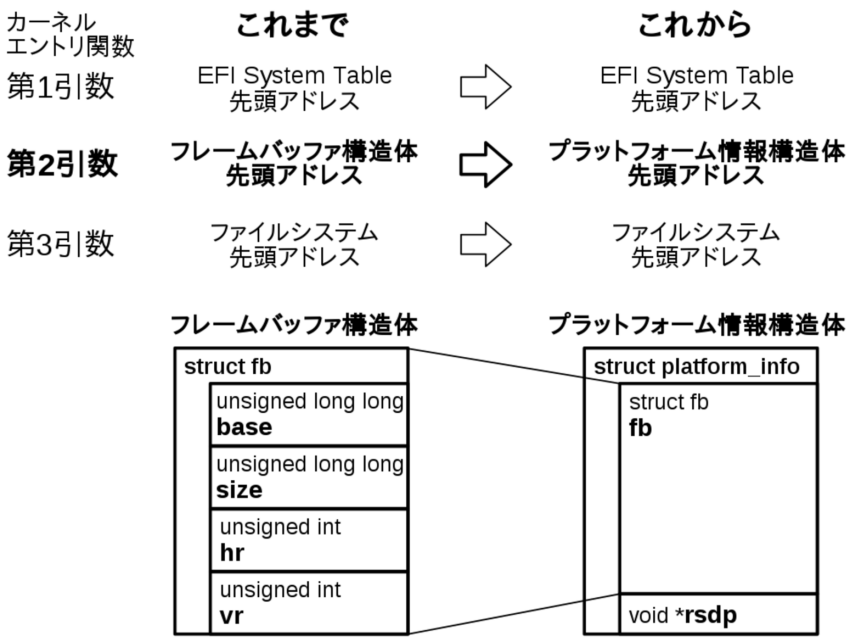
この項のサンプルディレクトリは「012_pass_rsdp」です。

1.4.1　やること

これまで EFI System Table の先頭アドレスや、フレームバッファ情報の先頭アドレス等を渡してきたように、RSDP もカーネル側のエントリ関数の引数として渡します。

ただし、カーネルへ渡したい情報は今後も増えると思われ、その都度引数を増やすのも面倒です。そのため、フレームバッファと RSDP を「プラットフォーム情報 (struct platform_info)」という一つの構造体にまとめ、その先頭アドレスをカーネルへ渡すことにします。

この項で行う変更を図示すると図 1.5 の通りです。



▲図 1.5　struct platform_info を追加し、カーネル側エントリ関数の第 2 引数で渡す

1.4.2　実装

変更を行うのは poiboot.c のみです。

まず、struct platform_info を定義し、グローバル変数 pi を作成します (リスト 1.8)。

▼リスト 1.8　012_pass_rsdp/poiboot.c(15 行目辺り)

```
/* ・・・省略・・・ */

#define MB             1048576 /* 1024 * 1024 */

/* 追加（ここから） */
struct __attribute__((packed)) platform_info {
        struct fb fb;
        void *rsdp;
} pi;
/* 追加（ここまで） */

void load_config(
        struct EFI_FILE_PROTOCOL *root, unsigned short *conf_file_name,
```

```
        unsigned long long *kernel_start, unsigned long long *stack_base,
        unsigned long long *fs_start);
/* ・・・省略・・・ */
```

また、前項で追加した処理は削除しておきます (リスト 1.9)。

▼**リスト** 1.9　012_pass_rsdp/poiboot.c(30 **行目辺り**)

```
/* ・・・省略・・・ */

void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
{
        efi_init(SystemTable);

        puts(L"Starting poiboot ...\r\n");

        /* 削除（ここから） */
        /* RSDP のシグネチャを表示して停止する */
        char *s = find_efi_acpi_table();
        putc(*s++);     /* 'R' */
        putc(*s++);     /* 'S' */
        putc(*s++);     /* 'D' */
        putc(*s++);     /* ' ' */
        putc(*s++);     /* 'P' */
        putc(*s++);     /* 'T' */
        putc(*s++);     /* 'R' */
        putc(*s);       /* ' ' */
        while (TRUE);
        /* 削除（ここまで） */

        /* ・・・省略・・・ */
```

そして、pi へパラメータを設定します。そして、カーネルの第 2 引数へ fb に代わり pi のポインタを指定するようにします (リスト 1.10)。

▼**リスト** 1.10　012_pass_rsdp/poiboot.c(60 **行目辺り**)

```
/* ・・・省略・・・ */

void efi_main(void *ImageHandle, struct EFI_SYSTEM_TABLE *SystemTable)
{
        /* ・・・省略・・・ */

        /* カーネルへ引数として渡す内容を変数に準備する */
        unsigned long long kernel_arg1 = (unsigned long long)ST;
        put_param(L"kernel_arg1", kernel_arg1);
        init_fb();
        /* unsigned long long kernel_arg2 = (unsigned long long)&fb;
         * ↑変更前 */
        /* 変更後（ここから） */
        pi.fb.base = fb.base;
```

```
        pi.fb.size = fb.size;
        pi.fb.hr = fb.hr;
        pi.fb.vr = fb.vr;
        pi.rsdp = find_efi_acpi_table();
        unsigned long long kernel_arg2 = (unsigned long long)&pi;
        /* 変更後（ここまで） */
        put_param(L"kernel_arg2", kernel_arg2);
        unsigned long long kernel_arg3;
        if (has_fs == TRUE)
                kernel_arg3 = fs_start;
        else
                kernel_arg3 = 0;
        put_param(L"kernel_arg3", kernel_arg3);

        /* ・・・省略・・・ */
```

動作確認のためにはカーネル側で RSDP を受け取る処理が必要です。そのため、動作確認は次の項で行います。

1.5 kernel: RSDP をブートローダーから受け取る

ブートローダーから RSDP を受けとるようにカーネル側を変更します。

この項のサンプルディレクトリは「013_dump_rsdp」です。

1.5.1 実装: RSDP を受け取り、シグネチャを表示

さっそく実装します。

まず、platform_info の定義を追加します (リスト 1.11)。

▼**リスト** 1.11　013_dump_rsdp/main.c

```
/* ・・・省略・・・ */
#include <common.h>

/* 追加（ここから） */
struct __attribute__((packed)) platform_info {
        struct framebuffer fb;
        void *rsdp;
};
/* 追加（ここまで） */

/* ・・・省略・・・ */
```

次に、この定義を使用してブートローダーから platform_info を受け取るようにカーネル側のエントリ関数 (start_kernel 関数) を変更します (リスト 1.12)。

▼**リスト** 1.12　013_dump_rsdp/main.c

```
void start_kernel(void *_t __attribute__ ((unused)), struct platform_info *pi,
                  void *_fs_start)                              /* ↑変更 */
{
        /* フレームバッファ周りの初期化 */
        fb_init(&pi->fb);       /* 変更 */
        set_fg(255, 255, 255);
        set_bg(0, 70, 250);
        clear_screen();

        /* ・・・省略・・・ */
```

start_kernel 関数の第 2 引数を platform_info 構造体へ変更し、fb_init 関数へ渡す framebuffer 構造体のポインタも platform_info 内のものを参照するように変更しています。

そして、platform_info の中の rsdp を使って、ブートローダー側で試した時と同様に RSDP のシグネチャを表示させてみます (リスト 1.13)。

▼**リスト** 1.13　013_dump_rsdp/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__ ((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* フレームバッファ周りの初期化 */
        fb_init(&pi->fb);
        set_fg(255, 255, 255);
        set_bg(0, 70, 250);
        clear_screen();

        /* 追加（ここから） */
        /* RSDP のシグネチャを表示 */
        char *s = (char *)pi->rsdp;
        putc(*s++);     /* 'R' */
        putc(*s++);     /* 'S' */
        putc(*s++);     /* 'D' */
        putc(*s++);     /* ' ' */
        putc(*s++);     /* 'P' */
        putc(*s++);     /* 'T' */
        putc(*s++);     /* 'R' */
        putc(*s);       /* ' ' */
        while (1);
        /* 追加（ここまで） */

        /* ・・・省略・・・ */
```

今の所、RSDP 表示の後の処理は行わないように、while の無限ループで止めています。

1.5.2　動作確認

実行すると図 1.6 のように RSDP のシグネチャが表示されます。

▲図 1.6　013_dump_rsdp の実行結果

バイナリ間の後方互換性

今回、「ブートローダーがカーネルへ渡す引数の内容を変える」という「ブートローダーとカーネルの間の仕様変更」を行いました。
ですが、実は、カーネル側がこの仕様変更を行う前のバージョンであっても (ブートローダー側だけ先にバージョンアップしてしまっても) 動作できるように変更を行っています。
platform_info 構造体の 1 番目のメンバーに framebuffer 構造体を置いているのがその点です。構造体自身の先頭アドレスとその 1 つ目のメンバーの先頭アドレスは同じなので、カーネルが platform_info 構造体を知らなくても、framebuffer 構造体だと思ってアクセスしてくれれば問題無いようになっています。

1.5.3　実装: RSDP の定義と ACPI 初期化関数追加

この項の最後に acpi.c というソースファイルを追加し、RSDP の定義とカーネル起動時に呼び出す ACPI 初期化関数を追加します。

まず、RSDP の定義を追加します (リスト 1.14)。

▼**リスト** 1.14　013_dump_rsdp/acpi.c

```
/* 追加（ここから）*/
struct __attribute__((packed)) RSDP {
        char Signature[8];
        unsigned char Checksum;
        char OEMID[6];
        unsigned char Revision;
        unsigned int RsdtAddress;
        unsigned int Length;
        unsigned long long XsdtAddress;
        unsigned char Extended_Checksum;
        unsigned char Reserved[3];
};
/* 追加（ここまで）*/
```

RSDP 構造体のメンバーで注目すべきは XsdtAddress です。その名の通り、XSDT を指すアドレスです。

その他のメンバーは使わないので気にしなくて良いです。

なお、acpi.c の外で RSDP の構造を気にする必要は無いため、acpi.c の中で RSDP を定義しています。

加えて、以降の処理のために RSDP から XSDT のアドレスを取得してグローバル変数へ格納するよう ACPI の初期化関数 (acpi_init) を acpi.c へ追加します (リスト 1.15)。

▼**リスト** 1.15　013_dump_rsdp/acpi.c

```
struct __attribute__((packed)) RSDP {
        /* ・・・省略・・・ */
};

/* 追加（ここから）*/
unsigned long long xsdt;

void acpi_init(void *rsdp)
{
        xsdt = ((struct RSDP *)rsdp)->XsdtAddress;
}
/* 追加（ここまで）*/
```

start_kernel 関数から呼び出せるように acpi.h を作成してプロトタイプ宣言を追加します (リスト 1.16)。

▼**リスト** 1.16　013_dump_rsdp/include/acpi.h

```
/* 追加（ここから） */
#ifndef _ACPI_H_
#define _ACPI_H_

void acpi_init(void *rsdp);

#endif
/* 追加（ここまで） */
```

そして、start_kernel 関数へ acpi_init 関数呼び出し処理を追加します (リスト 1.17)。

▼**リスト** 1.17　013_dump_rsdp/main.c

```
#include <x86.h>
#include <intr.h>
#include <pic.h>
#include <acpi.h>       /* 追加 */
#include <fb.h>
/* ・・・省略・・・ */

void start_kernel(void *_t __attribute__ ((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* フレームバッファ周りの初期化 */
        fb_init(&pi->fb);
        set_fg(255, 255, 255);
        set_bg(0, 70, 250);
        clear_screen();

        /* ACPI の初期化 */  /* 追加 */
        acpi_init(pi->rsdp);     /* 追加 */

        /* RSDP のシグネチャを表示 */
        /* ・・・省略・・・ */
```

最後に、Makefile へ acpi.o をビルドターゲットに追加します (リスト 1.18)。

▼**リスト** 1.18　013_dump_rsdp/Makefile

```
TARGET = kernel.bin
CFLAGS = -Wall -Wextra -nostdinc -nostdlib -fno-builtin -fno-common -Iinclude
LDFLAGS = -Map kernel.map -s -x -T kernel.ld
OBJS = main.o iv.o fbcon.o fb.o font.o kbc.o x86.o intr.o pic.o \
        acpi.o handler.o fs.o common.o  # acpi.o を追加

# ・・・ 省略 ・・・
```

これで、この項で実装する内容は終わりです。

1.6 kernel: XSDT を見てみる

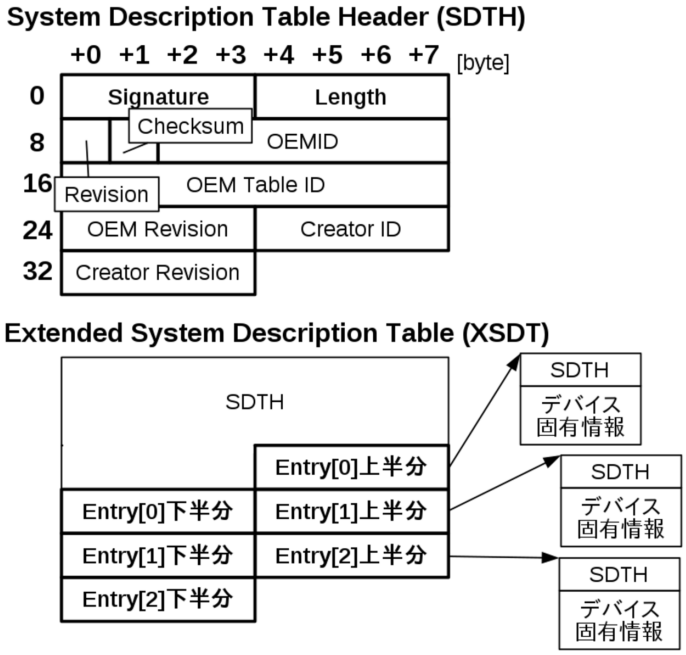
RSDP から XSDT の先頭アドレスがわかりました。なので、次は XSDT の中身を表示してみます。

この項のサンプルディレクトリは「014_dump_xsdt」です。

1.6.1 やること

この項では XSDT から参照できるすべてのテーブルのシグネチャを表示させてみます。

まず、XSDT の構造は図 1.7 の通りです。



▲図 1.7　XSDT の構造

System Description Table Header(SDTH) は共通のヘッダ構造です。XSDT から参照できる各デバイス等の固有のテーブルにも ACPI で管理するためのヘッダとして SDTH が付いています。

SDTH に続いて各デバイス等の固有のテーブルの先頭アドレスが並びます。各個のテーブルにもヘッダとして SDTH が付いていますので、先頭のシグネチャで何のテーブ

ルであるかを識別できます。

また、XSDT がいくつのテーブルを参照しているかは自身の SDTH の Length から知ることができます。Length には SDTH を含めたテーブルのサイズ (バイト) が入っているので、SDTH のサイズを引いてポインタサイズで割れば XSDT が参照するテーブル数になります。

1.6.2 実装

まず SDTH と XSDT の定義を追加します。SDTH は今後 acpi.c の外でも使うので acpi.h で (リスト 1.19)、対して XSDT は acpi.c の中でしか使わないので acpi.c で定義します (リスト 1.20)。

▼**リスト** 1.19　014_dump_xsdt/include/acpi.h

```
#ifndef _ACPI_H_
#define _ACPI_H_

/* 追加（ここから） */
struct __attribute__((packed)) SDTH {
        char Signature[4];
        unsigned int Length;
        unsigned char Revision;
        unsigned char Checksum;
        char OEMID[6];
        char OEM_Table_ID[8];
        unsigned int OEM_Revision;
        unsigned int Creator_ID;
        unsigned int Creator_Revision;
};
/* 追加（ここまで） */

void acpi_init(void *rsdp);

#endif
```

▼**リスト** 1.20　014_dump_xsdt/acpi.c

```
#include <acpi.h>       /* 追加 */

struct __attribute__((packed)) RSDP {
        /* ・・・省略・・・ */
};

/* 追加（ここから） */
struct __attribute__((packed)) XSDT {
        struct SDTH Header;
        struct SDTH *Entry[0];
};
```

```
/* 追加（ここまで） */

struct XSDT *xsdt;      /* 変更 */

void acpi_init(void *rsdp)
{
        xsdt = (struct XSDT *)((struct RSDP *)rsdp)->XsdtAddress;    /* 変更 */
}
```

併せてグローバル変数 xsdt の定義も XSDT 構造体のポインタへ変更しておきました。

次に、XSDT が参照するテーブルのシグネチャを表示する dump_xsdt 関数を実装します (リスト 1.21)。

▼**リスト** 1.21　014_dump_xsdt/acpi.c

```
#include <acpi.h>
#include <fbcon.h>      /* 追加 */

/* ・・・ 省略 ・・・ */

void acpi_init(void *rsdp)
{
        xsdt = (struct XSDT *)((struct RSDP *)rsdp)->XsdtAddress;
}

/* 追加（ここから） */
void dump_sdth_sig(struct SDTH *h)
{
        unsigned char i;
        for (i = 0; i < 4; i++)
                putc(h->Signature[i]);
}

void dump_xsdt(void)
{
        dump_sdth_sig(&xsdt->Header);
        puts("\r\n");

        unsigned long long num_sdts =
                (xsdt->Header.Length - sizeof(struct SDTH))
                / sizeof(struct SDTH *);
        puts("NUM SDTS ");
        putd(num_sdts, 2);
        puts("\r\n");

        unsigned long long i;
        for (i = 0; i < num_sdts; i++) {
                dump_sdth_sig(xsdt->Entry[i]);
                putc(' ');
        }
        puts("\r\n");
}
/* 追加（ここまで） */
```

シグネチャの表示処理は何度か使いそうなので dump_sdth_sig という関数へ分けました。

これらの関数のプロトタイプ宣言を acpi.h へ追加し (リスト 1.22)、main.c の start_kernel 関数から呼び出すように変更します (リスト 1.23)。

▼**リスト** 1.22　014_dump_xsdt/include/acpi.h

```
/* ・・・ 省略 ・・・ */

void acpi_init(void *rsdp);
void dump_sdth_sig(struct SDTH *h);     /* 追加 */
void dump_xsdt(void);                   /* 追加 */

#endif
```

▼**リスト** 1.23　014_dump_xsdt/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */

        /* ACPI の初期化 */
        acpi_init(pi->rsdp);

        /* 変更（ここから） */
        /* XSDT をダンプ */
        dump_xsdt();
        while (1);
        /* 変更（ここまで） */

        /* ・・・ 省略 ・・・ */
```

1.6.3　動作確認

例えば QEMU で実行すると図 1.8 のように XSDT が参照するテーブルの個数 (NUM SDTS) と各テーブルのシグネチャが表示されます。

▲図 1.8　014_dump_xsdt の QEMU 実行結果

HPET のテーブルのシグネチャは"HPET"なのですが、、無いです。
実機で実行してみると図 1.9 のように HPET のシグネチャも表示されます。

▲図 1.9　014_dump_xsdt の実機実行結果

一旦この項はここまでにして、次項でこれを解決します。

1.7　kernel: OVMF のバージョンを上げる

古いバージョンの OVMF[*8]の ACPI テーブルからは HPET が参照できない様で、OVMF のバージョンを上げると HPET の項目が表示されるようになります。そのため、この項で OVMF のバージョンを上げます。
なお、この項のサンプルディレクトリは前項に引き続き「014_dump_xsdt」です。

1.7.1　やること

やることは単に OVMF の新しいバイナリをダウンロードしてきて QEMU コマンドの引数に与えるように Makefile を修正するだけです。
これまで、OVMF のバイナリは"OVMF.fd"という単一のバイナリを使用していまし

[*8] UEFI のオープンソースのファームウェア実装。QEMU で UEFI を使用する際に使っています。

た。これは UEFI の実行コードのバイナリと、起動デバイスの優先順位などのパラメータを保存する先のバイナリとが 1 体となったものです。

最新の OVMF ではこれが 2 つのバイナリに分かれており、今後はこの 2 つのバイナリの形式のものを qemu コマンドへ指定して使用するようにします。

1.7.2 新しい OVMF を入手 (deb ファイルをダウンロード)

まずは新しい OVMF バイナリを入手します。

ここでは、新しい OVMF バイナリを含む Debian パッケージ (deb ファイル) をダウンロードし、展開します。

Debian 9(Stretch)、あるいは sid を使っている場合、それぞれのバージョンの ovmf パッケージに新しい OVMF が含まれています。その場合、APT で ovmf パッケージのインストールを行えば良いです。手順は次節の「新しい OVMF を入手 (APT で入手)」を参照してください。

それでは、手動で Debian パッケージをダウンロードし、展開します。ここでは Stretch の ovmf パッケージのパッケージファイル (deb 形式) を手動でダウンロードしてみます。

Debian パッケージは以下のウェブページから検索できます。

- Debian -- パッケージ
  - `https://packages.debian.org/`

ページを下へスクロールして「パッケージディレクトリを検索」のフォームにキーワードを入力し [検索] ボタンをクリックすると関連するパッケージを検索できます。

各パッケージのバージョンごとのページの URL は固定化されています。ovmf パッケージの Stretch バージョンのページは以下の通りです。

- Debian -- stretch の ovmf パッケージに関する詳細
  - `https://packages.debian.org/stretch/ovmf`

「ovmf のダウンロード」の項目の「アーキテクチャ」の箇所の「all」というリンクをクリックすると「ovmf_0~20161202.7bbe0b3e-1_all.deb のダウンロードページ」へ移動します。

実はダウンロードページも URL は固定なので、以下の URL です。

- Debian -- パッケージのダウンロードに関する選択 -- ovmf_0~20161202.7bbe0b3e-1_all.deb
  - `https://packages.debian.org/stretch/all/ovmf/download`

このページから Debian パッケージファイル (deb 形式) をダウンロードできます。「ftp.jp.debian.org/debian」をクリックすると日本のミラーサイトからダウンロードできます。

deb パッケージファイル名はバージョンアップにより変わるため、deb パッケージのリンクまでは固定化できませんが、2018 年 6 月 26 日現在は以下の URL です。

- `http://ftp.jp.debian.org/debian/pool/main/e/edk2/ovmf_0~20161202.7bbe0b3e-1_all.deb`

deb ファイルのダウンロードできたら、それを展開します。

deb パッケージファイルは dpkg コマンドで展開できます。以下のコマンドで「ovmf_0~20161202.7bbe0b3e-1_all.deb」という deb パッケージファイルを「ovmf」ディレクトリへ展開します (コマンド実行時に展開先のディレクトリが無ければ作ってくれます)。

```
$ dpkg -x ovmf_0~20161202.7bbe0b3e-1_all.deb ovmf
```

Windows の場合、dpkg コマンドを使った展開作業は、Microsoft Store で Debian GNU/Linux をインストールして使うと良いかと思います。

そして、ovmf ディレクトリ内を見ると、OVMF のファームウェアファイルがあることを確認できます。

```
$ cd ovmf
$ find
.
./usr
./usr/share
./usr/share/OVMF
./usr/share/OVMF/OVMF_CODE.fd
./usr/share/OVMF/OVMF_VARS.fd
./usr/share/doc
./usr/share/doc/ovmf
./usr/share/doc/ovmf/changelog.Debian.gz
./usr/share/doc/ovmf/copyright
./usr/share/ovmf
./usr/share/ovmf/OVMF.fd
./usr/share/qemu
./usr/share/qemu/OVMF.fd
```

1.7.3 新しい OVMF を入手 (APT で入手)

2018 年 6 月現在最新の安定板である Debian 9(コードネーム:Stretch)、あるいは開発版 (コードネーム:sid) を使っている場合は、以下のコマンドで ovmf パッケージをインストールすれば 2 つのバイナリの形式になった OVMF ファームウェアが入手できます[*9]。

```
$ sudo apt install ovmf
```

dpkg -L コマンドでパッケージでインストールされたファイルの一覧を確認すると以下の様に表示されます。

```
$ dpkg -L ovmf
/.
./usr
./usr/share
./usr/share/OVMF
./usr/share/OVMF/OVMF_CODE.fd
./usr/share/OVMF/OVMF_VARS.fd
./usr/share/doc
./usr/share/doc/ovmf
./usr/share/doc/ovmf/changelog.Debian.gz
./usr/share/doc/ovmf/copyright
./usr/share/ovmf
./usr/share/ovmf/OVMF.fd
./usr/share/qemu
./usr/share/qemu/OVMF.fd
```

"OVMF_CODE.fd"と"OVMF_VARS.fd"が、2 つのバイナリの形式になった OVMF ファームウェアです。

1.7.4 実装: Makefile を修正

最後に、Makefile の QEMU コマンドを修正します (リスト 1.24)。

▼**リスト** 1.24　014_dump_xsdt/Makefile

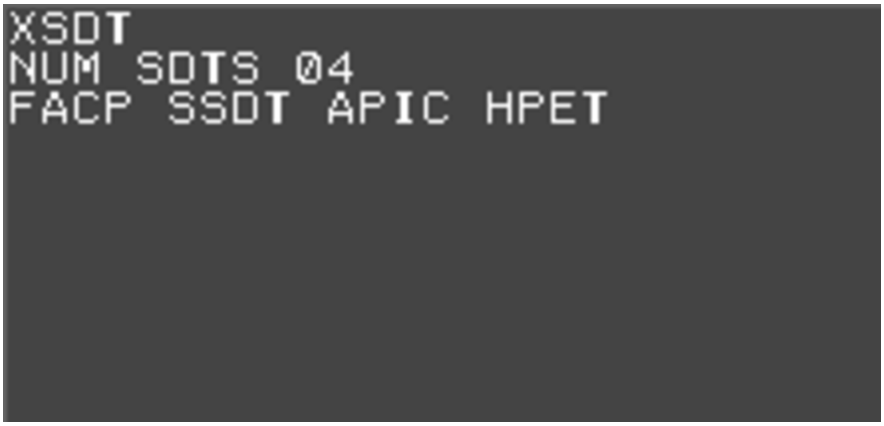
[*9] sid は日々新しいバージョンへ変わっているので、本書を読んだタイミングによってはパッケージの内容も変わっているかもしれません。

```
run: $(TARGET)
        cp $(TARGET) ../fs/
        # 変更（ここから）
        qemu-system-x86_64 -m 4G \
        -drive if=pflash,format=raw,readonly,file=$$HOME/ovmf/OVMF_CODE.fd \
        -drive if=pflash,format=raw,file=$$HOME/ovmf/OVMF_VARS.fd \
        -hda fat:../fs
        # 変更（ここまで）
```

一つの"-bios"オプションだった箇所を、2 つの"-drive"オプションへ変更しています。片方に"OVMF_CODE.fd"を指定し、もう片方に"OVMF_VARS.fd"を指定しているだけです。なお、本書では OVMF_CODE.fd と OVMF_VARS.fd をホームディレクトリ直下の ovmf ディレクトリに配置している想定で Makefile を書いていますので、異なる場合は適宜修正してください。

1.7.5 動作確認

新バージョンの OVMF を導入し、Makefile を書き換えた上で、改めて実行すると、図 1.10 の様に HPET のシグネチャが表示されることが確認できます。



▲図 1.10　014_dump_xsdt の実行結果 (新バージョンの OVMF 使用時)

1.8 kernel: HPET テーブルを見つける処理を追加する

XSDT がどのようになっているのかは確認できたので、ACPI 編の最後として HPET テーブルを XSDT の中から見つける処理を追加します。

この項のサンプルディレクトリは「015_find_hpet」です。

1.8.1 やること

この項では指定されたシグネチャのテーブルを XSDT から探して先頭アドレスを返す処理を「get_sdt」という関数で実装します。

1.8.2 実装

get_sdt 関数を実装するにあたり、現在 dump_xsdt 関数内で行っている「XSDT から参照できるテーブルの数 (num_sdts) を計算する処理」は get_sdt 関数でも必要になるので、acpi_init 関数で初期化時に行ってしまうようにします (リスト 1.25)。

▼**リスト** 1.25　015_find_hpet/acpi.c

```
/* ・・・省略 ・・・ */

struct XSDT *xsdt;
unsigned long long num_sdts;    /* 追加 */

void acpi_init(void *rsdp)
{
        xsdt = (struct XSDT *)((struct RSDP *)rsdp)->XsdtAddress;

        num_sdts = (xsdt->Header.Length - sizeof(struct SDTH)) /* 追加 */
                / sizeof(struct SDTH *);                        /* 追加 */
}

/* ・・・省略 ・・・ */

void dump_xsdt(void)
{
        dump_sdth_sig(&xsdt->Header);
        puts("\r\n");

        /* 削除:num_sdts 計算処理 */
        puts("NUM SDTS ");
        putd(num_sdts, 2);
        puts("\r\n");

        /* ・・・省略 ・・・ */
```

そして、get_sdt 関数はリスト 1.26 の様に実装できます。

▼**リスト** 1.26　015_find_hpet/acpi.c

```
/* ・・・ 省略 ・・・ */
#include <fbcon.h>
#include <common.h>     /* 追加 */

/* ・・・ 省略 ・・・ */

void dump_xsdt(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
struct SDTH *get_sdt(char *sig)
{
        unsigned long long i;
        for (i = 0; i < num_sdts; i++) {
                if (!strncmp(sig, xsdt->Entry[i]->Signature, 4))
                        return xsdt->Entry[i];
        }

        return NULL;
}
/* 追加（ここまで） */
```

文字数指定で文字列比較を行う strncmp 関数は本書で新たに common.c へ追加していますが、特に説明するほどでもないためここでは説明しません。実装が気になる場合はサンプルディレクトリ内を見てみてください (何の変哲もない実装かと思います)。

get_sdt 関数のプロトタイプ宣言を acpi.h へ追加し (コード省略)、main.c で get_sdt 関数を呼び出してみます (リスト 1.27)。

▼**リスト** 1.27　015_find_hpet/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */

        /* ACPI の初期化 */
        acpi_init(pi->rsdp);

        /* 変更（ここから） */
        /* HPET の情報を取得 */
        struct SDTH *hpet_table = get_sdt("HPET");
        dump_sdth_sig(hpet_table);
        while (1);
        /* 変更（ここまで） */

        /* ・・・ 省略 ・・・ */
```

1.8.3　動作確認

実行すると図 1.11 のように HPET のシグネチャ ("HPET") が表示されます。

▲図 1.11　015_find_hpet の実行結果

ここまでで、ACPI に関する部分は終わりです。

第 2 章

HPET

前章で取得した HPET テーブルを使って、HPET レジスタのアドレスを取得し、レジスタ経由で HPET を制御します。

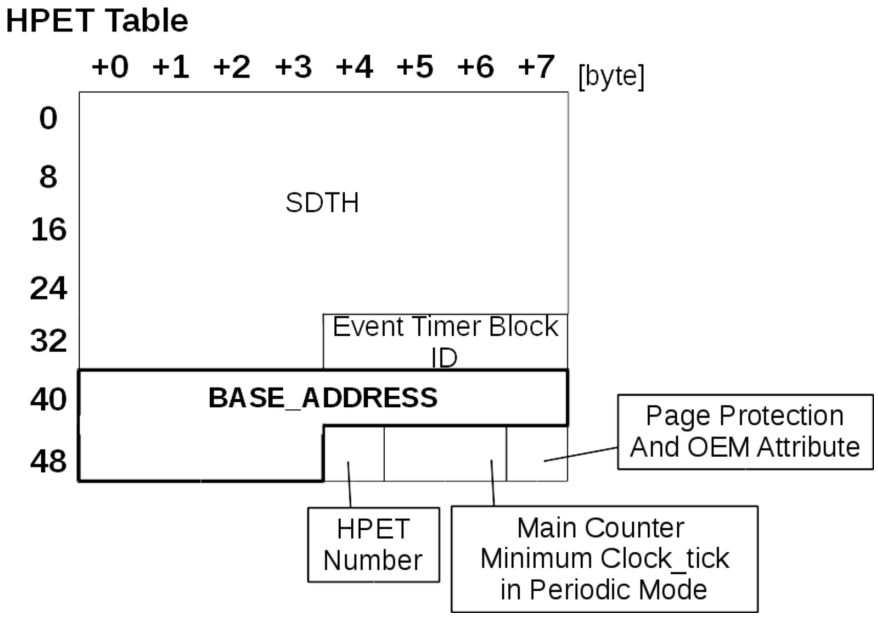
2.1　HPET の情報を表示してみる

HPET のレジスタ郡の先頭アドレスを取得し、試しに HPET のレジスタをいくつか見てみます。

この項のサンプルディレクトリは「020_dump_regs」です。

2.1.1　やること

まず、ACPI から取得した HPET のテーブルは、図 2.1 の構造になっています。



▲図 2.1　HPET テーブルの構造

SDTH 含め先頭から 40 バイト目にレジスタ郡の先頭アドレスを指す 12 バイトのエントリーがあります。HPET のデータシート曰く「12 バイトの ACPI アドレスフォーマット」とのことで、12 バイトの中の最後の 8 バイトにレジスタ郡の先頭アドレスが入っています。

レジスタがどのように並んでいるかは決まっているので、先頭アドレスが分かれば、アクセスしたいレジスタの位置までオフセットを足せば良いです。そうして得られたアドレスをポインタアクセスで読み書きすれば、レジスタに対してアクセスできます。

なお、64 ビットで使用時、HPET の各レジスタはサイズが 64 ビット (8 バイト) で、読み書きの単位も 64 ビットあるいは 32 ビットの単位で行うことが決められています。そのため、レジスタの特定の 1 ビットだけを書き換えたい場合も、64 ビット分全てを変数へ読み出して、読み出した変数上で目的の 1 ビットの書き換えを行った後に、書き換え済みの変数で 64 ビット分全てをレジスタへ書き戻す、といういわゆる read-modify-write を行います。

2.1.2 実装: レジスタ郡の先頭アドレス取得

まずはレジスタ郡の先頭アドレスを取得する処理を実装します。HPET に関する処理は hpet.c というソースファイルへ記述することにし、hpet_init という関数を作成してそこでレジスタのアドレス取得まで行います (リスト 2.1)。

▼**リスト** 2.1　020_dump_regs/hpet.c

```
/* 追加（ここから） */
#include <acpi.h>

struct __attribute__((packed)) HPET_TABLE {
        struct SDTH header;
        unsigned int event_timer_block_id;
        struct ACPI_ADDRESS base_address;
        unsigned char hpet_number;
        unsigned short minimum_tick;
        unsigned char flags;
};

unsigned long long reg_base;

void hpet_init(void)
{
        /* HPET table を取得 */
        struct HPET_TABLE *hpet_table = (struct HPET_TABLE *)get_sdt("HPET");

        /* レジスタの先頭アドレスを取得 */
        reg_base = hpet_table->base_address.address;
}
/* 追加（ここまで） */
```

なお、struct ACPI_ADDRESS は ACPI 側の仕様なので、acpi.h へ定義を追加します (リスト 2.2)。

▼**リスト** 2.2　020_dump_regs/include/acpi.h

```
#ifndef _ACPI_H_
#define _ACPI_H_

struct __attribute__((packed)) SDTH {
        char Signature[4];
        unsigned int Length;
        unsigned char Revision;
        unsigned char Checksum;
        char OEMID[6];
        char OEM_Table_ID[8];
        unsigned int OEM_Revision;
        unsigned int Creator_ID;
        unsigned int Creator_Revision;
```

```
};

/* 追加（ここから）*/
struct __attribute__((packed)) ACPI_ADDRESS {
        unsigned char address_space_id;
        unsigned char register_bit_width;
        unsigned char register_bit_offset;
        unsigned char _reserved;
        unsigned long long address;
};
/* 追加（ここまで）*/

/* ・・・ 省略 ・・・ */
```

include/hpet.h を作成し hpet_init 関数のプロトタイプ宣言を追加し (リスト 2.3)、main.c から呼び出すようにしておきます (リスト 2.4)。

▼**リスト** 2.3　020_dump_regs/include/hpet.h

```
/* 追加（ここから）*/
#ifndef _HPET_H_
#define _HPET_H_

void hpet_init(void);

#endif
/* 追加（ここまで）*/
```

▼**リスト** 2.4　020_dump_regs/main.c

```
/* ・・・ 省略 ・・・ */
#include <fs.h>
#include <hpet.h>       /* 追加 */
#include <common.h>

/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* フレームバッファ周りの初期化 */
        /* ・・・ 省略 ・・・ */

        /* ACPI の初期化 */
        acpi_init(pi->rsdp);

        /* 変更（ここから）*/
        /* HPET の初期化 */
        hpet_init();
        /* 変更（ここまで）*/
        while (1);
```

```
        /* ・・・ 省略 ・・・ */
```

最後に Makefile へ hpet.o をコンパイル対象に追加したら (リスト 2.5) この節での作業は終わりです。

▼**リスト** 2.5　020_dump_regs/Makefile

```
# ・・・ 省略 ・・・
OBJS = main.o iv.o fbcon.o fb.o font.o kbc.o x86.o intr.o pic.o \
       hpet.o acpi.o handler.o fs.o common.o   # hpet.o を追加
# ・・・ 省略 ・・・
```

2.1.3　実装: GCIDR を見てみる

それでは、何か HPET のレジスタを表示させてみます。

まずは HPET のハードウェア固有情報が格納されている読み取り専用のレジスタである「General Capabilities and ID Register(GCIDR)」の内容を表示させてみることにします。

レジスタ GCIDR の各ビットの意味は表 2.1 の通りです。表 2.1 では、使わないビットは省略しています。全てのビットを知りたい場合は HPET のデータシート[*1]を参照してください。

▼**表** 2.1　GCIDR **について**

ビット 0〜7	REV_ID	実装されている機能のリビジョン番号
ビット 8〜12	NUM_TIM_CAP	持っているタイマーの数
ビット 13	COUNT_SIZE_CAP	カウンタサイズ:32bit(=0)/64bit(=1)
ビット 14	RESERVED	予約領域
ビット 15	LEG_RT_CAP	LegacyReplacementRoute
	・・・	
ビット 32〜64	COUNTER_CLK_PERIOD	カウント周期 (フェムト秒)

特に COUNTER_CLK_PERIOD は重要な値です。HPET は固定周期でカウントアップするカウンタがあり、そのカウンタがあらかじめ設定された値になったら割り込みを発生させる、といった挙動が基本的な挙動となりますが、そのカウンタのカウントアップ周期が COUNTER_CLK_PERIOD で示されています。単位のフェムト秒は「10 の

[*1] 本書末尾の「参考情報」参照

マイナス 15 乗」秒です。なお、GCIDR は読み取り専用のレジスタですので、周期を変えることはできません。

また、COUNT_SIZE_CAP について、本書では 64 ビット幅の HPET を対象としているため、もしこの設定が 0(32 ビット) の場合は本書の 64 ビットで扱っている箇所を 32 ビットへ読み替える必要があります[*2]。

NUM_TIM_CAP が示す「タイマーの個数」は「比較器の個数」と考えた方が良いです。カウンタは 1 つの HPET で 1 つです。「この値になったら割り込みを発生させる」といった値の設定や割り込みの挙動設定などを行うレジスタが 1 つの HPET に対して NUM_TIM_CAP の個数分存在します。

LEG_RT_CAP は旧来の割り込み経路が使用できるか否かを示します。1 がセットされていると「LegacyReplacementRoute」という機能が使用できることを示し、旧来の PIC を使用した経路で割り込みを通知できます。このビットはあくまでも HPET が LegacyReplacementRoute に対応しているか否かで、実際に使用するか否かの設定は後述する別のレジスタで行います。

以上を踏まえて、GCIDR の定義と GCIDR を表示する関数 dump_gcidr を hpet.c へ追加します (リスト 2.6)。

▼**リスト 2.6**　020_dump_regs/hpet.c

```
#include <acpi.h>
#include <fbcon.h>      /* 追加 */

/* ・・・ 省略 ・・・ */

/* 追加（ここから） */
/* General Capabilities and ID Register */
#define GCIDR_ADDR     (reg_base)
#define GCIDR  (*(volatile unsigned long long *)GCIDR_ADDR)
union gcidr {
        unsigned long long raw;
        struct __attribute__((packed)) {
                unsigned long long rev_id:8;
                unsigned long long num_tim_cap:5;
                unsigned long long count_size_cap:1;
                unsigned long long _reserved:1;
                unsigned long long leg_rt_cap:1;
                unsigned long long vendor_id:16;
                unsigned long long counter_clk_period:32;
        };
};
/* 追加（ここまで） */

void hpet_init(void)
```

[*2] そもそも本書が 64 ビットの x86 を対象としているため、そのような環境で HPET だけ 32 ビットということは無いと思いますが。

```
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void dump_gcidr(void)
{
        puts("GCIDR\r\n");

        union gcidr r;
        r.raw = GCIDR;

        puts("REV ID             ");
        putd(r.rev_id, 3);
        puts("\r\n");

        puts("NUM TIM CAP        ");
        putd(r.num_tim_cap, 2);
        puts("\r\n");

        puts("COUNT SIZE CAP     ");
        putd(r.count_size_cap, 1);
        puts("\r\n");

        puts("LEG RT CAP         ");
        putd(r.leg_rt_cap, 1);
        puts("\r\n");

        puts("COUNTER CLK PERIOD ");
        putd(r.counter_clk_period, 10);
        puts("\r\n");
}
/* 追加（ここまで） */
```

リスト 2.6 について、hpet.c の上部でレジスタアクセスを簡単に行うための定数と共用体 (&構造体) を定義しています。

GCIDR はレジスタ郡の中の先頭のレジスタなので、hpet_init 関数で設定した reg_base がそのまま GCIDR のアドレスとなります。

定数「GCIDR」はレジスタアクセスを行うための定数です。この定数に対して読み書きを行うことでレジスタ GCIDR への読み書きが行えます。

続いて、レジスタのビットフィールドへの操作が簡単に行えるように構造体のビットフィールドを定義しています。共用体として unsigned long long 型の変数としてもアクセスできるようにしているのは、read-modify-write で 64 ビット分丸ごと読み出したり書き込んだりするためです。構造体側は読み出した後で特定のビットのみアクセスする際に使います。

以上の定義を使用して dump_gcidr 関数では GCIDR の内容を表示しています。「r.raw = GCIDR;」がレジスタ GCIDR の 64 ビット分すべてを共用体変数 r の val へ読み出している処理で、その後、変数 r の構造体側のメンバを使用して特定のビットのみ表

示させています。

あとは、hpet.h へプロトタイプ宣言を追加し (リスト 2.7)、main.c から呼び出します (リスト 2.8)。

▼**リスト** 2.7　020_dump_regs/include/hpet.h

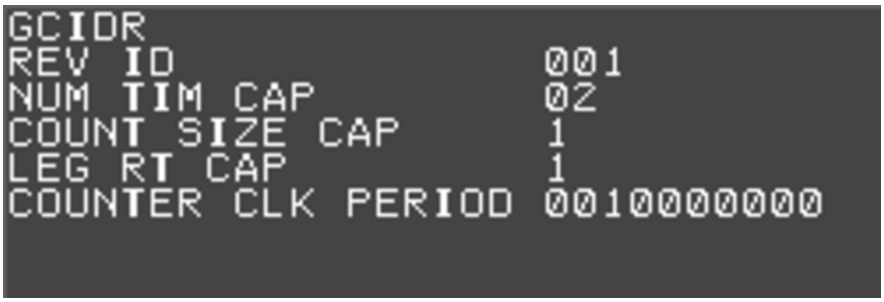
```
#ifndef _HPET_H_
#define _HPET_H_

void hpet_init(void);
void dump_gcidr(void);  /* 追加 */

#endif
```

▼**リスト** 2.8　020_dump_regs/main.c

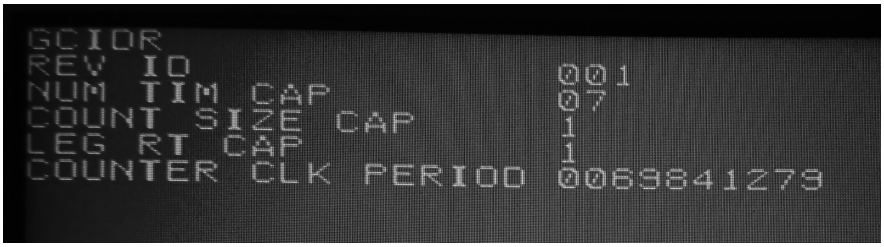
```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* フレームバッファ周りの初期化 */
        /* ・・・ 省略 ・・・ */

        /* ACPI の初期化 */
        acpi_init(pi->rsdp);

        /* HPET の初期化 */
        hpet_init();
        dump_gcidr();   /* 追加 */
        while (1);

        /* ・・・ 省略 ・・・ */
```

2.1.4　動作確認

実行してみると QEMU の場合、図 2.2 の様に表示されます。また、筆者の実機 (Lenovo ThinkPad E450) で試すと図 2.3 の通りでした。



▲図 2.2　GCIDR のダンプ結果 (QEMU)



▲図 2.3　GCIDR のダンプ結果 (実機)

図 2.2 と図 2.3 の結果から、表 2.2 がわかります。

▼表 2.2　QEMU と実機での GCIDR ダンプ結果からわかること

	QEMU	実機
タイマーの数 (NUM_TIM_CAP)	2	7
カウンタサイズ (COUNT_SIZE_CAP)	64bit	64bit
LegacyReplacementRoute(LEG_RT_CAP)	使用可	使用可
カウント周期 (COUNTER_CLK_PERIOD)	10ns	約 70ns

2.1.5　実装: その他のレジスタも見てみる

HPET の挙動を把握するために同様に以下 2 つのレジスタも見てみます。

- General Configuration Register (GCR)
- Main Counter Register (MCR)

GCR は HPET 自体の有効化/無効化などを含む全タイマーで共通の設定を行う読み書き可能なレジスタです (表 2.3)。

▼表 2.3　GCR について

ビット 0	ENABLE_CNF	HPET 自体の有効/無効
ビット 1	LEG_RT_CNF	LegacyReplacementRoute の有効/無効
ビット 2〜63	RESERVED	予約領域

ENABLE_CNF フラグは「HPET 自体を止めたい/開始させたい」場合に使っていきます。このフラグ無効化することで HPET のカウンタが止まり、割り込みも発生しなくなります。(逆に有効化すればカウンタ動作が開始し、別途設定する割り込み設定が有効化されていれば割り込みが発生するようになります。)

LEG_RT_CNF は、LegacyReplacementRoute を使用するか否かの設定です。本書の場合、前著[*3]にて PIC を使用していましたので、本著でも PIC を使って割り込みを扱うことにします[*4]。

次に、MCR は HPET に一つだけ存在するすべてのタイマーで共通の読み書き可能なカウンタです (表 2.4)。

▼表 2.4　MCR について

ビット 0〜63	カウンタ値

それでは、GCR と MCR の定義と、それぞれの内容を表示する dump_gcr 関数と dump_mcr 関数を hpet.c へ追加します (リスト 2.9)。

▼リスト 2.9　020_dump_regs/hpet.c

```
/* ・・・ 省略 ・・・ */

/* General Capabilities and ID Register */
#define GCIDR_ADDR     (reg_base)
#define GCIDR  (*(volatile unsigned long long *)GCIDR_ADDR)
union gcidr {
        /* ・・・ 省略 ・・・ */
};

/* 追加（ここから） */
/* General Configuration Register */
```

[*3] フルスクラッチで作る!x86_64 自作 OS

[*4] 新しい (今の) 割り込みコントローラである APIC は、また別の機会にチャレンジしてみたいです。本書では HPET がちゃんと制御できることにフォーカスします。

```
#define GCR_ADDR       (reg_base + 0x10)
#define GCR    (*(volatile unsigned long long *)GCR_ADDR)
union gcr {
        unsigned long long raw;
        struct __attribute__((packed)) {
                unsigned long long enable_cnf:1;
                unsigned long long leg_rt_cnf:1;
                unsigned long long _reserved:62;
        };
};

/* Main Counter Register */
#define MCR_ADDR       (reg_base + 0xf0)
#define MCR    (*(volatile unsigned long long *)MCR_ADDR)
/* 追加（ここまで） */

/* ・・・ 省略 ・・・ */

void dump_gcidr(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void dump_gcr(void)
{
        puts("GCR\r\n");

        union gcr r;
        r.raw = GCR;

        puts("ENABLE CNF ");
        putd(r.enable_cnf, 1);
        puts("\r\n");

        puts("LEG RT CNF ");
        putd(r.leg_rt_cnf, 1);
        puts("\r\n");
}

void dump_mcr(void)
{
        puts("MCR ");
        puth(MCR, 16);
        puts("\r\n");
}
/* 追加（ここまで） */
```

プロトタイプ宣言を hpet.h へ追加し (リスト 2.10)、main.c へ呼び出し処理を追加したら (リスト 2.11) 終わりです。

▼**リスト** 2.10　020_dump_regs/include/hpet.h

```
#ifndef _HPET_H_
#define _HPET_H_

void hpet_init(void);
void dump_gcidr(void);
void dump_gcr(void);    /* 追加 */
void dump_mcr(void);    /* 追加 */

#endif
```

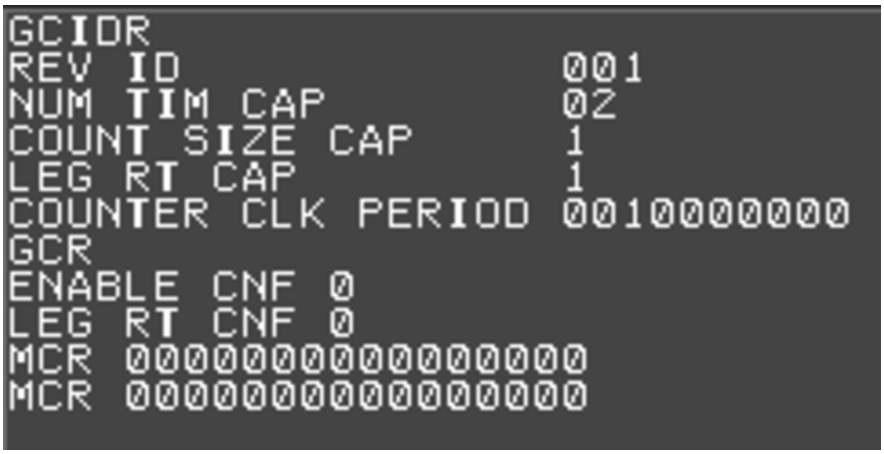
▼**リスト** 2.11　020_dump_regs/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* フレームバッファ周りの初期化 */
        /* ・・・ 省略 ・・・ */

        /* ACPI の初期化 */
        acpi_init(pi->rsdp);

        /* HPET の初期化 */
        hpet_init();
        dump_gcidr();
        /* 追加（ここから） */
        dump_gcr();
        dump_mcr();
        volatile unsigned long long wait = 1000000;
        while (wait--);
        dump_mcr();
        /* 追加（ここまで） */
        while (1);

        /* ・・・ 省略 ・・・ */
```

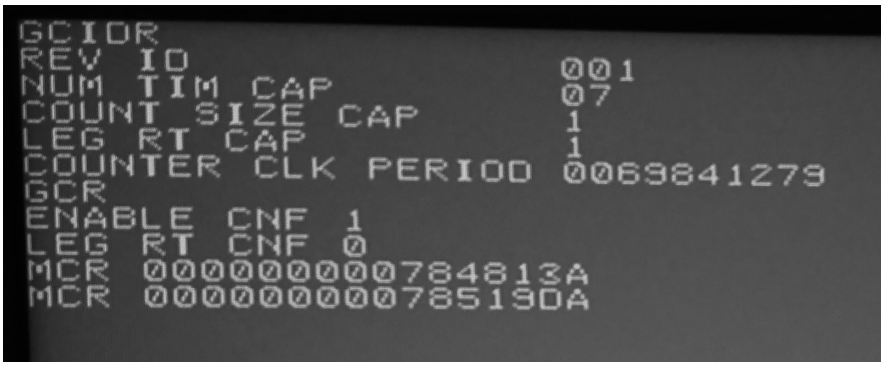
カウンタが動いていれば変化が分かるように dump_mcr 関数はウェイトをはさんで 2 回実行しています。

2.1.6　動作確認

実行すると QEMU では図 2.4 のように、実機では図 2.5 のように表示されました。



▲図 2.4　020_dump_regs の実行結果 (QEMU、GCR と MCR 追加版)



▲図 2.5　020_dump_regs の実行結果 (実機、GCR と MCR 追加版)

図 2.4 と図 2.5 の結果から、以下が分かります。

- QEMU は初期状態で HPET は動作していない (ENABLE_CNF=0) が、実機は既に動作しており (ENABLE_CNF=1)MCR も進んでいる
- LegacyReplacementRoute は QEMU・実機共に初期状態は無効 (LEG_RT_CNF=0) の状態

2.1.7 実装: HPET 初期化時に HPET を明示的に無効化

これまでの動作確認から、起動時に HPET が動いているか停止しているかはマシンによって (少なくとも QEMU と実機で) 異なることが分かりました。

次項から実装する HPET を動作させる関数を呼び出すまでの間、HPET が動いている必要は無いので、hpet_init 関数で HPET を一旦無効化しておくことにします (リスト 2.12)。

▼**リスト** 2.12　020_dump_regs/hpet.c

```
/* ・・・ 省略 ・・・ */

void hpet_init(void)
{
        /* HPET table を取得 */
        struct HPET_TABLE *hpet_table = (struct HPET_TABLE *)get_sdt("HPET");

        /* レジスタの先頭アドレスを取得 */
        reg_base = hpet_table->base_address.address;

        /* 追加（ここから）*/
        /* 使うまで HPET は止めておく */
        union gcr gcr;
        gcr.raw = GCR;
        gcr.enable_cnf = 0;
        GCR = gcr.raw;
        /* 追加（ここまで）*/
}

/* ・・・ 省略 ・・・ */
```

2.2 sleep を実装してみる

まずは、割り込みを使わない簡単な例として、指定された時間処理を止める「sleep 関数」を実装してみます。

この項のサンプルディレクトリは「021_sleep」です。

2.2.1 やること

前項で確認した GCR と MCR のレジスタを使って、以下の方針で実装します。

1. MCR の enable_cnf が無効であれば有効化する
2. MCR を監視して時間経過を待つ

3. MCR の enable_cnf を元に戻す

2.2.2 実装

さっそく実装してみます (リスト 2.13)。

▼**リスト** 2.13　021_sleep/hpet.c

```
#include <acpi.h>
#include <fbcon.h>

#define US_TO_FS       1000000000      /* 追加 */

/* ・・・ 省略 ・・・ */

void dump_mcr(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void sleep(unsigned long long us)
{
        /* 現在の main counter のカウント値を取得 */
        unsigned long long mc_now = MCR;

        /* us マイクロ秒後の main counter のカウント値を算出 */
        unsigned long long fs = us * US_TO_FS;
        union gcidr gcidr;
        gcidr.raw = GCIDR;
        unsigned long long mc_duration = fs / gcidr.counter_clk_period;
        unsigned long long mc_after = mc_now + mc_duration;

        /* HPET が無効であれば有効化する */
        union gcr gcr;
        gcr.raw = GCR;
        unsigned char to_disable = 0;
        if (!gcr.enable_cnf) {
                gcr.enable_cnf = 1;
                GCR = gcr.raw;

                /* sleep() を抜ける際に元に戻す（disable する） */
                to_disable = 1;
        }

        /* us マイクロ秒の経過を待つ */
        while (MCR < mc_after);

        /* 元々無効であった場合は無効に戻しておく */
        if (to_disable) {
                gcr.raw = GCR;
                gcr.enable_cnf = 0;
                GCR = gcr.raw;
        }
}
```

```
/* 追加（ここまで） */
```

sleep 関数の引数で指定できる単位はマイクロ秒 (us) としました。HPET のカウンタのカウント周期は数十 ns(QEMU では 10ns) でしたので、引数をナノ秒とすると引数に 1 を指定された時、最低でも数十ナノ秒 sleep してしまいます。関数の仕様として「引数には HPET の周期以上を指定すること」とするのも微妙に感じたので、単位を切り上げて引数にはマイクロ秒を指定することとしました。

sleep 関数冒頭では、MCR の現在のカウント値を取得し、引数 us をカウンタのカウント数へ変換した値を足すことで、指定された時間経過後のカウンタ値を求めています。

その後は、GCR で HPET が無効であれば有効化し (その際は「後で無効化する」旨のフラグをセットし)、MCR が先ほど求めたカウンタ値以上になるまで待機、最後に元々 HPET が無効であったならば無効化して終わりです。

実装の最後に hpet.h へ sleep 関数のプロトタイプ宣言を追加し (リスト 2.14)、main.c へ動作確認用の sleep 関数呼び出し処理を追加します (リスト 2.15)。

▼**リスト** 2.14　021_sleep/include/hpet.h

```
#ifndef _HPET_H_
#define _HPET_H_

#define MS_TO_US        1000            /* 追加 */
#define SEC_TO_US       1000000         /* 追加 */

void hpet_init(void);
void dump_gcidr(void);
void dump_gcr(void);
void dump_mcr(void);
void sleep(unsigned long long us);      /* 追加 */

#endif
```

▼**リスト** 2.15　021_sleep/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* フレームバッファ周りの初期化 */
        /* ・・・ 省略 ・・・ */

        /* ACPI の初期化 */
        acpi_init(pi->rsdp);
```

```
        /* HPET の初期化 */
        hpet_init();

        /* 変更（ここから）*/
        /* 5 秒 sleep してみる */
        puts("WAIT ...");
        sleep(5 * SEC_TO_US);
        puts(" DONE\r\n");
        /* 変更（ここまで）*/

        while (1);

        /* ・・・ 省略 ・・・ */
```

2.2.3 動作確認

実行すると"WAIT ..."が表示されてから (図 2.6)5 秒後に" DONE"が表示されます (図 2.7)。

▲図 2.6　021_sleep の実行結果 ("WAIT ...")

▲図 2.7　021_sleep の実行結果 (" DONE")

2.3 alert を実装してみる

次は割り込みを使ってみます。割り込みを使用した例の第 1 弾として、指定された時間経過後に指定された処理を行う alert 関数を実装してみます。

この項のサンプルディレクトリは「022_alert」です。

2.3.1 やること

やることは大きく分けて以下の 2 つです。

1. HPET の割り込み設定
2. ハンドラの設定

【HPET の割り込み設定】

HPET の割り込み設定は「Timer N Configuration and Capabilities Register (TNCCR)」というタイマー毎に存在するレジスタへ行います (表 2.5)。

▼表 2.5 TNCCR について

ビット 0	RESERVED	予約領域
ビット 1	INT_TYPE_CNF	エッジトリガー (=0)/レベルトリガー (=1)
ビット 2	INT_ENB_CNF	割り込み無効 (=0)/有効 (=1)
ビット 3	TYPE_CNF	非周期割り込み (=0)/周期割り込み (=1)
ビット 4	PER_INT_CAP	周期タイマーをサポートするか否か (ro)
ビット 5	SIZE_CAP	32 ビットサイズ (=0)/64 ビットサイズ (=1)(ro)
ビット 6	VAL_SET_CNF	タイマーのアキュムレータへのダイレクトアクセス許可
ビット 7	RESERVED	予約領域
ビット 8	32MODE_CNF	強制的に 32 ビットモードにする
ビット 9〜13	INT_ROUTE_CNF	割り込み経路設定
ビット 14	FSB_EN_CNF	Front Side Bus(FSB) を使用するか否か
ビット 15	FSB_INT_DEL_CAP	FSB をサポートしているか否か (ro)
ビット 16	RESERVED	予約領域
ビット 17〜63	INT_ROUTE_CAP	割り込みが配送される I/O APIC を示す (ro)

「(ro)」が付いているビットは書き換えできないビットです。

LegacyReplacementRoute 使用時、割り込み経路は旧来の PIC で決まっているので、INT_ROUTE_CNF 等は設定しても何も起こりません。

今回 TNCCR は以下のように設定します。

- INT_TYPE_CNF: エッジトリガー
- INT_ENB_CNF: 割り込み有効
- TYPE_CNF: 非周期割り込み (ワンショット)
- 32MODE_CNF: 0(64 ビットモードのまま)
- FSB_EN_CNF: FSB 使わない

- RESERVED: 0

その他のビットは特に意識する必要はないので、何もしません。

また、TNCCR の場合はレジスタへの書き込みアクセス時、RESERVED には 0 を書き込むことがデータシートで指定されている[*5]ので、明示的に 0 を書くようにします。

そして、「いつ割り込みを発生させるか」を設定するのが「Timer N Comparator Register (TNCR)」という、これもタイマー毎に存在するレジスタです (表 2.6)。このレジスタを指して単に「コンパレータ (比較器)」と呼んだりもします。

▼表 2.6　TNCR について

ビット 0〜63	割り込みを発生させたいカウンタの値

最後が GCR の leg_rt_cnf です。本書では PIC で割り込みを扱うので、このフラグを有効化しておきます。

【ハンドラの設定】

割り込みのハンドラの設定としては、まず、x86 CPU が持つ割り込み設定のテーブル (IDT) の設定と、PIC 側で HPET の割り込み番号の割り込みのマスクを解除する設定が必要です。

いずれも前著で set_intr_desc 関数 (intr.c)、enable_pic_intr 関数 (pic.c) を作っているので、それらを使います。

また、set_intr_desc 関数ではその割り込みが発生したときに呼び出すハンドラを引数でしていする必要があるので、HPET 割り込み発生時に呼び出される HPET 側のハンドラ関数も用意します。

なお、今回実装する alert 関数では、指定された時間経過後に実施したい処理を関数ポインタとして引数に指定することとします。HPET 側は alert 関数呼び出しで指定された関数を、割り込み発生時に HPET のハンドラから呼び出すことにします。

2.3.2　実装

【HPET の割り込み設定】

まず、hpet.c へ TNCCR と TNCR の定義を追加します (リスト 2.16)。

▼**リスト** 2.16　022_alert/hpet.c

[*5] IA-PC HPET Specification Rev 1.0a P.17 Timer N Configuration and Capability Register Field Definitions

```
#include <acpi.h>
#include <fbcon.h>

#define TIMER_N                0       /* 使用するタイマー番号 */  /* 追加 */

#define US_TO_FS       1000000000

/* ・・・ 省略 ・・・ */

/* Main Counter Register */
#define MCR_ADDR       (reg_base + 0xf0)
#define MCR    (*(volatile unsigned long long *)MCR_ADDR)

/* 追加（ここから） */
/* Timer N Configuration and Capabilities Register */
#define TNCCR_ADDR(n)  (reg_base + (0x20 * (n)) + 0x100)
#define TNCCR(n)       (*(volatile unsigned long long *)(TNCCR_ADDR(n)))
#define TNCCR_INT_TYPE_EDGE    0
#define TNCCR_INT_TYPE_LEVEL   1
#define TNCCR_TYPE_NON_PERIODIC        0
#define TNCCR_TYPE_PERIODIC    1
union tnccr {
        unsigned long long raw;
        struct __attribute__((packed)) {
                unsigned long long _reserved1:1;
                unsigned long long int_type_cnf:1;
                unsigned long long int_enb_cnf:1;
                unsigned long long type_cnf:1;
                unsigned long long per_int_cap:1;
                unsigned long long size_cap:1;
                unsigned long long val_set_cnf:1;
                unsigned long long _reserved2:1;
                unsigned long long mode32_cnf:1;
                unsigned long long int_route_cnf:5;
                unsigned long long fsb_en_cnf:1;
                unsigned long long fsb_int_del_cap:1;
                unsigned long long _reserved3:16;
                unsigned long long int_route_cap:32;
        };
};

/* Timer N Comparator Register */
#define TNCR_ADDR(n)   (reg_base + (0x20 * (n)) + 0x108)
#define TNCR(n)        (*(volatile unsigned long long *)(TNCR_ADDR(n)))
/* 追加（ここまで） */

/* ・・・ 省略 ・・・ */
```

併せて使用するタイマー番号の定義「TIMER_N」も追加しておきました。

次に、hpet_init 関数へ割り込み周りの初期設定を追加します (リスト 2.17)。

▼**リスト** 2.17　022_alert/hpet.c

```
/* ・・・ 省略 ・・・ */

unsigned long long reg_base;
unsigned int counter_clk_period;        /* 追加 */

void hpet_init(void)
{
        /* HPET table を取得 */
        struct HPET_TABLE *hpet_table = (struct HPET_TABLE *)get_sdt("HPET");

        /* レジスタの先頭アドレスを取得 */
        reg_base = hpet_table->base_address.address;

        /* 使うまで HPET は止めておく
         * 併せて Legacy Replacement Route 有効化 */    /* 追加 */
        union gcr gcr;
        gcr.raw = GCR;
        gcr.enable_cnf = 0;
        gcr.leg_rt_cnf = 1;                             /* 追加 */
        GCR = gcr.raw;

        /* 追加（ここから） */
        /* カウント周期を取得 */
        union gcidr gcidr;
        gcidr.raw = GCIDR;
        counter_clk_period = gcidr.counter_clk_period;

        /* 割り込み設定初期化 */
        union tnccr tnccr;
        tnccr.raw = TNCCR(TIMER_N);
        tnccr.int_type_cnf = TNCCR_INT_TYPE_EDGE;
        tnccr.int_enb_cnf = 0;
        tnccr.type_cnf = TNCCR_TYPE_NON_PERIODIC;
        tnccr.val_set_cnf = 0;
        tnccr.mode32_cnf = 0;
        tnccr.fsb_en_cnf = 0;
        tnccr._reserved1 = 0;
        tnccr._reserved2 = 0;
        tnccr._reserved3 = 0;
        TNCCR(TIMER_N) = tnccr.raw;
        /* 追加（ここまで） */
}

/* ・・・ 省略 ・・・ */
```

type_cnf の初期値を「非周期割り込み」側へ倒しているのは、これがどちらであるかによって val_set_cnf へ「書いてはいけない値」が変わるからです。

val_set_cnf はデータシートを読む限り、周期割り込み時のみ「動作中だけどコンパレータを書き換えるよ」ということを伝えるレジスタのようで、かつ「1 は書いて良いが 0 は書いてはいけない (自動的に 0 に戻る)」と決められています。また、「周期割り込みでは無いときに 1 を書いてはいけない」という決まりもあります。

そこで、初期化時は「非周期割り込み」として、val_set_cnf へは 0 を書いています。

また、リスト 2.17 では、カウンタのカウント周期をあらかじめ取得し、グローバル変

数 counter_clk_period へ格納するようにしています。

そして、alert 関数を追加し、割り込みや HPET 自体の有効化やコンパレータ設定を行います (リスト 2.18)。(hpet.h への alert 関数のプロトタイプ宣言追加もよしなにやっておきます。)

▼**リスト** 2.18　022_alert/hpet.c

```
/* ・・・ 省略 ・・・ */

void sleep(unsigned long long us)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void alert(unsigned long long us, void *handler)
{
        /* 非周期割り込みで割り込み有効化 */
        union tnccr tnccr;
        tnccr.raw = TNCCR(TIMER_N);
        tnccr.int_enb_cnf = 1;
        tnccr.type_cnf = TNCCR_TYPE_NON_PERIODIC;
        tnccr._reserved1 = 0;
        tnccr._reserved2 = 0;
        tnccr._reserved3 = 0;
        TNCCR(TIMER_N) = tnccr.raw;

        /* main counter をゼロクリア */
        MCR = (unsigned long long)0;

        /* コンパレータ設定 */
        unsigned long long femt_sec = us * US_TO_FS;
        unsigned long long clk_counts = femt_sec / counter_clk_period;
        TNCR(TIMER_N) = clk_counts;

        /* HPET 有効化 */
        union gcr gcr;
        gcr.raw = GCR;
        gcr.enable_cnf = 1;
        GCR = gcr.raw;
}
/* 追加（ここまで） */
```

引数で受け取った handler の設定などは次節で行います。

【ハンドラの設定】

まず、割り込み発生時に一番最初に呼び出されるハンドラ「hpet_handler」を handler.s に作成します (リスト 2.19)。

▼**リスト** 2.19　022_alert/handler.s

```
        .global default_handler
default_handler:
        jmp     default_handler

        /* 追加（ここから）*/
        .global hpet_handler
hpet_handler:
        push    %rax
        push    %rcx
        push    %rdx
        push    %rbx
        push    %rbp
        push    %rsi
        push    %rdi
        mov     %rsp, %rdi
        call    do_hpet_interrupt
        pop     %rdi
        pop     %rsi
        pop     %rbp
        pop     %rbx
        pop     %rdx
        pop     %rcx
        pop     %rax
        iretq
        /* 追加（ここまで）*/

        .global kbc_handler
kbc_handler:
        /* ・・・ 省略 ・・・ */
```

hpet_handler から呼び出される do_hpet_interrupt 関数も作成しておきます (リスト 2.20)。

▼**リスト** 2.20　022_alert/hpet.c

```
#include <pic.h>        /* 追加 */
#include <acpi.h>
#include <fbcon.h>
#include <hpet.h>       /* 追加 */

#define TIMER_N                 0       /* 使用するタイマー番号 */

#define US_TO_FS        1000000000

/* ・・・ 省略 ・・・ */

void sleep(unsigned long long us)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから）*/
void do_hpet_interrupt(unsigned long long current_rsp)
{
        /* HPET 無効化 */
```

```
        union gcr gcr;
        gcr.raw = GCR;
        gcr.enable_cnf = 0;
        GCR = gcr.raw;

        /* 割り込みを無効化 */
        union tnccr tnccr;
        tnccr.raw = TNCCR(TIMER_N);
        tnccr.int_enb_cnf = 0;
        tnccr._reserved1 = 0;
        tnccr._reserved2 = 0;
        tnccr._reserved3 = 0;
        TNCCR(TIMER_N) = tnccr.raw;

        /* ユーザーハンドラを呼び出す */
        if (user_handler)
                user_handler(current_rsp);

        /* PIC へ割り込み処理終了を通知 (EOI) */
        set_pic_eoi(HPET_INTR_NO);
}
/* 追加 (ここまで) */

void alert(unsigned long long us, void *handler)
{
        /* ・・・ 省略 ・・・ */
}
```

HPET の割り込み番号 (HPET_INTR_NO) は今後 hpet.c の外でも使うため hpet.h へ追加します (リスト 2.21)。alert 関数のプロトタイプ宣言も併せて追加しておきます (リスト 2.21)。

▼**リスト** 2.21　022_alert/include/hpet.h

```
#ifndef _HPET_H_
#define _HPET_H_

#define MS_TO_US        1000
#define SEC_TO_US       1000000
#define HPET_INTR_NO    32              /* 追加 */

void hpet_init(void);
void dump_gcidr(void);
void dump_gcr(void);
void dump_mcr(void);
void sleep(unsigned long long us);
void alert(unsigned long long us, void *handler);       /* 追加 */

#endif
```

なお、HPET の割り込み番号の定数「HPET_INTR_NO」を「32」として追加してい

ますが、これは PIC の時代から決められているものです[*6]。

また、HPET の C 言語部分のハンドラ (do_hpet_interrupt 関数) にスタックポインタを渡しているのは、次章でスケジューラを実装する際に使用するためです。

そして、do_hpet_interrupt 関数から呼び出される user_handler 関数も関数ポインタをグローバル変数として用意しておき、alert 関数内で関数ポインタを設定するようにします (リスト 2.22)。

▼**リスト** 2.22　022_alert/hpet.c

```
#include <pic.h>
#include <acpi.h>
#include <fbcon.h>
#include <hpet.h>
#include <common.h>     /* 追加 */
/* ・・・ 省略 ・・・ */

/* Timer N Comparator Register */
#define TNCR_ADDR(n)   (reg_base + (0x20 * (n)) + 0x108)
#define TNCR(n)        (*(volatile unsigned long long *)(TNCR_ADDR(n)))

void (*user_handler)(unsigned long long current_rsp) = NULL;    /* 追加 */

/* ・・・ 省略 ・・・ */

void alert(unsigned long long us, void *handler)
{
        /* ユーザーハンドラ設定 */        /* 追加 */
        user_handler = handler; /* 追加 */

        /* ・・・ 省略 ・・・ */
```

ここまでで各種ハンドラの作成は完了です。

x86 CPU の割り込み設定 (IDT) へ最初に呼び出すハンドラ (hpet_handler) の登録と、PIC へ HPET の割り込みマスクの解除を hpet_init 関数へ追加します (リスト 2.23)。

▼**リスト** 2.23　022_alert/hpet.c

```
#include <intr.h>       /* 追加 */
#include <pic.h>
#include <acpi.h>
#include <fbcon.h>
#include <hpet.h>
#include <common.h>
```

[*6] 割り込み番号 32 は IRQ 番号だと 0 で、厳密には IRQ0 が PIC の時代からタイマーの割り込みと決められています。IRQ が 32 番から始まるのはそのように割り込みの設定を行っているからで、詳しくは前著「フルスクラッチで作る!x86_64 自作 OS(パート 1)」をご覧ください。

```
#define TIMER_N                0       /* 使用するタイマー番号 */

#define US_TO_FS       1000000000

/* ・・・ 省略 ・・・ */

void hpet_handler(void);        /* 追加 */
void (*user_handler)(unsigned long long current_rsp) = NULL;

void hpet_init(void)
{
        /* ・・・ 省略 ・・・ */

        /* 割り込み設定初期化 */
        union tnccr tnccr;
        tnccr.raw = TNCCR(TIMER_N);
        tnccr.int_type_cnf = TNCCR_INT_TYPE_EDGE;
        tnccr.int_enb_cnf = 0;
        tnccr.type_cnf = TNCCR_TYPE_NON_PERIODIC;
        tnccr.val_set_cnf = 0;
        tnccr.mode32_cnf = 0;
        tnccr.fsb_en_cnf = 0;
        tnccr._reserved1 = 0;
        tnccr._reserved2 = 0;
        tnccr._reserved3 = 0;
        TNCCR(TIMER_N) = tnccr.raw;

        /* 追加（ここから）*/
        /* IDT へ HPET 割り込みのハンドラ登録 */
        set_intr_desc(HPET_INTR_NO, hpet_handler);

        /* PIC の割り込みマスク解除 */
        enable_pic_intr(HPET_INTR_NO);
        /* 追加（ここまで）*/
}

/* ・・・ 省略 ・・・ */
```

ここまでで hpet.c の追加作業は完了で、あとは main.c から呼び出すようにします (リスト 2.24)。

▼**リスト** 2.24　022_alert/main.c

```
/* ・・・ 省略 ・・・ */

void handler(void);     /* 追加 */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* フレームバッファ周りの初期化 */
        fb_init(&pi->fb);
        set_fg(255, 255, 255);
        set_bg(0, 70, 250);
        clear_screen();
```

```
        /* ACPI の初期化 */
        acpi_init(pi->rsdp);

        /* CPU 周りの初期化 */
        gdt_init();
        intr_init();

        /* 周辺 IC の初期化 */
        pic_init();
        hpet_init();    /* 追加 */
        kbc_init();

        /* ファイルシステムの初期化 */
        fs_init(_fs_start);

        /* 追加（ここから） */
        /* 5 秒後に alert をセットしてみる */
        puts("WAIT ...");
        alert(5 * SEC_TO_US, handler);
        /* 追加（ここまで） */

        /* CPU の割り込み有効化 */
        enable_cpu_intr();

        /* halt して待つ */
        while (1)
                cpu_halt();
}

/* 追加（ここから） */
void handler(void)
{
        puts(" DONE");
}
/* 追加（ここまで） */
```

2.3.3 動作確認

実行すると sleep の例と同様に、"WAIT ..."が表示されてから 5 秒後に" DONE"が表示されます。(見た目は同じなのでスクリーンショットは省略)

2.4 周期タイマー (ptimer) を実装してみる

最後に指定された周期で周期的に動作させる「周期タイマー (ptimer)」を実装してみます。

この項のサンプルディレクトリは「023_ptimer」です。

2.4.1 やること

使うレジスタは alert を実装した際に使ったものと同じで、一部設定を変更するだけで周期タイマーになります。

なお、コンパレータの挙動としては、非周期割り込みの際は、カウンタがコンパレータと一致したら割り込みが発生してそれで終わりでした。周期割り込みの場合、割り込みが発生すると、コンパレータに最後に書き込まれた値が現在のコンパレータへ自動で加算されます。

例えば、コンパレータに最後に書き込まれた値が 0x0123 の場合、以下の挙動となります。

1. カウンタが 0x0123 へ到達し、割り込み発生
2. コンパレータへは最後に書かれた値が自動で加算され、新たなコンパレータ値は 0x0246 となる
3. カウンタが 0x0246 へ到達し、割り込み発生
4. コンパレータへは最後に書かれた値が自動で加算され、新たなコンパレータ値は 0x0369 となる

実装の方針としては、周期タイマーの設定を行う「ptimer_setup 関数」と周期タイマーを開始させる「ptimer_start 関数」、停止させる「ptimer_stop 関数」の 3 つを実装することにします。

2.4.2 実装

さっそく ptimer_setup 関数と ptimer_start 関数、ptimer_stop 関数を実装してみます (リスト 2.25)。

▼**リスト** 2.25　023_ptimer/hpet.c

```
/* ・・・ 省略 ・・・ */

unsigned long long reg_base;
unsigned int counter_clk_period;
unsigned long long cmpr_clk_counts;     /* 追加 */

/* ・・・ 省略 ・・・ */

void alert(unsigned long long us, void *handler)
{
        /* ・・・ 省略 ・・・ */
}
```

```
/* 追加（ここから） */
void ptimer_setup(unsigned long long us, void *handler)
{
        /* HPET 無効化 */
        union gcr gcr;
        gcr.raw = GCR;
        gcr.enable_cnf = 0;
        GCR = gcr.raw;

        /* ユーザーハンドラ設定 */
        user_handler = handler;

        /* 周期割り込みで割り込み有効化 */
        union tnccr tnccr;
        tnccr.raw = TNCCR(TIMER_N);
        tnccr.int_enb_cnf = 1;
        tnccr.type_cnf = TNCCR_TYPE_PERIODIC;
        tnccr._reserved1 = 0;
        tnccr._reserved2 = 0;
        tnccr._reserved3 = 0;
        TNCCR(TIMER_N) = tnccr.raw;

        /* コンパレータ設定値を計算しておく */
        unsigned long long femt_sec = us * US_TO_FS;
        cmpr_clk_counts = femt_sec / counter_clk_period;
}

void ptimer_start(void)
{
        /* コンパレータ初期化 */
        union tnccr tnccr;
        tnccr.raw = TNCCR(TIMER_N);
        tnccr.val_set_cnf = 1;
        TNCCR(TIMER_N) = tnccr.raw;
        TNCR(TIMER_N) = cmpr_clk_counts;

        /* main counter 初期化 */
        MCR = (unsigned long long)0;

        /* HPET 有効化 */
        union gcr gcr;
        gcr.raw = GCR;
        gcr.enable_cnf = 1;
        GCR = gcr.raw;
}

void ptimer_stop(void)
{
        /* HPET 無効化 */
        union gcr gcr;
        gcr.raw = GCR;
        gcr.enable_cnf = 0;
        GCR = gcr.raw;

        /* 割り込みを無効化 */
        union tnccr tnccr;
        tnccr.raw = TNCCR(TIMER_N);
        tnccr.int_enb_cnf = 0;
```

```
        tnccr._reserved1 = 0;
        tnccr._reserved2 = 0;
        tnccr._reserved3 = 0;
        TNCCR(TIMER_N) = tnccr.raw;
}
/* 追加（ここまで）*/
```

ptimer_setup 関数でコンパレータ値を計算するまでで、実際の設定を ptimer_start 関数で行うようにしているのは、ptimer_stop 関数で止められた後、ptimer_start 関数で再開させる際にコンパレータを初期化する必要があるからです。

また、HPET 側の割り込みハンドラである do_hpet_interrupt 関数冒頭でタイマーと割り込みの無効化を行っているので、このままだと 1 周期目の割り込みでそれらが無効化されてしまいます。

is_oneshot という「現在ワンショット動作中であるか否か」のフラグを追加します (リスト 2.26)。

▼**リスト** 2.26　023_ptimer/hpet.c

```
/* ・・・ 省略 ・・・ */

unsigned long long reg_base;
unsigned int counter_clk_period;
unsigned long long cmpr_clk_counts;
unsigned char is_oneshot = 0;   /* 追加 */

/* ・・・ 省略 ・・・ */

void do_hpet_interrupt(unsigned long long current_rsp)
{
        /* 変更（ここから）*/
        if (is_oneshot == 1) {
                /* HPET 無効化 */
                union gcr gcr;
                gcr.raw = GCR;
                gcr.enable_cnf = 0;
                GCR = gcr.raw;

                /* 割り込みを無効化 */
                union tnccr tnccr;
                tnccr.raw = TNCCR(TIMER_N);
                tnccr.int_enb_cnf = 0;
                tnccr._reserved1 = 0;
                tnccr._reserved2 = 0;
                tnccr._reserved3 = 0;
                TNCCR(TIMER_N) = tnccr.raw;

                /* ワンショットタイマー設定を解除 */
                is_oneshot = 0;
        }
        /* 変更（ここまで）*/
```

```
        /* ユーザーハンドラを呼び出す */
        if (user_handler)
                user_handler(current_rsp);

        /* PIC へ割り込み処理終了を通知 (EOI) */
        set_pic_eoi(HPET_INTR_NO);
}

void alert(unsigned long long us, void *handler)
{
        /* ・・・ 省略 ・・・ */

        /* コンパレータ設定 */
        unsigned long long femt_sec = us * US_TO_FS;
        unsigned long long clk_counts = femt_sec / counter_clk_period;
        TNCR(TIMER_N) = clk_counts;

        /* ワンショットタイマー設定 */      /* 追加 */
        is_oneshot = 1;                  /* 追加 */

        /* HPET 有効化 */
        union gcr gcr;
        gcr.raw = GCR;
        gcr.enable_cnf = 1;
        GCR = gcr.raw;
}

/* ・・・ 省略 ・・・ */
```

最後に hpet.h へプロトタイプ宣言を追加し (コード省略)、main.c から呼び出してみます (リスト 2.27)。

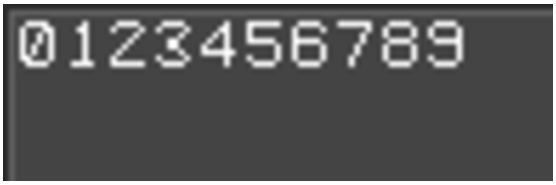
▼**リスト** 2.27　023_ptimer/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */

        /* ファイルシステムの初期化 */
        fs_init(_fs_start);

        /* 1 秒周期の周期タイマー設定 */             /* 変更 */
        ptimer_setup(1 * SEC_TO_US, handler);   /* 変更 */

        /* CPU の割り込み有効化 */
        enable_cpu_intr();

        /* 周期タイマースタート */                 /* 追加 */
        ptimer_start();                          /* 追加 */

        /* halt して待つ */
```

```
        while (1)
                cpu_halt();
}

void handler(void)
{
        /* 変更（ここから）*/
        static unsigned char counter = 0;
        if (counter < 10)
                putc('0' + counter++);
        else
                ptimer_stop();
        /* 変更（ここまで）*/
}
```

2.4.3 動作確認

実行すると「0」から「9」までの数字が 1 秒周期で出力されたところで停止します (図 2.8)。



▲図 2.8　023_ptimer の実行結果

第 3 章

スケジューラ

タイマーで周期割り込みを使うことができたので、簡単なスケジューラを実装してみます。

この章では、章を通して「030_sched」のサンプルディレクトリの内容を作っていきます。

3.1 やること

この章では簡単なスケジューラを実装しマルチタスクを実現してみます。

この章で実装するスケジューラはざっくりと以下のようなものとします。

- 各タスクに割り当てられた CPU 時間 (タイムスライス) を使い切ると強制的に別のタスクへ切り替える (プリエンプティブスケジューラ)
- タスクの数は全体で 2 つ (動作中のタスクと待ちタスク一つずつ)

実行中のタスク (コンテキスト) を別のタスクへ切り替える (コンテキストスイッチ) の方法やタスクの構造などの細かい実装については後ほど説明します。

また、一般的にはコンテキストスイッチの前に、次に実行するタスクを決める「ディスパッチ」の処理があります。今回のスケジューラの場合、タスク数は全体で 2 つなので、ディスパッチ処理は「今実行していない方のタスクを選択する」だけです。

3.2 実装: sched.c を用意

まず、スケジューラを実装していく「sched.c」(そして「sched.h」) を用意します。

ここではまず、「sched_init」という関数を用意して、5ms 周期の周期タイマーをセットアップし、「sched_start」関数で周期タイマーを開始させるようにしておきます (リス

ト 3.1)。

▼リスト 3.1　030_sched/sched.c

```
/* 追加（ここから） */
#include <hpet.h>
#include <pic.h>
#include <fbcon.h>

#define SCHED_PERIOD   (5 * MS_TO_US)

void schedule(void)
{
        set_pic_eoi(HPET_INTR_NO);
        putc('.');
}

void sched_init(void)
{
        /* 5ms 周期の周期タイマー設定 */
        ptimer_setup(SCHED_PERIOD, schedule);
}

void sched_start(void)
{
        /* 周期タイマースタート */
        ptimer_start();
}
/* 追加（ここまで） */
```

5ms 周期で実行されるハンドラ関数の名前は「schedule」としておきます。今後、この schedule 関数でコンテキストスイッチを行うようにしていきます。そのため、今回実装するスケジューラのタイムスライスは 5ms です。

ここまでで動作確認のため、main.c へ sched_init 関数と sched_start 関数呼び出しを追加します (リスト 3.2)。

▼リスト 3.2　030_sched/main.c

```
/* ・・・ 省略 ・・・ */
#include <hpet.h>
#include <sched.h>      /* 追加 */
#include <common.h>

/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */

        /* ファイルシステムの初期化 */
        fs_init(_fs_start);
```

```
        /* スケジューラの初期化 */        /* 追加 */
        sched_init();                    /* 追加 */

        /* CPU の割り込み有効化 */
        enable_cpu_intr();

        /* スケジューラの開始 */ /* 追加 */
        sched_start();                   /* 追加 */

        /* halt して待つ */
        while (1)
                cpu_halt();
}
```

sched.h を作成して sched_init 関数と sched_start 関数のプロトタイプ宣言を行い (リスト 3.3)、Makefile の OBJS へ sched.o を追加すると (リスト 3.4)、make、実行できます。

▼**リスト** 3.3　030_sched/include/sched.h

```
/* 追加（ここから）*/
#ifndef _SCHED_H_
#define _SCHED_H_

void sched_init(void);
void sched_start(void);

#endif
/* 追加（ここまで）*/
```

▼**リスト** 3.4　030_sched/Makefile

```
TARGET = kernel.bin
CFLAGS = -Wall -Wextra -nostdinc -nostdlib -fno-builtin -fno-common -Iinclude
LDFLAGS = -Map kernel.map -s -x -T kernel.ld
OBJS = main.o iv.o fbcon.o fb.o font.o kbc.o x86.o intr.o pic.o \
        sched.o hpet.o acpi.o handler.o fs.o common.o   # sched.o を追加

# ・・・ 省略 ・・・
```

実行すると 5ms 周期で'.' が描画されることを確認できます。

3.3　コンテキストスイッチの流れ

まず、コンテキストスイッチでやらなければならないことはざっくりと以下の 3 つです。

1. 今実行中のタスクの情報をどこかに退避する
2. 次に実行するタスクの情報を復帰する
3. 復帰したタスクを再開する

ある地点から実行を再開できるようにするために退避しておかなければならない情報は、割り込みの時と同じで、CPU が扱う各種レジスタの値です。それらの情報をスタックへ push することで退避しておきます。この時、次に実行する命令のアドレス (戻り先アドレス) も push しておきます。そして、push した後のスタックポインタをこれまで実行していたタスクに紐づく情報として別途用意するタスク毎の構造へ保存しておきます。

次に実行するタスクを復帰する際は、タスク構造へ保存しておいたスタックポインタをロードし、各種レジスタの値をスタックから pop するだけです。

最後に、スタックに push しておいた実行再開のアドレスを pop し、そのアドレスで実行を再開するとコンテキストスイッチ完了です。

3.4 これから実装すること

本書では HPET の周期タイマーから iretq するタイミングでコンテキストスイッチします。具体的な流れは以下の通りです。

1. 割り込み発生
2. CPU: フラグレジスタ (RFLAGS) と戻り先アドレス (CS と RIP) をスタックに積む
3. CPU: IDT に定義されたハンドラへジャンプ
4. kernel: 汎用レジスタをスタックに積む
5. kernel: 現在のスタックポインタ (RSP) を現在のタスク構造へ保存
6. kernel: 次のタスクの構造に保存されているスタックポインタを復帰
7. kernel: 汎用レジスタを pop
8. kernel: iretq 命令を実行
9. CPU: フラグレジスタを復帰し、戻り先アドレスを復帰

3.5 実装: コンテキストスイッチ

まず、「4. kernel: 汎用レジスタをスタックに積む」をですが、これは handler.s の HPET の割り込みハンドラ (hpet_handler) で実装済みです。

次に「5. kernel: 現在のスタックポインタ (RSP) を現在のタスク構造へ保存」について、前章で HPET の割り込みハンドラを追加する際、割り込みハンドラ hpet_handler

の入り口で全ての汎用レジスタを積み終わった後のスタックポインタを後続のハンドラ関数へ引数として渡すようにしていました。

ptimer_setup 関数等に与えるユーザー側のハンドラでも引数でそのスタックポインタを受け取れるようにしているため、スケジューラ側で周期割り込み時に呼び出される関数を用意し、その関数内でスタックポインタをタスクの構造へ保存するようにすれば良いです。

本書で扱う範囲において、タスクとして保持しておくべき情報はスタックポインタ以外に無いため、タスク構造はスタックポインタを格納する単なる配列とします。

以上を踏まえてスタックポインタ保存を実装するとリスト 3.5 の通りです。

▼**リスト** 3.5　030_sched/sched.c

```
#include <hpet.h>
#include <pic.h>
#include <fbcon.h>

#define SCHED_PERIOD   (5 * MS_TO_US)
/* 追加（ここから）*/
#define NUM_TASKS              2

unsigned long long task_sp[NUM_TASKS];
volatile unsigned int current_task = 0;
/* 追加（ここまで）*/

/* 変更（ここから）*/
void schedule(unsigned long long current_rsp)
{
        task_sp[current_task] = current_rsp;
        current_task = (current_task + 1) % NUM_TASKS;
        set_pic_eoi(HPET_INTR_NO);
}
/* 変更（ここまで）*/

void sched_init(void)
{
        /* 5ms 周期の周期タイマー設定 */
        ptimer_setup(SCHED_PERIOD, schedule);
}

/* ・・・ 省略 ・・・ */
```

配列 task_sp が、タスク毎のスタックポインタを格納する配列です。

また、タスク番号を指すグローバル変数 current_task の値を次に実行するタスク番号を指すように更新しています (ディスパッチ)。

続いて「6. kernel: 次のタスクの構造に保存されているスタックポインタを復帰」を実装します。

先ほど更新した current_task を使って task_sp[current_task] のようにすることで、次に実行するタスクのスタックポインタを取得できます。ただし、今はまだ task_sp[1]

に初期値を設定していないのでどんな値が復帰されるかは不定です。(タスク番号 1、すなわち今回新たに追加するタスクの初期設定は後ほどやります。)

スタックポインタ復帰はインラインアセンブラで実装します (リスト 3.6)。

▼**リスト** 3.6　030_sched/sched.c

```
/* ・・・ 省略 ・・・ */

void schedule(unsigned long long current_rsp)
{
        task_sp[current_task] = current_rsp;
        current_task = (current_task + 1) % NUM_TASKS;
        set_pic_eoi(HPET_INTR_NO);
        asm volatile ("mov %[sp], %%rsp"     /* 追加 */
                      :: [sp]"a"(task_sp[current_task]));       /* 追加 */
}

/* ・・・ 省略 ・・・ */
```

task_sp[current_task] を RSP レジスタへ格納しているだけです。

あとは、「7. kernel: 汎用レジスタを pop」と「8. kernel: iretq 命令を実行」です。

handler.s の hpet_handler のスタックポインタを退避していたので、その地点から iretq までを再現します。汎用レジスタを pop して、iretq します (リスト 3.7)。

▼**リスト** 3.7　030_sched/sched.c

```
/* ・・・ 省略 ・・・ */

void schedule(unsigned long long current_rsp)
{
        task_sp[current_task] = current_rsp;
        current_task = (current_task + 1) % NUM_TASKS;
        set_pic_eoi(HPET_INTR_NO);
        asm volatile ("mov %[sp], %%rsp"
                      :: [sp]"a"(task_sp[current_task]));
        /* 追加（ここから）*/
        asm volatile (
                "pop    %rdi\n"
                "pop    %rsi\n"
                "pop    %rbp\n"
                "pop    %rbx\n"
                "pop    %rdx\n"
                "pop    %rcx\n"
                "pop    %rax\n"
                "iretq\n");
        /* 追加（ここまで）*/
}

/* ・・・ 省略 ・・・ */
```

ここまでで、コンテキストスイッチの流れは実装完了です。

3.6 実装: 実験用のタスクを定義

今回のサンプルでは、「'A' を出力し続けるタスク (タスク A)」と「'B' を出力し続けるタスク (タスク B)」の 2 つで実験してみることにします。

実験程度なので、簡単のため、タスク A は main.c に (030_task_a)、タスク B は sched.c に (030_task_b) 実装してしまうことにします。

▼**リスト** 3.8　030_sched/main.c

```
/* ・・・ 省略 ・・・ */

void do_taskA(void);    /* 追加 */

void start_kernel(void *_t __attribute__((unused)), struct platform_info *pi,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */

        /* スケジューラの初期化 */
        sched_start();

        /* タスク A の開始 */   /* 追加 */
        do_taskA();             /* 追加 */

        /* halt して待つ */
        while (1)
                cpu_halt();
}

/* 追加（ここから） */
void do_taskA(void)
{
        while (1) {
                putc('A');
                volatile unsigned long long wait = 10000000;
                while (wait--);
        }
}
/* 追加（ここまで） */
```

▼**リスト** 3.9　030_sched/sched.c

```
/* ・・・ 省略 ・・・ */

#define SCHED_PERIOD   (5 * MS_TO_US)
#define NUM_TASKS              2
#define TASK_B_STASK_BYTES     4096    /* 追加 */

unsigned long long task_sp[2];
```

```
volatile unsigned int current_task = 0;
/* 追加（ここから）*/
unsigned char taskB_stack[TASK_B_STASK_BYTES];

void do_taskB(void)
{
        while (1) {
                putc('B');
                unsigned long long wait = 1000000;
                while (wait--);
        }
}
/* 追加（ここまで）*/

/* ・・・ 省略 ・・・ */
```

リスト 3.9 では、タスク B のスタックの確保も行っていて、実装方法としてはグローバル配列の定義で代用しています。(メモリを動的に確保する機能がまだ無いので。)

また、do_taskA 関数でもそうでしたが、do_taskB 関数で sleep を使用していないのは、sleep が複数のコンテキストから呼ばれることに対応していないためです[*1]。

3.7 実装: タスク B のスタックの初期化

最後に、タスク B のスタックを初期化すれば終わりです。

新たに生成されたタスクのスタックには、hpet_handler で汎用レジスタをスタックに積み終わった直後の状態を再現できれば、既に動いているタスクと整合性がとれます。

結論としてはリスト 3.10 のようにスタックを積んでおけば良いです。

▼**リスト** 3.10　030_sched/sched.c

```
/* ・・・ 省略 ・・・ */

void sched_init(void)
{
        /* 5ms 周期の周期タイマー設定 */
        ptimer_setup(SCHED_PERIOD, schedule);

        /* 追加（ここから）*/
        /* 予め TaskB のスタックを適切に積んでおき、スタックポインタを揃える */
        unsigned long long *sp =
                (unsigned long long *)((unsigned char *)taskB_stack
                                       + TASK_B_STASK_BYTES);
        unsigned long long old_sp = (unsigned long long)sp;
```

[*1] while で待っている箇所を sleep へ置き換えても動きますが、引数に与える時間に関わらず固定の時間 (体感的には一瞬) しか待機してくれないです。(おそらくロックの問題だと思います。) ただ、今回の場合は「文字が出すぎない様に」のためのものなので現状の sleep でも問題ありません。

```
        /* push SS */
        --sp;
        *sp = 0x10;

        /* push old RSP */
        --sp;
        *sp = old_sp;

        /* push RFLAGS */
        --sp;
        *sp = 0x202;

        /* push CS */
        --sp;
        *sp = 8;

        /* push RIP */
        --sp;
        *sp = (unsigned long long)do_taskB;

        /* push GR */
        unsigned char i;
        for (i = 0; i < 7; i++) {
                --sp;
                *sp = 0;
        }

        task_sp[1] = (unsigned long long)sp;
        /* 追加（ここまで） */
}

/* ・・・ 省略 ・・・ */
```

HPET の割り込みハンドラ (hpet_handler) にて、割り込みハンドラ入り口で汎用レジスタをスタックへ積んだ直後の RSP をコンテキストスイッチ時にタスクの構造へ保存しておくようにしていました。それと整合性を取るようにするには、「割り込み発生直後、スタックに汎用レジスタを積んだ状態」を再現すれば良いです。

汎用レジスタをどう積むかは、自身で割り込みハンドラに書いたものと同じように積めば良いです。

問題は、割り込みハンドラ呼び出し直後のスタックの状態がどうなっているか、ですが、これは CPU のデータシートで決められているもので、その通りにスタックを積むとリスト 3.10 のようになります。

3.8 動作確認

以上でソースコードの追加・変更は完了です。

ビルドし実行すると、図 3.1 のように、スケジューラにより'A' と'B' が出力されます。

```
BAABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABAB
AABBAABABBAABBABABABABABABABAABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABA
ABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABAB
ABBABABABABABABABABABABABABABABABABAABABABA
BABBABABAABABBABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABBA
```

▲図 3.1　030_sched の実行結果

おわりに

本書をお読みいただきありがとうございました！

本書では、前著で作ったカーネルの枠組みにスケジューラを追加しました。これにより、カーネルの技術要素として以下が (簡単なものではありますが、) 揃いました。

- デバイスドライバ
- ファイルシステム
- スケジューラ ← New

とはいえスケジューラは、動作するタスクが固定であったり、スケジューラ動作時に使えない機能 (sleep) があったりするのですが[*2]。

本著を読んで実際に作ってみた場合は、これらを直すも良し、これらは仕様としてアプリを作ってみるも良しで楽しんでみると良いと思います。

今後のネタとしては、カーネルとアプリをバイナリレベルで分離できるよう、システムコールを整備していきたいなと思っています。

あと、まだマウスをサポートしてないですね。特に自身のメイン機である ThinkPad のトラックポイントとかタッチパッド入力の取得方法を知りたい所です。タッチパッドはマルチタッチが扱えれば何か面白いことができそうです。

他にも ACPI でちゃんとシャットダウンできるようにするとか、ネットワーク対応とか、やりたいことは山積みです。

表紙について

ページ数の都合上、今回は「付録」での説明はしませんが、例によって表紙はサンプルプログラムを改造して作ったものです。

サンプルディレクトリ「A01_cover」にソースコードを格納していますので、興味があれば見てみてください。

[*2] スケジューラ動作時に意図通りに動かない問題は、共有資源をロックをしていないからかと思いますが。

参考情報

参考にさせてもらった情報

- Advanced Configuration and Power Interface Specification
  - `http://www.uefi.org/specifications`
  - ACPI の仕様を uefi.org で取りまとめているのか、`http://www.acpi.info/` より新しいバージョンの仕様書が uefi.org のページに置いてあるため、ACPI の仕様書は `http://www.uefi.org/specifications` を参照するようにしています
- IA-PC HPET (High Precision Event Timers) Specification
  - `http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf`
  - Intel 公式の HPET のデータシートです
- Intel 64 and IA-32 Architectures Software Developer Manuals
  - `https://software.intel.com/en-us/articles/intel-sdm`
  - Intel 公式の x86 CPU のソフトウェア開発マニュアルです
- 書籍「パソコンのレガシィ I/O 活用大全」(CQ 出版)
  - KBC や PIC 等、PC/AT 互換機で標準的 (ただし今や古い) な IC の使い方についてはほとんどこの本を参考にさせていただきました

本シリーズの過去の著作

サークル「へにゃぺんて」の同人誌は、「フルスクラッチで作る!」シリーズ含め、すべて以下の URL から PDF 版/HTML 版は無料でダウンロード/閲覧できます

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

「フルスクラッチで作る!」シリーズの著作としては以下があります。

- 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 1、パート 2
  - ブートローダーを作る上で必要な UEFI を直接叩く方法を紹介
- フルスクラッチで作る!x86_64 自作 OS
  - 本著の前著に当たる「x86_64 自作 OS」のパート 1(無印)
  - ハードウェアを抽象化しアプリへ提供するカーネルの基本的な枠組みを作り上げる
  - カーネルの機能を利用するアプリとして「画像ビューア」を作る

フルスクラッチで作る!x86__64 自作 OS パート 2 ACPI で HPET 取得してスケジューラを作る本

2018 年 8 月 10 日　コミックマーケット 94 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

(C) 2018 へにゃぺんて