

これにより、コンピュータの操作性が格段に向上した。^ズ1970年代から1980年代にかけて、ほとんどのコンピュータにはVDT技術と入力用の電子キーボードが搭載されていた。その後、VDTに代わってCRTやLCDが採用されるようになり、電子キーボードも汎用コンピュータの標準となった。

今日、私たちはコンピュータを使うたびにキーボードを使っています。キーボードのレイアウトのほとんどはタイプライターのものに残っており、使い方も同じです。しかし、電子機器の新しい時代のおかげで、キーボードにはさまざまな形があります。一般的なプラスチック製のキーボード、折り畳み式のキーボード、バックライト付きのキーボード、さらにはレーザーキーボードまで。

キーボードレイアウト

一般的なキーボードレイアウトは、QWERTYという文字が最初の5文字であることから、**QWERTYキーボード**と呼ばれています。QWERTY配列は、タイプライターの時代に、初期のタイプライターの機械的な限界から、タイピストのタイピング速度を遅くするために意図的に設計されたものです。これは主に、各キープレスの間の時間を短縮し、プリントヘッドに十分な時間を与えてジャムを起こさないようにするためである。

QWERTY配列は、現在もすべてのキーボードに採用されています。

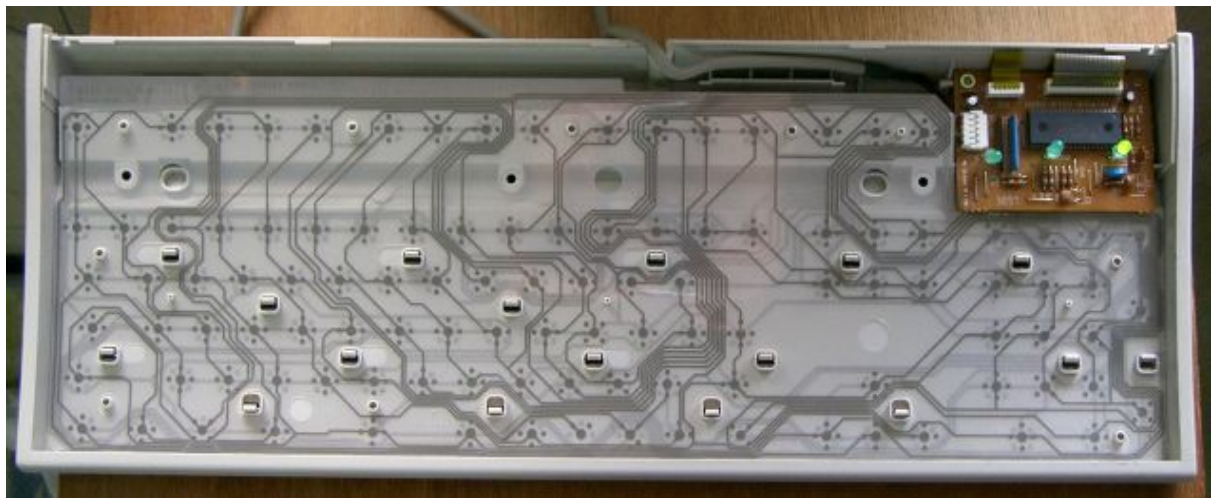
キーボード内部

キーボードのキーを押すと、実際に何が起こるのでしょうか？キーボードは、どのキーが押されているかをどうやってプログラムに伝えるのでしょうか？今、読まれている文章は、キーボードで入力されたものです。キーボードはどのようにしてこれを行うのでしょうか？見てみましょう。

注：具体的な内容は、キーボードの種類やモデルによって異なります。そのため、ここでは一般的な**102キー**のキーボードのみを取り上げています。

ケースを開く

キーボードが複雑なプリント基板から、マイクロプロセッサを搭載した一体型の基板になったことに驚かれるかもしれません。キーボードを開いてみると、このようになっています。



そう、それだ。いかにシンプルであるかがわかります。1枚の回路基板とグリッドです。上の写真ではグリッドが少し見づらかもしれませんが、しかし、よく見ると、グリッドのポイントが見え、そのポイントが一般的なキーボードのキーポジションと一致していることに気づくかもしれません。これを「**キーマトリクス**」と呼びます。ほとんどのキーボードでは、キーマトリクスを構成する回路がグリッドの各ポイントの間に途切れています。キーマトリクスのあるポイントの上にキーがあることを知って、キーを押すと、そのポイントにあるスイッチが押されて横の回路が完成し、電流が流れるようになります。キーの機械的な動きによって生じる線の振動は**バウンス**と呼ばれ、**キーボードエンコーダー**として知られるキーボード自身の**マイクロプロセッサ**によってフィルタリングされます。少し複雑に感じられるかもしれませんが、ご安心ください。次の2つのセクションでは、すべてをより詳しく見ていきます。

キーボードエンコーダ

キーボードに搭載されているマイクロプロセッサは、インテル社の最初のマイクロコントローラーでもある、オリジナルの**インテル8048**が使用されている。このコントローラーは**キーボード・エンコーダー**と呼ばれています。キーボードエンコーダーの種類は、キーボードによって大きく異なります。キーボード・エンコーダーには何百種類もの種類がありますが、基本的にはすべて同じことをしています。

キーグリッド内の行と列は、キーボードエンコーダーの8ビットI/Oポートに接続されています。キーが押されると、キーグリッド内のその場所にあるスイッチが閉じ、電流が流れて回路が完成します。この電流は、キーの位置に対応するポートのキーボードエンコーダのピンを有効にします。このように、コントローラはポートをスキャンするだけで、キーが押されているかどうかをポートラインがアクティブかどうかで確認することができます。

キーが押されていると、キーボードのエンコーダは**ROM (Read Only Memory)** の文字マップからその文字の**スキャンコード**を調べ、内部の**16バイト**のメモリに保存する。キーボードのプロセッサには、独自のタイマーと**33**の命令セットがあり、**128K**の外部メモリにアクセスすることもできる。タイマーを使って、キーが押されたかどうかを、ユーザーの入力か**バウンス**かで判断します。バウンドした場合は、通常、人間が入力できるよりもはるかに速い速度になります。タイマーが**0**になってもキーが押されたままであれば、リセットされ、文字が内部の**16バイト**のバッファに挿入される。

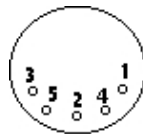
ここで重要なのは、**通信可能なキーボードコントローラが2つある**ということです。キーボードの内部にある**キーボードエンコーダ**と、**マザーボード上にあるキーボードコントローラ**です。もう1つのコントローラについては後ほどご紹介します。)今のところ、2つのコントローラがあり、**キーボードエンコーダはそのうちの1つであることを覚えておいてください**。

キーボードエンコーダは、**キーボードプロトコル**で定義された方法でシステムと通信します。その内容を見てみましょう。

キーボードプロトコル

キーボードエンコーダは、データをバイトとしてマザーボードのオンボードキーボードコントローラに送信します。データの送信方法は、キーボードのインターフェースで使用されている**プロトコル**によって異なります。これは通常、**5ピンDIN**コネクタ、**6ピンMini-DIN**コネクタ、**USB**コネクタ、**SDL**コネクタ、または赤外線 (**IR**) インターフェースを使用したワイヤレスです。

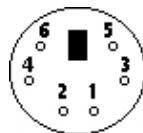
AT/XTキーボードに使用する**5ピン**の**DIN**コネクタは、通常、コンピュータの背面にあり、次のような形をしています。



1: クロック 2: データ 3: 該当なし 4: グランド 5: Vcc (+5V)

マザーボードは、**Vcc**端子と**Ground**端子を介して**PSU (Power Supply Unit)** から電源を供給します。クロック端子は、キーボードのデータとシステムクロックとの同期に使用されます。キーボードからのデータは、データピンを介してシリアルデータとして送信されます。

PS/2キーボードに使用されている**6ピン**の**Mini-DIN**コネクタは非常によく似ています。



1: データ 2:N/A 3: グランド 4: Vcc (+5V) 5: クロック 6: N/A

ここでは特に新しいことはありません。DINは特に何かを表しているわけではなく、これを開発した標準化団体 (**Deutsches Institut für Normung**、英語では**German Institute for Standardization**) を意味しています。

SDL (Shielded Data Link) コネクタは非常によく似ています。



A: N/A B: データ C: グランド D: クロック E: Vcc (+5V) F: N/A

ユニバーサル・シリアル・バス (USB) コネクタは、さまざまな機器で使用されている規格です。**USB**デバイスを直接扱うことは、かなり複雑なテーマです。**USB**デバイスには**4つのピン**しかありません。1: **Vcc (+5V)**、2: **Data-**、3: **Data+**、4: **Ground**。

USBレガシーサポートは、**USB**ポートを備えた最近のほとんどのコンピュータで採用されています。これは、これらのコンピュータのマザーボードが、**USB**キーボードやマウスを**PS/2**キーボードやマウスとしてエミュレートできることを意味します。このため、**PS/2互換のインターフェースを使用したUSBキーボードやマウスとの通信は機能します**。言い換えれば、私たちのように**USB**キーボードやマウスを持っていても心配する必要はありません。このチュートリアルコードとデモは、マザーボードが提供するエミュレーションのおかげで問題なく動作します。

ご覧のように、キーボードとコンピュータの間のインターフェースはそれほど複雑ではありません。キーボードコントローラとキーボードエンコーダの間で、データをビットとして送信する方法を提供しているだけです。そのデータは

マザーボード上のオンボードまたは一体型のキーボードコントローラーにルーティングされます。キーボードコントローラーが制御します。

キーボードコントローラー

システムケース内で使用されているキーボードコントローラーは、通常、オリジナルの**8042**キーボードコントローラーの形をしています。キーボードコントローラーは、キーボードのプロトコルを介してキーボードエンコーダーとのインターフェースを提供します。ほとんどの新しいシステムでは、キーボードコントローラーは独立した**IC**ではなく、**フロッピーディスクコントローラー (FDC)**、**パラレルポートインターフェース**、**シリアルポートインターフェース**、**マウスインターフェース**を含む、マザーボードの**スーパー入出力 (IO) コントローラー**の一部となっています。最近のシステムでは、**スーパーIOコントローラー**は、マザーボードのサウスブリッジにある**ISA (Industry Standard Architecture)**ではなく、**LPC (Low Pin Count)**バスを使用しています。

スキャンコード

スキャンコードとは、キーの状態を表すデータパケットのこと。キーが押されたり、離されたり、押し続けられたりすると、**スキャンコード**がコンピューターの**オンボードキーボードコントローラー**に送られる。スキャンコードには**2種類**あります。**メイクコード**と**ブレイクコード**である。**メイクコード**はキーが押されたときに送信され、**ブレイクコード**はキーが離されたときに送信されます。メイクコードとブレイクコードは、キーボードの各キーに固有のものです。すべてのスキャンコードを表す数字のセットがキーボードの**スキャンコードセット**です。

キーボードが使用できるスキャンセットは通常**3種類**あります。しかし、スキャン値はランダムなので、どのスキャンセットを使用しているかを簡単に判断する方法はありません。そのため、ルックアップテーブルを使って、スキャンコードが示すキーを決定する必要があります。

それでは、スキャンコード表を見てみましょう。**注意：これらの表は重要です。これらのテーブルは重要です！**キーボードのどのキーが押されたかを判断するのに必要です。また、これらの表に記載されているスキャンコードはすべて**16進法**です。

これらの表はかなり大きいので、別のリソースとして置くことにしました。表は[こちら](#)のリソースセクションをご覧ください。

例を挙げてみましょう。キーボードの**shift+A**キーを押すと、コンピューターに送られるメイクコードはどのようなのでしょうか？これを理解するために、一連の流れを見てみましょう。まず、シフトキーが押され、次に**A**キーが押されます。次に**A**キーを離し、続いてシフトキーを離します。スキャンコードセットが最近のキーボードのデフォルトのスキャンコードセットであると仮定すると、左シフトキーのメイクコードは**0x12**、ブレイクコードは**0xF0**と**0x12**です。また、**A**キーのメイクコードは**0x1C**、ブレイクコードは**0xF0**と**0x1C**です。したがって、このイベントが発生すると、以下のスキャンコードがコンピュータに送信されます。

```

キーイベント：シフトダウン Aダウン Aリリース シフトリリース スキャ
ンコード：          0x120x1C0xF0 0x1C0xF0
0x12
  
```

上記を見ると、送信されるスキャンコードは、**0x12**、**0x1C**、**0xF0**、**0x1C**、**0xF0**、**0x12**となることがわかります。

キーを押したままにしておくと、そのキーは**タイプコード**になります。つまり、キーを離すか別のキーを押すまで、キーボードはキーのコードを送り続けるのです。試してみてください。お気に入りのテキストエディターを開き、あるキーを押し続けます。しばらくすると、同じ文字が現れ、その後、その文字が連続して表示されます。**タイプディレイ**は、タイプモードに入るまでの待ち時間を決定し、**タイプレート**は、コンピュータに送信する**1秒間**の文字作成コードの量を決定します。タイプマチックモードでは、文字データはバッファリングされません。複数のキーを押し続けた場合、最後に押したキーだけがタイプマチックになります。

スキャンコードは、私たちにとって非常に重要なものです。スキャンコードがオンボードのキーボードコントローラーに送信されると、キーボードコントローラーはスキャンコードを内部メモリーに格納します。その後、キーボードコントローラーは、**割り込み要求 (IR)** ラインをハイに切り替えます。割り込みラインが**PIC (Programmable Interrupt Controller)** によってマスクされていない場合は、これによって**IRQ 1**が発生します。**IRQ**がマスクされていても、リードバッファはソフトウェアで読み取ることができるので、スキャンコードを読み取って、どのキーが離されたのか、押されたのかを判断することができます。

キーボードインターフェース。デバイスドライバの開発

この章では、すでに多くのことを取り上げてきました。インターフェース機器としてのキーボードの歴史、**QWERTY**キーボードのレイアウト、キーボードの内部を見て、その仕組みと主要部品を確認しました。また、スキャンコードセットやキーボードのプロトコルについても見てきました。まだすべてを理解していなくても心配いりません。次のいくつかのセクションでさらに詳しく説明します。また、キーボードのデバイスドライバーも開発する予定です。すごいでしょ？このセクションのすべてのコードは、最終的なデモにも使用されます。

キーボード・インターフェーシングポーリング^ズ

前のセクションで、キーボードを操作する際には**2つの**コントローラーがあることを覚えていますか？つまり、キーボード内部の**キーボードエンコーダー**と、マザーボード上の**キーボードコントローラー**です。この章では、1つのハードウェアデバイスを制御するために、複数の異なるコントローラとのインターフェースが必要になります。そうですね。これらのコントローラの**両方**と通信することができるのです。まあ、そんな感じです。キーボードエンコーダーにコマンドを送信すると、オンボードのキーボードコントローラーにもコマンドが送信されますが、キーボードプロトコルを介してキーボードエンコーダーに再ルーティングされます。

よし、これで両方のコントローラーと通信できるぞ。これは楽しいですね。両方のコントローラーが相互に機能しているということは、相互に通信もしているということです。キーボードエンコーダーは、さまざまなコードをオンボードのキーボードコントローラーに送信して保存します。これはスキャンコードであったり、エラーコードであったりします。これにより、キーボード・エンコーダーとオンボード・コントローラーの両方から情報を受け取ることができます。

これらの通信は、**IO**アドレス空間にマッピングされたコントローラのポートに対して、**IN**命令と**OUT**命令を使って読み書きするだけで行われます。これらのポートが何であるかを気にする必要はありませんでしたが、**IO**マッピングがコントローラでどのように機能するかを理解することは、ここでより重要になります。

これは、キーボードとのインターフェースのひとつです。キーが押されているか、押されていないかなどを確認するために、コントローラと手動で通信することができます。これを「キーボードの**ポーリング**」と呼びます。このようにして、キーボードコントローラにポーリングすることで、キーボードから最後のスキャンコードを取得することができます。

キーボードのインターフェイス。割り込み要求(**IRQ**)

PICのチュートリアルで、キーボードコントローラが割り込み線を使用するように設定できることを覚えていますか？キーが押されたり離されたりしたときに、キーボードコントローラが**IRQ 1**を発行するように設定することができます。これは、キーボードとの最も一般的なインターフェース方法です。

IRQ1が起動したときは、必ずキーボードコントローラにスキャンコードが実際に送られたかどうかをテストする必要があります。これは、キーボードコントローラをポーリングして最後のスキャンコードを取得することで行います。

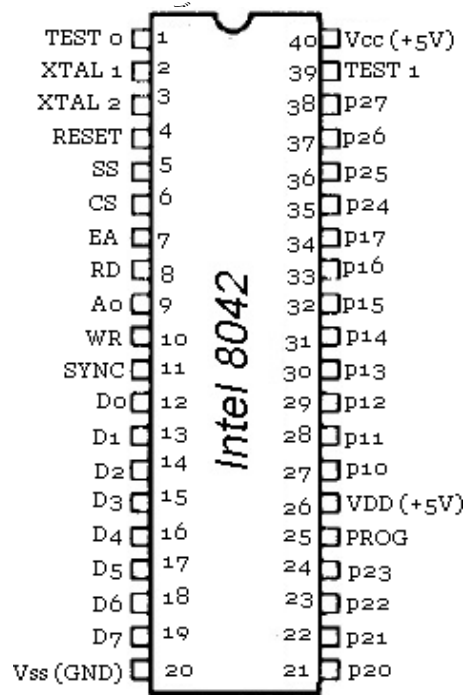
詳細**8042**キーボードマイクロコントローラ



オリジナルの**8042**マイクロコントローラ

キーボードエンコーダーとのインターフェースを担うマイクロコントローラーです。キーボードコントローラーは、**8042**マイクロコントローラーから始まったマイクロコントローラーファミリーの一部です。最近のコンピュータでは、キーボードコントローラーは独立した**集積回路 (IC)**ではなく、その機能はマザーボード自体にエミュレートされています。つまり、コントローラーの機能は、マザーボードのチップセットに組み込まれているのです。

キーボードコントローラーは、**2つの**モードで動作します。**ATコンパティブルモード**と「**PS/2コンパティブルモード**」です。コントローラがどちらのモードで動作するかによって、外界とのインターフェース方法が異なります。まず、コントローラを見てみましょう。



うん。その通りです。**P10-P17**ピンがコントローラーの入力ポート。ピン**P20-P27**は、コントローラの出力ポートです。そして、ピン**T0-T1**はコントローラのテストポートです。これらのピンの正確な意味は、コントローラの動作モードによって異なります。

これらのポートを操作するためのコマンドがありますので、後で詳しく見てみましょう。

他のピンのほとんどは、私たちにとって重要ではありません。ここに追加したのは、あくまでも補足のためであり、あなたがそれらを知る必要はありません。

XTAL 1と**XTAL 2**は、水晶振動子の入力端子です。CLKが外部から駆動されている場合、**XTAL 1**はグラウンドに接続することもできます。同様に、CLKが外部から駆動されている場合は、**XTAL 2**をCLKに接続することができます。

RESETはLow(0)にするとコントローラをリセットします。

SSは、マイクロコントローラのシングルステップ端子です。**CS**は、データレジスタポートのインターフェースに使用されるチップセレクト端子です。**EA (No, not the company ;)**は外部アクセス入力端子です。**OTP (ワンタイム・プログラマブル) ROM**をディセーブルにして、外部からコントローラにコマンドを送れるようにします。

RD 出力イネーブル入力：データレジスタポートのインターフェースに使用されます。**A0**はコマンド/データ・レジスタ・セレクト入力です。データ・レジスタ・ポートのインターフェーシングに使用します。**WR**はライトイネーブル入力ラインで、データレジスタポートのインターフェーシングに使用されます。

SYNCはクロック出力信号です。**D0~D7**はデータレジスタポートのインターフェースに使用します。**GND**はグラウンド端子(Vss)です。**Vdd**は、+5V入力端子である。**PROG**は、I/Oエクステンダアクセス時に8243へのアドレス/データストロープとして使用される。**Vcc**はもう一つの+5V入力端子である。

キーボードコントローラは、キーボードの動作を制御するためのインターフェースを提供します。これは、ポートI/O空間にマッピングされたポートを介して、キーボードコントローラと通信することで行います。ご存知のように、キーボードコントローラと通信するためには、INとOUTの命令を使い、そのマッピング方法を知る必要があります。では、見てみましょう。

ポートマッピング

i86アーキテクチャでは、キーボードとの通信に以下のポートが使用されています。

キーボード・コントローラー・ポート		
ポート	リード/ライト	ディスクリプション
キーボードエンコーダ		
0x60	リード	リード入力バッファ
0x60	書く	送信コマンド

オンボード・ マ ターボード・コントローラ		
0x64	リード	ステータスレジスタ
0x64	書く	送信コマンド

このテーブルは悪くないと思いますよ ;)基本的には**キーボードエンコーダにコマンドを送るには、ポート0x60にコマンドバイトを書き込みます**。しかし、これを行う前に、安全のためにキーボードコントローラのステータスレジスタのビット0（出力バッファフル）が0であることを確認する必要があります。もし、キーボードコントローラのステータスレジスタのビット1（入力バッファフル）が1であれば、データは入力バッファに入っていて、読める状態になっています。**ポート0x60から読み出すと、キーボードエンコーダからこのデータを取得することができます**。キーボードエンコーダから読み込んだデータは、通常はキーボードから得られます。しかし、マイクロコントローラを再プログラムして、特定の値を返すようにすることもできます。

ポート0x64に値を書き込むことで、オンボードのキーボードコントローラにコマンドバイトを送信することができます。**ポート0x64から読み出すと、キーボードコントローラのステータスバイトを取得することができます**。

これらのことを知っていれば、これらのコントローラとの間でコマンドバイトやデータを読み書きするルーチンを簡単に提供することができます。ここでは、これらのコントローラが使用するIOポートを抽象化しています。

```
enum KYBRD_ENCODER_IO
{
    kybrd_enc_input_buf    =    0x60,
    kybrd_enc_cmd_reg      =    0x60
};

enum KYBRD_CTRL_IO
{
    kybrd_ctrl_stats_reg   =    0x64,
    kybrd_ctrl_cmd_reg     =    0x64
};
```

ここでは、コントローラのコマンドに関する知識を必要とするため、これらのコントローラを操作するためのルーチンについては、まだ説明しません。

レジスター

ステータスレジスタ

これは、20番目のアドレスラインを有効にすることを説明したときに見覚えがあるかもしれません。ステータスレジスターを読み出すには、**I/Oポート0x64から読み出すだけです**。返される値は、特定のフォーマットに従った**8ビットの値**です。このフォーマットは、コントローラのモードに応じて少しずつ異なります。

ここにもう一度書いておきます。重要なものは**太字**にしました。

- **ビット0** : 出力バッファステータス
 - 0: 出力バッファが空、まだ読まないでください
 - **さい 1**: 出力バッファがいっぱい、読んでください ;)
- **ビット1** : 入力バッファステータス
 - 0: 入力バッファが空、書き込み可能
 - **1**: 入力バッファが満杯、まだ書き込まないでください
- **ビット2** : システムフラグ
 - 0 : パワーオンリセット後に設定
 - 1 : キーボードコントローラのセルフテスト (Basic Assurance Test, BAT) が正常に終了したときに設定されます。
- **ビット3** : コマンドデータ
 - 0: 入力バッファへの最後の書き込みはデータ (ポート0x60経由)
 - 1: 入力バッファへの最後の書き込みがコマンド (ポート0x64経由) だった場合
- **Bit 4**: キーボードロック
 - 0: ロックされている
 - 1: ロックされていない
- **ビット5** : Auxiliary Output
 - buffer full PS/2 Systems:
 - 0 : ポート0x60からの読み出しが有効かどうかを判断 有効な場合、0=キーボードデータ
 - 1 : マウスデータ、ポート0x60からの読み出しが可能な場合のみ
 - ATシステム。
 - 0: OKフラグ
 - 1 : キーボードコントローラからキーボードへの送信がタイムアウトした。これは、キーボードが存在しないことを示している可能性があります。

- ビット**6** : タイムアウト
 - 0: OKフラグ
 - 1: タイムア
 - ウト PS/2
 - 一般的なタイムアウト
 - AT:

- キーボードからキーボードコントローラ^ズへの送信のタイムアウト。パリティエラーの可能性あり（この場合、ビット6と7の両方がセットされます）
- ビット7：パリティエラー
 - 0：OKフラグ、エラーなし
 - 1：最終バイトのパリティエラー

キーボードの現在の状態を判断し、何ができて何ができないのかを知るためには、ステータスレジスタを読む必要があります。例えば、キーボードが接続されていない状態で、キーボードにコマンドを送信したくはありません。そこで、コマンドを送信する前に、現在の状態を読み込んでテストしたいのです。

また、キーボードコントローラが反応するよりもプロセッサが命令を実行する方がはるかに速いということも考慮する必要があります。このため、キーボードコントローラが次のコマンドを実行できるようになるまで、待たなければならないことが多々あります。これを確認するには、ステータスレジスタを読み込んで、ビット0 (**Output Buffer Full**) をテストし、次のコマンドを送信してもよいかどうかを確認する必要があります。これを行わないと、前のコマンドは破棄され、新しいコマンドが実行を開始しますが、これは好ましくないことです。

別のコマンドを送信したり、データを読み出す前に、コントローラの準備が整うのを待つことが重要です。

ステータスレジスターの読み書きには、ビットマスクを使うことができます。この章の最後に掲載したデモのものです。各ビットが上のリストの正しいビットとどのように一致しているかに注目してください。

```
enum KYBRD_CTRL_STATS_MASK
{
    kybrd_ctrl_stats_mask_out_buf    =    1,           //00000001
    kybrd_ctrl_stats_mask_in_buf     =    2,           //00000010
    d ctrl stats mask system         =    4,           //00000100
    kybrd_ctrl_stats_mask_cmd_data   =    8,           //00001000
    kybrd_ctrl_stats_mask_locked     =    0x10,        //00010000
    d ctrl stats mask aux_buf        =    0x20,        //00100000
    kybrd_ctrl_stats_mask_timeout    =    0x40,        //01000000
    kybrd_ctrl_stats_mask_parity     =    0x80,        //10000000
};
```

すばらしいですね。というわけで、あとはポート**0x64**にあるキーボードコントローラのステータスレジスタを読み込めばいいわけです。そして、上のビットマスクに基づいて、好きなビットをテストして、そのステータスをチェックします。

つまり、キーボードコントローラのステータスレジスタから読み出すために必要なのは

読み取りと書き込み入力バッファ `read_status ()` { キーボードコントローラの状態を読み取る。

コマンドを送信するには、まず、キーボードコントローラがコマンドを受信できる状態になっていることを確認します。これは、入力バッファが `return_incomplete` (KBBD_CTRL_STATS_REG) になっているかどうかを確認することで行われます。これは、キーボードコントローラのステータスレジスタを読み、ビットをテストすることで行います。このビットが **0** であれば、バッファは空なので、コマンドバイトを送信します。(これらの情報はすべて、上に示したステータスレジスタのビットレイアウトの中にあることを覚えておいてください)

キーボードエンコーダーは、以下のよう非常によく似て受信します。キーボードエンコーダーに送られたコマンドは、まずキーボードコントローラーに送られることを覚えておいてください。このため、キーボードコントローラー自体がコマンドを受信できる状態になっていることを確認する必要があります。

```
//! kkybrdコントローラの入力バッファがクリアされるのを待つ
while (1)
    if ( (kybrd_ctrl_read_status () & KYBRD_CTRL_STATS_MASK_IN_BUF) ==
        0) break;

outportb (KYBRD_CTRL_CMD_REG, cmd)です。
```

```
// ! キーボードエンコーダバッファの読み込み
uint8_t kybrd_enc_read_buf () {

    return inportb (KYBRD_ENC_INPUT_BUF)となります。
}

// ! キーボードエンコーダへのコマンドバイトの送信
void kybrd_enc_send_cmd (uint8_t cmd) {

    // ! kkybrdコントローラの入力バッファがクリアされるのを待つ
    while (1)
        if ( (kybrd_ctrl_read_status () & KYBRD_CTRL_STATS_MASK_IN_BUF) == 0) break;

    // ! コマンドバイトをKybrdエンコーダに送信
    outportb (KYBRD_ENC_CMD_REG, cmd);
}
```

キーボード・エンコーダ・コマンド

ポート0x60にコマンドバイトを書き込むと、キーボードコントローラはその値をキーボードエンコーダに直接送信します。以下に、コマンドバイトの一覧を示します。

コマンド一覧	
コマンド	ディスクリプション
0xED	セットLED
0xEE	エコーコマンド。診断テストとして0xEEをポート0x60に返す
0xF0	オルタネイトスキャンコードの設定
0xF2	2バイトのキーボードIDコードを、ポート0x60から読み込まれる次の2バイトとして送信する
0xF3	オートリピートのディレイとリピート率の設定
0xF4	キーボードの有効化
0xF5	パワーオン状態にリセットし、イネーブルコマンドを待つ
0xF6	電源を入れた状態に戻し、キーボードのスキャンを開始する
0xF7	すべてのキーをオートリピートにする (PS/2のみ)
0xF8	すべてのキーにメイクコードとブレイクコードを送信するように設定 (PS/2のみ)
0xF9	すべてのキーでメイクコードのみを生成するように設定
0xFA	すべてのキーにオートリピートを設定し、make/breakコードを生成する
0xFB	単一のキーをオートリピートに設定する
0xFC	1つのキーを設定して、MakeとBreakのコードを生成する
0xFD	ブレイクコードのみを生成するキーを1つ設定
0xFE	最終結果の再送
0xFF	キーボードの電源をリセットし、セルフテストを開始する。

小さなコマンドはすべて上の表に記載されています。ここでは、より複雑なコマンドについて詳しく見ていきましょう。

Command 0xED - Set Light Emetting Diodes (LED's)

このコマンドは、キーボードのLEDを設定するために使用します。ポート0x60に書き込まれた次のバイトでキーボードのLEDが更新され、以下のようなフォーマットになります。

- ビット0 : スクロールロックLED (0 : オフ 1 : オン) ビット1 : ナムロックLED (0 : オフ 1 : オン) ビット2 : キャップスロックLED (0 : オフ 1 : オン)

その他のビットはすべて0でなければなりません。

す。パラメーターが真か偽かに基づいて、ビットをどのように設定またはクリアするかに注目してください。また、最初にコマンドバイトをキーボードエンコーダに書き込み、次にデータバイトを書き込んでいることにも注目してください。これらは両方ともキーボード・エンコーダーのコマンド・レジスタに送られます。KYBRD_ENC_CMD_SET_LEDは、0xED（使用するコマンド・バイト）の定数です。魔法は使いません。）

```
// !LEDをセット
void kkybrd_set_leds (bool num, bool caps, bool scroll)

{ uint8_t data = 0;

  //! ビットの設定またはクリア
  data = (scroll) ?(data | 1) : (data & 1);
  data = (num) ?(num | 2) : (num & 2);
  data = (caps) ?(num | 4) : (num & 4);

  コマンドの送信 -- キーボードの発光ダイオード (LED) の更新
  kybrd_enc_send_cmd (KYBRD_ENC_CMD_SET_LED);
  kybrd_enc_send_cmd (データ) です。
}
```

Command 0xF0 - Set alternatate scan code set (PS/2 Only)

このコマンドは、使用するスキャンコードセットを設定します。ポート0x60に書き込まれる次のバイトは、以下のフォーマットのバイトでなければなりません。

- **ビット0** : ポート0x60に設定されている現在のスキャンコードを返す
- **ビット1** : スキャンコードセ
- **ット1の設定** **ビット2** : スキ
- **ャンコードセット2の設定** **ビ**
- **ット3** : スキャンコードセッ
- **ト3の設定**

その他のビットはすべて0にしてください。

コマンド 0xF3 - オートリピートの遅延とリピート率の設定

このコマンドでは、オートリピートの遅延時間とリピートレートを設定します。ポート0x60に書き込まれる次のバイトは、以下のフォーマットでなければなりません。

- **ビット0~4** : リピートレート0 : 約30chars/sec~0x1F : 約2chars/sec
- **ビット5-6**リピートディレイ00 : 1/4秒、01 : 1/2秒、10 : 3/4秒、11 : 1秒

他のビットはすべて0にしてください。

リターンコード

ご存知のように、キーボードエンコーダはシステムのオンボードキーボードコントローラと通信します。返される値のほとんどはスキャンコードですが、時にはエラーを返すこともあります。これらの値は、キーボードデコーダからポート0x60を通じてシステムに送信されます。

戻り値は以下のいずれかです。

リターンコード	
値	ディスクリプション
0x0	内部バッファオーバーラン
0x1-0x58, 0x81-0xD8	キープレス・スキャンコード
0x83AB	F2コマンドから返されるキーボードIDコード
0xAA	リセット後のBAT (Basic Assurance Test) 中に戻ってきた。またL.シフトキーのメイクコード
0xEE	ECHOコマンドからの帰還
0xF0	一部のメーカーコードの接頭語 (PS/2には適用されません)
0xFA	キーボードアクノリッジからキーボードコマンド
0xFC	ベーシック・アシュアランス・テスト (BAT) の失敗 (PS/2のみ)
0xFD	対角線上の故障 (PS/2を除く)
0xFE	最後のコマンドを再送するようにシステムに要求するキーボード
0xFF	キーエラー (PS/2のみ)

オンボード・キーボード・コントローラーの コマンド

これらのコマンドの中には、「A20」の章ですでに紹介したものもあります。しかし、ここに挙げたコマンドの多くは新しいもので、中には非常にレベルの低いものもあります。つまり、これらのコマンドの中には、特定のラインをコントロールすることができるものもあります。

がコントローラに接続されています。そのため、コントローラのラインと、キーボードデバイスとのインターフェースを取り上げなければなりません。その他のコマンドでは、コントローラの内蔵RAMを読み書きすることができます。

コマンド一覧	
コマンド	ディスクリプション
共通コマンド	
0x20	読み取りコマンドバイト
0x60	ライトコマンドバイト
0xAA	セルフテスト
0xAB	インターフェーステスト
0xAD	キーボードの無効化
0xAE	キーボードの有効化
0xC0	リード入力ポート
0xD0	リード出力ポート
0xD1	書き込み出力ポート
0xE0	リードテスト入力
0xFE	システムリセット
0xA7	Mouseポートの無効化
0xA8	マウスポートの有効化
0xA9	テストマウスポート
0xD4	マウスへの書き込み
非標準コマンド	
0x00-0x1F	Read Controller RAM
0x20-0x3F	Read Controller RAM
0x40-0x5F	Write Controller RAM
0x60-0x7F	Write Controller RAM
0x90-0x93	Synaptics Multiplexer Prefix
0x90-0x9F	ライトポート13-ポート10
0xA0	著作権を読む
0xA1	ファームウェアバージョンの読み込み
0xA2	スピードを変える
0xA3	スピードを変える
0xA4	パスワードがインストールされているかどうか
0xA5	パスワードの読み込み
0xA6	パスワードの確認
0xAC	ディスグノシス・ダンプ
0xAF	キーボード版を読む
0xB0-0xB5	コントローララインのリセット
0xB8-0xBD	コントローララインの設定
0xC1	連続入力ポートポール、ロー
0xC2	連続入力ポートポール、ハイ
0xC8	コントローララインP22、P23のブロック解除
0xC9	ブロックコントローラライン P22、P23
0xCA	リードコントローラモード
0xCB	ライトコントローラモード
0xD2	ライト出力バッファ
0xD3	ライトマウス出力バッファ
0xDD	A20アドレスラインの無効化

0xDF	A20アドレスラインの有効化
0xF0-0xFF	パルス出力ビット

これだけの数のコマンドがあるんですね。ここですべてのコマンドをカバーするには、とても長い時間がかかると思いませんか？A20のコマンドについては、すでに「A20」の章で説明しています。このシリーズでは携帯性を重視していますので、上記のような一般的なコマンドのみを取り上げます。しかし、ここで紹介していないコマンドについては、興味のある読者の方を探していただきたいと思います。

次のセクションまでサンプルコードを説明するつもりはありません。むしろ、ここではコマンドそのものを見て、次のセクションからそれらを参照することになります。

コマンド0x20 - コマンドバイトの読み込みとコントローラRAMの読み込み

上の表を見てください。コマンド0x20～0x3Fは、コントローラRAMの読み出しに使われていることに気付きますか？それなのに、コマンド0x20はコマンドバイトの読み出しにも使われています。いったい何が起こっているのでしょうか？

実は、この2つは同じものを指しています。コマンドバイトは、コントローラのRAM内に格納されています。つまり、コマンドバイトを読むときは、コントローラの内部RAMから読んでいることになります。いいですか？

コントローラのRAMから読み出す場合、コマンドの最後の6ビットは、RAM内の読み出す場所を示します。一部のMCAシステムでは、RAM内の32のロケーションすべてにアクセスできます。他のシステムでは、0、0x13-0x17、0x1D、0x1Fのバイトにのみアクセスできます。

これらの場所は

- オフセット0：コマンドバイト
- Offset 0x13 (MCA): nonzero when password is
- enabled Offset 0x14 (MCA): nonzero when password is matched
- オフセット0x16-0x17 (MCA)：パスワード照合時に破棄される2つのメイクコードを与える オフ
- セット0x1D。
- オフセット0x1F。

ここでは、コマンドバイトの方が重要です。このバイトは、ここに示す特定のビットフォーマットに従います。見た目ほど複雑ではありませんので、ご安心ください。

- ビット0：キーボード割り込みイネー
 - ブル 0：割り込みを無効にする
 - 1：キーボード出力のバッファがいっぱいになったときにIRQ 1を送信する
- ビット1：マウスインタラプトイネーブル
 - ISA未使用
 - EISA / PS2
 - 0：マウスの割り込みを無効にする
 - 1：マウスの出力バッファがいっぱいになったときにIRQ12を送信する
- ビット2：システムフラグ（ステータスレジスタのビット2でも可） 0：コールドリブート
 - 1：ウォームリブート（BATはすでに完了しています）。
- ビット3：キーボードロックを無視する
 - PS/2：未使用
 - AT
 - 0：アクションなし
 - 1：ステータスレジスタのビット4を強制的に1にする（ロックしない）
- Bit 4: Keyboard Enable
 - 0: Enable Keyboard
 - 1: クロックラインをLowにしてキーボードを無効にする
- ビット5：Mouse Enable
 - EISAまたはPS/2
 - 0：マウスを有効にする
 - 1: クロックラインをLowにすることでマウスを無効にする
 - ISA
 - 0：PCモードでは、11ビットコードを使用し、パリティチェックとスキャン変換を行う
 - 1：PCモードでは、8086コードを使用し、パリティチェックを行わず、スキャン変換も行わない
- ビット6：トランスレーション
 - 0: 翻訳なし
 - 1: キーのスキャンコードを翻訳する。MCAタイプ2のコントローラはこのビットを設定できません
- ビット7：未使用、0にすべき

このコマンドが必要になることはないと思いますので、ルーチンは書いていません。

コマンド**0x60** - コマンドバイトの書き込みとコントローラ**RAM**の書き込み

コマンドバイト**0x60**～**0x7F**は、上記と非常によく似ており、上記と同じ**RAM**の位置に書き込むことができます。より重要なのは、コントローラの**RAM**のバイト**0**（コマンドバイト）を読み出すことで、これはコマンドバイト**0x60**を送信することで可能です。

上記のコマンドに加えて、エンドデモ用にこのコマンドのためのルーチンは書かれていません。

コマンド 0xAA - セルフテスト

このコマンドは、コントローラにセルフテストを実行させます。テストの結果は、ポート0x60から読める出力バッファに返されます。テストが成功した場合は0x55、失敗した場合は0xFCが返されます。

以下は、ルーチンの例です。最初にKYBRD_CTRL_CMD_SELF_TESTコマンド（コマンド0xAA）をキーボードコントローラに送信しているところに注目してください。その後、キーボードコントローラの出力バッファがデータで満たされるのを待ちます。これで、テストが完了したかどうかわかります。テストが完了すると、出力バッファの結果が0x55であればtrue（テスト成功）、そうでなければfalse（テスト失敗）を返します。

コマンド 0xAB - キーボードのセルフテスト

このコマンドを実行すると、コントローラとキーボード間のシリアルインターフェースをテストします。テストの結果は、ポート0x60で読める出力バッファに格納されます。

```
KYBRD_CTRL_SEND_CMD (KYBRD_CTRL_CMD_SELF_TEST);
```

結果は以下のいずれかとなります。0: 成

```
// ! 出力バッファがいっぱいになるのを待つ
```

- 功、エラーなし (1)
- 1: キーボードのクロックラインが低い read_status () & KYBRD_CTRL_STATS_MASK_OUT_BUF) break;
- 2: キーボードのクロックラインが高
- 止まり 3: キーボードのデータが0x55であれば、テスト合格
- ンが高止まり (key_read_buf () == 0xFF) ? true : false;

ご覧のとおり、これらはすべてハードウェアエラーです。エラーが発生した場合は、キーボードを無効にしてリセットすることをお勧めします。それでもダメな場合は、キーボードが故障している可能性があります。

コマンド 0xAD - キーボードの無効化

このコマンドにより、コントローラはキーボードクロックラインをディセーブルにし、コマンドバイトのビット4（キーボードイネーブル）を設定します。コマンドバイトのフォーマットについては、「コマンドバイトの読み出し」の項を参照してください。

つまり、このコマンドはキーボードを無効にします。

キーボードの現在の状態を保存しておくと、システムがキーボードの現在の状態を把握できるようになります。これは、demosのキーボードドライバでは、_kkybrd_disableを通じて行われます。

コマンド 0xAE - Enable Keyboard

このコマンドを実行すると、コントローラはキーボードクロックラインを有効にし、コマンドバイトのビット4（キーボードイネーブル）をクリアします。コマンドバイトのフォーマットについては、「コマンドバイトの読み出し」の項を参照してください。

```
_kkybrd_disable = true;
```

つまり、このコマンドはキーボードを有効にします。

ここでは、デモから抜粋したルーチンの例を紹介します。いかに簡単かお分かりいただけると思います。)

```
//! キーボードを有効にする
void kkybrd_enable ()
{
    KYBRD_CTRL_SEND_CMD (KYBRD_CTRL_CMD_ENABLE) です。
    _kkybrd_disable = false;
}
```

コマンド 0xC0 - 入力ポートの読み込み

このコマンドは、入力ポート（コントローラの**P10～P17**ライン）を読み取り、そのバイナリ値をポート**0x64**から読み取れる出力バッファにコピーします。このポートが持つラインをまだ見ていないので、今から見てみましょう。

- **Line P10 / Bit 0:** キーボードデータ入力、ISAでは未使用
- **Line P11 / Bit 1:** マウスデータ入力、ISAでは未使用
- **Line P12 / Bit 2:** ISA、EISA、PS/2では未使用
- **Line P13 / Bit 3:** ISA, EISA, PS/2では未使用です。
- **Line P14 / Bit 4:** 0: 512KBマザーボードRAM, 1: 256K RAM
- **Line P15 / Bit 5:** 0: 製造用ジャンパー装着、1: 未装着
- **Line P16 / Bit 6:** 0: CGAディスプレイ 1: MDAディスプレイ
- **Line P17 / Bit 7:** 0: キーボードロック 1: ロックなし

ジャンパがアクティブな場合、BIOSは無限大の診断ループを実行することがあります。ライン**P13**と**P14**は、クロック切り替え用に設定することができます。

これらが複雑に見えても気にしないでください -- 上記を見ると、最近のコンピュータではこのコマンドがあまり役に立たないことがわかるでしょう。ビット**0**、**1**、**2**、**3**はもう使われません。最近のコンピュータは**512KB**以上のRAMを搭載しているので、ビット**4**はほとんど役に立ちません。ビット**5**は、キーボードテスト用のジャンパーが取り付けられているかどうかをテストするために使用できます（ほとんどのユーザーは行いませんが）。ビット**6**はビデオアダプターから情報を得られるので必要ありません。ビット**7**はほとんどのユーザーがキーコードのロックを望まないで、ほとんど必要ありません。なるほど、とても便利なコマンドですね。それは可能ですが、ほとんどのコンピュータには必要ありません。

このコマンドは非常に便利なので、私はこのコマンドのためのルーチンを書かないことにしました。

コマンド 0xD0 - 出力ポートの読み込み

このコマンドは、コントローラの出力ポート（**P2**）から読み込んで、その結果をポート**0x64**の出力バッファに格納することを指示します。このコマンドを発行した後にポート**0x64**から読み込むことで、コントローラの出力ポートのビットを確認することができます。

コントローラの出力ポートは、ちょうどコントローラの**P20～P27**ラインです（前から覚えていますか？このコマンドを実行すると、これらのラインのバイナリ値が出力バッファに格納されます。

出力ポートのピンとその内容については、まだ説明していません。（**A20**の章で説明しましたが、詳しくは説明していません）なので、ここで説明します。

- **Line P20 / Bit 0:** 0: CPUをリセット、1: 通常動作
- **Line P21 / Bit 1:** 0: A20ラインを強制、1: 有効 **Line**
- **P22 / Bit 2:** マウスデータ。ISAでは未使用
- **Line P23 / Bit 3:** マウスクロック。ISAでは未使用
- **Line P24 / Bit 4:** 0: IRQ 1 not active, 1: IRQ 1 active
- **Line P25 / Bit 5:** 0: IRQ 12 not active, 1: IRQ 12 active
- **Line P26 / Bit 6:** キーボードクロック
- **Line P27 / Bit 7:** キーボードへのデータ

そうです。ビット**2**と**3**は、**ISA (Industry Standard Architecture)** コンピュータ（最近のほとんどのコンピュータ）では使われなくなりました。第**4**ビットと第**5**ビット（ライン**P24**と**P25**）は、PICライン**IR1**と**IR12**で**プログラム可能な割り込みコントローラ (PIC)** に接続されています。第**6**ビットと第**7**ビットには、現在のキーボードクロックとデータ信号が含まれています（ラインがアクティブかどうかに関わらず）。

今のところ、ここに書かれている内容はほとんど役に立たないものばかりですね。これらのビットの多くは、コントローラの現在の動作に関するエレクトロニクスレベルのもので、我々のニーズには役に立ちません。つまり、最初の**2**行（ビット**0**と**1**）を除いては、システムをリセットするか、**20**番目のアドレスラインを有効/無効にするかを制御します。しかし、ビット**0**を読んでも意味がありません。このラインはアクティブ（**1**）でなければならず、正常に動作していることを意味します。これがないとシステムが再起動してしまいます。したがって、ここで役に立つビットは**A20**ラインだけです。これは、少なくとも読み出し動作の場合に当てはまります。

このコマンドがポート**0x64**で発行された場合、結果のバイトは出力バッファに置かれ、ポート**0x60**からバイトを読み出すことで読み出すことができます。

すぐに**A20**を無効にしなければならないという心配はありません。また、キーボードからシステムをリセットする別の方法もあります。このように、このコマンドは私たちのニーズにはほとんど役に立たないので、ルーチンを書かないことにしました。

コマンド 0xD1 - Write Output Port

このコマンドは、出力バッファ（ポート**0x60**）からバイトをコピーし、コントローラの出力ポートラインにバイトを配置します。これらのラインの説明については、前のセクション（Read Output Port Command）を参照してください。

ほとんどの場合、問題が発生しないように、変更したい特定のビットをビットごとにORし、その他のビットは変更しないようにします。

このコマンドはいくつかの点で便利です。コントローラーが使用する**IRQ**の有効/無効、**A20**ゲートの有効/無効、さらにビット**0**を設定することでシステムをリセットすることができます。繰り返しになりますが、変更可能なビットのリストは前のセクションをご覧ください。

コマンド 0xE0 - Read Test Input

このコマンドは、コントローラ上のテストポートラインからバイナリ値を取得し、出力バッファに配置することで、ポート**0x60**から読み取ることができます。

テストポートは、マイクロコントローラの**TEST 0**と**TEST 1**のラインです（本章のコントローラのピンアウト図を参照してください）。この章では、テストポートについて説明していませんので、説明します。

- **Line TEST 0 / Bit 0** : キーボードクロック（入力）
- **Line TEST 1 / Bit 1**: AT - キーボードデータ(入力) PS/2 - マウスクロック(入力)

その他のビットは未定義とみなし、読み出さないでください。

このコマンドはあまり役に立たないかもしれませんが、コントローラは他の分野でも使用されている可能性があり、テストポートの方が役に立つかもしれないことを覚えておいてください。結局のところ、それはテストのためにあるのです。

コマンド 0xFE - システムリセット

コントローラの出力ポート（ピン**P0**）のビット**0**をパルスさせ、**CPU**をリセットします。これは基本的に、**Write Output Port**コマンドを送信してビット**0**をリセットするのと同じことです。システムをきれいにリセットしたい場合は、このコマンドを送信してください。

すべてのシステムで動作するとは限りませんのでご注意ください。動作するかどうかを確認する簡単な方法は、上記のルーチンの後に**kybrd_enc**がまだ実行されているかどうかを確認することです :)

デモ

```
// ! 出力ポートに1111110を書き込む（リセット系ラインをLowにする）
kybrd_ctrl_send_cmd (KYBRD_CTRL_CMD_WRITE_OUT_PORT);
kybrd_enc_send_cmd (0xfe);
```




最初のインタラクティブ・デモ

デモダウンロード

これまでのデモの中で最も複雑なものです。前章のコードに加え、キーボードドライバと基本的なコマンドラインインタフェース (CLI) を追加して、より興味深いものにしています。このため、初めてのインタラクティブなデモであり、独自のコマンドで拡張することも可能です。

また、このデモでは、各キーの読み取り間の遅延に使用される **sleep ()** ルーチンを追加し、デバッグ出力のコンソールルーチンに画面のスクロールを可能にしています。かっこいいでしょう？

CLI自体は非常にシンプルなものなので、そのコードを紹介するつもりはありません。むしろ、このデモのより重要な部分に焦点を当てたいと思います。

キーボードつなぎ合わせ

このデモのキーボードドライバのルーチンのいくつかをすでに見てきました。キーボードのエンコーダやコントローラとの通信や、有効化、無効化、テスト、LEDの更新、システムのリセットなど、さまざまな重要な機能のルーチンを見てきました。ここまでは良かったのですが、すべてを結びつける重要なディテールがいくつか欠けています。それを見てみましょう。

キーボード。現在の状態を保存する

ご存知のように、キーボードのどのキーもいつでも押せるようになっています。そのため、各キーをスキャンして、キーが押されているかどうかを確認する方法が必要です。しかし、キーボードエンコーダーは、この機能を備えています。さらに簡単なことに、キーボードエンコーダーはスキャンコードをオンボードのキーボードコントローラに直接送り、それによってIRQ 1が呼び出されます。

IRQ 1がマスクされていない限り、独自の割り込みハンドラをIRQ 1に設置して、キーボードエンコーダからスキャンコードが送信されるたびに通知を受けることができます。これは何を意味するのでしょうか？私たちの割り込みハンドラは、スキャンコードがキーボードコントローラに送信されるたびに呼び出されます。これはいつでも起こりうることです。

このため、ハンドラ内でスキャンコードをキーボードコントローラにポーリングすることで、スキャンコードを何らかの方法で判定する必要があります。しかし、あるキー (caps lockやnum lockキーなど) が押されていない場合には、異なる動作をさせたい場合があります。これらのキーは、押されたときにオンまたはオフになっているはずですが、では、shiftのような他のキーはどうでしょうか？これらのキーは、押したままにして、キーを離れたときに離す必要があります。

このため、これらのキーの現在の状態と最後に読み取ったスキャンコードを保存し、IRQが戻った後に再び取り出すことができるような方法を考え出す必要があります。これを実現するには

現在の状態をいくつかのグローバル変数や構造体に保存し、それらを単純に使用することができます。

キーボード。割り込み処理

これは重要なことです。キーを押したり離したりするたびに、数バイトのスキャンコードがキーボードコントローラに送られることを覚えていますか？これが発生すると、キーボードコントローラは**PIC (Programmable Interrupt Controller)** に信号を送り、**IRQ 1**を発生させます。そうです、読者の皆さん、この信号を受けて、PICはキーボード割り込みハンドラを実行します。

割り込みハンドラの目的は、ドライバの現在の状態を更新することと、スキャンコードをドライバとシステムが使用できる形式に変換して解釈することです。そう、それだけのことなんだよ)

割り込みハンドラは、すべてを結びつけるものです。ちょっと大きいので、この文章には載せませんが、皆さんにもぜひ見ていただいて、その動きを確認していただきたいと思います。

キーボード初期化

キーボードコントローラは、プログラマブル割り込みの**IRQ1**ラインに間接的に接続されていることを覚えていますか？PICを使って**IRQ**を割り込みベクター**32 (IRQ 0)** からマッピングしたので、**IRQ 1**は割り込みベクター

33。このため、割り込みベクター**33**を使用するためには、**setvect**ルーチンを使用して割り込みハンドラをインストールする必要があります。

それ以外は非常にシンプルです。**kkybrd_set_leds**ルーチンを使用して、現在のドライバの状態（グローバルとして保存）をクリアし、**LED**をクリアするだけです。

結論 `void kkybrd_install (int irq) {`

この章はこれで終わります。システムが面白くなってきたと思いませんか？このデモを發揮させて、ある程度使えるようになるかもしれません。しかし、現状ではできることが限られています。他のプログラムを走らせて作業を代行することができたら便利だと思いませんか？あるいは、他のファイルをディスクから読み込むこともできたら便利だと思いませんか？ファイルシステム全体の構造を抽象化することは非常に複雑なテーマで、ここでは1つのディスクからファイルを読み込むことに焦点を当ててみます。

```

kkybrd_bat_res = true;
scancode = 0;
しかし、ここで問題が発生する。最低限、ディスクからファイルをロードするためには、まずフロッピーディスクコントローラ (FDC) をプログラムしなければならないのだ。これが次の章のテーマです。お楽しみに
:)
_numlock = _scrolllock = _capslock =
false; kkybrd_set_leds (false, false,
false);

```

マイク `//! シフト、ctrl、alt の各キー`
BrokenThorn Entertainment 社。現在、**DoE** と **Neptune Operating System** を開発中です。質問やコメントはありますか？お気軽にお問い合わせください。

はありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ[私に教えてください](#)。



第18章

ホーム

第20





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - FDCプログラミング by Mike, 2009

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。



8272A フロッピーディスクコントローラ

はじめに

やったーいよいよフロッピーディスクを扱う時がやってきました。この章では、フロッピードライブとフロッピーディスクのプログラミングについて、ほぼすべての知識を網羅しています。

この章のメニューをご紹介します。FDCとFDDの

- 歴史
- ディスクレ
- イアウト
CHS, LBA
- FDD構造 FDC
- ハードウェア
- FDCとの連携
- FDCレジスターとコマンド

歴史

フロッピーディスクコントローラー (FDC) は、フロッピーディスクドライブ (FDD) とのインターフェイスとなるコントローラーです。PCでは通常、このような形で

nec ? pd765 fdc.PS/2はインテル82077Aを使用し、ATはインテル82072Aのマイコンを使用しています。フロッピーディスクドライブ(FDD)は、フロッピーディスクにデータを読み書きすることができる装置です。

1971年、IBM Direct Access Storage Product ManagerであるAlan Shugartに雇われたDavid L. Nobleは、System/370メインフレーム用の新しいストレージテープフォーマットの開発を試みた。IBMは、**ICPL (Initial Control Program Load)** 用のマイクロコードをリロードする際に、テープドライブよりも小型で高速なものを作りたいと考えていた。ノーブルズのチームは、「ミノー」というコードネームで「メモリーディスク」という製品を開発した。これは、80キロバイトの容量を持つ、読み取り専用の8インチのディスクレットである。1971年に発売され、すべての「System/370」メインフレームに同梱された。

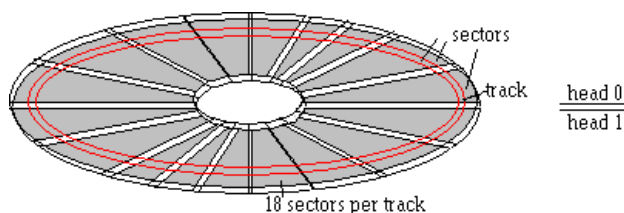
アラン・シュガートがIBMを辞めてメモレックス社に移ると、彼のチームは1972年に初の読み書き両用フロッピーディスクドライブ「Memorex 650」

を出荷した。フロッピーディスクは、IBMが8インチ、5と1/4インチ、3と1/2インチのフォーマットを発明した。

ディスク構造

物理的レイアウト

ディスクの構造を理解することは重要です。ここでは、フロッピーディスクのレイアウトを紹介します。



これは、一般的な3-1/2インチフロッピーディスクの物理的なレイアウトです。ここではヘッド1（表側）を見ていますが、セクタは512バイトを表しています。トラックとはセクタの集合体である。

注：1セクタは512バイトで、フロッピーディスクの1トラックは18セクタであることを忘れないでください。

上の写真を見て、思い出してください。

各トラックは通常、512バイトのセクタに分割されています。フロッピーでは、1トラックに18セクタあります。

- シリンダーとは、同じ半径を持つトラックの集まりのことです（上の写真の赤いトラックが1つのシリンダーです）。フロッ
- ピーディスクには2つのヘッドがあります（写真に表示されています）。
- セクター数は2880。

より深く理解するために、**CHS**を見てみましょう。次はそれを見てみましょう

シリンダー／ヘッド／セクター（CHS）

セクター

セクター」とは、簡単に言えば512バイトのグループのこと。つまり、セクター1は、ディスクの最初の512バイトを表しています。

ヘッド

ヘッド」（またはフェース）は、ディスクの側面を表しています。ヘッド0が表側、ヘッド1が裏側になります。ほとんどのディスクは1面しかないので、ヘッドも1つしかありません（「ヘッド1」）。

トラック

トラックとは、ディスクを1周すること。フロッピーディスクの場合、1つのトラックには18セクタが存在する。

シリンダー番号は、1枚のディスクのトラック番号を表す。フロッピーディスクの場合は、読み取りを行うトラックを表します。

1トラックに18セクターあります。片面80トラック。

CHSについて

フロッピーディスクのアドレスは、CHSフォーマットを使用している。ディスクの任意の位置から読み書きするためには、FDCにRead/Write Headからディスク上の正確なトラック、シリンダー、セクタに読み書きを行う。

リニア・ブロック・アドレッシング（LBA）

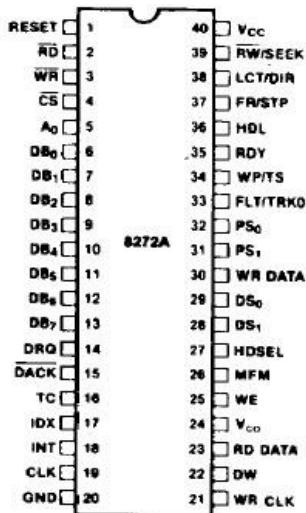
また、**LBA（Linear Block Addressing）**を用いて、より抽象的なディスクの読み書き方法を提供することもできます。LBAでは、セクタ0～2880までのディスク上の任意のセクタに対して読み書きが可能となる。

フロッピーのインターフェイス

ソフトウェアは、フロッピーディスクコントローラを介してフロッピーディスクドライブを制御することで、フロッピーディスクドライブとのインターフェースを実現しています。ここでは、フロッピーディスクコントローラの種類の違いから、初代の「8272A」を取り上げます。冒頭の画像は、8272Aの代表的なIC（集積回路）コントローラです。ここではこのICを見ていきます。

詳細82072A フロッピーマイクロコントローラ

8272Aは40ピンのICです。ここではそれを見てみましょう。ここでは、40ピンすべてを簡単に見ていきますが、ここでは、電子工学の分野にまで踏み込んでいくので、詳細には見ません。



これらのピンのほとんどは、コントローラのプログラミングにはあまり役に立ちません。しかし、他のピンは理解するのに重要です。それでは見ていきましょう。わかりやすくするために、すべてのピンを簡単に見ていきます。FDCは間接的に、**PIC（Programmable Interrupt Controller）** やシステムバス、**DMAC（Direct Memory Access Controller）** などと通信していることがわかります。

- **RESET**端子 - FDCをアイドル状態にします。すべての出力ラインをLOWにします。**Vcc**ピンは、+5Vの電源入力です。
- **GND**端子 - は、グランド端子です。
- **CLK**端子 - 典型的な単相8MHz方形波クロック信号。
- **RD**端子 - FDCに現在の動作がリード動作であることを伝える。
- **WR**端子 - 同様の機能を持ちますが、書き込み操作を行います。
 - これらは、ソフトウェアによる**I/Oリード/ライト**操作でコントロールバスによって設定されます。
- **CS**端子-チップセレクト
- **DB0 - DB7**ピン - 双方向の8ビットデータバス。システムの**プライマリデータバス**に間接的に接続されます。
- **A0**端子 - データ/ステータスレジスタセレクト端子。High(1)の場合は、FDCの**データレジスタ**の内容をデータバスに配置するように指示します。ロー(0)の場合は、**ステータス・レジスタ**の内容をデータ・バスにコピーします。これは、出力データバスピンDB0～DB7を介して行われ、さらに、ソフトウェアで読み取ることができるシステムデータバスを介して行われます。
- **DRQ**ピン - データダイレクトメモリアクセス(**DMA**)リクエストピン。このラインが**High(1)**の場合、**FDCはDMAリクエストを行っています**。
- **DACK**ピン - **DMAアックノレッジ**ピン。コントローラがDMA転送を行っているときは、このラインは**LOW(0)**になります。
- **TC**端子 - DMA転送が完了すると、FDCは**Terminal Count**端子であるTCをH(1)にします。

- **IDX**端子 - FDCがディスクトラックの先頭にあるときに**High**になります。

- **INT**端子 - FDCが割り込み要求(**IR**)を送信すると、High(1)になります。の**IR6**に間接的に接続されています。
- **PIC (Programmable Interrupt Controller)**。
- **RW/Seek**端子 - 読み書き両用モードのシークモードを設定する。1: シークモード, 0: リード/ライトモード
- **LCT/DIR**端子 - 低消費電流/方向指定端子。
- **FR/STP**端子 - フォールトリセット/ステップ端子。
- **HDL**端子 - **Head Load**端子。FDDのRead/Writeヘッドをディスクに接触させるコマンドです。
- **RDY**端子 - レディ端子。FDDがデータを送信または受信する準備ができていることを示します。
- **WP/TS**端子 - ライトプロテクト/ツースайд端子。リード/ライトモードでは、メディアがライトプロテクトされている場合にHighに設定します。シークモードの場合、メディアが両面の場合にHighに設定します。
- **FLT/TRK0**ピン - フォールト/トラック0ピン。Read/Writeモードでは、FDDのフォルトが検出されるとHighになります。
- **PS0~PS2**端子 - 前置補正 (**Pre-shift**) 端子です。**MFM**モード時には、事前補正の状態を書き込みます。
- **WR DATA**端子 - ライトデータ端子
- **RD DATA**端子 - リードデータ端子
- **DS0 - DS1**端子 - ドライブセレクト端子
- **HDSEL**端子 - ヘッドセレクト端子。High(1)の時はFDCがヘッド1にアクセスするように設定します。Lowの時はヘッド0となります。
- **MFM**端子 - Highの場合、FDCは**MFM**モードで動作します。Low(0)の場合は、**FM**モードで動作します。
- **WE**端子 - ライトイネーブル端子。
- **VCO**端子 - **VCO Sync**端子。0であれば、**PLL**内の**VCO**を禁止する。1で**VCO**を有効にします。
- **DW** 端子 - データウィンドウ端子。PLLにより生成され、FDDからのサンプルデータに使用される。
- **WR CLK**端子 - ライトクロック

FDCは、**DMA (Direct Memory Access)** コントローラの有無にかかわらず動作可能です。非DMAモードで動作している場合は、プロセッサとFDCの間でデータバイトが転送されるたびに**IRQ 6**が生成されます。DMAモードでは、プロセッサがFDCにコマンドを読み込み、FDCとDMAコントローラの制御下ですべてのデータ転送が行われます。

これは重要なことです。FDCのピンをすべて把握する必要はありません。むしろ、**FDCは3つの主要なコントローラと通信していることを覚えておいてください**。1つ目は、4つのフロッピーディスクドライブ (**FDD**) 内部コントローラ、**プログラマブルインタラプトコントローラ (PIC)**、**ダイレクトメモリアクセス (DMA) コントローラ**のいずれかです。ソフトウェアは、プロセッサ標準の**IN/OUT**ポートi/o命令によってFDCと通信します。

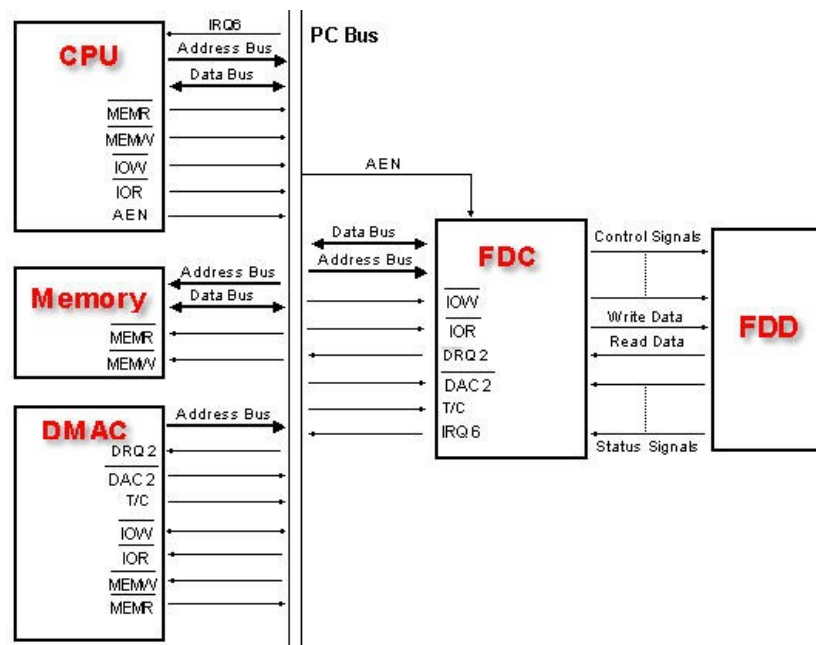
FDCのいくつかのレジスタは、プロセッサのI/Oアドレス空間にマッピングされています。標準的なI/Oポートの読み出しと同様に、入出力動作の間、プロセッサはコントロールバスに**READ**または**WRITE**ラインを設定し、アドレスバスにポートアドレスを設定します。これは、**システムバス**または**ISA (Industry Standard Architecture)** バスで行われます。

最近のハードウェアでは、FDCはISAバスに直接接続されておらず、スーパーI/O ICとして統合されており、スーパーI/Oの**Low Pin Count**バスを通じてプロセッサと通信している。

なるほど!」と思いました。ソフトウェアがFDCと通信する方法はわかりました。PICとDMAの出番は?

上のピンリストを見ると、FDCには**INT**というピンがあることがわかります。このラインは、**プログラマブルインタラプトコントローラのIR 6ライン**に間接的に接続されています。FDCは、データのバイトが読み書きできる状態になると、このラインをハイ (1) に引きます。これにより、PICのIR 6ラインもハイに引き出されます。ここからはPICが制御します。他のラインをマスクアウトし、サービスを受けられるかどうかを判断します。プロセッサの**インタラプトアクトレッジ (INTA)** ピンをアクティブにして、プロセッサに割り込みを通知します。プロセッサは、割り込みを処理しても問題ないことを確認すると、INTAラインをリセットして、PICに処理を行うことを許可します。PICは、このIRQが使用するようにマッピングされた (PICの初期化時に設定された) 割り込みベクターを配置します。プロセッサはIRQを受け取り、そのアドレスをidtrから取得して、ほら、私たちの割り込みが呼ばれます。

FDCは、DMAモードで動作するようにプログラムすることもできます。DMAは、まだ見たことのないコントローラです。そのため、ここではあまり触れてくありません。しかし、完璧を期すために次の章で説明するかもしれません。**FDCはDMAチャンネル2に接続されています**。

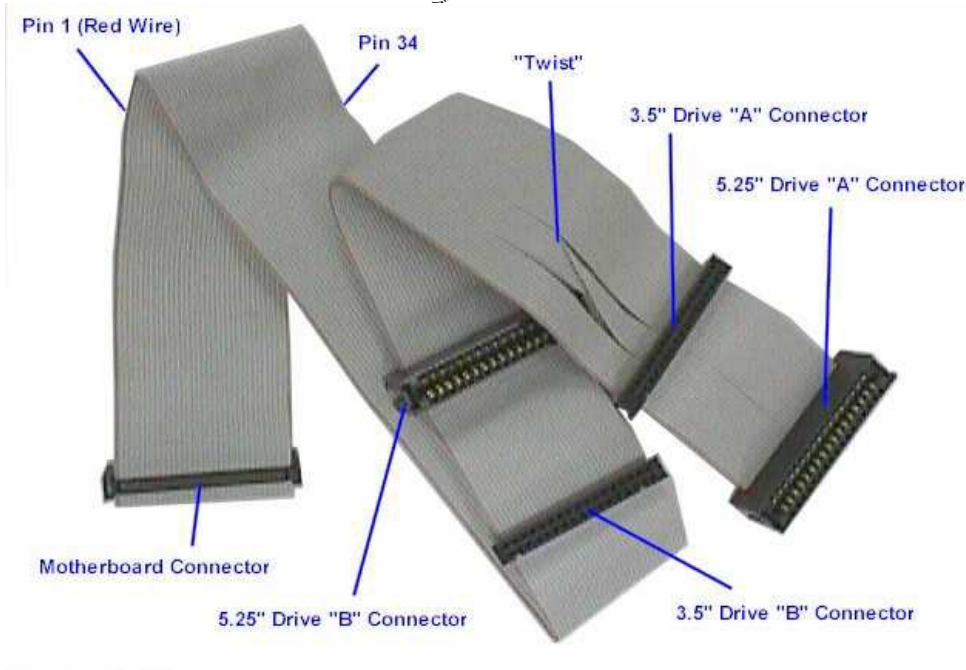


これがFDCのハードウェアのすべてです。**FDCは1つのコンピュータシステム内に複数設置することができます**。各FDCには最大4台のフロッピーディスクドライブ (**FDD**) を接続することができます。これは重要なことです。**FDCと通信する際には、どのFDDに対するリクエストかを選択しなければならないことが多いです**。

フロッピーインターフェースケーブル

FDCとFDDは、フロッピーインターフェースケーブルを介して通信します。フロッピーインターフェースケーブルは、**PATA (Parallel ATA)** ケーブルの一種です。

ウエスタンデジタル社から発展した**IDE (Integrated Drive Electronics)** ケーブル。



上記のケーブルにねじれがあることに気づくはずですが。それを少し短く説明します。このケーブルには40本のピンがあります。この40本のピンを介して、FDCはケーブルに接続されている異なるFDDと会話することができます。

FDCとの通信に使われるレジスタの中には、コントローラやケーブルの入力ピンを検出できるものがあります。このため、せめてケーブルの40本のラインを少しでも見ておくべきだろう。

フロッピーインターフェースケーブルのピン			
ピン	説明	ピン	ディスクリプション
0	リセット	20	DDRQ
1	グラウンド	21	グラウンド
2	データ端子7	22	I/O Write
3	データ端子8	23	グラウンド
4	データ端子6	24	I/O リード
5	データ端子9	25	グラウンド
6	データ端子5	26	IOCHRDY
7	データ端子 10	27	ケーブルセレクト (CS
8	データ端子4	28	DDACK
9	データ端子11	29	グラウンド
10	データ端子3	30	インタラプト
11	データ端子12	31	(接続なし)
12	データ端子2	32	アドレス1
13	データ端子13	33	GPIO_DMA66_Detect
14	データ端子1	34	アドレス 0
15	データ端子 14	35	アドレス2
16	データ端子0	36	チップセレクト1
17	データ端子 15	37	チップセレクト3
18	グラウンド	38	アクティビティ
19	キーまたは Vcc_in	39	グラウンド

- 後日、さらに追加されます。

FDCプログラミング

FDCの動作モード

最近のFDCのほとんどは、オリジナルの8272マイクロコントローラよりも進化しています。後方互換性を確保するために、新しいFDCはコントローラにピンを追加し、特定のモードで動作するときに異なるレジスタと通信できるようにしています。例えば、ステータスレジスタAモードは、コントローラがPC-ATモードで動作しているときにのみアクセスできます。コントローラをリセットすると、コントローラはデフォルトの82077Aモードで動作します。

IRQの待ち受け

FDCはIRQ6を使用していることを覚えていますか？FDCは、リードコマンドやライトコマンドが完了した後、またはモードによっては1バイト転送するごとにバイトを送信します。また、初期化時にコントローラがリセットされるとIRQを送信します。

2021/11/15 13:10

化中にのみ割り込みが発生するということです。

オペレーティングシステム開発シリーズ

ズ

しかし、どのような場合でも、コマンドが完了したことを知るために、**IRQ**が発生するのを待つ必要があります。これを実現するには、**IRQ**が発火したときにグローバルを設定し、**IRQ**を待ち、発火したときにグローバルをリセットする**irq_wait**のような関数を用意します。

それでは、早速やってみましょう。まず、**IRQ**です。

```
const int FLOPPY_IRQ = 6;

// ! IRQが発生したときに設定
static volatile uint8_t _FloppyDiskIRQ = 0;

void _cdecl i86_flpy_irq () { 。

    _asm add esp, 12
    _asm pushad
    _asm cli

    // ! irqが発生
    _FloppyDiskIRQ = 1;

    // ! ハルに終了を伝える
    interruptdone( FLOPPY_IRQ
    );

    _asm sti
    _asm popad
    _asm iretd
}
```

これは、**PIT**の**IRQ**のように簡単そうですね。)そうですね、あとは待つだけです。

```
// ! IRQの発火を待つ
inline void flpydisk_wait_irq () {...

    IRQが発生するのを待つ while (
        _FloppyDiskIRQ == 0 )
        ;
    _FloppyDiskIRQ = 0;
}
```

シンプルでいいですね。つまり、読み取りや書き込みなどのコマンドを送るとしたら、単に **flpydisk_wait_irq()** を呼び出すだけです。これが完了すると、コマンドが終了し、続行しても安全であることがわかります。かっこいいでしょ?)

DMA?

えっ? **FDC**を**DMA**モードでプログラミングするの? でも、まだ**DMA**の話をしていないじゃないですか。そうそう、これが問題なんですよ。

私は当初、**FDC**を**Non-DMA**モードでプログラムするつもりでした。しかし、多くのエミュレータや一部のハードウェアでは**DMA**モードがサポートされていませんでした。そこで、移植性を確保するためにも、**DMA (Direct Memory Access Controller[DMAC])** を利用するのがベストだと判断しました。

しかし、**DMA**についてはまだ詳しく説明していないため、問題が発生しています。そこで、**DMA**のインターフェースを説明なしに丸投げするのではなく、3つの基本的な**DMA**ルーチンをハックして、後でより詳細に書き直すことにしました。)

flpydisk_initialize_dmaは、基本的に**DMA**が使用するバッファを物理アドレス**0x1000~0x10000(64k)**に作成します。ディスクからセクターを読み出すとき、**DMA**はセクターデータをこの場所に置くので、上書きされてしまうので、何も無いことを確認してください。他の場所を選んでも構いませんが、いくつかのルールがあります。

- バッファは**64k**の境界を越えることはできません。最高のパフォーマンスを得るためには、**64k**の境界に留める必要があります。
- 書き込み先のメモリ領域は、アイデンティティマップされているか、フレームアドレスがページにマッピングされている必要があります。**DMA**は常に物理メモリで動作する

デモではバッファに**0x1000 + 64k**を使用しているので、変更したくない場合はそのままにしておきます。

dma_readと**dma_write**は、**FDC**から送られてきたデータの読み書きを開始するよう**DMA**に指示するだけです。これは、**FDC**に読み書きを指示したセクターになります。例えば、**FDC**にセクタの読み出しを指示すると、**FDC**は**DMA**にセクタデータを渡し、**DMA**に設定したバッファ (**0x1000**) に置かれます。かっこいいでしょ?

```
// ! Phys addr 1k-64kを使用するためにDMAを初期化する void flpydisk_initialize_dma () {。

    outportb (0x0a, 0x06); //DMAチャンネル2をマスクする
    outportb (0xd8, 0xff);   る
    outportb (0x04, 0);      マスターフリップフロップをリ
    outportb (0x04, 0x10);   セットする
    outportb (0xd8, 0xff);   7/8アドレス=0x10007 フリップをリセットする
    outportb (0x05, 0xff);   //0x23ffまでのカウント (3.5インチフロッピーディスクのトラックのバイト数)
    outportb (0x05, 0x23);   //外部ページレジスタ = 0
    outportb (0x80, 0);      //DMAチャンネル2のアンマスキング
    outportb (0x0a, 0x02)
}
// ! リードセクタのためのDMAの準備 void flpydisk_dma_read () {。

    outportb (0x0a, 0x06); //DMAチャンネル2をマスクする
    outportb (0x0b, 0x56); //シングル転送、アドレスインクリメント、オートユニット、リード、チャンネル2
```

```
        outportb (0x0a, 0x02); //DMAチャンネル2のマスク解除
    }

    ///書き込み転送のためのDMAの準備 void
    flpydisk_dma_write () {

        outportb (0x0a, 0x06); //DMAチャンネル2をマスクする
        outportb (0x0b, 0x5a); //シングル転送、アドレスインクリメント、オートイニット、ライト、チャネル2
        outportb (0x0a, 0x02); //DMAチャネル2のアンマスク
    }
```

上記のコードが理解できなくても、心配はいりません。DMAに関するすべてのことは、次のチュートリアルでDMAを詳しく説明する際に、書き直して説明します。

FDCポートマッピング

FDCには、i86のI/Oアドレス空間にマッピングされた4つの外部レジスターがあります。これらはソフトウェアが標準的なI/O命令でアクセスすることができます。これらのレジスターを太字にした。

一部のシステムでは、FDCに提供される外部レジスターの数が、主要な4つのレジスターよりも多

い場合があります。第2のFDCは、通常、I/Oポート0x370~0x377にマッピングされます。

2つの異なるFDCのための2つのポートセットがあるので、この表には両方のポートセットが含まれます。

フロッピーディスクコントローラポート			
ポート(FDC 0)	ポート(FDC 1)	リード/ライ ト	ディスクリプション
プライマリFDCレジスタ			
0x3F2	0x372	ライトオンリ	デジタル出力レジスタ (DOR
0x3F4	0x374	読み取り専用	メイン・ステータス・レジスタ (MSR
0x3F5	0x375	読み取り/書き込み	データレジスタ
0x3F7	0x377	読み取り専用	ATのみ。コンフィギュレーション・コントロール・レジスタ (CCR
0x3F7	0x377	ライトオンリ	ATのみ。デジタル入力レジスタ (DIR
その他のFDCレジスタ			
0x3F0	0x370	読み取り専用	PS/2のみ。ステータスレジスタA (SRA
0x3F1	0x371	読み取り専用	PS/2のみ。ステータスレジスタB (SRB
0x3F4	0x374	ライトオンリ	PS/2のみ。データレート選択レジスタ (DSR

次のセクションでは、レジスターを少しずつ詳しく見ていきます。とりあえず、重要なものを取り上げます。この章では、完璧を期すために、他のレジスターについても紹介することにするかもしれませんが、今のところ、上に示した最初の4つのレジスターにのみ焦点を当てます。

これらのコードはすべて、この章の最後にあるデモに含まれていることを忘れないでください。

```
enum FLPYDSK_IO
{...
    FLPYDSK_DOR           = 0x3f2です。
    FLPYDSK_MSR           = 0x3f4です。
    FLPYDSK_FIFO          = 0x3f5, //データレジスタ
    FLPYDSK_CTRL          = 0x3f7
};
```

レジスター

ステータスレジスタA (SRA) (PS2モードのみ

このレジスタを知っている必要はありません。ここにあるのは完全性のためだけです。

このレジスタは、コントローラのいくつかのインターフェースピンの状態を監視するリードオンリーのレジスタです。コントローラがPC-ATモードの場合はアクセスできません。これは読み取り専用のレジスタです。

このレジスタの正確なフォーマットは、コントローラのモデルによって異なります。

- ビット0 DIR
- ビット1 WP
- ビット2 INDX
- ビット3 HDSEL
- ビット4 TRKO
- ビット5 STEPフリップ/フロップ
- ビット6 DRV2
- ビット7 INTERRUPTラインの状態 (割込み保留

警告これらのビットは、コントローラのモデルによって変わることがあります。

このレジスタが複雑に見えても気にしないでください。本シリーズでは使用しませんのでご了承ください。

ステータスレジスタB (SRB) (PS/2モードのみ)

このレジスタを知っている必要はありません。ここにあるのは完全性のためだけです。

上記のレジスタと同様に、FDCのいくつかのラインの状態を監視することができます。FDCがPC-ATモードの時はアクセスできません。このレジスタは読み取り専用です。

- **ビット0** MOT EN0 (モータ・イネーブル0) **ビット1** MOT EN1 (モータ・イネーブル1) **ビット2** WE フリップ/フロップ
- **Bit 3** リードデータ (RDDATA) フリップ/フロップ **Bit 4** ライトデータ (WRDATA) フリップ/フロップ **Bit 5** ドライブセレクト 0
- **ビット6** 未定義、常に1
- **ビット7** 未定義、常に1

警告これらのビットは、コントローラのモデルによって変わることがあります。

このレジスタが複雑に見えても気にしないでください。本シリーズでは使用しませんのでご了承ください。

データレートセレクトレジスタ (DSR)

このレジスタを知っている必要はありません。ここにあるのは完全性のためだけです。

このレジスタは、ドライブ制御信号のタイミングを変更することができる書き込み専用のレジスタです。I/Oポート0x3f4 (FDC 0) または0x374 (FDC 1) に書き込むことで使用されます。

これは8ビットのレジスタです。以下のようなフォーマットになっています。

- **ビット0** DRATE SEL0 **ビット1** DRATE SEL1 **ビット2** PRE-COMP 0
- **ビット3** PRE-COMP 1
- **ビット4** PRE-COMP 2
- **Bit 5** Must be 0
- **Bit 6** POWER DOWN : 内部クロックを停止し、内部発振器を停止する
- **ビット7** S/W RESET。内部振動子のリセット

PRE-COMP 0~PRE-COMP 2は少し複雑です。これらは、フロッピードライブなどの磁気メディアで発生する**ビットシフト**に対して、**WRDATA**出力ピンを調整します。前置補正の遅延を調整するには、これらのビットを以下のいずれかに設定します。

- **000** デフォルト (250~500Kbps : 125ns、1Mbps : 41.67ns)
- **110** 250 ns
- **101** 208.33 ns
- **100** 166.67 ns
- **011** 125 ns
- **010** 83.34 ns
- **001** 41.67 ns
- **111** 無効化

DRATE SEL0 - DERATE SEL1でデータレートを設定します。有効な値を以下に示します。

- **00** 500 Kbps
- **10** 250 Kbps
- **01** 300 Kbps
- **11** 1Mbps

警告データレートをドライブの許容範囲を超えて設定すると、エラーが発生する可能性があります。

デジタル出力レジスタ (DOR)

Yey!最初の便利なレジスター!これは知っておいて損はない。

これは**書き込み専用**のレジスタで、FDDのモーター制御、動作モード (DMA, IRQ) , リセット, ドライブなど、FDCのさまざまな機能を制御することができます。フォーマットを持っています。

- **ビット0-1** DR1, DR2
 - 00 - ドライブ0
 - 01 - ドライブ1
 - 10 - ドライブ2
 - 11 - ドライブ3
- **ビット2** REST
 - 0 - コントローラのリセット
 - 1 - コントローラ対応
- **ビット3** モード
 - 0 - IRQチャンネル
 - 1 - DMAモード
- **ビット4~7** モーターコントロール (ドライブ0~3)
 - 0 - ドライブのモーター停止
 - 1 - 駆動用モーターの起動

これは簡単なことです。基本的には、FDCの機能を制御するコマンドを送信する際には、どのドライブ用か (1台のFDCが4台のFDDと通信できることを覚えておいてください!)、コントローラのリセット状態、動作モード (FDCはDMAとIRQの両方のモードで動作できることを覚えておいてください!)、特定

のFDD内部モーターの状態を選択するためのビットパターンを構築するだけです。

以下にその例を示します。最初のコピーディスクドライブ（FDD 0）のモーターを起動させたいとします。**FDDのモーターを起動させるには、FDDに対して読み書きの操作を行う前に行う必要があります。モーターを起動させるには、起動または停止させたいドライブに対応するビット（4-7）を設定します。**他のビットを0にしておくと、通常の動作（IRQモード、コントローラのリセット）になります。DORがプロセッサのi/oアドレス空間のポート0x3f2にマッピングされていることを知っているの、これは非常に簡単になります。まず、読みやすさを向上させるために、レジスタのビットマスクを作成します。これらのコードはすべて、このチュートリアル最後にあるデモにも含まれていることを覚えておいてください。

```
enum FLPYDSK_DOR_MASK {...
    flpydisk_dor_mask_drive0 = 0, //00000000 =ここでは念のため
    flpydisk_dor_mask_drive1 = 1, //00000001
    flpydisk_dor_mask_drive2 = 2, //00000010
    FLPYDSK_DOR_MASK_DRIVE3 = 3, //00000011
    flpydisk_dor_mask_reset = 4, //00000100
    flpydisk_dor_mask_dma = 8, //00001000
    flpydisk_dor_mask_drive0_motor = 16, //00010000
    flpydisk_dor_mask_drive1_motor = 32, //00100000
    flpydisk_dor_mask_drive2_motor = 64, //01000000
    flpydisk_dor_mask_drive3_motor = 128 //10000000
};
```

上記のビットマスクを使用すると、設定したい異なるビットをビットごとにORするだけでよいのです。たとえば、フロッピーディスクドライブ0のモーターを起動するには、次のようにします。

```
outportb (FLPYDSK_DOR, FLPYDSK_DOR_MASK_DRIVE0_MOTOR | FLPYDSK_DOR_MASK_RESET)となります。
```

FLPYDSK_DORは先ほど0x3f2と定義しましたが、これはDOR FDCレジスタのi/oアドレスであることを思い出してください。また、上記はコントローラをリセットします。

この同じモーターをオフにするには、モータービットを設定せずに同じコマンドを送信します。

```
outportb (FLPYDSK_DOR, FLPYDSK_DOR_MASK_RESET)。
```

警告モーターの起動には時間がかかります。内蔵のFDDモーターは機械式であることを忘れないでください。他の機械装置と同様に、ソフトウェアの動作速度よりも遅くなる傾向があります。このため、FDDモーターを起動する際には、読み取りまたは書き込みを行う前に、必ず少し時間を置いてから回転させてください。

DORは書き込み専用のレジスタです。これを実現するために、ルーチンを作成します。

```
次の重要登録に移ります。write_dor (uint8_t val ) {.
```

メイン・ステータスレジスタ (MSR)

```
outportb (FLPYDSK_DOR, val)を書き込みま
```

メイン・ステータス・レジスタ (MSR) は、特定のビットフォーマットに従っています。これは予想外だったでしょう。さて、話を元に戻しましょう (シヤレです)。MSRのフォーマットは次のとおりです。

- **ビット0** - FDD 0 : FDDがシークモードでビジー
- の場合は1 **ビット1** - FDD 1 : FDDがシークモードでビジーの場合は1 **ビット2** - FDD 2 : FDDがシークモードでビジーの場合は1 **ビット3** - FDD 3 : FDDがシークモードでビジーの場合は1 0 : 選択されたFDDはビジーではありません。
 - 1 : 選択したFDDがビジー状態
- **Bit 4** - FDC Busy; ReadまたはWriteコマンド進行中 0:
 - Not busy
 - 1: ビジー
- **ビット5** - 非DMAモードのFDC
 - 0: DMAモードのFDC
 - 1: FDCはDMAモードではない
- **Bit 6** - DIO: FDC ICとCPU間のデータ転送の方向 0: FDCはCPUからのデータ
 - を期待している
 - 1: FDCがCPU用のデータを持っている
- **ビット7** - RQM。データレジスタはデータ転送の準備ができています 0: データレジスタは準備ができていない
 - 1: データレジスタレディ

このMSRはシンプルなものです。FDCとディスクドライブの現在のステータス情報を含んでいます。コマンドを送信したり、FDDから読み出したりする前に、常にFDCの現在のステータスをチェックして、準備ができていないことを確認する必要があります。

ここでは、このMSRがビジー状態かどうかを読み取る例を示します。まず、コードで使用されるビットマスクを定義します。上の図のようなフォーマットになっていることに注目してください。

```
enum FLPYDSK_MSR_MASK
{...
    flpydisk_msr_mask_drive1_pos_mode = 1, //00000001
    flpydisk_msr_mask_drive2_pos_mode = 2, //00000010
    flpydisk_msr_mask_drive3_pos_mode = 4, //00000100
    flpydisk_msr_mask_drive4_pos_mode = 8, //00001000
    flpydisk_msr_mask_busy = 16, //00010000
    flpydisk_msr_mask_dma = 32, //00100000
    flpydisk_msr_mask_dataio = 64, //01000000
    flpydisk_msr_mask_datareg = 128 //10000000
};
```

簡単でしょうか？では、FDCがビジー状態（BUSYフラグがセットされている）かどうかをテストしてみましょう。FLPYDSK_MSRがMSRのi/oポートアドレスである0x3f4であることを知っているの、必要なことは次のとおりです。

```
if ( inportb (FLPYDSK_MSR) & FLPYDSK_MSR_MASK_BUSY )
    //!FDCは忙しい
```

読み書きのコマンドを送るときは、このビットが0になるまで待てばいいのです。

読みやすくするために、これをルーチンに隠すことにしたので、ここに紹介します。このルーチンは、FDCのステータスを返すだけです。

テープドライブレジスタ（TDR）

このレジスタを知っている必要はありません。ここに示すのは完全性のためだけです。

```
return inportb (FLPYDSK_MSR);
```

このレジスタは、特定のドライブの初期化時に、そのドライブにテープ・ドライブ・サポートを割り当てることができます。このレジスタはリード/ライトレジスタで、サイズは8ビットです。--七か七、--最初の2ビットのみが定義されています。--ドライブ0はフロッピーブートデバイス用に予約されているため、選択することはできません。そのため、以下のビットリストには含まれていません。

- 00: None.
- 01: ドライブ1
- 10: ドライブ2
- 11: ドライブ3

このレジスタは、ハードウェアリセットのみでリセットされます。ソフトウェアのリセットでは効果がありません。テープドライブに詳しくなくても心配ありません。このレジスタは私たちには適用されませんし、このシリーズでは使用しません。このレジスタは我々には適用されず、このシリーズでは使用されません。)

データレジスタ

これは、8ビットまたは16ビットのリード/ライトレジスタです。レジスタの実際のサイズは、コントローラの種類によって異なります。すべてのコマンドパラメータとディスクデータの転送は、データレジスタへの読み出しと書き込みを行います。このレジスタは、特定のビットフォーマットに従わず、一般的なデータに使用されます。I/Oポート0x3f5(FDC 0)または0x375(FDC 1)からアクセスできます。

注：このレジスタを読み書きする前に、まずマスター・ステータス・レジスタ（MSR）のステータスを読み、そのレジスタが有効であることを確認する必要があります。

覚えておってください。すべてのコマンドバイトとコマンドパラメータは、このレジスタを介してFDCに送信されます。以下のコマンドセクションでこの例を見ることができますので、まだあまり気にしないでください。

無効なコマンドが発行された場合、データレジスタから返される値は0x80です。

以下のルーチンは、このレジスタから読み取り、デモで使用されます。このルーチンは、データ・レジスタが読み書き可能な状態になるまで待機し、その後、データの読み取り（read_data関数）または書き込み（send_command関数）を行います。

```
void flpydisk_send_command (uint8_t cmd) {
    // ! データレジスタの準備が整うまで待ちます。データ・レジスタにコマンドを送る
    for (int i = 0; i < 500; i++)
        if ( flpydisk_read_status () & FLPYDSK_MSR_MASK_DATAREG )
            return outportb (FLPYDSK_FIFO, cmd);
}

uint8_t flpydisk_read_data () {...
    (int i = 0; i < 500; i++) // ! 上の関数と同じですが、読み込み用のデータレジスタを返します。
        if ( flpydisk_read_status () & FLPYDSK_MSR_MASK_DATAREG )
            return inportb (FLPYDSK_FIFO);
}
```

デジタル入力レジスタ（DIR）

このレジスタを知っている必要はありません。ここにあるのは完全性のためだけです。

さて、デジタル出力レジスタ（DOR）がありましたので、これは予想していたことだと思いますが :) このレジスタは、コントローラのすべての動作モードで読み取り専用です。PC-ATモードではビット7のみが定義され、その他のビットは未定義であり、使用してはいけません。他の動作モードでは、ビット7は定義されません。

ビット7（DSK CHG）は、FDCのDSK CHG端子をモニターします。本章冒頭のピン配置を見ると、DSK CHG端子がないことがわかります。これは、FDCの最新モデルと初代モデルの違いに関係しています。新しいモデルでは、DMA GATEやDRATE SEL0/1など、FDCの新しいピンを監視するために、このレジスタの異なるビットが追加・変更されています。このレジスタの値は、FDCの動作モードに固有のものです。

なお、このレジスタのビットは、モデルによって変更されることがあります。

コンフィグレーション・コントロール・レジスタ（CCR）

PC/ATモードでは、このレジスタはデータレートセレクトレジスタ（DSR）と呼ばれ、最初の2ビット（ビット0=DRATE SEL0、ビット1=DRATE SEL1）のみが設定されています。もう一度見てみましょう...

- **00** 500 Kbps
- **10** 250 Kbps
- **01** 300 Kbps
- **11** 1Mbps

ビット2は、モデル30/CCRモードではNOPRECであり、機能はありません。その他のビットは未定義で、コントローラによって変化する可能性があります。他のレジスタと同様に、このレジスタに書き込めるようにルーチンを作成しました。

```
void flpydisk_write_ccr (uint8_t val) {.
    // ! コンフィグレーションコントロールを書
    き出します (FLPYDSK_CTRL, val) 。
}
```

コマンド

アブストラクト

コマンドは、FDCに接続されたFDDを制御して、読み出しや書き込みなどの異なる動作を行うために使用される。コマンドは、データバス（D0-D7）端子を介して、データレジスタに書き込み操作で書き込まれます（コントロールバスにはIOおよびWRITEコントロールラインが設定されています）。

警告コマンドやパラメータバイトを送信する前に、まず**MSR（Main Status Register）**のビット7をテストして、データレジスタがデータを受信する準備ができていることを確認してください。

13個（コントローラによってはそれ以上）のコマンドがあります。各コマンドのサイズは1〜9バイトです。FDCは、最初のコマンドバイトから何バイトを期待するかを知っている。つまり、最初のバイトは、FDCに何をしてほしいかを伝える実際のコマンドである。FDCは、このコマンドからさらに何バイトを期待するかを知っています（コマンドパラメータ）。

コマンドは、トラックの片方のヘッドでのみ動作します。両方のヘッドで動作させたい場合は、**Multiple Track Bit**を設定する必要があります。これらのコマンドの多くは、ビットフォーマットに従っています（後述します）。ここからが複雑なのです。

コマンドバイトは、下位4ビットのみを実際のコマンドに使用します（それ以上の場合もあります）。これらのコマンドバイトの上位ビットは、コマンドのオプション設定用です。私はこれを**拡張コマンドビット**と呼んでいますが、正式な名称はありません。これらのビットの中には、私たちが使用する必要のあるほとんどのコマンドに共通するものがいくつかあります。これらのビットについては、後ほどコマンドバイトの中で説明します。

さて、まずはコマンドリストを見てみましょう。次に、それぞれのコマンドを個別に見ていきます。どれもコマンドバイトの最初の4ビットしか使っていないことに注目してください。

```
enum FLPYDSK_CMD
{...
    fdc_cmd_read_track      = 2,
    fdc_cmd_specify         = 3,
    fdc_cmd_check_stat      = 4,
    fdc_cmd_write_sect      = 5,
    fdc_cmd_read_sect       = 6,
    fdc_cmd_calibrate       = 7,
    fdc_cmd_check_int       = 8,
    fdc_cmd_write_del_s     = 9,
    fdc_cmd_read_id_s       = 0xa,
    fdc_cmd_read_del_s      = 0xc,
    fdc_cmd_format_track    = 0xd,
    FDC_CMD_SEEK            = 0xf
};
```

FDCにコマンドを送るには、FIFOと呼ばれるデータレジスタにコマンドを書き込まなければならないことを覚えておいてください。そのためには、まず**MSR**のビット7をチェックしてデータレジスタの準備が整うのを待つ必要があります。**flpydisk_read_status()**はMSRからの値を返すだけなので、もっと簡単なメソッドの中にすべてを隠すことができます。

拡張コマンドビット

これらのコマンドの中には、レジスタの準備が整うまでデータを渡さなければならないものがあります。また、複数のバイトを返すものもあります。読みやすくするために、すべての500シフトフォーマット、パラメータのバイトを表にしました。各コマンドには、説明とサンプル・ルーチンが付いています。

さて、拡張コマンドビットの話をしたときに、上のコマンドが4ビットしかないことを覚えていませんか？この上位4ビットは、さまざまな用途に使用できます。

例えば、**Write Sector**コマンドのフォーマットは**M F 0 0 0 1 1 0**で、最初の4ビット（**0 1 1 0**）がコマンドバイト、上位4ビット（**M F 0 0**）が各種設定を表しています。**M**はマルチトラック、**F**はコマンドの動作密度モードを選択するための設定です。

ここでは、一般的なビットのリス

- トを紹介します。**M** - マル

チトラック・オペレーショ

ン

- 0 : シリンダーの片側のトラックで動作 1 : シリンダーの両側のトラックで動作
- F - FM/MFMモード設定
 - 0 : FM (単密度) モードで動作 1 : MFM (倍密度) モードで動作
- S - スキップモード設定
 - 0 : 削除済みデータのアドレスマークをスキップ
 - 1 : 削除済みデータのアドレスマークをスキップする
- HD - ヘッドナンバー
- DR0 - DR1 - ドライブ番号ビット (2ビットで最大4台のドライブに対応)

M、F、Sの各ビットは多くのコマンドに共通しているため、それらをまとめて表示することにしました。これらのビットを設定するには、これらの設定と使用したいコマンドをビット和するだけです。

```
enum FLPHYDSK_CMD_EXT {...
    fdc_cmd_ext_skip      = 0x20, //00100000
    fdc_cmd_ext_density   = 0x40, //01000000
    fdc_cmd_ext_multitrack = 0x80, //10000000
};
```

GAP 3

GAP3とは、物理ディスク上のセクタ間のスペースのこと。**GPL (Gap Length)** の一種です。

```
enum FLPHYDSK_GAP3_LENGTH {...
    flpydisk_gap3_length_std = 42,
    flpydisk_gap3_length_5_14 = 32,
    flpydisk_gap3_length_3_5 = 27
};
```

一部のコマンドでは、**GAP3**コードを通す必要があるため、そこは注意が必要です。)

セクターあたりのバイト数

一部のコマンドでは、セクタごとのバイト数を入力する必要があります。しかし、これらのバイト数は任意の大きさにすることはできず、常に計算式に従います。

```
2^n * 128, ここで ^ は "to power of" を表します。
```

n は0~7の数字です。 $2^7 * 128 = 16384$ (16Kバイト) なので、7以上にはなりません。FDCでは、1セクタあたり最大16Kバイトまで選択することが可能です。しかし、ほとんどのドライブはこれをサポートしていないかもしれません。

このリストには、最も一般的なものが含まれています。

```
enum FLPHYDSK_SECTOR_DTL {...
    flpydisk_sector_dtl_128 = 0,
    flpydisk_sector_dtl_256 = 1,
    flpydisk_sector_dtl_512 = 2,
    flpydisk_sector_dtl_1024 = 4
};
```

...だから、コマンドでセクタあたりのバイト数を要求されたら、512とは言わない！むしろ、FLPHYDSK_SECTOR_DTL_512、つまり2とする。

コマンドにパラメータを渡す方法

思い出せば、多くのコマンドではパラメーターを渡す必要があります。パラメーターを渡すには、コマンドが送られてきたのと同じ方法でパラメーターを送るだけです。例えば、**specify**コマンドでは、2つのパラメータを渡す必要があります。これがないとコマンドが起動しませんので...

```
flpydisk_send_command
(FDC_CMD_SPECIFY);
flpydisk_send_command (data);
flpydisk_send_command (data2);
```

それが全てです。)

コマンドから戻り値を取得する方法

プログラミングの関数のように戻り値を無視できるのではなく、FDCでは戻り値を何らかの形で処理する必要があります。もちろん、無視してもいいのですが、FDCから受け取る必要があります。それが終わるまでFDCはそれ以上のコマンドを許さない。

コマンドがデータを返す場合、そのデータはFIFO (データレジスタ) に--1つずつ--返されます。そのため、読むためには、FIFOを継続的に読み込んで、返されたデータをすべて取得する必要があります。

注 : コマンドがデータを返す場合は、待つ必要のある割り込みが送られてきます。これにより、コマンドが終了し、**FIFO**から戻り値を読んでも問題ないことを知ることができます。

戻り値の良い例として、**read sectors**コマンドがあります。このコマンドは、完了を知らせるために**IRQ**を待つ必要があり、7バイトを返します。そのため、返されたデータバイトをすべて読み出すには、データレジスタから**1**つずつ読み出す必要があります。

-
- - ◦
 -
 -
 -

```
for (int j=0; j<7; j++)
    flpydisk_read_data () です。
```

もちろん、エラーチェックのためには、実際にいくつかの戻り値を確認する必要があります。

ライトセクター

- 形式です。M F 0 0 0 1 1
- 0 Paramaters:
 - x x x x x HD DR DR0 シ
 - リンダー
 - ヘッド
 - セクター番号 セ
 - クターサイズ ト
 - ラック長 GAP3
 - の長さ データ長
- 戻る。
 - リターンバイト 0 :
 - ST0 リターンバイト
 - 1 : ST1 リターンバイト 2 : ST2
 - Return byte 3: 現在のシリンダー
 - Return byte 4: 現在のヘッド
 - Return byte 5: セクター番号
 - Return byte 6: セクターサイズ

このコマンドは、FDDから1セクタを読み出すものです。セクター内の1バイトごとに、FDCは割り込み6を発行し、ディスクから読み込んだバイトをデータレジスタに入れて、読み込めるようにします。

セクターを読む

- 形式です。M F S 0 0 1 1
- 0 Paramaters:
 - x x x x x HD DR1 DR0 = HD=ヘッド DR0/DR1=デ
 - ィスクシリンダー
 - ヘッド
 - セクター番号 セ
 - クターサイズ ト
 - ラック長 GAP3
 - の長さ データ長
- 戻る。
 - リターンバイト 0 :
 - ST0 リターンバイト
 - 1 : ST1 リターンバイト 2 : ST2
 - Return byte 3: 現在のシリンダー
 - Return byte 4: 現在のヘッド
 - Return byte 5: セクター番号
 - Return byte 6: セクターサイズ

このコマンドは、FDDから1セクタを読み出すものです。セクター内の1バイトごとに、FDCは割り込み6を発行し、ディスクから読み込んだバイトをデータレジスタに入れて、読み込めるようにします。

以下は、デモで使用したルーチンです。まず、DMAを設定して読み出し動作の準備をします。その後、リードセクターコマンド（FDC_CMD_READ_SECT）を実行し、コマンドM、F、Sの各ビットを設定します（マルチトラックリード、倍密度、削除されたアドレスマークのスキップ。これらの一覧は上記を参照してください）。

その後、すべてのコマンドパラメータを渡して、読み出しコマンドを開始します。セクターサイズはFLPYDSK_SECTOR_DTL_512（Bytes per sector）で、値は2です（詳細は上記のBytes per sectorの項を参照してください）。次のパラメータはGAP3の長さです。ここでは、標準的な3-1/2インチフロッピーディスクのGAP3長（FLPYDSK_GAP3_LENGTH_3_5、27）を渡します。

データ長パラメータバイトは、セクタサイズが0の場合のみ有効で、それ以外の場合は0xffとします。この

コマンドは完了後にIRQを送信するため、IRQを待つ必要があります。

```
void flpydisk_read_sector_imp (uint8_t head, uint8_t track, uint8_t sector)
{
    uint32_t st0, cyl;

    flpydisk_dma_read ()を使用してください。

    //! セクターの読み込み
    flpydisk_send_command (
        fdc_cmd_read_sect | fdc_cmd_ext_multitrack
        | fdc_cmd_ext_skip | fdc_cmd_ext_density) 。
    )
    flpydisk_send_command ( head << 2 | _CurrentDrive );
    flpydisk_send_command ( track );
    flpydisk_send_command ( head );
    flpydisk_send_command ( sector );
    flpydisk_send_command ( FLPYDSK_SECTOR_DTL_512
    ); flpydisk_send_command (
        ( ( sector + 1 ) >= FLPY_SECTORS_PER_TRACK )
        ?FLPY_SECTORS_PER_TRACK : sector + 1 )となっています。
    flpydisk_send_command ( FLPYDSK_GAP3_LENGTH_3_5 );
}
```

```

flpydisk_send_command ( 0xff );

// ! IRQを待つ
flpydisk_wait_irq ();

// ! ステータス情報の読み込み
for (int j=0; j<7; j++)
    flpydisk_read_data () です。

// ! 割り込みを処理したことをFDCに知らせる
flpydisk_check_int (&st0,&cyl);
}

```

...IRQが発火した後、7つのリターンバイトをすべて読み込みます。そして、flpydisk_check_int()でSENSE_INTERRUPTコマンドを送り、FDCに割り込みを処理したことを伝えます。(後述の「**インタラプトステータスの確認**」の項を参照してください)。

待つ...。データはどこにあるの？上のコマンドを見ると、FDCにデータをどこに置くかを指示していません。これは面白い問題だと思いませんか？

FDCの動作モードにもよりますが、Non-DMAモードでは、1バイトごとにIRQ6を発射します。ディスクから読み込んだデータのバイトはFIFOに入っています。DMAモードでは、データをDMAに渡し、DMAはそのデータをバッファ (DMAに置くように指示した場所) に入れます。

つまり、今回のケースでは、DMAバッファを0x1000に設定しましたよね？上記のルーチン呼び出し後、セクタデータは0x1000になります!かっこいいでしょう？DMAに別のアドレスを与えることで、その位置を変更することができます。

ドライブデータの修正/指定

- フォーマット 0 0 0 0 0 0
- 0 1 1 パラメーター
 - S S S S H H H H - S=ステップレート H=ヘッドアンロード時間
 - H H H H H NDM - H=Head Load Time NDM=0(DMA Mode)または1(DMA Mode)の場合
- Return: なし

このコマンドは、FDCに接続されているメカニカル・ドライブの制御情報をFDCに伝えるために使用されます。このコマンドの操作を簡単にするために、そのためのルーチンを書いてみましょう。

```

void flpydisk_drive_data (uint32_t stepr, uint32_t loadt, uint32_t unloadt, bool dma )
{
    uint32_t data = 0;

    flpydisk_send_command (FDC_CMD_SPECIFY) 。

    data = ( (stepr & 0xf) << 4 ) | (unloadt & 0xf); flpydisk_send_command (data);

    data = (loadt) << 1 | (dma==true) ? 1 : 0;
    flpydisk_send_command (data);
}

```

ステータスの確認

- フォーマット 0 0 0 0 0 1
- 0 0 パラメーター
 - x x x x x HD DR1 DR0 返
- します。
 - バイト 0 : ステータス・レジスタ 3

(ST3) このコマンドは、ドライブ・ステータス

を返します。

ドライブのキャリブレーション

- フォーマット 0 0 0 0 0 1
- 1 1 パラメーター
 - x x x x x 0 DR1 DR0 戻
- ります。なし

このコマンドは、リード/ライトヘッドをシリンダ 0 に配置するために使用します。完了後、FDC は割り込みを発行します。ディスクのトラック数が80以上の場合は、このコマンドを数回発行する必要があります。このコマンドを発行した後は、必ず正しいトラックにあるかどうかを確認してください (**Check Interrupt Status** コマンド)。

コマンドを実行した後、まだシリンダ0に到達していない場合は、再度コマンドを実行します。0番のシリンダーを見つけたら、モーターをオフにして成功を返します。10回やっても見つからなければ中止します。

このコマンドを実行する際には、モーターが動作していることを確認する必要があることに注意してください。また、SENSE_INTERRUPTコマンド (flpydisk_check_int()コール) を使って、現在のシリンダーの状態を確認していることにも注目してください。

```

int flpydisk_calibrate (uint32_t drive) {
    uint32_t st0, cyl;

    if (drive >= 4)
        return -2;

    // ! モーターをオンにする
}

```

```

flpydsk_control_motor (true);

for (int i = 0; i < 10; i++) {
    。

    // ! コマンド送信
    flpydsk_send_command ( FDC_CMD_CALIBRATE
    ); flpydsk_send_command ( drive );
    flpydsk_wait_irq ();
    flpydsk_check_int ( &st0, &cyl);

    // ! シリンダー0は大丈夫だったのか? もしそうなら、私た
    ちは終わりです if (!cyl) {}。

    flpydsk_control_motor
    (false); return 0;
}

}

flpydsk_control_motor
(false); return -1;
}

```

割り込みステータスの確認

- フォーマット000010
- 00 パラメーターNone
- Return:
 - バイト0: ステータスレジスタ0 (
 - ST0) バイト1: 現在のシリンダー

割り込み復帰時のFDCの状態の情報を確認するためのコマンドです。

```

void flpydsk_check_int (uint32_t* st0, uint32_t* cyl) {

    flpydsk_send_command (FDC_CMD_CHECK_INT);

    *st0 = flpydsk_read_data ()です。
    *cyl = flpydsk_read_data ()です。
}

```

シーク/パークヘッド

- フォーマット000011
- 11 パラメーター
 - x x x x x HD DR1 DR0 - HD=ヘッド DR1/DR0=ドラ
 - イブシリンダー
- 戻る。なし

このコマンドは、リード/ライトヘッドを特定のシリンダーに移動させるために使用します。**calibrate** コマンドと同様に、このコマンドを複数回送信する必要があります。試行錯誤のたびに **check_int ()** を呼び出して現在のシリンダーを取得することに注目してください。その後、現在のシリンダーが探しているシリンダーであるかどうかをテストします。もしそうでなければ、もう一度試します。もしそうであれば、成功を返します。

無効なコマンド

無効なコマンドがFDCに送られてきた場合、FDCはそれを無視してスタンディモードになります。

FDCのリセット

```

if (_currentDrive >= 4)
    1を返す。

for (int i = 0; i < 10; i++ ) {}。

// ! コマンドを送る
flpydsk_send_command (FDC_CMD_SEEK);
flpydsk_send_command ( (head) << 2 | _CurrentDrive);
flpydsk_send_command (cyl);

// ! 結果フェーズのIRQを待つ
flpydsk_wait_irq ();
flpydsk_check_int (&st0,&cyl0);

// ! シリンダーを見つけたか
? if ( cyl0 == cyl )
    0を返す。
}

1を返す。
}

```

コントローラーの無効化

DOR RESETラインがLowの場合、コントローラはディセーブル状態になります。つまり、DORレジスタに0を書き込むだけで、コントローラをディセーブルにすることができます。

```
void flpydisk_disable_controller () {...
    flpydisk_write_dor (0) です。
}
```

コントローラーの有効化

コントローラを有効にするには、DORでRESETラインをハイにします。また、FDCをDMAモードで動作させたいので、DORでそのビットも設定する必要があります。

```
void flpydisk_enable_controller () {...
    flpydisk_write_dor ( FLPYDSK_DOR_MASK_RESET | FLPYDSK_DOR_MASK_DMA );
}
```

コントローラーが無効化された後に有効化されると、割り込みが発生します。この間に、コントローラやドライブの設定を再初期化する必要があります。

FDCの初期化

コントローラのリセット時には、コントローラを再初期化する必要があります。コントローラをリセットすると、IRQ 6が起動します。IRQ 6が発行された後、FDCに接続されているすべてのドライブにSENSE_INTERRUPTコマンドを送信する必要があります(**flpydisk_check_int**を4回呼び出します)。

その後、コントローラの再設定を行います。**CCR(Configuration Control Register)**には、データレートを設定するビットが2つしかないことを覚えておいてください。両方とも0にすることで、データレートを500Kbpsに設定します。これはデフォルトの値としては良いと思います。

次に **flpydisk_drive_data** を呼び出し、**Fix Drive Data / Specify** コマンドをコントローラに送信して、以下のようなドライブの機械的情報を設定します。ステップレート、ヘッドのロードとアンロードの時間、DMAモードをサポートしているかどうかなど、ドライブの機械的な情報を設定します。

そして、0番シリンダーになるようにドライブを再調整します。

```
void flpydisk_reset () {
    uint32_t st0, cyl;

    //! コントローラをリセットする
    flpydisk_disable_controller
    (); flpydisk_enable_controller
    (); flpydisk_wait_irq ();

    //! CHECK_INT/SENSE INTERRUPTコマンドを全ドライブに送信 for
    (int i=0; i<4; i++)
        flpydisk_check_int (&st0,&cyl) です。

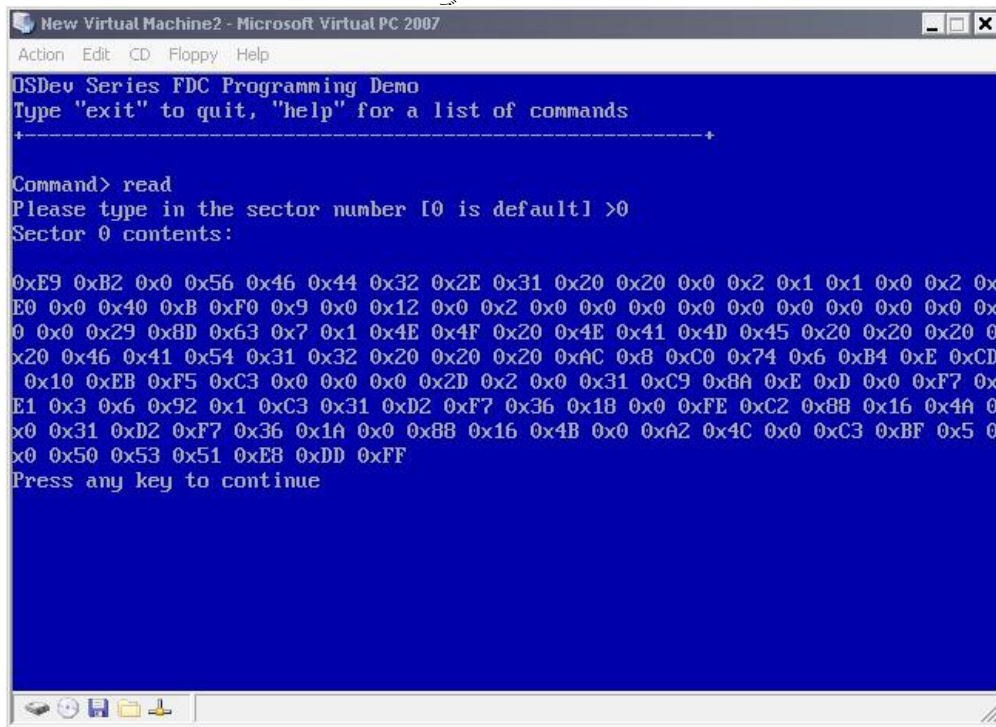
    //! 転送速度 500kb/s
    flpydisk_write_ccr (0);

    steprate=3ms, unload time=240ms, load time=16ms flpydisk_drive_data (3,16,240,true);

    //! ディスクのキャリブレーションを行う
    flpydisk_calibrate ( _currentDrive
    );
}
```

リセットされた後、ドライブは私たちが使用できる状態になります。

デモ



FDCデモの様子
デモダウンロード

注：このデモには、**VPC**が最初に読み込んだセクターしか読み込まないという既知のバグがあります。これはできるだけ早く解決される予定です。Bochsで実行した場合、既知の問題はありません。

アップデートと変更点

文字列から整数への変換 - `stdio.h/stdio.cpp`

このデモをよりインタラクティブなものにするために、文字列を整数に変換するのに使われる3つの関数を標準ライブラリに入れました。これには、**strtol**、**strtoul**、**atoi**が含まれます。このデモでは**atoi**を使って、ユーザーから入力された文字列を使用可能な整数に変換しています。

フロッピードライブのインストール - `flpydisk.cpp`

フロッピードライブには、デモで簡単に設定できるように、優れたインストールルーチンが用意されています。HALの**setvect()**ルーチンを使って割り込みハンドラをインストールし、転送用のDMAを初期化し、コントローラをリセットして使用可能な状態にしているだけです。

```
void flpydisk_install (int irq)
{
    // ! irqハンドラのインストール
    setvect (irq, i86_flpy_irq)
    .

    // ! FDCのDMAを初期化する
    flpydisk_initialize_dma ();

    // ! fdcをリセット
    する
    flpydisk_reset ();

    // ! ドライブ情報の設定 flpydisk_drive_data
    (13, 1, 0xf, true);
}
```

デモでは、初期化時にこの関数を呼び出し、ドライバーからの読み取りを試みる前にドライバーを設定します。

任意のセクタの読み取り - LBAとCHS - `flpydisk.cpp`

ドライブは、CHSの詳細を2つの優れた関数の後ろに隠しています。ドライブはCHS(Cylinder/Head/Sector)で動作し、LBA(Linear Block Addressing)については何も知らないことを知っているのので、この2つの間で変換するルーチンを提供しなければなりません。これにより、物理的なCHSを気にすることなく、セクタ番号を渡して読み書きすることができます。

LBAをCHSに変換する公式を覚えていますか？ここではそれを応用してみましょう。

```
void flpydisk_lba_to_chs (int lba,int *head,int *track,int *sector) {
    *head = ( lba % ( FLPHY_SECTORS_PER_TRACK * 2 ) ) / ( flpy_sectors_per_track );
    *track = lba / ( FLPHY_SECTORS_PER_TRACK * 2 ) となります。
    *sector = lba % FLPHY_SECTORS_PER_TRACK + 1;
}
```


2021/11/15 13:10

るようになりました。かっこいいでしょう？

オペレーティングシステム開発シリーズ

ズ

ディスクから任意のセクタを読み込めるようにしたいので、そのためのルーチンを用意します。また、すでに **flpydisk_read_sector_imp** は、読み取りコマンドをコントローラに送信するコードを含んでおり、このルーチンは非常にシンプルです。

```
uint8_t* flpydisk_read_sector (int sectorLBA) {
    if (_currentDrive >= 4) return 0;

    /// LBAセクタをCHSに変換 int
    head=0, track=0, sector=1;
    flpydisk_lba_to_chs (sectorLBA, &head, &track, &sector);

    /// モーターをオンにして、追従を求める
    flpydisk_control_motor (true);
    if (flpydisk_seek (track, head) != 0)
        return 0;

    /// セクタを読み込んでモーターをオフにする
    flpydisk_read_sector_imp (head, track, sector); flpydisk_control_motor (false);

    /// 警告：これは少しハック的なものです
    return (uint8_t*) DMA_BUFFER;
}
```

デモがセクタを読みたいときはいつでも、このルーチン呼び出します。このルーチンは、セクタをディスク上の物理的な位置(CHS)に変換します。モーターをオンにして、このセクタがあるシリンダーにシークします。その後、**flpydisk_read_sector_imp** を呼び出して、実際にセクタを読み出すマジックを行い、その後、モーターをオフにします。

flpydisk_read_sector_imp コールの後、セクタのデータはDMAバッファにあるはず。このバッファへのポインタを返します。このバッファには、先ほど読み込まれたセクタ・データが入っています。かっこいいでしょう？

これは、すべてを結びつける魔法のルーティンです。)

新規読み取りコマンド - main.cpp

このデモは、前回のデモをベースにしています。そのため、前回のデモで構築されたコマンドラインインターフェース (CLI) をそのまま使用しています。これにより、このデモは今までで最も複雑なデモにもなっています。

CLIのコマンドリストに新しいコマンド「**read**」を追加しました。このコマンドでは、ディスクから任意のセクタを読み取ることができます。これは、このチュートリアルで作成されたフロッピードライバを使用して行います。

このコマンドは関数の中にあり、デモでは**readと入力**して実行します。これは**512バイトを4つの128バイトのブロックに分けて読みやすくダンプ**します。各ブロックが終わると、次のチャンクに進むためにキーを押すように促されます。新しい**atoi**関数を使って、入力されたセクタ番号 (LBAセクタ番号) を**int**に変換し、それを読み込みます。読者の皆さん、これが魔法を起こす関数です。

結論

```
void cmd_read_sect () {...
    uint32_t sectornum = 0;
    char sectornumbuf[4];
    uint8_t* sector = 0;

    DebugPrintf ("0 is default] >"); get_cmd (sectornumbuf, 3);
    sectornum = atoi (sectornumbuf);

    DebugPrintf ("%u", sectornum);

    /// ディスクからセクタを読み込む
    sector = flpydisk_read_sector (sectornum);

    セクタの表示 if (sector!=0) {...
        int i = 0;
        for ( int c = 0; c < 4; c++ ) {
            for (int j = 0; j < 128; j++)
                DebugPrintf ("0x%x ", sector[ i + j ]);
            i += 128;

            DebugPrintf("Press any key to continue\\"); getch ();
        }
        その他
        DebugPrintf ("Error reading sector from disk")と表示されます。

        DebugPrintf ("\\Done.");
    }
}
```

今回のチュートリアルでは、DMAについて説明します。DMAをプログラミングするためのインターフェイスを作成し、FDCドライバの中でそれをうまく利用していきます。この後は、またファイルシステムの話ですね。心配しないでください - この後はマルチタスクです!

次の機会まで。

マイク

BrokenThorn Entertainment社。現在、DoEと[NeptuneOperating System](#)を開発中です。質問や

コメントはありますか?お気軽に[お問い合わせください](#)。

あなたも記事の改善に貢献したいと思いませんか?もしそうなら、ぜひ[私に教えてください](#)。



第19章

[ホーム](#)



オペレーティングシステム開発シリーズ

オペレーティング・システムの開発 - 8237A ISA DMAC

by Mike, 2009

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

注：今後、デモ名は「**Demo00**」（00は章名）という形式で表記します。これは、デモの名前と章の名前が関連しないという問題を解決するためと、読者が特定のデモがどの章を参照しているかを簡単に知ることができるようにするためです。古いチャプターもこの設定で更新されます。すべてのチャプターが更新されたら、このコメントは削除されます。

また、**Virtual PC**のバグは、本章では修正されていますが、前章ではまだ修正されていません。前の章のデモはVirtual PCでうまく動作しませんでした。この章のデモはDMAコードのマイナーなバグフィックスとアップデートでうまく動作するようになりました。このデモはVirtual PCとBochsでテストされ、動作しました。前の章がこの修正で更新されたら、このコメントは削除されます。

はじめに

歓迎します。:)

この章では、**DMAC (Direct Memory Access Controller)** について詳しくご紹介します。DMACは、ソフトウェアを使わずに、デバイスからのデータブロックを直接メモリに転送する方法を提供します。これにより、ソフトウェアではなくハードウェアがデータを転送するため、非常に高速なデータ転送が可能になります。

今日のリストはこちらです。

- DMAの歴史
- DMAハードウェア
- ア DMAポート
- DMAレジスター
- DMAコマンド

アブストラクト

ダイレクトメモリアクセス (DMA) とは、最近のコンピュータに搭載されている機能で、プロセッサとのやりとりなしに大きなデータブロックを移動させることができるデバイスです。この機能は、フロッピーディスクのプログラミングの章でご紹介したように、非常に便利です。デバイスがデータブロックを転送している間、プロセッサはデータがメモリや他のデバイスに転送されているかどうかを気にすることなく、ソフトウェアの実行を続けることができます。基本的な考え方は、DMAデバイスが自分でタスクを実行するようにスケジュールすることです。カッコいいでしょう？

バスやアーキテクチャの設計によって、ダイレクトメモリアクセスを実行する方法は異なります。今回は**ISA Direct Memory Access Controller (DMAC)**に焦点を当てますが、念のため他の方式についても取り上げることにしました。

ISA

ISA (Industry Standard Architecture) では、**インテル8237マイクロコントローラ**をベースにしたコントローラを介して、DMAリクエストを集中的に行うことができます。ATXマザーボードの設計では、コントローラは1つしかありませんでした。しかし、ATX以降のアーキテクチャでは、コントローラは**2つ**あります。これは、**PIC (Programmable Interrupt Controllers)** がスレーブ化されているのと同じです。両方のコントローラは常に**4MHz**で動作します。

性能やデバイスの数が限られているため、新しいデバイスではPIOやUDMAを代わりに使う傾向があります。しかし、レガシーデバイスではISAでDMAがサポートされています。

これらのデバイスはすべて、コントローラの**チャンネル**に接続されている。これらのチャンネルとともに、各チャンネルには**DACK (DMA Acknowledge)** ラインと**DRQ (DMA Request)** があります。

ここでは、両**DMAC (Direct Memory Access Controllers)** の標準的な割り当てを紹介します。

- XTです。
 - **チャンネル0** : システムで使用、使用不可 (DRAM Refresh、廃止予定)
 - **チャンネル1** : 使用可能、標準的なDMA割り当てなし
 - **チャンネル2** : フロッピーディスクコントローラ
 - **チャンネル3** : ハードディスク・コントローラ (PIOまたはUDMAを推奨)
- ATのみ。
 - **チャンネル4** : XTコントローラにカスケード接続 - スレーブDMAコントローラをマスターに入力
 - **チャンネル5** : 使用可能、標準的なDMA割り当てなし (16ビット)
 - **チャンネル6** : 使用可能、標準的なDMA割り当てなし (16ビット)

- **チャンネル7** : 使用可能、標準的なDMA割り当てなし (16ビット)

転送を開始するために、ソフトウェアはチャンネルアドレスとカウントレジスタに、物理メモリ内の転送を完了する場所と転送サイズを設定します。その後、そのメモリブロックからの読み出しまたは書き込みを設定し、コントローラを転送完了に向けて動作させます。転送が完了すると、転送を開始したデバイスから**IRQ (Interrupt Request)** が発行され、システムソフトウェアがそれをキャッチして処理を行います。これは重要なことです。以上が、DMAを使って転送を開始する際に必要となる手順です。

PCI

PCIデバイスは同じDMAコントローラを共有しておらず、中央にDMAコントローラがあるわけでもない。代わりに、**PCIローカルバス上**のPCIデバイスが、**PCIバスコントローラ**に**バスマスター**になることを要求する (バスをコントロールする)。その後、物理メモリへの読み書きのリクエストがノースブリッジに渡され、ノースブリッジはそのリクエストをメモリ操作に変換して**メモリコントローラ**に送ります。

PCIの転送は4GBの物理メモリに制限されています。しかし、デバイスとPCIブリッジが**Double Address Cycle (DAC)**または同様の技術を実装している場合、PCIコントローラが4GBの物理メモリを超えて読み書きのリクエストを開始することができます。

ISA DMAハードウェア

ダイレクトメモリアクセスコントローラ (DMAC)

ISA (Industry Standard Architecture) では、オリジナルの**Intel 8237 DMA**チップをベースにしたコントローラを使用しています。最新のDMACはより多くの機能を備えていますが、ほとんどが**8237マイクロコントローラ**との互換性を持っています。新しいPCにはより高度なDMACが搭載されていますが、すべての始まりとなったデバイスを見ることはいつでも素晴らしいことです。そこで、**デュアル・インライン・パッケージ (DIP)** で配布されているオリジナルの**8237A**コントローラのピンダイアグラムをご紹介します。

IOR	1	40	A7
IOW	2	39	A6
MEMR	3	38	A5
MEMW	4	37	A4
*	5	36	EOP
READY	6	35	A3
HACK	7	34	A2
ADSTB	8	33	A1
AEN	9	32	A0
HREQ	10	31	VCC (+5V)
CS	11	30	DB0
CLK	12	29	DB1
RESET	13	28	DB2
DACK2	14	27	DB3
DACK3	15	26	DB4
DREQ3	16	25	DACK0
DREQ2	17	24	DACK1
DREQ1	18	23	DB5
DREQ0	19	22	DB6
GND/Vss	20	21	DB7

これが、この章でプログラミングすることになるコントローラです。たくさんのピンがありますが、それほど複雑ではありません。重要なものを中心に見ていきましょう。

- **Pin 1 (IOR)** I/O Read
- **Pin 2 (IOW)** I/O Write
- **Pin 3 (MEMR)** メモリリード
- **Pin 4 (MEMW)** メモリライト
- **Pin 5**
- **ピン6 (READY)**
- **Pin 7 (HACK)** Hold
- Acknowledge **Pin 8 (ADSTB)**
- Address Strobe **Pin 9 (AEN)**
Address Enable
- **Pin 10 (HREQ)** ホールドリクエスト
- **Pin 11 (CS)** チップセレクト
- **Pin 12 (CLK)** クロック
- **端子13 (RESET)** リセット
- **ピン14-15 (DACK)** DMA アカウレッジ
- **ピン16-19 (DREQ0-DREQ3)** DMAリクエスト
- **Pin20 (GND/Vss)** グランド
- **ピン21-23 (DB0-DB3)** データバス ビ
- **ン24-25 (DACK)** DMA Acknowledge ビ
- **ン26-30 (DB4-DB7)** データバス ピン
- **31 (Vcc)** +5V電源
- **32-35番ピン (A0-A3)** アドレスライン
- **Pin 36 (EOP)** End Of Process
- **Pins 37-40 (A4-A7)** アドレスライン

これは悪くないですね。**20番ピン**がグランド、**31番ピン**が電源です。**12番ピン (CLK)** は、どのコントローラにも共通しているものです。プロセッサのCLK端子に接続し、コントローラ内の動作タイミングを制御するクロック信号を入力します。**CS (チップセレクト)**) 端子も、ほとんどのコントローラに共通して見られるものだ。データバス上のI/Oデバイスとしてコントローラを選択するために使用

されます。**RESET**は、コントローラの内部レジスタ (Status, プ

リクエスト、テンポラリー、コマンド)、内部のフリップフロップをクリアし、マスクレジスタを設定します。ここまでは、特に目新しいことはありませんね。システムアドレスバスに接続するゲルネリックアドレスライン**A0~A7**があります。入力時には、CPUは**A0-A3**にデータを書き込み、読み出すレジスタを選択することしかできません。すべてのピンは(物理的なメモリアドレスへの)出力に使用されますが、DMAリクエスト時にのみ有効になります。最後に、システムデータバスに接続する汎用の**D0-D7**ピンです。

次に、より興味深いピンについて説明します。ここまでの説明で、DMACはシステムのアドレスとデータバスに接続していることがわかりました。読者の多くは、このことにあまり驚かないでしょう。しかし、ご想像のとおり、DMACはCPUからの直接の注意を必要とします。そのため、DMACがCPUと通信できるように、CPUに接続するラインをいくつか用意しており、その逆も可能です。これは、**HACK**および**HREQ**ピンで行われます。**HACK (Hold Acknowledge)**は、CPUがDMACにシステムバスの完全な制御権を与えたときにHighに保たれます。これによりDMACは、メモリコントローラにデータを送信しても大丈夫なタイミングを知ることができます。結局のところ、DMACとプロセッサの両方が同じシステムバスを同時に使用することはできないのだ。

DMACは、システムバスがプロセッサによって使用されていない場合に限り、物理メモリに直接データを転送する。DMACがシステムバスを必要とするのは、メモリコントローラにデータを送信し、メモリ変換や物理メモリへの読み書きを行うためである。

なるほど、CPUはDMACにシステムバスの乗っ取りを伝える方法を持っているのですね。しかし、そもそもDMACはどうやってCPUにシステムバスの必要性を伝えるのだろうか？それが、**HREQ (Hold Request)**ラインである。DMAに接続されたデバイス(フロッピーコントローラなど)から**DMAリクエスト (DRQ)**がトリガーされ、その「チャンネル」が現在無効になっていない場合、コントローラは次のクロックサイクルでHREQをハイレベルにして、リクエストを完了するためにシステムバスのコントロールが必要であることをCPUに通知する。

DR0~DR3 (DMAリクエストライン)は、デバイスがDMAにリクエストを通知するために使用するラインです。例えば、**フロッピー・ドライブ・コントローラ (FDC)**は、通常、**DR2**(「チャンネル2」)を使用するように接続されています。そこで、そのチャンネルを有効にして、FDCがDMACを使用するようにプログラムすると、FDCにリードまたはライトコマンドが送られると、FDCは**RQ2**ラインをアクティブにして、DMACに注意が必要であることを通知する。ここからは、そのチャンネルをプログラムしたモードに応じて、DMACを介してすべての読み取りまたは書き込みが行われます。

ここで、もうひとつ重要なポイントがあります。DRQラインが4本しかないことを知っていること。**1台のDMACに接続できる機器は4台まで。**これはかなり限定的ですね。i86アーキテクチャでは、2つのDMACをくっつけることで、この問題をある程度解決しています。ここでは、その方法をご紹介します。

これまでのところ、すべてが順調に進んでいます。ソフトウェアがCPUにDMACのプログラムを指示することはわかったが、CPUはDMACにレジスタの読み書きが必要であることをどのように伝えるのだろうか？それが**IOR (I/O Read)**と**IOW (I/O Write)**のピンです。同様に、DMACはメモリコントローラに対して、**MEMR (Memory Read)**や**MEMW (Memory Write)**を出力して、メモリの読み出しや書き込みの制御線をアクティブにすることで、読み出しや書き込みを行うことを伝えている。**EOP (End Of Process)**は、リクエストが完了したときにデバイスに信号を送るためのものです。リクエストが完了するのは、そのチャンネルの**ターミナルカウント (TC)**に到達したときです。これはプログラム可能なカウンタ値です。**AEN (Address Enable)**は、コントローラが内部の8ビットアドレスラッチレジスタをシステムアドレスバスにロードすることを通知するために使用されます。**ADSTB(Address Strobe)**は、上位アドレスバイトを外ラッチレジスタにストロブするために使用します。

いろいろあるんですね。コントローラが行う操作の詳細は、モードや転送タイプによって異なります。しかし、基本的な手順は同じです。デバイスがDMACに通知し、DMACがシステムバスを介してCPUに制御を通知する。DMACは制御を待ちます。DMACは制御を待ち、制御を受けるとチャンネル・アドレス・レジスタを内部ラッチ・レジスタにロードします。そこから、MEMR、MEMWを設定し、必要に応じてメモリの読み書きを行うことになる。待てよ、何だって？メモリからの読み出しはわかると思いますが、書き込みをしたらどこに行くのでしょうか？

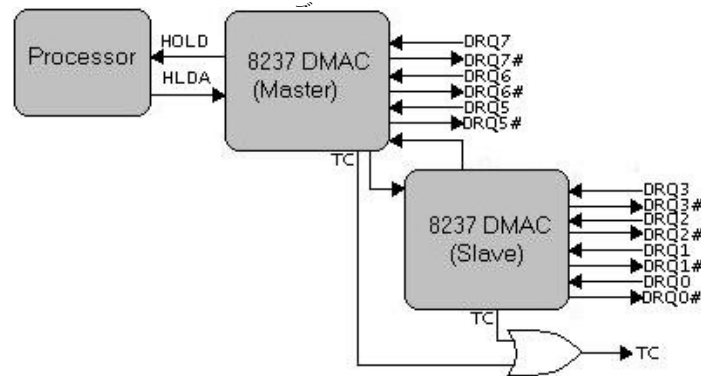
もう一度**FDCの章**を見てください。DMACやCPU、メモリーコントローラなどと同じデータバスに接続するピン**D0~D7**があることに注目してください。つまり、メモリからの書き込み時に必要なのは、MEMRラインをアクティブにしてアドレスバスにアドレスをアップロードし、メモリコントローラがデータを変換してデータバスに載せることだけなのだ。FDCは書き込み要求を待っているのだから、読み込んだデータを取り込み、FDCに送られてきた書き込みコマンドで設定されたディスクに書き込む。ディスクからの読み出しの場合も基本的には同じですが、DMACは代わりにMEMWラインをアクティブにします。メモリコントローラは、FDCから送られてきたデータバスから書き込むべきデータを取り込みます。すべてが順調に進むと、DMACはプロセッサの**HREQ**ラインを解放し、CPUが再びバスを完全に制御できるようになります。

ここで重要なのは、プロセッサはDMACの終了を待つことはできないということです。CPUは、システムバスへのアクセスが必要になると、**HACK**ラインをローレベルにします。この間、DMACは処理を続けるためにHACKラインが再びハイレベルになるまで待つしかありません。

読者の皆さん、こんにちは。i86では、DMACをもう1つ追加して、使用できるチャンネル数を8にした。まあ、そんな感じです。それでは見てみましょう。

x86のDMAC

最近のPCには2つのDMACが搭載されているのを覚えていますか？両方のDMACは、2つのPICが接続されているのと同じような方法で接続されています...ただ、逆方向です。ええっ！？そうなんですよ。)では、見てみましょう。



DMACは、ISAバスの制御を行う際に、プロセッサの**HOLD**および**HLDA (Hold Acknowledge)** ピンを使用する。DMACはHOLDを通じてプロセッサに信号を送り、プロセッサはHLDAを通じてこの要求を承認する。また、プライマリDMACがDRQの5～7を持っているのに対し、セカンド（スレーブ）にはDRQの0～3が含まれていることにも注目してほしい。DRQとは、**DMAリクエスト**のことである。これらのラインは、DMACを使用するシステム内のさまざまなデバイスに接続する。デバイスがDMACの使用を要求すると、ラインをハイレベルにしてDMACに信号を送ります。画像をもう一度見てみると、面白いことがわかるかもしれません。**DRQ4はどこにある？**

DRQ4は両方のデバイスに存在しますが、それぞれのDMACを接続するものです。それらは画像に示されています（ラベルは付いていません）。DRQラインはDMACに信号を送るために使われるので、これによってプライマリDMACとセカンダリDMACがお互いに正しいDRQラインを上げるように信号を送ることができる。つまり、**DMACをプログラミングする際には、DRQ4がプライマリコントローラとスレーブコントローラを接続するために使用されていることを忘れてはならないのだ。そのため、DRQ4は使用できません。**上の図を振り返ると、プライマリDMACまたはスレーブDMACのいずれかが完了した（**TC (Terminal Count)** ラインを上げた）場合にtrueを出力する**ORゲート**も見られる。TCラインは、DMACに送信された転送要求が完了したときに上がります。

さて、ここで覚えておかなければならない重要なことを、参考のためにまとめておきましょう。

- DMAは常に物理メモリ上で動作し、仮想メモリ上では動作しない
- i86アーキテクチャのDMACを使用するには、8台のデバイスしか接続できません。
- DRQ4（チャンネル4）は、プライマリDMACとセカンダリDMACの接続に使用されており、使用することはできません。

また、これらがどのように構成されているかについても興味深いものがあります。スレーブDMACは、マスターDMACに接続する**最初の**DMACであり、その逆**ではないのです**。スレーブDMACがチャンネル0～3（厳密にはプライマリDMACへの接続に使われる4も）を担当し、プライマリDMACがチャンネル5～7を担当するのはこのためです。なんだか変ですよね？このように、2つのPICと一緒に動作する方法とは多少異なります。また、これらのコントローラが接続されているため、**マスターDMACは16ビットDMACのように動作し、スレーブDMACは8ビットDMACのように動作することにも注意が必要です。**これが原因です。

- **1台目のDMACがスレーブ（8ビット）、2台目がマスター（16ビット）。**

ISA DMAインターフェース

ポートマッピング

DMAコントローラが2つあるので、ポートも2組あります。

汎用レジスター

ISA DMACポート		
DMAC 0ポート（スレーブ）	DMAC 1ポート（マスター）	ディスクリプション
0x08	0xD0	ステータスレジスタ（リード）
0x08	0xD0	コマンドレジスタ（ライト）
0x09	0xD2	リクエストレジスタ（ライト）
0x0A	0xD4	シングルマスクレジスタ（ライト）
0x0B	0xD6	モードレジスタ（ライト）
0x0C	0xD8	バイトポインタフリップフロップのクリア (Write)
0x0D	0xDA	中間レジスタ（リード）
0x0D	0xDA	マスタークリア（ライト）
0x0E	0xDC	クリアマスクレジスタ（ライト）
0x0F	0xDE	ライトマスクレジスタ（ライト）

これらのレジスタについては、次のセクションで詳しく説明します。これらのレジスタは、両方のDMACを操作する際に使用されます。これらのレジスタは、ポートマップドI/Oを通して読み書きすることができます。つまり、標準的なi86の**in**および**out**命令を使用します。

2021/11/15 13:10

オペレーティングシステム開発シリー

異なることにも注意してください。スレーブは**8**ビット、マスターは**16**ビットであることを覚えていますか？

読みやすくするために、これらの醜い数字を列挙して抽象化してみましょう。

```
enum DMA0_IO {  
  
    DMA0_STATUS_REG = 0x08,  
    DMA0_COMMAND_REG = 0x08,  
    DMA0_REQUEST_REG = 0x09,  
    DMA0_CHANMASK_REG = 0x0a,  
    DMA0_MODE_REG =  
    0x0b, DMA0_CLEARBYTE_FLIPFLOP_REG =  
    0x0c, DMA0_TEMP_REG = 0x0d,  
    DMA0_MASTER_CLEAR_REG = 0x0d,  
    DMA0_CLEAR_MASK_REG = 0x0e,  
    DMA0_MASK_REG = 0x0f  
  
};
```

これらの値が上の表と一致していることに注目してください。次に**DMAC 2**について。

それで**DMA0_IO**をご覧ください。

チャンネルレジスタ

DMAC 1は、上記のレジスタに加えて、各チャンネルのアドレスやカウンタを制御するための以下のレジスタが用意されています。

```
DMA1_STATUS_REG = 0xD0,  
DMA1_COMMAND_REG = 0xD0,  
DMA1_REQUEST_REG = 0xD2,  
DMA1_CHANMASK_REG = 0xD4,  
DMA1_MODE_REG = 0xD6,  
DMA1_CLEARBYTE_FLIPFLOP_REG = 0xD8,  
DMA1_TEMP_REG = 0xDA,  
DMA1_UNMASK_ALL_REG = 0xDC,  
DMA1_MASK_REG = 0xDE
```

ISA DMACチャンネルポート

ISA DMACチャンネルポート		
DMAC 0ポート (マスター)	DMAC 1ポート (マスター)	ディスクリプション
0x0	0xC0	チャンネル0アドレス/チャンネル4アドレス
0x1	0xC2	チャンネル0カウンタ/チャンネル4カウンタ
0x2	0xC4	チャンネル1アドレス/チャンネル5アドレス
0x3	0xC6	チャンネル1カウンタ/チャンネル5カウンタ
0x4	0xC8	チャンネル2アドレス/チャンネル6アドレス
0x5	0xCA	チャンネル2カウンタ/チャンネル6カウンタ
0x6	0xCC	チャンネル3アドレス/チャンネル7アドレス
0x7	0xCE	チャンネル3カウンタ/チャンネル7カウンタ

上の表をもう一度見てください。マスターDMACのチャンネル0のアドレスは.....えっ、ポート0！？このシリーズでは、i/oポート0を見つけたことで、歴史的な瞬間を迎えます。)

また、マスターDMACが16ビットであるのに対し、スレーブDMACは8ビットであることも覚えているだろうか。これは重要な特徴で、特にここでは、スレーブDMACには8ビットの値を、マスターDMACには16ビットの値を読み書きできることを意味しています。

さて、これらのレジスタの詳細を説明する前に、まずこれらを隠すことにしましょう。下の列挙を見ると、派手なことは何も行われていないことがわかるでしょう。これらのレジスタはすべてポートマップドI/Oでアクセスされることを覚えておいてください。言い換えれば、x86マシン命令のin/outを使って読み書きすることができます。

```
enum DMA0_CHANNEL_IO {...  
  
    DMA0_CHAN0_ADDR_REG = 0, //!<その通り、i/oポート0  
    DMA0_CHAN0_COUNT_REG = 1です。  
    dma0_chan1_addr_reg = 2,  
    dma0_chan1_count_reg = 3,  
    dma0_chan2_addr_reg = 4,  
    dma0_chan2_count_reg = 5,  
    dma0_chan3_addr_reg = 6,  
    dma0_chan3_count_reg = 7,  
  
};
```

...そしていよいよDMAC2へ....

これらのDMACの基本的な目的は、..チャンネルをどのように開始するかをDMACに伝える方法を提供することです。各チャンネルには、ベースアドレスとカウンタがあります。ベースアドレスは、読み書きを開始するメモリ上の位置で、カウンタは、そのチャンネルでどれだけ転送するかをDMACに伝えるREGで重要なのは、これらが仮想ではなく常に物理的なアドレスであるということです。

例を挙げてみましょうDMA1CHAN5 ADDRが使用する0xc4ベースアドレスを設定するには、上記の表に示されている正しい i/o ポートに書き込みます。DMA0_CHAN0_ADDR_REGが0で、DMA1_CHAN7_ADDR_REGが表の最後の値(0xcde)だとするとこれは単にCHAN6すべてのサンプルコードは、この章の最後にあるデモに含まれています。

よく似た方法でDMA1CHAN7_ADDR_REGを0xc4と置き換えて、DMAC1のアドレス4078とチャンネルを指定するlow, highを書き込むことができます。

ここで重要なのは、これらのレジスタが16ビットであるということです。DMACが1つのチャンネルから一度に転送できるのは最大でも64kであることを意味しています。

```

unsigned short port = 0;
switch (channel) {

```

```

    case 0: port = DMA0_CHAN0_ADDR_REG; break;
    case 1: port = DMA0_CHAN1_ADDR_REG; break;
    case 2: port = DMA0_CHAN2_ADDR_REG; break;
    case 3: port = DMA0_CHAN3_ADDR_REG; break;
    case 4: port = DMA0_CHAN4_ADDR_REG; break;
    case 5: port = DMA0_CHAN5_ADDR_REG; break;
    case 6: port = DMA0_CHAN6_ADDR_REG; break;
    case 7: port = DMA1_CHAN7_ADDR_REG; break;
    default: port = DMA1_CHAN6_COUNT_REG; break;
}
outputb(port, low);
outputb(port, high);
}

```

また、これらは物理的なアドレスであることに**も注意が必要です**。システムソフトウェアがページングを有効にしている場合、チャンネルが使用する場所を、使用されるメモリの領域を**識別**して同じ仮想アドレスにマッピングする必要があります。

そこで、おさらいです。8つのチャンネルがあることを知り、そのチャンネルのデバイスがDMACを使えるようにした後、チャンネルレジスタの1つに書き込むことで、チャンネル情報（メモリの位置、読み書きの条件）を与えて、DMACへの読み書きを開始することができる。このデータはどこから来るのかと聞かれるかもしれません。また、読み出した場合、そのデータはどこに行くのか？これは、そのチャンネルを制御しているデバイス次第です。例えば、フロッピーディスクドライブでは、リードコマンドをフロッピーディスクドライブコントローラー（FDC）に送ると、FDCはDMACに通知して転送を開始します。DMACは、ベースとなる物理アドレス、チャンネルの動作（リードかライトか、この場合はリードであることが望ましい）、バッファのサイズを取得し、あとは自分で書き込みます。FDCはDMACにデータを転送し続け、DMACはそのチャンネルに格納されているアドレスが指すバッファにデータを配置することになる。FDCはDMACにデータを転送し続け、DMACはそのチャンネルに格納されているアドレスが指すバッファにデータを入れます。ここでは、そのチャンネルのアドレスとカウントレジスタに書き込むことで、バッファの位置とサイズを設定します。

待つて、待つて、待つて。DMACは一度に64Kしか転送できないことを覚えていますか？さらに悪いことに各チャンネルのベースアドレスも同じ制限があり、DMACがアクセスできるのは64KのRAMに限られるということになります。これは悪い制限だと思いませんか？これを解決するのが、外部ページレジスタです。もっと詳しく見てみましょう。

拡張ページアドレスレジスター

ページレジスタは、チャンネルが設定されているメモリロケーションがどのページに存在するかを設定するために使用されます。この8ビットをチャンネルのベースアドレスに追加すると（チャンネルのベースアドレスを0xFFFFFとする）、実質的に8ビット増えることになり、最大16MBのメモリにアクセスできるようになります。これがページレジスタの仕組みです。

これらのページレジスタには、そのチャンネルの転送アドレスの上位8ビットのみが格納されます。これは、これらのページレジスタの値が常に64kの倍数であることを意味し、重要な特徴です。

案の定、ここでちょっとした問題が発生する。DMACが1つだった当初のパソコンは、AT/EISA/MCAとそれ以降のパソコンとではi/oポートが異なっていた。また、DMACが2つになったことで、レジスターが追加され、ビット数も増えた。オリジナルのPCのページレジスタは4ビットしか追加されていない（A16～A19）。一方、新しいコンピュータでは、ベースチャンネルアドレスに8ビット（A16～A23）が追加された。

ISA DMAC拡張ページ・アドレス・レジスタ	
ポー ト	ディスクリプション
0x80	チャンネル0（オリジナルPC）／エクストラ／診断ポート
0x81	チャンネル1(オリジナルPC)/チャンネル2(AT)
0x82	チャンネル2(オリジナルPC)/チャンネル3(AT)
0x83	チャンネル3(オリジナルPC)/チャンネル1(AT)
0x84	エクストラ
0x85	エクストラ
0x86	エクストラ
0x87	チャンネル0(AT)
0x88	エクストラ
0x89	6チャンネル（AT）
0x8A	チャンネル7（AT）
0x8B	チャンネル5（AT）
0x8C	エクストラ
0x8D	エクストラ
0x8E	エクストラ
0x8F	チャンネル4（AT）／メモリーリフレッシュ／スレーブ接続

さて、ちょっと待つてください。さて、上の表で気にしなければならないのは、ATのポートだけです。これは、すべてのチャンネル外部ページレジスタが、チャンネルの設定時に保存したチャンネルのベースアドレスにさらに8ビットを追加することを意味します（前のセクションを参照）。例えば、フロッピーコントローラをプログラミングする際、フロッピーがDMAチャンネル2を使用することがわかっています。例えば、フロッピーディスクコントローラをプログラムする際、フロッピーディスクはDMAチャンネル2を使用することがわかっています。例えば、64kよりも低い位置にバッファを保存したい場合、チャンネルのアドレスをどこかに設定すればよいのです。まあ、そんなところ です。また、そのアドレスの上位8ビットを決定するために使用されるページレジスタを設定する必要があります。つまり、ページレジスタを設定するには

- 0に設定すると、ページ0、アドレスに何も追加されま
- せん 1に設定すると、ページ1、アドレスに64kが追加されます
- 2に設定すると、ページ2、128Kがアドレスに追加される
- 255に設定すると、Page 255 = 255*64K=0xFF0000となり、上位8ビット全てが設定され、16、320K、約16MBがアドレスに追加される。

ページテーブルのページを変更すると、DMAが読み書きするアドレスが変わることに注目してほしい。これにより、DMACは最大16MBの

2021/11/15 13:10

オペレーティングシステム開発シリー

メモリーに効率よくアクセスできるようになります。すごいでしょ？まだ少し制限がありますが、**64K**に制限されるよりはずっと良いと思いませんか？

他のレジスターと同様に、醜いマジックナンバーを隠すことができます。

これらのレジスタを設定するために必要なことは、どのレジスタに書き込まれているかを（どのチャンネルが渡されているかで）判断し、そのレジスタに値を書き込むことです。

注目は、`dma_set_external_page_register(2, 0x1000);` のような呼び出しで、チャンネル2のページレジスタに0x1000を設定することができます。いいですか？

レジスター

上記のレジスタに加えて、`case` ブロックでは以下のレジスタも利用可能です。

コマンドレジスタ

このレジスタは、DMACの制御に使用されます。以下はレジスタ4には何も書き込まないでください。

- ビット0: **MMT** Memory to Memory Transfer
 - 0: Disable
 - 1: 有効
- ビット1: **ADHE** チャンネル0アドレスホールド
 - 0: Disable
 - 1: 有効
- ビット2: **COND** Controller Enable
 - 0: Disable
 - 1: 有効
- ビット3: **COMP** タイミング
 - 0: ノーマル
 - 1: コンプレッサー
- ビット4: **PRIO** プライオリティ
 - 0: 固定プライオリティ
 - 1: ノーマルプライオリティ
- ビット5: **EXTW** ライト選択
 - 0: レイトライト選択
 - 1: 拡張ライトセレクション
- ビット6: **DROP** DMA Request (DREQ)

- **0:** DREQセンス・アクティブ・ハイ
- **1:** DREQセンス・アクティブ・ロー
- **ビット7 : DACKP DMAアックノレッジ(DACK)**
 - **0:** DACKセンス・アクティブ・ロー
 - **1:** DACKセンス・アクティブ・ハイ

これらのビットのほとんどは、i86アーキテクチャでは動作しません。唯一動作するのはビット2で、これを使ってコントローラを有効にしたり無効にしたりすることができます。メモリからメモリへの直接の転送も便利だと思うでしょうが、そうではありません。他のビットを使うと、何もできなかったり、予想外の結果になったりします。

これらは、本章最後のデモにあるdma.hというヘッダーファイルに含まれています。ここではビットマスクとして表示されています。

モードレジスタ (ライト)

このモードは、コントローラのモードを設定します。以下のようなフォーマットになっています。

- **ビット0~1: SEL0, SEL1** チャンネルセレクト
 - **00** : チャンネル0
 - **01** : チャンネル1
 - **10** : チャンネル2
 - **11** : チャンネル3
- **ビット2-3 : TRA0, TRA1** 転送タイプ
 - **00** : コントローラのセルフテスト
 - **01** : ライト転送
 - **10** : リード転送
 - **11** : 無効
- **ビット4 : AUTO** 転送完了後の自動再初期化（デバイスがサポートしている必要があります）
- **ビット5 : IDEC**
- **6-7ビット : MOD0, MOD 1** モード
 - **00** : トランスファー・オン・デマンド
 - **01** : シング
 - **10** : ブロック
 - **11** : カスケード

このレジスタは重要です。 チャンネルを設定し、メモリブロックを読み書きする準備をするためには、このレジスタに動作モードを書き込む必要があります。しかし、このレジスタに書き込む前に、何かを変更する前に、モードを設定したいチャンネルをマスクオフ（ディセーブル）することを常にお勧めします。これは、現在使用中のチャンネルのモードを変更すると、データが破損するなどの問題が発生するためです。

まず最初にしたいことは、醜い数字を意味のある名前で隠すことですが、それがここにあります。しかし、これは少し違います。これらの列挙は、マスクとフラグの組み合わせです。マスクは上のリストのビットフォーマットと一致しています。フラグは単純化のためにあります。フラグは、上のリストの必要なビットをセットまたはクリアすることで、オプションをビットワイズオアすることができます。例えば、チャンネル番号とチャンネルのモードを組み合わせ、自動初期化による単一転送の読み取りに設定するには、次のようにします。

| dma_mode_transfer_singleです。かっこいいでしょ？

どちらのコントローラでもフォーマットは同じなので、エニュームは1つだけです。

```
enum DMA_MODE_REG_MASK {

    dma_mode_mask_sel = 3,

    DMA_MODE_MASK_TRA =
    0xc, DMA_MODE_SELF_TEST
    = 0,
    dma_mode_read_transfer =4,
    dma_mode_write_transfer = 8,

    DMA_MODE_MASK_AUTO =
    0x10, DMA_MODE_MASK_IDEC
    = 0x20。

    DMA_MODE_MASK = 0xc0,
    DMA_MODE_TRANSFER_ON_DEMAND=
    0,
```

```

DMA_MODE_TRANSFER_SINGLE =
0x40, DMA_MODE_TRANSFER_BLOCK =
0x80, DMA_MODE_TRANSFER_CASCADE
= 0xC0
};

```

DMA0_MODE_REGが0x0b (DMA 0モードレジスタ)、**DMA1_MODE_REG**が0xd6 (2番目のDMAモードレジスタ) だとすると、特定のチャンネルのDMAモードを設定するために必要なことは以下の通りです。

このレジスタでは、任意のチャンネルのモードを設定することができます。いいでしょう？例えば、フロッピードライブの書き込みを準備したい場合は、**dma_set_mode (2, 0x5A)**を実行します。(フロッピーはプライマリDMACのチャンネル2を使用していることを覚えていませんか？) と0x56 `dma = (channel < 4) ? 0 : 1;`
`= 01010110`のバイナリです。上のリストと比較すると、Mode=01 (Single Transfer)、AutoInitが設定されている(完了後に自動初期化する)、転送タイプ=01 (Write)、チャンネル2(10)となっています。

`dma_mask_channel (channel)`。
DMA_MODE_MASK_AUTOビットは便利なビットです。これにより、コントローラのリセットとチャンネルバスのアドレスとカウンタの設定を行うことで、DMACを最初から初期化することができます。このビットが設定されていない場合は、読み出しや書き込みのたびにDMACを再初期化する必要があります。

注：AutoInitビット (**DMA_MODE_MASK_AUTO**) は、Virtual PCではあまりサポートされていないようです。このため、Virtual PCでの移植性を維持するために、AUTOINITではなく、読み書きのたびにDMACを再初期化する方法を選択しました。他のエミュレータやマシンではサポートされていないかもしれません。

`dma_set_mode (channel, dma_mode_read_transfer | dma_mode_transfer_single | dma_mode_mask_auto)` となっています。

このレジスタは、ソフトウェアがDMACに直接送信することを可能にします。最初の2ビットはチャンネルの選択に使用されます。例えば、00=チャンネル0、01=チャンネル1、10=チャンネル2、11=チャンネル3となります。3番目のビットは、0の場合、チャンネル要求ビットをリセットし、1の場合はセットする `dma_set_1` の場合 `uint8_t channel` を設定します。

- ビット0-1: チャンネルセレクト `dma_set_mode (channel,`
- ビット2: 0=チャンネル要求ビットをリセット `1=要求ビットをセット`
`dma_mode_write_transfer | dma_mode_transfer_single | dma_mode_mask_auto)` となっています。

Request RegisterはMemory-to-Memoryの操作に使用されます。コマンドレジスタを思い出すと、i86アーキテクチャではMemory-to-Memoryトランザクションを有効にすることはできませんし、すべきでもありません。このため、このレジスタは重要ではありません。

チャンネルマスクレジスタ (ライト)

このレジスタでは、1つのDMAチャンネルをマスクすることができます。ビット0と1でチャンネルを設定します (00=チャンネル0、01=チャンネル1、10=チャンネル2、11=チャンネル3)。ビット4は、チャンネルをマスクするかアンマスクするかを決定します。ビット4が0であれば、チャンネルのマスクを解除します。1であれば、マスクされます。他のビットはすべて未使用です。

- ビット0-1: チャンネルセレクト
- ビット2: 0=unmasksチャンネル、1=masksチャンネル

チャンネル その他のビットは未使用です。

マスクレジスタ (ライト)

このレジスタには、どのチャンネルが現在マスクされているか、またマスクされていないかの情報が含まれています。この8ビットのレジスタの上位4ビットは常に未使用です。下位4ビットは、4つのチャンネルのいずれかをマスクまたはアンマスクするために使用されます。例えば、ビット0はチャンネル0、ビット1はチャンネル1、というようにです。注：カスケード接続により、チャンネル4をマスクすると、チャンネル4,5,6,7もマスクされます。

- 例えば、任意のチャンネルをマスク（無効化）するルーチンを用意し、必要なのはそれぞれのビットを設定することだけです。

同様に、voice mask channel (uint8_t) の channel をクリアすればよい。

```
if (channel <= 4) {
    outp |= 1 << (channel - 5);
}
```

複数のチャンネルを同時に設定するため、`MAC_CHANMASK_REG`は、複数のチャンネルを同時にマスクしたり、マスクを解除したりすることができます。

outportb(DMA1_CHANMASK_REG, channel)となります。

ステータスレジスタ

ステータスレジスタのフォーマットは以下の通りです。

- **ビット0：TC0** チャンネル0が**転送完了（TC）**に達した場合に設定 **ビット**
- **1：TC1** チャンネル1が**転送完了（TC）**に達した場合に設定 **ビット2：**
- **TC2** チャンネル2が**転送完了（TC）**に達した場合に設定 **ビット3：TC3**
- チャンネル3が**転送完了（TC）**に達した場合に設定 **ビット4：REQ0** チャンネル0が**DMAリクエスト（DRQ）**を保留中の場合に設定 **ビット5：**
- **REQ1** チャンネル1が**DMAリクエスト（DRQ）**を保留中の場合に設定
- チャンネル0が**DMAリクエスト（DRQ）**を保留している場合は**REQ0**を設定 **ビット5：**チャンネル1が**DMAリクエスト（DRQ）**を保留している場合は**REQ1**を設定 **ビット6：**チャンネル2が**DMAリクエスト（DRQ）**を保留している場合は**REQ2**を設定 **ビット7：**チャンネル3が**DMAリクエスト（DRQ）**を保留している場合は**REQ3**を設定

このレジスタはあまり有用ではありません。ほとんどの場合、**DMAC**を制御しているデバイスは、転送が完了したときに**IRQ**を送信するので、このレジスタをポーリングして情報を得る必要はありません。最初の**4ビット**は、そのチャンネルの転送が完了しているかどうかを示し、最後の**4ビット**は、そのチャンネルに保留中の**DMA**リクエストがあるかどうかを示します。

ISA DMAコマンド

コントローラには、ソフトウェアがコントローラにコマンドを送信できるようにするための特別なレジスタが用意されています。これらのコマンドは特定のビットフォーマットを必要とせず、簡単な入出力操作で起動することができます。

DMACは、アドレスバス（A0-A3ライン）のデータと、OROおよびIOWラインのステータスによって、コマンドを認識します。

これらのレジスタには特別なものは何もないことに注意してください。これらは、本章の冒頭にある汎用レジスタの表にも記載されています。

クリア バイト ポインタ フリップフロップ

これは特別なi/oアドレスポートで、8ビットDMAC（プライマリDMAC）で作業する際に、16ビット転送の間にフリップフロップを制御することができる。

両方のDMACに2つのポートがあります。

ISA DMAC フリップフロップ ポート	
ポー ト	ディスクリプション
0x0C	DMAC 0 (16ビット) スレーブ(書き込み)
0xD8	DMAC 1 (8ビット) マスター(書き込み)

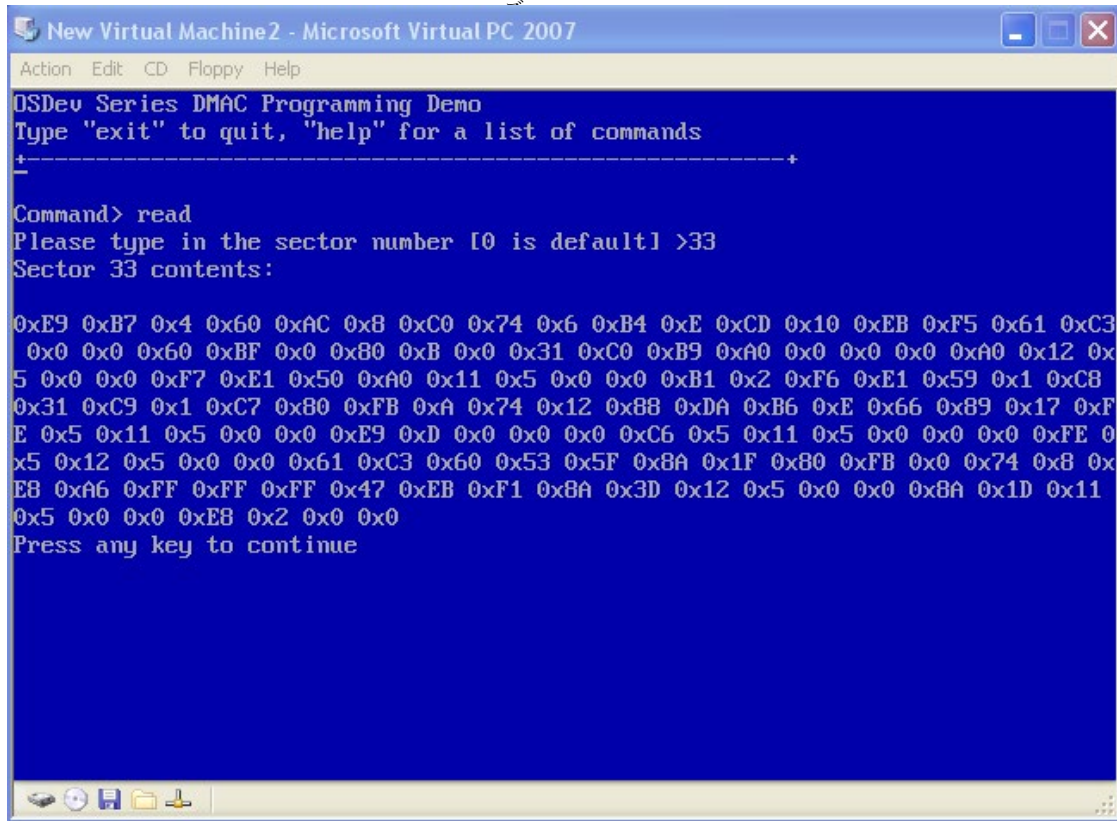
よく似た方法で、以下のレジスタに任意の値を書き込むことで、DMACをリセットすることができます。

// ! このレジスタに何が書き込まれるかは問題ではありません。
 outportb((dma==0) ? DMA0_CLEARBYTE_FLIPFLOP_REG : DMA1_CLEARBYTE_FLIPFLOP_REG, 0xff)
 となります。)

ISA DMAC UnMask All Ports	
ポート	ディスクリプション
0x0E	DMAC 0 (16ビット) スレーブ(書き込み)
0xDC	DMAC 1 (8ビット) マスター(書き込み)

このコマンドも同様に、この例のように書き込まれてしまっている。ドウェアプログラミングのコマンドがこのように簡単にできたらいいと思いませんか？

```
// ! このレジスタに何が書き込まれても問題ありません
outportb(DMA1_UNMASK_ALL_REG, 0xff);
```



Virtual PCで動作するデモ デモ
モダウンロード

やったー、またデモの時間だー悪いニュースは、このデモは前の章と全く同じに見えるということです（ただし、BochsとVirtual PCの両方で動作するようになりました）。良いニュースは、このデモが新しいDMAインターフェースを使用するようにアップグレードされていることです。

新しいコードの核となる部分は、本章のすべてのコードを含むHAL - **dma.h**および**dma.cpp**にあります。しかし、1つだけマイナーな変更点があります。DMACのModeレジスタのAUTOINITビットはVirtual PCではあまりサポートされていないので、**dma_set_read**および**dma_set_write**ルーチンは、デモコードではビットをセットしません。

リード読み込み操作の間、チャンネルを準備する **flpydisk_read_sector_imp** ルーチンは、DMAC を初期化し、DMAC をリード操作のために準備します。readの残りの部分(編集されています)は前章と同じで、FDCにREADコマンドを送信する役割を担っています。**DMA_BUFFER**は、DMACの転送に使用できるフリーメモリのバッファに過ぎません。**dma_initialize_floppy**は、新しいDMAミニドライバーを使ってDMACを初期化し、FDCにDMAモードREAD-TRを使用できるDMAに準備します。FDCを初期化した後、ドライバーの**dma_set_read**ルーチンをチャンネル**FDC_DMA_CHANNEL**に呼び出して、DMACのREAD操作の準備をします。**FDC_DMA_CHANNEL**はチャンネル2である（FDCがDMACのチャンネル2を使用することを覚えているだろうか）。

```

// !書き込みのためにチャンネルを準備する
void dma_set_write (uint8_t channel) {...}

// ! セクターの読み込み
void flpydisk_read_sector_imp (uint8_t head, uint8_t track, uint8_t sector)
{
    dma_mode_write_transfer | dma_mode_transfer_single) 。)
    { uint32_t st0, cyl;
}

// ! DMAの初期化
dma_initialize_floppy ((uint8_t*) DMA_BUFFER, 512 )である。

// ! リード転送用のDMAを設定
dma_set_read ( FDC_DMA_CHANNEL );

// ! 残りのコードは同じです....
}

```


次の機会まで。

マイク

BrokenThorn Entertainment社。現在、DoEと[NeptuneOperating System](#)を開発中です。質問

やコメントはありますか？お気軽に[お問い合わせください](#)。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ[私に教えてください](#)。

リファレンス

- 82C37A CMOS High Performance Programmable DMA Controller データシート
- "The Undocumented PC"

ご質問やご意見がございましたらお気軽に[お問い合わせください](#)。



第20章

ホーム

第22





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - ファイルシステムとVFS

by Mike, 2010

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

OS開発のための終わりのなきシリーズ、第22章へようこそこれは第22章というよりも、OS開発シリーズの2年目にあたります。

今回もまたファイルシステム関連のチュートリアルです (心配しないでください、これが最後です ;))。最初のは、ブートコードからメインのブートローダプログラムをロードするために必要でした。2番目のものは、メインのブートプログラムがカーネルをロードするために必要でした。今度は、カーネルがプログラムをロードして実行できるように、カーネル用にもう一つが必要です。しかし、この章と他の2つの章には違いがあります。この章はアセンブリ言語ではなくC言語で行われます。)

しかし、今回は新たな試みとして、**仮想ファイルシステム (VFS)** にも注目します。これにより、あらゆるファイルシステムドライバと異なるディスクデバイスを同じように扱うことができるようになります。これは、ローカルディスクドライブだけでなく、あらゆるネットワークファイルシステムとのインターフェースにも使用できます。

いいですか？

ファイルシステム

アブストラクト

ファイルシステム

ファイルシステムは、情報を読み書きするための論理的な方法を定義するものです。ファイルシステムは、情報を読み書きするための論理的な方法を定義したもので、**仕様**の一つと言えます。ほとんどのPCのファイルシステムは、デスクトップの「ファイル」と「フォルダ」の概念に基づいています。

ファイルシステムには様々な種類があります。広く使われているもの (FAT12、FAT16、FAT32、NTFS、ext (Linux)、HFS (古いMACで使われている) など) もあれば、特定の企業が社内でのみ使用するファイルシステム (GFS - Google File Systemなど) もあります。他のファイルシステムは、特定の企業が社内でのみ使用するものです。また、独自のファイルシステムを開発・設計することもできます。

ファイルシステムは、データの保存と整理のために使用されます。ファイルシステムは、リムーバブルメディア (フロッピー、フラッシュドライブ、CD、DVD)、ローカルドライブ (ハードディスクドライブ)、ネットワーククライアント上のファイルやディレクトリに簡単にアクセスする方法を提供します。ファイルシステムは、インメモリイメージとしても存在します。例えば、特別なタイプのファイルシステムの「足型」を含んだファイルを読み込むことができます。

ファイルとフォルダー

ファイルとは、プログラムやユーザーにとって何かを表すデータの集まりです。このデータは、私たちが望むものであれば何でもあります。それは、データをどのように解釈するかによります。例えば、**テキストファイル**は、テキスト情報を含んでいます。ファイルは、何かのイメージであることもあります。**フォルダ**は、ファイルを論理的にまとめたものです。**ディレクトリ**とも呼ばれています。

ディレクトリは、大量のファイルを管理するための手段です。ディレクトリは通常、**ツリー構造**になっています。これを**ディレクトリツリー**と呼びます。すべてのディレクトリとファイルの親となるディレクトリは1つだけで、**ルートディレクトリ**です。**ファイルパス**は、ディレクトリツリー内のファイルの位置を示します。例えば、「**a:\myfile.txt**」というファイルは、myfile.txtがファイル名です。**a:\mydir\myfile.txt**は「**mydir**」というサブディレクトリにある「**myfile.txt**」というファイルで、「**a:**」というデバイスのルートディレクトリにあります。

ファイルとフォルダーのネーミング

フォルダやファイルの名前は、そのファイルやフォルダを表す文字列であり、通常はその内容によって表される。ファイルシステムでは、ファイル名とフォルダ名の意味合いが異なり、それぞれに制約があります。例えば、FAT12では、ディレクトリエントリ内のファイル名やフォルダ名を**11**バイトの配列 (ファイル名が8、拡張子が3。これは「**8.3命名規則**」とも呼ばれています。) により、ファイル名とフォルダ名は**11**文字に制限されます。一方、NTFSは**LFN (Long File Name)** をサポートしており、**255**文字に制限されています。また、NTFSでは、ファイル名をファイル属性とともに**マスターファイルテーブル**に格納しています。

ほとんどのファイルシステムのファイル名は、大文字と小文字を区別しません。しかし、ファイルシステムによっては、ファイル名を内部的に異なる方法で保存している場合があります。例えば、フロッピーディスク上のファイルに**8.3**の小文字のファイル名を付けていても、OSからそのファイルを読み込むときにはすべて大文字のファイル名を使うことができることをご存知でしょうか。**Windows**はファイル名のLFNを表示しますが、FAT12の**8.3**ファイルのエントリには、その**8.3**の全大文字のファイル名だけが表示されます。これが可能になるのです。

ファイルの種類

シンボリックリンクの

シンボリックリンクとは、パスを短くするための方法です。例えば、`a:/folder/link.lnk`は、`a:/otherfolder/subfolder/yet another folder/link.txt`を指します。これで、テキストファイルに簡単にアクセスできます。また、シンボリックリンクは

フォルダを整理するためによく使われます。Windowsのスタートメニューのようなものです。プログラムへのシンボリックリンクが含まれています。シンボリックリンクを実装するのはそれほど難しくありません。指定されたノード（リンク）を見つけます。それはリンクのように見えるので、実際のパスを取得して、代わりにそのファイルを読みます。

Windowsショートカットは、シンボリックリンクの一種です。

パイプ

IPC (InterProcess Communication) の一種である「パイプ」。パイプとは、通常、2つ以上のプロセス間に存在する**仮想ファイル**のことです。最も良い例は、Unixの**stdout**、**stdin**、**stderr**でしょう。これらは通常のファイルとして扱われますが、**stdout**に書き込まれたデータは画面に表示されます（または**stdout.txt**に表示されます）。

特殊なファイルタイプ

メタファイル

ファイルシステムの中には、ファイルシステム専用の特別なファイルやフォルダを実装しているものもあります。通常、同じディレクトリに同じ名前のファイルやフォルダを2つ置くことはできません（フォルダと同じ名前のファイル名も置けません）。そのため、これらの隠しファイルを使ってファイルやフォルダの名前を付けることも、実装によってはできない場合があります。

例えば、NTFSではファイルシステム用にいくつかのメタファイルが用意されています。これらのファイルは、システムドライブのルートディレクトリ（通常はC:）に配置されます。例えば、**\$MFT**、**\$MFTMirr**、**\$LogFile**などのファイルがあります。隠しファイルの表示やシステムファイルの表示にチェックを入れても表示されることはありませんが、ここに上記のような名前のファイルを作成するとどうなるかを見てみましょう。これらのファイルは他の場所でも作成できますが、ルートディレクトリに作成すると、メタファイルのために「ファイルが既に存在します」というエラーが発生します。

デバイスファイル

Unix系システム、DOS（ひいてはWindows）には、デバイスを表す特殊な「ファイル」である**デバイスファイル**があります。例えば、**NUL**(ヌルデバイス)、**CLOCK\$**、**PRN**(プリンタ)などです。ここでは、デバイスファイルの一覧を紹介します。

- con prn
- aux
- clock\$
- nul
- COM0、COM1、...COM9
- LPT0、LPT1、...LPT9

これらの名前はDOSやWindowsでは特別な意味を持つため、ファイルやフォルダに上記のような名前をつけることはできません。

...としています。

は、一部のファイルシステムで採用されている特殊なファイルです。は、カレントディレクトリを参照するファイル情報を含むファイルのファイル名である。は、そのファイルの親ディレクトリを参照する情報を含むファイルのファイル名で、「...」は、そのファイルの親ディレクトリを参照する情報を含むファイルのファイル名である。例えば、**c:\mydir\file.txt**というファイルがあり、**c:\mydir**がカレントディレクトリだった場合、パス名...はC:を、パス名...はc:を参照します。

ファイルシステムの種類

フラットファイルシステム

フラットファイルシステムとは、サブディレクトリを持たないファイルシステムである。その代わりに、すべてのファイルは同じ（ルート）ディレクトリにある。初期のコンピュータシステムの多くはフラットファイルシステムを採用していた。最近のOSでは、より高度な階層型ファイルシステムが採用されている。フラットファイルシステムは、小型で導入しやすい反面、新規に開発するのが難しい。

階層型ファイルシステム

このタイプのファイルシステムは、サブディレクトリをサポートしています。最近のほとんどのファイルシステム（**FAT12**、**FAT16**、**FAT32**、**etx**、**NTFS**を含む）はこのカテゴリーに当てはまる。（**FAT12**の最初のバージョンは、フラット・ファイル・システムでした。しかし、その後のバージョンではサブディレクトリをサポートしています）。

ジャーナリング・ファイルシステム

このタイプのファイルシステムでは、ファイルシステムの変更の「ジャーナル」を使用します。これは、システムがファイルやディレクトリに加えようとする変更を、手順を完了する前に記録したものです。これにより、ファイルシステムの操作（ファイルの書き込みなど）中にクラッシュが発生した場合、ジャーナルを読んで変更を元に戻し、ファイルシステムを修復することができます。

ファイルシステムドライバ

ファイルシステムが「ファイル」や「ディレクトリ」を読み書きするための仕様が定められたものであるのに対し、**ファイルシステムドライバ**は、特定の種類のファイルシステムを実装するためのものです。ファイルシステムドライバの例としては、マイクロソフト社の**NTFS**ファイルシステムを実装した**ntfs.sys**があります。また、ファイルシステムドライバは、大規模なソフトウェアの中にミニドライバとして実装されることもあります。ブートローダがその良い例です。ブートローダは、別のドライバプログラムなしでディスクからファイルをロードできる必要があるため、ブートローダ自体の中に、さまざまな種類のファイルシステムに対応した複数のファイルシステムミニドライバが含まれています。シリーズでブートローダを開発された方は、すでに**FAT12**ファイルシステムを経験され、当社のブートローダ用に**FAT12**ミニドライバを開発されています。

仮想ファイルシステム (VFS)

アブストラクト

仮想ファイルシステム(VFS)は、特定のファイルシステム実装の上にある抽象化レイヤーです。ソフトウェアは、VFSを介してストレージデバイスにアクセスします。これにより、ソフトウェアは、使用されているデバイスやファイルシステムの知識がなくても、異なるストレージデバイスに読み書きすることができます。また、インストールされたファイルシステムやデバイスの数に関わらず、同じコードで動作させることができます。

基本的な考え方は、単一のシステムインターフェースで、あらゆるファイルシステムを統一的に扱えるようにすることです。**Windows**、**Linux**、**Mac OS**はそれぞれ異なる方法でVFSをサポートしています。

インプリメンテーション

VFSを導入するにはさまざまな方法があります。

マウントポイント一覧

マウントポイントリストとは、マウントされているファイルシステムとそのマウントされている場所のリストです。例えば、ファイルを読み取る必要がある場合、**OS**は通常、VFSのReadFile()関数を呼び出し、マウントされたファイルシステムのリストを検索して、ファイルが入っているデバイスとファイルシステムを見つけます。そして、そのファイルシステムのReadFile()関数に読み込み要求を渡します。

ノードグラフ

ノードグラフは、ファイル、フォルダ、マウントポイントなど、さまざまなタイプのファイルを表すノードのグラフを含んでいます。各ファイルノード構造には、通常、ファイルを読み書きするためのファイルシステム固有のルーチンへの関数ポインタが含まれています。

例えば、次のようなFILE構造を作ることができます。

```
typedef struct _FILE {
    チャー //ファイル名
    uint32_t flags; //flags
    uint32_t fileLength; //ファイルの長さ
    read_func_t read; //ファイルの読み込み、書き込み、オープン、クロ
    open_func_t open;
    close_func_t close;
    write_func_t write;
} FILE, *PFILE;
```

関数のポインタがこのFILE構造体に格納されていることに注目してください。例えば、ファイルを読みたいので、fopen()を呼び出し、最終的にVFSのOpenFile()関数を呼び出します。VFSのファイル操作ルーチンが行う必要があるのは、特定のFILEの関数ポインタに制御を渡すことだけです。

これにより、VFSのOpenFile(PFILE, const char *filename)とすることができます。

DOSとWindows (file) file->open (filename)となります。

DOSおよびWindowsでは、マウントされたファイルシステムを表すために、「a」から「z」までの文字が割り当てられています。Windowsでは、ドライブレターとそのオブジェクトマネージャ名の間にシンボリックリンクが張られます。例えば、ドライブレター「c」の場合、例えば、ドライブレターc:(symbolic link name \\C:)は、オブジェクト名\Device\HardDiskVolume1のデバイスオブジェクトにマッピングされます。ファイルシステムは、デバイスオブジェクトを所有するために自らを登録することができます。オブジェクトを所有しているファイルシステムが見つかった場合、ファイルパス名の残りの部分（この例では "myfile.txt"）は、そのファイルシステムのFileOpen()関数に渡されます。

ドライブレターの割り当て

Windowsは、マウントされたファイルシステムを表すデバイスやパーティションへのドライブレターの割り当てをサポートしています。（起動時に、デバイスオブジェクトを所有するファイルシステムドライバーが登録されていない場合、Windowsはデバイスに対してRAWミニドライバーを使用します）。ドライブレターは、ネットワーク共有ドライブ、仮想ディスクイメージ、ローカルまたはネットワーククライアント内の別の場所へのシンボリックリンクを参照することもできます。また、ドライブレターは、ネットワーク共有ドライブ、仮想ディスクイメージ、ローカルまたはネットワーククライアント内の別の場所へのシンボリックリンクを指すこともあります。

インターフェース

ここでは、VFSの実装を簡単にするために、ドライブレターの割り当てとマウントポイントのリストを使用します。このシリーズで紹介するOSには、デバイス管理やI/O管理がないので、シンプルな実装にする必要があります。

個人的には、ファイルシステムドライバよりもVFSを先に開発することをお勧めします。そうすれば、VFSのインターフェースやフレームワークがすでに完成しているからです。

ファイル

C言語を使ったことがある人なら、悪名高いFILE*データ型を知っているでしょう。FILE*は、ファイルオブジェクトへのポインタを表す**抽象データ型(ADT)**です。ISO Cでは、C言語の実装において、FILE型を定義しなければならないと規定されていますが、どのようなものかは定義されていません。

は、構造体の内部にある。つまり、**FILE***が**ISO C**であるのに対し、構造体の中身はインプリメンテーションで定義されている。

ファイルの現在の状態を表すファイル構造を、好きなように定義することができます。ファイルには名前とサイズがあり、これはすでに**2**つのメンバです。さらに、**ファイルの終端 (EOF)** を示すフラグや、ファイル固有のフラグが必要で、そのために**2**つのメンバが必要です。また、ファイルの現在の位置（クラスターとクラスターのオフセット）を追跡する方法も必要で、現在は次のようになっています。

簡単な**typedef struct _FILE** 必要に応じて識別のために使用することができます。

ファイルの種類

```
charname[32]
uint32_tflag
s;
uint32_tfile;
uint32_tid;
uint32_te
of;
uint32_tp
osition;
#define FS_FILE 0
#define FS_DIRECTORY 1
#define FS_INVALID 2
FILE, *PFIL
```

これまで説明してきたファイル、ディレクトリ、シンボリックリンクなど、さまざまな種類があります。ここでは、わかりやすくするために、ファイルとディレクトリに注目します。これらは、ファイルの種類を表すために、上記の**FILE**構造の**flags**メンバで使用されます。

オペレーション

ファイルに対して行うことのできる代表的な操作があります。開く

- Close
- Read
- Write
- Mount
- Unmount

ファイルオブジェクト（ファイルやディレクトリなど、ファイルの種類は問わない）のオープンとクローズを行うのが**Open**と**Close**、ファイルの種類に応じた読み込みと書き込みを行うのが**Read**と**Write**である。これらはすべて、標準的なCファイル**I/O**関数を通じてプログラマに公開されている。

VFSでは、**fsys.h**にある**Volume Manager**を通じて公開されています。

例えば、**extern void volOpenFile (const char* fname)**。このルーチンは、**FILE**オブジェクトを返す**volOpenFile()**ルーチンを呼び出します。ここでは、**extern void volReadFile (PFIL file, unsigned char* Buffer, unsigned int Length)**や**extern void volCloseFile (PFIL file)**、**extern void volRegisterFileSystem (PFILSYSTEM, unsigned int deviceID)**、**extern void volUnregisterFileSystem (PFILSYSTEM, unsigned int deviceID)**、**extern void volUnregisterFilesystemByID (unsigned int deviceID)**。複雑に聞こえるかもしれませんが、ご心配なく。しかし、デモでの実装方法は非常に簡単で

ボリュームマネージャーの導入

ファイルシステムの抽象化

まず必要なのは、ファイルシステム固有の情報を抽象化する方法です。これには、ファイルシステムの名前や、ファイルに対して実行可能な操作が含まれます。これには、関数ポインタを使用します。

```
typedef struct _FILE_SYSTEM
{
    char Name [8];
    (*Directory) (const char* DirectoryName);
    (*Mount) ();
    (*Read) (PFILファイル, unsigned char* Buffer, unsigned int Length);
    (*Close) (PFIL)を使用しています。
    (*オープン) (const char* FileName);
} filesystem, *pfilesystem;
```

インプリメンテーション

ボリュームマネージャは、デモのVFSを実装しています。ファイル `fsys.h` と `fsys.cpp` の中にあります。ドライブレターの割り当てを使ってデバイスを表現することを覚えていますか？26個のデバイスが存在するので、**DEVICE_MAX**という定数を作っておくと便利です。各デバイスは1つのマウント可能なファイルシステムしか持てないので、それらをリスト（マウントポイントリストのようなもの）にして保存します。

その仕組みは、`DEVICE_MAX` 26 ファイルシステムをポインタのリストとして保存しているので、ポインタが有効であれば、ファイルシステムがそこに登録されていることになります。配列の各要素は、それが参照するドライブレターを表しています。つまり、`'a'`は `_FileSystems[0]` に、`'b'`は `_FileSystems[1]` などがああります。ファイルシステムは、自分を書き込んでいるディスクを管理する責任があります。

このメソッドを使用すると、非常に基本的ですが、簡単にデバイスにアクセスすることができます。例えば、`volOpenFile()`は、パスの最初の文字（ドライブレター）をチェックして、そのデバイスにファイルシステムが登録されているかどうかをリストで調べるだけです。登録されていれば、そのファイルシステムの`open()`メソッドを呼び出して、ファイル名をドライバに渡すことができます。デフォルトでは`'a'`を使用しますが、入力パスに`:'`が含まれている場合は、代わりにデバイスの最初の文字を使用します。これにより、`volOpenFile`を2つの方法で呼び出すことができます。**"myfile.txt"**のような文字列を渡す場合と、**"a:myfile.txt"**を渡す場合です。**"a"**はファイルが入っているデバイスです。かつこいいでしょ？

```
FILE volOpenFile (const char* fname) { if
    (fname) {...
        // ! デフォルトでデバイス 'a' に
        設定されている unsigned char
        device = 'a';

        // ! ファイル名
        char* filename = (char*) fname;

        // ! どのような場合でも、fname[1]==':' の場合は、最初の文字はデバイス文字でなければなりませ
        ん if (fname[1]==':' の場合) {...

            device = fname[0];
            filename += 2; // パス名から削除する
        }

        // ! ファイルシステムを呼び出す
        if (_FileSystems [device - 'a']) {...

            // ! ボリューム固有の情報を設定し、ファイルを返す
            FILE file = _FileSystems[device - 'a']->open (filename); file.deviceID =
            device;
            ファイルを返します。
        }

        }

        FILEファイルです。
        file.flags = FS_INVALID;
        return file;
    }
```

その他のファイル操作ルーチンは基本的にすべて同じです。VFSがどのようにファイルシステムを保存しているかを知ると、`volRegisterFileSystem()`ファミリーのルーチンがどのように動作するかを推測できるでしょう。これらのルーチンが基本的に行うことは、ファイルシステムへのポインタをリストに格納するか、クリアすることです。

さて、`void volRegisterFileSystem(PFILESYSTEM fsys, unsigned short deviceID)`を呼び出して自分自身を登録します。`fopen()`を呼び出し、それが`VolOpenFile()`を呼び出し、さらにそれがファイルシステムの`open()`メソッドを呼び出します。これですべてが整いましたが、何かが必要です。非常に重要なものがあります。

そうですね、そろそろ突っ込んでみましょうか.....また。

FAT12 - テイクスリー

はじめに

シリーズを通して、過去に2回、FAT12を見て、実装しました。そのため、今回もFAT12を詳細に取り上げるつもりはありません。しかし、今回はFAT12の復習を兼ねて、Cドライバのコードとその動作を紹介します。

必要に応じて第11を参照しながらお読みください。

ブートセクター

重要なファイルシステム情報の多くが、ブートストラッププログラズとともにブートセクターに保存されていることを覚えていますか？具体的には、ブートセクタ内の**BPB (Bios Paramater Block)** に格納されています。

ファイルシステムをマウントする際には、BPBから情報を読み取り、後で使用するためにこの情報を保存する必要があります。そのためには、ブートセクタに一致する構造を作ればいいのです。

```
typedef struct _BOOT_SECTOR {
    uint8_t          Ignore[3]です。          最初の3バイトは無視される (jmp命令)。
    バイオパラメーターブロッ          Bpb:          //BPB構造
    ク
    バイオパラメーターブロッ          BpbExt:          //拡張されたBPB情報
    クネクス
    } bootsector, *pbootsector;          フィラー[448]です。          //構造体を512バイトにする必要がある
```

ブートセクターがどのように見えるかの良い例として、**Stage1**のブートローダプログラムがメモリ上でどのように見えるかを考えてみましょう。**Stage1**の一番最初の命令 (第4章のデモ、**Stage1.asm**を参照) は**jmp loader**でした。これは3バイトの命令なので、上の構造の最初の3バイトは、**jmp**命令の**オペレーションコード (OPCode)** です。

また、第4では、**OEM Paramater Block** (別名 : **Bios Paramater Block (BPB)**) について説明しました。**BPB**は、3バイトのジャンプ命令の直後にあります。このため、**BIOSPARAMATERBLOCK**はこの構造の中で次の位置にあります。また、**FAT32**などの他のファイル・システム用に**BPB**を拡張した**BIOSPARAMATERBLOCKEXT**構造体も提供しています。

ブートセクタの最後の448バイトには、ブートセクタの残りのプログラムコードが含まれています。今のところ重要ではないので、フィラー・メンバーのパディングとして処理しています。これにより、**BOOTSECTOR**構造がディスク上のブート・セクター(512バイト)と正確に同じサイズになることが保証されます。

BIOSPARAMATERBLOCKは、BPBのフォーマットを定義する構造体です。第5章で詳しく説明しましたが、ブートセクターと同じ構造です。

```
typedef struct _BIOS_PARAMATER_BLOCK {
    uint8_t          OEMName[8]です。
    uint16_t          BytesPerSector;
    uint8_t          SectorsPerCluster;
    uint16_t          ReservedSectors;
    uint8_t          NumberOfFats;
    uint16_t          NumDirEntries;
    uint8_t          NumSectors;
    uint16_t          メディア、
    uint16_t          SectorsPerFat、
    uint32_t          SectorsPerTrack、
    uint32_t          HeadsPerCyl、
    HiddenSectors、
    LongSectors。
} biosparamaterblock, *pbiosparamaterblock;
```

上記の構造は、より親しみやすいものになっているはずです :) そうでない場合は、第5章の説明をお読みください。

しかし、**BIOSPARAMATERBLOCKEXT**は、新しいかもしれません。**BPB**についてはすでに詳しく説明し、過去に**FAT12**の解析に使用しましたが、**FAT12**のブートセクタはBPBの拡張メンバーに依存していません。しかし、**FAT32**はそうである。

```
typedef struct _BIOS_PARAMATER_BLOCK_EXT {
    uint32_t          SectorsPerFat32です   FATあたりのセクタ数
    uint16_t          フラグです。          //flags
    uint16_t          バージョンです。      バージョン
    uint32_t          RootClusterです。      ルートディレクトリを起動する
    uint16_t          InfoClusterです。
    uint16_t          BackupBootです。      //ブートセクタのコピーの位置
    uint16_t          Reserved[6]を参照し   てください。
} biosparamaterblockext, *pbiosparamaterblockext;
```

これですべてです :) これらの構造体は、ファイルシステムドライバがBPB内のデータを参照するための簡単な方法です。これらの構造体は、ファイルシステムドライバがBPB内のデータを参照し、後にファイルシステムで使用するための簡単な方法を提供します。あとは、ブートセクタを読み込んで、**PBOOTSECTOR**を介してデータにアクセスするだけです。)

前章で開発したフロッピーディスクのドライバを使って、セクタを読み取る。

必要なのはそれだけです。重要な情報はすべて **bootsector.bpb** に入っています。あとは、ファイルシステムをマウントするだけです...
PBOOTSECTOR ブートセクタ;

ファイルシステムのマウントの読み込み

`bootsector = (PBOOTSECTOR) floppydisk_read_sector (0)`となります。

BPBの情報がメモリに入ったところで、ファイルシステムを使用するための準備をする必要があります。まず、必要な情報を決めることから始めます。

さて、ディスク上の総セクタ数を知る必要があります。また、ディレクトリエントリの総数も必要です。他にも、**ファイルアロケーションテーブル (FAT)** やルートディレクトリの使用に役立つ情報があります。

さて... **BPBのSECTOR MOUNT INFO** (セクタが格納されているのを覚えていますか？) これを利用して、BPBの情報の一部をMOUNT_INFO構造体にコピーすればよいのです。

さてさて、FAT12/16マウントのディスクで、最初のFATとルートディレクトリの位置を確認してみましょう。

ブートセクタ	エクサ	uint32_t numSectors;	uint32_t fatOffset;	uint32_t numRootEntries;	uint32_t rootOffset;	uint32_t rootSize;	uint32_t fatSize;	uint32_t fatEntrySize;	uint32_t fatEntrySize.
ブートセクタ	エクサ	uint32_t numSectors;	uint32_t fatOffset;	uint32_t numRootEntries;	uint32_t rootOffset;	uint32_t rootSize;	uint32_t fatSize;	uint32_t fatEntrySize;	uint32_t fatEntrySize.
ブートセクタ	エクサ	uint32_t numSectors;	uint32_t fatOffset;	uint32_t numRootEntries;	uint32_t rootOffset;	uint32_t rootSize;	uint32_t fatSize;	uint32_t fatEntrySize;	uint32_t fatEntrySize.

2つのFATがあることに注目。最初のFATは、ディスクのブートセクタの直後にあります。このため、MOUNT_INFOのfatOffsetを1に設定しています。また、Root Directoryは両方のFATの直後にあることに注意してください。これを知っていると、ルート・ディレクトリの開始セクタを求める簡単な計算ができます。**(NumberOfFATs * sectorsPerFAT) + 1**。ブートセクタのために1を加える必要があります。

これで、最初のFATとルートディレクトリの位置がわかりました。ルート・ディレクトリのサイズを求めるために必要なのは、ルート・ディレクトリのエントリ数と各エントリのサイズです。FAT12の各ディレクトリエントリは、32バイトのサイズを持つ特定の構造形式です。したがって、必要なのは**ブートセクタ->Bpb.NumDirEntries * 32**です。これは、ディレクトリが占めるバイト数です。これをセクタあたりのバイト数で割って、セクタ数に変換します。

それだけではありませんが、FAT12ドライバーの初期化は完了です。簡単でしょう？重要なファイルシステム情報はMOUNT_INFOに入っているので、あとはディレクトリを解析してファイルシステムを構築していきましょう。

ディレクトリの解析

フォーマット > Bpb.SectorsPerFat) + 1;

```
MountInfo.numSectors = bootsector->Bpb.NumSectors;
MountInfo.fatOffset = 1;
MountInfo.fatSize = bootsector->Bpb.SectorsPerFat;
MountInfo.fatEntrySize = 8;
MountInfo.numRootEntries = bootsector->Bpb.NumDirEntries;
MountInfo.rootOffset = (bootsector->Bpb.NumberOfFats * bootsector->Bpb.SectorsPerFat) + 1;
MountInfo.rootSize = (bootsector->Bpb.NumDirEntries * 32) / bootsector->Bpb.BytesPerSector;
```

FAT12のディレクトリエントリは、サブディレクトリの情報を提供する32バイトの構造体で構成されています。各ディレクトリ・エントリは以下の形式を持っています。

```
typedef struct _DIRECTORY {
    uint8_t    ファイル名[8];           //ファイル名
    uint8_t    Ext[3];                 //拡張子 (8.3ファイル名フォーマット)
    uint8_t    アトリビュート。         //ファイルの属性
    uint8_t    Reserved;
    uint8_t    TimeCreatedMs;          作成時間
    uint16_t   TimeCreated;
    uint16_t   DateCreated;            作成日
    uint16_t   DateLastAccessed;
    uint16_t   FirstClusterHiBytesです
    uint16_t   LastModTime;             最終更新日/時間
    uint16_t   LastModDate;
    uint16_t   FirstClusterです。       ファイルデータの最初のクラスタ
    uint32_t   ファイルサイズ。         サイズ(バイト)
} directory, *pdirectory;
```

それがすべてです :) これはディレクトリエントリです。私たちのDIRECTORY構造に格納されている情報は、サブディレクトリであったり、ファイルであったりします。**Filename**と**Ext**は、ファイルまたはディレクトリの8.3フォーマット名を含む。

Attribは、ファイルやディレクトリの属性を含みます。参考までに以下のような値があります。読み取り

- 専用 : 1
- 非表示: 2
- システムです。 4

- Volume Labelです。8
- サブディレクトリー
- 0x10 アーカイブ0x20
- デバイス: 0x60

これは必要ないので、シリーズでは使用しませんのでご了承ください。ただし、好きな方はご自身のシステムでファイルの属性を操作・設定するサポートを行うことができます。

この構造体のすべての日付メンバーは、特定のビットフォーマットに従います。

- ビット0~4: 曜日 (0~31)
- ビット5-8: 月 (0-12)
- ビット9-15: 年

この構造のすべてのタイムメンバーは、特定のビットフォーマットに従います。

- ビット0~4: セカンド
- ビット5-10分
- ビット11-15: Hour

本連載では、ファイルやディレクトリの日時情報を変更したり取得したりする必要がないため、これらを使用していません。しかし、読者の皆様には、後からでも好きなように機能を追加していただきたいと思います。

FAT12 フォーマットのフロッピーディスクでは、クラスタは 1 セクタ (512 バイト) と同じ大きさであることを覚えておいてください。このため、DIRECTORYの**FirstCluster**フィールドは、ファイルの最初のセクタを指す。したがって、このセクタを読み取ることで、ファイルの最初の512バイトを効果的に読み取ることができます。

それでは、ディレクトリを解析してファイルを探してみましょう。

パーシング

ディレクトリには、ディレクトリエントリ構造のリストが含まれていることを覚えておいてください。このことを知っていれば、ディレクトリを解析してファイルやディレクトリを見つけることがとても簡単になります。

まず、ルートディレクトリの読み込みから始めます。ファイルシステムのマウント時にBPBからルートディレクトリのセクタを取得し、**_MountInfo.rootOffset**に格納したことを思い出してください。したがって、必要なのはセクターをロードして、**DIRECTORY***を使ってディレクトリエントリにアクセスすることだけです。

そして、ファイル名をループして比較し、一致するものを探します。**ToDosFileName()**を使って、入力ファイル名をDOS 8.3のファイル名形式に変換します。例えば、入力ファイル名「Myfile.txt」をFAT12内部フォーマット「MYFILE TXT」に変える。

セクターを読み込んで、セクター内の各エントリを比較します。また、ファイル名が一致するかどうかを単純な**strcmp()**呼び出しでテストできるように、ファイル名をC言語の文字列に変換していることに気づくでしょう。一致するものが見つかったら、**FILE**構造体に記入してそれを返します。

見てみましょう。

DirectoryEntry構造体は、`const char* DirectoryName`名をFILE ます。myfile.txt」のような入力ファイル名を、DOS 8.3ファイルシステムのフォーマットである「MYFILE TXT」に変換し、`DosFileName`に格納します。

```
file;
unsigned char* buf;
PDIRECTORY ディレクトリ
for (int sector=0, sector<14; sector++) {...
    // ! 8.3のディレクトリ名を取得
    char DosFileName[11];
    ToDosFileName (DirectoryName, DosFileName, 11);
    DosFileName[11]=0;
    // ! ディレクトリ情報の取得
    directory = (PDIRECTORY) buf;
```

ルートディレクトリから読み込んでいます。ルートクラスタは、ファイルシステムがマウントされたときに**Bios Paramater Block (BPB)** から取得した情報を含む**_MountInfo**に格納されています。**_MountInfo.rootOffset**には、ルートディレクトリの最初のクラスタが格納されています。ルートディレクトリには、最大で 224 個の **DIRECTORY** エントリが含まれます。1つの**DIRECTORY**エントリは32バイトで、**224*32=7168**バイト、**7168バイト÷512バイト (512バイトで1クラスタ) =14**となります。つまり、ルートディレクトリは**14**個のクラスタで構成されています。

これを知っていれば、ディレクトリ全体を一度に読み込むのではなく、セクターごとに読み込み、各部分を解析することができます。

```
// !セクターあたり16エントリ
for (int i=0; i<16; i++) {

    // ! 現在のファイル名を取得
    char name[11];
```

```
memcpy (name, directory->Filename, 11);
name[11]=0;

//! 一致するものはありますか?
if (strcmp (DosFileName, name) == 0) {...
```

1つのDIRECTORYエンタリが32バイトであることを知ると、1クラスタ512バイト÷32バイト=16。つまり、1つのセクタに16個のDIRECTORYエンタリがあることになります。そこで、各エンタリをループしてファイル名を比較し、探しているファイルやディレクトリを見つけます。**file.currentCluster**には、後で読むためのファイルの最初のクラスタが格納され、**file.fileLength**には、ファイルのサイズがバイト単位で格納されます。**directory->Attrib**には、ファイルの属性が格納されます。ここでは、そのDIRECTORYエンタリ属性に基づいて設定しています。

あと少し...。ファイルやディレクトリがまだ見つからない場合は、次のDIRECTORYエンタリに移動します。ファイルが見つからない場合は、FS_INVALIDを設定して戻ります。

```
定めます strcpy (file.name,
DirectoryName); file.id= 0;
file.currentCluster = directory->FirstCluster;
file.eof= 0;
file.fileLength= directory->FileSize;

//! 次のディレクトリへ
ディレクトリ++。

}

//! ファイルタイプの設定
if (directory->Attrib == 0x10)
file.flags = FS_DIRECTORY;

//! ファイルが見つからない その他
file.flags = FS_INVALID;
return file;

file.flags = FS_FILE;

}

//! ファイルを返す
return file;
```

これで完了です。上記のルーチンはFAT12のディレクトリやファイルに対して動作します。このルーチンを呼び出すと、ルート・ディレクトリ内の任意のフォルダやファイル名を検索し、その情報を返します。

サブディレクトリ

古いバージョンのFAT12はフラットでしたが、このファイルシステムの新しいバージョンではサブディレクトリをサポートしています。これにより、ディレクトリを利用して多くのファイルをより簡単に管理できるようになった。例えば、大規模なOSでは、OS固有のファイルをシステム・ディレクトリに分けたり、ユーザー・プロファイルを含むユーザー・ディレクトリに分けたりするのが良いでしょう。

サブディレクトリとは、普通のファイルにDIRECTORYフラグを設定したものです。このため、まずはファイルの読み方を知る必要があります。

ファイル読み込み

フォーマット

さて、これでディレクトリを解析してファイルを探すことができるようになりました。次に、ファイルの内容を読み取る方法が必要です。技術的には、ファイルのディレクトリエンタリ構造の**FirstCluster**フィールドを指定するだけで、ファイルの最初の512バイトを読み取ることができることを覚えておってください。1つ以上のクラスタを読み取るには、**ファイルアロケーションテーブル (FAT)**を解析する必要があります。

FATはクラスタ番号を含むいくつかのエンタリで構成されていることを思い出してください。これらのエンタリのサイズはファイルシステムに依存します。FAT12はエンタリあたり12ビット、FAT16は16ビット、FAT32は32ビットです。

FATはリンクされたリストではなく、物理的なディスク全体を表すエンタリのテーブルであると考えます。ディスクの最初のクラスタは、FATの最初のエンタリで表されます。ディスクの最初のクラスタはFATの最初のエンタリで表現され、2番目のクラスタは2番目のエンタリで表現され、以下同様です。つまり、クラスタとFATエンタリの間には1対1の関係があります。これにより、FAT12でのファイルの読み書きが容易になります。

ファイルの読み込み

ファイルを読み取るには、そのファイルの現在のクラスタを読み取るだけです。その後、FATテーブルを解析してディスク上の次のクラスタを探します。次のクラスターを見つけたら、次のファイルを読むために「現在のクラスター」を更新します。

読み込むクラスタは、ファイルのオープン時に設定されました。このルーチンの最初の呼び出しでは、**file->currentCluster**は**DIRECTORY->FirstCluster**です。

このクラスタは、ディスク上のデータ領域へのオフセットです。FAT12フォーマットのディスクのフォーマットを思い出して、FATとデータ領域の位置を確認してみましょう。



ブートセクター	エクストラリザーブドセクション	ファイルアロケーションテーブル1	ファイルアロケーションテーブル2	ルートディレクトリ (FAT12/FAT16のみ)	ファイルやディレクトリを含むデータ領域。
---------	-----------------	------------------	------------------	---------------------------	----------------------

各FATは9個のセクタを取ることを覚えておいてください。2つのFATがあるので、 $9+9=18$ となります。また、前節でルートディレクトリが14セクタであると結論づけました。 $18+14=32$ 。これは、FATとルートディレクトリの両方が占めるセクタ数です。ここまでの計算式は、**32 + file->currentCluster**です。1を引く必要があり、**32 + (file->currentCluster - 1)**となります。これが読み込まれるセクタであり、ファイルデータを含みます。

次のコードがFatRead(PFILE, file, unsigned char *Buffer, unsigned int Length)が9セクタあるので、9セクタすべてを読むのではなく、どのセクタを読むべきかを決定する。

まず、次のクラスターがどこにあるかを物理的オフセットを取得します。これを行うためには、クラスターの値にクラスターのサイズを掛けます。これは**FAT_Offset**に格納された**unsigned char* physSector**のサイズが416バイトの場合、**FAT32**を使用している場合は4倍します。**FAT16**を使用している場合は、クラスター・エントリごとに2バイトを使用するので、2倍します。しかし、**FAT12**ではどうでしょうか？**FAT12**はクラスターエントリあたり12ビットを使用する。これは8バイトの値を必要とします(2バイト目は4ビット目は8ビットの半分なので0.5)なので、クラスターエントリあたり1.5ビットになります。

この後、このバイト・オフセットをBufferのsectorで割るだけで、読み込むべきFATのセクタが得られる。リマnderはこのセクタ内のオフセットで、FATから読み出すためのクラスターとなります。これがentryOffsetになります。

FATは**uint8_t** **FAT [SECTOR_SIZE*2]**と定義されています。FATのセクタを1つではなく2つメモリに読み込んでいることに注目してください。なぜこのようなことをするのでしょうか?セクタ・サイズが512バイトであることを知ると、512バイト×8＝4096ビット/セクタとなります。4096バイト÷12ビット(FATエントリの場合)で341.3333...などとなります。つまり、あるエントリは第1セクタと第2セクタの間に位置することになります。これでは、ファイルを読み込むときに問題が生じます。このため、第1セクタの最後のクラスタ値が破損しないように、追加のセクタをロードする必要があります。

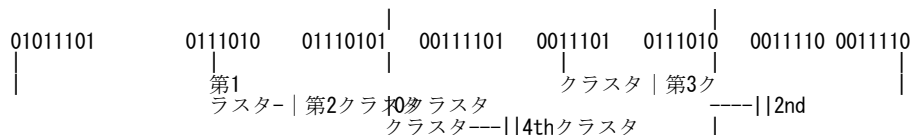
```

FATセクターが読み込まれ、unsigned short *fat_ptrを読み込ませ、currentCluster + (file->currentCluster / 2); //1.5倍にする
int FAT_Sector = 1 + (FAT_Offset / SECTOR_SIZE);
ここで問題が発生します。unsigned intの値をshort型に代入することはできません。そこで、uint16_tを使って
16ビットを読み取ることにします。uint16_tの値は2バイトの値を読み取ることはできません。そこで、uint16_tを使って
第1のFATセクタの読み込み。今度は12ビットの値のビット数が多すぎるという問題があります。
sector = (unsigned char*) fplydisk_read_sector ( FAT_Sector ); memcpy
もう少し詳しく見てみましょう。FAT_Sectorは512のFATだとします。ここではFATをバイトに分割し、12ビットのエントリをマークします。(これは
第6章からの引用です。)

```

```
// ! 第2のFATセクターの読み込み
sector = (unsigned char*) fipdyisk_read_sector ( FAT_Sector + 1 ); memcpy
(FAT + SECTOR_SIZE, sector, 512);
```

注： バイト単位で区切られた2進数。各12ビットの
FATクラスター・エントリが表示されます。



偶数クラスタは第1バイトのすべてを、第2バイトの一部を占めていることに注目してください。また、すべての奇数クラスタは、第1バイトの一部をオコピーしますが、第2バイトのすべてをオコピーすることに気がつきます。

このことを念頭に置くと、クラスタが偶数の場合は、次のクラスタに属するため、上位4ビットをマスクアウトする。クラスターが奇数の場合は、4ビット下にシフトします（最初のクラスターで使用したビットを破棄するため）。

さて、以上のことを踏まえた上で、この関数を完成させましょう。

```

//! 次のクラスタのエントリを読む
uint16_t nextCluster = *( uint16_t*) &FAT [entryOffset];

//! エントリーが奇数か偶数かをテストする
if( file->currentCluster & 0x0001 )
    nextCluster >>= 4; //上位12ビットを取得
その他

```

```

        nextCluster &=
        0xFFFF; //下位12ビットを取得

        // ! ファイルの終わりのテスト
        if ( nextCluster >= 0xff8 ) {...

            file->eof = 1;
            return;

        }

        // ! ファイル破損のテスト if (
        nextCluster == 0 ) {...

            file->eof = 1;
            return;

        }

        // ! 次のクラスターを設定
        file->currentCluster = nextClusterとなります。
    }
}

```

ファイルの書き込み

[(誠人の声) チャプターアップデートで完成するんだ]

サブディレクトリ

サブディレクトリとは、**DIRECTORY**属性が設定されたファイルのことです。サブディレクトリから読み取るためには、そのディレクトリ名を持つディスク上の**FAT12**ファイルを探し出し、**FAT**を使った他のファイルと同様の方法で読み取ればよいのです。

ファイルが読み込まれた後、最初のバイトから最後のバイトまでは、単なる**DIRECTORY**エントリーの配列です。このディレクトリを読むためには、ルートディレクトリと同じように**DIRECTORY**エントリーを解析します :-)これがディレクトリ内のファイルやフォルダになります。

見てみましょう。

```

FILE fsysFatOpenSubDir (FILE kFile,
                        const char* filename) {

    FILEファイルです。

    // ! 8.3のディレクトリ名を取得
    char DosFileName[11];
    ToDosFileName (filename, DosFileName, 11);
    DosFileName[11]=0;
}

```

filename は検索したいファイルまたはディレクトリ、**kFile** は解析したいサブディレクトリです。**myfile.txt**」のような入力ファイル名を、DOS 8.3ファイルシステムのフォーマットである「**MYFILE TXT**」に変換し、**DosFileName**に格納します。

```

// ! ディレクトリの読み込み while (! kFile.eof) {
// ! ディレクトリの読み込み unsigned char
// ! ディレクトリの読み込み buf[512];
// ! ディレクトリの読み込み fsysFatRead (&file, buf, 512) です。
// ! セットダイレクトート PDIRECTORY pkDir = (PDIRECTORY) buf;

```

ファイルは、解析したいサブディレクトリです。**FAT12**では普通のファイルなので、ファイルのセクタを読み込むことになります。ファイルは**DIRECTORY**エントリーの配列で構成されています。**DIRECTORY**のメンバーに簡単にアクセスできるように、**pkDir**を使ってセクターの内容を指し示します。それでは、ディレクトリを検索してみましょう...

```

// ! バッファの16エントリ
for (unsigned int i = 0; i < 16; i++) {...

    // ! 現在のファイル名を取得
    char name[11];
    memcpy (name, pkDir->Filename, 11);
    name[11]=0;

    // ! マッチ?
    if (strcmp (name, DosFileName) == 0) {...

```

DIRECTORYの各エントリは32バイトです。セクター(**FAT12**のクラスターでもある)は512バイトです。**512バイト÷32バイト=16個**の**DIRECTORY**エントリがセクタごとに存在します。そこで、**16個**のエントリすべてをループさせて名前を比較します。検索しているファイル名と一致するファイル名が見つければ、そのファイルは見つかったことになります。

```

// ! 見つかったので、ファイル情報を設定 strcpy (file.name, filename);
file.id=0;
file.currentCluster = pkDir->FirstCluster;
file.fileLength= pkDir->FileSize;
file.eof=0;

```

```

        }
        // ! ファイルタイプの設定
        if (pkDir->Attrib == 0x10)
            ファイル.flags = FS_DIRECTORY;
        その他
            file.flags = FS_FILE;

        // ! ファイルを返す
        return file;
    }

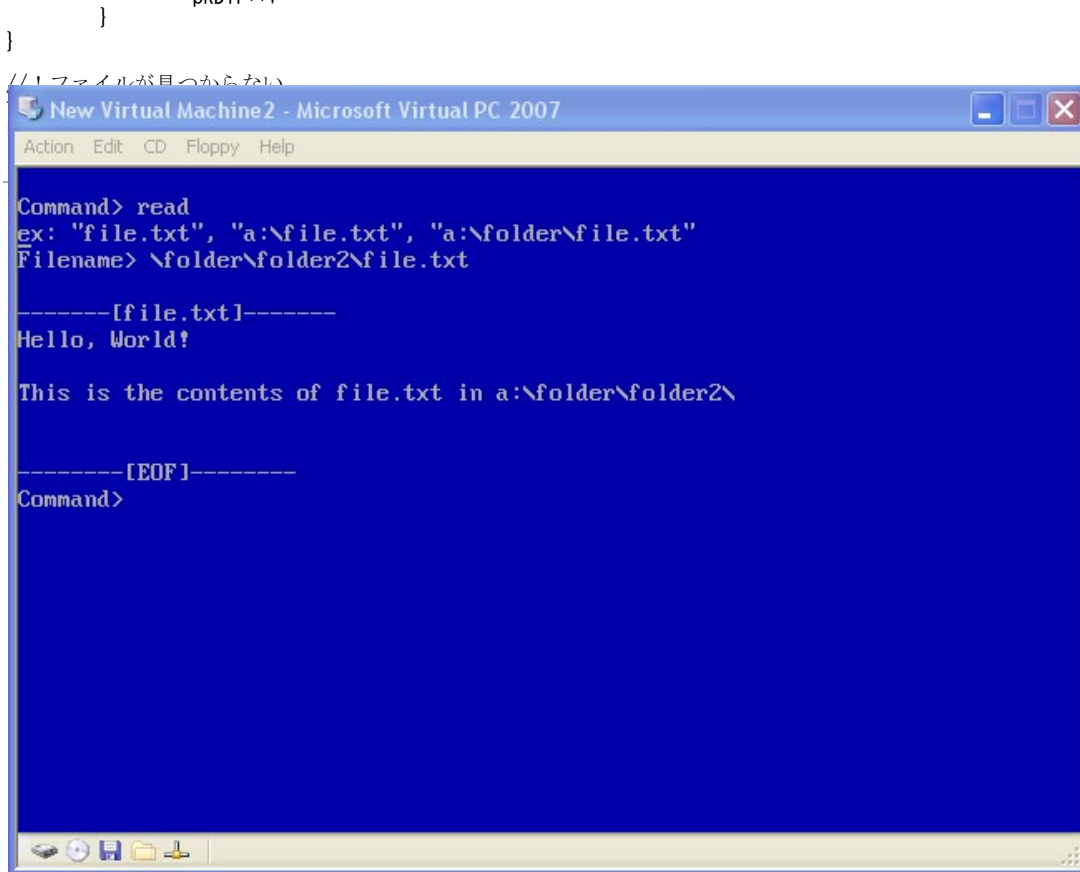
```

ファイルが見つかると、**FILE**構造を記入します。最初のファイルクラスタ（後で読めるように）、ファイルサイズ（EOFがいつなのかわかるように）、属性（ファイルまたはディレクトリ）です。

ファイルが見つからなかった場合は、次のエントリに進むだけです。このループは、ファイルの最後まで続きます。ファイルが見つからなかった場合は、**FS_INVALID**を設定して戻ります。

このルーチンと私たちの **FsysFatDirectory** エントリに類似点に注目してください。

デモ



OSでファイルを見る
デモダウンロード

本章のデモでは、これまでに説明した内容をすべて盛り込み、**VFS**と**FAT12**のミニドライバを実装しています。複数のファイルシステムのサポート、ディスク・デバイスのサポート、サブディレクトリのサポート、ファイルの読み込みと表示が可能です。

また、このデモは大きなファイルを表示することができ、マルチクラスターファイルのための「キーを押して続行」機能を実装しています。こ

のデモでは、CRT の **string.c** に **strchr()** ISO C ルーチンを組み込み、テキストの解析を支援しています。また、このデモでは **read** コマンドを使って、生のセクタではなく、ファイルを探して表示できるようになりました。

このデモでは、ボリュームマネージャは非常にシンプルで、**fsys.cpp**に実装されています。ファイルシステムの登録と解除、ファイルシステムの抽象化を管理します。**volOpenFile()**を呼んでファイルを開くことができます。デフォルトでは**a:file.txt**を開きますが、任意のディレクトリの任意のファイルを開くように呼び出しても動作します。

すべてのファイルシステムがサブディレクトリをサポートしているわけではありません。そのため、サブディレクトリのサポートはファイルシステムのドライバに任せています。その代わりに、ボリュームマネージャはパス名のドライブレター部分のみを処理します。たとえば、**volOpenFile ("a:\\folder\\file.txt")** を呼び出すと、**volOpenFile**は「**\\folder\\file.txt**」をデバイス「**a**」に登録されているファイルシステムに渡します。ファイルシステムのドライバは、ディレクトリのパス名を解析し、サブディレクトリやファイルを開く役割を果たします。

FAT12ミニドライバの場合、この特別なルーチンは**fsysFatOpen()**であり、ディレクトリ・パス(例えば"`..`")を解析し、ファイルやディレクトリを解析して読み取るための他のファイル・システム・ルーチンと呼び出す役割を担っている。

以上です :-)この章がFAT12を扱う最後の章となるでしょう。このため、ファイルやディレクトリのディスクへの書き込みについては、もう少し後にアップデートを予定しています。

結論

この章は楽しかったですね。ディスクからファイルを読み込むことができるようになりました。分かっている、分かっている、"そろそろだな!"と。:)これで、マルチタスクやプログラムの実行に向けて、大きく飛躍することができます。しかし、マルチタスク化の前に、ローダーについて説明しておきましょう。ローダーは、プログラムの読み込みと実行、アドレス空間へのマッピングを行います。また、アドレス空間でのヒープ管理やスタック管理についても説明する必要があります。

メモリ管理の章を大幅に更新する予定なので、ヒープとスタックの管理をメモリ管理の章の次の章に移すかもしれません。いずれにしても、変更点については随時お知らせしていきます。

しかし、これは、私たちがマルチタスクに飛び込む時期が近づいていることを意味しています。その後は?ユーザーモード

です。では、次回をお楽しみに。

マイク

BrokenThorn Entertainment社。現在、DoEと[NeptuneOperating System](#)を開発中です。質問や

コメントはありますか?お気軽に[お問い合わせください](#)。

あなたも記事の改善に貢献したいと思いませんか?もしそうなら、ぜひ[私に教えてください](#)。



第21章

ホーム

第23章





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - ユーザーランド

by Mike, 2010

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

Welcome!

前章では、**VFS**を見て、テキストファイルを読み込んで表示しました。この**VFS**を使って、実行可能なプログラムファイルも読み込むことができます。これには、ドライバ、プログラムソフト、共有、ランタイムライブラリなどが含まれます。

この章では、ユーザーランド・ソフトウェアのサポートに踏み込んでいきます。また、システム**API**とその仕組みについてもご紹介します。

さあ、始めましょう。

保護レベル

アセンブリ言語の輪

カーネルランド

第5章では、アセンブリ言語で使われているリングの概念を簡単に説明しました。このリングは異なる保護レベルを表しています。これらの保護レベルは、ハードウェアの詳細であり、ハードウェアによって実装されます。

リング**0**で動作するソフトウェアは、最も制御しやすい。例えば、ハードウェア**PIO**、**MMIO**、プロセッサのハードウェア制御やテーブル（CPUのキャッシュ制御や**MMR**など）などがあります。

特権的な命令のリストは第7章に示されていますが、ここでは完全なものとして記載します。

保護レベルが**0**より大きい状態で実行されているソフトウェアが上記の命令を実行しようとした場合、プロセッサは以下を生成します。**保護障害（#PF）**の例外。

特権レベルの指示	
インストラクション	説明
LGDT	GDTRにGDTのアドレスをロードする
LLDT	LDTのアドレスをLDTRにロードする
LTR	タスクレジスタをTRにロードする
MOV コントロールレジスタ	データをコピーしてコントロールレジスタに格納
LMSW	新しいマシンのロード ステータス WORD
CLTS	コントロール・レジスタのタスク・スイッチ・フラグをクリア CR0
MOV デバッグレジスタ	データをコピーしてデバッグ用レジスタに格納
INVD	ライトバックしないでキャッシュを無効にする
INVLPG	TLBエントリの無効化
WBINVD	ライトバックでキャッシュを無効にする
HLT	ハルトプロセッサ
RDMSR	モデル・スペシフィック・レジスター（MSR）の読み出し
WRMSR	MSR（Model Specific Registers）の書き込み
RDPMSR	パフォーマンスモニタリングカウンターの読み込み
RDTSC	読み取り時間 スタンプカウンタ

このため、リング**0**で動作するソフトウェアを**カーネルランド**または**カーネルモード**と呼びます。リング**0**は**スーパーバイザーモード**とも呼ばれます。

これまでのシリーズで書いてきたソフトウェアは、すべてカーネルモードのソフトウェア、つまりカーネルとミニドライバでした。マイクロカーネルやハイブリッドでは、通常、シリーズで使用しているものよりも高度なドライバ・インターフェーシング・スキームを採用しており、ドライバを適切にインストールしたり、ドライバをカーネルとは完全に分離したユーザー・モードで実行したりすることができます。カーネルの一部をユーザーモードにすることも可能ですが、それは設計次第です。

システムの初回起動時には、**BIOS**や**OS**が起動できるように、システムはスーパーバイザーモードで動作しています。

ユーザーランド

2021/11/15 13:11

オペレーティングシステム開発シリーズ

これは、マシンを保護するためのもので、リング1〜3で動作するソフトウェアに起因するエラーが発生した場合、プロセッサは**一般保護（#GP）**例外を使用してシステムエグゼクティブまたはカーネルに問題を通知します。

ほとんどのオペレーティングシステムは、カーネルモードとユーザーモードの2つのモードシステムを採用しています。x86ファミリーでは4つの保護モードをサポートしていますが、これらのオペレーティングシステムでは、アーキテクチャ間の移植を容易にするために2つのモードのみを使用しています。

これらのOSでは、カーネルモードのソフトウェアはリング0で動作し、ユーザーランドのソフトウェアはリング3で動作するように設計されています。リング1と2は使用されません。ドライバー・ソフトウェアは、ハードウェア・デバイスにアクセスするためにリング0で動作するか、提供されているドライバーAPIまたはシステムAPIを使用してハードウェア・デバイスと通信するためにリング3で動作します。

ユーザーモードのソフトウェアは、ハードウェアデバイスに直接アクセスすることができないため、システムのタスクを完了するためにオペレーティングシステムに通知する必要があります。これは、文字の表示、ユーザーからの入力、文書の印刷などを含みます。これらの機能は、ライブラリやAPIの形でユーザーモードソフトウェアに提供される。これらのライブラリやAPIは、システムAPIと通信します。

システムAPI.....この言葉を目にしたことがある人は多いだろう。システムAPIについては、もう少し詳しく見てみましょう。今回は、ユーザーモードについて詳しく見ていきましょう。

リング -1

最近のプロセッサには、ハイパーバイザーのリング0アクセスを許可する特別な保護レベルを持つものがあります。これは「リング」と呼ばれることもあります。

-1".

ユーザーランドへようこそ

ユーザーモードに入るにはいくつかのステップがあります。(さあ、簡単だとは思わなかったでしょう。)でも、それほど悪いことはありません。

ステップ1：グローバル記述子テーブル

まず、グローバルディスクリプターテーブル(GDT)に戻る必要があります。GDTは、プロテクトモードを初めて設定する際に必要となる、大きな醜い構造体でした。GDTには、プロセッサの情報を含む8バイトのエントリのリストが含まれています。GDTのエントリのビットフォーマットをもう一度見てみましょう。(重要な部分は太字にしました)

- **56〜63ビット** 目ベースアドレスの24〜32ビット目
- **ビット55** : グラニュラリティ
 - **0** : なし
 - **1** : リミットが4K倍になります。
- **ビット54** : セグメントタイプ
 - **0** : 16ビット
 - **1** : 32ビット
- **ビット53** : 予約済み-0にすべき
- **ビット52** : OS用の予約
- **ビット48〜51** : セグメントリミットのビット16〜19
- **ビット47** : セグメントがメモリ内にある (仮想メモリで使用)
- **ビット45-46記述子の特権レベル 0: (Ring**
 - **0) Highest**
 - **1: (Ring 1)**
 - **2: (Ring 2)**
 - **3 : (リング3) 最下位**
- **ビット 44** : ディスクリプタータイプ
 - **0** : システム記述子
 - **1** : コードまたはデータ記述子
- **ビット41-43** ディスクリプター・タイプ
 - **ビット43** : 実行可能セグメント
 - **0** : データセグメント
 - **1** : コードセグメント
 - **ビット42** : 拡張方向 (データセグメント)、準拠 (コードセグメント)
 - **ビット41** : 読み出し可能、書き込み可能
 - **0** : 読み出しのみ (データセグメント)、実行のみ (コードセグメント)
 - **1** : 読み取りと書き込み (データセグメント)、読み取りと実行 (コードセグメント)
- **ビット40** : アクセスビット (仮想記憶で使用) **ビット16〜39** **ビット16-39** : ベースアドレスの0-23ビット **ビット0-15** : セグメントリミットの0-15ビット

そうですか。上記の**DPL (Descriptor Privilege Level)** ビットは、そのディスクリプターに使用される特権レベルを表しています。つまり、これらのビットを3に設定することで、その記述子を実質的にユーザーモードの記述子にすることができます。

そこでまず、GDTに2つの新しいディスクリプターを作成します。1つはユーザーモードデータ用、もう1つはユーザーモードコード用です。これは、**i86_gdt_initialize**を修正して、ユーザーモードコードとデータのための2つの新しいGDTエントリを追加することで行われます。それでは早速やってみましょう。

```
gdtの初期化
int i86_gdt_initialize () {

    // ! などなど...。

    // ! デフォルトのユーザーモードコード記述子を設定
    gdt_set_descriptor (3,0,0xffffffff),
        i86_gdt_desc_readwrite|i86_gdt_desc_exec_code|i86_gdt_desc_codedata|
        i86_gdt_desc_memory|i86_gdt_desc_dpl,
        i86_gdt_grand_4k | i86_gdt_grand_32bit | i86_gdt_grand_limithi_mask) を使用していま
        す。)

    // ! デフォルトのユーザーモードデータ記述子を設定
    gdt_set_descriptor (4,0,0xffffffff),
        i86_gdt_desc_readwrite|i86_gdt_desc_codedata|i86_gdt_desc_memory|
        i86_gdt_desc_dpl,
```

```

        i86_gdt_grand_4k | i86_gdt_grand_32bit | i86_gdt_grand_limit_mask) を
        使用しています。)

    等々....

    0を返す。
}

```

上記のコードは、他のGDTエントリを作成するときに行ったものと同じですが、1つだけ変更があります。I86_GDT_DESC_DPLフラグに注目してください。これにより、両方のDPLビットが2に設定され、ユーザーモード（リング3）用になります。これらのフラグはすべて、以前の章でプロテクトモードについて説明したときのものです。

必要なのは、これだけです。なお、ユーザーモードコード記述子はGDTのインデックス3に、ユーザーモードデータ記述子はインデックス4にインストールされています。セグメントレジスタには、使用するセクタのオフセットが含まれていることを覚えておいてください。各GDTエントリは8バイトサイズなので、**コードセクタ0x18 (8*3)**、**データセクタ0x20 (8*4)**となります。

したがって、これらのセクタを使用するには、上記のセグメントセクタの1つを、使用するセグメントレジスタにコピーするだけです。

DPL

DPL (Descriptor Protection Level) とは、セグメント記述子の保護レベルのことです。例えば、カーネルのコードセグメントとデータセグメントのDPLは、リング0のアクセスに対して0となります。

RPL

Requested Protection Level (RPL) は、ソフトウェアがCPLをオーバーライドして新しい保護レベルを選択できるようにするものです。これにより、ソフトウェアは、リング0からリング3など、他の保護レベルの変更を要求することができます。RPLはディスクリプターセクタのビット0と1に格納されています。

待って、何? セグメントセクタは、GDTへの単なるオフセットであることを覚えておいてください。例えば、0x8バイトはリング0のコード記述子のオフセットです。0x10は、データセクタのオフセットでした。0x8と0x10は**セグメントセクター**です。GDTエントリはすべて8バイトなので、セグメントセクターの値は常に8、16、24、32など、8の倍数になります。8は2進法では1000です。つまり、セグメントセクターの値がどのようなものであっても、下位3ビットは0になります。

RPLは、セグメントセクタの下位2ビットに格納されます。セグメントセクターが0x8の場合、RPLは0になります。0xb (0x8だが、最初の2ビットがセットされており、1000ではなく1011のバイナリ) の場合、RPLは3になります。

CPL

カレントプロテクションレベル (CPL) は、現在実行中のプログラムの保護レベルです。CPLはSSとCSのビット0と1に格納されています。

GDTエントリのサイズは8バイトであることを覚えておいてください。プロテクトモードのセグメントレジスタには、セグメントセクタ (GDTエントリのオフセット) が含まれているため、下位3ビットはゼロであることが保証されています。CSとSSの下位2ビットは、ソフトウェアのCPLを格納するために使用されます。

保護レベル

ソフトウェアがセグメント・レジスタに新しいセグメントをロードしようとする、プロセッサはソフトウェアのCPLとロードしようとしているセグメントのRPLとのチェックを行います。RPLがCPLよりも高ければ、ソフトウェアはセグメントをロードできません。そうでない場合、プロセッサは**一般保護フォルト (#GP)** を発生させます。

RPLの仕組みを理解しておくことは、ユーザーモードに切り替える際に必要な情報となります。

ステップ2: スイッチ

これで、ユーザーモードに切り替えることができます。

ジャンプを実行するには2つの方法があります。**SYSEXIT**命令を使う方法と、**IRET**を使う方法です。どちらの方法にも一長一短がありますので、詳しく見ていきましょう。本連載では、移植性を考慮してIRETを使用しています。

SYSEXIT命令

このセクションは、今後も発展させていく予定です。

IRET / IRETD 命令

SYSEXITを使うよりも移植性が高いため、多くのOSがこの方法を採用しています。より大きなOSでは、SYSEXITが使えない場合のバックアップ方法として、この方法をサポートしているかもしれません。

さて、ではIRETはどのようにして切り替えを行うのでしょうか。第3章で、モードを切り替えるときの方法を思い出してください。IRETはトラップリターン命令です。IRETを実行すると、ユーザーモードのコードに戻るようにスタックフレームを調整することができます。

IRETDが実行されると、スタックには以下のものがあると期待されます。

- SS
- ESP
- EFLAGS
- CS
- EIP

IRETDにより、プロセッサはスタックから取得したCS:EIPにジャンプします。また、スタックから上記の値をEFLAGSレジスタに設定します。

SS:ESPには、スタックから取得したSSとESPの値が設定されます。ズ

これらの値は、**INT**命令が実行されると自動的にスタックにプッシュされます。このため、通常の場合、これらの値は変更されません。しかし、これらの値を変更することで、**IRET**にモードスイッチを実行させることができます。

さて、まずはセグメントセクターの設定です。下位2ビットが希望するRPLを表すことを思い出してください。ここでは、ユーザーモードに3を設定します。では、その設定を行います。

```
void enter_usermode () {
    _asm {
        クライアント
        mov     ax, 0x23; ユーザーモードのデータセクタは0x20 (GDTエントリ3) です。また、RPL
        を3に設定 mov ds, ax
        mov es, ax
        mov fs, ax
        mov gs, ax
    }
```

ここで、ユーザーモードへの切り替えを行います。これは、**IRET**用のスタックフレームを構築し、**IRET**を発行することで行われます。

```
    プッシュ                SSでは、上記と同じセクタを使用しています。
    0x23 プッシュ            ESP
    シュ esp                EFLAGS
    シュ fd                 CSでは、ユーザーモードコードセクターは0x18です。RPL 3では0x1bとなります。
    プッシュ                EIPが先
    0x1b
    lea eax, [a]
    push eax
a:
    add esp, 4 // スタックの
    固定
}
```

スタックフレームが上のリストにあったものと一致していることに注目してください。**IRETD**命令により、上記のコードではリング3内で0x1B:aが呼び出されます。

しかし、ちょっとした問題があります。上記のルーチンを使用したり、カーネル内で別の方法でユーザーモードに切り替えようとすると、**ページフォルト (PF)** の例外が発生します。これは、カーネルのページがカーネルモードアクセス専用マップにマッピングされているためです。この問題を解決するには、別の方法でユーザーモードに移行するか、ユーザーモードのソフトウェアがアクセスできるようにカーネルをマッピングする必要があります。

今のところ、ユーザーモードのソフトウェアがアクセスできるようにカーネルをマッピングするだけです。これには**vmngr_initialize()**の更新が必要です。

ルーチンで、PTEとPDEのUSERビットを設定します。

より複雑なオペレーティングシステムでは、このアプローチは使われません。この方法は、ユーザーモードのソフトウェアがアクセスできるようにカーネルページをマッピングした場合にのみ機能しますが、これは良くありません。より推奨されるアプローチは、カーネルだけのアクセスのためにカーネルページをマップしておき、ユーザプログラムをロードするときに、カーネルのロードコンポーネントがユーザーモードページをマップするようにすることです。スタックとヒープのアロケータは、プログラムのスタックとヒープの領域をユーザーモードにマッピングします。この現在の方法では、カーネルのスタックをユーザーランドと共有しています。大きなシステムでは、このような方法をとるべきではありません。

v8086モードへの移行

v8086モードに入るためには、ユーザーモードのタスクが必要です。したがって、上記の手順を実行すれば、v86モードにも入ることができます。ただし、1つだけ若干の修正が必要です。

EFLAGSレジスタのフォーマットを思い出してください。ビット17 (VM) は、**v8086モード制御フラグ**です。IRETを実行する際にEFLAGSの値をスタックにプッシュするので、v86モードに入るためには、EFLAGSのビット17をセットしてからスタックにプッシュすればよい。これにより、IRETはリターン時にEFLAGSレジスタのVMビットを設定します。

これだけで、v8086モードに入ることができます。

デザインに関する注意事項

上記の方法は、ユーザーモードに入るための簡単な方法ですが、コストがかかります。上記の方法が機能するためには、リング3ソフトウェアがカーネルメモリにアクセスできるようにカーネル領域をマッピングする必要があります。このため、リング3で実行中のソフトウェアは、プロテクトモードによる制限はあるものの、カーネルルーチンを直接呼び出したり、カーネル空間をゴミ箱に入れたりすることができます。

上記の問題を解決する方法として、カーネルメモリをリング0ソフトウェア用に予約しておくことが考えられます。カーネルのロードコンポーネントは、プログラムをロードしている間に、プロセスに必要なリング3のメモリ領域をマッピングすることができます。

この点については、次の章でOS用のローダーを開発する際に詳しく説明します。

カーネルランドへの切り替え

ステップ1: TSSの設定

x86アーキテクチャは、ハードウェアによるタスク切り替えをサポートしています。つまり、プロセッサが異なるタスクを選択できるように、ハードウェアで定義された構造がアーキテクチャに含まれています。

2021/11/15 13:11

オペレーティングシステム開発シリーズ

最近のOSの多くは、移植性を考慮して、ハードウェアのタスクスイッチングを利用していません。これらのOSでは、一般的にソフトウェアによるタスク切り替え方式を採用しています。

タスクステートセグメント (TSS)

TSSの構造はかなり大きい。

TSSは、**MSDOS**で導入されたタスクスイッチが行われる前のマシンの状態に関する情報を格納するために使用されます。たくさんのメンバーがあるので、見てみましょう (push, 1)

```
#ifndef _MSC_VER...
#pragma pack (push, 1)
#endif

struct tss {
    uint32_t prevTSS;
    uint32_t cr3;
    uint32_t esp0;
    uint32_t iomap;
    uint32_t esp1;
    uint32_t esp2;
    uint32_t ss0;
    uint32_t ss1;
    uint32_t ss2;
    uint32_t eip;
    uint32_t eax;
    uint32_t ecx;
    uint32_t ebx;
    uint32_t esp;
    uint32_t ebp;
    uint32_t esi;
    uint32_t edi;
    uint32_t cs;
    uint32_t ss;
    uint32_t fs;
    uint32_t gs;
    uint32_t ldt;
    uint16_t iomap;
};
```

これらのフィールドは、**MSDOS**で導入されたタスクスイッチが行われる前のマシンの状態に関する情報を格納するために使用されます。このため、この構造体のいくつかのフィールド、特にリング0スタックとセクタフィールドを設定する必要があります。

Step 2: TSSのインストール

記述子セグメント

TSSはその名前からもわかるように**セグメント**です。他のセグメントと同様に、TSSにも**GDT**へのエントリが必要です。これにより、TSSを制御することができます。タスクが**ベシ**にインアクティブかの設定、どのソフトウェアがアクセスできるか (DPL)、その他のフラグを記述子で設定することができます。

Base Addressフィールドには、弊社が設定したTSS構造のベースアドレスを入力してください。

LTR命令

LTR (Load Task Register) 命令は、**TSR**レジスタにTSSをロードするための命令です。例えば、以下のようになります。

```
uint16_t iomap;
};

void flush_tss (uint16_t sel)
#ifdef _MSC_VER...
#pragma pack (pop, 1)
#endif
_asm ltr [sel]
}
```

axは、TSSのセグメントセクタです。このアーキテクチャは、ハードウェアによるタスク切り替えをサポートしているため、TSRには、現在のタスクを定義するTSSのアドレスが格納されています。

タスクステートレジスタ (TSR) は、**TSSセクタ**、**TSSベースアドレス**、**TSSリミット**を格納するレジスタです。ただし、ソフトウェアで変更できるのは、**TSSセクタのみ**です。

TSSのインストール

TSS構造をインストールするには、まず**TSS**のGDTエントリをインストールします。を呼び出して、**TSS**を現在のタスクとして選択します。上記の**flush_tss**。

```
void install_tss (uint32_t idx, uint16_t kernelSS, uint16_t kernelESP) {...

    // ! TSSのディスクリプターをインストール
    する uint32_t base = (uint32_t)
    &TSS;
    gdt_set_descriptor (idx, base, base + sizeof (tss_entry),
        I86_GDT_DESC_ACCESS|I86_GDT_DESC_EXEC_CODE|I86_GDT_DESC_DPL|I86_GDT_DESC_MEMORY,
        0);

    // ! TSSの初期化
    memset ((void*) &TSS, 0, sizeof (tss_entry));

    TSS.ss0 = kernelSS;
    TSS.esp0 = kernelESP;

    TSS.cs=0x0b;
    TSS.ss=0x13;
    TSS.es=0x13;
    TSS.ds=0x13;
    TSS.fs=0x13;
    TSS.gs=0x13。

    // !!!フラッシュtss
    flush_tss (idx * sizeof (gdt_descriptor)) です。
}
```

上記のコードでは、**TSS**は**tss_entry**構造体のグローバル構造体定義となっています。**TSSs**のセクタエントリを、前のタスク（ユーザーモードセクタ）とリング0スタック（カーネルスタック、kernelSS:kernelESPに位置する）に合わせて設定します。**flush_tss**はTSSをTSRにインストールします。

追加指示

この他にも便利な命令がいくつかあります。これらの命令はすべて、ユーザーモードのソフトウェアで実行できます。

VERR命令

VERR (Verify Segment is Readable) は、セグメントが読み取り可能かどうかを確認するために使用できます。読み取り可能であれば、プロセッサはゼロ・フラグ (ZF) を1に設定します。この命令はどのプロビジョンスペックでも実行できます。

```
verr [ebx]
jz .readable
```

VERW命令

VERW (Verify Segment is Writable) は、セグメントが書き込み可能かどうかを確認するために使用できます。書き込みが可能であれば、プロセッサはゼロ・フラグ (ZF) を1に設定します。この命令はどのプロビジョンスペックでも実行できます。

```
lslw [ebx]
jz .readable
```

この命令は、セクタのセグメントリミットをレジスタにロードするために使用できます。

```
arpl ebx, esp
jz .success
```

この命令は、セクタのRPLを調整するために使用することができます。この命令は**arpl dest,src**という形式で、**dest** はメモリローションまたはレジスタ、**src** はレジスタです。**dest**のRPLが**src**よりも小さい場合、**dest**のRPLビットは**src**のRPLビットに設定されます。例えば、以下のようになります。

```
arpl ebx, esp
```

システムAPI

アブストラクト

システムAPIは、ソフトウェアがオペレーティングシステムと対話するためのツール、ドキュメント、およびインターフェースを提供する。オペレーティングシステムによって用語が異なる場合がありますが、基本的な考え方は同じです。例えば、**Windows**ではこのAPIを「ネイティブAPI」と呼んでいます。

システムAPIは、ソフトウェアがオペレーティングシステムやデバイスドライバと相互作用することを容易にする。システムAPIは、ユーザーモードのソフトウェアとカーネルモードのソフトウェアの間のインターフェースである。ソフトウェアがシステムの情報を必要としたり、ファイルの作成などのシステムタスクを実行する場合、ソフトウェアはシステムコールを呼び出すことになる。

システムコールは、**システムサービス**とも呼ばれ、オペレーティングシステムが提供するサービスのことで、このサービスは通常、機能やルーチンです。ソフトウェアは、システムのタスクを実行するためにシステムコールを呼び出すことができます。

デザイン

sysenter / sysexit

このセクションは、今後も発展させていく予定です。

ソフトウェアインタラプト

ほとんどのシステムAPIは、ソフトウェア割り込みを使って実装されています。ソフトウェアは**int 0x21**のような命令を使ってオペレーティングシステムのサービスを呼び出すことができます。例えば、**DOS**の**Terminate**関数を呼び出すには次のようにします。

```
mov ax, 0x4c00 ; 関数 0x4c (終了) リターンコード 0 int 0x21
; DOSサービスの呼び出し
```

上のコードでは、**AH**は関数番号を含んでいます。**int 0x21**は、**DOS**を呼び出すために**0x21**の割り込みベクターを呼び出します。

上記を動作させるためには、**OS**は割り込みベクトル**0x21**の**ISR**をインストールする必要があります。この**ISR**は、**AH**を比較して正しいカーネルモード関数に制御を渡す**FSM (Finity State Machine)** となります。以上が、読者の皆様へのデザインです。

ソフトウェア割り込みは、**SYSENTER**や**SYSEXIT**よりも移植性に優れています。このため、ほとんどの**OS**がこの方法をサポートしています（他の方法と一緒にサポートしている場合もあります）。このシリーズでは、この方法を使用します。

例

システムAPIは通常、数百のシステムコールで構成されています。

これは、いくつかのオペレーティングシステムと、それらがサポートしている方法の一覧です。**INT**番号は、上記の方法によるソフトウェア割り込みベクター番号です。

- **DOS** : **INT 0x21**
- Win9x (95,98)**の場合。 **INT 0x2F**
 - **WinNT(2k,XP,Vista,7)**の場合。 **INT 0x2E**, **SYSENTER/SYSEXIT**, **SYSCALL/SYSRET**
 - 211以上の機能
- **Linux**です。 **INT 0x80**, **SYSENTER/SYSEXIT**
 - 190以上の機能

基本システムAPI

ステップ1：システムコールテーブル

ほとんどのシステムAPIは、すべてのサービスを含むシステムコールテーブルを実装しています。このテーブルには、スタティック、ダイナミック、自動生成、またはその3つの組み合わせがあります。大規模なオペレーティングシステムでは、通常、システムコールの自動生成されたダイナミックサイズのテーブルを採用しています。これは、このテーブルに含まれる可能性のあるシステムサービスの数が多いため、手作業で作成するのは非常に面倒です。

今回の目的では、カーネル内にシステムサービステーブルを定義すればよいでしょう。このテーブルには、カーネルの中で呼び出し可能にしたいさまざまな関数のアドレスが含まれています。

うーん、この表は**Linux**と**Windows**ですね。次の章では、このリストにさらに追加していきますが、それほど複雑なものではありません。

DebugPrintfは**syscalls**からアクセス可能であり(カーネルページがマッピングされているため)、**DebugPrintf**は**privedge**命令を使用していないため、ユーザーモードのソフトウェアは技術的には問題なくこのルーチンを直接呼び出すことができます。オペレーティングシステムや実行ソフトウェアの設計によっては、これがセキュリティや安定性の問題を引き起こす可能性があります。

このような理由から、**DebugPrintf**、カーネルページはカーネルモードからのみアクセスできるようにしておくことが推奨されています。ソフトウェアに複雑さをもたらしますが、最終的にはその努力に見合う結果が得られるかもしれません。

ステップ2：サービスディスパッチャ

次のステップは、サービスディスパッチャのISRを作ることです。その前に、どのISRを使うかを決めなければなりませんが、ここではLinuxに従って0x80を使います。しかし、多くのOSでは異なるベクターを使用しているため、好きなベクターを使用することができます。では、ISRをインストールしてみましょう。

ISRはHAL層が管理するIDTに格納されていることを思い出してください。また、第15章では、各IDT記述子がそれぞれのDPL設定を持っていることを思い出してください。**IDTエンティティのDPLがCPLよりも小さい場合、GPFが発生します。**つまり、ユーザーモードに入ると、DPL 3のIDTディスクリプタを持つISRしか呼び出せないのです。リング3のソフトウェアからシステム割り込みを呼び出せるようにするには、このISRを正しいフラグでインストールする必要があります。

しかし、現在のHALサブシステムの設計では、`setvect()`を呼び出すだけでは、特定のフラグを設定することができないため、これを行うことができません。この問題を回避するために、**`setvect()`に第2パラメーターを追加し、オプションのフラグを設定できるようにしました。**これはC++のデフォルトパラメータ機能を利用しているため、他のコードを更新する必要はありません。

それがすべてで、`syscall_init()` {

`syscall_dispatcher`はシステムドライバ用のISRです。このISRは、`_syscalls`で関数を検索して、どのシステムサービス呼び出すかを決定する必要があります。通常、`ret`のAPIはEAXを使い、関数番号を識別し、`DESC`でこれを同じことをするつもりです。上で定義したシステムサービステーブルのおかげで、EAXをインデックスとして使うことができます。そのため、呼び出す関数は**`_syscalls [eax]`**となります。

さて、ここで呼び出す関数へのポインタができました。しかし、ここでちょっとした問題があります。上記は、EAXで与えられた値に基づいて、必要なサービス関数のポインタを効果的に取得します。しかし、それがどんな関数なのかわかりません。また、その関数に何を渡せばいいのか、どれだけのパラメータを持っているのかもわかりません。

解決策としては、関数呼び出しのためにすべてのレジスタをスタックにプッシュすることが考えられます。サービスはすべてC言語のルーチンなので、C言語の関数が期待する方法でパラメータを渡さなければなりません。

```
// ! バウンドチェック
if (idx >= MAX_SYSCALL)
    // ! サービスを呼び出す
    asm {
        // ! サービスを受ける
        static void fnct = _syscalls[idx];
        push edx
        push ecx
        push ebx
        call fnct
        add esp, 20
        iretd
    }
```

これで完了です。`add esp, 20`は、プッシュした20バイトをスタックからポップします。また、ISRから**`IRETD`**に戻っていることに注意してください。の指示を受けています。

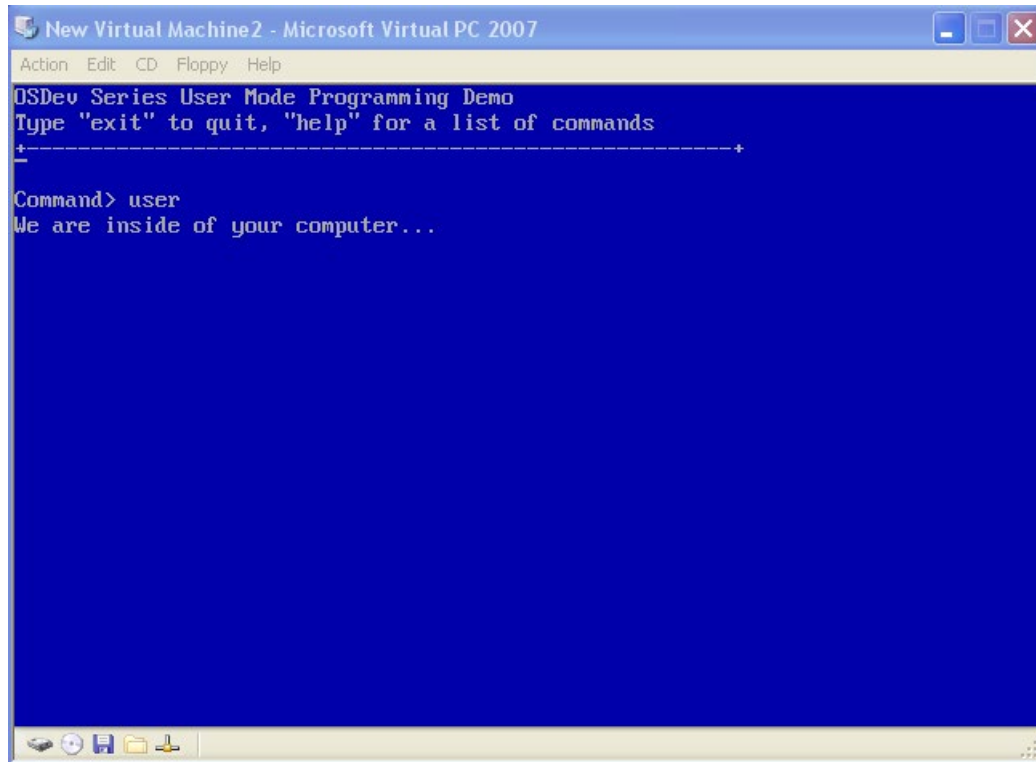
システムソフトウェアやエグゼクティブがそれぞれの割り込みベクタにISRをインストールした後、ソフトウェアはソフトウェア割り込みを発行することでISRを呼び出すことができます。例えば、**`syscall_init`**を呼び出してISRをインストールした場合、次のようにシステムサービスを呼び出すことができます。

デザインに関する注意事項

```
xor eax, eax ; 関数 0, DebugPrintf lea
```

ほとんどのOSでは、割り込みシステムコールやレジスタの詳細をC言語のインターフェイスに置き換えています。大規模なOSのシステム・サービスを直接呼び出すことは可能ですが、システムがユーザーランド・ソフトウェアに提供するシステム・サービスを中心に、標準的なCインターフェイスを開発することをお勧めします。

大規模なOSでは、ディスプレイにメッセージを表示するシステムサービスを持たないのが普通です。むしろ、ユーザーランドのソフトウェアから呼び出すことができるサービスが含まれており、ユーザーAPIがカーネルモードのサービスやサーバー、デバイスドライバとやりとりできるようになっているのです。このため、大規模なOSでは、数百の関数呼び出しからなるシステムAPIが一般的です。



ユーザーモードへの移行
デモダウンロード

新規・変更ファイル

この章では、シリーズのデモにいくつかのファイルを追加しています。以下

- のファイルが含まれます: `hal/tss.h`
- `hal/tss.cpp`
 - 本章で説明するTSSルーチンを含む `hal/user.cpp`
- ユーザーモード切り替えルーチンを含む `kernel/main.cpp`

の章では、以下のファイルも変更しています。

- カーネル/`mmngr_virtual.cpp`
 - `vmngr_initialize` が更新され、カーネルページをユーザーがアクセスできるようになり
- ました `hal/hal.h/cpp`
 - `set_vect()`に第2パラメーターを追加しました `hal/gdt.h`
- `MAX_DESCRIPTOR`は、追加されたGDTエントリ`hal/gdt.cpp`のために6に再定義されました。
- ユーザーモード記述子のインストールを含むようにアップグレードしました
- した `kernel/main.cpp`
 - 新しい変更点を反映して更新

結論

ユーザーランドへようこそ

これで、ユーザーランドとカーネルランドの切り替えに必要なものがすべて揃いました。これで、ユーザーモードのページをマッピングしたり、ユーザーモードでプログラムをロードしたり、実行したりすることができるようになりました。システムがタスクを管理していないため、OS のカーネルにきれいな形で戻る機能はまだありません。これについては次の章で説明します。

次の機会まで。

マイク

BrokenThorn Entertainment社。現在、DoEとNeptune Operating Systemを開発中です。質問

やコメントはありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ私に教えてください。





オペレーティングシステム開発シリーズ

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

24: プロセスマネジメント

by Mike, 2012, 2013

"デバッグ"がソフトウェアのバグを取り除く作業であるならば
となると、プログラミングとはそれを入れる作業のことを指すのでし
よう。
- エドガー・ダイクストラ

はじめに

この章では、プロセス管理とマルチタスクのトピックについて詳しく説明します。これまでの章では、タスクをサポートする基本的なモノリシック・オペレーティング・システムを構築してきました。これまでのOS開発では、複雑な設計をシンプルにするために、複雑さを犠牲にしましたが、複雑な設計でもある程度の詳細を提供することができました。本章では、この傾向を継続していきます。最後に紹介するデモは、決してOSを開発する唯一の方法ではありません。本章では、以下の項目について説明します。

1. プロセス
2. スレッドとタスク
3. プロセスの内側を見る
4. プロセス管理
5. スケジューリング
6. リンク

プロセス管理

プロセス管理とは、オペレーティングシステムがプロセス、スレッドを管理し、プロセスが情報を共有できるようにし、プロセスリソースを保護し、システムリソースを要求するプロセスに安全な方法で割り当てるプロセスのことです。オペレーティングシステムの開発者にとって、このプロセス管理は非常に困難な作業であり、設計が非常に複雑になる可能性があります。では、それぞれの機能を詳しく見ていきましょう。

プロセスとスレッド

この章の残りの部分では、主にプロセスとスレッドについて説明します。プロセスの作成とは、実行可能なイメージを読み込み、それを実行するための少なくとも1つの実行パス（スレッド）を作成することです。

プロセス間通信

IPC (Inter-Process Communication) は、多くのオペレーティングシステムで採用されている、プロセス間の通信を可能にする技術です。これは通常、メッセージパッシングによって行われます。プロセスは、他のプロセスにメッセージを送信することをオペレーティングシステムに要求し、オペレーティングシステムは、可能であれば他のプロセスにメッセージを送信し、キューに入れることができます。IPCは、ファイル、パイプ、ソケット、メッセージパッシング、シグナル、セマフォ、共有メモリ、メモリマップドファイルなど、さまざまな方法で実装することができます。オペレーティングシステムは、これらのIPCの方法のいずれかまたはすべてを実装することができます。IPCは、ハイブリッドカーネルやモノリシックカーネルの設計でも多用されていますが、マイクロカーネルの設計では間違いなく最も顕著です。

本章では、主にプロセスとスレッドの作成に焦点を当てているため、IPCについては触れません。IPCについてはもう少し後に説明するかもしれませんが、おそらく本章の追加となるでしょう。

プロセス保護

複数のプロセスを同じアドレス空間にロードすると、両方のプロセスがお互いに読み書きできるという基本的な問題が発生します。この問題を解決するには、プロセスをそれぞれの仮想アドレス空間に読み込み、物理アドレス空間の別々の場所にマッピングするのが簡単です。プロセスはより多くのスレッドの作成を要求することができますが、これはプロセスごとに行われるため、すべてのスレッドはプロセスと同じアドレス空間を共有することになります。

また、プロセス保護は、プロセスを必要最小限の制御でマッピングすることでも採用されています。例えば、カーネルランドにいる必要のあるプロセスはカーネルスペースに、カーネルランドにいない必要のないプロセスはユーザースペースに配置します。

この章では、プロセスを作成する際に、これらの両方を利用します。プロセスは、ユーザー空間と独自の仮想アドレス空間にマッピングされます。これは、プロセスがカーネルページにアクセスできないことを意味します（つまり、カーネルのスタックや構造体をゴミ箱に入れることはできません）。また、プロセスが他のプロセスをゴミ箱に入れることもできません。

資源配分

リソースの割り当てとは、システムのリソース（ファイルやデバイスハンドルなど）を要求するプロセスに安全に渡す方法のことです。シリーズOSの初期状態では、今のところ気にする必要のある資源はありません。しかし、必要に応じて、割り当てられたプロセスリソースは、通常、**プロセス制御ブロック**内に格納されます。なぜシステムリソースの割り当てを管理する必要があるのかというと、マルチタスク環境において、2つのプロセスが同時に同じファイルを開いて書き込もうとする場合を考えてみてください。

そこで、以下ではプロセスとスレッドの作成を中心に説明します。まず、プロセスとは何か、何がプロセスを構成するのかを明確に定義することから始めます。

プロセス

プロセスとは、メモリ上にあるプログラムのインスタンス、またはプログラムの一部のことで、プロセスは、映画やビデオの再生、ゲームのプレイ、あるいはこの文章を書いているエディタの実行など、複雑なタスクを実行するために、オペレーティングシステムやエグゼクティブによって実行されます。要するに、プロセスはプログラムであると言えますが、プログラムは複数のプロセスを含むことができます。例えば、文字列を表示する基本的なプログラムは、独自のプログラムファイルに組み込まれているかもしれません。プログラムをロードすると、**OS**やエグゼクティブは他のプログラムファイルをロードすることになります。つまり、プロセスが呼び出して使用する実行コードを含む**共有ライブラリ**を**ダイナミックにロードすることになります**。これらのプログラムファイルはすべて同じプロセスの一部であり、1つのプロセスが複数のプログラムファイルのインスタンス、あるいは複数のインスタンスを持つことができるのはそのためです。

プロセスは、エミュレートされた環境またはハードウェア環境において、**中央処理装置 (CPU)** または複数のCPUまたはCPUコアによって実行されます。**ハイパースレッディング**や**並列パイプライン**をサポートするCPUでは、異なるプロセスの複数の命令を同時に実行することもできます。つまり、プロセスはシーケンシャル (1命令ずつ) に実行されるのではなく、環境やハードウェアの構成に応じて、さまざまな方法で実行される可能性があります。**IA32 CPU**ファミリーのデフォルトでは、これらの機能は無効になっています。つまり、コンピュータの起動時には、**CPU**はすべての命令を一度に実行します (ただし、**命令キャッシュバッファ**にキャッシュする場合もあります)。しかし、オペレーティングシステムやエグゼクティブがプロセスに対してこれらの機能を有効にした場合、プロセスやシステムは**マルチプロセッサに対応できるように設計しなければなりません**。しかし、これは高度なテーマであり、エラーが発生しやすいので、上級編で説明します。いくつかのプロセスは、**スレッド**や**タスク**に分割することができます。次はこれらについて見ていきます。

スレッドとタスク

スレッドは、**プロセス内の単一の実行経路**として定義できます。例えば、最も基本的な例では、メッセージを表示して戻りだけのプログラムがあります。

```
#include <stdio.h>
int main (int argc, char** argv) {
    printf ("Hello, world!"); return
    0;
}
```

この例では、プロセスは1つのスレッドを持っています。スレッドは**main()**で始まり、プロセスが終了するとスレッドも終了します。(ただし、実際にはそうではないかもしれないことに注意してください。**main()**を呼び出すランタイムライブラリにスレッドが含まれている可能性があるからです)。では、マルチスレッドの例を見てみましょう。

```
#include <stdio.h> static
int _notExit = 0;

int thread (void* data) {
    while (_notExit) {
        /* 便利なことをする */
    }
    return 0; /* スレッドが終了する ( ランタイムに戻り、Terminate Thread を呼び出す */)
}

int main (int argc, char** argv) {
    CreateThread (thread);
    printf ("Hello, world!");
    return 0;
}
```

この例では、**CreateThread** がオペレーティングシステムを呼び出し、新しい**実行フロー**として **thread()** を設定します。**CreateThread()**が呼び出された後、**スレッド()**がオペレーティングシステムまたはエグゼクティブによって呼び出され、**スレッド()**と**メイン()**の両方の内部で同時に、どちらかが終了するまで**実行が続けられます**。すべてのプロセスはスレッドですが、スレッドはプロセスではないことに注意してください。プロセスは単一のスレッドを含むことも、多数のスレッドを含むこともできます。プロセス内のスレッドは同じグローバル変数にアクセスして共有することができますが、コンパイラによっては**スレッドローカル変数**もサポートしています。

スレッドをサポートするオペレーティングシステムは、**マルチスレッドに対応している**と言われています。**Windows**、**Linux**、**Mac OS**などがこれにあたる。**タスク**はスレッドと同義です。タスクは、オペレーティングシステムやエグゼクティブが実行するタスクを表します。したがって、**マルチスレッド**に対応しているOSは、事実上**マルチタスク**に対応していることになります。ただし、すべてのマルチタスクOSがマルチスレッディングに対応しているわけではないことに注意が必要です。

プロセスとは一体何なのか？ここでは、「プロセス」を「プログラムのインスタンス」または「プログラムの一部」と定義しましたが、ここでは「プロセス」の内部をさらに詳しく見てみましょう。

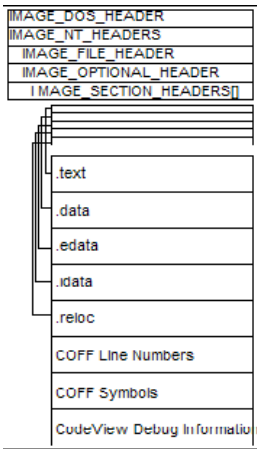
プロセスの内側を見る

プロセスを分解してみると、最も基本的なレベルではコードとデータしかありません。プログラミングの経験がある人なら、これは納得できるだろう。すべてのプログラムは、データに対する動作や操作をCPUに指示する命令に過ぎません。プログラマーが、プログラムの開発を容易にするために、プログラムの「データ」部分と「コード」部分を分離する傾向があるのも理解できるだろう。**.data**と**.text** (プログラムコード用) は、後にプログラムバイナリの中の様々な種類の**セクション**のうちの2つです。セクションは、プログラムの開発に役立つだけでなく、プログラム・バイナリの中にさまざまな種類のものをどのように格納するか基準にもなります。

まず、プログラムセクションと、**プロセスのアドレス空間**にプログラムセクションを配置する方法について説明します。次に**シンボル情報**、**デバッグ情報**、**エクスポートとインポートのテーブル**、そしてそれらがどのように使用できるか。

プログラムセクション

ここでは、**.data** と **.text** という2つのセクション (または**セグメント**) について説明しました。プログラムファイルには、これらのセクションとその他のセクションが含まれています。オペレーティングシステムやエグゼクティブは、プロセスが正しく実行されるように、各セクションをアドレス空間にロードすることができます。また、ロード時に**セクションを再配置することもできます**。これにより、オペレーティングシステムやエグゼクティブは、必要に応じて各セクションの最適な位置を見つけ、それに応じてプロセスを更新することができます。ただし、**プログラムファイルのフォーマットによってサポートする内容が異なるため**、すべてのプログラムファイルのフォーマットがセクションの再配置をサポートしているわけではありません。



Portable Executable (PE) ファイルフォーマットは、Windows OSで使用される主要なプログラムファイルフォーマットです。PEファイル形式は、コード、データ、リソースデータ、シンボリック情報、マニフェストデータなど、さまざまなセクションをサポートしています。各セクションは、リンカーやコンパイラによって書き込まれたバイナリファイル内に格納される。どのように格納されているかを確認するためには、PEファイル形式のフォーマットを見てみる必要があります。

上記は、PE実行ファイルの中身をイメージしたものです。イメージをメモリにロードし、セクションを再配置する必要がない限り、他のタイプのバイナリファイルと同様にファイルの内容を解析することができます。このようにして、シリーズのブートローダーはカーネルイメージをロードすることができるのです。シリーズのカーネルはセクションの再配置を必要としないので、ブートローダーはファイルをロードし、ヘッダの中からエントリポイントを見つけて、それを直接呼び出すことができます。これは簡単にできます。

```
PEヘッダからエントリポイントを取得 */
IMAGE_DOS_HEADER dosHeader = (IMAGE_DOS_HEADER*) imageBase;
IMAGE_NT_HEADERS ntHeaders = dosHeader->e_lfanew;
IMAGE_OPTIONAL_HEADER optHeader = &ntHeaders->OptionalHeader;
void (*EntryPoint) (void) = (void (*EntryPoint) (void) ) optHeader->AddressOfEntryPoint + optHeader->ImageBase;

/* プログラムのエントリーポイント
を呼び出す */ EntryPoint ();
```

同様に、他のヘッダーを解析して、セクション情報、デバッグ情報、シンボリック情報など、ファイルから必要なあらゆる種類の情報を抽出することができます。**カーネルデバッグ**や**ユーザーモードデバッグ**は、通常、デバッグを容易にするためにシンボリック情報やデバッグ情報を使用します。言い換えれば、デバッグ情報を含む(または含まない)PEイメージを構築することができます。デバッグ情報を含むように構築した場合、リモートでデバッグを取り付けてソースレベルのデバッグを行うことができます。

実行ファイルの中に異なるセクションがあると、実行ファイルの解析や記述が容易になります。これらのセクションは、データを格納するための一貫した場所と方法を提供し、ヘッダからそれらを参照することができます。たとえば、PEファイルには、実際のリソース(文字列テーブル、ビットマップイメージ、プログラム情報、カーソルなど)を格納する**.rscs**セクションがあります。リソースを見つけるためには、**OptionalHeader->DirectoryEntries [IMAGE_DIRECTORY_ENTRY_RESOURCE]**のディレクトリエントリを解析するだけで、**.rscs**セクション内のリソースデータを指すリソースツリー構造へのRVA(相対ポインタ)が得られます。重要なのは、実行ファイルのフォーマットは**PE**ファイルの仕様で定義された特定のフォーマットを持っているということです。フォーマットが決まっているので、ファイルから情報を得るための標準的な方法があります。

GCCやCL (マイクロソフトのコンパイラ) のような多くのコンパイラは、プログラマが**セクションを定義**することができます。つまり、プログラマーは自分でセクション名を定義し、必要なものをそのセクションに入れることもできるのです。オペレーティングシステムのカーネルや幹部は、通常、異なる目的のために特別なセクションを定義します。例えば、**Linux**も**Windows**も、特別なセクションを定義しています。**.INIT**セクションには、1回限りの初期化コードとデータが格納されています。初期化が完了すると、OSカーネルはこのセクションを解放し、他の用途に再利用することができます。

共通部分

異なるオブジェクトファイルやアーキテクチャーに共通するセクションの名前とタイプがあります。これらのセクションを認識し、何に使用されているかを知ることは重要です。それらは以下の通りです。

1. .テキスト
2. .データ
3. .bss
4. .rodata

.textセクションは、プログラムコードを含むセクションに与えられる一般的なセクション名です。これは**コードセグメント**とも呼ばれています。システムによっては、このセクションに書き込みができないように、読み取り専用になっている場合がありますが、これはコードの自己修正を妨げることになります(これは通常、推奨されません)。

.dataセクションは、その名の通り、プログラムが使用する**スタティックデータ**や**グローバルデータ**を格納します。常にかき込み可能です。

.bssセクションは、**.data**セクションの一部で、通常、ゼロに初期化された静的に割り当てられたデータに使用されます。このセクションは**.bss**セクションは、OSローダーによって常にクリアされるため、その中のデータはすべてゼロになります。ウィキペディアによると、「**.bss**」という名前は、当初、**United Aircraft Symbolic Assembly Program**の**Block Started by Symbol**を意味していました。**.bss**セクションにはすべてのヌル変数が含まれているため、オブジェクトファイルの中でスペースを取りません。

.rodataセクションには、読み取り専用の静的に割り当てられたデータが格納されています。**Linux**や**Unix**の環境でよく見られます。

一時データ用のセクションがないことに注意してください。一時変数はスタックに格納されるので、プログラムファイルに格納する必要がないことを思い出してください。

Microsoft Visual Cのカスタムセクション

Microsoftのコンパイラには、プログラマが特定のセクションやカスタムセクションにデータやコードを配置する際に使用できるプラグマディレクティブがいくつか用意されています。これらは

1. `alloc_text`
2. `code_seg`
3. `const_seg`
4. `data_seg`
5. `bss_seg`
6. `init_seg`
7. セクション

プログラムローダは、プログラムが持つ特別なセクションを気にする必要はなく、メモリにロードすることだけを考えればよい。プログラム（つまりプログラマー）が責任を負うのです。

ここでは、特別なセクションに関数を追加するために`alloc_text`を使用する例を示します。

```
error_t DECL mmInitialize (SystemBoot* mb) { return
    SUCCESS;
}
#pragma alloc_text (".init", mmInitialize);
```

上の例では、`mmInitialize`が`.init`セクションに追加されます。これは、一部のオペレーティング・システム・カーネルやエグゼクティブが使用している便利な戦術です。例えば、オペレーティング・システムのカーネルやエグゼクティブは、初期化コードやデータを、特別な`.init`セクションです。初期化が完了すると、OSはそのセクションを解放してメモリの一部を取り戻すことができます。

象徴的な情報

記号情報とは、プログラマーがアドレスの名前として与える**記号**のことです。例えば、次のような関数を呼び出す場合**`printf()`**では、コンパイラやリンカはどのようにして何をすべきか知っているのでしょうか？もう少し詳しく見てみましょう。

"**`printf`**"は、ライブラリで定義されている関数のシンボルです。**`printf()`**を呼び出すと、コンパイラがビルド時に管理している**シンボルテーブル**にシンボル「**`printf`**」が追加されます。**関数の名前がシンボルである**ことに注目してください。同様に、**スタティック変数やグローバル変数もシンボルです**。アドレスにつける名前（アセンブリ言語の**ラベル**のようなもの）はすべてシンボルであるということが出来ます。このように、シンボルは2つのものを持っています。名前とアドレスです。

`printf`のコードを含むライブラリをリンクせずにビルドした場合、コンパイラはコード全体を機械語に翻訳することができないため、最終的な実行ファイルを出力することができません。つまり、アセンブラのように、以下のようなコードでは、関数が何であるかわからないと何もできないのです。

```
コール _printf
```

アセンブラは、シンボル `_printf` のアドレスを知らないと、この命令を完全にアセンブルできません。シンボルについて何も知らないと、アセンブラはアドレスを知ることができません。この問題を解決するために、シンボルが**外部**にあることを宣言し、アセンブラやコンパイラは実行ファイルではなく**オブジェクトファイル**を出力します。命令を機械語に部分的に変換しますが、次のような形になります。

```
0xe8 _printf
```

これにより、別のプログラム（**リンカー**）を使ってシンボルを解決することができます。リンカは、異なるオブジェクトファイルやライブラリのエクスポートシンボルテーブルを見て、シンボル「`_printf`」を探します。シンボルが見つければ、リンカーはファンクションコードのアドレスを取得し、そのアドレスでマシンコードを更新して、最終的に実行可能なプログラムを適切にリンクして出力します。シンボルが見つからない場合は**未解決**となり、リンカーは有名な「未解決の外部シンボル」というエラーを出します。

実行イメージの記号情報は、デバッガが人間が読める情報（関数や変数）名を表示するために使用できますが、その代償としてプログラムのファイルサイズが大きくなります。

シンボルに関する追加情報をシンボル名自体に「格納」する方法は様々です。これは、**ビルド環境や呼び出し規則によって異なります**。標準的なC言語の呼び出し規則は**CDECL**で、すべての名前の前にアンダースコアを付けるだけです。例えば、"`printf()`"を呼び出した場合、その**CDECLシンボル名**は"`_printf`"となります。**C++のシンボリック名**は、コンパイラによって異なり、名前以外にも多くの情報（戻り値のデータ型やオペランドの型、名前空間、クラス、テンプレート名など）を保持しています。例えば、CL（マイクロソフトのコンパイラ）の「`void h(void)`」という関数は、シンボリック名「`?h@YXXZ`」に変換されます。ここでは、名前のマングリング形式の詳細については触れない。

ここで面白いことに気がつきました。C言語のシンボリック名には、戻り値のデータ型やオペランドに関する情報は何も格納されていませんが、名前のマングリングによるC++のシンボリック名には格納されています。これは理解できますが、言語間の多くの違いの1つです。Cコンパイラでは、異なるオペランド型やオペランド数の関数をエラーなしで呼び出すことができます（検出されると警告が出るかもしれませんが）、C++コンパイラではエラーが出ます。

テーブルのエクスポートとインポート

プログラムライブラリやオブジェクトファイル内のシンボルは、他のライブラリやプログラムで使用するために**エクスポート**することができます。**エクスポートされたシンボル**は、コンパイラやリンカーに、それぞれのシンボルを**エクスポートテーブル**に追加するように指示するだけです。プログラム・ファイルや共有ライブラリ（Windows DLL）は、他のプログラムやデバッガが使用するためにシンボルをエクスポートすることができます。同様に、プログラムファイルは使用するためにシンボルの**インポート**を要求することができます。ここで、上記の`printf()`の例を完成させることができます。

Microsoft C Runtime Libraryは、プログラムファイルとともに読み込まれる共有ライブラリです。オペレーティングシステムやエグゼクティブは、プログラムの**インポートテーブル**を見ることができ、プログラムファイルの動作に必要なDLLを知ることができます。デフォルトでは、CL(マイクロソフト社のコンパイラ)は、インポートテーブルを含むMicrosoft C Runtime Libraryのインポートスタティックライブラリとリンクするので、シンボルが追加され、それぞれのDLLがテーブルに含まれます。オペレーティングシステムまたはエグゼクティブは、プログラムが必要とするすべての共有ライブラリファイルはまだメモリにロードしていない場合は、プログラムファイルの**インポートアドレステーブル(IAT)**をこれらの他のDLLのアドレスで更新する必要があります。ロードされたMicrosoft C Runtime Library DLLには、`_printf`のコードが含まれているだけでなく、シンボル`_printf`もエクスポートされているため、OSはランタイム中にこれらをリンクします。(この点については後ほど詳しく説明します)。

したがって、プログラムファイルから「`printf()`」を呼び出すと、ジャンプテーブルが呼び出され、更新されたIATアドレスが呼び出され、CランタイムライブラリDLLの「`_printf`」関数が呼び出されます。

これまでに、プロセス、スレッド、タスクについて説明し、プログラムファイルとは何か、どのように動作するのかを見てきました。本章の目標は、複数のプロセスやタスクをロード、実行、管理できるようになることです。次はそれについて見ていきましょう。

プロセスマネジメント

プロセスマネジメントとは、ソフトウェアシステムにおける**プロセス**の管理です。先にプロセスを「メモリ上のプログラムまたはプログラムの一部」と定義しました。つまり、プロセスを管理するということは、メモリ上のプログラムの複数のインスタンスを協調して管理することを意味します。これは最近のOSでは一般的に要求されていることで、カーネルやエグゼクティブに実装されています。プロセス管理をサポートするオペレーティングシステムは、**マルチタスクオペレーティングシステム**と考えられます。

表現

OS設計者は、プロセスを管理するために、OSの設計基準や必要なシステムリソースを考慮して、プロセスをどのように表現するのが最適切かを判断する必要があります。プロセスは次のように構成されています。

- ◆ メモリ上の実行ファイルのイメージ (マシンコードとデータ)。
- ◆ プロセスが使用しているメモリとその仮想アドレス空間
- ◆ プロセスを表現するための記述子
- ◆ プロセスの状態情報 (レジスタ、スタック、属性など)

オペレーティングシステムは、プロセスを管理し、システムリソースを要求したプロセスに公平に割り当てることが求められます。それぞれを詳しく見てみましょう。

メモリ上の実行ファイルのイメージ

実行可能なプログラムは、プログラムの読み込みや管理を容易にするために、ディスク上にファイルとして格納されている。**プログラムをロードするには、オペレーティング・システム・ローダーがファイルをメモリにロードする**。ローダーは、ファイルの種類 (オペレーティングシステムが扱うことのできる実行可能ファイルでなければならない) を理解し、場合によってはこれらのファイルの種類の機能 (リソースやデバッグ情報など) をサポートすることもできなければならない。

メモリ上の実行ファイルのイメージは、イメージのマシンコードとデータの現在の表現であり、ある時点でメモリ上にどのように表示されているかを示しています。ここで「イメージ」という言葉を使っているのは、メモリ上のものの「スナップショット」を表している。例えば、カメラで大きなバイトの配列を見て写真を撮るようなものです。バイトの配列は、マシンコードであったり、データであったり、そのどちらでもないものである。プログラムの命令だけが知っています。

プログラムイメージの中には、他のプログラムやOSにとって有用なデータが含まれていることがあります。例えば、プログラムファイルにはデバッグ情報が含まれていることがあります。例えば、プログラムファイルにはデバッグ情報が含まれており、プログラムにデバッガを取り付けて、その情報を利用することができます。

要するに、OSがファイルを実行するためには、ディスクからメモリのどこかにファイルをロードできる必要があります。これは、ファイルをそのままメモリにロードするようなものです。オペレーティングシステムや他のプログラムは、その後、プログラムファイルから必要な有用なデータを得ることができます。

プロセスが使用しているメモリとその仮想アドレス空間

プロセスは通常、オペレーティングシステムと同様に、動的にメモリを割り当て、スタックスペースを使用するコールを持っています。**オペレーティングシステムは、プロセスが使用するプロセススタックとヒープメモリのためのスペースを割り当ててする必要があります**。例えば、オペレーティング・システムは通常、すべてのプロセスにデフォルトのスタック・サイズを割り当てます。しかし、**プロセスの実行ファイルは、プロセスが必要とする場合、より大きなスタックスペースを割り当てようオペレーティングシステムに指示することができます**。

プロセスのヒープは違います。スタックはプロセスを実行する前にオペレーティングシステムによって割り当てられますが、ヒープはそうではありません。その代わり、各プロセスはユーザーモードで独自のヒープアロケータを持っています。これは C ランタイム・ライブラリ (CRT) に実装されており、`malloc`、`free`、`realloc`、`brk`、`sbrk` の各関数の使い慣れたインターフェイスを使用しています。CRTとリンクしているプログラムは、これらの関数を呼び出してメモリを割り当てることができます。しかし、CRTとリンクされていないプログラムは、独自のヒープアロケータを実装するか、ヒープアロケータを実装している他のライブラリとリンクする必要があります。

CRT ランタイムは、ユーザーモードのヒープアロケータ (通常はフリーリスト) を実装しています。C関数の`malloc`は、System APIを使用してOSを呼び出す`brk`を呼び出すことがあります。C関数の`brk`は、必要に応じてヒープを拡張するために、より多くの仮想メモリを割り当てるために、OSを呼び出します。

簡単に言うと、ユーザーモードのヒープは次のように動作します。プログラムが`malloc`を呼び出し、その`malloc`が`brk`を呼び出し、`brk`がSystem APIを使ってOSを呼び出し、ヒープ用の仮想メモリを確保します。`malloc`および`free`関数群は、独自のユーザーモードヒープアロケータを実装しています。これらの関数は、仮想アドレス空間からメモリを割り当てたり解放したりするためにOSを呼び出すだけです。

プリエンプティブ・マルチタスクでは、すべてのプロセスが独自の仮想アドレス空間を持ちます。これは、すべてのプロセスが自分のページディレクトリと関連するページテーブルを持たなければならないことを意味します。プロセス固有の情報を管理するために、**PCB** (

Process Control Block) を使用します。次にそれを見てください。

プロセスを表現するための記述子

プロセスコントロールブロック (PCB) は、プロセスやタスクに関する情報を格納するために使用されるデータ構造です。PCBには、割込み記述子ポインタ、ページディレクトリベースレジスタ (PDBR) などの情報が含まれます。保護レベル、実行時間、プロセスの状態、プロセスフラグ、VM86フラグ、優先度、およびプロセスID (PID)。PCBはより多くの情報を含んでいるかもしれませんが、実際にはOSに依存しています。

オペレーティングシステムは、プロセスを管理するためにPCBのリンクされたリストを使用することがあります。新しいプロセスを作成するとき、オペレーティングシステムは、新しい仮想アドレス空間を割り当て、イメージをロードしてマッピングし、新しいPCB構造をリストに添付する必要があります。スケジューラは、実行するプロセスを決定し、現在の状態を保存するためにPCBリストを使用します。

プロセス状態情報

プロセス状態の情報には、ある時点でのプロセスの全レジスタ状態、インメモリ状態、入出力要求状態などが含まれます。プロセスの状態は、タスクを切り替える際にPCBに保存されます。これは、マルチタスクOSの心臓部であるスケジューラによって行われます。さらに、プロセスの現在の実行状態は、オペレーティング・システムによるプロセスの実行を制御するために使用されます。

最も単純なケースでは、状態はRUNNINGかNOT RUNNINGのどちらかです。このモデルでは、作成されたばかりのプロセスは、NOT RUNNINGキューに格納され、実行中のときだけRUNNINGと表示されます。NOT RUNNINGとなったプロセスは、RUNNINGプロセスが終了するか、スケジューラ内のプロセスディスパッチャに割り込まれるまで、メモリ上に存在しますが、待機状態となります。

3つの状態からなるプロセス管理モデルでは、プロセスは「実行中」、「準備中」、「ブロック中」のいずれかになります。RUNNINGプロセスが、プロセスの待ち時間を必要とするものへのアクセスを要求した場合(I/O要求など)、オペレーティングシステムはプロセスをRUNNINGからBLOCKEDに変更することができます。要求が実行できるようになると、プロセスはRUNNINGまたはREADYのいずれかの状態に移行することができます。READY状態のプロセスは、プロセス・ディスパッチャによる実行の準備ができていることを意味するだけです。RUNNING状態のプロセスは、すでに実行されています。

最後のモデルは、「5つの状態のプロセス管理モデル」です。このモデルでは、5つの状態を利用しています。SUSPEND BLOCKED」、「BLOCKED」、「SUSPEND READY」、「READY」、「SUSPENDED」です。

スケジューリング

スケジューラーとは、OSのカーネルやエグゼクティブのコンポーネントで、タスクの切り替えやCPU使用率の割り当てを行うものです。オペレーティングシステムでは、次に実行するタスクを決定するためにスケジューリングアルゴリズムを採用しています。一般的なスケジューリングアルゴリズムとしては、先入れ先出し、最短残り時間、固定優先プリエンプティブ、ラウンドロビン、マルチレベルキューなどがありますが、これらに限定されるものではありません。WindowsとLinuxの両方で使用されている最も一般的なアルゴリズムは、**マルチレベルのフィードバックキュー**です。

基本的なプロセス管理のサポート

これで、基本的なプロセス管理サポートを実装することができます。目標はシンプルさなので、vm86タスクのサポートやI/Oリソースの割り当てなど、高度なマルチレベルのフィードバックシステムは実装せず、シンプルだが効率的なスケジューラに焦点を当てます。

そのためには、サポートを追加するために何をしなければならないのか、その目的を考えてみましょう。

1. 実行可能なイメージをメモリにロードし、パーシングする。
2. プロセス用のPCBのリストを管理します。
3. ユーザーモードのタスクをサポートします。
4. 複数の仮想アドレス空間をサポート
5. 各プロセスにスタックスペースを割り当てます。デフォルトのサイズは4kです。
6. スケジューリングアルゴリズムの選択とタスクスイッチの実装

これはマルチタスクに対応するための目標です。プロセスはユーザーモードのプロセスになります。しかし、マルチタスクは**プロセス管理**と**スケジューリング**の両方に依存します。このため、本章では、マルチタスクをサポートするフレームワークの構築に焦点を当てますが、1つのスレッドを持つ1つのプロセスのみを許可します。これは、次の章でスケジューラーを実装する際に拡張されます。

プロセスコントロールブロック

私たちのシステムのPCB構造はシンプルなものになります。

```
#define PROCESS_STATE_SLEEP 0
#define PROCESS_STATE_ACTIVE 1

#define MAX_THREAD 5

typedef struct _process {
    int id;
    int priority;
    pdirectory* page;
    Directory;
    int state;
    /* typedef struct _process* next; */
    /* thread* thread List; */
    thread threads[MAX_THREAD];
    /*
    note: 以下のような情報を追加することができます。
    - LDT記述子 [ 使用されている場合 ]
    - 使用されているプロセッサ数
    - ユーザータイムとカーネルタイム
    - 実行オブジェクトなど
    */
} process;
```

この構造体にさらに追加することもできますが、実際に必要なのは上記のものだけです。この構造体には、プロセスID (PID)、優先度、仮想アドレス空間が格納されていることに注目してください。コメントされている2つのエンタリは、完全性を保つためにだけに用意されています。一般的なOSでは、これらはプロセスとスレッドのリンクリストになるはずですが、これにはカーネルのヒープアロケータが必要ですが、私たちは書いていません。簡単にするために、プロセス内の5つのスレッドオブジェクトを配列として保存します。

最後に必要なのは、スレッドを処理する方法です。すべてのプロセスは、エントリーポイントで実行を開始する最大1つのスレッドを持っています。

```
typedef struct _thread {
    process* parent;
    void*initialStack;
    void*stackLimit;
    void*kernelSt
ack; uint32_t priority;
    intstate;
    trapFrame フレーム。
}thread;
```

スレッド構造体は、プロセス内のスレッドに関する一般的な情報を格納します。この構造体には、親プロセスへのポインタ、スレッドスタック、優先度、状態（実行中かどうか）、およびトラップフレームに関する情報が格納されています。トラップフレームは、実行中のスレッドの現在のレジスタ状態を格納します。

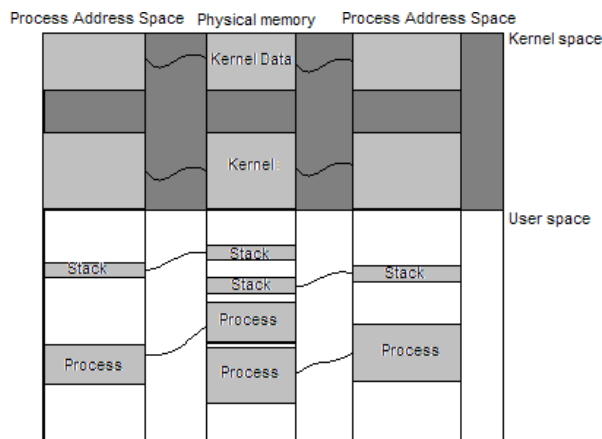
```
typedef struct _trapFrame {
    uint32_t esp;
    uint32_t ebp;
    uint32_t eip;
    uint32_t edi;
    uint32_t esi;
    uint32_t eax;
    uint32_t ebx;
    uint32_t ecx;
    uint32_t edx;
    uint32_t flags;
    /*
    注：これにさらにレジスターを追加することができます。完全なトラップフレームのためには、追加する必要があります。
    - デバッグレジスター
    - セグメントレジスター
    - エラー状態 [ もしあれば ]
    - V 86 モードセグメントレジスタ [ 使用されている場合 ]
    */
}trapFrame;
```

本章では、マルチタスクをまだ実装していないので、トラップフレーム構造はあまり使用しません。しかし、次の章では、各スレッドの現在の状態を保存するスケジューラを開発するので、トラップ・フレーム構造をより多く使用することになります。

仮想アドレス空間

複雑なのは、複数の仮想アドレス空間をサポートしたい場合です。各プロセスのアドレス空間は、4GBのアドレス空間全体で構成されており、カーネルコードとデータは2GBに配置されています。プロセスを切り替える際には、アドレス空間を切り替える必要がありますが、アドレス空間の下位2GB（「ユーザーランド」）のみを切り替えることができます。言い換えれば、ユーザーモードのプロセスが実行されているとします。タスクを切り替えるためには、カーネル内のスケジューラを何らかの方法で呼び出す必要があります。しかし、これはカーネルコードが同じアドレス空間になければならないことを意味しています。もしそうでなければ、即座にクラッシュしてしまいます。

この問題を解決するためには、カーネルコードをすべてのプロセスのアドレス空間にマッピングする必要があります。どうやってそんなことができるのかと思われるかもしれませんが、しかし、複数の仮想アドレスがメモリ上の同じ物理フレームを参照できることを考えると、より明確になるかもしれません。言い換えれば、カーネルのスタックとコードを両方のアドレス空間にマッピングすることができます。次の図をご覧ください。



上の画像では、2つの仮想アドレス空間と物理アドレス空間が表示されています。プロセススタックの位置とコードの位置が、物理メモリの異なる位置を共有していることに注目してください。つまり、仮想メモリマネージャを使って、同じ基本的な仮想アドレスの位置を、異なる物理アドレスフレームにマッピングしているのです。ここで、カーネルについて考えてみましょう。カーネルは、単一のアドレス空間を持つ環境で起動します。カーネルは、初期化プロセスにおいて、自分自身を自分のアドレス空間にマッピングします。問題を防ぐためには、カーネル空間を他のプロセスのアドレス空間にもマッピングできるようにしておく必要があります。カーネルはすでに自分のアドレス空間にマッピングされており、物理メモリのある場所に配置されています。つまり、カーネルは他のプロセスのアドレス空間にも自分自身を再マッピングすることができます。

シリーズのカーネルでは、カーネル自身が1MBの物理的な領域から3GBの仮想的な領域にマッピングされます。カーネルは、自分自身を各プロセスのアドレス空間にマッピングするために、すべてのプロセスの3GBの領域を1MBの物理的なものにマッピングしなければなりません。カーネルとカーネルスタックは、すべてのプロセスのアドレス空間の同じ場所にマッピングされなければなりません。

オペレーティングシステムでは、カーネル全体ではなく、カーネルの一部をプロセスアドレス空間にマッピングすることもできます。これは大規模なシステムではよくあることです。

アドレス空間の管理

異なるアドレス空間の仮想ページをマッピングすることができるようにする必要があります。具体的には、次のようなことができるようにする必要があります。

1. 任意のページディレクトリからページテーブルを作成
2. 任意のページディレクトリから任意の物理アドレスを仮想アドレスにマッピング
3. 任意のページディレクトリから任意の仮想マッピングの物理アドレスを取得する
4. 新しいアドレス空間の創造

シリーズの仮想メモリマネージャは、現在この機能をサポートしていません。しかし、すぐに実装することができますので、今からでも実装してみましょう。

ページテーブルの作成

ページテーブルを作成するために必要なことは、空きフレーム（ページテーブルは1024個のPTEで構成され、4096バイトが1ページのサイズであることを思い出してください）を割り当て、それをページディレクトリのPDEのフレームに追加することです。**Virt >> 22**では、仮想アドレスからディレクトリのインデックスを取得するだけです。**pagedir [directory_index]**のPDEが0であれば、このページテーブルは存在しないことがわかるので、物理メモリマネージャを使って割り当てます。存在していれば、割り当ての必要はありません。最後に、ページテーブルをクリアして、現在ビットを0（存在しない）にします。

```
int vmmngr_createPageTable (pdirectory* dir, uint32_t virt, uint32_t flags) {
    pd_entry* pagedir = dir->m_entries;
    if (pagedir [virt >> 22] == 0) {
        void* block = pmmngr_alloc_block();
        if (!block)
            return 0; /* デバッグを呼ぶべき */
        pagedir [virt >> 22] = ((uint32_t) block) | flags;
        memset ((uint32_t*) pagedir[virt >> 22], 0, 4096);

        ページテーブルをディレクトリにマッピングする。
        vmmngr_mapPhysicalAddress (dir, (uint32_t) block, (uint32_t) block, flags);
    }
    return 1; /* 成功 */
}
```

この機能により、任意のページディレクトリのページテーブルを作成することができます。

物理アドレスのマッピング

次に不足している機能は、異なるページディレクトリの物理アドレスと仮想アドレスをマッピングできるようにすることです。これは簡単です。

```
void mapPhysicalAddress (pdirectory* dir, uint32_t virt, uint32_t phys, uint32_t flags) {
    pd_entry* pagedir = dir->m_entries;
    if (pagedir [virt >> 22] == 0)
        createPageTable (dir, virt, flags);
    ((uint32_t*) (pagedir[virt >> 22] & ~0xffff))[virt << 10 >> 10 >> 12] = phys | flags;
}
```

この関数は、以前に仮想メモリマネージャに実装した基本的な機能を踏襲しています。有効なページテーブルがあるかどうかをテストし、存在しないとマークされていればページテーブルを作成します。最後の行ではマッピングを行います。

この機能により、任意の仮想アドレス空間の物理アドレスと仮想アドレスをマッピングすることができます。

物理的アドレスの取得

次に不足している機能は、先ほどとは逆に、特定のアドレス空間から任意の仮想アドレスの物理アドレスを取得することです。

```
void* getPhysicalAddress (pdirectory* dir, uint32_t virt) {
    pd_entry* pagedir = dir->m_entries;
    if (pagedir [virt >> 22] == 0)
        return 0;
    return (void*) ((uint32_t*) (pagedir[virt >> 22] & ~0xffff))[virt << 10 >> 10 >> 12];
}
```

この関数は、その仮想アドレスに有効なページテーブルがあるかどうか（存在するかどうか）をテストし、PDEとPTEを参照解除して物理フレームを返します。

新しいアドレス空間の作成

各プロセスは、それぞれの仮想アドレス空間で動作します。そのためには、複数のアドレス空間を作れるようにする必要があります。

```
pdirectory* createAddressSpace () {
    pdirectory* dir = 0;

    ページディレクトリの確保 */
    dir = (pdirectory*) pmmngr_alloc_block ()です。
    if (!dir)
```

```

        0を返す。

    メモリをクリアする（すべてのページテーブルを存在しないものとしてマークす
    る） */ memset (dir, 0, sizeof (pdirectory) )。
    dirを返す。
}

```

この関数のシンプルさに注目してください。この関数が行うことは、ブロックを割り当ててクリアするだけです。これは、ページディレクトリがアドレス空間を表し、1つのページディレクトリが4096バイトであることから理解できます。クリアすることで、すべてのPDEで現在のビットを0にしていることになります。

プロセスを実行する際には、作成したばかりの新しいアドレス空間に切り替えることができればなりません。つまり、この新しいページディレクトリをPDBRにロードする必要があるのです。この機能はPMMですでに実装されています。しかし、単に空のページディレクトリをPDBRにロードしただけでは、直後に必ずトリプルフォールトが発生します。この原因は簡単で、カーネルコードやスタックがこの新しいアドレス空間にマッピングされていないからです。

これを解決するには、カーネル空間をマッピングする必要があります。興味深いことに、次のようにすれば、現在のページディレクトリ（PDBRに格納されている）をこの新しいアドレス空間にコピーすることができます。

```
memcpy (dst->m_entries, cur->m_entries, sizeof (uint32_t)*1024);
```

必要なのはこれだけです。ページテーブルはすでに元のページディレクトリにマッピングされているので、ページテーブルのコピーを気にする必要はありません。以上の作業により、効果的にアドレス空間のコピーが作成され、カーネルのページテーブルが両方のアドレス空間にマッピングされ、これが必要な状態になります。

スレッドの作成

スレッドを作成するためには、まず `createThread` 関数に必要なものと、スレッド作成が実際に意味するものを決定する必要があります。スレッドとは単一の実行経路であると定義したことを思い出してください。これを踏まえて、必要なのはエンタリーポイント関数です。この関数が完了すると、オペレーティングシステムを呼び出してスレッドを終了させます。これは通常、スレッドの作成と終了を単純化するために、システムAPI（Win32 APIなど）によって行われます。

スレッドを作成するために必要なのは、スレッド構造体を割り当ててプロセスに追加することだけです。当初、この機能はこのデモのために実装する予定でしたが、25章のマルチスレッドに回すことになりました。

プロセス作成

プロセスを作成するためには、OSの専用ローダーコンポーネントがすでに存在している必要があります。ローダーコンポーネントは、実行ファイルの読み込みと解析、BSSセクションのクリア、セクションのアラインメント、その他、ダイナミックリンクライブラリのダイナミックローディングなど、必要と思われることを行います。ローダーの作成は、特にPEのような複雑なファイルフォーマットの場合、複雑な作業となります。そのため、この章の目的であるプロセス管理に集中できるよう、このシリーズではよりシンプルなソリューションを選ぶことにしました。

プロセスをクレートするためには、プロセスとは何か、スレッドのそれとどう違うのかを明確に理解する必要があります。具体的に言うと

1. スレッドにはそれぞれ専用のスタックがあり、プロセス自体にはスタックはありません。
2. 各プロセスには、少なくとも1つのスレッドが必要です。これは、プロセスのエンタリーポイントから始まります。
3. 各プロセスは、独自の仮想アドレス空間を持つ必要があります。プロセス内のスレッドは、そのプロセスと同じアドレス空間を共有します。
4. 各プロセスは、実行可能なイメージとしてディスクからロードする必要があります。これには通常、別のローダーコンポーネントを使用します。

シリーズには専用のイメージローダーコンポーネントがないため、簡単にするために、`createProcess` という1つの関数でこれらのステップをすべて実行します。この関数は次のようなステップを踏みます。

1. 実行ファイルの読み込み
2. プロセスのアドレス空間を作成します。
3. PCB（Process Control Block）を作成します。
4. メインスレッドを作成します。
5. イメージをプロセスの仮想アドレス空間にマッピングします。

これは1つの機能としてはかなりの量です。後で実行ファイルのロードを専用のローダに移すことができます。簡単にするために、このルーチンはブートルoaderと同じ基準を想定しています。つまり、ロードされるイメージは、セクタアラインされたセクション（/align:512フラグを使用）を持っていないければならず、Microsoft Windowsのランタイムライブラリとリンクされていないのです。将来的にはこの機能をデモに追加するかもしれませんが、これはローディングコードを複雑にしますし、先に述べたように、通常はローダーコンポーネントが処理します。

プロセス・アドレス・スペース構造

現在、シリーズOSのカーネルは、アイデンティティマップドメモリーの1MB以下に多くのカーネル構造がロードされています。これには、カーネルスタック、初期ページディレクトリテーブル、ページテーブルなどが含まれます。また、DMAC（Direct Memory Access Controller）のメモリー領域もこの領域に配置されている可能性があります。また、カーネルが他のメモリー領域（ディスプレイメモリーなど）も利用していることも考慮しなければなりません、それらもこのアイデンティティマップド領域にあります。

これはすべて、単純化のために行われたものです。一般的なカーネルは、PIC（Position Independent Code）を使って、カーネルメモリー

内のカーネルスタックと初期ページテーブルを初期化します。PICは、上位のカーネルが他の場所でロードされたときに起動するためのものでもあります。

物理的なベースアドレス。これを正しく行うのは難しいので、今回のシリーズでは採用しませんでした。しかし、その結果、1MB以下のカーネル構造がいくつかできてしまい、混乱してしまいました。

いろいろなものを移動させるよりも、0~4MBをカーネルモード専用に確保するのがベストな選択だと思いました。これにより、カーネルは何も手を加えずに機能し続けることができ、ディスプレイ出力やその他の基本的なことのためにメモリをリマップすることも問題ありません。つまり、アドレス空間は次のようになります。

```
0x00000000-0x00400000 - カーネルの予約
0x00400000-0x80000000 - ユーザーランド
0x80000000-0xffffffff - カーネルの予約
```

つまり、すべてのプロセスは、4MBと2GBの領域内にイメージベースを持たなければなりません。ここでは、すべてのユーザーモードプロセスのベースアドレスとして4MBを使用します。最初の4MBはカーネルモードのページとして同一にマッピングされたままで（これはすでに行われています）、カーネル自体は3GBにマッピングされたままです。つまり、プロセスのすべてのページは、4MBと2GBの間のユーザーモードページとしてマッピングされます。

プロセスの作成

それでは早速、関数を見てみましょう。これは、一般的にローダーで行われるソフトウェアを含んでいるため、かなり長いルーチンになっています。今回のデモでは、新しいアドレス空間を作るのではなく、現在のアドレス空間にイメージをロードしてマッピングしていますが、両方とも実装されています。これは、このソフトウェアが一度に1つのプロセスしか実行できないように設計されているためです。第25章のマルチタスクでは、この点を変更します。

vmnmgr_createAddressSpaceと**mapKernelSpace**という2つの新しい関数に注目してください。vmnmgr_createAddressSpaceとmapKernelSpaceという2つの新しい関数に注目してください。つまり、カーネルメモリ、スタック、ページディレクトリ、ディスプレイメモリを新しいアドレス空間にマッピングします。これらの機能は使用されていませんが、次の章で使用されます。

validateImage関数は、画像のヘッダーを解析し、サポートされているかどうかを確認するだけです。最後に、初期のスレッド構造を作成しますが、マルチスレッドには対応していません。プロセスごとに1つのスレッドを想定しており、1つのプロセスの1つのスレッドしか実行できません。

```
int createProcess (char* appname) {
    IMAGE_DOS_HEADER* dosHeader;
    IMAGE_NT_HEADERS* ntHeaders;
    FILE file;
    pdirectory* addressSpace;
    プロセス* proc;
    thread* mainThread;

    Thread: unsigned char
            *memory;
            uint32_t i;
    unsigned char buf[512];

    ファイルを開く*1
    file = volOpenFile (appname);
    if (file.flags == FS_INVALID)
        0を返す。
    if ((file.flags & FS_DIRECTORY) == FS_DIRECTORY) return
        0;

    /* 512バイトをバッファに読み込む */
    volReadFile (&file, buf, 512);
    if (! validateImage (buf)) {
        volCloseFile (&file);
        return 0;
    }
    dosHeader = (IMAGE_DOS_HEADER*)buf;
    ntHeaders = (IMAGE_NT_HEADERS*) (dosHeader->e_lfanew + (uint32_t)buf)。

    プロセスの仮想アドレス空間の取得 */
    //addressSpace = vmnmgr_createAddressSpace ();
    addressSpace = vmnmgr_get_directory ()です。
    if (!addressSpace) {
        volCloseFile (&file);
        return 0;
    }
    /*
     * カーネル空間をプロセスのアドレス空間にマッピングし
     * ます。新しいアドレス空間を作る場合のみ必要
     */
    //mapKernelSpace (アドレススペース) です。

    PCBを作成する。
    proc = getCurrentProcess();
    proc->
    >id = 1;
    proc->pageDirectory = addressSpace;
    proc->priority =
    1;
    proc->state =
    PROCESS_STATE_ACTIVE;
    >threadCount = 1;

    スレッド記述子の作成 */
    mainThread = &proc->
    >threads[0]; mainThread->kernelStack = 0;
    mainThr
    ead->parent = proc;
    mainThr
    ead->priority = 1;
    mainThread->state =
    PROCESS_STATE_ACTIVE; mainThread->initialStack = 0;
    mainThread->stackLimit = (void*) ((uint32_t) mainThread->
    >initialStack + 4096); mainThread->imageBase = ntHeaders->OptionalHeader. ImageBase;
    mainThread->imageSize = ntHeaders->
    >OptionalHeader. SizeOfImage; memset (&mainThread->frame, 0, sizeof
    (trapFrame));
    mainThread->frame.eip = ntHeaders->OptionalHeader. AddressOfEntryPoint
    + ntHeaders->OptionalHeader. ImageBase;
```

```
mainThread->frame.flags = 0x200;
```

```
/* 上で読み込んだ512ブロックと残りの4kブロックをコピーします。
```



```

memory = (unsigned char*)pmmngr_alloc_block();
memset (memory, 0, 4096);
memcpy (memory, buf, 512)。

画像のメモリへのロード */
for (i=1; i <= mainThread->imageSize/512; i++) { if
    (file.eof == 1)
        ブレークします。
    volReadFile (&file, memory+512*i, 512);
}

/* ページをアドレス空間にマッピングする */
vmmngr_mapPhysicalAddress (proc->pageDirectory,
    ntHeaders->OptionalHeader.ImageBase,
    (uint32_t) memory,
    i86_pte_present|i86_pte_writable|i86_pte_user)になります。

/* 残りの画像のロードとマッピング */
i = 1;
while (file.ef != 1) {...
    新しいフレームを割り当てる */
    符号化されていない char* cur = (符号化されていない char*)pmmngr_alloc_block();
    /* ブロックを読む
    /* int curBlock =
    0;
    for (curBlock = 0; curBlock < 8; curBlock++) { if
        (file.eof == 1)
            ブレークします。
        volReadFile (&file, cur+512*curBlock, 512);
    }
    ページをプロセス・アドレス空間にマップする */
    vmmngr_mapPhysicalAddress (proc->pageDirectory,
        ntHeaders->OptionalHeader.ImageBase + i*4096,
        (uint32_t) cur,
        i86_PTE_PRESENT|i86_PTE_WRITABLE|i86_PTE_USER);
    i++;
}

ユーザ空間スタックの作成 ( プロセスesp= 0 x 100000 ) */ void*
stack =
    (void*) (ntHeaders->OptionalHeader.ImageBase
        + ntHeaders->OptionalHeader.SizeOfImage + PAGE_SIZE);
void* stackPhys = (void*) pmmngr_alloc_block ();

ユーザプロセスのスタック空間をマッピングする。
vmmngr_mapPhysicalAddress (addressSpace, (uint32_t) stack, (uint32_t) stackPhys,
    i86_PTE_PRESENT|i86_PTE_WRITABLE|i86_PTE_USER)となります。

/* 最終的な初期化 */ mainThread->
initialStack = stack;
mainThread->frame.esp =
    (uint32_t)mainThread->initialStack; mainThread->frame.ebp =
    mainThread->frame.esp;

ファイルを閉じてプロセスIDを返す */
volCloseFile(&file);
return proc->id;
}

```

プロセスの実行

プロセスを実行するには、そのプロセスのメインスレッドからEIPとESPを取得し、ユーザーモードに落として実行すればよいのです。しかし、ここで問題が発生します。どのプロセスを実行すればいいのかをどうやって判断するのか？スケジューラがまだないため、一度に1つのプロセスしか実行できません。これは、現在どのプロセスで作業しているかを保存するグローバルプロセスオブジェクトを使用して行われます。**GetCurrentProcess()**は、このオブジェクトへのポインタを返します。そのメインスレッドからESPとEIPを取得し、プロセスのアドレス空間に切り替え、ユーザーモードに落として実行します。

enter_usermodeを呼んでいないことに注意してください。これは、ユーザーモードのソフトウェアがカーネルオンリーのページにアクセスできないためです。これ呼び出すと、ページフォルトになってしまいます。代わりに、ユーザーモードに落ちて、IRETDを使って直接プログラムを実行します。

```

void executeProcess () {
    process* proc = 0;
    int entryPoint = 0;
    unsigned int procStack = 0;

    実行中のプロセスを取得 */
    proc = getCurrentProcess();
        if (proc->id==PROC_INVALID_ID)
            return;
    if (!proc->pageDirectory)
        を返すことができます。

    /* メインスレッドのespとeipを取得 */
    entryPoint = proc->threads[0].frame.eip;
    procStack = proc->threads[0].frame.esp;

    プロセスアドレス空間への切り替え */
    asm cli
    pmmngr_load_PDBR ((physical_addr)proc->pageDirectory)。

    ユーザーモードでプロセスを実行 */
    asm {
        movax,          0x23; ユーザーモードデータセクタは0x20 (GDTエントリ3)。また、RPLを3に設定
        movds, ax
        ムーブ、アックス
        movfs, ax
        movgs, ax
        ;
    }
}

```

```

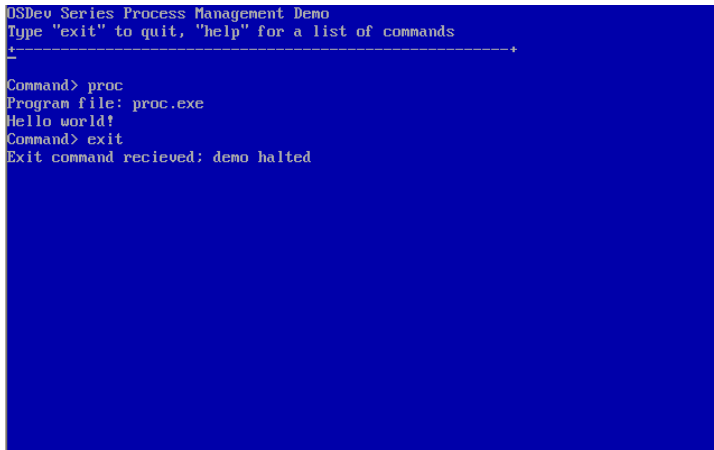
        スタックフレームの作成
        ;

        push0x23; SS, 上記と同じセクタを使用していることに注意してください。
        push[procStack]; stack
        push0x200; EFLAGS
        push0x1b; CS, ユーザーモードコードセクタは0x18。RPL 3では0x1bとなります。

        push[entryPoint]; EIP
        iretd
    }
}

```

デモ



システムAPIを使用するusermodeプロセスの実行

デモダウンロード

これは、開発中のオペレーティングシステムにとって重要なマイルストーンです。このマイルストーンは、インタラクティブ性とセルフホスティングシステムデザインの始まりを意味します。このデモでは、本章、メモリ管理の章、PEのロードの章で説明した内容を用いて、実行イメージ（**Portable Executable format**）をロードする**proc**（プロセス）コマンドを実装し、それをユーザーランドの独自のアドレス空間にマッピングし、2つのシステムコールを通じて**システムAPI**を使用してカーネルと対話します。ユーザーランドのプロセスがカーネルのテキスト端末を使って文字列を表示するために使用できる**DebugPrintf**と、プロセス自体を終了させるために使用される**TerminateProcess**です。このシステムコールは、これまでの章で説明した**ソフトウェア割り込み**を利用して実装されています。

プロジェクト "proc"

使用したusermodeプロセスは**proc**と呼ばれています。これは32ビットのPE実行可能イメージとして構築され、イメージベースは4MB、512バイトのセクションアラインメントです。参考までにこのプロジェクトのソースを紹介します。

```

void processEntry () { ...
    char* str="Hello world!";

    __asm {
        /* カーネル端末にメッセージを表示 */ mov ebx, str
        mov eax, 0
        int 0x80

        /* 終了 */ mov
        eax, 1
        int 0x80
    }
    /* のために (; ; ) 。
```

このプロセスでは、メッセージの表示と終了にシステムコールが使用されていることに注目してください。これらのシステムコールは、これまでの章で実装したシステムAPIに追加されたものです。**Int 0x80関数0**は**DebugPrintf**を、**Int 0x80関数1**は**TerminateProcess**という新しい関数を呼び出しています。デモの機能を向上させるために、同様の方法でファイルの読み込みや入力などのシステムサービスを追加することができます。

TerminateProcessは、プロセスリソースをクリーンアップし、実行をカーネルコマンドシェルに戻す役割を果たします。**int 0x80**が実行されると、CPUはカーネルモードに移行し、CS、SS、ESPをTSSのそれぞれの値に戻すことを思い出してください。このように、システムコールが実行されるたびに、CPUはカーネルランドで同じアドレス空間を実行しています。これにより、**TerminateProcess**から直接カーネル関数を呼び出したリ、カーネルコマンドシェルを呼び出したリすることができます。

```

extern 「C」 {
void TerminateProcess () {
    プロセス* cur = &_amp;proc;
    if (cur->id==PROC_INVALID_ID)
        return;

    /* スレッドの解放 */ int
    i=0;
    thread* pThread = &cur->threads[i];

    スタックの物理アドレスを取得する。
    void* stackFrame = vmmngr_getPhysicalAddress (cur->pageDirectory,

```

```

        (uint32_t) pthread->initialStack);

    スタックメモリをアンマップして解放する。
    vmmngr_unmapPhysicalAddress (cur->pageDirectory, (uint32_t) pthread->initialStack);
    pmmngr_free_block (stackFrame);

    イメージメモリをアンマップして開放する */*.
    for (uint32_t page = 0; page < pthread->imageSize/PAGE_SIZE; page++) { uint32_t
        phys = 0;
        uint32_t virt = 0;

        /* ページの仮想アドレスを取得 */
        virt = pthread->imageBase + (page * PAGE_SIZE);

        ページの物理的アドレスを取得 */
        phys = (uint32_t) vmmngr_getPhysicalAddress (cur->pageDirectory, virt)です。

        /* unmap and release page */
        vmmngr_unmapPhysicalAddress (cur->pageDirectory, virt);
        pmmngr_free_block ((void*)phys);
    }

    カーネルセクタの復元 */
    __asm {
        クライアント
        mov eax, 0x10
        mov ds, ax mov
        es, ax mov fs,
        ax mov gs, ax
        sti
    }

    カーネルのコマンドシェルに戻る */ run ();

    DebugPrintf ("^^^"); for (;;)
}
} // extern "C"

```

バグレポート

過去のいくつかのデモでは修正されていますが、他のデモではまだ存在する可能性があるバグが再発しています。今後、このバグが存在する可能性のあるすべてのデモについて、修正プログラムをアップロードする予定です。このバグは **vmmngr_initialize** にあり、一部のデモでは物理メモリマネージャを初期化する前にこの関数を呼び出しており、これらのデモではカーネル空間のマッピングが不適切なため、ページフォルトやトリプルフォルトが発生する可能性があります。本章のデモでは（再び）解決されていますので、更新されたコードを **main.cpp** と **mmngr_virt.cpp** で確認してください。

ファイルリストの更新

- ◆ **sysapi.h** - **_syscalls**が更新され、**DebugPrintf**と**TerminateProcess**が含まれるようになりました。
- ◆ **task.h** - New.
- ◆ **task.cpp** - 新規。
- ◆ **main.cpp** - **proc**コマンドを追加しました。また、VMMのバグフィックス。
- ◆ **mmngr_virt.h** - 新しいアドレス空間関数です。
- ◆ **mmngr_virt.cpp** - 新しいアドレス空間関数。また、VMMのバグフィックス。
- ◆ **image.h** - PEのイメージ構造と定義。
- ◆ **proc/main.cpp** - 新規です。

結論

この章では、プロセス、スレッド、プロセス管理、および基本的なプロセス管理サポートの構築について説明しました。この章では、ユーザーモードのプログラムをディスクから実行するために必要なことをすべて網羅しており、これはオペレーティングシステムにとって大きな節目となりました。

次章では、本章で実装したプロセス管理機能をベースに、スケジューラーを構築し、プリエンブティブなマルチタスクのサポートを完成させます。

次の機会まで。

～Mike () です。

OS開発シリーズ編集部

リソース

以下のリンクは、より直接的で正確な情報を提供するために参照されたものです。さらなる情報を得るために参照してください。

[http://en.wikipedia.org/wiki/Process_\(コンピューティング\)](http://en.wikipedia.org/wiki/Process_(コンピューティング))

[http://en.wikipedia.org/wiki/Scheduling_\(computing\)#Scheduling_disciplines](http://en.wikipedia.org/wiki/Scheduling_(computing)#Scheduling_disciplines)

[http://en.wikipedia.org/wiki/Process_management_\(computing\)](http://en.wikipedia.org/wiki/Process_management_(computing))

追加リンク

以下のリンクは、このトピックに関連する追加のチュートリアルやリソースです。これらは、教材の補足として、あるいは異なるデザインを提供する際に役立つかもしれません。もし、追加した方が良いと思われるリンクをご存じでしたら、ぜひ教えてください。また、リンク先には、次の章まで詳しく見ないマルチタスクの概念が含まれている場合があります。

http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html



第23章

ホーム



オペレーティングシステム開発シリーズ

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

25: プロセスマネジメント 2

"Controlling complexity is the essence of computer programming" -
Brian W. Kernighan

by Mike, 2015

1. はじめに

Welcome!

前章では、プロセス間通信（IPC）、保護、リソースの割り当て、プロセス制御ブロック（PCB）、プロセスの実行状態、プロセスのアドレス空間など、プロセス管理の基本的なトピックについて詳しく説明しました。また、シングルトasksのサポートと基本的なシングルトasksの実装についても詳しく説明しました。本章では、前章の続きとして、マルチタスク、スケジューリング、セキュリティ、相互排除に重点を置いて、それぞれのトピックについてさらに詳しく説明します。具体的には、以下の項目について説明します。

1. マルチスレッドです。
2. マルチタスクですね。
3. InitとIdleのプロセス。
4. カーネル/ユーザー共有データスペース。
5. 相互排除とセマフォ。
6. コンカレント・プログラミングの紹介。
7. スケジューリングアルゴリズム。
8. MP規格への導入。

本章では、前章を読んでいただいたことを前提に、実際の設計や実装に焦点を当てた、より高度な内容をお届けします。前章と同様に、まずこれらのトピックの背後にある理論に触れ、次にユーザーランドプロセスに完全なマルチスレッドを実装するデモを紹介します。なお、本章ではMP規格について簡単に紹介していますが、詳細については後述します。MPサポートを実装するには、APICの適切なサポートが必要ですが、これは高度なトピックです。

2. プロセスの状態管理

本連載では、プロセスについて多くのことを説明してきましたので、今回は、プロセスの状態とプロセスの作成についてのおさらいです。前章では、プロセスを作成するための関数を実装しました。今回のデモでは、この関数を修正して、プロセスが適切に実行されるように、プロセス用の新しいタスクを作成します。プロセスのスケジューリングと密接な関係があるので、状態管理についても確認しておきましょう。

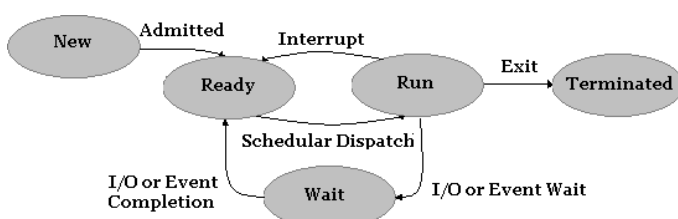
プロセスの**状態**とは、そのプロセスが採用している現在のアクティビティのことです。最低でも、プロセスは**作成**、**実行**、**実行準備**、**終了**が可能で、これだけでも4つの状態が考えられます。

- ◆ **新しい**。プロセスの作成中です。
- ◆ **実行中**です。プロセスが実行されています。
- ◆ **準備完了**です。プロセスを実行する準備ができています。
- ◆ **終了しました**。プロセスが終了しました。

これは良いスタートだと思います。しかし、もっと良い方法があります。例えば、あるプロセスが動作していて、そのプロセスがディスクから大きなファイルを読み取るリクエストを送ってきたとします。しかし、複数のプロセスが存在するシステムでは、ディスクはそのプロセスからの要求を処理するのに忙しくなる可能性があります。私たちのプロセスは、**入出力要求**が完了するまで**待つ**必要があります。別の例として、2つのプロセスがありますが、それらのプロセスは**シグナル**で相互に通信しているとします。プロセスは、シグナルが**発生**するのを**待つ**必要があります。これが5番目の状態になります。

- ◆ **待ちます**。プロセスは、I/Oリクエスト、例外、またはシグナルの完了を待っている。

これらをまとめると、プロセスは次のような状態を経ることになります。



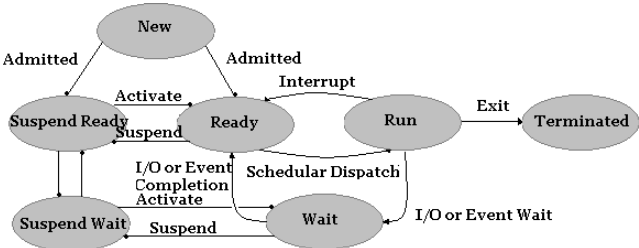
上の図は、現在の状態モデルを示しています。**新しい**プロセスは、システムの**Ready**キューに入れられます。が終了すると

スケジューラ ディスパッチャーが実行するプロセスを選択すると、プロセスは**実行**状態になります。ここから、プロセスはいくつかの状態変化を取ることができます。**割り込み**や**例外**が発生した場合、**スケジューラ ディスパッチャー**は他のプロセスに切り替える必要があり、その際にはプロセスを**レディキュー**に戻すことになります。代わりに、プロセスがファイルから読み取ろうとした場合、プロセスは**I/O要求**を開始し、要求が完了するまで**Wait**キューに入れられます。I/O要求が満たされると、プロセスは**Ready**キューに戻され、再び**Scheduler**ディスパッチャによって選択されることになります。最後に、プロセスの実行中にいつでも、プロセスは終了します。

時には、プロセスを**サスペンド**することが有効な場合があります。これは、プロセスをメモリから取り出し、その状態をディスクに保存するものです。これは、システムリソースを解放し、より優先度の高い他のプロセスを実行できるようにする場合に特に有効です。このためには、最低でもあと2つの状態が必要です。

- ◆ サスペンドレディ。
- ◆ サスペンド・ウェイ
- ト。

これらを先ほどの図に加えると、以下のようになります。



Ready」または「**Wait**」状態のプロセスは、システムのリソース需要に応じて**一時停止**されることがあります。デザインの必要性に応じて、この他にも様々な状態を追加することができますが、ほとんどの汎用OSでは、上記の状態図で十分です。

ここでは、**Ready**、**Run**、**Terminated**の各状態についてのみ説明します。しかし、次のような実装も可能です。付属のデモで**スリープ**機能を適切にサポートするために、**Wait**状態にしています。

3. コンカレント・プログラミング

そこで、プロセスの状態と状態管理、プロセスの作成について見てみました。また、前章ではメモリ上のプロセスについて深く掘り下げました。最後に、**マルチタスク**について説明します。マルチタスクの中心となるのは、次の章で紹介する**スケジューラディスパッチャー**です。スケジューラディスパッチャーは、プロセスの状態を移動させたり、プロセスの実行をスケジューリングする役割を担っています。このセクションでは、これらの機能を使用します。スケジューラの話をする前に、複数の**スレッド**が**実行**されているときのマルチタスクについて詳しく見ておきましょう。2つのスレッドやプロセスが**同時に**実行され、お互いにデータを共有する場合、2つの実行スレッド間のアクティビティを同期させることが重要になります。

同時進行とは、プロセスの現在の状態がわからないことを意味します。複数のプロセスが並行して実行され、互いにデータを共有している状態を「**同時実行**」といいます。**同時実行プログラミング**は、同時実行中のプロセスまたはスレッド間で**共有リソース**へのアクセスを**同期**させるために使用される一連の技術を定義します。

クリティカルセクションの問題

シングルコアのシステムでは、OSは各プロセスに少量の実行時間を割り当てます。システムは、同時に実行されているさまざまなプロセスを迅速に切り替えます。プロセスはいつでも中斷することができます。また、**並列実行**をサポートするシステムでは、異なるプロセスの命令を同時に実行することがあります。

現在のプログラミングの問題を見るために、次のような命令を持つ2つのプロセスを考えてみましょう。

プロセスA mov eax, [count]. EAXのインクリメント mov [count], eax	プロセスB mov ebx, [count]. dec ebx mov [count], ebx
---	---

これらのプロセスを同時に実行する場合、スケジューラーが2つのプロセスを切り替える際に、何らかの順序で**インターリーブ**されることになります。プロセスのインターリーブには様々な方法がありますが、1つの方法は次のようなものです。

```
mov eax, [count]
inc eax
mov ebx, [count]
dec ebx
mov [count], eax
mov [count], ebx
```

countが2つの異なるプロセス間で共有されている場合、ここで大きな問題に気づくかもしれません。**実行順序**が制御されていないため、スケジューラーが2つのプロセスを切り替えることを決定する**タイミング**によって異なる結果が得られる可能性があり、**count**の値が有効であることを保証できません。どちらが先に変数を読み書きするかによって結果が変わります。これは**レースコンディション**と呼ばれています。

競合状態に対抗するには、他のプロセスが使用している間、変数を**保護**する必要があります。何らかの方法で2つのプロセスを**同期**させる必要があります。これは、**クリティカルセクション問題**の一部です。

複数のプロセッサを搭載したシステムでは、1つのプロセスを実行する際に、現在の実行状態と現在の命令ストリームがインターリーブされるため、この問題はさらに悪化します。

問題。

並行して実行されるプロセスの同期を制御する方法が必要です。クリティカルセクションが要求された場合、クリティカルセクションが完了するまで、**1つのプロセスだけがクリティカルセクション内のコードを実行するようにしなければなりません**。さらに言えば、クリティカルセクションに入っている間、他のプロセスやスレッドが実行されないようにしなければなりません。

基準は

- ◆ **Mutual Exclusion**の略。あるプロセスがクリティカル・セクションで実行されているとき、他のプロセスはクリティカル・セクションで実行されていない。
- ◆ **プログレス**。プロセスは、クリティカルセクションに入るまで、いつまでも待っているわけではありません。
- ◆ **バウンデッド・ウェイティング**。そのクリティカルセクションに入るためのリクエストをしてから、実際に入るまでの時間は拘束されていなければなりません。

セマフォ

では、どのようにして**相互排除**を実施するか。2つのプロセスの間に何らかの**協力関係**が必要です。**プロセスAが共有リソース上で動作していて、プロセスBがそのリソースへのアクセスを必要とする場合、プロセスBには待ってもらいたい**。しかし、**プロセスAがリソースを使い終わったら、プロセスBがそのリソースを使えるようになったことを知らせる必要があります**。このように、常に1つのプロセスだけが共有リソースを使用することができます。これが**相互排除**です。

私たちができることは、リソースが現在使用されているかどうかを追跡するために、別の変数を導入することです。この変数を**ロック**と呼びます。そして、このロックを使って、他のリソースを追跡することができます。

- ◆ ロックが**1**の場合、そのリソースは他のプロセスによって使用されて
- ◆ います。ロックが**0**の場合、そのリソースは自由に使用できます。

この種のロックには特別な名前があります。それは「**ミューテックス**」と呼ばれるものです。ミューテックスは2つの値しか持たず、**バイナリセマフォ**とも呼ばれます。必要なことを思い出してください。一方のプロセスが**待機し**、もう一方のプロセスが**シグナルを送る**必要があります。これらは、この章を通して使用する基本的な関数です。

```
atomic Wait (Semaphore S) {
while (S <= 0)
S.Queueにプロセスを配置してブロックし
ます。S--;
}
atomic Signal (Semaphore S) {
S++;
}
```

ミューテックスは、0か1の値しか持たない**2値セマフォ**に過ぎません。**セマフォ**は一般的なロックであり、制限はありません（つまり、ミューテックスが2つの値しか持たないのに対し、一般的なセマフォはそうではありません）。また、上記のコードには**atomic**キーワードがあることにも注目してください。これは、コードが実行されても決して中断されないことを意味しています。つまり、**1つのプロセス上でコードのブロックとして正しい順序で実行されることが保証されているのです**。これらは**1つのユニットとして扱われることになっています**（そのため、アトミックオペレーションと呼ばれています）。

残念ながら、上で示したような単純なものではありません。**アトミック操作はハードウェアに依存しているため、動作させるためにはプロセスの支援が必要です**。具体的には、**LOCK**命令のプレフィックスを利用する必要があります。この点については、後にこれらのプリミティブを実際のコードに実装する際に、より詳しく説明します。

今のところ、セマフォの使用例を見るのが一番だと思います。セマフォは最初に導入するときには難しいかもしれませんが、マルチプロセッシングを完全にサポートするためには、セマフォを頻繁に使用することになるので、セマフォの使い方を練習することが重要です。

◆ 例

このセクションの冒頭では、異なるプロセスを入れ替える際に、命令フローがどのように交錯するかを示しました。問題は、どちらのプロセスもいつでも実行可能であり、リソースを共有しているため、リソースの整合性を確認する方法がないことでした。この問題はセマフォで解決できます。**count**が2つのプロセスで共有されるグローバル変数であると仮定すると、セマフォを使用してそのアクセスを同期させることができます。**signal**と**wait**は**アトミックな操作**であることに注意してください。

プロセスA count++です。 シグナル (S) です。	プロセスB を待つことになります。 カウンタ--。 シグナル (S) です。
-------------------------------------	---

スピンロック

相互排除は、クリティカルセクション問題を解決するための最初の基準です。これは、あるプロセスがクリティカルセクションに入ると、他のプロセスはクリティカルセクションに入ることができないということです。この機能を実装するためには、相互排除を保証できるような**アトミックな操作**を実装する何らかの方法が必要になります。1つのアイデアは、単純な変数をロックの役割として使用することです。ロックが**1**であれば、あるプロセスがクリティカルセクションの内部にいることになります。そこで、最初のアイデアは

int lock=0;

プロセスA while(1) { if (!lock) lock = 1; do_something(); lock=0; }	プロセスB while(1) { if (!lock) lock = 1; do_something(); lock=0; }
---	---

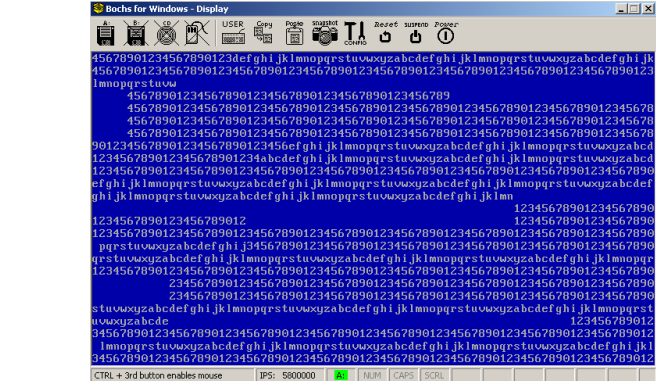
かなりシンプルですね。ロックは0から始まるので、最初に実行されたプロセスはこれを検知してロックを設定します。それが終わるとロックを解除し、2番目のプロセスがそれを使えるようにします。これはある程度うまくいきますが、まだ大きな問題があります。例えば、プロセスAがロックが0であることを検出したが、プロセスAがロックを設定する前にプロセスBによって中断されたとします。そこで、プロセスBはロックが0であることも検出し、今度はロックを設定します。そのため、プロセスBがdo_somethingのどこかで中断されても、プロセスAはロックが0のままであるかのよう

つまり、ここでの問題は、ロックへのアクセスと設定が、中断されることなく行えることを保証できないということです。この操作はアトミックではありません。

実際にアトミックな操作をしなくても何が起きるかをイメージできるように、2つのスレッドがあるとします。1つ目のスレッドは文字のa-zを表示し、2つ目のスレッドは数字の0-9を表示します。この2つのスレッドは、後で開発するスケジューラを使って同時に実行されます。以下がそのスレッドです。

プロセスA void task_1() { char c='a'; while(1) { DebugPutc(c++); if (c>'z') c='a'; } }	プロセスB void task_1() { char c='0'; while(1) { DebugPutc(c++); if (c>'9') c='0'; } }
---	---

この2つのタスクが並行して実行されると、出力はインターリーブされた混乱状態になります。理由は、両方のプロセスが共有リソースを無頓着に読み書きしているからです。先に述べたようにロックを導入したとしても、出力はあまり良くなりません。この例では、共有リソースはビデオメモリと、カーソルの位置決めとスクロールを担当するDebugPutcが使用するグローバル変数です。あるプロセスが現在のxやyの位置を読み取ったり、スクロールの準備をしていると、そのプロセスが中断され、最初のプロセスが知らないうちに位置やその他のグローバル変数が変更されてしまう可能性があります。



セマフォを使わないサンプル。出力がめちゃくちゃになっていることに注意してください。

この問題を解決するためには、単純なロックだけではなく、何かが必要です。私たちの方向性は良いのですが、ハードウェアのサポートが必要です。もし、ロック変数のテストと設定を1回の操作で行うことができる方法があれば、絶対に中断されないことが保証されるので（つまりアトミック）、最終的には相互排除基準を満たすことができます。

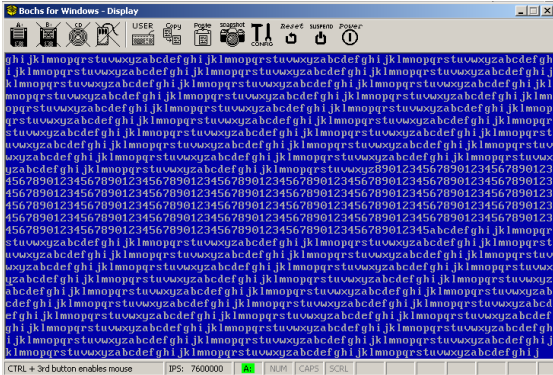
そのようなハードウェアプリミティブの一つが、LOCK命令プレフィックスです。このプレフィックスは、その命令が実行されている間、システムバスをリード/ライトからロックします。データベースがロックされているので、アトミック性が保証されています。そのため、ロック変数の設定やテストを行う際には、単純に LOCK XCHG や LOCK BTS を使用することができます。例えば、以下ようになります。

```
inline void acquire(int* lock) {...  
  _asm {  
    mov eax, [lock].  
A: ロックBTS [EAX], 0  
    ポーズ  
    jc a  
  }  
}  
  
inline void release(int* lock) {...  
  _asm {  
    mov eax, [lock]  
    mov [eax], 0  
  }  
}
```

これで、これらの関数を呼び出して、ロックの獲得と解放ができるようになりました。

プロセスA void task_1() { char c='a'; while(1) { acquire(lock); DebugPutc(c++); release(lock); if (c>'z') c='a'; } }	プロセスB void task_2() { char c='a'; while(1) { acquire(lock); DebugPutc(c++); release(lock); if (c>'z') c='a'; } }
---	---

そして望み通りの結果を得ることができる。



スピンロックを使ったサンプルの実行。表示がきれいに整ったことに注目してください。

4. 古典的な同時実行問題

生産者・消費者問題（境界線付きバッファ問題）

これは、最初に見ていく古典的な同時実行問題です。2つの独立したプロセスがあるとします。1つは**プロデューサ**、もう1つは**コンシューマ**と呼ばれます。また、両方のプロセスが使用している共有バッファがあるとします。プロデューサーはバッファにデータを入れる役割を果たし、コンシューマはデータを取り出す役割を果たします。これが、古典的な「**プロデューサー/コンシューマ問題**」 (**Bounded Buffer Problem**) の基本的な設定です。問題は、プロデューサーが、バッファがすでにいっぱいになっている場合にデータを追加しないようにすることと、コンシューマが、空のバッファからデータを取り出そうとしないようにすることです。この問題は、複数のプロデューサーとコンシューマがいる場合に、より興味深いものとなります。

例

これは、Bounded Buffer問題の解決策です。これは、1つのプロデューサーとコンシューマが同時に動作していることを想定しています。

セマフォc=0です。

Semaphore s = BUFFER_SIZE;

while (true)

item = produce

read();

al(s)

e(item)。

ProducerConsumer

{while (true) {

() ;wait(c);

wait(s);item =

write(item);sign

signal(c);consum

}}

読者と作家の問題

典型的な**リーダー/ライター問題**は、あるオブジェクトが多くのプロセスで共有され、**リーダー**と**ライター**という2種類のプロセスが存在する場合です。**読者**は共有データを読みますが、変更することはありません。**ライター**はデータを読み、変更することができます。**多くのリーダーが同時にデータを読むことができます。**

例です。

この問題には様々な解決策やバージョンがありますが、これはそのうちの一つです。ここでは、複数の読者を同時に許可するために、2つのセマフォを使用していることに注意してください。

セマフォのc = 1;

セマフォのs = 1;

int count = 0;

ライター

while(true)

{

wait(c);

write();

signal(c);

}

リーダー

{while(true)

{

wait(s)です。

if (count == 0)

wait(c);

signal(s);

read(s);

を待つことになり

ます。

カウント--

if (count == 0)

signal(c);

} の信号を送信しま

す。

5. プロセス間通信

IPC (Inter-Process Communication) とは、OSがサポートする技術で、プロセスが他の実行中のプロセスと信号やデータを共有することを可能にする。IPCプロトコルの実装にはさまざまな種類の技術があるが、ここではよく使われるものを紹介する。

パイプ

パイプは、プロデューサーとコンシューマーの間でデータを保存するために円形のバッファを使用する基本的な技術です。プロデューサーはバッファにデータを書き込み、コンシューマーはバッファからデータを読み出します。データのプロデューサーとコンシューマーは複数存在することができます。パイプには、匿名パイプと名前付きパイプの2種類があります。**名前付きパイプ**には名前が付けられ、仮想ファイルシステムのファイルオブジェクトとして表示されます。システム内のどのプロセスでも名前付きパイプを開くことができます。**匿名パイプ**は、親プロセスから継承した子プロセスのみが開くことができます。

オペレーティングシステムは、コンシューマーとプロデューサーの間で共有されるデータストリームを**格納する機能**、ストリームを**読み書きする機能**、読み取るべきデータがないときにパイプから読み取ろうとするプロセスを**ブロックする機能**を提供する必要があります。オペレーティングシステムは、上で説明した相互排除技術を使って、読み書きを**同期**させなければなりません。これは通常、**先入れ先出し (FIFO)** の**円形バッファ**を使用し、**セマフォ**を使用して読み書き時のアクセスを同期させます。

パイプは**ファイルシステムのオブジェクト**です。パイプを開くと、**ファイル記述子**のポインタが戻ってきます。そのため、ファイルの**Read**および**Write**メソッドを使用して、パイプをファイルのように読み書きすることができます。**開いたファイルハンドル**は**子プロセスに継承**されますので、パイプも継承されます。

パイプは、ファイルシステムの記述子と同じように管理することができます。**プロセス・パラメータ・ブロック**には、ファイル・ディスクリプターやパイプなどのシステム・オブジェクトへのオープンな参照をすべて格納した**プロセス・ハンドル・テーブル**へのポインタが格納されています。また、**デバイスファイル**をすでにサポートしているシステムでは、些細なことでも実装することができます。

メッセージの受け渡し

基本的には、プロデューサーが**メッセージ**を送信し、コンシューマーがそれを受信するというシンプルなアイデアです。しかし、メッセージの受け渡しを**同期**で行うか、**非同期**で行うかによって、さらに問題が出てきます。また、メッセージをどのように**保存**するか、どこで**管理**するか、メッセージの**フォーマット**、メッセージが期待されたフォーマットで期待されたプロセスに配信されているかどうかをどのように検証するかなども考えなければなりません。

では、メッセージとは一体何なのでしょう？メッセージは、プロセスが望むものなら何でもあります。コンシューマーとプロデューサーは、メッセージをどのように解釈するかについて、何らかの**プロトコル**に同意する必要があります。両者はメッセージの**データ構造**を知っている必要があります。オペレーティングシステム側では、**マイクロカーネルのようにOSが定義したメッセージでない限り**、OSはデータのフォーマットを気にしません。

オペレーティングシステムには、最低限、メッセージの送受信をサポートする機能が実装されている必要があります。

同期型メッセージパッシング

同期メッセージパッシングには、最低でも2つの関数が必要です。**J**と**K**をプロセス識別子(PID)とすると

- ◆ send(J, message)
- ◆ receive(K, &message)

producerは**send**を呼び出してメッセージを投稿します。同期メッセージパッシングでは、producerは、**J**が**receive**を呼び出してメッセージを取得するまで、**中断されたキュー**に入れられます。**J**が**receive**を呼び出すと、オペレーティングシステムは、**J**に送られたメッセージを直接コピーし、**J**を再開することができます。その後、オペレーティングシステムは、プロデューサーを**待ち行列**に戻し、スケジューラが実行できるようにします。同期メッセージ・パッシングでは、同時に実行されるのは一方（プロデューサーまたはコンシューマー）だけなので、メッセージ・キューは必要ありません（他方は**サスペンド**または**ウェイティング**になります）。

非同期のメッセージパッシング

非同期のメッセージパッシングにも最低2つの関数が必要です。

- ◆ send(J, message)
- ◆ receive(K, &message)

プロデューサーは **send** を呼び出してメッセージをポストし、コンシューマーは **receive** を呼び出してメッセージを取得します。非同期メッセージパッシングでは、OSはプロセスごとに**メッセージキュー**を管理します。プロデューサーはいつでもメッセージを送信でき、中断されることはありません。メッセージはメッセージキューの最後にコピーされます。その後、消費者はメッセージキューの先頭からメッセージを受け取ることができます。メッセージキュー自体は**カーネルメモリ**内に割り当てられ、プロセス制御ブロック内の専用ポインタがキューを指し示します。

ここで、興味深い問題があります。同期メッセージパッシングでは、プロセスが**receive**を呼び出したときに、メッセージを送信したプロセスがないと、他のプロセスが**send**を呼び出すまでプロセスが中断してしまいます。非同期のメッセージパッシングでは、2つの選択肢があります。

- ◆ 受信やを呼び出したプロセスを**中断**することができます。
- ◆ 受信側にステータスコードを返させて、現在のプロセスを**継続**させることができます。

やはり、もう少し機能を充実させた方が良いのではないのでしょうか。

- ◆ send(process, message)
- ◆ receive(process, &message)
- ◆ sendrec(process, &message)
- ◆ notify(process, message)

共有メモリ

同じ物理フレームを2つ以上のプロセスの仮想アドレス空間にマッピングすると、そのプロセス間で**共有**されます。両方のプロセスが同じページを読み書きできるようになります（ページをマッピングする際に設定された**セキュリティ属性**によります。（例えば、物理フレームをプロセスAでは読み取り/書き込みとしてマッピングし、プロセスBでは読み取り専用としてマッピングすることができます）。）オペレーティングシステムでは、一般的に**メモリマップドファイル**によって共有メモリをサポートしています。例えば、Windowsでは、まず、名前の付いたメモリマップドファイルオブジェクトの**CreateFile**または**OpenFile**を呼び出し、続いて、メモリの領域をプロセスアドレス空間にマップし、そのポインタを返す**MapViewOfFile**を呼び出します。

6. スケジューリング

スケジューラーは、システムリソースの割り当てを行います。システムリソースには、CPU、メモリ、システムデバイスなどがあります。スケジューラーは一般的に多数存在しますが、**短期**、**中期**、**長期**の3つのカテゴリーに分類される傾向があります。

1. **長期スケジューラー**は、プロセスをシステムに入れたり、終了させたりする役割を担っています。
2. **中期スケジューラー**は、プロセスの中断・再開を担当します。
3. **短期スケジューラー**は、CPU時間の割り当てとプロセスのディスパッチを担当します。

このセクションでは、マルチタスク・システムを実装するためのコア・コンポーネントである短期スケジューラについて主に説明します。そこで、ここでのデモの目標は、**短期スケジューラ**を作成することです。

スケジューリングアルゴリズム

使用できるアルゴリズムには様々なものがあり、中には複雑なものもあります。ここでは、一般的なアルゴリズムを紹介しますが、デモをシンプルにするため、**ラウンドロビン**方式を採用します。

First Come First Serve

FCFS (First Come First Serve) では、ジョブは来た順に実行されます。アルゴリズムはその名の通りシンプルで、スケジューラーが最初のジョブを選択して実行させます。次に2番目のジョブ。次に**2番目のジョブ**、そして次のジョブ.....という具合です。このアルゴリズムは、**Ready Queue**に入っているジョブを、入ってきた順に循環させます。前のジョブが終了するまで、新しいジョブは開始されません。このアルゴリズムは、プリエンベティブなマルチタスクにはあまり適していません。

例

次の例では、P1が時刻0に到着し、P2が時刻1に到着し、P3が時刻2に到着しています。これらのプロセスは、実行されるために**Readyキュー**に入れられます。P1は最初のジョブなので、アルゴリズムはそれを選択して実行します。次にP2が選択されますが、これはP1が完了してからです。P2が選択されるのは時刻5になってからです。

ProcessArrive ランタイム サー	
ビスタタイム	
P1050	
	P2135
P328	8

最短の仕事を最初に

SJF (Shortest Job First) アルゴリズムでは、各ジョブの実行に必要な時間をシステムが知る方法が必要です。このアルゴリズムでは、**Ready Queue**から次に実行されるジョブのうち、時間の差分が最も小さいものを選択します。このアルゴリズムには、**プロセスの飢餓**の問題があります。時間差が小さいジョブが優先されると、ジョブが**Ready Queue**に残ってしまいます。この例は、前述の**FCFSアルゴリズム**と非常によく似ていますが、時間差を計算する必要があるため、実際にはほとんど実装されていません（プロセスが実行される時間を事前に知るためには、ソフトウェアがオラクルである必要があります）ので、別の例は必要ないと思います。

優先キュー

このシステムでは、各ジョブに**優先度の高い番号**を割り当てることができます。優先度の高いジョブが先に選択される。これが**優先スケジューリングアルゴリズム**の基本的な考え方です。優先度をどのように決定するかは設計者の自由です。同様に、2つの優先度が同じ場合にどのように処理するかも設計者次第である。ひとつのアイデアは、デフォルトの優先順位を設け、それをユーザーが調整できるようにすることです。2つの優先度が同じ場合は、**FCFS**や**SJF**を使ってどちらを使うかを決めます。もう一つのアイデアは、**プロトコル**に基づいて優先順位を計算することです。プロトコルは、システム管理者が割り当てたり、システムのリソースやメモリの制約を測定して算出したりします。後述するように、優先順位は他のスケジューリングアルゴリズムと一緒に使われることが多いです。

要約すると、**Ready Queue**から優先度の高いジョブを選択すればよいのです。このアルゴリズムは、**SJF**と同様、優先度の高いプロセスが優先度の低いプロセスを駆逐してしまうため、**プロセスの飢餓状態**に陥ります。

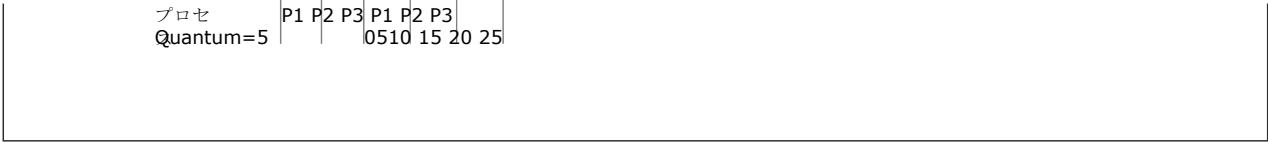
ラウンドロビン

システムは、各プロセスに**量子**と呼ばれる実行するためのタイムスライスを与えます。その後、システムは現在実行中のプロセスを**先に実行**し、別のプロセスの実行を許可します。プロセスは、**レディキュー**に表示された順に選択されます。すべてのプロセスの実行が許可されるため、このアルゴリズムはプロセスを飢えさせることはありません。システムは、実行するように選択されたプロセスの**実行状態**を保存・復元するために、**コンテキストスワッピング**を行います。**コンテキストスワッピング**については、後ほど**マルチタスク**について説明する際に取り上げます。

例

プロセスP1、P2、P3と時間量子5が与えられたとき、**ラウンドロビン (RR)** アルゴリズムは、まずP1を選択して実行します。量子時間が経過すると、システムはP1を**プリエンブト**します。P1は **Ready Queue** の後ろに移動します。システムはP1の**コンテキスト**を保存します。アルゴリズムがP2を選択すると、システムは**コンテキストスイッチ**を実行します。P2が実行できるようになりました。

| | | | | |



マルチレベル・キュー

次に何を実行するかを決めるのに、1つの**Ready Queue**を使う代わりに、**複数のReady Queue**を使つてはどうでしょうか。特権レベルと別のスケジューリングアルゴリズムを、**マルチレベルキュー**に組み合わせることで、両方の世界を手に入れることができるというものです。

基本的な考え方は、**複数のキュー**を持つことです。そして、これらのキューは**異なる優先度**のためのものです。例えば、**5つの優先度がある場合、5つのキューを持つことになります**。アルゴリズムはまず、優先度の高いキューから、優先度に基づいて実行するジョブを選択します。そのキューに複数のジョブがある場合は、別のアルゴリズム（**RR**など）を使って実行するジョブを決定します。また、異なる優先度のキューに対して、異なるスケジューリングアルゴリズムを使用することもできます。このアルゴリズムは、**優先度の高いスケジューリング**と同じ理由で、**プロセスを飢えさせてしまう**可能性があります。このように、素晴らしいアルゴリズムがありますが、**プロセスのスタービング**を防ぐにはどうしたらよいでしょうか。

マルチレベル・フィードバック・キュー

Multilevel Feedback Queueは、**プロセスの飢餓**を防ぐために、**マルチレベルキュー**を改良したものです。マルチレベルキューの問題点は、ある優先度LのプロセスがキューLに挿入されたとき、優先度がLよりも大きい新しいジョブを投入するだけで、プロセスを飢えさせてしまうことでした。つまり、ある優先度のキューから別の優先度のキューにプロセスを移動させることができるのです。

上記の例では、優先度Lのプロセスは、ある程度の時間が経過すると、より高い優先度のキューに移動します。これは、プロセスが最高の優先度のキューに到達するまで続きます。したがって、プロセスが飢餓状態に陥ることはありません。また、重要なシステムタスクを実行する必要がある場合に役立つかもしれない、より低い優先度のキューに移動させることで、ジョブの優先度を下げることができます。マルチレベルのフィードバックキューを実装する際の難しさは、いつプロセスを移動させるべきかを決定することです。これは、今日の最新のオペレーティングシステムで使用されている最も一般的なアルゴリズムです。

例

以下は、**マルチレベルキュー**の例です。ここでは **3** つのキューがあり、システムプロセスが最も高い優先度を持ち、アプリケーションが最も低い優先度を持ちます。異なるキューからジョブを選択するために、それぞれのキューに異なるスケジューリングアルゴリズムを使用することができます。スケジューラは、最も優先度の高い空でないキューを選択します。そして、別のアルゴリズム（**FCFS**や**RR**など）を使用して、そのキューからジョブを選択します。**マルチレベルフィードバックキュー**では、システムは異なるキュー間でプロセスを移動させることができます。例えば、**L3→L2→L1**と時間をかけてジョブを移動させることで、優先度を上げて実行できるようにします。したがって、プロセスの飢餓状態は発生しません。

キューレベル	優先キュー
	L1 システムプロセス
	L2バッチジョブ
	L3アプリケーション

7. マルチタスキング

この章では、これまで**多く**の内容を取り上げてきました。そしてようやく、この章のメインとなる部分にたどり着くことができます。**マルチタスクである**。すべてをコードにまとめていきます。

最初に**プロセスの状態管理**を取り上げたのは、**スケジューラ**や**マルチタスクコンポーネント**がプロセスを選択し、異なる**状態**間で移動させる必要があるからです。例えば、**スケジューラ**はプロセスを**Ready**から**Running**に切り替える必要があります。より高度なページング技術（**ページ・スワッピング・アルゴリズム**など）をサポートする予定であれば、**Suspended**状態との間でプロセスを切り替えられるようにする必要があります。システムは、**Suspended**プロセスと、メモリ内で完了信号を待っているプロセスとを区別する必要があります。どちらのプロセスも**PCB (Process Control Block)**を持ち、システムリソースを使用しますが、**Suspended**プロセスはメモリを使用しません。また、プロセスを一時停止する方法が必要でした。そのために、**Wait**ステートを導入しました。ご覧のように、状態管理はマルチタスクを実装するための重要なコンポーネントです。そのため、最初にこれを取り上げました。

次に見たのは「**プロセス作成**」です。状態管理でどのように使われるのかを詳しく調べました。**第24章**では、**CreateProcess**関数を実装しました。この関数では、**PE (Portable Executable)** イメージをメモリにロードし、仮想アドレス空間にマッピングし、ユーザーモードで実行したことを思い出してください。このセクションでは、この関数を使って新しいプロセスを作成し、**スケジューラ**が選択する**Readyキュー**に追加していきます。

続いて、**コンカレント・プログラミング**の入門編を見てみましょう。**クリティカルセクション問題**」「**相互排除**」「**セマフォ**」などのトピックを取り上げました。**並行処理**とは、複数のプロセスやスレッドが**非同期**に実行されることです。並行プログラミングは、**非同期**のプロセス間の通信を**同期**させるための技術を提供します。並行プログラミングは**難しく**、**正しい**方法はありません。並行処理を使用すると、コードには必ずバグがあります。この章のテーマがマルチタスクであることから、コンカレント・プログラミングを紹介しました。共有リソースはマルチタスクと密接な関係があるので（一般的には共有ライブラリ、シグナル、メッセージパッシングなどの形で）、ここでは簡単な紹介をしました。

続いて、**IPC (Inter-Process Communication)** の紹介をします。IPCは、シンプルなOS以外では重要な役割を果たします。また、マルチタスクでIPCをサポートするシステムには、本章で説明するコンカレント・プログラミング技術が必要です。**システムコール**を使ってIPCの一形態をすでに使用しています。

最後に**スケジューリングアルゴリズム**を取り上げました。スケジューラは、オペレーティングシステムの心臓部です。実行するプロセスを選択する役割を担っており、マルチタスクシステムの中核となるアルゴリズムです。

さて、いよいよ今回は、マルチタスクOSの世界に飛び込んで、まとめていきます。ご存知のように、マルチタスクには3つ

のタイプがあります。

1. プリエンプティブ
2. ノンプリエンプティブ
3. 協同組合

ここでは、「プリエンプティブ・マルチタスク」に注目します。

プラン

今回は、**ラウンドロビン (RR)** というスケジューリングアルゴリズムを使用します。このアルゴリズムでは、選択されるプロセスにリソースとして**量子**を割り当てられることが必要です。そこで必要になるのが**クロック**です。システムには様々な種類のクロックがあります。

1. プログラマブル・インターバル・タイマー (PIT)
2. アドバンスト・プログラマブル・インタラプト・コントローラ (APIC) タイマー
3. リアル・タイム・クロック (RTC)
4. ハイパフォーマンス・イベント・タイマー (HPET)
5. などです。

今回のデモでは、PITがすでにサポートされているため、PITを使用することにします。これで、使用するスケジューリングアルゴリズムとクロックが決まりました。第24章では、**PCB (Process Control Block)** と **TCB (Thread Control Block)** を紹介しました。今回はTCBを拡張して、現在のスレッドの状態を保存したり、ユーザーモードからカーネルモードへの切り替えに必要な情報を追加します。

```
typedef struct _thread {
    uint32_t esp;
    uint32_t rnelEsp;
    uint32_t rnelSs;
    struct _process* parent;
    uint32_t ority;
    intstate ktime_tsleepTimeDelta
}thread;
```

スレッドに関連するタスクを作成するには、いくつかの低レベルのものがが必要です。**スタック**は、現在の**レジスタコンテキスト**を格納します。上の構造体の**esp**フィールドに指定されたスタックに、レジスタコンテキストを格納します。**スケジューラ**は、タスクの**作成**、タスクの**管理**、およびタスクの**切り替え**を行います。以下のセクションでは、それぞれの機能を詳しく説明します。なお、サンプルコードはすべて本章最後のデモプログラムで使用しています。

レディキュー

まず、これらのタスクを格納する場所が必要です。タスクは、カーネルのメモリアロケータによって、ページングされていないプールから動的に割り当てられるべきです。しかし、シリーズにはカーネルアロケータが実装されていないので、実装には配列を使うしかありません。**ラウンドロビン・スケジューリング**に必要な**先入れ先出し**の機能は、循環型の待ち行列を使って実装することができます。これは、キューの一番上の要素を削除して後ろに追いやるだけで、次のタスクに移行できるというものです。つまり、新しいタスクがキューの一番上になるのです。

```
スレッド _readyQueue [THREAD_MAX];
int _queue_last, _queue_first;
スレッド _idleThread;
スレッド*_currentTask;
スレッド _currentThreadLocal;

キューをクリアします。
void clear_queue()
{
    _queue_first = 0;
    _queue_last = 0;
}

スレッドを挿入します。*/
bool queue_insert(thread t) {
    _readyQueue[_queue_last % THREAD_MAX] = t;
    _queue_last++;
    return true;
}

スレッドを削除します。ス
レッド queue_remove() {
    糸t;
    t = _readyQueue[_queue_first % THREAD_MAX];
    _queue_first++;
    return t;
}

キューの先頭を取得します。
*/ スレッド queue_get() {
    return _readyQueue[_queue_first % THREAD_MAX];
}
```

今回の例では、準備の整ったタスクのための単一のキューを実装するだけです。タスクは、キューをシャッフルすることで、いつでも削除、追加することができます。ここで、**_currentTask** ポインタに注目してください。第 25 章では、このポインタは常に **_currentThreadLocal** を指し、現在実行中のスレッドのローカルコピーを保存します。**ISR** はこのポインタを使ってスレッドの状態を保存したり復元したりします。次のセクションでは、この**ISR**について説明します。

割り込みサービスルーチン (ISR) について

さて、最初の課題は、タイマーがトリガーされたときにスケジューラーを呼び出すことです。ハードウェア割り込みは、割り込みコントローラ（ここではレガシーの**Programmable Interrupt Controller (PIC)**）によって発生します。もちろん、**マルチプロセッサ(MP)**やCPU間のIRQに使用される**APIC(Advanced PIC)**などもありますが、本シリーズではシンプルにするためにレガシーPICインターフェースのみをサポートしています。PICは、PITから送られてくるIR#0信号のように、ハードウェアデバイスがPICに信号を送ると、CPUに信号を上げます。そして、PICは別の信号（この場合はCPUのIRQライン）を上げてCPUに通知します。どのIRQが呼び出されるかは、PICをどのようにプログラムしたかによります。IR#0をISR33にマッピングするようにPICをプログラムしたことを思い出してください。これが意味するところは、PITが発火するたびに、CPUは現在のコードの実行を停止し、リターンcs、eip、フラグを現在のスタックにプッシュしてから、**IDT (Interrupt Descriptor Table)** にインストールしたISR、つまり**IDT[33]**を呼び出すということです。

つまり、タイマーISRはすでに割り込みベクター33にインストールされています。これはプロテクトモードの設定時に行いました。これは、ハードウェア割り込みを有効にするために必要でした。それはそれでいいのですが、私たちがしたいのはそれを**を上書き**することです。

これを実現するのが**割り込みチェイニング**です。前の章で割り込みチェイニングを紹介しましたが、実際に実践することはありませんでした。今までは、です。必要なのは、古いISRを取得して、独自のISRをインストールすることです。今すぐ実行しましょう。

```
/* register isr */
old_isr = getvect(32);
setvect (32, scheduler_isr, 0x80);
```

簡単ですね。**IDT**の話をしたときに**getvect**と**setvect**を実装しました。それを**IDT[32]**にインストールしたのは、そこに**PIT** ISRがあったからです。これは、**old_isr**に保存して、新しいISRである**scheduler_isr**をインストールするためです。

以上のことを考慮して、PITが起動するたびに、代わりに**scheduler_isr**が呼ばれることになります。さて、次は難しい部分、つまりISRの記述です。ISRが何を必要とするのか、いつ呼び出されるのかを考えてみましょう。**ISRはいつでも呼び出すことができます**。しかし、**タスクが実行されているときは常に呼び出されます**。必要なのは、現在のレジスタの状態を保存し、スケジューラーを呼び出すことです。PICに**EOI (End-Of-Interrupt)** を送ることも忘れずに。

まず、今回のデモのために実装されたISRを紹介し、以下ではそれが何をしているのかを詳細に説明するために、パーツごとに分解していきます。

```
__declspec(naked) void _cdecl scheduler_isr () {
    _asm {
        ;
        ; 割り込みを解除してコンテキストを保存します。
        ;
        ; クリプ
        ; シュド
        ;
        ; 現在のタスクがない場合は、単にリターンします。
        ;
        mov eax, [_currentTask] cmp
        eax, 0
        jz interrupt_return
        ;
        ; セレクターを保存します。
        ;
        ; プッシュ
        ; ユ ds
        ; プッシュ
        ; ユ es
        ; プッシュ
        ; ユ fs
        ; プッシュ
        ; ユ gs
        ;
        ; カーネルセグメントに切り替えます。
        ;
        mov ax, 0x10
        mov ds, ax
        mov es, ax
        mov fs, ax
        mov gs, ax
        ;
        ; SAVE STATION
        ;
        mov eax, [_currentTask] mov
        [eax], esp
        ;
        ; 呼び出しスケジューラー。
        ;
        call scheduler_tick
        ;
        ; を復元します。
        ;
        mov eax, [_currentTask] mov
        esp, [eax].
        ;
        ; tss_set_stack (kernelSS, kernelESP)を呼び出します。
        ; このコードは、後でユーザーのタスクで必要になります。
        ;
        push dword ptr [eax+8] push
        dword ptr [eax+12] call
        tss_set_stack
        esp, 8を追加
        ;
        ; EOIを送信して文脈を復元します。
        ;
        ; ポップ
        ; G、ポ
        ; プ F
        ; 、ポッ
        ; プ ES、
        ; ポップ
        ; DS
        interrupt_returnで
        ;
        ; 古いISRを呼び出す必要があるかどうかをテストします。
        ;
        mov eax, old_isr
    }
```

```
cmp eax, 0
jne chain_interrupt
;
```

old_isrがnullの場合、EOIを送信して戻る。

```

;
mov al, 0x20
out 0x20, al
popad
iretd
;
old_isrが有効であれば、それにジャンプする。これは
PITタイマーの割り込みハンドラを作成します。
;
chain_interrupt。
ポバド
jmp old_isr
}
}

```

ISRは、**現在のレジスタ・コンテキストの保存**と、**現在のタスクのスタック・ポインタの保存**を行います。その後、**スケジューラを呼び出して、現在のタスクのスタック・ポインタを復元し、前に保存したレジスタ・コンテキストを復元**します。すべてが復元されているので、ISRが戻ってきたときには、タスクは問題なく実行され続けています。ISRは一見複雑に見えますが、実際にはそうではありません。もう少し詳しく見てみましょう。他のISRと同様に、まず最初に行うことは、現在のレジスタの状態を保存してスタックに保持することです。つまり、ISRは次のように始まります。

```

__declspec(naked) void _cdecl scheduler_isr () {
    _asm {
        cli
        pushad

        ポバドア
        イレド
    }
}

```

PITによってインストールされたISRの上にISRをインストールするので、ここでは細心の注意を払う必要があります。これは、**scheduler_isrがクロックティックごとに呼び出されることを意味します。setvectを呼び出してインストールすると、レディキューにタスクがないうちにPITが起動してしまうことがあります**。実行するタスクがないときは、何もすることがないので、ISRを返してほしいだけです。また、割り込みを無効にしても、元に戻さないことに気づくかもしれません。これは問題ありません。現在実行中のタスクは、**FLAGS**レジスタを通じて割り込みを有効にします。**FLAGS**レジスタはすべてのケースで保存されているので、**IRETD**を発行すると、リターン時に**FLAGS.IF**が有効になり、割り込みが再び有効になります。私たちのISRは次のようになります。

```

__declspec(naked) void _cdecl scheduler_isr () {
    _asm {
        cli
        pushad
        ;
        ; 現在のタスクがない場合は、単にリターンします。
        ;
        mov eax, [_currentTask] cmp
        eax, 0
        jz interrupt_return

        ;
        ; <実際のISRコードはこちら
        ;
    }
}

```

```

interrupt_returnです。
ポバドア
イレド
}
}

```

最後に、PITハードウェアが**scheduler_isr**を呼び出しているため、PITドライバのISRは決して呼び出されていないことを覚えておく必要があります。私たちは**割り込みを連鎖させたい**のです。つまり、先にインストールされていた古いISRがあれば、それを実行するチャンスを与えたいのです。これは、そのISRに**ジャンプする（呼び出さない）**ことで行われます。別のISRを呼び出すときには、そのISRが別の割り込みを連鎖させるか、**連鎖を断ち切るためにEOI（End-Of-Interrupt）** コマンドを発行することを念頭に置く必要があります。別のISRを呼び出す場合、技術的にはまだ割り込みを処理しているので、EOIを送信する必要はなく、IRETDも必要ありません。しかし、別のISRを呼び出さず、元のプロセスに制御を戻す場合は、両方が必要です。つまり、私たちのISRは次のようになります。

```

__declspec(naked) void _cdecl scheduler_isr () {
    _asm {
        ;
        ; 割り込みを解除してコンテキストを保存します。
        ;
        ; クリブ
        ; シュド
        ;
        ; 現在のタスクがない場合は、単にリターンします。
        ;
        mov eax, [_currentTask] cmp
        eax, 0
        jz interrupt_return

        ;
        ; <実際のISRコードはこちら
        ;
    }

    interrupt_returnで
    ;
    ; 古いISRを呼び出す必要があるかどうかをテストします。
    ;
    mov eax, old_isr
    cmp eax, 0
    jne chain_interrupt
    ;
    ; old_isrがnullの場合、EOIを送信して戻る。
    ;
    mov al, 0x20
    out 0x20, al
    popad
    iretd
    ;
    ; old_isrが有効であれば、それにジャンプする。これは

```



```
}
```

```
/* 時計の目盛りごとに呼び出されます。*/
```

```
void scheduler_tick () {
    /* ディスパッチャーを実行する
    だけです。*/ dispatch();
}
```

以上が、その内容です。上記は**ラウンドロビン方式**のスケジューリングを実装しており、一定の**量子**が経過するとタスクを入れ替えます。タスクは先ほど実装した**Ready Queue**に格納されます。これで残るのは、タスクの作成です。

上記は複数のスレッドに対して動作しますが、異なるプロセスに属するスレッドに対しては動作しません。典型的な解決策は、現在のスレッドの親プロセスと新しいスレッドを比較することです。両者が同じプロセスに属していれば、ディスパッチャは単にリターンすることができます。両者が異なるプロセスに属している場合、ディスパッチャは **VMM** を呼び出して新しいプロセスのアドレス空間に切り替える必要があります。サンプルコードをシンプルにするために、この章ではこれを避けることにしました。しかし、次の章でアドレス空間の管理について詳しく説明する際には、これをサポートする予定です。

タスク作成

例えば、**スケジュール**関数が **_currentTask** ポインタを別のタスクに更新したとします。そこで、この関数がISRに戻ると、ISRはIRETDを発行する前に、この新しいタスクからスタックとレジスタのコンテキストを設定します。これはうまくいきますが、タスクがすでにスタックとレジスタ・コンテキストをスタック上に持っている場合に限りです。

そのため、最初にタスクを作成する際に設定する必要があります。そこで、基本的なスタックフレームを設定し、タスクの**esp**と**eip**をスタックと**エン트리**ポイントの関数に設定します。スタックフレームは、私たちのISRが期待しているものでなければなりません。ISRに戻ると、まずPOP GS、POP FS、POP ES、POP DSを行い、次にPUSHAを行い、続いてIRETDを行います。PUSHAはEAX, EBX, ECX, EDX, ESI, EDI, ESP, EBPをポップします。そしてIRETDはEIP、CS、FLAGSをポップします。つまり、これがタスクが生成されたときの初期スタックフレームになるはずですが。

```
typedef struct _stackFrame {
    uint32_t gs;
    uint32_t fs;
    uint32_t es;
    uint32_t ds;
    uint32_t eax;
    uint32_t ebx;
    uint32_t ecx;
    uint32_t edx;
    uint32_t esi;
    uint32_t edi;
    uint32_t esp;
    uint32_t ebp;
    uint32_t eip;
    uint32_t cs;
    uint32_t flags.
}stackFrame;

タスク task_create (uint32_t entry, uint32_t esp) { ス
    レッド t;
    stackFrame* frame = ((stackFrame*) esp);
    frame->flags = 0x202;
        FRAME->CS= 8;
    フレーム
        ->eip=
    (uint32_t)entry; フレーム ->ebp=
    0;
        frame->esp= 0;
        frame->edi= 0;
        FRAME->ESI= 0;
        フレーム->edx= 0;
        フレーム->ecx= 0;
        フレーム->ebx= 0;
        フレーム
        フレーム
    ム->eax= 0;
    ム->ds= 0x10;
        フレーム->es= 0x10;
        フレーム->fs= 0x10;
    frame
        ->gs=0x10;
    t.esp = (uint32_t) frame;
    t.ss = 0x10;
    を返します。
}
```

これはほとんどのタスクで動作しますが、1つだけ例外があります - **初期タスク**です。今回作成したISRは、現在実行中のコードがタスク内にある場合にのみ動作します。これも鶏と卵の問題です。これを回避するためには、特別なタスクオブジェクトを作成し、マルチタスクを開始する準備ができたならそれを実行する必要があります。

```
static thread _idleTask;
void task_execute(thread t) {...
    _asm{
        mov esp, t.esp
        pop gs
        ポップfs
        ポップes
        ポップds
        ポップad
        iretd
    }
}

スケジューラを初期化します。*/
void scheduler_initialize(void) {...

    レディ・キューをクリアします
    。*/ clear_queue();

    プロセス・リストをクリアしま
    す。*/ init_process_list();

    /* アイドル・スレッドを作成して追加します。*/
    _idleThread = thread_create(idle_task, (uint32_t) create_kernel_stack(), true);

    /* 現在のスレッドをアイドル・タスクに設定して追加する。*/
    _currentThreadLocal = _idleThread;
    _currentTask= &_amp;currentThreadLocal;
```

```

queue_insert(_idleThread);

/* register isr */
old_isr = getvect(32);
setvect (32, scheduler_isr, 0x80);
}

/* アイドルタスク*/
void idle_task() {
    while(1) _asm pause;
}

```

上記はすべてをまとめています。アイドルタスクを作成し、キューに追加し、ISRをインストールして、初期タスクを実行します。初期タスクが実行されると、PITが起動するたびにISRが呼び出され、必要に応じてスケジューラを呼び出して現在のタスクを更新します。

8. MPの紹介

ここでは、他のプロセッサを起動するための標準的なインターフェイスと**IPI (Inter-Processor Interrupt)**を提供するために設計された**MP (Multi-Processor) 仕様**について、ごく簡単に紹介します。これは、コンカレント・プログラミングの難易度を一気に高めることができるため、上級者向けのトピックだと考えています。当社のスケジューラは一度に1つのタスクしか実行しませんが、MPでは独立したCPUのスケジューリングを行う低レベルのスケジューラを実装することができ、複数のタスクを同時に実行することができます。MP規格の詳細を知りたい方は、MP仕様書をご覧くださいになることをお勧めします。お使いのシステムが、MPで使用されるIOAPIC、LAPIC、ICIをすでにサポートしている必要があります。

マルチプロセッシングには**対称型マルチプロセッシング (SMP)**と**非対称型マルチプロセッシング (ASP)**がある。SMPではすべてのプロセッサが同じ種類であるのに対し、ASPでは同じ種類ではない。デスクトップシステムではASP方式は非常に珍しいため、ほとんどのシステムはSMPのみをサポートしている。しかし、MP規格はどちらにも対応しており、より多様なマシンタイプに対応できるように拡張性を持たせており、さらにOSが様々なタイプのシステムに対応できるようにしている。

システムが最初に起動するとき、ハードウェアは**Boot-Strap Processor (BSP)**を選択し、起動する唯一のプロセッサとして機能します。BSPは、最初に起動するプロセッサであり、最後にシャットダウンするプロセッサでなければなりません。オペレーティングシステムは、BSPから他の**アプリケーションプロセッサ (AP)**に**STARTUP IPI**を送信し、APを起動させることができます。他のAPは、BSPでも他のAPでも起動することができます。b>STARTUP IPI (およびINIT IPI)は、オペレーティングシステムが他のプロセッサを起こすために送るものです。

オペレーティングシステムは、システムがMPをサポートしているかどうかを検出するために、まず、フローティング**MPフローティングポインタ**構造体を検索する必要があります。この構造体には、**MPコンフィグレーションテーブルの物理アドレス**が含まれている。コンフィギュレーション・テーブルは**読み取り専用**である。このテーブルには、**ローカルAPIC (LAPC) のメモリマップドアドレス**、**プロセッサエントリ (プロセッサLAPIC IDを含む)**、**IOAPICエントリ (IOAPICベースのメモリマップドアドレスを含む)**、バス、および**割り込みコンフィギュレーションエントリ**が格納されています。オペレーティングシステムは、BSPのLAPIC IDを記憶して、BSPが最後にシャットダウンされるようにしなければなりません。

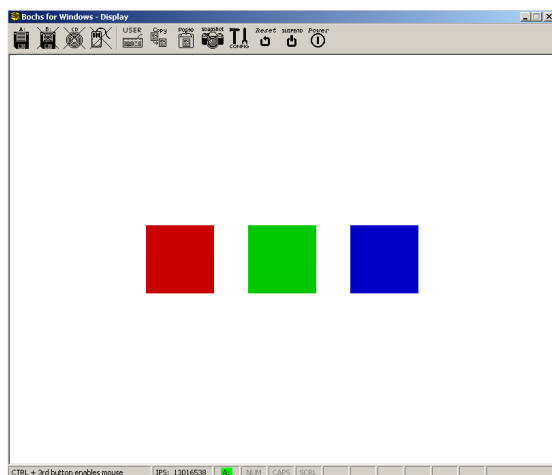
別のAPをウェイクするためには、BSP LAPICまたは別のAP LAPICを介して**INIT IPI**を送信すればよいのです。LAPICのメモリマップされたレジスタは、MPコンフィギュレーションテーブルのプロセッサ情報に格納されています。その後、そのAPに**STARTUP IPI**を送り、実行を開始する必要があります。本当にそれだけのことなのです。INIT IPIは、APをリセットさせます。STARTUP IPIは、リアルモードで指示された場所で実行を開始させます。オペレーティングシステムは、BSPで行ったように、APIをプロテクトモードやロングモードで構成するためのリアルモードスタブルーチンを提供する必要があります。

以上、マルチプロセッサ・システムの簡単な紹介をしました。他のプロセッサ（またはプロセッサコア）を起動するのは非常に簡単で、スケジューラを実装した後にSMPを試してみることをお勧めします。MPについては、APICを使った後のチュートリアルで詳しく説明する予定です。MPに興味のある方のために、その概要と方向性をお伝えしたいと思います。

9. デモ

[\[デモをダウンロードする\]](#)をクリックしてください。

新しいコードのほとんどは上記の文章で説明されており、私たちは最初のリリースを準備しているだけです。ストレステストと最終的な統合で問題が発生しなければ、デモは来週か再来週中にリリースされる予定です。



3つのタスクを実行する800x600x32モードのデモ。

このデモは、私たちにとって初めての本格的なグラフィカル・デモです。このデモでは、3つのタスクを同時に実行し、それぞれのタスクが実行中にビデオメモリの色を選択して循環することで、実行中であることを視覚的に示しています。テキストベースのデモではなく、グラフィカルなデモを選んだのは、次のような理由からです。

グラフィックスシリーズを読んでいなくても大丈夫です。グラフィックスシリーズに目を通していなくても、ここで説明します。

Bochs Graphics Adapter (BGA) について

この章の主要なテーマに集中できるように、コードをできるだけシンプルにするため、システムがISA用に構成されていることを前提にBGAを使用することにしました。このコードはBochs固有のもので、実際のシステムでは動作しません。実際のシステムでは、PCIバスのインフラをスキップする必要がありますが、これはより高度な章のトピックになるかもしれません。

```
#define VBE_DISPI_IOPORT_INDEX      0x01CE
#define VBE_DISPI_IOPORT_DATA      0x01CF
#define VBE_DISPI_INDEX_XRES       0x1
#define VBE_DISPI_INDEX_YRES       0x2
#define VBE_DISPI_INDEX_BPP        0x3
#define VBE_DISPI_INDEX_ENABLE     0x4
#define VBE_DISPI_DISABLED         0x00
#define VBE_DISPI_ENABLED          0x01
#define VBE_DISPI_LFB_ENABLED      0x40

void VbeBochsWrite(uint16_t index, uint16_t value) {
    outportw (VBE_DISPI_IOPORT_INDEX, index); outportw
    (VBE_DISPI_IOPORT_DATA, value);
}

void VbeBochsSetMode (uint16_t xres, uint16_t yres, uint16_t bpp) {
    VbeBochsWrite (VBE_DISPI_INDEX_ENABLE, VBE_DISPI_DISABLED);
    VbeBochsWrite (VBE_DISPI_INDEX_XRES, xres);
    VbeBochsWrite (VBE_DISPI_INDEX_YRES, yres);
    VbeBochsWrite (VBE_DISPI_INDEX_BPP, bpp) とな
    ります。
    VbeBochsWrite (VBE_DISPI_INDEX_ENABLE, VBE_DISPI_ENABLED | VBE_DISPI_LFB_ENABLED);
}
```

ビデオモードの設定は、**VbeBochsSetMode**を呼び出すだけです。この例では、**800x600x32**がよくサポートされているようなので、**800x600x32**を使用しています。ISAのリニアフレームバッファ(LFB)は、あらかじめ定義された**0xe0000000**の位置にあります。しかし、ここではページングを有効にしているので、LFBを使用するためには、仮想アドレス空間にマッピングする必要があります。今回のデモでは、**0x200000**仮想アドレスにマッピングします。このマッピングは、LFBのサイズをページ数で計算し、VMMを呼び出して各ページをマッピングします。

```
void* VbeBochsMapLFB () {...}

BGAのLFBはISAシステムではLFB_PHYSICALになります。*/
#define LFB_PHYSICAL 0xE0000000
#define LFB_VIRTUAL 0x200000

LFBを現在のプロセスのアドレス空間にマッピングする。*/
int pfcount = WIDTH*HEIGHT*BYTES_PER_PIXEL/4096; int
c;
for (c = 0; c <= pfcount; c++)
    vmmngr_mapPhysicalAddress (vmmngr_get_directory(), LFB_VIRTUAL + c * 0x1000, LFB_PHYSICAL + c * 0x1000, 3) を実行します。

LFBへのポインタを返します。*/
return (void*) LFB_VIRTUAL;
}
```

上記の関数により、**0x200000**に書き込むことで、**LFB**に描画することができます。ディスプレイ上のゴミを掃除するために、次にそれをクリアします。大量のピクセルを描画する必要があるので、32ビットモード用に関数を最適化するようにしています。この関数は画面を白にします。

```
void fillScreen32 () {
    uint32_t* lfb = (uint32_t*) LFB_VIRTUAL;
    for (uint32_t c=0; c<WIDTH*HEIGHT; c++)
        lfb[c] = 0xffffffffです。
}
```

32 Bits Per Pixelモードでは、ピクセルカラーは、赤8ビット、緑8ビット、青8ビットで構成されます。上位8ビットは今回の目的では無視されますが、通常は透明度の値として使用されます。ここでは3つのタスクを使って、3つの長方形をレンダリングし、3つの色の強度を循環させています。ここでは、ディスプレイ上の異なる場所にレンダリングするので、同時実行性の問題を心配する必要はありません。ディスプレイのメモリは共有されていますが、各タスクは別々の部分にレンダリングします。

```
void rect32 (int x, int y, int w, int h, int col) {
    uint32_t* lfb = (uint32_t*) LFB_VIRTUAL;
    for (uint32_t k = 0; k < h; k++) for
        (uint32_t j = 0; j < w; j++)
            lfb[(j+x) + (k+y) * WIDTH] = col;
}
```

<pre>void kthread_1 () { int col = 0; bool dir = true; while(1) {...} rect32(200, 250, 100, 100, col << 16)。 if (dir){ if (col++ == 0xfe) dir=false; }else if (col-- == 1) dir=true; } }</pre>	<pre>void kthread_2 () { int col = 0; bool dir = true; while(1) {...} rect32(350, 250, 100, 100, col << 8)。 if (dir){ if (col++ == 0xfe) dir=false; }else if (col-- == 1) dir=true; } }</pre>	<pre>void kthread_3 () { int col = 0; bool dir = true; while(1) {...} rect32(500, 250, 100, 100, col); if (dir) { if (col++ == 0xfe) dir=false; }else if (col-- == 1) dir=true; } }</pre>
--	---	---

スレッドスタック

通常、スレッドは2つの独立したスタックを持っています。ユーザーモードで実行するときのスタックと、カーネルモードで実行するときのスタックです。スレッドがユーザーモードで実行されているとき、CPUは**タスクステートセグメント (TSS)** の**esp0**および**ss0**フィールドを取得することで、カーネルスタックに切り替えることを思い出してください。スケジューラは、TSSを新しいスレッドのカーネルモードスタックに更新する役割を担っています。ただし、第25章については、すべてのスレッドがカーネル空間で実行されるため、TSSが参照されることはありません。言い換えれば、**第25章のスレッドは、カーネルモードスタックという1つのスタックしか持ちません。**

次の2つの章でアドレス空間の管理について説明しますが、その中でユーザーモードのスレッドをサポートする予定です。将来のアドレス空間アロケータを使用して、各ユーザーモードスレッドのためにユーザー空間にスタック空間を確保します。これは、**スレッドがユーザーモードとカーネルモードの両方のスタックを持つ**ことを意味します。

スレッドは、**CPL (Current Privilege Level)** が0のコードを実行する際に、カーネルモードスタックを使用します。CPLが**TSS**からの**RPL (Requested Privilege Level)** よりも小さい場合、CPUは自動的にこれをロードします。つまり、ユーザーモードのスレッドが実行されていて、PITが発火したとします。すると、CPUは**SS=TSS.ss0**、**ESP=TSS.esp0**を設定します。そして、**リターンCSとIPをこの新しいスタックにプッシュし、ISRを呼び出します**。ISRが終了すると、**IRET**を実行してユーザーモードのコードとスタックに戻ります。

このため、ユーザーレベルのスレッドは、最低でも**2つの**独立したスタックを持たなければなりません。最初のスタックはカーネル空間にマップされていなければならない、もう一つのスタックはユーザー空間にマップされていなければならない、プログラムが実行中にアクセスできなければなりません。カーネルレベルのスレッドは**1つの**スタックしか必要ありません。

アドレス空間アロケータがないので、まだユーザーモードのスタックをうまく割り当てることができず、（ハックしないと）ユーザーレベルのスレッドをサポートできません。また、まだ適切なカーネルモードアロケータがないので、カーネルレベルのスタックの割り当てもうまくサポートできません。これらは次の章か2章のトピックになります。

そこで、第25章では、カーネルメモリに領域を確保し、**4k**ブロックごとにスタックを割り当てることにしました。

```
void* create_kernel_stack() {
    ph
    ysical_addrp;
    virtual_addrlocation.
    void*ret;

    4kのカーネルスタック用にこの領域を確保している。*/ #define
    KERNEL_STACK_ALLOC_BASE 0xe0000000

    スタック用に4kフレームを確保する。*/ p =
    (physical_addr) pmmngr_alloc_block(); if
    (!p) return 0;

    /* 次の空き4kメモリブロック*/
    location = KERNEL_STACK_ALLOC_BASE + _kernel_stack_index * PAGE_SIZE;

    カーネル空間にマッピングします。*/
    vmmngr_mapPhysicalAddress (vmmngr_get_directory(), location, p, 3);

    スタックの一番上を返します。*/ ret =
    (void*) (location + PAGE_SIZE);

    /* 再び呼ばれたら次の4kを確保する準備をする。*/
    _kernel_stack_index++;

    /* そして、スタックの先頭を返す。
    */ return ret;
}
```

Back to Sleep())

フロッピーディスクの読み取り動作を遅らせるために、非常に基本的な**スリープ**機能を実装したことを思い出してください。この実装では、単に**ビジー・ループ**に入って時間を浪費していました。今度はそれをスレッドシステムに採用してみましょう。

基本的な考え方は、**sleep**は関数を呼び出したスレッドを**一時停止**させることです。つまり、現在のスレッドの状態を**READY**から**BLOCK**に調整し、タスクスイッチを強制的に行う必要があります。スケジューラはブロックされたスレッドを追跡し、適切に処理する必要があります。これは通常、他のオペレーティングシステム・コンポーネントからの**シグナル**を介して行われます。例えば、あるスレッドがデバイスの準備が整うのを待っている場合、そのスレッドはブロックされることがあります。ここでシステムは、そのスレッドがドライバからシグナルを受け取るまで待つ必要があります。それまでは、スケジューラは他のスレッドの実行に移るべきです。デモを比較的シンプルにするために、私たちは少し違ったやり方を選びました。

必要なのは、現在実行中のプログラムの状態を変更し、タスクスイッチを強制的に行うことです（**int 33**を介して**ISR**を直接呼び出すことで）。スケジューラには、ブロックされたスレッドをチェックしながら、実行する新しいスレッドを選択するためのロジックコードが含まれます。次のスレッドがブロックされている場合は、そのスリープ時間のデルタをデクリメントし、スリープ時間のデルタがゼロになった時点でスレッドを目覚めさせます。

このデモでは**sleep**を使用していませんが、ディスクドライバのコードは**sleep**に依存しています。これで、ディスク・デバイスからの読み取りを試みるスレッドは、適切にスリープできるようになりました。

メインプログラム

最後に、メインプログラムを見てみましょう。第25章のデモでは、スタックをカーネル空間に移動し、ブートローダから渡されたブートパラメータブロックの静的なコピーを作成した後、スタックを再調整しました。その後、上述のサービスを使用して、ビデオモードの設定、スケジューラの初期化、3つのスレッドの作成とレディキューへの追加を行いました。スレッドはカーネル空間で動作するため、カーネルスタックのみが割り当てられており、**create_kernel_stack**を呼び出して割り当てます。

また、第25章で作成したスレッドシステムと互換性を持たせるために、第24章のプロセス生成・管理コードを完全に書き換えました。ただし、次の章で行うユーザーモードスタックの割り当てをサポートするまでは完成しません。

```
void _cdecl kmain (multiboot_info* bootinfo) {...
    カーネルサイズを保存し、ブートインフォをコピーします。*/
    _asm movword ptr [kernelSize], dx
    memcpy(&bootinfo, bootinfo, sizeof(multiboot_info));

    /* スタックの調整*/
    _asm lea esp, dword ptr [_kernel_stack+8096]
    init (&bootinfo);

    ビデオモードの設定とフレームバッファのマッピングを行う。*/
    VbeBochsSetMode(WIDTH, HEIGHT, BPP);
    VbeBochsMapLFB();
}
```

`fillScreen32 ()` です。


```

スケジューラを初期化します。
*/ scheduler_initialize ();

カーネル・スレッドを作成します。*/
queue_insert (thread_create(kthread_1, (uint32_t) create_kernel_stack(), true));
queue_insert (thread_create(kthread_2, (uint32_t) create_kernel_stack(), true));
queue_insert (thread_create(kthread_3, (uint32_t) create_kernel_stack(), true));

/* アイドル・スレッドを実行し
ます。*/ execute_idle ();

/* これは決して実行されてはいけません。
*/ for (;;) _asm {cli
hit]となります。
}

```

10. 結論

この章では、スケジューリング・アルゴリズム、SMPの概要、コンカレント・プログラミング、そしてプリエンプティブなラウンドロビン・スケジューラの実装について説明しました。また、BDA(Bochs Graphics Adapter)を使った高解像度ビデオモードの紹介や、状態管理、IPC技術の紹介も行いました。

次の章では、いよいよカーネルやユーザモードのアロケータ、アドレス空間の割り当て、ページスワッピング、ページフォルトの処理などのメモリ割り当てアルゴリズムについて説明します。この章では、フリーリストとスタックアロケータ、SLABアロケータ（およびその変種）、ZoneとArenaアロケータ、Buddyアロケータ、ユーザ空間の管理、再帰的ページディレクトリ、ページファイルとスワップ空間、およびその他のトピックを取り上げます。この章の内容を発展させて、ユーザーモードのプロセスローディングをサポートしていきます。膨大な量の資料が出てくるため、この章は1つまたは2つの別々の章になるかもしれません。

次の機会まで。

～Mike () です。

OS開発シリーズ編集部

[ホーム](#)



オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - Portable Executable (PE) by Mike, 2011

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

Welcome!

ようし、これは長くなりそうだ。

この章では、上級者向けのトピックであるPE実行ファイル形式を取り上げます。PEリソース、ダイナミックリンクなどのカバーを見ていきます。また、この章は、情報を可能な限り完全なものにするために、より多くの情報を含むアップデートを予定しています。

本章で紹介する内容のほとんどは情報提供のみを目的としており、完全性を保つためと、読者がサポートを提供したいと思う場合にのみ掲載しています。また、これらの情報の多くは、公式のPE仕様書にも記載されていますのでご注意ください。

この章が終われば、ローダーを開発してシングルタスク環境をサポートするために必要なものがすべて揃うことになります。

さあ、始めましょう。

ファイル形式

アブストラクト

PE (Portable Executable) ファイル形式は、WindowsやReactOSなどのWindowsライクなOSで使用されている標準的な実行ファイル形式です。また、**EFI (Extensible Firmware Interface)** マシンでの起動に使用される標準的なファイル形式でもあります。

PEの実行ファイルフォーマットは、再配置、シンボルテーブル、リソース、ダイナミックバインディングなどをサポートする複雑なフォーマットです。

条件

VA (バーチャルアドレス)

仮想アドレス (VA) は、現在のプログラムの**仮想アドレス空間 (VAS)** 内の線形アドレスです。PEの実行形式では、すべてのアドレスが仮想アドレスです。これらのアドレスは32ビットのリニアアドレスです。

RVA (相対的仮想アドレス)

RVA (Relative Virtual Address) とは、実行プログラムの**ベースアドレス**からの相対的なVAのことです。PEの実行形式では多くの部分でRVAが使用されていますので、RVAとは何か、そしてRVAからリニアアドレスを得る方法を知っておくことが重要です。RVAはベースアドレスからの単なるオフセットでしかありません。ですから、リニアアドレスを得るには、ベースにRVAを加えるだけです。

リニアアドレス=ベースアドレス+RVA

解析時に多くの箇所での計算を行う必要があるため、この点は重要です。

セクションとセクションテーブル

セクション

高度な実行ファイルフォーマットでは、リンクプロセスを簡素化し、ソフトウェアに構造を与えるために、**プログラムセクション**を使用するのが一般的です。セクションは、実行イメージやオブジェクトファイルに格納される命令やデータの標準的な方法を提供することで、リンクプロセスを簡素化します。

セクションには通常、その中にどのような要素があるかに関連した名前があります。例えば、「**.data**」は、変数や初期化されていないデータを含む一般的なセクション名です。その他のセクション名には、歴史的な背景があります。例えば、**.text**は、実行可能なコードやオブジェクトコードを含むセクションの典型的な名前です。**.bss**は、一般的にグローバルな、プログラム全体で初期化されたデータに使用されます。

C++ツールチェインを例にとると、グローバルな名前空間や**静的**に定義された変数は**.bss**に格納されます。また、コンパイル後に生成されるバイトコードは**.text**に格納されます。

PEの実行ファイル形式には、通常、以下のセクション名が1つ以上含まれています。

- .テキスト
- .データ
- .bss
- .アーチ
- .edata
- .idata
- .pdata
- .rdata
- .reloc
- .rsrc
- .sbss
- .sdata
- .srdata
- .xdata

セクションテーブル

プログラムファイルやオブジェクトファイルには、複数のセクションが含まれています。各セクションの基本位置とセクション名は、通常、**セクションテーブル**に格納されている。セクションテーブルは、単純な構造体のリンクリストであったり、ハッシュテーブルであったり、さまざまな形式があります。

シンボルマーク

C++でプログラミングをしていると、有名な「**未定義シンボル**」のリンカーエラー（Cの実装によっては警告）に遭遇することが多いと思います。その通り、完全にエラーなしでコンパイルとリンクを行うことができます。）これは、関数を呼び出したり、変数を参照する際に、リンクの段階で定義が解決できなかったために起こります。

関数や変数は、リンカーによって**シンボルと呼ばれます**。シンボルには、名前のほか、データ型や値などの情報が含まれています。コンパイル時には、最終的なプログラムをリンクできるようにするために、これらのシンボルを追跡する必要があります。もし、現在の**翻訳ユニット**で定義されていないシンボルが使用されていて、それが**EXTERN**シンボルである場合、コンパイル担当者はシンボルをオブジェクト・ファイルに書き込む際に、**EXTERN**シンボルとしてマークする必要があります。

リンクの段階で、**EXTERN**とマークされたシンボルがまだ値を持っていない（シンボルが定義されていない）場合、リンカーは上記のエラーを発行します。

シンボルは、モジュール、翻訳ユニット、またはライブラリ間で変数や関数を定義するためのものです。

プログラム全体、およびプログラムがリンクしているライブラリ全体で、同じ名前のシンボルは**1つ**しかありません。このため、高級言語を使用すると名前が衝突する可能性が高いため、変数名や関数名には通常、**名前のマングリング**が適用されます。もちろん、これはアセンブリ言語には適用されません。適用される名前のマングリングは複数の要因に依存し、ツールチェーンによって異なります。

ちょっと見てみましょう。ここでは、いくつかの**C**関数の宣言と、その右にマングルされたシンボル名を示しています。マングルされた名前の中の数字は、パラメタのバイト数です。

```
void _cdecl function (int i); ->
function void _stdcall function(int i); ->
function@4 void _fastcall function(int i); ->
@function@4
```

また、**_cdecl**の呼び出し規則を持つ関数には、前にアンダースコアが付いているだけです。これにより、アセンブリ言語を使ってC言語の関数を簡単に定義することができ、Cコードでその関数を簡単に呼び出すことができます。

C++の名前の付け方には標準がありません。あるコンパイラでは**?h@@YAXH@Z**のようなシンボリックな名前を生成しますが、他のコンパイラでは**7h FiやW?h\$h(n)v**は、**void h(int)**という**同じ機能**のためのものです。このため、アセンブリ言語での使用は現実的ではありません。しかし、それでも可能である。

記号表

セクションテーブルと同じように、**シンボルテーブルがあります**。シンボルテーブルは、ソフトウェアがシンボル名とシンボルに関する情報（エクスポートされたシンボルであるかどうか、データタイプ、プロパティなど）を検索するための手段です。シンボルテーブルは通常、情報のリンクリスト、またはハッシュテーブルで実装されています。

構造

アブストラクト

MSVC++の章で、PE実行形式の構造を見てきました。PE実行ファイルをメモリにロードすると、そのメモリには、ロードしたファイルの正確なコピーが格納されます。つまり、PEファイルフォーマットの最初の構造内の最初のバイトは、実際にファイルがメモリにロードされた場所からの最初のバイトに位置しています。

例えば、PEファイルを**1MB**にロードした場合、インメモリのフットプリントは以下ようになります。



上の画像は、MSVCの章を読んだことのある読者には見覚えがあるはずですが。上の図を見ると、PEファイルが**1MB**まで読み込まれていた場合、ディスク上の最初の構造体である**IMAGE_DOS_HEADER**がメモリ上のその位置から始まり、その後にはファイル内の残りの構造体（パディングを含む）が続きます。

上の画像も単純化しすぎていて、決してPEファイルフォーマットの全体像を示しているわけではありません。PEファイルフォーマットの構造はかなり大きく、多くの構造体やテーブルで構成されています。

ここでは、完全なフォーマットをご紹介します。

- 1. **IMAGE_DOS_HEADER**構造体（重要）
- 2. STUBプログラム
- 3. **IMAGE_FILE_HEADER**構造体 [COFFヘッダ]（重要）
- 4. **IMAGE_OPTIONAL_HEADER**構造体（重要）
- 5. セグメントテーブル
- 6. リソーステーブル
- 7. 居住者名テーブル
- 8. モジュール・リファレンス・テーブル
- 9. インポートネーム表
- 10. エントリーテーブル
- 11. 非居住者名表
- 12. セグメント
 - 1. データ
 - 2. 情報

上の表は、ファイル形式の最初から最後までを示しています。**重要**と書かれた項目は、プログラムを実行するために解析方法を知る必要があります。その他の情報は、情報提供のみを目的としています。**IMAGE_OPTIONAL_HEADER** 構造体で重要なメンバは、エントリポイントのアドレスを含むメンバと、イメージベースアドレスだけです。

このファイルの各セクションの解析については、次のセクションで詳しく説明します。また、テーブルやディレクトリを解析する際に使われる他の構造についても同様に紹介します。

IMAGE_DOS_HEADER 構造体

IMAGE_DOS_HEADERは、PEファイルの最初の構造体です。この構造体には、プログラムファイルとそのロード方法に関するグローバルな情報が含まれています。この構造体に含まれる情報のほとんどは、DOSソフトウェアに関連するものであり、後方互換性のためにのみサポートされています。

構造はフォーマットに沿っています。

```
typedef struct IMAGE_DOS_HEADER { // DOS .EXEヘッダー
    uint16_t e_magic;           // "MZ"を含むこと
    uint16_t e_cblp;           // ファイルの最終ページのバイト数
    uint16_t e_cp;             // ファイルのページ数
    uint16_t e_crlc;           // リロケート
    uint16_t e_cparhdr;         // ヘッダーのサイズ (段落数)
    uint16_t e_minalloc;        // 割り当てる最小段落と最大段落
    uint16_t e_ss;              // ローダーが設定する初期SS:SP
    uint16_t e_sp;              // チェックサム
    uint16_t e_csum;            // 初期のCS:IP
    uint16_t e_ip;              // リロケーション・テーブルのアドレス
    uint16_t e_cs;              // リロケーション・テーブルのアドレス
    uint16_t e_lfarlc;          // オーバーレイ数
    uint16_t e_ovno;            // リセプション
    uint16_t e_res[4];          // OEM ID
    uint16_t e_oemid;           // OEM情報
    uint16_t e_oeminfo;         // OEM情報
    uint16_t e_res2[10];        // 予約
} image_dos_header; // 新しいEXEヘッダのアドレス
```

さてさて、この構造には面白いものがたくさんあります。初期CS:IPと初期SS:SPのメンバーは、OSが通常CSのためにスタックスペースとコード記述子の値を割り当てているので、無視してください。これらのメンバーは、DOS領域やv8086モードを必要とするソフトウェアの時に顕著でした。

STUBプログラム

さて、それでは、**IMAGE_DOS_HEADER**構造体の直後にDOSスタブプログラムがあることに注目してください。これ、実は便利なプログラムなんです。これは、DOSの中からWindowsのプログラムを実行しようとする、「このプログラムはDOSモードでは実行できません」と表示するプログラムです。

リンクオプションの**/STUB**を使ってスタブプログラムを変更することができます。

```
/stub=myprog.exe
```

DOSが実行ファイルをロードしようとする、DOSは**IMAGE_DOS_HEADER**構造体を解析し、有効なDOSプログラムであるため、DOSスタブプログラムを実行しようとします。Win32サブシステムで実行する場合、Windowsローダはスタブプログラムを無視します。

image_nt_headers

STUBプログラムの後には、PEヘッダー構造体のフォーマットを含む、**IMAGE_NT_HEADERS**という構造体があります。以下がその構造体です。

```
typedef struct _IMAGE_NT_HEADERS
{
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER OptionalHeader;
} image_nt_headers; *pimage_nt_headers;
```

記号は「PE\0\0」と一致しなければなりません (\0はヌル文字)。**IMAGE_FILE_HEADER**には、ローダが使用する追加情報と、**IMAGE_OPTIONAL_HEADER** 構造体のサイズがすべて含まれています。**IMAGE_OPTIONAL_HEADER** は、ファイルの中で一番大きくて重要な構造体です。また、サイズは定義されていません。

この構造体の位置を特定するために、OSローダーは**IMAGE_DOS_HEADER**の**e_lfanew**メンバを使用する必要があります。**e_lfanew**はメモリ上のこの構造体へのRVAなので、この構造体の位置を特定するために、ローダーは次のように実行する必要があります。

これは、**imageBase**がプログラムのファイルのメモリに読み込まれた場所を参照していると仮定しています。DOSなどの古いOSでは、このヘッダーのメンバーを認識していないので、**IMAGE_NT_HEADERS**はDOSでは無視されます。**imageBase**は、**imageBase** (pFile->e_lfanew + imageBase);

この構造体は、他の2つのヘッダー構造体のフォーマットを含んでいます。では、最初の構造体を見てみましょう。

イメージファイルのヘッダー

IMAGE_FILE_HEADER は、**COFF (Common Object File Format)** のヘッダー構造です。以下のような形式になっています。

```
typedef struct _IMAGE_FILE_HEADER
{
    USHORT Machine;
    USHORT NumberOfSections; // セクションテーブルのセクション数
    ULONG TimeDateStamp;     // 番組リンクの日付と時刻
    mp;                      // シンボルテーブルのRVA
    ULONG PointerToSymbolTable; // テーブル内のシンボル数
    ULONG NumberOfSymbols;    // IMAGE_OPTIONAL_HEADER のサイズ (バイト)
    ; USHORT
    SizeOfOptionalHeader;     //
    USHORT Characteristics;
} image_file_header; *pimage_file_header;
```

この構造はあまり複雑ではありません。上記のほとんどは、デバッガにとってのみ有用です (シンボルテーブルの解析)。**SizeOfOptionalHeader** は重要です。**IMAGE_OPTIONAL_HEADER** はサイズが定義されていないので、このメンバで構造体のサイズを知ることができます。

Machineは以下の値のいずれかです。x86マシン

- 0x014c (x86マシン用)
- 0x0200 (x64マシン用)
- 0x8664 (AMD64マシン用)

通常のケースでは、**x86**アーキテクチャ用に開発しているため、**0x014c**となるはずですが、

特性は、リンカーがビットごとにORすることができるビットフラグで構成されており、ローダに実行イメージの種類異なる特性を知らせることができます。以下はそのフォーマットです。

- **ビット0** : セットされている場合、画像は再配置情報を持たない。
- **ビット1** : セットされている場合、ファイルは実行可能です。
- **ビット2** : セットされている場合、画像には**COFF**ライン番号がありません。
- **ビット3** : セットされている場合、画像には**COFF**シンボルテーブルのエントリがありません。
- **ビット4** : セットされている場合、イメージの作業セットをトリムする。(Windowsのメモリ管理に依存します。廃止されました)
- **ビット5** : セットされている場合、ローダーは実行ファイルが**2GB**以上の**VA**を扱えると仮定します。
- **ビット6** : セットされている場合、ローダーはイメージが**32**ビットワードをサポートしていると仮定します。
- **ビット7** : セットされている場合、画像はデバッグ情報を持たない
- **Bit 8** : 設定されている場合、イメージはネットワークドライブから直接実行されません (Windows特有)。
- **ビット9** : 設定されている場合、画像は**SYSTEM**ファイルとして扱われます。
- **ビット10** : セットされている場合、イメージは**DLL**ファイルとして扱われる
- **ビット11** : セットされている場合、イメージはシングルプロセッサのマシンでのみ実行されます。
- **ビット12** : セットされている場合、ビッグエンディアン。 **obsolete**フラグ

Windowsのヘッダには、**IMAGE_FILE_RELOCS_STRIPPED**や**IMAGE_FILE_EXECUTABLE_IMAGE**などの定数が定義されており、これらのフラグを設定する際に使用することができます。

ご覧のように、この構造のほとんどは、画像をどのようにロードするかというローダーの情報のみのためのものです。でも、待ってください。リソース、シンボルテーブル、デバッグ情報などはどうなっているのでしょうか？**IMAGE_OPTIONAL_HEADER**にサイズが定義されていない理由が見えてきました。ちょっと見てみましょう。

image_optional_header

うわ、出てきた。これは、このファイルの中で最も複雑な構造です。良いニュースは、この構造を見たことがあるということです。

```
struct _IMAGE_OPTIONAL_HEADER
{
    USHORT   マジックですね。                // そうでない数字
    UCHAR    MajorLinkerVersion;            // リンカーバージョン
    UCHAR    MinorLinkerVersion;
    ULONG    SizeOfCodeです。                // .textのサイズ (バイト)
    ULONG    SizeOfInitializedDataです。     // .bss (およびその他) のサイズ (単位: バイト)
    ULONG    SizeOfUninitializedDataです。   // .data, .sdataなどのサイズ (バイト)
    ULONG    AddressOfEntryPointです。       // エントリーポイントのRVA
    ULONG    BaseOfCodeです。                // .テキストのベース
    ULONG    BaseOfDataです。                // .データのベース
    ULONG    イメージベースです。            // イメージベースVA
    ULONG    SectionAlignmentです。          // ファイルセクションアライメント
    ULONG    FileAlignmentです。             // ファイルアライメント
    USHORT   MajorOperatingSystemVersion;    // Windows専用です。イメージの実行に必要なOSバージョン
    USHORT   MinorOperatingSystemVersionの略
    USHORT   MajorImageVersion。            // プログラムのバージョン
    USHORT   MinorImageVersion。
    USHORT   MajorSubsystemVersion。        // Windows専用です。サブシステムのバージョン
    USHORT   MinorSubsystemVersion;
    ULONG    Reserved1;
    ULONG    SizeOfImageです。                // 画像のサイズ (バイト)
    ULONG    SizeOfHeadersです。             // ヘッダー (およびスタブプログラム) のサイズ (単位: バイト)
    ULONG    CheckSumです。                  // チェックサム
    USHORT   サブシステムです。              // サブシステムの種類
    USHORT   DllCharacteristics;            // DLLのプロパティ
    ULONG    SizeOfStackReserveです。         // スタックのサイズ、バイト単位
    ULONG    SizeOfStackCommitです。         // コミットするスタックのサイズ
    ULONG    SizeOfHeapReserveです。         // ヒープのサイズ、単位: バイト
    ULONG    SizeOfHeapCommitです。         // コミットするヒープのサイズ
    ULONG    LoaderFlagsです。               // 使われなくなった
    ULONG    NumberOfRvaAndSizesです。       // DataDirectoryのエントリ数
    IMAGE_DATA_DIRECTORY ディレクトリ [IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
};
// image optional header, *pimage optional header;
```

まず、最後のメンバーである**DataDirectory**を見てみましょう。定数である **IMAGE_NUMBEROF_DIRECTORY_ENTRIES** は、長年にわたって変更されてきました。これは、この構造体のサイズを変えることができるメンバーです。このメンバーについては、後ほど詳しく見てみましょう。

明らかにオプションではないのに、なぜ「オプション」ヘッダーと呼ばれているのか気になりますよね。これは、**COFF**オブジェクトファイルではオプションとなっているからです。実行イメージではオプションではありませんが、オブジェクトファイルではオプションです。)

マジックは以下のいずれかになります。

- **0x10b** : 32 ビットの実行形式
- **0x20b** : 64 ビットの実行形式
- **0x107** ROMイメージ

通常の場合は、**0x10b**になるはずですが。

この構造のメンバーの多くは、それほど複雑ではありません。

サブシステム・メンバーは、Windows固有のものです。プログラムを正しく実行するために必要なサブシステムを Windows に伝えます。以下の値のいずれかになります (ここでは完全性を期して掲載しています)。

- **0** : 不明
- **1** : ネイティブサブシステム
- **2** : GUIサブシステム
- **3** : CUIサブシステム
- **5** : OS/2 CUIサブシステム
- **7** : POSIX CUIサブシステム
- **9** : Windows CE GUIサブシステム

2021/11/15 13:12

ム 10 : EFI

- 11 : EFIブートドライバ
- 12 : EFIランタイムドラ
- イバ 13 : EFI ROM

オペレーティングシステム開発シリーズ

- 14: Xbox
- 16: ブートアプリケーション

DllCharacteristicsメンバは、ローダにDLLに関する情報を与えるビットフラグを含んでいます。以下のような形式になっています。

- **ビット0-3**: 予約済み
- **ビット4**: セットされている場合、DLLは再配置可能である
- **ビット5**: セットされている場合、強制的にコードインテグリティチェックを行う
- **ビット6**: セットされている場合、画像は**DEP (Data Execution Prevention)** に対応しています。
- **ビット7**: セットされている場合、画像を分離してはならない
- **ビット8**: セットされている場合、画像は**構造化された例外処理 (SEH)** を使用し
- ません **ビット9**: セットされている場合、画像はバインドされません
- **ビット10**: 予約
- **ビット11**: セットされている場合、イメージは**Windows Driver Model (WDM)** ドライバーである。
- **ビット12**: 予約
- **ビット13**: 画像がターミナルサーバーに対応している

AddressOfEntryPoint は重要なものです。このメンバには、イメージのエントリーポイント関数のRVAが含まれています（DLLにはエントリーポイントが必要ないので、これはNULLで構いません）。

それがすべてです。**.text**、**.data**、**.bss**などの他のメンバが何であるかに興味があるかもしれません。厄介な見た目のものもあります。まだ見していない**DataDirectory**のメンバ。

これらのメンバについては後ほど詳しく見ていきます。今は、プログラムの実行を見てみましょう。

プログラムの実行

この段階では、**プログラムを実行するだけであれば**、すべての情報が提供されています。プログラムをロードした後、ローダがすべきことは、オプションヘッダから**AddressOfEntryPoint**メンバを探し出し、そのアドレスを呼び出すことです。これは**RVA**であることを覚えておいてください。つまり、ローダはエントリーポイント関数へのリニアアドレスを得るために、このアドレスを**ImageBase**に追加する必要があります。

ここではその一例をご紹介します。

```
// ! loadedProgram は、イメージが読み込まれた場所です
IMAGE_DOS_HEADER* pImage = (IMAGE_DOS_HEADER*)
loadedProgram;

// ! NT HEADERSに移動します。
IMAGE_NT_HEADERS* pHeaders = (IMAGE_NT_HEADERS*)(loadedProgram + pImage->e_lfanew);

// ! オプションヘッダからイメージベースとエントリーポイントのアドレスを取
得 int base = pHeaders->OptionalHeader.ImageBase;
int entryPoint = pHeaders->OptionalHeader.AddressOfEntryPoint;

// ! エントリーポイントの関数はbase+entryPointにある
void (*entryFunction) () = (entryPoint +
base);

// ! プログラムのエントリーポイン
トであるentryFunction () を呼び
出す。
```

PEの実行ファイルの実行に必要なのは、これだけです :)

データディレクトリ

アブストラクト

リソース、シンボルテーブル、デバッグ情報、インポート、エクスポートテーブルなどは、オプションのヘッダにある**DataDirectory**メンバからアクセスできます。このメンバは、**IMAGE_DATA_DIRECTORY** の配列で、これらの情報を含む他の構造体にアクセスするために使用できます。**IMAGE_DATA_DIRECTORY** の形式があります。

```
DataDirectoryはIMAGE_DATA_DIRECTORYの配列であることを覚えておいてください。この配列の各エントリでは、アクセスしたいさまざまなデータに
アクセスするRVAとVirtualAddress;
// テーブルのRVA DWORD Size; // テーブル
インデックスのエントリーは
pimage_data_directory, *pimage_data_directory;
• 以下の通りです。0: 輸
出用ディレクトリ
• 1: インポートディレクトリ
• 2: リソースディレクトリ
• 3: 例外ディレクトリ
• 4: セキュリティディレクトリ
• 5: ベースリロケーション
• テーブル 6:デバッグディ
レクトリ
• 7: 説明文字列
• 8: マシン値 (MIPS GP) 9:
• TLSディレクトリ
• 10: Load configuration
• directory 14: COM+ data
directory
```

例えば、エクスポートテーブルを読み取る必要がある場合は、**DataDirectory[0]**を参照します。リソースを読みたい場合は**DataDirectory[2].VirtualAddress**:

それぞれのセクションには、特定のデータを解析するために必要な独自の構造があります。ここでは、その中でも特に便利なものをご紹介します。

エクスポートテーブルの読み込み

エクスポートテーブルには、ライブラリやDLLからエクスポートされたすべての関数が含まれており、そのDLL内の関数アドレス、名前、序列番号などが記載されています。Win32 API関数の**GetProcAddress()**は、モジュールのエクスポートテーブルを序列番号または名前で解析し、そこからアドレスを返すことで動作します。このように、エクスポートテーブルを読み取ることは有用な方法の一つです。

エクスポート・テーブルを解析するには、まずエクスポート・ディレクトリ構造を取得する必要があります。これは、**DataDirectory[0]**を取得することで行います。

```
PIMAGE_DATA_DIRECTORY DataDirectory = &OptionalHeader->DataDirectory [0];
PIMAGE_EXPORT_DIRECTORY exportDirectory = (PIMAGE_EXPORT_DIRECTORY) (DataDirectory->VirtualAddress + ImageBase);
```

IMAGE_DATA_DIRECTORY構造体の**VirtualAddress**はRVAなので、イメージベースに追加する必要があることを覚えておいてください。次に **exportDirectory** のポイントは、この素晴らしい構造にあります。

```
typedef struct _IMAGE_EXPORT_DIRECTORY
{
    uint32_t Characteristics;
    uint32_t TimeDateStamp;
    uint16_t MajorVersion;
    uint16_t MinorVersion;
    uint32_t Name;
    uint32_t Base;
    uint32_t NumberOfFunctions;
    uint32_t NumberOfNames;
    uint32_t** AddressOfFunctions;
    uint32_t** AddressOfNames;
    uint16_t** AddressOfNameOrdinal;
}image_export_directory,*pimage_export_directory;
```

これは簡単なことです。**AddressOfFunctions**は、関数アドレスの配列を指すRVAです。ただし、関数のアドレスもRVAです。**AddressOfNames**は、関数名のリストへのポインタです。しかし、これらのアドレスはすべてRVAなので、関数名とアドレスを正しく取得するためには、イメージベースに追加する必要があります。

AddressOfNameOrdinalは、序列のリストに対するRVAです。**ordinals**は、エクスポートされた機能を表す単なる数字であり、アドレスではないため、RVAではありません。

エクスポートテーブルを正しく解析するには、ループで行う必要があります。例えば、以下のようになります。

```
これを利用してGetProcAddress()を実装し、DLLの関数を呼び出すことができます。
DWORD DFunctionOrdinalAddressArray = (DWORD)ExportDirectory->AddressOfNameOrdinal + ((PBYTE)imageBase);
DWORD PFunctionNameAddressArray = (DWORD)ExportDirectory->AddressOfNames + ((PBYTE)imageBase);
インポートテーブルの読み込み
LPSTRFunctionName = FunctionNameAddressArray [i] +
```

輸出の表を読むだけでは不十分だったかも知れませんが、輸入の表を読むのはそれほど難しくありませんが、輸出の表よりも少し複雑です。OK、OK、輸入表を読むことに何か意味があるの？それは、**GetProcAddress()**がライブラリのインポートテーブルにエントリを書き込むことで、**GetProcAddress()**を呼び出すことなく、ライブラリやDLL間での関数呼び出しが可能になります。**Windows**では、遅延ロードされたDLLやシステムDLLでこれを行います。

インポートテーブルを読むためには、インポートディレクトリ構造を見つける必要があります。これは**DataDirectory[1]**にあります。

重要なのは、**ImportDirectory**が各モジュールの**FunctionName**と**Ordinal**を指しているアドレスで、これらのエントリはそれぞれ、インポートされたモジュール（インポートDLLなど）を表しています。これは、**GetProcAddress()**を呼び出すことで、**GetProcAddress()**を呼び出すことなく、ライブラリやDLL間での関数呼び出しが可能になります。**Windows**では、遅延ロードされたDLLやシステムDLLでこれを行います。

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR
{
    union {
        uint32_t 特徴 // 0はNULLインポートディスクリプターの終了を意味する
        uint32_t OriginalFirstThunkです。 // RVAからINT
    };
    uint32_t TimeDateStampです。 // モジュールの時間/日付、またはその他のプロパティ（以下参照）
    uint32_t ForwarderChainです。 // フォワーダチェーンID
    uint32_t 名前を教えてください。 // モジュール名
    uint32_t FirstThunkです。 // RVA→IAT（バインドされたIATが実際のアドレスを持っている場合）
}IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR *PIMAGE_IMPORT_DESCRIPTOR;
```

ここで重要なのは、**Name**、**OriginalFirstThunk**、**FirstThunk**がRVAであることです。つまり、データを適切に解析するためには、イメージベースにアドレス（これらはポインタです）を追加する必要があります。**Name**は、**kernel32.dll**のような、インポートされたモジュール名を指すRVAです。これはヌルで終端しています。

インポートディスクリプターの配列を扱っていることを覚えていますか？この配列にあるインポートディスクリプターの数を知るにはどうしたらいいのでしょうか？配列はNULLの**IMAGE_IMPORT_DESCRIPTOR**で終わっていますので、各エントリをループする簡単な方法は次のとおりです。

```
IMAGE_IMPORT_DESCRIPTOR* lpImportDesc;
while (! lpImportDesc->FirstThunk)
{
    ...

    // ! ここでlpImportDescと連携します。
```

```
lpImportDesc++; // 次のエントリに移動します。
```

TimeStampには、適切な時間/日付、または以下の値のいずれかを指定します。0: モ

- ジュールがバインドされていない
- -1: 画像が結合されている。実時間/日付スタンプの保存

ForwarderChainは、**DLLの前方参照**をサポートしている場合にのみ使用されます。これにより、DLL間の呼び出しを他のDLLに転送することができます。例えば、Windowsの**kernel32.dll**の一部の呼び出しは、他のDLLに転送されます。

FirstThunkはIATを指し、**OriginalFirstThunk**はインポートされたすべての関数を表す構造体の配列を指します。これが**Import Name Table (INT)**です。これらのメンバーはいずれもRVAです。

そういえば、もう一つの構造が出てきているのをご存知でしょうか。見てみましょう。

```
typedef struct _IMAGE_THUNK_DATA
{
    union {
        uint32_t* Function;           // インポートされた関数のアドレス
        uint32_t Ordinal;            // 関数の序列値
        PIMAGE_IMPORT_BY_NAME AddressOfData; // 輸入された名前のRVA
        DWORD ForwarderString1;      // RVAからフォワードへの文字列
    } ul;
}image_thunk_data, *pimage_thunk_data;
```

OriginalFirstThunkは、**IMAGE_THUNK_DATA**構造体の配列を指すRVAです。うわ、やった

、また構造体だ。この構造体は小さいものです。

```
typedef struct _IMAGE_IMPORT_BY_NAME
{
    uint16_t Hint; // 使用可能な序列番号
    uint8_t N;     // 関数の名前、NULL終端
}image_import_by_name, *pimage_import_by_name;
```

それがすべてです。第1パラメータは0でも構いませんが、これは関数がどのような序列番号を使用するかをロードに示唆するだけです。**Name**は、関数の名前を表す文字の配列です。

ここからが本題です。**IAT**は、機能を表すアドレスのリストに過ぎない。関数とは？この **IMAGE_THUNK_DATA** 配列内の関数です。**IMAGE_THUNK_DATA**構造体を見てみると、関数名を表すユニオンがあるだけです。これは、**インポートネームテーブル (INT)** です。

例えば、**IMAGE_THUNK_DATA[3]**に格納されている関数の現在のアドレスを取得したいとします。そのアドレスは、**IAT** の 3 番目の **dword** で、**IMAGE_IMPORT_DESCRIPTOR->FirstThunk** で読み取ることができます。

そこで、機能名とアドレスを取得してみましょう。

```
符号付き整数 (count) = 0
while (lpThunk->ul.Function) {

    //! 関数名の取得
    char* lpFunctionName = (char*)((uint8_t*)imageBase + (uint32_t)lpThunk->ul.AddressOfData.Name);

    //! IATに入ってこの関数のアドレスを取得する
    uint32_t* addr = (uint32_t*)((uint8_t*)imageBase + lpImportDesc->FirstThunk) + count;

    // lpFunctionNameがヌル終端の関数名を指すようになりました。
    // addrがこの関数のアドレスを指すようになる

    count++;
    lpThunk++;
}
```

イメージバイnding

ここからが面白いところです。**IAT**には、**ランタイム**時にも**ビルド**時にも、インポートされた関数のアドレスを入れることができます。**束縛された画像**とは、ビルド時に**IAT**が関数に束縛される画像のことです。束縛されていない画像とは、ロード時に**OS**ローダーによって**IAT**が埋められた画像のことです。

画像に境界がある場合は、以下のようにして外部DLLの関数を呼び出すことができます。

```
__declspec (dllimport) void function ();
function (); // myDll:function()を呼び出します。
```

画像に境界がない場合、**IAT**にはジャンクが含まれています。上記のコードを動作させるために**IATを更新するのは、OSローダーの責任です**。これは、ロードされたDLLモジュールのエクスポートテーブルを読み込んで (**GetProcAddress()**を呼び出し)、そのインポート関数の**IAT**エントリを上書きすることで実行できます。**IAT**の上書きは、上記の手順で行うことができます - 関数の**IAT**エントリを取得したら、それを上書きするだけです:)

この方法は、DLLや他のモジュールに**フック**をインストールする際にも有効です。

サポートリソース

はじめに

Windowsカーネルが、ディスクから何も読み込まずに画像を表示したり、XML設定ファイルを操作したりできることを不思議に思ったことはありませんか？リソースを追加する作業をしていて、それを**OS**でサポートできないかと考えたことはありませんか？答えは、**"Yes, of course!"** です。

しかし、リソースの解析は、他のディレクトリタイプよりも少し複雑です。他のセクションと同様に、ベースとなるオプションヘッダの **DataDirectory** メンバから取得できる **IMAGE_RESOURCE_DIRECTORY** 構造体。

```
PIMAGE_DATA_DIRECTORY DataDirectory = &OptionalHeader->DataDirectory [2]である。  
PIMAGE_RESOURCE_DIRECTORY resourceDirectory = (PIMAGE_RESOURCE_DIRECTORY) (DataDirectory->VirtualAddress + ImageBase)。
```

これらのセクションへのアクセス方法にパターンがあることに気付きましたか？そうだ、新しい構造にしよう。

この構造では、最後の3つを除くIMAGE_RESOURCE_DIRECTORY_ENTRYはあまりありません。
Win32リソースを扱ったことのある方は、リソースがIDや名前で識別できることをご存知かもしれません。この構造体の2つのメンバーは、これらのエントリの数と、エントリの総量 (NumberOfNamedEntries + NumberOfIdEntries)を知ることができ、これはすべてのエントリをループするのに便利です。お察しのとおり、エントリはDirectoryEntriesの配列に入っています。DirectoryEntriesは、IMAGE_RESOURCE_DIRECTORY_ENTRY構造体の配列で構成されていて、以下のよう
なフォーマットになっています。

さてさて、この新しい構造は、IMAGE_RESOURCE_DIRECTORY_ENTRYリソースディレクトリを表しています。

リソースディレクトリ構造

リソースからリソースディレクトリまでの立ち止まって考えてみましょう。(いいですか、リソースはツリーとして保存されていることを知っておくことが重要で
す。このツリーは次のような構造になっています。

```
DWORD NameOffset;  // 0x00000000  
WORD NameIsString;  // 0x00000000  
WORD DataIsDirectory;  // 0x00000000  
union {  
    WORD ID;  // 0x00000000  
    struct {  
        DWORD OffsetToData;  // 0x00000000  
        struct {  
            DWORD OffsetToDirectory;  // 0x00000000  
            struct {  
                WORD ResourceID;  // 0x00000000  
                WORD ResourceNameOffset;  // 0x00000000  
            }  
        }  
    }  
};  
// ImageResourceDirectoryEntry, *pImageResourceDirectoryEntry;  
// ...etc...
```

リソースグループにはいくつかの種類があり、そのグループに入っているリソースの種類を知ることができます。グループIDは以下の通りです。

- 1 - カーソル
- 2 - ビットマップ
- 3 - アイコン
- 4 - メニュー
- 5 - ダイアログ
- 6 - スtring
- 7 - フォントディ
- レクトリ 8 - フォ
- ント
- 9 - アクセラレータ
- 10 - RcData
- 11 - メッセージデー
- ブル 16 - パージョ
- ン
- 17 - DigInclude/li>
- 19 - Plug and Play
- 20 - VXD
- 21 - アニメーションカ
- ーソル 22 - アニメー
- ションアイコン 23 -
- HTML
- 24 - マニフェスト

リソースを見つけるためには、このツリーをトラバースする必要があります。ツリーには3つの層しかないと考えれば、難しいことではないというのが良い点です。

まず、リソースディレクトリ内のすべてのエントリをループする方法を見てみましょう。

```
//! ディレクトリの最初のエントリを取得  
IMAGE_RESOURCE_DIRECTORY_ENTRY* lpResourceEntry = lpResourceDir->DirectoryEntries;  
  
全てのエントリーをループする  
int entries = lpResourceDir->NumberOfIdEntries + lpResourceDir->  
NumberOfNamedEntries; while (entries-- != 0) {...
```

```
    // ! ビットマップリソースを探す (id=2)
    if (lpResourceEntry->Id == 2) {
        以下をご覧ください。
    }
    lpResourceEntry++;
}
```

これは簡単ですね。IMAGE_RESOURCE_DIRECTORY_ENTRYのIdメンバーは、グループIDを格納するために使用されます。ビットマップを探すとしたら、ルートディレクトリのビットマップグループにあるはずなので、ID=2のエントリを探します。

IMAGE_RESOURCE_DIRECTORY_ENTRYは、リソースエントリとディレクトリの両方を表しているの、それが何であるかをどうやって見分けるのでしょうか？なぜかというももちろん、DataIsDirectoryメンバーです。このメンバーが設定されていれば、それはディレクトリです。あ、でもディレクトリだったらどうやって読むの？ちょっと見てみましょう。

これは、lpResourceEntryがDataIsDirectoryの場合、上記は OffsetToDirectory から新しいディレクトリへのオフセットを取得し、それを startOfResourceSection に追加します。lpResourceEntryのOffsetToDirectoryはRVAではないのです。そうなんですよ・・・。なぜ、マイクロソフト、なぜ！？

lpResourceEntry += startOfResourceSection;

リソースセクションの開始点は、実際には IMAGE_RESOURCE_DIRECTORY_ENTRY 配列の最初のメンバのアドレスになります。つまり、このアドレスを OffsetToDirectory で得たオフセットに加えれば、このディレクトリのIMAGE_RESOURCE_DIRECTORY構造体へのポインタを得ることができるのです。これで、ディレクトリのエントリを読み取る作業が開始されます。)

もし、特定のリソースのためにディレクトリを解析している最中であれば、ディレクトリ内のすべてのリソースエントリをループしてください。resourceEntry IDフィールドが、探そうとしているリソースID（ここではプログラム固有のID）と一致すれば、リソースデータが見つかったことになります。

リソースデータは、.....ゾンデ！構造体に格納されています。ディレクトリエントリ構造のOffsetToDataメンバから取得することができます。に似ています。OffsetToDirectoryメンバで、これもリソースセクションの開始からのオフセットです。

ポインタを取得すると、リソースデータを取り出すことができます。では、その構造を見てみましょう。

```
typedef struct _IMAGE_RESOURCE_DATA_ENTRY
{
    uint32_tOffsetToData;
    uint32_t
    Size; uint32_t
    tCodePage
    ; uint32_t
    tReserved
};
image_resource_data_entry, *pimage_resource_data_entry;
```

その通りです。OffsetToDataは実際のリソースデータを指すRVAで、Sizeはそのデータのサイズをバイト単位で表しています。例えば、ビットマップのリソースを探している場合、OffsetToDataはビットマップのBITMAPINFOHEADER構造を指すRVAとなり、これは任意のビットマップローダで処理することができます。

結論

この章はこれで終わりです。今後は、デバッグデータやCOMDATSなどの項目を追加していく予定です。

この章にはデモはありません。この章は主に、PE実行ファイル形式の内部動作とその作業に興味がある人のために公開されています。メインのシリーズでは、プログラムのロードと実行だけを行うことがありますので、その他の情報は完全性のためにのみ提供されています。デモのためにテキストで提供されたコードは、すべて動作確認済みです（若干の修正を加えています）。

次の章では、PE実行ファイルフォーマットの使用と、ユーザーモードプログラムをサポートするローダーの構築を行います。続いて、マルチタスクについて説明します。

次の機会まで。

マイク
BrokenThorn Entertainment社。現在、DoEとNeptuneOperating Systemを開発中です。質問

やコメントはありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ私に教えてください。





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - 8259A PICマイクロコントローラ

by Mike, 2007

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。



8259A PICマイクロコントローラの全ピンが表示されています。

はじめに

歓迎します。:)

このチュートリアルでは、非常に重要なトピックを取り上げています。**プログラム可能なインタラプトコントローラ**です。このマイクロコントローラを初期化して、IRQにマップングする必要があります。これは、割り込みを設定したり、割り込み要求を処理する際に必要となります。

今回は、初めてのコントローラチュートリアルです。このチュートリアルでは、それぞれのデバイスについて深く掘り下げ、それらを扱うための実用的なインターフェイスを構築しています。保護モードに入っているため、ガイドとなるものがないことを忘れないでください。一歩間違えると、予想外の結果になってしまいます。手がかりがない分、それぞれのコントローラと直接コミュニケーションをとらなければなりません。そのため、本連載ではハードウェア・プログラミングの概念を重視し、読者がハードウェア・レベルのプログラミングをより深く理解できるようにしています。

このチュートリアルでは、学んだことをすべて試してみます。このチュートリアルでは、学習したことをすべて試すことができます。できるだけ簡単にするようにします。

いいですか？

準備

これは、多くのマイクロコントローラ・プログラミング・チュートリアルの最初のものです。このチュートリアルでは、各マイクロコントローラのほぼすべての機能をカバーしています。メインシリーズでは、必要に応じてこれらのチュートリアルを参照し、これらのコントローラが必要とするものをカバーしていきます。

このチュートリアルはかなり複雑です。**8259A**マイクロコントローラをハードウェアとソフトウェアの両方の観点から取り上げ、PCとの接続や動作を正確に理解します。また、このマイクロコントローラのすべてのコマンド、レジスタ、および部品をカバーします。

歴史

**To do - 近いうちにこのセクションを追加する予定です*。*

8259A PICはハードウェア割り込みを扱うため、まずは割り込みとは何か、どのように動作するのかを基本的に理解する必要があります。

割り込み

割り込みとは、ソフトウェアやハードウェアの注意を必要とする外部の非同期信号のことです。これにより、現在のタスクを中断して、より重要なことを実行することができます。

ハードにはならない。割り込みは、ゼロ除算などの問題をトラップするのに役立つ方法を提供します。プロセッサが現在実行中のコードに問題を発見した場合、その問題を解決するために実行する代替コードをプロセッサに提供します。

その他の割り込みは、ソフトウェアをルーチンとしてサービスする方法を提供するために使用されることがあります。これらの割り込みは、システム内の任意のソフトウェアから呼び出すことができます。これは、リング3のアプリケーションがリング0レベルのルーチンを実行する方法を提供するシステムAPIによく使われます。

特に、非同期に状態が変化する可能性のあるハードウェアから情報を受け取る方法として、インタラプトは多くの用途があります。

割り込みの種類

割り込みには、「ソフトウェア割り込み」と「ハードウェア割り込み」の2種類があります。

ソフトウェアインタラプト

ソフトウェア割り込みとは、ソフトウェアで実装・起動される割り込みのことです。通常、プロセッサの命令セットには、ソフトウェア割り込みを処理するための命令が用意されています。x86アーキテクチャの場合、これらは通常**INT imm**、**INT 3**です。また、**IRET**、**IRETD**命令も使用します。

例えば、ここではソフトウェア命令で割り込みを発生させます。

これらの命令は、ソフトウェアによる割り込みの生成や、**割り込みルーチン (IR)** の実行に使用できます。

ここでは、ソフトウェア割り込みは取り上げません。**8259A PIC**マイクロコントローラは、ハードウェア割り込みのみ対応しています。ソフトウェア割り込みについては、別のチュートリアルで取り上げます。

ハードウェアインタラプト

ハードウェア割り込みとは、ハードウェアデバイスによって引き起こされる割り込みのことです。通常、これらは注意を必要とするハードウェアデバイスです。ハードウェアインタラプトハンドラは、このハードウェア要求を処理するために必要となります。

スプリアスインターラプト

これは、割り込みラインの電氣的干渉や、ハードウェアの不具合によって発生するハードウェア割り込みです。これは絶対に避けたいことです。

割り込みモード

割り込みにはいくつかのモードとクラスがあり、それらをカバーする必要があります。PICのプログラミングでは、モードを選択する必要があります。**注**

：このセクションでは、**8259A PIC**のハードウェア・ピン・レイアウトの知識が必要になるかもしれません。これについては次のセクションで説明します。**レベルトリガ**

レベルトリガ割り込みは、PICのIR (Interrupt Request) ラインに電流 (1) が流れたときに発生すると判断されます。あるデバイスが信号を送る (この設定ラインをアクティブにして)、割り込みが処理されるまでその状態を維持します。

レベルトリガの割り込みラインは、回路がそれに対応できるように設計されていれば、複数の割り込みで共有することができます。

このモードは、ラインを共有する方法のため、好ましいモードです。**IR**ラインがアクティブになると、**CPU**は同じラインを共有しているすべてのデバイスを検索し、どのデバイスが信号をアクティブにしているかを見つけ出します。見つかった後、**CPU**はすべての機器を再チェックし、サービスを必要とする他の機器がないことを確認します。

この方法の問題点は、より高い優先順位で処理しなければならない割り込みがあった場合、他の割り込みが処理されるまで、他のすべての割り込みが永久にブロックされてしまうことです。結局のところ、一度にアクティブにできるのは1つのラインだけです。

エッジトリガ

エッジトリガ割り込みは、PICのIR (Interrupt Request) ラインに電流 (1) が流れたときに発生するとされている。デバイスはシングルパルスで信号を送り (このラインをアクティブにする)、ラインを元の状態に戻します。

エッジトリガの割り込みラインは、回路がそれに対応できるように設計されていれば、複数の割り込みで共有するこ

とができます。パルスが短すぎて検出できない場合は、検出されません。

これは、割り込み要求を通知する電流パルスに過ぎないため、エッジトリガモードでは、レベルトリガのように**IRQ**ラインを共有する問題は発生しません。

もちろん、**IRQ**ラインには1パルスの電流が流れるだけなので、割り込みを見逃す可能性もあります。これは、初期のコンピュータで**CPU**のロックを引き起こす原因となりました。

しかし、最近ではこのようなロックアップは時間の経過とともに減少しています。

ハイブリッド

この2つのモードにはそれぞれ長所と短所があります。多くのシステムでは、この2つのモードのハイブリッドが採用されています。具体的には、ほとんどのシステムでは、**CPU**の**NMI (Non Maskable Interrupt)** ピンでエッジトリガとレベルトリガの両方の割り込みをチェックしています。**この目的は、NMIピンがシステムの重大な問題を知らせるために使用され、大きな問題やシステム全体の誤動作、場合によってはハードウェアの損傷を引き起こす可能性があるからです。**

Non Maskable Interrupt (ノンマスカブル・インタラプト) とは、まさにこのことで、どのデバイスからも無効にされたり、マスクされたりすることはありません。これにより、ハイブリッドセットアップと同様に、**NMI**ピンがセットされた場合、システムは大きな問題を起こすことなく穏やかに終了することができます。

シグナルされたメッセージ

この種のハードウェア割り込みは、物理的な割り込みラインを使用しません。このタイプのハードウェア割り込みは、物理的な割り込みラインを使用せず、システムバスなどの別の媒体を利用してメッセージを送信します。このタイプの割り込みは、エッジトリガ割り込みと同様に、デバイスが媒体上にパルス状の電流を送るだけです。

この種のシステムでは、制御バス上にメッセージ信号による割り込み番号を示す特別な割り込みラインを使用することができます。これらの番号は、一連のビットとしてメディア上で送信されるため、他の割り込みタイプのように1本の割り込みラインに制限されることはありません。また、このタイプの割り込みは、他の割り込みタイプのように、1本の割り込みラインに限定された制約がないため、下層システムが許す限り、多くの割り込みを管理することができます。また、これらのタイプの割り込みは、割り込みベクターの共有にも対応しています。

PCI Expressでは、これらのタイプの割り込みを使用しています。

以上です。

さて、ここには多くの情報があります。**8259A**は、レベルトリガとエッジトリガの割り込みしかサポートしていません。このため、**8259A**マイクロコントローラを扱う際には、これらに主眼を置く必要があります。

割り込みベクターテーブル

IVT(Interrupt Vector Table)は、**インタラプトベクター**のリストです。**IVT**には**256個のインタラプト**があります。

割り込みルーチン(IR)

割り込みルーチン(IR)は、**割り込み要求(IRQ)**を処理するための特別な機能です。

プロセッサが**INT**などの**割り込み命令を実行すると、IVT (Interrupt Vector Tabl)** 内のその位置で**IR (Interrupt Routine)** が実行されます。

つまり、私たちが定義したルーチンを実行するだけです。難しくないでしょう？この特別なルーチンは、**AX**レジスタの値に基づいて、通常実行する**割り込み関数**を決定します。これにより、1つのインタラプトコールに複数のファンクションを定義することができます。例えば、**DOS**の**INT21h**関数**0x4c00**のように。

覚えておいてください。**割り込みを実行すると、自分が作成した割り込みルーチンが単純に実行されます。**例えば、**INT 2**という命令は、**IVT**のインデックス**2**の**IR**を実行します。いいですか？

IVTマップ

IVTは、物理メモリの最初の**1024**バイト、アドレス**0x0**から**0x3FF**までに配置されています。**IVT**内の各エントリは**4**バイトで、次のような形式になっています。

- **バイト0: 割り込みルーチン(IR)**のオフセットローアドレス **バイト**
- **1:** **IR**のオフセットハイアドレス
- **バイト2 :** **IR**のセグメント**Low**アドレス
- **バイト3 :** **IR**のセグメント**ハイ**アドレス

IVTの各エントリには、単に呼び出す**IR**のアドレスが含まれていることに注意してください。これにより、メモリ上の任意の場所 (**Our IR**) に簡単な関数を作成することができます。**IVT**に関数のアドレスが格納されていれば、すべてがうまくいきます。

2021/11/15 13:13

オペレーティングシステム開発シリー

では、IVTを見てみましょう。最初の数個の割り込みは予約されており、そのままです。

x86 インタラプトベクターテーブル (IVT)		
ベースアドレス	割り込み番号	説明
0x000	0	0で割る
0x004	1	シングルスステップ (デバッグ)
0x008	2	ノンマスカブルインタラプト (NMI) 端子
0x00C	3	ブレークポイント (デバッグ)
0x010	4	オーバーフロー
0x014	5	バウンドチェック
0x018	6	未定義のオペレーションコード (OPCode) 命令
0x01C	7	コプロセッサなし
0x020	8	ダブルフォールト
0x024	9	コプロセッサ・セグメント・オーバーラン
0x028	10	タスクステートセグメント (TSS) が無効
0x02C	11	セグメントが存在しない
0x030	12	スタックセグメントオーバーラン
0x034	13	GPF (General Protection Fault) について
0x038	14	ページフォルト
0x03C	15	未分類
0x040	16	コプロセッサのエラー
0x044	17	アライメントチェック (486+のみ)
0x048	18	マシンチェック (Pentium/586+のみ)
0x05C	19-31	予約済みの例外
0x068 - 0x3FF	32-255	ソフトウェアの使用のためのインタラプトフリー

難しいことはありません。これらの割り込みは、それぞれIVT内のベースアドレスに配置されています。

プロテクトモード(PMode)での割り込み処理

プロテクトモードでは、各IVTエントリが、**IDT (Interrupt Descriptor Table)** 内で定義された割り込みルーチン (IR) を指す必要があります。IDTについては、このチュートリアルとは直接関係がないため、別のチュートリアルで詳しく説明します。

IDTは、実行する割り込みルーチン(IR)のベースアドレスを記述した**割り込み記述子**の配列で、保護レベルやセグメント情報などの追加情報を含んでいます。**PMode**では、使用されるメモリマップを定義するグローバルディスクリプターテーブル (GDT) を使用します。割り込みルーチンの多くは、**GDT**によってマップされたコード記述子の中に入ります。これが**PMode**でIDTが必要な理由です。

今はまだ理解できなくても気にしないでください。とりあえず、256個の関数ポインタの配列で、IVTと同じようにマッピングされていると思ってください（普通はそうです）。

ハードウェアインタラプト

割り込みには、ソフトウェアで発生させるもの (**INT**、**INT 3**、**BOUND**、**INTO**などの命令で使用) と、ハードウェアで発生させるものがあります。

ハードウェア割り込みは、**PC**にとって非常に重要です。これにより、他のハードウェアデバイスが、何かが起ころうとしていることを**CPU**に知らせることができます。例えば、キーボードのキーストロークや、内部タイマーの1クロック分の目盛りなどです。

これらの割り込みが発生したときに、どのような**IRQ (Interrupt Request)** を生成するかをマッピングする必要があります。こうすることで、ハードウェアの変化を追跡することができます。では、これらのハードウェア割り込みを見てみましょう。

x86ハードウェアインタラプト		
8259A 入力端子	割り込み番号	説明
IRQ0	0x08	タイマー
IRQ1	0x09	キーボード
IRQ2	0x0A	8259Aスレーブコントローラ用カスケード
IRQ3	0x0B	シリアルポート2
IRQ4	0x0C	シリアルポート1
IRQ5	0x0D	ATシステム。パラレルポート2PS/2システム：予約
IRQ6	0x0E	ディスクドライブ
IRQ7	0x0F	パラレルポート1
IRQ8/IRQ0	0x70	CMOS リアルタイムクロック
IRQ9/IRQ1	0x71	CGAの垂直方向のリトレース
IRQ10/IRQ2	0x72	予約
IRQ11/IRQ3	0x73	予約
IRQ12/IRQ4	0x74	ATシステム：予約済みPS/2：補助的なデバイス
IRQ13/IRQ5	0x75	FPU
IRQ14/IRQ6	0x76	ハードディスク・コントローラ
IRQ15/IRQ7	0x77	予約

各デバイスについては、まだまだあまり気にする必要はありません。**8259A**のピンについては、次のセクションで詳しく説明します。この表に記載されている割り込み番号は、これらのイベントが発生したときに実行されるデフォルトの**DOS割り込み要求 (IRQ)** です。

ほとんどの場合、新しい割り込みテーブルを作り直す必要があります。そのため、ほとんどのオペレーティングシステムでは、**PIC**が使用する割り込みをリマップして、IVT内の適切な**IRQ**を呼び出すようにする必要があります。これは、リアルモードのIVTでは**BIOS**が行ってくれます。このチュートリアルでは、後ほどこの方法を説明します。

8259プログラマブルインタラプトコントローラ

8259マイクロコントローラ・ファミリは、**プログラマブル・インタラプト・コントローラ(PIC)集積回路(IC)**のセットです。**チュートリアル7**をもう一度見てみましょう...の下に**プロセッサ・アーキテクチャ**」のセクションでは、**プロセッサが独自のPICマイクロコントローラを内蔵していることに注目してください**。これは非常に重要な点です。

回路設計上の制約から、**PICは8つのIRQしかサポートしていません**。これは大きな制限である。デバイスが増えるにつれ、**IBM**はこの制限が非常に悪いものであることにすぐに気づきました。このため、ほとんどのマザーボードには**セカンダリ (スレーブ) PIC**マイクロコントローラが搭載されており、**プライマリPICと一緒に動作します**。

プロセッサを使用しています。今日では、これは非常に一般的なことです。1つのPICは、別のPICと「カスケード接続」（一緒に動作可能）することができます。これにより、追加のPICでより多くのIRQをサポートすることができます。

対応するPICの数が多いほど、より多くのIRQを処理できます。カスケード接続して最大64のIRQをサポートすることができます。いいですね。

覚えておいてください。ほとんどのコンピュータには2つのPICがあり、1つはプロセッサ内部に、1つはマザーボードに搭載されています。システムによってはこれがない場合もあります。覚えておいてください。各PICは最大8つのIRQをサポートしています。覚えておいてください。各PICは相互に通信することができ、PICの数に応じて最大64のIRQが可能です。

難しいことはありません :)

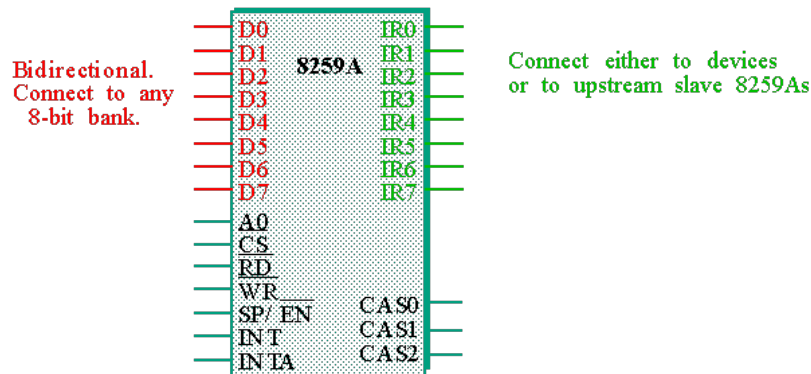
8259 ハードウェア

マイクロコントローラーがハードウェアレベルでどのように動作するかを理解することは、ソフトウェア側の動作を理解するのに役立ちます。PICはハードウェアの割り込み時にのみ使用されることを覚えておいてください。

8259Aマイクロコントローラ

このチュートリアルの上部には、すべての電子ピンが表示された実際の8259 Dual Inline Package (DIP)の画像があります。ここでは、より分かりやすくするために、コントローラをよりシンプルなグラフィックで表現します。これらの図には表示されていませんが、8259が持つ唯一のピンは、GND (グラウンド) とVcc (入力電圧) です。これらのピンは、このチュートリアルの一冊上の写真に表示されています。

まず、これからプログラミングする内容について説明します。



8259Aプログラマブルインタラプトコントローラ」です。

上の画像の各線は、コントローラの各電子ピンを表しています。これらの電子ピンは、コントローラと他のシステムとの接続に使用されます。

このチップは、OS内のIRQを処理するためにプログラムする必要があります。各ピンで詳しく見てみましょう。重要なピンは太字にしました。

- **WR**端子：ライトストローク信号に接続します (Pentiumでは8本のうちの1本) **RD**端子
- 子：IOCR (Input Output Control Routine) 信号に接続します。INT端子：マイ
- クロプロセッサのINTR端子に接続します。
- INTA端子：マイクロプロセッサのINTA端子に接続します。A0端子
- : 異なるコマンドワードを選択可能
- CS端子：プログラミングや制御のためにチップを有効にする。
- SP/EN端子：スレーブプログラム (SP) /イネーブルバッファ (EN) 。
 - スレーブプログラム (1=マスター、0=スレーブ)
 - Enable Buffer (バッファモード時のデータバストランジを制御)
- CAS0,CAS1,CAS2端子です。カスケード接続されたシステムにおいて、マスターのPICコントローラからスレーブの
- PICコントローラへの出力に使用します。D0~D7端子。D0~D7端子：8ビットのデータコネクタ端子です。

ここには、いくつかの重要なピンがあります。ピンD0~D7は、外部デバイスがPICと通信するための手段です。これは小さなデータバスのようなもので、PICにデータを送信する方法を提供します。

PIC同士を接続できることを覚えておいてください。これにより、最大64個のIR番号をサポートすることができます。言い換えれば、64個のハードウェア割り込みです。CAS0、CAS1、CAS2ピンは、これらのPICの間で信号を送信する方法です。

INTピンとINTAピンを見てください。プロセッサのINTピンとINTAピンがPICのこれらのピンに接続されていることを「プロセッサの視点」の項で思い出してください。割り込みを実行しようとするとき、プロセッサはFLAGSレジスタの割り込みフラグ (IF) とトラップフラグ (TF) をクリアして、INTRピンを無効にすることを覚えておいてください。PICのINTピンは、プロセッサのINTRピンに接続します。

つまり、プロセッサは、割り込みを実行する際に、基本的にPICのINTピンを無効にします。

これにより、ピンIR0-IR7を他のPICにストリームすることができます。これらの8つのピンは、実行される8ビットの割り込み番号を表しています。これらのラインは、他のPICコントローラに割り込み番号を送信する方法を提供し、そのコントローラが代わりに処理できるようにします。

ここで重要なのは、複数のPICを組み合わせることで、より多くの割り込みルーチン数をサポートできるということです。IRラインは、他のPICのデータラインに接続して、データを転送します。ラインは8本 (8ビット) しかないので、最大で8台のPICを接続し、64個の割り込み番号をサポートすることができます。

なるほど...。ここにはたくさんのことが書かれていますね。プロセッサがプライマリPICに接続する方法、PICが他のPICと結合してPICのチャイを作る方法を説明しました。

これは素晴らしいことですが、全く役に立ちません。割り込みはどのようにしてハードウェアを介して実行されるのか？このコントローラが「プログラマブル」なのはなぜか？どうやってPICをプログラムして、自分のニーズに合わせて働かせることができるのか？

PICのプログラミングは、PICが持つ8ビットのデータラインを通じてコマンドバイトを送信することを中心に行われます。この8ビットのコマンドバイトは、PICが何をすべきかを示す特定のフォーマットに従っています。PICをプログラムするためには、これらのコマンドを知る必要があります。これについては後ほど説明します。

PICがどのように動作するかを詳しく見てみましょう。これは、8259Aのピンや、割り込み信号がどのように送られるかを理解するのに役立ちます。

8259A コネクション

注：このセクションでは、デジタル・ロジック・エレクトロニクスの知識が必要な場合があります。

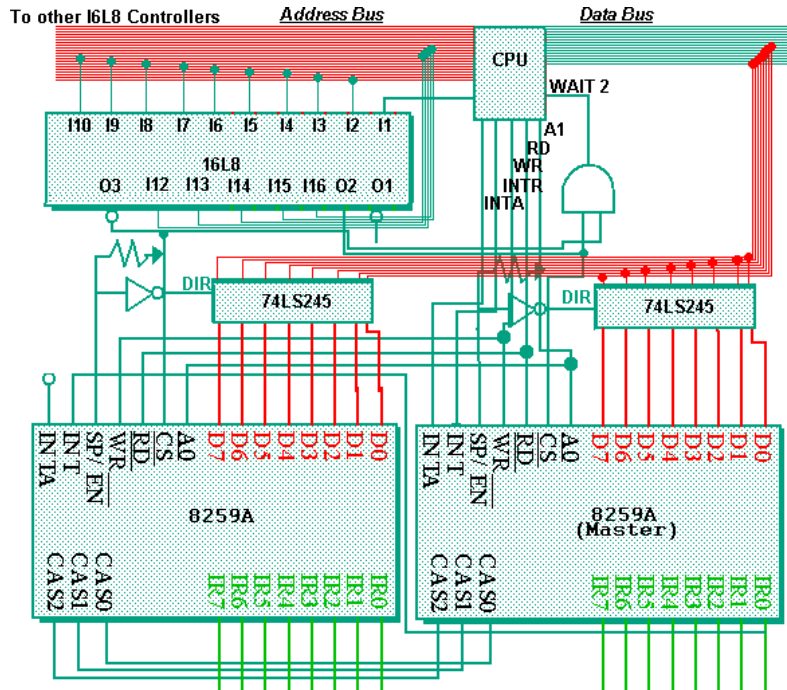
なるほど...。ここまでは、8259AのPICピンを見てきました。では、これらのピンを別の視点から見て、一般的なコンピュータの中でどのように見えるかを見てみましょう。

PICとプロセッサの接続

まず、現在のほとんどのコンピュータには8259A PICが2台搭載されていると言ったことを覚えていますか？これは半分しか当てはまりません。**Processor Architecture**で、プライマリPICがプロセッサに組み込まれていることを思い出してください。これには理由があって、すぐにわかります。

簡単に説明すると、今回のシステムには2つのPICコントローラがあり、どちらもマザーボード上に直接設置されているとします（いずれもプロセッサとは統合されていません）。

これをグラフにしてみると、こんな感じになります。



なるほど...。ここには多くのことが起こっています。これはIOサブシステムとISAバスの一部を表示しており、8259Aコントローラが共通の16L8を介してどのようにシステムバスに接続しているかを示しています。

このシリーズでは、デジタル・ロジック・エレクトロニクスは必須条件ではありませんので、分からなくても全く気にしないでください（笑）。)また、この画像は細部が不足しています。とはいえ、部品間の基本的なリンクや接続は表示されています。

上の画像を見ると、いくつかの重要な注意点があります。

スレーブコントローラがプライマリコントローラにどのように接続しているかに注目してください。

プロセッサと直接接続する必要があるのはプライマリPICだけであることに注目してください。最近のコンピュータでは、このような緊密な統合が必要なため、プライマリPICをプロセッサの内部に直接組み込んで、この依存関係を最小限にしています。

また、CAS0-CAS2ピンが2番目のPICに直接接続されていることにも注目してください。これにより、プライマリPICがセカンダリPICにコマンドを送ることができます。

そして、周知の通り、IRラインはそのラインを制御する他のコントローラに接続されています。例えば、ハードウェア割り込み0は、タイマーの割り込みを表します。8254プログラマブルインターバルタイマー (PIT) コントローラは、直接接続されているため、IR0ラインを通してプライマリPICに信号を送ります。この信号は、割り込みが処理されるまでアクティブな状態を維持する電流か、一定時間保持される単一パルスのいずれかになります。PICに何を監視させたいのかを制御することができます。これについては、後で詳しく説明します。

さて、ここからが本題です。)悪名高き8259A PICマイクロコントローラです。

ハードウェア割り込みの実行方法

すべてのマイクロプロセッサの下側には、コネクタがあります。これは平板状のものと、マザーボードに接続するためのピン状のものがあります。これらのピンのうち2つは、INTRとNMIのピンです。これに加えて、割り込みの完了を確認するためのもう1つのピン、INTAがある。

ソフトウェア割り込みは、ハードウェア割り込みとは扱いが異なります。どちらの割り込みも、メモリ上のアドレス0~0x3ffにある「割り込みベクターテーブル」の中にあります。

覚えておいてください。プログラマブルインタラプトコントローラで処理されるのは、ハードウェアインタラプトのみです。

割り込みの発生

デバイスコントローラが割り込みを発生させる必要がある場合、何らかの方法でPICに信号を送る必要があります。ここでは、このデバイスをタイマーとし、割り込みライン0を使用すると仮定します。

1. タイマーコントローラは、IR0ラインをアクティブにすることでPICに信号を送ります。これにより、その状態は0（電源なし）から1（電源がラインを通っている）に変わります。
2. PICは、IRR (Interrupt Request Register) の内部でIRQを表すビットを設定します。この例では、ビット0が1に設定されます。
3. PICはIMR (Interrupt Mask Register) を調べて、割り込みを処理できるかどうかを確認します。
 - 割り込みの処理が可能な場合、PICは処理を待っている優先度の高い割り込みがあるかどうかを判断します。優先度の高い割り込みがある場合は、優先度の高い割り込みが処理されるまで割り込み要求は無視されます。
 - 割り込みの処理が可能で、より優先度の高い割り込みがない場合、PICは次のステップに進みます。
4. PICは、INTAピンを通じてプロセッサに割り込みが発生したことを通知します。これで、プロセッサは割り込みが発生

生したことを知ります。

プロセッサは、割り込みを認識します。

1. CPUは、現在の命令の実行を完了します。
2. CPUはRFLAGS内のIF (Interrupt Flag) を調べます。
 - IFがセットされている場合、CPUはINTRピンを介して割り込み要求をPICに返します。IFがクリアされた場合、割り込み要求は無視されます。
3. PICはINTRを介してアクリリジメント信号を受信します。
4. PICは、割り込みベクター番号をD0~D7端子に入れます。
 - この割り込みベクター番号は、PICの初期化時にICW (Initialization Control Word) 2から取得します。これについては後で説明します。
5. また、PICはIRQ番号をD0~D7に

6. PICは、**インサービスレジスター (ISR)** 内の正しいビットを設定します。~~この~~例では、ビット**0**です。これは、インタラプト**0**が現在サービスされていることを示します。

これで、プロセッサは実行すべきIRQ番号と割り込みベクター番号を手に入れたことになりま

インタラクション

- 1. プロセッサは、現在のプロセスを中断します。EFLAGS、CS、EIPをスタックにプッシュします。
- 2. プロセッサは、（PICから与えられた）割り込みベクター番号を使用します。
 - リアルモードでは、CPUはIVTにオフセットします。プロテクトモードでは、プロセッサはIDTにオフセット
 - されます。リアルモードでは
 - CPUがオフセットしてIVTに正しく入力されていること
 - CPUは呼び出したい割り込みのベースアドレスをCS:IPにロードする 割込
 - みが制御を行う。
 - プロテクトモード。
 - CPUは、ロードされたIDTを使って、オフセットして
 - ゲートディスクリプタのセクタ・ファイルは、CSセグメントセクタに読み込まれま
 - す。ゲートディスクリプタのoffset fieldはEIPに読み込まれます。
 - ページングが有効な場合、このアドレスはリニアアドレスから物理アドレスに変換されます。ここで
 - 、CPUは現在の状態に対してアーキテクチャ固有のセキュリティチェックを行います。
 - これで、割り込みルーチンは、ゲートディスクリプタ+CS:EIPから制御を受けることができます。

割り込みサービスルーチン

現在、ISRはハードウェア割り込みを処理するために実行されています。ISRは、特定のデバイスを処理するために必要なあらゆる動作を行うことができます。例えば、デバイスへのデータの読み書き、ステータスレジスタの読み出し、コマンドの送信などです。

この間、すべての割り込みはIMR（Interrupt Mask Register）によってマスクアウトされます。つまり、割り込みを終了させる要求が出されるまで、すべてのハードウェア割り込みが禁止されます。このためには、EOI（End of Interrupt）コマンドをPICに送信する必要があります。

EOI信号がプライマリPICのコマンドレジスタを介してPICに送信された後、PICはインサースビスレジスタ（IRR）の appropriateビットをクリアし、新たな割り込みを処理する準備が整います。

その後、割り込みサービスルーチンはIRETD命令を実行し、割り込みが発生したときにプロセッサがプッシュしたEFLAGS、CS、EIPレジスタをポップします。

これにより、第一のタスクに制御が戻ります。

8259A レジスター

8259Aは、プロセッサと同様に、いくつかの内部レジスタを持っています。

コマンドレジスタ

これは、マイクロコントローラーにコマンドを送信するための書き込み専用のレジスタです。送信可能なコマンドには様々なものがあります。いくつかのコマンドは、他のレジスタからの読み出しに使用され、他のコマンドは、初期化やデータの送信に使用されます。ここでは、これらのコマンドについて説明します。

ステータスレジスタ

これは、PICのステータスを決定するためにアクセスすることができるリードオンリーのレジスタです。

割り込み要求レジスタ(IRR)

このレジスタは、確認応答を保留している割り込みを指定します。

注：このレジスタは内部にあり、直接アクセスすることはできません。

割り込み要求レジスタ(IRR)		
ビット数	IRQ番号（プライマリコントローラ	IRQ番号（スレーブコントローラ
0	IRQ0	IRQ8
1	IRQ1	IRQ9
2	IRQ2	IRQ10
3	IRQ3	IRQ11
4	IRQ4	IRQ12
5	IRQ5	IRQ13
6	IRQ6	IRQ14
7	IRQ7	IRQ15

ビットがセットされている場合は、デバイスからの割り込み信号があり、PICはCPUに信号を送りましたが、割り込みを進めるためにCPUからの確認を待っている状態です。

イン・デバイス・レジスタ（ISR

このレジスタは、すでにアクノリッジされているが、EOI（End of Interrupt）信号を待っている割り込みを指定します。EOI信号は、割り込みの終了を決定する非常に重要な信号です。

注：8259Aに割り込みを認識させるために、割り込み完了時にEOI信号を送る必要があります。そうしないと、未定義の動作や誤動作の原因になります。これについては後で説明します。

注：このレジスタは内部にあり、直接アクセスすることはできません。

インサースビスレジスター（ISR		
ビット数	IRQ番号（プライマリコントローラ	IRQ番号（スレーブコントローラ
0	IRQ0	IRQ8
1	IRQ1	IRQ9
2	IRQ2	IRQ10
3	IRQ3	IRQ11
4	IRQ4	IRQ12
5	IRQ5	IRQ13
6	IRQ6	IRQ14

7	IRQ7	ズ	IRQ15
---	------	---	-------

割り込みマスクレジスタ(IMR)

これにより、このレジスタで指定された割り込みを実行する前に、特定のより重要な割り込みを実行することに集中することができます。

これは8ビットのレジスタで、各ビットが割り込みを無効にするかどうかを決定します。このビットが0の場合、割り込みは有効です。1であれば、割り込みデバイスは無効になります。

割り込みマスクレジスタ(IMR)		
ビット数	IRQ番号（プライマリコントローラ）	IRQ番号（スレーブコントローラ）
0	IRQ0	IRQ8
1	IRQ1	IRQ9
2	IRQ2	IRQ10
3	IRQ3	IRQ11
4	IRQ4	IRQ12
5	IRQ5	IRQ13
6	IRQ6	IRQ14
7	IRQ7	IRQ15

これは重要なレジスタで、特定のデバイスからの割り込みを有効にしたり無効にしたりすることができます。これらのIRQはそれぞれ、上に示したx86 Hardware Interruptsの表に記載されているデバイスを表しています。

例えば、COM1（シリアルポート1）を有効にしたいとします。x86ハードウェア割り込みテーブルを見ると、これはIRQ4にマッピングされています。したがって、COM1の割り込みを有効にするためには、プライマリPICの割り込みマスクレジスタのIRQ4ビットを設定すればよいことになります。このレジスタはソフトウェアのポート番号0x21にマッピングされています（この点については後述します）ので、このポート位置に書き込むことでビットを設定するだけです。

```
で      al, 0x21
そして al, 0xEF
アウト 0x21, al

          プライマリPICのIMR（Interrupt Mask Register）で読み込まれる
          ; 0xEF => 11101111b.これにより、ALのIRQ4ビット（Bit5）が設定さ
          れます。
          の値をIMRに書き戻す。
```

かつこよすぎて学校に行けない B)

ハードウェア割り込みが発生すると、8259AはEOI（End of Interrupt）信号を受信するまで、他のすべての割り込みをマスクします。割り込みの完了時にEOIを送信する必要があります。これについては後述します。

8259A ソフトウェアポートマッピング

他のハードウェアコントローラと同様に、BIOSのPOSTでは、各コントローラがソフトウェアポートの特定の領域を使用するようにマッピングされます。このため、PICコントローラと通信するためには、ソフトウェアポートを使用する必要があります。

8259A ソフトウェアポートマップ	
ポートアドレス	説明
0x20	プライマリPICコマンド&ステータスレジスタ
0x21	Primary PIC Interrupt Mask Register and Data Register
0xA0	セカンダリ（スレーブ）PICコマンド&ステータスレジスタ
0xA1	セカンダリ（スレーブ）PIC割り込みマスクレジスタおよびデータレジスタ

Primary PICのInterrupt Mask RegisterがPort 0x21にマッピングされていることに注目してください。どこかで見たことがあるような気がしますね。

コマンドレジスタとステータスレジスタは、同じポート番号を共有する別のレジスタです。コマンドレジスタは書き込み専用で、ステータスレジスタは読み出し専用です。これは重要な違いで、PICは書き込みラインと読み込みラインのどちらがセットされているかによって、アクセスするレジスタを決定します。

各デバイスのレジスタと通信したり、PICを制御したりするためには、これらのポートに書き込みができる必要があります。それでは、PICのコマンドを見てみましょう。

8259Aコマンド

PICの設定は非常に複雑です。PICの設定は、初期化や動作に使用される様々な状態を含むビットパターンである、一連のCommand Wovrdsを通して行われます。少し複雑に見えるかもしれませんが、それほど難しいことはありません。

そのため、まずはPICコントローラを初期化し、その後、PICを操作・制御する方法をご紹介します。

初期化コントロールワード（ICW

PICを初期化する目的は、PICのIRQ番号を我々のものに再マッピングすることです。これにより、ハードウェア割り込みが発生したときに、適切なIRQが生成されるようになります。

PICを初期化するためには、プライマリPICコマンドレジスタにコマンドバイト（Initialization Control Word（ICW）として知られている）を送信する必要があります。これがICW 1です。

最大で4つの初期化制御ワードがあります。これらは必須ではありませんが、しばしば必要となります。ここでは、その内容をご紹介します。

注：システム内に複数のPICがあり、互いにカスケード接続される場合は、両方のPICにICWを送信する必要があります。

ICW 1

これは、PICを初期化するために使用される一次制御ワードです。これは、一次PICコマンドレジスタに入れなければならない7ビットの値です。これは、フォーマットです。

初期化コントロールワード（ICW） 1		
ビット数	価値	説明
0	IC4	セット(1)の場合、PICは初期化中にIC4を受信することを期待します。
1	SNGL	セット（1）の場合、システム内のPICは1つだけです。クリアした場合、PICはスレーブPICとカスケード接続され、ICW3をコントローラに送る必要があります。

2	ADI	設定(1)されている場合、CALLアドレス間隔は4、それ以外は8となります。これはx86では通常無視され、デフォルトでは0に設定されています。
3	LTIM	(1)に設定されている場合、レベルトリガモードで動作します。未設定(0)の場合、エッジトリガモードで動作します。
4	1	初期化ビット。PICを初期化する場合は1を設定

5	0	MCS-80/85：インタラプト・ベクタ・アドレス x86アーキテクチャ。0でなければならない
6	0	MCS-80/85：インタラプト・ベクタ・アドレス x86アーキテクチャ。0でなければならない
7	0	MCS-80/85：インタラプト・ベクタ・アドレス x86アーキテクチャ。0でなければならない

ご覧のように、ここにはたくさんのことが起こっています。これらのうちのいくつかは以前に見たことがあります。これらのビットのほとんどはx86プラットフォームでは使用されていないので、これは思ったほど難しくありません。

プライマリPICを初期化するためには、initil ICWを作成し、適切なビットを設定するだけです。つまり、...

- **ビット0** - ICW 4を送信するために1に設定します。
- **x86**アーキテクチャには2つのPICがあるので、プライマリPICをスレーブとカスケード接続する必要があります。0のままにしてください。
- **ビット2** - CALLアドレス間隔。x86では無視され、8のままなので、0にしておきます。
- **ビット3** - エッジトリガ/レベルトリガモードビット。デフォルトでは、エッジトリガになっていますので、0のままにしておきます。
- **ビット4** - 初期化ビット。1に設定
- **ビット5...7** - x86では未使用、0に設定されています。

上の図を見ると、最終的なビットパターンは**00010001**、つまり**0x11**になります。そこで、PICを初期化するために、ポート**0x20**にマッピングされているプライマリPICコントローレジスタに**0x11**を送ります。

カスケード接続を有効にしているので、ICW 3をコントローラにも送信する必要があります。また、ビット0を設定しているので、ICW4も送信する必要があります。これらについては後ほど説明します。とりあえず、ICW 2を見てみましょう。

```
 ; SetupでプライマリPICを初期化します。Send ICW 1
mov al, 0x11
out 0x20, al
```

このコントロールワードは、PICが使用するIVTのベースアドレスをマッピングするために使用されます。**これは重要です。**
2つのPICがあることを忘れないでください。この2つ目のPICでカスケード接続しているので、ICW 1を2つ目のPICのコマンドレジスタに送る

初期化コントロールワード (ICW) 2		
ビット数	価値	説明
0-2	A8/A9/A10	MCS-80/85モード時のIVT用アドレスビットA8-A10。
3-7	a11(t3)/a12(t4)/a13(t5)/a14(t6)/a15(t7)	MCS-80/85モード時のIVT用アドレスビットA11~A15。 80x86モードでは、割込みベクターアドレスを指定します。 x86モードでは0に設定してもよい。

初期化の際には、ICW2をPICに送り、使用するIRQのベースアドレスを伝える必要があります。ICW1をPICに送った場合（初期化ビットが設定されている場合）、次にICW2を送る必要があります。**そうしないと、定義されていない結果になることがあります。**誤った割り込みハンドラが実行される可能性が高くなります。

PICのデータ・レジスタに置かれるICW 1とは異なり、ICW 2はプライマリPICのソフトウェア・ポート**0x21**、セカンダリPICのポート**0xA1**として、データ・レジスタに送られます。（PICソフトウェア・ポートの完全なリストについては、「**8259Aソフトウェア・ポート・マップ**」の表を参照してください。）

さて、先ほどICW 1を両方のPICに送ったと仮定して（上のセクションを参照）、ICW 2を両方のPICに送ってみましょう。これでベースIRQアドレスが両PICにマッピングされます。

これは非常にシンプルですが、PICをどこにマッピングするかに注意しなければなりません。最初の31個の割り込み（**0x0-0x1F**）は予約されていることを覚えておいてください（上記の**x8割り込みベクターテーブル (IVT)** 表を参照）。そのため、これらのIRQ番号を使用しないようにしなければなりません。

最初の8つのIRQはプライマリPICが処理するので、プライマリPICをベースアドレスの**0x20**（32進数）に、セカンダリPICを**0x28**（40進数）にマッピングします。各PICには8つのIRQがあることを覚えておいてください。

```
ICW 2をプライマリPICに送信
ム      al, 0x20      プライマリPICではIRQ0~7を処理していました。IRQ 0は割込み番号0x20にマッピングされました
一      0x21, al      。
ブ
セカンダリコントローラにICW 2を送信
ウ      al, 0x28      セカンダリPICはIRQの8~15を処理します。IRQ8は割り込み0x28を使用するようにマッピングされま
ト      0xA1, al      した。
ア
ウ
ト
```

簡単でしょう？次の作品に期待しましょう

ICW 3

これは重要なコマンドワードです。PICが相互に通信する際、どのIRQラインを使用するかを知らせるために使用されます。

ICW 3 プライマリPIC用コマンドワード

初期化コントロールワード (ICW) 3 - プライマリPIC		
ビット数	価値	説明
0-7	S0-S7	スレーブPICに接続するIRQ (Interrupt Request) を指定します。

ICW 3 セカンダリPIC用コマンドワード

初期化コントロールワード (ICW) 3 - セカンダリPIC		
ビット数	価値	説明
0-2	ID0	マスターPICが接続に使用するIRQ番号 (バイナリ表記で
3-7	0	予約済み、0でなければならない

ICW1の中でカスケードを有効にする時は必ず**ICW3**を送らなければなりません。これにより、どのIRQを使って通信するかを設定することができます。**8259A**マイコンは、他のPICデバイスに接続するために**IR0-IR7**ピンに依存していることを覚えておいてください。これにより、**CAS0-CAS2**ピンを使用して相互に通信します。

各PICに互いの情報と、どのように接続されているかを知らせる必要があります。このためには、マスターと関連するPICの両方で使用するIRQラインを含むICW 3を両方のPICに送信します。

覚えておってください。80x86アーキテクチャでは、IRQライン2を使用してマスターPICとスレーブPICを接続します。

これを知った上で、両方のPICのデータレジスタに書き込む必要があることを忘れずに、上図のフォーマットに従う必要があります。なお、プライマリPIC

のICW 3では、各ビットが割り込み要求を表しています。つまり...

ICW 2（プライマリPIC）のIRQライン	
ビット数	IRQライン
0	IR0
1	IR1
2	IR2
3	IR3
4	IR4
5	IR5
6	IR6
7	IR7

IRQ2はICW3のビット2ですので、IRQ2を設定するためにはビット2（0100バイナリ、つまり0x4）を設定する必要があります。ICW 3

をプライマリPICに送信する例を示します。

```
ICW 3をプライマリPICに送る
mov al, 0x4          ; 0x4 = 0100 2ビット目 (IR Line 2)
out 0x21, al         ; プライマリPICのデータレジスタへの書き込み
```

これをセカンダリPICに送るには、2進法で送ることを覚えておかなければなりません。上の表を参照してください。IRQラインを表すには、ビット0〜2だけが使われていることに注意してください。バイナリー表記にすることで、8つのIRQラインを参照して選択することができます。

ICW 2（セカンダリPIC）のIRQライン	
バイナリー	IRQライン
000	IR0
001	IR1
010	IR2
011	IR3
100	IR4
101	IR5
110	IR6
111	IR7

十分にシンプルです。上の表では、2進法<->12進法の会話に従っているだけであることに注意してください。

IRQライン2で接続されているため、ビット1を使用する必要があります（上図参照）。

ここでは、プライマリとセカンダリの両方のPICコントローラにICW 2を送信する完全な例を示します。

```
ICW 3をプライマリPICに送る
mov al, 0x4          ; 0x04 => 0100, 2ビット目 (IRライン2)
out 0x21, al         ; プライマリPICのデータレジスタへの書き込み

ICW 3を2次PICに送る
mov al, 0x2          ; 010 => IRライン2
out 0xA1, al         ; 2次PICのデータレジスタへの書き込み
```

それが全てです。）

さて、これで両方のPICがIRQライン2を使って通信するように接続されました。また、両方のPICが使用するベース割り込み番号を設定しました。

これは素晴らしいことですが、まだ終わりではありません。ICW1をビルドアップする際、ビット0がセットされていれば、PICはICW4を送ることを期待していることを覚えておいてください。そのため、最終的なICWであるICW4をPICに送る必要があります。

ICW 4

イエーイ!これは、最終的な初期化コントロールワードです。これは、すべての動作を制御します。

初期化コントロールワード（ICW） 4		
ビット数	値	説明
0	UPM	セット(1)の場合、80x86モードである。MCS-80/86モードの場合はクリアされます。
1	AEOI	セットされていると、最後のインタラプトアクノリッジパルスで、コントローラは自動的にEOI（End of Interrupt）動作を行います。
2	M/S	BUFが設定されている場合のみ使用。1に設定されている場合、バッファーマスターが選択されます。バッファースレーブの場合はクリアされます。
3	BUF	設定されている場合、コントローラはバッファードモードで動作します。
4	SFNM	特別なフルネストモード。カスケード接続されたコントローラが多数存在するシステムで使用されます。
5-7	0	予約済み、0でなければならない

これはかなり強力な関数です。5〜7ビット目は常に0なので、他のビットやピースに注目してみましょう。

PICは、80x86が登場する以前から、汎用的なマイクロコントローラとして設計されてきました。そのため、システムに合わせてさまざまな動作モードが用意されていますが、その中

2021/11/15 13:13

オペレーティングシステム開発シリーズ

のひとつに「**特殊フルネストモード**」というものがあります。

ズ

x86ファミリーはこのモードをサポートしていないので、ビット**4**を**0**に設定しておくで安心です。

ビット3はバッファードモードに使用します。動作モードについては後で説明するのではなく、とりあえず0にしておきます。ビット2は、ビット3が設定されているときにのみ使用されるので、これを0に設定します。これにより、ビット1もほとんど使用されません。

そのため、ビット0を設定するだけで、PICの80x86モードが有効になります。簡単

ですね。つまり、ICW 4を送るために必要なのは、次のようになります。

```
mov     アル、          ビット0で80x86モードに
abl     1
; ICW 4をプライマリとセカンダリの両方のPICに送信
out0x21, al
アウト0xA1、アル
```

これは、このチュートリアルの中で最も簡単なコードスニペットでしょう。使えるうちに使ってみてください。:)

PICを初期化する - まとめてみる

信じられないかもしれませんが、これについてはすでに説明しました。PICの初期化では、正しいICWをPICに送るだけでいいのです。

ここでは、前のセクションのすべてをまとめてPICを初期化し、すべてがどのように組み合わせられているかを理解しましょう。

```
;*****
8259A PICで割り込みテーブルの32-47を使用するようにマッピングします。
;*****

#define ICW_1 0x11; 00010001 バイナリです初期化モードを有効にして、ICW 4を送信しています

#define PIC_1_CTRL 0x20; プライマリPIC制御レジスタ
#define PIC_2_CTRL 0xA0; セカンダリPICコントロールレジスタ

#define PIC_1_DATA 0x21; プライマリPICデータレジスタ
#define PIC_2_DATA 0xA1; セカンダリPICデータレジスタ

#define IRQ_0 0x20; IRQ 0-7 は 0x20-0x27 の割り込みを使用するようにマップされてい
#define IRQ_8 0x28; IRQ8-15は0x28-0x36の割り込み

使用するようにマップされている MapPIC:

; Send ICW 1 - Begin initialization -----

; SetupでプライマリPICを初期化します。ICW 1を送信

    mov al, ICW_1
    out PIC_1_CTRL, al

; Send ICW 2 - Map IRQ base interrupt number -----

    2つのPICがあることを忘れないでください。この2つ目のPICでカスケード接続しているので、ICW 1を2つ目のPICのコマンドレジスタ    out PIC_2_CTRL,
    alに送ります。

    ICW 2をプライマリPICに送信

        mov al, IRQ_0
        out PIC_1_DATA, al

    ICW 2をセカンダリ・コントローラに送信

        IRQ_8に移動
        out PIC_2_DATA, al

; Send ICW 3 - 両方のPICを接続するためにIRラインを設定する -----

    ICW 3をプライマリPICに送る

    mov ab al, 0x4                0x04 => 0100, 2ビット目 (IRライン2)
    le アウト PIC_1_DATA, アル    プライマリPICのデータレジスタへの書き込み

    ICW 3を2次PICに送る

    mov ab al, 0x2    ----- 010=> IRライン2
    le アウト PIC_2_DATA, al    2次PICのデータレジスタへの書き込み

ICWの送信 4 - x86モードの設定

                                mov al, 1; ビット0は80x86モードを有効にする

    ICW 4をプライマリとセカンダリの両PICに送信

        out PIC_1_DATA, al
        out PIC_2_DATA, al

; All done. データレジスタをヌル化

    mov al, 0
    out PIC_1_DATA, al
    out PIC_2_DATA, al
```

それほど難しくなかったでしょう？このコードですべてをカバーしました。

これでPICの初期化が完了しました。ハードウェア割り込みが発生すると、あらかじめIVT (Interrupt Vector Table) のどこかに定義しておいた32~47番の割り込みが呼び出されます。これにより、ハードウェア割り込みを追跡することができます。カッコいいでしょう？

オペレーション・コマンド・ワード (OCW)

うれしいですね。さて、醜い初期化作業が終わったところで、いよいよPICの標準的な制御と操作に集中することができます。そのためには、**OCW（Operation Control Words）**を使って、さまざまなレジスタの書き込みや読み出しを行います。

OCW 1

OCW 1は、**IMR（Interrupt Mask Register）**内の値を表す。現在のOCW 1を知るには、IMRから読めばよい。

IMRは、ステータスレジスタと同じポートにマッピングされていることを覚えておいてください。ステータスレジスタはリードオンリーなので、PICではリードかライトかでアクセスするレジスタを決定しています。

前述のPICレジスタの説明の際にIMRレジスタを見ました。

OCW 2

これは、PICを制御するための主要なコントロールワードです。それでは見てみましょう。

オペレーション・コマンド・ワード（OCW） 2		
ビット数	価値	説明
0~2	L0/L1/L2	コントローラが反応すべきインタラプトレベル
3~4	0	予約済み、0でなければならない
5	EOI	EOI（End of Interrupt）リクエスト
6	SL	セクション
7	R	回転オプション

じゃあ、いいか！？ビット0～2は、現在の割り込みの割り込みレベルを表します。3-4ビットは予約済みです。5-7ビットは、興味深いビットです。これらのビットの各組み合わせを見てみましょう。

OCW2コマンド			
Rビット	SLビット	EOIビット	説明
0	0	0	自動EOIモードでの回転（CLEAR）
0	0	1	非特定の EOI コマンド
0	1	0	操作なし
0	1	1	特定のEOIコマンド
1	0	0	自動EOIモード（SET）での回転
1	0	1	特定できないEOIでローデート
1	1	0	優先順位設定コマンド
1	1	1	特定のEOIに基づいて回転する

さて...この表、今のままでは分かりにくいと思いませんか？上記のコマンドの多くはかなり高度なものです。では、どんなことができるのか見てみましょう。

EOI（End of Interrupt）の送信

ご存知のように、ハードウェア割り込みが発生すると、プライマリコントローラにEOI信号が送られるまで、他のすべての割り込みは**割り込みマスクレジスタ**内でマスクされます。つまり、**割り込みルーチン（IR）**の最後に、すべてのハードウェア割り込みが有効になるようにEOIを送信する必要があります。

上の表を見ると、コントローラにEOIの信号を送るために、特定ではないEOIコマンドを送ることができます。EOIビットはOCW2のビット5なので、ビット5（100000バイナリ＝0x20）を設定するだけでよい。

```
EOIをプライマリPICに送る

movab    a1, 0x20          OCW2のビット4を設定
le       アウト    0x20, a1    プライマリPICコマンドレジスタへの書き込み
```

結論

PICはプログラムが複雑なマイクロコントローラーです。このチュートリアルでは多くのことを説明してきましたが、さらに難しくなってきました。

うまく説明できたでしょうか？OS開発シリーズのプライマリ・チュートリアル・シリーズでは、このチュートリアル内のすべての内容を、あるべき場所に配置します。これは、割り込みの設定、割り込み処理、ハードウェア割り込みの間の接着剤となるでしょう;)

このチュートリアルは、より多くの内容を提供し、8259Aマイクロコントローラに関するすべての詳細を説明できるように拡張する予定です。

このチュートリアルは、本編が使用するサイドチュートリアルです。そのため、このチュートリアルのためのデモはありません。しかし、すべてをまとめるために、デモを作ることになるかもしれません。ただし、そのためにはIDTと割り込み処理を詳しく説明する必要がありますが、これはPICには関係ありません。直接的には関係ありませんが。

このチュートリアルが、PICをプログラミングする際の疑問や、ボンネットの中で実際に何が起きているのかを理解する助けになれば幸いです。それではまた次回、お会いしましょう。

マイク
BrokenThorn Entertainment社。現在、DoEとNeptune Operating Systemを開発中です。 質問

やコメントはありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ私に教えてください。

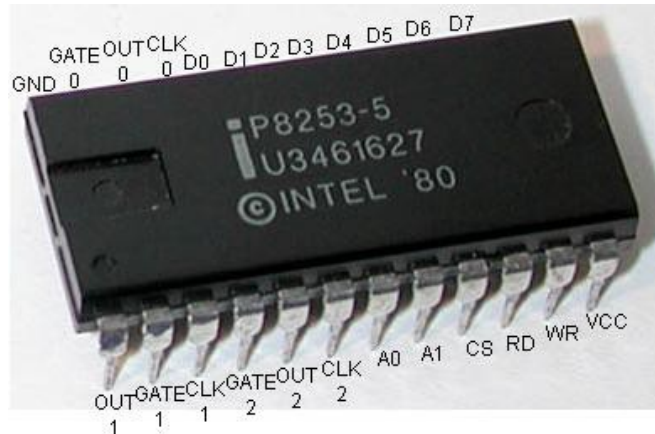


オペレーティングシステム開発シリーズ

オペレーティング・システムの開発 - 8253プログラマブル・インターバル・タイマ

by Mike, 2008

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。



8253 PIT マイクロコントローラ (全ピン表示付き)

ご注意：このチュートリアルでは、ハードウェア割り込み処理や8259 Programmable Interrupt Controller (PIC)に関する知識が必要になる場合があります。これらの情報については、[このチュートリアル](#)をご覧ください。

はじめに

歓迎します。:)

このチュートリアルでは、システム・タイミングとIntel 8253 Programmable Interval Timer (PIT)のプログラミングについて知りたいことをすべて説明します。

8253 PITには長い歴史があり、ほぼすべてのx86 PCで重要な役割を果たしています。これは「システムクロック」と呼ばれ、PCの中で非常に重要な機能を担っています。この...「このチップは、もはや独立したチップとして（正確にはDIP (Dual Inline Package) として）流通しておらず、むしろマザーボードのサウスブリッジ・チップセットに組み込まれています。

しかし、8253のすべてが残っています。そのため、入出力設備、ハードウェア、そして8253のプログラミング方法も同じです。このDIPと旧式のDIPの間には、速度以外の違いはないので、ここではシンプルにするために旧式の8253のDIPを見ていきます。

このチュートリアルの冒頭の写真は、これから見ていくもの、プログラミングするものを表示しています。

楽しくやりましょう。)

プログラマブル・インターバル・タイマー

プログラマブル・インターバル・タイマー(PIT)は、プログラムされたカウントに達すると、割り込みを発生させるカウンタです。8253および8254マイクロコントローラは、i86アーキテクチャに対応したPITで、i86対応システムのタイマとして使用されます。

これらのPICには、異なる目的で使用する3つのタイマーが含まれています。最初のタイマーは通常、システムクロックとして使用されています。タイマー2はRAMのリフレッシュに使われ、タイマー3はPCのスピーカーに接続されています。すべての接続を見るのはもう少し後になりますので、今はあまり詳しく説明しません。

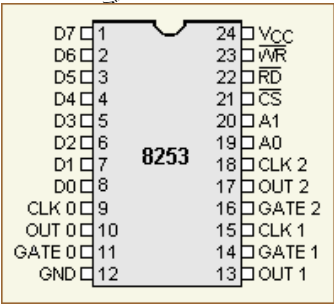
ここでは、そんな有名なPITのひとつをご紹介します...。8253マイクロコントローラ。

8253ハードウェア

ソフト面を見る前に、実際に何をプログラムしているのかを知っておくと便利です。そこで今回は、まず8253のハードウェアを取り上げ、その動作とPCの他の部分との接続について学びます。また、ソフトウェア側で必要となる内部レジスタ、ピン配置、コマンドワードなどについても見ていきます。

8253ハードウェア。説明

8253 PITはシンプルなインターフェイスで、プログラミングもそれほど難しくありません。



そうです。私たちがプログラムするチップがあるのです。

これがピンの完全なレイアウトです。このチュートリアルでは、これらのピンを参照しますので、ピンが何であるかを知っておくことが重要です。

- **D0...D7** : 8ビットのデータライン。データバスに接続されているので、コマンドの読み出しや送信が可能です。
- **CLK 0, CLK 1, CLK 2**: クロック入力端子。3つの独立したカウンター用の3端子です。
- **OUT 0, OUT 1, OUT 2** : 出力データラインです。 **GATE 0, GATE 1, GATE 2**: ゲートデータライン。3つの独立したカウンター用の3つの端子があります。
- **Vcc** : 入力電圧
- **WR** : ライトイネーブル。このラインがアクティブになると、8253にデータを書き込んでい
- ることを知らせます。 **RD**: リードイネーブル。この信号がアクティブになると、8253はデ
- ータを読み出すことができる **CS**: チップセレクト信号
- **A0, A1** : アドレスライン。どのレジスタにアクセスしているかを判断するために

使用する。悪くないですね。ここには見ておくべき重要なピンがいくつかある。

D0〜D7ピンは、システムのデータバスに接続します。これらのピンは、コントローラにデータを送信または読み取る際に、データを運びま

す。

Vccと**GND**で回路が完成します（電圧入力、グランド出力）。

WRピンは、コントローラに書き込み中であることを伝えます（データピンへの入力を期待します）。このピンの信号が "Low "のとき、現在データを送信しています。 **RD**端子も同様ですが、データを読み込んでいることをコントローラに伝えます。 **CS**ピンは、コントローラが**RD**ピンと**WR**ピンに対して何をすべきかを決定する特別なピンです。 **CS**ピンが "Low "であれば、コントローラはRDとWRピンに反応します。 **CS**が "Low "でない場合、それらは無視されます。 **WR**と**RD**はシステムコントロールバスに接続します。 **CS**ピンは、ポートi/o操作のためにシステム・アドレス・バスに接続します。

A0, A1ピンはシステムアドレスバスに接続されており、どのレジスタにアクセスしているかを判断するために使用されます。 **A0, A1**ピンはシステムアドレスバスに接続されており、どのレジスタにアクセスしているかを判断するために使用されます。

CLK端子、**OUT**端子、**GATE**端子の3つのグループがあることに注目してください。これにはちゃんと理由があります。 **8253/8254**マイコンには、3つの独立したタイマーが搭載されています。

この仕組みをもう少し詳しく見てみましょう。

8253のハードウェア。カウンター

8253は、3つのカウンタで構成されています。カウンタ0、カウンタ1、カウンタ2です。各カウンタには2つの入力端子があります。 **CLK**(クロック入力)と**GATE**の2つの端子と、出力用の1つの端子--**OUT**。

カウンタは3つあるので、システム内でそれぞれ異なる目的で使用されます。各カウンタは16ビットのダウンカウンターです。

一般的なコンピュータでは、第1タイマの**OUT**ピンを**PIC (Programmable Interrupt Controller)** に接続して、クロックの刻みごとに割り込みを発生させます。これは通常、**システムタイマー**として使用されます。2番目のカウンタは、**DRAM**メモリをリフレッシュするためにメモリコントローラにタイミング信号を生成するために使用された。3つ目のカウンタは、PCのスピーカーに音を出すために使用されます。

お察しの通り、**PIT**はカウンタが0になると**OUT**ピンを使ってこれらのデバイスに信号を送ります。 **PIT**のカウンタが0になると、単純に折り返して再スタートします。

CLKは、タイマーのクロック入力です。現在の動作モードに応じて、**GATE**端子と併用することができます。以下の表は、**GATE**に流れる電流がLow, Rising, Highのいずれかである場合の動作を示しています。

GATE 入力端子の動作			
モード	低い、または低くなる	ライジング	高
0	カウントダウンの無効化	-	カウントを有効にする
1	-	カウントを開始し、次のCLKの後にOUTをリセットする。	-
2	カウントを無効にし、OUTをHighにする	カウンターを再投入し、カウントを開始する	カウントを有効にする
3	カウントを無効にし、OUTをHighにする	カウント開始	カウントを有効にする

4	カウントダウンの無効化	-	ズ	カウントを有効にする
5	-	カウント開始	-	-

8253のカウンターはチャンネルと呼ばれています。8253/8254 PITには3つのチャンネルがありますが、それぞれのチャンネルについてもう少し詳しく見てみましょう...

チャンネル0

チャンネル0は、**8259 PIC**に接続され、**割り込み要求 (IRQ)** を生成します。PITs **OUT**ピンはPICの**IRO**ピンに接続します。通常、BIOSはこのチャンネルを**65536**カウントで構成し、出力周波数は**18.2065Hz**になります。これにより、**54.9254ms**ごとに**IRQ 0**が発生します。

ほとんどのx86マシンで使用されている主要なタイマーです。クロックレート（カウンター0の**CLK**ピンで通知される）は**1193181.6666...Hz**です。Hzで、これは**NTSC**サブキャリア周波数の3分の1にあたります。これは、旧来の**CGA PC**との互換性を保つために必要なものである。

多くのシステムでは、チャンネル0が**システムクロック**として動作するようにプログラムされています。これは、チャンネル0の**OUT**ピンが**PIC**の**IRO**ラインに間接的に接続されているために可能です。設定するモードに応じて、タイマーを適切な周波数に設定し、**PIC**の**IRO**ラインを一定の割合で有効にすることができます。その後、リセットされて、また最初からやり直します。**PIC**は**ハードウェア割り込み**を扱うため、まず**PIC**を再プログラムする必要があります。

また、最も番号の小さい割り込み（**IRQ0**）に接続されているため、他のハードウェア割り込みよりも優先順位が最も高くなっています。

最も低い周波数は、昔の**DOS**システムを搭載したコンピュータに使われていたもので、約**18.2Hz**です。最高周波数は**1メガヘルツ**強。

リアルモードのOSでは、BIOSは通常、**IRQ0**の発火回数を **0000:046C** にインクリメントし、実行中のプログラムがこれを読み取ることができます。

チャンネル1

多くのビデオカードやBIOSは、独自の用途のために第2チャンネルを再プログラムすることがあります。このチャンネルは元々、メモリコントローラが**DRAM**メモリをリフレッシュするためのタイミングパルス信号を生成するために使用されていました。現代では、メモリコントローラがリフレッシュを行うため、この信号は必要ありません。このため、どのような機器でこのカウンターが使用されるかは保証されていません。

チャンネル2

このチャンネルは、**PCスピーカー**に接続して音を発生させます。**PCスピーカー**は通常、2つのレベルの出力を持つ矩形波を生成するようになっています。しかし、真の意味で定義された2つの矩形波レベルの間を行き来することが可能です。これを**PWM(Pulse-Width Modulation)**といいます。

このチャンネルの設定は、モード3にプログラムし、トーンの周波数レートを設定します。

PCスピーカーを直接プログラムすることもできます。[チュートリアル7](#)を振り返ると、PCスピーカーがポート**0x61**にマッピングされていることがわかります。このポートは、スピーカーがどのように動作するかを定義します。

- **ビット0**：(1) に設定すると、スピーカーの状態はビット1に従う
- **ビット1**：設定されている場合 (1)、スピーカーは**PIT**を使用し、設定されていない場合 (0)、スピーカーは**PIT**との接続を無効にする

ビット0がセットされていれば、残りのバイトには音の周波数を表すビットのパターンが含まれています。スピーカーから最大**8ビット**の音を発生させることができますが、これはちょっとカッコいいですね。

また、**PIT**を使用しないようにデバイスを無効化できることにも注目してください。起動時に、BIOSはスピーカーが**PIT**チャンネル2を使用し、モード3で動作するように設定します。タイミングの問題が発生する可能性があるため、スピーカーは**PIT**を使用するようにしておくことをお勧めします。

念のため、例を挙げておきます。

結論 スピーカーを無効にして、チャンネル2の使用を停止する

```
movdx, 0x61
```

一度設定された**カウンタ**は、**他の**コントロールワードで変更されるまで、その状態が維持されます。

この**カウンタ**を使って、何かを発生させることができそうですね。チャンネル1はもう使われていないので、安全だと思って自分で使うことはできません。そのため、**チャンネル0と2を使用**することをお勧めします。

チャンネル0を使って、割り込みハンドラを起動することができます。割り込みハンドラは、カーネルが使用するカウンタをインクリメントすることができます。この特別な小さなカウンタ変数は、システムにおいて非常に重要な役割を果たしています。システムタイマーです。これらのことはすぐにわかるでしょう。)

さて、ここまではピンの構成と、異なるデバイスで使用される3つのタイマーについて見てきました。次は何ですか？

これらのタイマーをプログラミングする際には、初期化を行う必要があります。各チャンネルは**6つの異なるモード**をサポートしていることを覚えておいてください。これらのモードの中には非常に便利なものもありますが、そうでないものもあります。他のモードはそうではありません。これらのモードを理解するために、それぞれのモードを見てみましょう。少し詳しくなりますが、すでにご存じの方、あるいは期待されている方もいらっしゃると思いますので、ご了承ください。)

8253のチャンネルモード

各カウンタは、6つのモードのうち1つにプログラムできることを覚えておいてください。これを行うには、コントローラに**初期化制御ワード (ICW)**を送信します。このコマンドワードのフォーマットについては後ほど説明します。ここでは、各モードについて説明します。

モード0：端子カウントでのインタラプト

このモードでは、カウンタは最初の**COUNT**値にプログラムされ、その後、入力クロック周波数 (**CLK**信号) に合わせてカウントダウンしていきます。**COUNT**が0のとき、**コントロールワード**が書き込まれた後、カウンタは**OUT**ピンをイネーブルにして (ラインをハイにして)、接続されているデバイスに信号を送ります。カウントは、**COUNT**がプログラムされた1クロックサイクル後に開始されます。**OUT**ラインは、カウンタが新しい値または同じ値で再ロードされるか、別のコントロールワードがコントローラに書き込まれるまで、ハイレベルのままです。

このモードは基本的に、0までカウントダウンするタイマーを設定することができます。その後、新しいカウント番号を再ロードするか、カウンタを再初期化するために新しいコントロールワードを再ロードする必要があります。

モード1：ハードウェアトリガーによるワンショット

このモードでは、カウンタは一定のクロックパルス数ごとに出力パルスを与えるようにプログラムされています。**コントロールワード**が書き込まれると、すぐに**OUT**ラインがハイレベルになります。**COUNT**が書き込まれた後、カウンタは**GATE**入力の立ち上がりエッジまで待機します。パルス出力中にトリガが発生した場合、8253は再びトリガされます。**GATE**の立ち上がりエッジが検出されてから1クロック後に**OUT**はLOWになり、**COUNT**が0になるまでLOWを維持します。その後、**OUT**は次のトリガがかかるまでHIGHになり、**GATE**入力の立ち上がりエッジが検出されるまで再び待機します。

モード2：レートジェネレーター

このモードでは、カウンタを "**divide by n** "カウンタに設定します。このカウンタは、一般的にリアルタイムシステムクロックの生成に使用されます。カウンタは、初期の**COUNT**値にプログラムされます。カウントは次のクロックサイクルで開始されます。**OUT**は、**COUNT**が0になるまでHighのままです。は1に達します。その後、1クロックパルスの間、**OUT**はLOWになります。その後、**OUT**は再びHighになり、**COUNT**は初期値にリセットされます。このプロセスは、新しい制御ワードがコントローラに送られるまで繰り返されます。

ハイパルス間の時間は、「**COUNT**」の現在値に依存し、次の式で算出されます。

$$\text{COUNT} = \text{入力 (Hz)} / \text{出力の周波数}$$

COUNTが0になることはなく、nから1までの範囲でしかありません (nは**COUNT**の初期値)。

さて、ちょっと立ち止まってみましょう。カウンタ0がPICに接続されていることを覚えていますか？**カウンタ0のOUTラインは、間接的にPICのIROラインに接続されています**。これを知っていると、IROラインがLowの時、PICは我々が定義したIRQ 0ハンドラを呼び出します。

カウンタをモード2に設定すれば、一定の割合で割り込みが発生するようにタイマーを設定することができます。あとは、上の式をもとに**COUNT**の値を決めるだけです。これは、OSの**システムタイマー**の設定によく使われます。結局のところ、IRQ 0は、定義した周波数レートで、クロックティックごとに呼び出されることになります。

確かに、モード2は重要なモードですね。

モード3：スクウェア・ウェーブ・ジェネレータ

このモードは、モード2とよく似ています。ただし、**OUT**は期間の半分はHigh、残りの半分はLowになります。**COUNT**が奇数の場合、**OUT**は $(n+1)/2$ カウントの間ハイになります。**COUNT**が偶数の場合は、 $(n-1)/2$ カウントの間、**OUT**はLowになります。

それ以外はモード2と同じです。**COUNT**の初期値を設定するには、モード2の計算式を使う必要があります。スピーカーがPITを使用するように設定されている場合は、通常使用するチャンネルをこのモードを使用するように設定する必要があります。

モード4：ソフトウェア・トリガ・ストローブ

カウンタは、初期の**COUNT**値にプログラムされています。カウントは次のクロックサイクルで開始されます。**OUT**は、**COUNT**の値が大きくなるまでHighのままです。**COUNT**が0になると、カウンタは1クロックの間、**OUT**をLowにします。その後、再び**OUT**をHighにリセットします。

モード5：ハードウェア・トリガ・ストローブ

カウンタは、初期の**COUNT**値にプログラムされています。**OUT**は、コントローラが**GATE**入力の立ち上がりエッジを検出するまでHighのままです。これが起こると、カウントが開始されます。**COUNT**が0になると、1クロックサイクルの間、**OUT**はLOWになります。その後、**OUT**は再びハイレベルになります。このサイクルは、コントローラが**GATE**の次の立ち上がりエッジを検出するまで繰り返されます。

8253レジスタ

8253には、アクセスできるレジスタがいくつかあります。これらのレジスタのほとんどは互いによく似ているので、わかりやすくするために同じ表にまとめておきます。この表は、8253の対応するラインがアクティブなときの各レジスタとその機能を示しています。**RD**ラインと**WR**ラインが読み出しと書き込みの動作を決定することに注目してください。また、**A0**と**A1**のラインが、どのレジスタにアクセスしているかを決定していることにも注目してください。

[チュートリアル7](#)のポートテーブルを見ると、**システムタイマ**がBIOSによってポート0x40-0x4Fを使用するようにマッピングされていることがわ

8253 PIT内部レジスタ						
レジスタ名	ポートアドレス	RDライン	WRライン	A0ライン	A1ライン	機能

カウンター0	0x40	1	0	0	0	ロードカウンタ0
		0	1	0	0	カウンタ0の読み出し
カウンター1	0x41	1	0	0	1	ロードカウンタ1
		0	1	0	1	カウンタ1の読み込み
カウンター2	0x42	1	0	1	0	ロードカウンタ2
		0	1	1	0	カウンタ2の読み込み
コントロールワード	0x43	1	0	1	1	ライトコントロールワード
NA		0	1	1	1	操作なし

それ以外の0x44～0x4fのポートアドレスは未定義です。

システムは、実行している操作に応じて、適切なラインをアクティブにします。カウンタレジスタを設定する際には、まずどのようにロードするかをコントローラに知らせる必要があります。これは、最初にコントロールワードを設定することで行われます。それでは、これらのレジスターを詳しく見てみましょう...

カウンターレジスター

各カウンタレジスタには、PITがカウントダウンに使用する**COUNT**値が格納されています。これらはすべて16ビットのレジスタです。これらのレジスタに書き込みや読み出しを行う場合は、まずPITに制御ワードを送る必要があります。なぜそれを直接できないのかと思われるかもしれません。これには理由があって、データの大きさに関係しています。PITには8本のデータライン（ピンD0～D7）しかありません。しかし、カウンターレジスターは8ビットではなく、すべて16ビットです。

このため、PITはカウンタレジスタに書き込まれたデータをどのようにして知るのでしょくか？カウンタレジスタの16ビットの中のどのバイトを設定しているのか、どうやって知っているのでしょうか？そうでは**ありません**。コマンド・ワードを送信することで、PITにデータが入ってくること、そしてそのデータをどうするかを知らせることができます。これについては次に説明します。

コントロール・ワード・レジスタ

これは私たちにとって重要なことです。

このレジスタは、コントローラの動作モードを決定し、設定するために使用される重要なレジスタです。RD、A0、A1ラインをイネーブルにすることでアクセスできます。このレジスタは書き込みのみ可能で、読み出しはできません。

コントロール・ワード・レジスタは、シンプルなフォーマットを使用しています。最初は表にしようかと思っていたのですが、リスト形式の方が簡単かもしれないので、ここで紹介します。

- **ビット0：(BCP)** バイナリカウンタ
 - **0**：バイナリ
 - **1**：BCD（バイナリーコード化された10進法）
- **ビット1～3：(M0、M1、M2)** 動作モード。それぞれの説明は上記のセクションを参照してください。
 - **000**：モード0：インタラプトまたはターミナルカウント
 - **001**：モード1：プログラマブルワンショット
 - **010**：モード2：レートジェネレータ
 - **011**：モード3：矩形波ジェネレータ
 - **100**：モード4：ソフトウェア・トリガ・ストローブ
 - **101**：モード5：ハードウェア・トリガ・ストローブ
 - **110**：未定義、使用しないでください
 - **111**：未定義、使用しない
- **ビット4-5：(RL0、RL1)** リード／ロードモード。カウンタレジスタにデータを読み込んだり、送信したりします
 - **00**：カウンタ値は、I/O書き込み動作時に内部制御レジスタにラッチされます。
 - **01**：最下位バイト（LSB）のみ読み出しまたは読み込み
 - **10**：最上位バイト（MSB）のみ読み出しまたは読み込み
 - **11**：LSBの次にMSBを読み出しまたは読み込み
- **ビット6-7：(SC0-SC1)** セレクトカウンター。それぞれの説明は上記のセクションを参照してください。
 - **00**：カウンタ0
 - **01**：カウンタ1
 - **10**：カウンタ2
 - **11**：不正な値

さて、それでは！ちょっとしたことをやってみましょうか。あとは、コントロール・ワード・レジスタに書き込んで、コントロール・ワードを構築し、コントローラを初期化するだけですわね？もちろんだよ。そうだな...

基本的には、特定の目的のために**カウンター**を初期化したい。そのため、カウンター、カウンタのカウンタモード、動作モードを設定するためのコントロールワードを構築する必要があります。その後、カウンター自体を初期化します。一度初期化すると、カウンターは（そのモードに応じて）それ自体を継続することができるので、この作業は一度だけ行う必要があることを覚えておいてください。

一例を挙げて、すべてをまとめてみましょうか。

すでにPICを初期化し、割り込み0のハンドラがあるとします。100Hz（10ミリ秒に1回）ごとにIRQ 0を発生させるタイマーをセットアップしたいとします。PITのチャンネル0がPICのIROラインに接続されていることがわかっているため、チャンネル0をプログラムしてこれを実行します。

COUNT = 入力hz / 周波数

```
mov    dx, 1193180 / 100    100hz、または10ミリ秒
abl
e
```

```
FIRST は PIT にコマンドワードを送信します。バイナリカウントを設定します。
モード3、チャンネル0のLSBを先に読み込んでからMSBを読み込む。

ム      al, 110110b
一      0x43, al

これでチャンネル0に書き込めるようになりました。LSBを先にロードしてからMSBをロードする」というビットを設定したので、つまり
ax, dx
アウト  0x40, al      ;LSB
xchg    あ、アル
アウト  0x40, al      ;MSB
```

最初にコントロールワードを設定し、次にカウンタ0レジスタに書き込んでいることに注目してください。

そうなんですか！？うん。以上により、カウンタ0が10ミリ秒ごとにIRQ0を起動するようにプログラムされます。

結論

8253と8254のPITは、非常に使い勝手の良い小さなチップです。様々なデバイスに使用でき、様々な目的に使用できます。

私たちが必要とするのは、PCのスピーカーからの信号出力と、最新のシステムソフトウェアでは非常に重要な機能であるシステムタイマーの両方を実現することです。さらに、スピーカーを直接操作する方法や、PITとの接続を無効にする方法も検討しましたが、これにはメリットとデメリットがあります。システムソフトウェアの設計者は、システムを開発する際に、さまざまな無限の可能性の中から長所と短所を判断し、それを両立させることができるのです。

ここまで、標準的なx86ベースのPCマザーボードに搭載されているPITの詳細、ピン配置、およびその接続について見てきました。最近のパソコンでは、PIT自体がマザーボードのサウスブリッジと一体化しています。

私は、チップ自体にさらに追加して、一般的なコンピュータシステムの中でチップがどのように接続されているかを正確に説明しようと考えています。いっそのこと、分解してしまってもいいかもしれません。

チュートリアル 16: Kernel:タイミングと例外処理では、[8259APIC](#)とこのチュートリアルのすべてをまとめます。両方のデバイスを実装し、インターフェースを作成します。もしかしたら、もうちょっと…。

Welcome back to kernel Land!

次の機会まで。

マイク

BrokenThorn Entertainment社。現在、DoEとNeptuneOperating Systemを開発中です。 質問やコメントはありますか？お気軽に[お問い合わせください](#)。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ[私に教えてください](#)。



オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - グラフィックス1

by Mike, 2010

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

Welcome!

待って、何？もうグラフィック？そうなんです、OSの超クールなGUIの開発を始めますよ。:)そうではありませんが、その方向に向けてのスタートです。

本章は、グラフィックスプログラミングを扱うミニシリーズの最初の章です。この章では、VESA VBE、Video BIOS、VGA用の直接ハードウェアプログラミング、そしておそらくSVGAのコンセプトを取り上げます。また、2Dベクトルレンダリングや画像など、グラフィックの概念やレンダリングについても取り上げる予定です。もしかしたら、もう少し後には3Dも扱うかもしれません。

興奮していますか？OS開発シリーズのスピンオフであるこのミニシリーズには、たくさんのクールな素材が登場します。しかし、素晴らしいコンピューターグラフィックスの世界に飛び込む前に、基本的なルールを決めておかなければなりません。一口にCGといっても、その扱い方や方向性はさまざまです。コンピューター・グラフィックスは複雑なテーマです。1つの章ではカバーできません。いや、できるんですよ。ただ、1つの.....とてもとても長い章になってしまうのですが。

そのため、これを段階的に行うことにしました。最初の章では、リアルモードやv86モードでのグラフィックの扱いについて説明します。システムBIOSの割り込みを使い、グラフィックスの基本的な概念を説明します。第2章では、「Video BIOS Extensions (VBE)」と「Super VGA」を取り上げます。第3章では、グラフィックス・パイプラインの直接ハードウェア・プログラミングを取り上げた、より小さなミニシリーズの第1章となります。第3章では、グラフィックス・パイプラインのダイレクト・ハードウェア・プログラミングを取り上げます。

この章では、リアルモードのVideo BIOSを使って、リアルモードのグラフィックスを操作することから始めましょう...

基本コンセプト

アブストラクト

コンピューター・グラフィックス (CG) は、今さら説明するまでもありません。コンピューター、アニメーション、ビデオゲーム業界に革命をもたらしました。コンピューターグラフィックスの分野は、コンピューターのディスプレイ上でグラフィック効果を生み出す能力の開発、創造、継続を網羅しています。1Dグラフィックスから、2D、3D、そして4Dグラフィックスのシミュレーションソフトまで。

歴史

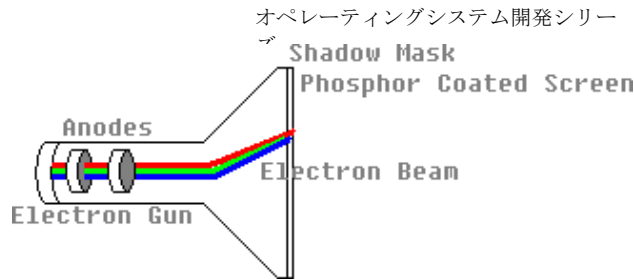
コンピューターグラフィックス産業は、1960年代の「ワールウィンド」のような初期のプロジェクトから生まれた。Whirlwindは、ビデオディスプレイ出力を使用した最初のコンピューターであり、CRT (Cathode Ray Tube) 技術の導入にも貢献した。Whirlwindは、最終的にSAGE (Air Force Semi Automatic Ground Environment) コンピューターシステムの開発につながった。ブラウン管は、1897年にフェルディナンド・ブラウンが開発した「ブラウン管」が最も古いものである。

Special Interest Group on GRAPHics and Interactive Techniques (SIGGRAPH) は、Association for Computing Machinery (ACM) のSIGGRAPHグループによって運営されています。1969年にAndy van Damによって始められたSIGGRAPHグループは、世界各地でSIGGRAPHカンファレンスを開催しています。これらのカンファレンスには、エンジニアリング、グラフィックス、映画、ビデオゲーム業界の企業から何千人ものプロフェッショナルが参加しています。

グラフィックス・ハードウェアの進歩により、より強力なグラフィックス・デザインが可能になった。液晶ディスプレイ (LCD) などの他のディスプレイ技術が登場すると、CRT技術の使用は減少していった。

VDT (Video Display Terminals) は、VDU (Video Display Unit) とも呼ばれる初期のディスプレイ端末。

CRT (Cathode Ray Tube) とは



アブストラクト

CRTは、電子銃と蛍光体ターゲットで構成された真空管である。ブラウン管の前面全体を**ラスタ**と呼ばれるパターンで繰り返し走査する。画像は、赤、緑、青の3色の電子ビームの強度をディスプレイ上の任意の位置で変化させることで生成される。電子ビームは、**シャドウマスク**層を通過した後、蛍光体を塗布したスクリーンに当たる。

問題点

CRTモニターは少量のX線を放出することがあります。また、ディスプレイを常に再スキャンしているため、**低リフレッシュレート**（60Hz以下）ではちらつきが見られることがあります。また、CRTには有害な蛍光体が含まれている場合があります。このため、米国環境保護庁（EPA）は、CRTを適切なリサイクル施設に買い取らなければならないという規則を作りました。最後に、CRTには真空のガラスが入っているため、外側のガラスが破損すると、CRTが破裂する可能性があります。これは、ガラスが危険な速度で外側に飛び散る原因となります。最近のCRTは、CRTの飛散を防ぐために一定の対策が施されている。

CRTの周波数をソフトウェアで制御することが可能です。周波数が高くなると、CRTを本来の用途よりも高速に動作させることが可能となり、CRTが破裂する可能性が高くなります。このため、**CRTコントローラー（CRTC）**の取り扱いには十分な注意が必要です。しかし、最近のCRTはこのようなことがないように保護されています。

VGA

アブストラクト

VGA（Video Graphics Array）は、1987年にIBMが発売したアナログコンピュータディスプレイ規格。アレイ」と呼ばれるのは、もともとMDA、CGA、EGAが使用していたISA（Industry Standard Architecture）ボードの数十個のロジックチップに代わる1つのチップとして開発されたからである。ISAボード1枚に収まっていたため、マザーボードへの接続が非常に容易でした。

VGAは、ビデオバッファ、ビデオDAC、CRTコントローラ、シーケンサユニット、グラフィックスコントローラ、属性コントローラで構成されています。これらの構成要素については、後の章で詳しく説明します。

ビデオバッファ

ビデオバッファは、ビデオメモリとしてマッピングされたメモリのセグメントです。メモリのどの領域がビデオメモリにマッピングされるかは変更できる。起動時には、BIOSは0xA0000にマッピングします。つまり、ビデオメモリは0xA0000にマッピングされます。（チュートリアル7のリアルモードアドレスマップを覚えていますか？）メモリマッピングについては、この章の少し後に詳しく説明します。

ビデオDAC

Video Digital to Analog Converter（DAC）には、ビデオデータをディスプレイに送るアナログビデオ信号に変換するためのカラーパレットが含まれています。この信号は、赤、緑、青の色の濃さをアナログで表したものです。詳しくは後で説明しますので、まだ理解できなくてもご安心ください。

CRTコントローラー

このコントローラは、水平・垂直同期信号のタイミング、ビデオバッファのアドレッシング、カーソルやアンダーラインのタイミングを生成する。詳しくは、後ほどVGAハードウェアの説明の際に説明します。

シーケンサー

シーケンサは、ビデオメモリの基本的なメモリタイミングと、再生バッファのフェッチを制御するためのキャラクタクロックを生成します。これにより、システムはアクティブなディスプレイのインターバル中にメモリにアクセスすることができます。もう一度言いますが、これについてはまだ詳しくは説明しません。

グラフィックスコントローラ

ビデオメモリとアトリビュートコントローラー、ビデオメモリとCPUの間のインターフェースである。表示がアクティブな時間帯には、ビデオバッファ（ビデオメモリ）からメモリデータが送られ、アトリビュートコントローラに送信される。グラフィックスモードでは、このデータをパラレルからシリアルビットプレーンデータに変換してから送信する。テキストモードでは、パラレルデータだけが送信されます。

まだ理解できなくても大丈夫です。ここでは、あまり詳しく説明するつもりはありません。後日、ビデオドライバーの開発について説明するときに、すべてを詳しく説明します。とりあえず、覚えておいてください。グラフィックスコントローラは、ビデオメモリからのパラレルデータをもとにディスプレイを更新します。これは、ディスプレイの使用時間に応じて自動的に行われます。つまり、ビデオメモリ（デフォルトは0xA0000にマッピングされています）に書き込むことで、現在のモードに応じて、実質的にビデオディスプレイに書き込むことになります。これは、文字を印刷するときに重要です。

グラフィックス・コントローラが使用するアドレス範囲を変更することが可能であることを覚えておいてください。初期化の際、BIOSはビデオメモリを0xA0000にマッピングするためにこれを行います。

ビデオモード

ビデオモード」とは、表示の仕様である。つまり、ビデオメモリがどのように参照され、そのデータがビデオアダプターでどのように表示されるかを記述したものです。

VGAは2種類のモードに対応しています。APAグラフィックス」と「テキスト」です。

APAグラフィックス

APA (All Points Addressable) とは、ビデオモニターやドットマトリックスなど、ピクセルアレイで構成されたデバイスにおいて、すべてのセルを個別に参照できる表示モードのことである。ビデオディスプレイの場合は、すべてのセルが「ピクセル」を表し、すべてのピクセルを直接操作することができます。そのため、ほとんどのグラフィックモードがこの方式を採用しています。このピクセルバッファを変更することで、画面上の個々のピクセルを効果的に変更することができます。

ピクセル

ピクセル」とは、ディスプレイ上で表現できる最小単位のこと。ディスプレイ上では、色の最小単位を表しています。つまり、基本的には1つのドットである。各ピクセルのサイズは、現在の解像度とビデオモードに大きく依存します。

テキストモード

テキストモードとは、APAのように、画面上のコンテンツを内部的にピクセルではなく文字で表現する表示モードです。

テキストモードを採用しているビデオコントローラでは、2つのバッファを使用します。1つは、表示される各文字のピクセルを表すキャラクターマップ、もう1つは、各セルにどのような文字があるかを表すバッファです。文字マップバッファを変更することで、文字そのものを変更することができ、新しい文字セットを作成することができます。また、各セルにどのような文字が入っているかを表すスクリーンバッファを変更することで、画面に表示される文字を変更することができます。テキストモードの中には、文字の色や、点滅、下線、反転、明るくするなどの属性を設定できるものもあります。

MDA, CGA, EGA

VGAはMDA、CGA、EGAをベースにしていることを忘れてはならない。VGAはこれらのアダプタが行うモードの多くをサポートしています。これらのモードを理解することで、VGAの理解が深まります。

MDA

私が生まれる前（真面目な話）の1981年、IBMはPC用の標準的なビデオディスプレイカードを開発した。それが「モノクロディスプレイアダプター (MDA) 」と「モノクロディスプレイ・プリンタアダプター (MDPA) 」である。

MDAには、グラフィックモードは一切ない。唯一のテキストモード（モード7）では、80列×25行の高解像度のテキスト文字を表示することができた。

このディスプレイアダプターは、古いPCで使われていた一般的な規格でした。

CGA

1981年には、IBMがCGA (Color Graphics Adapter) を開発し、PCの最初のカラーディスプレイ規格となりました。

CGAは、1ピクセルが4バイトに制限されていたため、16色のカラーパレットしかサポートしていなかった。

た。CGAは、以下の2つのテキストモードと2つのグラフィックモードをサポートしていた。

- 40x25文字（16色）テキストモード 18x25
- 文字（16色）テキストモード 320x200ピク
- セル（4色）グラフィックモード
- 640x200ピクセル（モノクロ）グラフィックモード

ディスプレイアダプタを使って、「文書化されていない」新しいビデオモードを作成したり、発見したりすることができます。これについては後で詳しく説明します。

EGA

1984年にIBMが発表したEGA（Enhanced Graphics Adapter）は、最大640×350ピクセルの解像度で16色のディスプレイを実現した。

VGAアダプタは、80x86マイクロプロセッサ・ファミリーと同様に下位互換性があることを覚えておいてください。このため、後方互換性を確保するために、BIOSは80列×25行をサポートするモード7（MDAに由来する）で起動します。これは、私たちにとって重要なことです。

ビデオメモリー

メモリーマップドI/O（MMIO）

Memory Mapped I/Oについてご存知の方は、この部分は読み飛ばしてください。

プロセッサは、**RAMやROMデバイスからの読み出しで動作することができます**。アプリケーション・プログラミングでは、このようなことはまずありません。これを可能にするのがMMIOデバイスです。**Memory Mapped I/Oは、ハードウェアデバイスが自身のRAMやROMをプロセッサの物理アドレス空間にマッピングすることを可能にします**。これにより、プロセッサは、アドレス空間内のその場所へのポインタを使用するだけで、さまざまな方法でハードウェアRAMやROMにアクセスできるようになります。これは、MMIOデバイスが、プロセッサやシステムメモリが使用するのと同じ物理アドレスとデータバスを使用することで可能になります。

しかし、メモリマップドI/Oは、プロセッサの物理アドレス空間へのマッピングであり、実際のコンピュータメモリではないことを忘れてはならない。アーキテクチャによっては、MMIOデバイスマッピングを使用するか、その後ろに隠れているシステムメモリを使用するかをバンクスイッチで切り替えることができるものもありますが、そうでないものもあります。これが意味するところは、MMIOデバイスによって "隠された" 実際のシステムメモリのアドレスにアクセスできないということです。例えば、CMOS RAMメモリは0x400番地の物理アドレス空間にマッピングされています。これはメインシステムメモリとは異なります。ポインタで0x400にアクセスすると、MMIOに対して常にCMOS RAMメモリにアクセスすることになります。i86アーキテクチャでは、システムメモリのこの場所にアクセスすることはできません。

MMIOデバイスは、限られたシステムメモリで高解像度のビデオ表示を可能にしたり、システムメモリ内では失われてしまう情報を、バッテリーによって最新の状態に保たれたデバイス（CMOS RAM）から得ることを可能にするなど、ハードウェアをよりコントロールすることができます。MMIOデバイスのもう一つの例は、システムBIOS ROMそのものです。MMIOは、システムの物理アドレス空間にマッピングされたROMからプロセッサがBIOSを実行することを可能にします。すごいでしょ？

これがグラフィックと何の関係があるのかと思われるかもしれませんが。ビデオメモリは、物理アドレス空間にマッピングされたRAMです。ビデオメモリは、MMIOを使用するビデオディスプレイデバイスによって管理されます。**MMIOメモリがどのように管理されているかはデバイス次第であり、必ずしも直線的ではありません**。グラフィックスモードによって、このメモリの扱い方が異なるため、MMIOデバイスであることを理解することが重要です。

MMIOのアドレス空間領域の面白いところは、ページングにより、任意の仮想アドレスにマッピングし、そのアドレスからアクセスできることです。つまり、例えばビデオメモリを任意の仮想アドレスにマッピングし、その仮想アドレスからビデオメモリにアクセスすることができるのです。これはもちろん、物理アドレス空間のフレームにページがマッピングされることと関係があります。

また、**MMIOのメモリはシステムメモリにはないことも覚えておいてください**。コンピュータのシステムメモリは、アクセスしようとしているMMIOアドレスのサイズより大きい必要はありません。例えば、システムメモリが2GBしかない場合でも、0xFC000000の物理アドレス空間にマッピングされたRAMがあれば、MMIOデバイスにエラーなくアクセスすることができます。

この文字が見えますか？私はあなたのコンピュータの中にいて、**ビデオRAM（VRAM）に存在しているのです**。VRAMとはビデオメモリのことで、**ビデオフレームバッファ**とも呼ばれています。目の前にあるすべてのピクセルと、それ以上のものが含まれています。

標準VGA

ビデオメモリは、通常はビデオカードやオンボードビデオアダプターなどのビデオデバイスの内部に格納されている。標準的なVGAカードには256KBのVRAMが搭載されている。しかし、SVGA+カードでは、より多くのビデオメモリを搭載している

2021/11/15 13:13
ものも珍しくありません。

オペレーティングシステム開発シリーズ

しかし、やはり、高解像度のビデオモードでは、すべての画素をなんとか保存しなければなりませんよね。

第7章のメモリマップを覚えていますか？標準VGAのメモリは、**0x000A0000~0x000BFFFF**にあることがわかります。
0xBFFFF - 0xA0000 = 0xA0000で、655360バイト、つまり640KBです。

ここで覚えておいていただきたいのは、ビデオメモリはPCのアドレス空間ではこの位置にマッピングされているということです。つまり、ここに書き込むということは、ビデオアダプタにあるビデオメモリに書き込むということです。これは**Memory Mapped I/O**の一種です。

ビデオメモリにアクセスするときは、通常、実際のビデオRAMへの「ウィンドウ」を使ってアクセスします。これは典型的

- なものです。0xA0000 - EGA/VGAグラフィックモード(64KB)
- 0xB0000 - モノクロテキストモード(32KB)
- 0xB8000 - カラーテキストモードおよびCGA (32 KB)

モードごとにアドレスのマッピングが異なるため、モノクロディスプレイアダプターとカラーアダプターを同一マシン上で組み合わせることが可能です。これにより、デュアルモニター構成のパソコンでも問題なく動作させることができます。もちろん、これは標準的なVGAです。

スーパーVGA

スーパーVGAやその他のディスプレイアダプタは、一般的に動作が異なります。スーパーVGA以上の解像度のディスプレイアダプタでは、VRAMが高いアドレス範囲にマッピングされていることが珍しくありません。通常は標準VGAのメモリマップ範囲をサポートしていますが、高解像度のビデオモードや追加機能を提供するために、他のメモリ範囲を使用することもできます。例えば、私のNVideo GeForce 7600 GTには、使用可能な4つのメモリ範囲があります。0xA0000~0x000BFFFF（見覚えがありますか？）、0xFC000000~0xFCFFFFFF、0xD0000000~0xDFFFFFFF、0xFD000000~0xFDFFFFFFです。これはあなたのシステムでは異なる可能性があります。

リニアフレームバッファ (LFB)

現在のディスプレイのビデオメモリ全体を物理アドレス空間にマッピングすることができれば、リニアフレームバッファのように動作するように設定することができます。リニアフレームバッファとは、ピクセル単位でパックされたフレームバッファのことで、直線的に読み書きすることができます。例えば、**buffer[0]**はバッファの1番目の要素、**buffer[1]**は2番目の要素で、特に何もありません。まあ、実際にはありますが。標準的なVGAはLFBモードをサポートしていません。上記のモード0x13を覚えていますか？これは、リニアフレームバッファの効果を生み出す唯一の標準VGAビデオモードです。

これはちょっとわかりにくいかもしれませんが。結局のところ、ビデオメモリの読み書きが直線的でない場合、どのように「他の」方法で読み書きできるのでしょうか？これは標準VGAがプレーナーデバイスであることと関係があります。これについては、次のセクション以降で説明します。

バンク切り替え

スーパーVGAやそれ以上の解像度のビデオモードは、アダプタに搭載されている完全なビデオメモリへの「ウィンドウ」を使用する方法を提供します。例えば、上記のグラフィックスモードでは、**0xA0000~0xB0000**の間の64KBの領域に制限されています。これを「窓」とし、この64Kの窓を「移動」させることができれば、もっと大きなビデオメモリ領域にアクセスすることができます。たとえば、次のようになります。



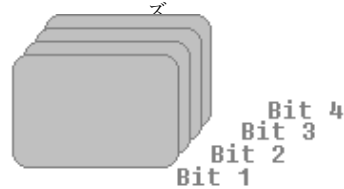
```
unsigned char* vidmem = 0xA0000;

vidmem[0] = 0; /* writes to topleft of screen
movwindow (1); /* moves window (see second picture)
vidmem[0] = 0; /* writes to topleft of screen + size of window
vidmem[0] always points to the topleft of the window, not screen
```

これを「バンク切り替え」といいます。バンク」とは、より大きなビデオメモリへのウィンドウのことです。標準的なVGAは64Kのグラフィック領域しか持たないため、ウィンドウのサイズは通常64Kです。

プレーナーメモリ

さて、ここで少し厄介なことが起こります。標準的なVGAのモードはプレーナーメモリモードで動作します。これはVGAのネイティブメモリモデルです。



4 planes, each sharing the same 64K window.
Each pixel's bits share the same location but different plane.

```
unsigned char* vmem = 0xa0000;
```

```
vmem[0] = pixel_bit 0
set_plane(1)
vmem[0] = pixel_bit 1
set_plane(2)
vmem[0] = pixel_bit 2
set_plane(3)
vmem[0] = pixel_bit 3
```

上記は、モード12hのプレーナーメモリーフォーマットの例です。モード12hは1ピクセルあたり4ビット。画素を描画するには、プレーナーのビットをセットしたりアンセットしたりする必要がある。これをよく理解するために、64kブロックのビデオメモリがあるとします。それを平らな紙のように想像し、その後ろにさらに3枚の紙を置きます。それぞれの紙は、この同じ64kのメモリ領域を共有する64kの「プレーン」です。それぞれのプレーンは、それが使用されているピクセルに関する少しの情報を保持しています。

この章では必要ないので、プレーナーのメモリとその仕組みを理解することを心配する必要はありません。しかし、VGAとモード12hについて詳しく説明する際には重要になります。本章では、プレーナーメモリの動作の詳細を隠したモード0x13を使用するため、今は必要ありません。

奇数/偶数メモリのアドレッシング

Odd / Even Memory Addressingは、Planer Memoryモデルを使用し、すべてのテキストモードで使用されます。すべての偶数アドレスはプレーン0または2で動作し、奇数アドレスはプレーン1または3で動作します。例えば、以下のようになります。

テキストモードでのビデオメモリへの書き込みを思い出し、次のようにします。

0 プレーン0 オフセット0
1 プレーン1 オフセット0
2 プレーン2 オフセット0
3 プレーン3 オフセット0

テキストモードでは、vmem[0]はプレーン0に格納され、プレーン1に属性バイトが格納されます。プレーン2には、フォントデータが格納されます。vmem[1]はプレーン1に格納され、プレーン3に属性バイトが格納されます。BIOSが起動時にインストールしたフォントを上書きすることになります。グラフィックモードでプレーン2を上書きして、テキストモードに戻ると、フォントデータが壊れているため、BIOSのテキスト出力ルーチンが期待通りに動作しないということです。

[アトリビュートプレーン]のオフセット2

テキストモードに戻りたい場合は、独自のフォントを保存するか、デフォルトフォントをバックアップしてプレーン2に書き戻してからテキスト出力ルーチンを使用する必要があります。

本章ではプレーナーメモリーモデルを使用しないため、Odd/Evenアドレッシングモデルも使用しません。

カラーパレット

パレットは、ルックアップテーブルのようなものです。カラーパレットは、色のルックアップテーブルのようなものです。例えば、実際の色情報のリストをテーブルに格納しておきます。そして、そのテーブルへのインデックスを別のテーブルにすることができます。

索引表 | カラーパレット

0	赤 (0)、緑 (0)、青 (0)
1	赤 (0)、緑 (0)、青 (1)
2	赤 (0)、緑 (1)、青 (0)
...	

上の例では、インデックスを使用するだけで、好きな色を参照できます。ルックアップテーブル（カラーパレット）が作成された後は、色を参照したいときにはいつでもインデックスを使うことができるので、ストレージスペースを大幅に節約できます。

例えば、カラーパレットを使用するビデオモードでは、ビデオメモリーがインデックスバッファの役割を果たします。そのため、先ほど作成したパレットを使ってピクセルを描画するには、使用したい色の**インデックス**を書き込めばよい。

VGAで**unsigned char* p[0xa0000]**でウェアで処理されます。パレットの色を好きなようにコントロール、変更することができます。ただし、パレットの操作にはVGAハードウェアのプログラミングが必要なので、ここではあまり触れません。VGAハードウェアの話になった時に説明します。心ください。

パレット・アニメーション

さて、ここで一步引いて考えてみましょう。上の例をもう一度見てみましょう。ビデオディスプレイでは、ピクセルの色をインデックスで決定します。例えば、カラーパレットのインデックス**1**（上の例ではLike）が別の色に変わったとしたらどうでしょう？上の例では、カラーパレットのインデックス**1**は明るい青色です。つまり、パレットビデオモードであれば、ビデオメモリのどこかに「**1**」を書き込むと、いつでもその鮮やかな青色になるということです。つまり、**memset (vidmem, 1, VIDMEM_SIZE)** を実行するだけで、ビデオディスプレイをこの色にすることができるのです。カッコいいでしょう？

ビデオディスプレイの表示色は、「カラーパレット」テーブルの中にあるインデックスの色で決まることがわかっているため、どのパレットエントリの色でも変更することができます。これにより、パレットの色を何らかの方法で更新するだけで、画面上の色を変化させることができます。これを「**パレットアニメーション**」と呼びます。

パレットアニメーションは、炎のアニメーションや氷のエフェクトなど、見栄えのするクールなエフェクトを多数作成することができます。

モード 0x13

アブストラクト

ビデオモード**0x13**は、256色320x200の解像度を持つIBMの標準的なVGA BIOSのモード番号である。256色のパレットを使用し、正方形のピクセルを持たず、**バックドピクセルフレームバッファ**として**ビデオメモリ**にアクセスすることができた。これはどういうことかということ、ビデオメモリをあたかもリニアバッファのようにアクセスできるようにしたということだ。ビデオメモリへのポインタを取得するだけで、ポインタが**unsigned char***であると仮定して、**pointer[0]=ピクセル1、pointer[1]=ピクセル2...**となります。これは、特定のハードウェアレジスタの設定（ビデオモードの「設定」）によって可能になります。標準的なVGAは、それ自体ではこのようなビデオメモリへのアクセスを提供しません。

ここで重要なのは、ビデオモードでは、解像度、ビデオメモリへのアクセス方法、そのモードを動作させるためのハードウェアの設定などが定義されていることです。ここですべてを理解できなくても気にしないでください。後の章でVGAハードウェアについて説明する際に詳しく説明します。

ビデオモード**0x13**は簡単に扱える（そして速い）ので、この章ではこれを使うことにしました。他のモードの中には、VGAハードウェアの経験が必要なものもありますが、その複雑さゆえに、この章では避けたいと思います。しかし、心配しないでください。後ほど、いくつかのモード（640x480x4カラーのモード**12h**など）を紹介する予定です。

ビデオモード**0x13**は、DOS時代にビデオゲームによく使われていましたが、プログラムが簡単で速度も速いのが特徴です。幅**320**×高さ**200**ピクセルの解像度で、256色のカラーパレットを持つビデオ構成です。プレーナー方式のビデオメモリモードですが、**リニアフレームバッファ(LFB)**として動作するため、プログラムが簡単です。

カラーパレット

モード**0x13**は、256色のカラーパレットを持つ。モード**0x13**のビデオメモリには、パレットインデックスが格納されているだけで、ビデオデバイスはインストールされているパレットのカラーテーブルからレンダリングする色を決定する。デフォルトでは、このようなカラーテーブルになっています。



モード 13h カラーパレット

例えば、上の図を見ると、最初の色(0)は黒、色1は青、色2は緑、などとなっています。これらの色をビデオディスプレイに書き込むには、上記のルックアップテーブルでこれらのインデックスを使用します。

上のコードの `unsigned char *p = 0xa0000` インデックスがパレットの色と一致していることを確認してください。

パレットの変更

```
*p = 0; //黒のピクセル
*(p++) = 1; // 青色のピクセル
*(p++) = 4; // 赤のピクセル
```

パレットを好きな色に変更する255個のピクセルを、そのための簡単なBIOS割り込みはありません（とにかくVBEを使わないと無理です）。ほとんどの割り込み呼び出しは、**VGA Digital to Analog Converter (DAC)** の内部にある個々のパレットレジスタまたはすべてのパレットレジスタを設定または取得するために使用されます。これにはVGAハードウェアの知識が必要ですが、この章では簡単にするために避けたいと思っています（心配しないでください、すぐに説明するつもりです！）。

ビデオBIOSインターフェース

VGAビデオBIOSインターフェースは、ビデオ割り込みのセットです（ソフトウェア割り込み0x10）。これらはBIOS割り込みであるため、リアルモードまたはv86モードでしか使用できません。

ビデオモードの設定

INT 0x10 機能 0

ビデオモードの設定は、BIOS割り込み0x10の機能0：入力呼び出すことで行

- うことができます。
 - AH = 0
 - AL = ビデオモー
- ド出力
 - AL = ビデオモードフラグ (Phoenix, AMI BIOS)
 - AL = CRTコントローラ(CRTC)モードバイト (Phoenix 386 BIOS v1.10)

CRTCは、ビデオハードウェアを直接プログラムする場合に必要なコントローラの一つであるため、今後多くの場面で目になることになるでしょう。

この割り込みでは、任意のテキストモードやビデオモードを設定することができます。例えば、以下は320x200x8ビット[モード0x13]に切り替えます：（全てのコードサンプルはデモにあることを覚えておいてください。

```
mode13hです。
mov ah, 0
mov al, 0x13
int 0x10
レット
```


簡単でしょう？

以上のことから、グラフィックモードにすることができます。確かに、リアルモードかv86モードでしか動作しませんが、これほど簡単な方法はありません。ビデオモードを理解していなくても心配ありません。ビデオモードについては後で説明します。

ビデオモードの取得

INT 0x10 機能 0xF

BIOS割り込み0x10関数0xF：入力呼び出すことで、ビデオモードを取得するこ

- とができます。
 - AH = 0xF
- 出力
 - AH = 文字列数 AL = 表示モード番号
 - BH = アクティブページ

この割り込みは簡単なもので、現在のビデオやテキストのモードを取得するのに使用できます。まだ「アクティブなページ」の部分は気にしないでください。ビデオモードを理解していなくても心配ありません。後ほど説明します。

その他のビデオBIOSインターラプト

INT 0x10 機能 0xB/BH=1

ビデオBIOSのINT 0x10の機能0xB：入力呼び出すことで、パレットの設定

- が可能です。
 - AH =
 - 0xB BH = 1
 - BL = パレットID
 - 00h 背景、緑、赤、茶/黄 01h 背景、シアン、マゼン
 - タ、白

この割り込みは、すべてのシステムでサポートされているわけではありません。

INT 0x10 機能 0xC

この割り込みを使って、ディスプレイにピクセルを書き込むことができます。

- 入力
 - AH = 0xC
 - BH = ページ番号
 - AL = ピクセルカラー
 - ビット7がセットされている場合、256色モードを除き、値はスクリーン
 - 上でXORされる CX = 列
 - DX=ロウ
- ・アウトプット
 - AL = ピクセルの色

この割り込みは、グラフィックモードでのみ使用できます。

INT 0x10 機能 0xD

Video BIOS INT 0x10関数0xB: Input呼び出すことで、ピクセルを読

- み取ることができます。
 - AH = 0xC
 - BH=ページ番号 CX
 - =コラム
 - DX=ロウ

この割り込みは、グラフィックモードでのみ動作します。

プリミティブ

最初のピクセルをプロットする

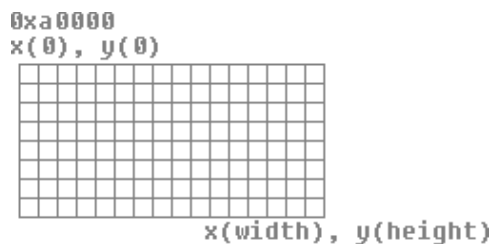
"あらゆるビデオゲームを作る秘訣は、ピクセルの色を変える能力である。"- Teej

この章では多くのことを学びましたが、まだ画面上にピクセルを描くことができません。どうなっているのでしょうか？この章の最後に、グラフィックスの最も基本的なプリミティブである「ピクセルのスクリーンへの描画」の基本を紹介することになりました。

今回はMode 0x13での作業なので、リニアフレームバッファのように動作することを覚えておいてください。つまり、`vidmem[0]`がビデオメモリの1バイト目、`vidmem[1]`が2バイト目ということになります。また、モード0x13では、各ピクセルの1バイトをカラーパレットへのインデックスとして使用することを覚えておいてください。つまり、次のようにピクセルを簡単に書くことができるのです。

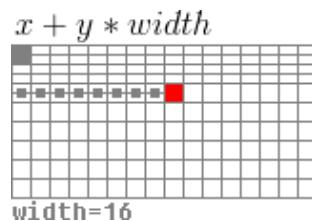
かつこのように、`unsigned char *p = 0xa0000`ピクセルが出来上がりました。
`p[0] = 1; // 青いピクセル`

直交座標系で考えるとわかりやすいですね。このシステムでは、XやYといった座標を使って、このような2次元のグラフ上の位置を表現します。



ビデオメモリの左上隅は $\mathbf{v} = [0,0]$ で、 \mathbf{v} は2次元のベクトルである。これはビデオディスプレイの最初のピクセルです。最後のバイトは、 $\mathbf{v} = [\text{width}, \text{height}]$ です。各座標がピクセルであると仮定すると、画面上の任意の位置にピクセルを描くことができる公式を思いつくことができます。

上のグラフで $\mathbf{v} = [0,0]$ の位置からスタートしたとします。この位置に幅を加えると、常に元の位置のすぐ下になってしまいます。例えば、上のグラフでは、幅=16です。左上からスタートしたとすると、右に16を数えると、スタートした場所の真下（次の線上）にいることになります。このことから、 $\mathbf{y} * \text{width}$ とすることで \mathbf{y} を計算することができます。その後、 \mathbf{x} （その行のオフセット）を加えれば、数式が完成します。



任意の $\mathbf{[x,y]}$ の位置にピクセルを描画するには、 $\mathbf{x + y * width}$ という式を使います。これで、次のような簡単なルーチンを作ることができます。

```

;-----;
ピクセルをレンダリングします。
;cl = color ax = y bx = x
;es:bp = バッファ
;-----;
ピクセルになります。
; [x + y * width] = col

プーシャ
mov di, VGA_MODE13_WIDTH
mul di ; ax = y * width add
ax, bx ; xの追加
mov di, ax
mov byte [es:bp + di], cl ; plot pixel
popa
レット

```

es:bpはビデオディスプレイ、またはレンダリングしたい他のバッファを指します。**cl**は使用したいカラーインデックス、**ax**はY位置、**bx**はX位置です。

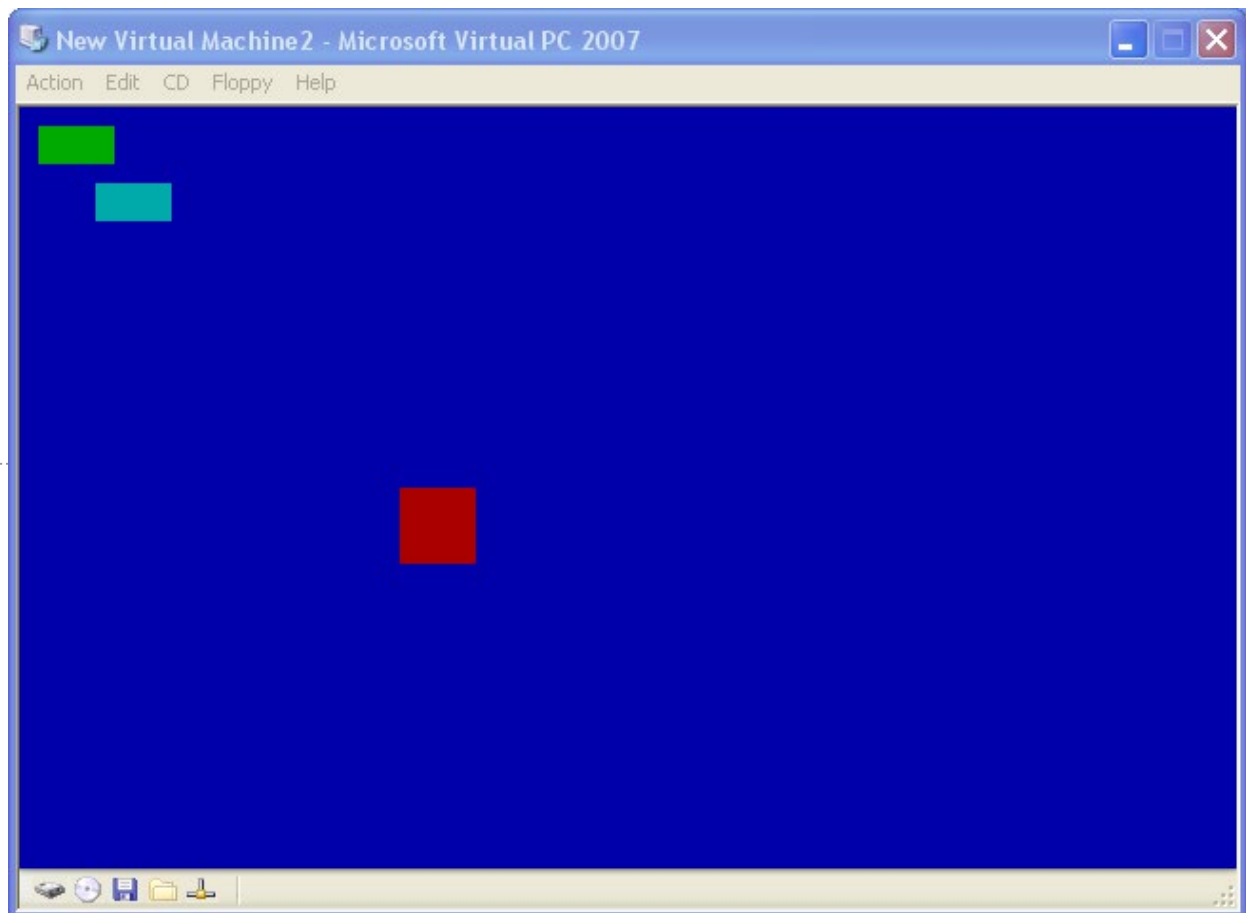
画面の消去

画面を消去するにはいくつかの方法があります。これは重要なことで、ビデオモードを切り替えたときに、画面にゴミがたくさん表示されることがあります。

1つの方法としては、上記のピクセルルーチンを**width * height**の回数だけ呼び出すことができます。もっと良い方法は、一度に複数のピクセルを書き込むことです。例えば、モード**13h**のピクセルサイズが**1**バイトであることを知っていれば、ワードサイズレジスタに**2**バイト（**2**ピクセル）を格納して、それを利用することが簡単にできます。

es:bpはビデオメモリまたは他のバッファを指し、**cl**は使用したい色を指します。

デモ
 クリアスクリーン
 cl = カラー
 ;-----;
 clrscr。



Mode 13h リアルモードでのデモ走行

このデモでは、これまで説明してきた内容に、水平方向の線を描画する「**line**」というルーチンを追加して、少しだけスパイスを加えています。

結論

この章では以上です。

次の章では、**VESA VBE**と、それを使って高解像度のグラフィックモードをどのように扱うかについて説明します。また、**スーパーVGA**、**バンクスイッチング**、**ダブルバッファリング**、**トリプルバッファリング**、**ページフリップ**などのグラフィックコンセプトについても説明します。そうです、**VBE**で高解像度を実現するのです！)

また、次の章では**C**言語に戻り、グラフィックプリミティブをいくつか取り上げる予定です。**VGA**のハードウェアについては、**VBE**の後に取り上げようと思っています。**VGA**のハードウェアは非常に複雑なので、グラフィックや**VGA**の複雑な話題はもう少し後にしたいと思っています。

次の機会まで。

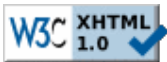
マイク

BrokenThorn Entertainment 社。現在、**DoE**と[Neptune Operating System](#)を開発中です。質問やコメント

はありますか？お気軽に[お問い合わせ](#)ください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ[私に教えてください](#)。

ホーム





オペレーティングシステム開発シリーズ

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

グラフィックス2 : VGAおよびSuperVGA

"複雑さをコントロールすることは、コンピュータ・プログラミングの本質である"。

ブライアン・W・カーニガン (Brian W. Kernighan)

by Mike, 2013

1. はじめに

Welcome!

前回の記事では、Video BIOSファームウェア上のVGAに重点を置いて、ビデオデバイスに関する概念を紹介しました。今回は、VGA、SuperVGA、Video BIOSをサポートするビデオデバイスのハードウェアとファームウェアのインターフェースの違いを見て、それらのインターフェースをサポートする共通のビデオインターフェースをC言語で実装してみます。以下は、そのトピックです。

1. VGAハードウェア
2. VGA BIOS
3. Vesa BIOS拡張機能
4. Bochs VBEインターフェース

まず最初に、パーソナルコンピュータで最も多くサポートされており、最も古い規格の1つである**VGA (Video Graphics Array)** 規格を紹介する。VGA規格は、標準的な方法でビデオハードウェアと対話する方法を提供していますが、低解像度の表示モードや、最新のディスプレイデバイスに存在するグラフィックアクセラレーションのサポートがないなどの制限があります。ここでは、ハードウェアインターフェースとVGA BIOSファームウェアインターフェースの両方を見ていきます。VGA BIOSインターフェースは、VGAハードウェアインターフェースよりもはるかにシンプルですが、**real**または**v86**プロセッサモードでしか使用できません。ハードウェアインターフェースは、どのプロセッサモードでも使用できます。

続いて、**VESA BIOS Extensions (VBE)** に移ります。VBEは、**VESA (Video Electronic Standards Association)** が策定した規格で、高解像度の表示モードやモニター機能をサポートするためのBIOS拡張機能の標準セットを提供しています。BIOSの拡張機能であるため、すべてのパソコンがサポートしているわけではありません。また、リアルモードまたは**v86**モードからしか使用できません。

この章の目的は、ビデオモードの変更とディスプレイメモリへのアクセスを行うことです。この章の終わりには、VGAまたはSuperVGAを使用したデモを作成することができるでしょう。この後の章では、グラフィックスに焦点を当てていきます (SuperVGAのハードウェアグラフィックスをサポートする場合もあります)。

1.1. BIOSとの連携

本章のメイントピックに入る前に、少し回り道をして、BIOSを詳しく見てみましょう。前回の記事で、VGA用のBIOSサービスのいくつかを使用したことを思い出してください。ソフトウェアがこれらのサービスを使用する場合は、リアルモードまたは**v8086**モードで実行する必要があります。これはプロテクトモードやロングモードのソフトウェアにとっては問題となります。そのため、先に進む前にこれを解決する方法を見つける必要があります。ただし、BIOSサービスを使用する予定がない場合は、このセクションをスキップしてください。

プロテクトモード (ロングモードではない) からBIOSを呼び出すには2つのアプローチがあります。

1. リアルモードに落としてBIOSを呼び出す
2. バイチャル8086モード

第一の方法は、よりシンプルですが、より完全に設計されたシステムでは非常に不便です。2番目の方法は、最も一般的に使用されている方法ですが、最も困難な方法でもあります。ユーザーモード、割り込みディスパッチ、タスクスイッチ、命令エミュレーションが必要です。

方法1

1つ目の方法は、ソフトウェアが必要に応じてプロテクトモードからリアルモードに切り替えられるようにすることです。この方法は、ソフトウェアの設計に一定の制限 (仮想メモリをサポートしないとか、上位半分のカーネルをサポートするとか) を加えないと、サポートが複雑になり、その価値がなくなってしまいます。しかし、この方式は最もシンプルな実装方法であり、追加のソフトウェアサポートも最小限で済みます。以上の理由から、今回のデモではこの方法を採用しましたが、ソフトウェアシステムが十分に大きい場合には、方法2を使用することを強くお勧めします。

この方法を実装するためには、**32ビット**のプロテクトモードと**16ビット**のリアルモードの間のインターフェイスとして機能するルーチンまたはルーチンのセットが必要です。これらのルーチンは、ルーチンの入力値と出力値を保持しながら、以下のことを行う必要があります。

1. 予約しなければならない現在のシステム状態を保存します。最も基本的なケースでは、プロテクトモードのスタックとIDTRです。
2. ハードウェア割り込みを無効にする (CLI命令)。
3. オリジナルのIVTをリロードします。これは、IDTR.sizeを0xffffに、IDTR.baseを0に設定することで行われます。
4. 16ビットプロテクトモードへのジャンプを行います。
5. CR0.PMビットのクリアによるプロテクトモードの解除
6. 16ビットのリアルモードコードへのジャンプを行う。
7. すべてのリアルモードセグメントを設定し、割り込みを有効にして、BIOSを呼び出します。
8. 32ビットプロテクトモードへのジャンプを行います。

9. 保存されたシステム状態を復元する。(1)のような最も基本的なケースでは、セクタ、プロテクトモードスタック、IDTR です。

このルーチンは、システムがサポートするものがより要求されるようになると、非常に複雑になります。上記のリストは大変そうですが、システムがページングを使用しておらず、カーネルイメージが1MB以下であれば、難しいことはありません。つまり、プロジェクトのベースアドレスが64Kで、ページングが無効になっていると仮定することで、ルーチンを比較的シンプルに保つことができます。

デモでは、BIOSの呼び出しに**io_services**というメソッドが使われています。上記の手順でリアルモードに落ちます。**io_services**は以下のようになります。

extern void io_services (unsigned int num, INTR* in, INTR* out);

num は割り込み番号、**in** は **INTR** 構造体へのポインタ、**out** は出力 **INTR** 構造体へのポインタです。**INTR**は、レジスタの値を格納する構造体のセットです。**io_services**と**INTR**はどちらもかなり大きいので、本文では省略します。デモでは**bios.asm**を参照してください。

◆例を示します。

この例では、関数 **io_services** と **INTR** 構造体を使用して BIOS を呼び出し、ビデオモードを設定します。Cコードはプロテクトモードで実行されていることに注意してください。

```
void vga_set_mode (int mode) {...
    //コールBIOS */*
    INTR in, out;
    in.eax.val = mode;
    io_services (0x10, &in, &out)です。
}
```

ソフトウェアがロングモードの場合は、デバイスを直接プログラムするか、エミュレーターを書くしかありません。

方法2

2つ目の方法は、**v8086**モードを使用することです。この方法は、BIOSファームウェアの呼び出しをサポートするための長期的な方法ですが、最も要求の厳しい方法でもあります。最低でも**v8086**モードでは、OSが以下をサポートしている必要があります。

- 1. ユーザーモードのプロセス
- 2. タスクの切り替え
- 3. 割り込みディスパッチ
- 4. 命令エミュレーション

仮想**8086**モードは、ユーザーモードプロセスとしてのみ実行できます。しかし、これには問題があります。ユーザーモードのプロセスはカーネルモードの命令 (**int**など) を実行できないため、BIOSを呼び出すことができず、目的が達成できません。つまり、**v8086**プロセスが**int**(割り込み)命令を実行すると、**GPF(General Protection Fault)**が発生してしまうのです。これを解決するにはどうしたらいいでしょうか？

ここで完全に解決策がないわけではありません。**v8086**プロセスはBIOSを呼び出すことはできませんが、カーネルは呼び出すことができます。GPFが発生すると、事実上、カーネルが呼び出されます。カーネルのGPFハンドラは、GPFが発生した原因を検出し、次のように対処します。

- 1. 現在のプロセスを確認する **v8086** フラグ
- 2. 設定されている場合、**v86_monitor**を呼び出す
- 3. 設定されていない場合は、GPFを続行し、場合によってはプロセスを終了させる

v86_monitorは、すべての**v8086**プロセスのカーネルGPFハンドラから呼び出される特別な関数です。ここでは、**v8086**プロセスがBIOSを呼び出そうとする（または、カーネルモードの命令を使おうとする）たびに、カーネルGPFハンドラが呼び出され、**v8086**タスクを「監視」するために**v86_monitor**が呼び出されるということです。

v86_monitorは、**v8086**タスクが使おうとした問題命令をエミュレートする役割を持つ**v8086モニター**を実装します。例えば、**v8086モニター**は、問題のある命令（CPUから与えられたCS:EIPを持つ）を割り込み呼び出しとして検出し、IVT [n]（n = 呼び出すBIOS番号）を呼び出すことで、それをエミュレートします（ただし、IVT命令のポインタは、線形ではなく、segment:offset形式であることに注意してください）。

2. VGA（Video Graphics Array

SuperVGAだけを見たい方は、この項を読み飛ばしても構いません。

VGA（Video Graphics Array） は、1987年にIBMのPS/2コンピュータ用に初めて導入されたディスプレイハードウェアのデザインであるが[1]、現在ではディスプレイの標準規格として各組織で広く採用されている。サポートされているビデオモードの最高解像度は640x480x16色です。PCメーカーに広く採用されたため、VGAは現在のPCでもサポートされている最も古い規格の1つとなっています。

VGAに続いてIBMの**XGA（Extended Graphics Array）** 規格が導入されたが、メーカーごとに拡張機能が実装され、最近のPCでは**SuperVGAアダプタ**が一般的に使われている。ほとんどのSuperVGAカードは、VGA規格との下位互換性がある。

この記事は、VGAの複雑さゆえに、網羅的な説明にはならないことをご了承ください。VGAハードウェアの詳細については資料[2]と[3]を参照してください。また、VGAに関する多くの大規模な書籍を読むことをお勧めします。

2.1. ビデオモード

VGAがサポートするビデオモードとモード番号には、標準的なセットがあります。**ビデオモード**とは、ディスプレイの構成と、解像度、ビット深度（1ピクセルあたりのビット数）、色数、**メモリモード**などのプロパティのことを指します。**標準的なビデオモード番号**は、**0h、1h、2h、3h、4h、5h、7h、Dh、Eh、Fh、10h、11h、12h、13h**です。モード番号自体は特別なものではなく、ビデオBIOSが特定のビデオモードを参照するために使用するものです。もっと多くのモードがあるかもしれませんが、それらは標準ではありません。

モード	解像度	色数	ビット深度
解像度	色数	ビット深度	モード

0h	40x25 テキスト	16色	Dh	320x200	16色
1h	40x25 テキスト	16色	え	640x200	16色
2h	80x25 テキスト	16色	Fh	640x350	2色
3h	80x25 テキスト	16色	10h	640x350	16色
4h	320x200	4色	11h	640x480	2色
5h	320x200	4 グレー	12h	640x480	16色
7h	80x25 テキスト	2色	13h	320x200	256色

標準のVGAでサポートされている最高の解像度は、640x480x16色の**モード12h**です（面白いことに、**Windows XP**のロゴ画面は**モード12h**で動作します）。（それ以上の解像度を得るには、後述する「SuperVGA」を使用する必要があります。

2.2. VGAファームウェア

まず、VGAのファームウェアインターフェースと、**Video BIOS**が提供するファシリティについて見ていきます。今回は、ビデオモードを設定するための**割り込み0x10関数0**を中心に、いくつかの機能を紹介しました。**Video BIOS**は、より抽象的なインターフェースを通じて、VGAハードウェアの設定、取得、および操作のためのサービスを提供します。おそらく、ソフトウェアがハードウェアを直接制御するよりも、ビデオBIOSの機能を使用する方がはるかに安全で、シンプルで、移植性が高いでしょう。

ここでは、ソフトウェアがビデオサービスに使用できる一般的な設備を紹介します。もちろん、これらは**BIOS**割り込みなので、リアルモードまたは**v8086**モードで、**BIOS**ファームウェアを持つシステムでしか使用できません。また、これらはカーネルモードのソフトウェアでしか使用できません。

INT 0x10 Function 0 - Set Video Mode

入力します。

- AH=0
- AL = ビデオモード

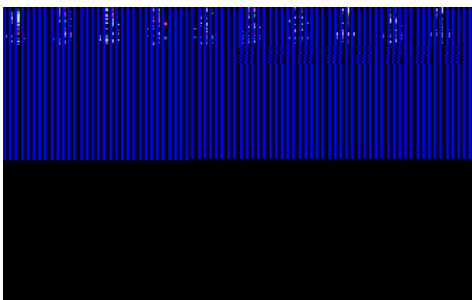
出力します。

- AL=ビデオモード フラグ (Phoenix, AMI BIOS)
- AL = CRTコントローラ(CRTC)モードバイト (Phoenix 386 BIOS v1.10)

例 この機能は、ビデオモードを設定します。

```
void vga_set_mode (int mode) {...
    コールBIOS /*。
    INTR in, out;
    in.eax.val = mode;
    io_services (0x10, &in, &out)です。
}
```

これで、上記の関数を使って、Mode 13hなどの任意のVGA BIOSビデオモードを設定することができます。結果は次のようになります。



VGAモード設定後の結果。

上の画像を見ていただければわかるように、ディスプレイにはたくさんのゴミがあります。この「ゴミ」は、実際にはモード切り替え前に**VGA**メモリにあったものです。これには、プレーン1のテキスト文字やプレーン2の**VGA**フォントなど、4つの**プレーン**（これらについては後述します）すべてのものが含まれています。ディスプレイをクリアすると、これらすべてがクリアされる。これは、ソフトウェアがテキストモードに戻らなければならない場合、ディスプレイをクリアすることによってソフトウェアが**VGA**フォントをクリアしてしまうため、問題となる。この問題を解決するには2つの方法があります。

1. BIOSから**VGA**フォントの新しいコピーをアップロードする
2. メモリをクリアして後で復元する前に、**VGA**フォントを保存してください。

上記のいずれかを実行することで、グラフィックモードに切り替わり、エラーなくテキストモードに戻ることができます。

INT 0x10 Function B - Set Palette (テキストモードのみ)

入力します。

- AH=B

BH=1

- ◆ BL=パレットID。
 - 00h 背景、緑、赤、茶/黄 01h 背景、シアン、マゼンタ
 - 、白

◆ 例 次のコードは、VGAパレットを設定するものです。

```
void vga_set_palette (int id) {
    コールBIOS */*
    INTR in, out;
    in.eax.val = 0xB;
    in.ebx.val = 0x0100 | id;
    io_services (0x10, &in, &out)
    となります。
}
```

INT 0x10 Function C - Write pixel

入力します。

- ◆ AH=C
- ◆ BH=ページ番号
- ◆ AL=ピクセルカ
- ◆ ラー CX=列
- ◆ DX=行

出力します。

- ◆ AL=ピクセルカラー

◆ 例 以下は、ピクセルを書き込みます。これはデモでは使うべきではないので省略されるかもしれませんが、念のために提供しています。

```
void vga_plot_pixel (int col, int x, int y) {
    /* call BIOS */
    INTR in, out;
    in.eax.val = 0x0C00 | col;
    in.ebx.val = 0;
    in.ecx.val = x;
    in.edx.val = y.
    となります。
    io_services (0x10, &in, &out)です。
}
```

なお、Bochsエミュレータの一部のバージョンでは、この割り込みをサポートしていません。しかし、VirtualPCには実装されています。上記の割り込みを使って、ピクセルをプロットすることができます。

INT 0x10 Function F - Get Video Mode

入力します。

- ◆ AH=F

出力します。

- ◆ AH=文字列の数 AL=表示ページ番号
- ◆ BH=アクティブページ
- ◆

◆ 例 次のコードは、モード情報を返すものです。

例

```
void vga_get_mode (unsigned int* col, unsigned int* dispPage, unsigned int* actPage)
{ INTR in, out;
  /* サニタリーチェック */
  if (!col || !dispPage ||
      !actPage) return;

  in.eax.val = 0xf;
  io_services (0x10, &in, &out)です。
  *dispPage = out.ax.r.al;
  *actPage = out.bx.r.bh;
}
```

2.3. VGAハードウェアインターフェース

VGAのハードウェアインターフェイスは非常に複雑で、5つのコントローラと、ポートI/Oアドレス空間からアクセス可能な100以上のハードウェアレジスタで構成されている。現在の標準的なVGAのビデオモードは、標準的なレジスタの構成によって定義されている。その理由は

ポートI/Oアドレス空間を使用することで、ソフトウェアはどのプロセッサモードでもVGAハードウェアとインターフェイスすることができます。もちろん、**in/out**ファミリーの命令が実際にハードウェアとやりとりするためには、ソフトウェアがスーパーバイザレベル（リング0）である必要があります。従って、VGAハードウェアにアクセスできるのはカーネルモードのソフトウェアだけである。ユーザーモードのソフトウェアがアクセスしようとする、**in/out**命令の実行時に一般保護フォルトが発生します。

ハードウェアを見る前に、警告があります。この警告は、無効なデータやデバイスがサポートしている範囲外のデータ（例えば、デバイスが安全に動作できない過剰な周波数設定など）に対する保護機能が欠如している可能性があるすべてのディスプレイモニターやビデオカードに適用されます。最近のモニターでは、無効な設定にはエラーメッセージが表示されたり、出力されなかったりします。必ず最初にエミュレーターや仮想環境でドライバソフトをテストし、ドライバソフトが正常に動作することを確認してから実際のハードウェアでテストしてください。

ビデオメモリー

VGAのメモリアレイアウトは、現在アクティブになっているビデオモードの種類によって異なります。VGAは以下のメモリアレイアウトに対応しています。

- 1. リニア
- 2. プラナー
- 3. パレット
- 4. 4 カラーモード

次に、これらのモードのすべてと、どのような場合に使用されるかを見てみましょう。メモリモデルの違いを知ることは、ディスプレイのメモリを正しく読み書きするために重要です。ここでは、最も複雑なリニアモードを最後に取り上げます。その前に、まずビデオメモリへのアクセス方法を知っておく必要があります。

システムアーキテクチャの章で見たメモリマップを思い出してください。ここでは、再びそれを紹介します。


- ◆ 0x00000000 - 0x000003FF - リアルモードインタラプトベクターテーブル
- ◆ 0x00000400 - 0x000004FF - BIOSデータエリア
- ◆ 0x00000500 - 0x00007BFF - 未使用
- ◆ 0x00007C00 - 0x00007DFF - 当社ブートローダ
- ◆ 0x00007E00 - 0x0009FFFF - 未使用
- ◆ 0x000A0000 - 0x000BFFFF - ビデオRAM (VRAM) メモリ
- ◆ 0x000B0000 - 0x000B7777 - モノクロ・ビデオメモリ
- ◆ 0x000B8000 - 0x000BFFFF - カラー・ビデオメモリ
- ◆ 0x000C0000 - 0x000C7FFF - ビデオROM BIOS
- ◆ 0x000C8000 - 0x000EFFFF - BIOSシャドウ領域
- ◆ 0x000F0000 - 0x000FFFFFF - システムBIOS

メモリマップによると、ビデオメモリは0xA0000~0xBFFFFの物理アドレス空間にマッピングされています。これは0x1FFFFバイトのメモリ、つまり131072バイトで128Kとなります。[2]によると、VGAハードウェアは最大256Kのメモリを持っていますが、マッピングされているのはそのうちの128Kのみです。マッピングされていないメモリは、VGAのアドレスデコードメカニズムを変更することでアクセス可能になります（ここでは説明しませんが）。以下のいくつかの例でこれを示します。

プラナーメモリーモード（16ビットカラーモード

VGAのメモリは、4つのプレーンがそれぞれ64kのメモリとして参照されます。これらは、互いに接続された異なるメモリバンクと考えることができます（実際にはそうではないかもしれませんが）。これらのウィンドウは、プレーンです。

16色モードでは、1色につき4ビットが使用されます。この4ビットは、各プレーンの同じ位置に格納されています。

 **例**4ビットの画素は、plane0[0]、plane1[0]、plane2[0]、plane3[0]に格納できます。これにより、1つのピクセルがディスプレイメモリに表示されます。4ビットのピクセルは4つのプレーンすべてに格納されており、各プレーンにはピクセルの1ビットしか格納されていないことに注目してください。また、ピクセルは4つのプレーンすべての同じ位置（インデックスは0）にあることにも注目してください。

プラナーメモリーモードでの画素の書き込み例は、レジスタやビデオモードの設定方法を確認した後、少し後に紹介します。残念ながら、ピクセルの書き込みには、書き込み先のプレーンを選択するためのハードウェアレジスタへの書き込みも必要となるため、まだ例を示すことができません。

パレット・メモリー・モード（256色モード

パレットメモリーモードでは、各画素はカラーテーブルへのインデックスという数字で表されます。このカラーテーブルがパレットである。代表的なのは256色モードで、各画素が8ビットである。その他のパレットモードとしては、16色（4ビットピクセル）や2色のみのモノクロ（1ビットピクセル）モードがある。

カラーテーブルは以下のようになっています。これは16色モードで使われるパレットで、実際、システム起動時のデフォルトはVGAテキストモードのパレットである。

インデックス	カラー名	インデックス	カラー名
0	ブラック	8	ダークグレー
1	ブルー	9	ライトブルー

2	グリーン	10	ライトグリーン
3	シアン	11	ライトシアン
4	レッド	12	ライトレッド
5	マゼンタ	13	ライトマゼンタ
6	ブラウン	14	黄色
7	ライトグレー	15	ホワイ

ここでは、テキストモードでのパレットの使用例を紹介します。

◆ **例を示します。** テキストモードでは、各文字は文字コード（通常はASCII文字）と属性バイトを含むことを思い出してください。属性バイトは、上記パレットへのインデックスである。言い換えれば、プロテクトモードでテキストモードを扱ったことがある人は、すでにパレットメモリーモードを扱ったことがあるということです！

DOSのビデオゲームでは256色モードが広く使われていた。このモードは、標準的なVideo BIOSのモード番号にちなんで「**モード13h**」と呼ばれ、有名になった。モード13hは、非常に高速で、リニアで、一度に256色を画面に表示できるという興味深いモードである。つまり、このモードのビデオメモリーは、各ピクセルが1バイトで、メモリー内で隣り合って保存される**リニアなものなのだ**。モード13hが面白いのは、VGAがリニアではなく平面的な表示デバイスだからだ。

◆ **例次のCコードは、256色モードのピクセルをモード13hのあるxとyの位置にプロットするpixel_256という関数を定義しています。モード13hは320x200なので、ピッチ（ここではピッチは幅）は320であることを思い出してください。また、VGA BIOSの割り込みによるプロットピクセルサービスとは異なり、このサービスはあらゆるPCのエミュレータやバーチャルマシンで動作します。**

```
#define VGA_VRAM 0xA0000
#define PITCH 320
void pixel_256 (unsigned char color, unsigned int x, unsigned int y) {
    unsigned char* fb= (unsigned char*) VGA_VRAM;
    符号付整数      offset= y * PITCH + x;
    fb [offset] = カラー。
}
```

上記の例は、Mode 13hを使った作業のシンプルさを示すものでもあります。複数のピクセルに連続して書き込むだけで、複数のピクセルをレンダリングできます。256色モードでは、各ピクセルがバイトとして表現されるため、符号なしのcharを使用します。

パレットメモリーモードでは、一度に表示できる色数に上限があります。これは、そのモードで使用されているカラーテーブルのエントリ数と同じです。例えば、256色モードでは、最大で256色までしか画面に表示できません。16色モードでは16色しか表示できない。ソフトウェアでは、カラーテーブル自体を変更して異なる色を表示することもできる。

リニアメモリーモード

リニアメモリーモードとは、リニアフレームバッファ（LFB）をサポートするビデオモードのことである。リニアメモリーとは、C言語の配列のように連続したバイトの配列のことで、リニアメモリーモデルを持つモードの例として、モード13hがある。また、悪名高い「モードX」（360x480x256色）や「モードQ」（チェーン4 256x256x256色）は、「モード13h」を改良したものだ。

VGAはリニアなデバイスではなく、リニアなメモリーモードをサポートしていないことを思い出してほしい。モード13h、モードX、モードQなどのモードは、いずれもリニアメモリーモデルのように錯覚させる方法でハードウェアを構成するブラナーモードである。

リニアメモリーモードでピクセルを書き込むには、ピクセル位置のオフセットを計算して、フレームバッファにピクセルを書き込むだけでよいのです。

◆ **例** 次のコードは、リニアメモリーモードで8ビットのピクセルをプロットします。この例は上の例と同じですが、より一般的なもので、リニアメモリーモデルを持つどのモードにも適用できます。

```
#define VGA_VRAM 0xA0000
#define PITCH 320
void pixel_256 (unsigned char color, unsigned int x, unsigned int y) {
    unsigned char* fb= (unsigned char*) VGA_VRAM;
    符号付整数      offset= y * PITCH + x;
    fb [offset] = カラー。
}
```

ハードウェア

VGAのハードウェアデザインは、以下のコンポーネントで構成されています。これらの中には見覚えのあるものもあるかもしれません。

1. CRTコントローラー
2. シーケンサー
3. グラフィックスコントローラ
4. RAMDAC
5. ビデオメモリー
6. 属性コントローラ

すべてのハードウェアレジスタは、I/Oポート空間にマッピングされています。つまり、CPUのin/out命令群を使ってアクセスすることができます。ほとんどのコントローラは、**アドレスレジスタとデータレジスタ**の2つのレジスタをマッピングしています。**アドレス・レジスタ**には、読み書きしたい特定のレジスタの**インデックス**が格納され、**データ・レジスタ**にはそのレジスタのデータが格納されます。このことは、いくつかの例を見ることでより明確になるでしょう。

これらの部品を見る前に、VGAハードウェアの複雑さを強調したいと思います。VGAの詳細を網羅した本が何冊も出版されていますが、1つの記事ですべての詳細を網羅するのは非常に困難な作業です。ここでは、各コンポーネントの機能と、そのコンポーネントが使用するレジスタのセットについてのみ説明します。レジスタの数が多いため、ここではレジスタの説明はしません。残念ながら、ハードウェアレジスタの知識がないと、ビデオモードの設定方法を説明することができません。この問題を解決するために、すべてのレジスタのリストと、ビデオモードの設定の表を示します。

を最後に記載しています。これにより、**読者は各レジスタの詳細やフォーマットを気にすることなく、VGAのビデオモードを設定することができる。**ただし、VGAに興味のある方は[3]や[4]を参考にして、各レジスタの詳細を読むことをお勧めします。

要するに、レジスタの詳細はあまり気にしないでいい。ビデオモードリストを見れば、どのレジスタにどのような値を入れればよいかがわかります。

CRTコントローラー（CRTC）

★ 警告

CRTコントローラー（CRTC）の設定を誤ると、ビデオカードや接続されているモニターにダメージを与える可能性があります。まれではありますが、CRTやLCDモニターの不適切な設定や誤設定により、焼損や爆発する例が知られています。

CRTコントローラー（CRTC）は、ディスプレイ・モニターへのビデオ・データの出力を制御する役割を担っています。CRTCは、アドレスレジスタとデータレジスタによってアクセスされます。**アドレスレジスタは3D4hに、データレジスタは3D5hにあります（Miscellaneous Output Register I/O Address selectフィールドが設定されている場合は、それぞれ3B4h、3B5h）。**アドレス・レジスタを使用して、アクセスしたいCRTCレジスタを選択します。その後、データレジスタを使用して、そのCRTCレジスタから読み書きすることができます。

レジスタの数が多いため、各レジスタの完全な技術的レビューは行いません。CRTレジスタの詳細については、FreeVGAプロジェクトのページをご参照ください。ご要望があれば、将来的にはVGAに関するトピックを拡張するかもしれません。

0	水平方向のトータルレジスタ
1	終了 水平表示レジスター 水平ブラ
2	ンキングレジスター開始 水平ブラン
3	キングレジスター終了 水平リトレ
4	スレジスター開始 水平リトレースレ
5	ジスター終了 垂直トータルレジスタ
6	ー
7	オーバーフローレジスタ
8	Preset Row Scan Line
9	Register Maximum Scan
10	Line Register Cursor Start
11	Register
12	カーソルエンドレジスタ
13	スタート・アドレス・ハイ・レ
14	ジスタ スタート・アドレス・ロ
15	ー・レジスタ カーソル・ロケー
16	ション・ハイ・レジスタ カーソ
17	ル・ロケーション・ロー・レジ
18	スタ パーティカル・リトレース
19	・スタート・レジスタ パーティ
20	カル・リトレース・エンド・レ
22	ジスタ パーティカル・ディスブ
23	レイ・エンド・レジスタ オフセ
24	ット・レジスタ

アンダーライン・ロケーション・

レジスタ 垂直ブランキング開始

レジスタ 垂直ブランキング終了

レジスタ CRTCモード・コントロ

ール・レジスタ

ラインコンペアレジスタ

CRTCレジスタは、ディスプレイの水平・垂直方向のリトレース期間とブランキング期間のピクセルクロックのタイミングを制御することができます。これらのタイミングは、ディスプレイの出力（**ビデオ解像度**）やリフレッシュレートの制御に役立ちます。その他のCRTCレジスタでは、ビデオメモリのスタートアドレスの変更（**プリセットロースキャンライン**と**スタートアドレスレジスタ**、ハードウェアカーソルの位置（**カーソルロケーションレジスタ**）とアンダーライン（**アンダーラインロケーションレジスタ**））が可能です。**CRTC**モードコントロールレジスタでは、CRT自体といくつかのアドレッシングモードの制御が可能です。

グラフィックスコントローラ

グラフィックスコントローラは、CPUとビデオメモリのインターフェースを管理する役割を担っています。**アドレスレジスタは3CEhに、データレジスタは3CFhにマッピングされています。**グラフィックスコントローラのレジスタにアクセスするには、アドレスレジスタにレジスタのインデックスを書き込み、データレジスタからリードまたはライトします。

以下に、標準的なレジスタの一覧を示します。各レジスタの詳細な説明については、FreeVGAプロジェクトをご参照ください。

0	SET/RESETレジスタ
1	イネーブルセット/リセット
2	レジスタ カラーコンペアレ
3	ジスタ データローテートレ
4	ジスタ
5	リードマップセレクトレジスタ グ
6	ラフィックモードレジスタ 雑グラ
7	フィックレジスタ カラードントケ
8	アレジスタ
	ビットマスクレジスタ

シーケンサー

シーケンサーは、ビデオデータとRAMDACの間のインターフェースを管理します。**アドレス・レジスタは3C4hに、データ・レジスタは3C5hにマッピングされています。**シーケンサのレジスタにアクセスするには、アドレス・レジスタにレジスタのインデックスを書き込み、データ・レジスタからリードまたはライトします。

以下に、標準的なレジスタの一覧を示します。各レジスタの詳細な説明については、FreeVGAプロジェクトをご参照ください。

0	レジスターのリセット
1	クロックモードレジスタ
2	マップマスクレジスタ キ
3	チャクタマップレジスタ
4	シーケンサメモリモードレジスタ

属性コントローラ

属性コントローラは、VGA用の21個のレジスタで構成されています。コントローラーには、I/Oポートスペースにマッピングされた2つのレジスターがあり、1つは**3C0h**、もう1つは**3C1h**にある。他のコントローラとは異なり、**アトリビュートコントローラでは、3C0hのレジスタをデータ書き込みポートとアドレスポートの両方として使用する。3C1hのレジスタは、データリードレジスタです。**アトリビュートコントローラと通信するためには、ソフトウェアはまずポート**3C0h**にレジスタのアドレスを書き込み、**続いてそのレジスタに書き込むデータを書き込まなければなりません。データの読み出しは、ポート3C0hに書き込んだ後、3C1hから読み出すことで可能です。**また、アトリビュートアドレスレジスタには特定のフォーマットがあります。

属性アドレスレジスタ							
7	6	5	4	3	2	1	0
		PAS	アドレス				

アドレスとは、アクセスするためのレジスタ・インデックスのことです。以下のレジスタ一覧表をご参照ください。

PAS (Palette Address Source) ホストまたは**EGA**ディスプレイアダプタによるパレットデータへのアクセスを決定します。**0**であれば、ホストはパレットRAMにアクセスでき、アダプタはディスプレイメモリがパレットにアクセスすることを禁止します。**1**の場合、ディスプレイ・メモリはパレットRAMにアクセスでき、ホストはアクセスを禁止されます。

アトリビュートコントローラーには、カラーインデックスと色を対応させる**16**個のパレットレジスタのセットが格納されている。このパレットの小ささは**EGA**の欠点である。**VGA**では、これらのレジスタを引き続き使用するが、パレット情報を保存する代わりに、**2番目のカラーインデックスレジスタのセットにアドレスを保存する。**ここでは、その仕組みの詳細を説明しませんが、興味のある方は**[4]394ページ**をご覧ください。

以下に、標準的なレジスタの一覧を示します。各レジスタの詳細な説明は、FreeVGAプロジェクトまたは**[4]**を参照してください。

汎用レジスタ 0-15パレットエントリレジスタ	
16	モードコントロールレジスタ
これらのレジスタリストは、まだまだ終わらないようです。VGAの主要なコントローラのレジスタリストを見てきましたが、ビデオモードで使われる一般的な使用と雑多なデータのためのレジスタのセットはもっとあるので、見ておく必要があります。これらのレジスタは、 一般的なレジスタ または 外部レジスタ と呼ばれるかもしれません。	
以下に、標準的なレジスタの一覧を示します。各レジスタの詳細な説明は、FreeVGAプロジェクトまたは [4] を参照してください。 なお、カラーアダプタでVGA (EGAではない) を想定しています。 モノクロのアダプタとEGAのアダプタでは、使用するポートが異なる場合があります。	

ライトポート。 3C2h リードポート3CCh	その他の出力レジスタ 機能制御レジスタ
ライトポート。 3BAh リードポート3CAh	入力ステータス#0レジスタ 入力ステータス#1レジスタ
ポートを読む。 3C2h リードポート3BAh	スタ

カラーレジスター

これらのレジスタは、我々ソフトウェアが**256**色のパレットを操作・管理するためのものです。**256**色モードで作業する場合には重要なレジスタ群であり、ビデオモードを設定した後に使用することもあるので、よく理解しておくといでしょう。また、これがこれから見る最後のレジスタ群であることを知っておくとよいでしょう。

以下に、標準的なレジスタの一覧を示します。各レジスタの詳細な説明は、FreeVGAプロジェクトまたは**[4]**を参照してください。

ライトポート。 3C8h リードポート3C8h	PELアドレス書き込みモードレジスタ PELアドレスリードモードレジスタ
ライトポートです。 3C7h	PELデータレジスタ
ライトポート。 3C9h リードポート3C9h	DACステートレジスタ PELマスクレジスタ
ポートを読む。 3C7h	タ
ライトポート。 3C6h リードポート3C6h	

2.4. 標準ビデオモード

ビデオハードウェアを設定することで、異なる特性を持つさまざまな表示モードを定義することができます。しかし、ビデオハードウェアがサポートする多くのモードは、多くのモニターがサポートしていません。また、他の設定が望ましくない効果をもたらすこともあります。いくつかのVGAモードの設定は標準となっており、ほとんどのモニターでサポートされています。以下に、標準的なビデオモードとその構成の一覧を示します。**標準的なビデオモードの番号は、モード0h、1h、2h、3h、4h、5h、7h、Dh、Eh、Fh、10h、11h、12h、13hです。**その他の注目すべきモード、例えば**Mode X**や**Mode Q**はよくサポートされていますが、標準モードではありません。

その他のリストは[3]または[4]を参照してください。以下の表は[4]の304〜305ページから採用しました。この表は、各モードが変更するレジスタの異なるセットに分かれています。一番上の行はモード番号、左の列は特定のレジスタにアクセスするためのインデックスです。**4**]によると、この値は標準的なビデオBIOSが使用する**初期のモード状態**を示しています。

ビデオモードを設定するには、ソフトウェアがビデオ出力を無効にし、変更するすべてのハードウェアレジスタに関連する値を書き込む必要があります。これにより、読み取り/書き込みモード、アドレスの開始、水平/垂直ブランキング期間とタイミング、リフレッシュレート、解像度、グラフィックモードの種類など、すべてが設定されます。

正確さには万全を期していますが、以下の表には気づけなかった誤りがあるかもしれません。この表は[4]で発表されたものと一致するように書かれていますが、書き換えの際に誤りがあるかもしれません。この点については、原著者に敬意を表します。

汎用レジスタ

	モード															
インデックス	0	1	2	3	4	5	6	7	D	E	F	10	11	12	13	
0	63	63	63	63	63	63	63	A6	63	63	A2	A3	E3	E3	63	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70	
3	4	4	5	5	4	4	5	FF	4	4	FF	4	4	4	4	

シーケンス・レジスタ

	モード															
インデックス	0	1	2	3	4	5	6	7	D	E	F	10	11	12	13	
0	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	
1	9	9	1	1	9	9	1	0	9	1	1	1	1	1	1	
2	3	3	3	3	3	3	1	3	F	F	F	F	F	F	F	
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	2	2	2	2	2	2	6	2	6	6	6	6	6	6	E	

CRTCレジスター

	モード															
インデックス	0	1	2	3	4	5	6	7	D	E	F	10	11	12	13	
0	2D	2D	5F	5F	2D	2D	5F	FF	2D	5F	FF	5F	5F	5F	5F	
1	27	27	4F	4F	27	27	4F	FF	27	4F	FF	4F	4F	4F	4F	
2	28	28	50	50	28	28	50	FF	28	50	FF	50	50	50	50	
3	90	90	82	82	90	90	82	FF	90	82	FF	82	82	82	82	
4	2B	2B	55	55	2B	2B	54	FF	2B	54	FF	54	54	54	24	
5	A0	A0	81	81	80	80	80	FF	80	80	FF	80	80	80	80	
6	BF	BF	BF	BF	BF	BF	BF	FF	BF	BF	FF	BF	B	B	BF	
7	1F	1F	1F	1F	1F	1F	1F	FF	1F	1F	FF	1F	3E	3E	1F	
8	0	0	0	0	0	0	0	FF	0	0	FF	0	0	0	0	
9	C7	C7	C7	C7	C1	C1	C1	FF	C0	C0	FF	40	40	40	41	
A	6	6	6	6	0	0	0	FF	0	0	FF	0	0	0	0	
B	7	7	7	7	0	0	0	FF	0	0	FF	0	0	0	0	
C	0	0	0	0	0	0	0	FF	0	0	FF	0	0	0	0	
D	0	0	0	0	0	0	0	FF	0	0	FF	0	0	0	0	
E	0	0	0	0	0	0	0	FF	0	0	FF	0	0	0	0	
F	31	31	59	59	31	31	59	FF	31	59	FF	59	59	59	31	
10	9C	9C	9C	9C	9C	9C	9C	FF	9C	9C	FF	83	EA	EA	9C	
11	8E	8E	8E	8E	8E	8E	8E	FF	8E	8E	FF	85	8C	8C	8E	
12	8F	8F	8F	8F	8F	8F	8F	FF	8F	8F	FF	5D	DF	DF	8F	
13	14	14	28	28	14	14	28	FF	14	28	FF	28	28	28	28	
14	1F	1F	1F	1F	0	0	0	FF	0	0	FF	F	0	0	40	
15	96	96	96	96	96	96	96	FF	96	96	FF	63	E7	E7	96	
16	B9	B9	B9	B9	B9	B9	B9	FF	B9	B9	FF	BA	4	4	B9	
17	A3	A3	A3	A3	A2	A2	C2	FF	E3	E3	FF	E3	C3	E3	A3	
18	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	

グラフィックスコントローラ

モード インデックス	0	1	2	3	4	5	6	7	D	E	F	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	10	10	10	10	30	30	0	10	10	0	0	10	0	0	40
6	0E	0E	0E	0E	0F	0F	0D	0A	5	5	5	5	5	5	5
7	0	0	0	0	0	0	0	0	0	F	5	0	5	F	F
8	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF

属性コントローラ

モード インデックス	0	1	2	3	4	5	6	7	D	E	F	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	13	13	17	8	1	1	8	1	3F	1	1
2	2	2	2	2	15	15	17	8	2	2	0	2	3F	2	2
3	3	3	3	3	17	17	17	8	3	3	0	3	3F	3	3
4	4	4	4	4	2	2	17	8	4	4	18	4	3F	4	4
5	5	5	5	5	4	4	17	8	5	5	18	5	3F	5	5
6	6	6	6	6	6	6	17	8	6	6	0	14	3F	14	6
7	7	7	7	7	7	7	17	8	7	7	0	7	3F	7	7
8	10	10	10	10	10	10	17	10	10	10	0	38	3F	38	8
9	11	11	11	11	11	11	17	18	11	11	8	39	3F	39	9
A	12	12	12	12	12	12	17	18	12	12	0	3A	3F	3A	0A
B	13	13	13	13	13	13	17	18	13	13	0	3B	3F	3B	0B
C	14	14	14	14	14	14	17	18	14	14	0	3C	3F	3C	0C
D	15	15	15	15	15	15	17	18	15	15	18	3D	3F	3D	0D
E	16	16	16	16	16	16	17	18	16	16	0	3E	3F	3E	0E
F	17	17	17	17	17	17	17	18	17	17	0	3F	3F	3F	0F
10	8	8	8	8	1	1	1	0E	1	1	0B	1	1	1	41
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0F	0F	0F	0F	3	3	1	0F	0F	0F	5	0F	0F	0F	0F
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ビデオモードを設定するには、上の表にある、モードが変更するレジスタをすべて設定する必要があります。以下の例では、モードの設定を説明しています。



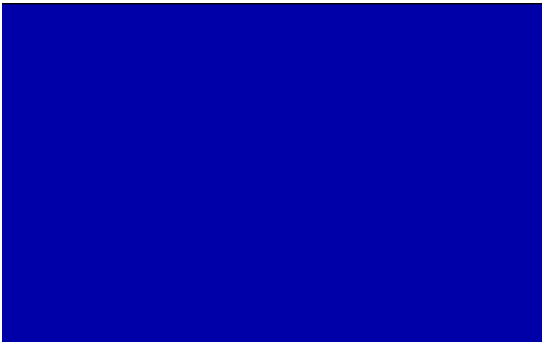
例

ビデオモードを設定するためには、そのモードに関連するレジスタにすべての値をアップロードする必要があります。例えば、これを行う関数 **uploadRegisters** があれば、次のようにしてモード **13h** を設定することができます。

```
unsigned char _mode13h = {
    /* 一般的なレジスタ */
    0x63, 0, 0x70, 0x4,
    シーケンサー */
    0x3, 0x1, 0xF, 0, 0xE,
    /* CRTC */
    0x5F, 0x4F, 0x50, 0x82, 0x24, 0x80, 0xBF, 0x1F, 0, 0x41,
    0, 0, 0, 0, 0, 0x31, 0x9C, 0x8E, 0x8F, 0x28, 0x40, 0x96, 0x89, 0xA3, 0xFF,
    /* グラフィックス */
    0, 0, 0, 0, 0, 0x40, 0x5, 0xF, 0xFF,
    /* 属性 */ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0xA, 0xB, 0xC, 0xD, 0xE, 0xF, 0x41, 0, 0xF, 0, 0
}

モード13hを設定する */
uploadRegisters (&_mode13);
```

レジスタの値を書き込んだ後、ビデオモードの変更に成功しました。



VGAモード13hで動作し、ディスプレイメモリをクリアするデモ

以前、平面メモリモデルを見たとき、ハードウェアレジスタを見るまでは、ピクセルをプロットする例を完成させることができなかったことを思い出してください。ここではその例を紹介します。

例

次のコードは、8ビットのピクセルを平面的にプロットします。

```
void pixel_p (unsigned char color, unsigned int x, unsigned int y)
{
    ...
    符号付き整数* fb          = (unsigned int*) 0xa0000;
    符号付き整数              offset= y * pitch +
    (x/8); bankSwitch (offset >> 16);

    グラフィックコントローラのビットマスクレジスタに書き込んでプレーンを選択する */
    outputb(0x3CE,8)
    outputb(0x3CF,0x80 >> (x &
    7)); fb [bankoffset] = color;
}
```

3. SuperVGAインターフェース

SuperVGAとは、SuperVGA (SVGA) をはじめ、XGA、SXGA、SXGA+、UXGA、QXGA、QSXGAなど、2560x2048以上の解像度に対応した規格を含むディスプレイハードウェアのクラスを指す。SuperVGAは、メーカーがVGAを拡張する形で始まった規格で、現在のSuperVGAとは別の規格となっています。つまり、**SuperVGA機器のすべてを網羅した規格は存在しないのだ**。各SuperVGAカードはそれぞれ異なり、ハードウェアやファームウェアのインターフェースも異なります。そこで、問題が発生します。標準的な使い方がないのに、どうやってSuperVGAをサポートするのか？この問題は、ほとんどのメーカーがカードの技術仕様を公開していないことで、さらに深刻化しています。その代わりに、WindowsやLinuxなどの異なるOS用にブラックボックスのドライバーを書いています。

このため、SuperVGAデバイスに対応するためには、2つの選択肢しかありません。

1. 対応したいディスプレイデバイスの種類ごとに、デバイスドライバを書きます。
2. VBE (Vesa Bios Extensions) の使用

1つ目の方法は、仕様書がないと作業ができないので難しいです。仕様書を手でできないデバイスの場合、リバースエンジニアリングに頼るしかありません。今回のSuperVGA「カード」は、Bochsエミュレータで使われているものを使用します。これを選んだのは、Bochsを使っている人なら誰でも使えるからです。ただし、ドライバのコードはBochs専用となります。

2つ目の方法は、**VBE (Vesa Bios Extensions)** を使用することです。VBEは、SuperVGAハードウェアのためのBIOS割り込み拡張の標準セットです。VBEは、非常にシンプルで、SuperVGAを扱うための単一の標準的なインターフェースを実現しています。ただし、リアルモードまたはv8086モードが必要で、BIOS割り込みを使用するため、すべてのマシンがVBEをサポートしているわけではありませんし、拡張機能でもありません。

3.1. VBE (Vesa Bios Extensions) ファームウェアインターフェース

VESA Bios Extensions (VBE) は、SuperVGAモードで動作するためのBIOS割り込みサービスを定義しています。VBEは、**VESA Bios Extensions (VBE) Core Standard**で定義されています(参考文献[5])。(カーネルモードのソフトウェアは、リアルモードまたはv8086モードでBIOSサービスを呼び出すことができます。

VBEモード番号

VBEでは、ビデオBIOSと同様に、**モード番号**を定義しています。モード番号は次のような形式になっています。

VBEモード番号							
15	14	13	12	11	10	9	ビット 0...8
DM	LFB	Reserved, set to 0				モード	

Mode numberは、実際のモード番号です。ビット8がセットされていれば、VESA定義の標準モードです(詳細は後述します)。

LFBは、モードを**LFB (Linear Frame Buffer)** またはバンクスイッチングモードとして選択します。0の場合、バンクスイッチングを使用し、標準的なVGAフレームバッファを使用します。1の場合は、リニアフレームバッファを使用します。

DMは、モード設定時にディスプレイメモリをクリアするかどうかを選択します。0であれば、ディスプレイメモリをクリアします。1の場合は、ディスプレイメモリをクリアしません。

VESA定義のモードは、BIOSベンダーがサポートを推奨するビデオモード番号の標準セットです。以下の表は、グラフィックモードの一覧です。

モード	解像度	色深度	モード	解像度色
-----	-----	-----	-----	------

			深さ		
100h	640x480	256	113h	800x600	32K
101h	640x480	256	114h	800x600	64K
102h	800x600	16	115h	800x600	16.8M
103h	800x600	256	116h	1024x768	32K
10Dh	320x200	32K	117h	1024x768	64K
10Eh	320x200	64K	118h	1024x768	16.8M
10Fh	320x200	16.8M	119h	1280x1024	32K
110h	640x480	32K	11Ah	1280x1024	64K
111h	640x480	64K	11Bh	1280x1024	16.8M
112h	640x480	16.8M			

11Bhを超えるモード番号も定義できますが、標準ではありません。そのため、より高い解像度のモードをサポートすることも可能です。

 **例**

VBEモード番号**113h**は、バンクスイッチングを使用する**800x600x32K**のカラーモードを選択し、VBEモード番号**8113h**は、リニアフレームバッファ（LFB）を使用する**800x600x32**のカラーモードを選択します。モード番号のフォーマットを覚えよう

3.2. VBEサービス

次に、VBE BIOSのサービスについて説明します。これらのサービスはすべて[5]で参照できますので、覚えておいてください。ここでは、ビデオモードの設定とディスプレイのメモリアクセスに必要な、最も重要な3つのサービスについてのみ説明します。その他の割り込みについては、仕様書（読みやすい仕様書の一つです）を参照してください。

INT 0x10 Function 4F00h - VBE Controller Informationの取得

入力します。

- AX=4F00h
- ES:DI=VbeInfoBlock構造体へのポインタ(以下の例を参照)

出力します。

- AX=ステータス

構造です。

vbeInfoBlock構造は、以下のようなフォーマットになっています。

```
typedef struct _vbeInfoBlock {...
    uint8_    tsignature[4        ]; // "VESA"
    uint16_t  version              ; // 0x0200(VBE 2.0)または0x0300(VBE 3.0)のいずれか
    uint32_t  oemString            ; // OEM名へのファープインタ
    uint8_t   capabilities[4]; // 能力
    uint32_t  videoModesPtr       ; // ビデオモードリストへの遠いポインタ
    uint16_t  totalMemory         ; // 64Kブロック単位のメモリサイズ
    uint16_t  oemSoftwareRev;
    uint32_t  oemVendorNamePtr;
    uint32_t  oemProductNamePtr;
    uint32_t  oemProductRevPtr;
    uint8_    reserved [222];
    uint8_    toemData [256]です。
}vbeInfoBlock;
```

vbeInfoBlock.capabilities フィールドのフォーマットは以下の通りです。


能力			
ビット	2	1	0
31...3			
予約	D2	D1	D0

D0: 0の場合、DACは6ビット/色に固定されます。1の場合、DACの幅を各色8ビットに切り替えることができます。

D1: 0の場合、コントローラは標準VGAモードをサポートします。1の場合、コントローラは標準VGAモードをサポートしません。

D2: 0の場合、RAMDACは通常の動作をしています。1の場合は、ファンクション9のブランク・ビットを使用するように仕様が指示されています。

以下の例では、この割り込みを呼び出すことができます。

 **例**

ここでは、本当に多くのことを語る必要はありません。割り込みを呼び出すには、構造体を使って **rm_bios** ルーチンを呼び出します。**SEG** と**OFFSET**はそれぞれリアルモードの**セグメント**と**オフセット**を計算するマクロです。これは、32ビットのリニアアドレスをBIOSに与える際に、16ビットのセグメント：オフセットアドレスに変換するために使用します。

```
void vbe_get_descr (vbeInfoBlock* descr) {
    INTR イン、アウト。
    サニティチェック */
    if (!descr)
```

を返すことができます。

```
コールBIOS */*。
in.eax.val = 0x4F00です。
in.es = SEG((unsigned int) descr);
in.edi.val = OFFSET((unsigned int) descr);
io_services (0x10, &in, &out)です。
}
```

INT 0x10 Function 4F01h - Get VBE Mode Info

入力します。

- AX=4F01h
- CX=モード番号(VBEのモード番号のフォーマットを思い出してください！)
- ES:DI=ModeInfoBlock構造体へのポインタ(以下の例を参照)

出力します。

- AX=ステータス

構造です。

ModeInfoBlockは次のような構造を持っています。構造体のサイズが大きいため、すべてのメンバーについては説明しません。後の章で説明される可能性のある重要なメンバーはコメントされている。

```
typedef struct _modeInfoBlock {
    uint16_t attributes;
    uint8_t windowA,
    windowB; uint16_t
    granularity; uint16_t
    windowSize;
    uint16_t セグメントA、セグメン          INT 0x10関数へのptr 0x4F05 */ (注)
    トB。                                /* スキャンラインあたりのバイト数 */
    uint32_t winFuncPtr;
    uint16_t pitch, resolutionX, resolutionY; /* 解像度*1
    uint8_t wChar, yChar, planes, bpp, banks; /* バンクの数 */*/*。
    uint8_t memoryModel, bankSize,
    imagePages; uint8_t reserved0;

    uint8_t readMask, redPosition          ; /* カラーマスク */*/*。
    uint8_t greenMask,
    greenPosition; uint8_t blueMask,
    bluePosition;
    uint8_t reservedMask,
    reservedPosition; uint8_t
    uint16_t physBaseAddress;              LFBモードでのLFBへのポインタ */*/*。
    uint32_t offScreenMemOff;
    uint16_t offScreenMemSize;
    uint8_t reserved1 [206];
}modeInfoBlock;
```

次の例では、割り込みの呼び出しを行っています。



例

次の関数は、上記の割り込みを使用して基本的なバンク切り替えを行います。**SEG**と**OFFSET**を使用して、**out**の32ビットのリニアアドレスを16ビットのsegment:offset far pointerに変換していることに注意してください。

```
void vbe_get_mode (int mode, modeInfoBlock* descr) {...
    INTR イン、アウト。

    サニティーチェック */
    if (!descr)
        を返すことができます。

    BIOSを呼び出す */
    in.eax.val =
    0x4F01; in.ecx.val
    = mode;
    in.es = SEG ((unsigned int)descr);
    in.edi.val = OFFSET((unsigned int)descr);
}
io_services (0x10, &in, &out);
```

INT 0x10 Function 4F02h - Set VBE Mode

最後にご紹介するのは、ディスプレイモードを設定するためのインタラプトです。これは、VGAやVBEで定義されたSuperVGAモードや、標準ではない拡張モードの設定に使用できます。

入力します。

- AX=4F02h
- BX=モード番号 (VBEのモード形式を覚えておこう！)。

出力します。

- ◆ AX=ステータス

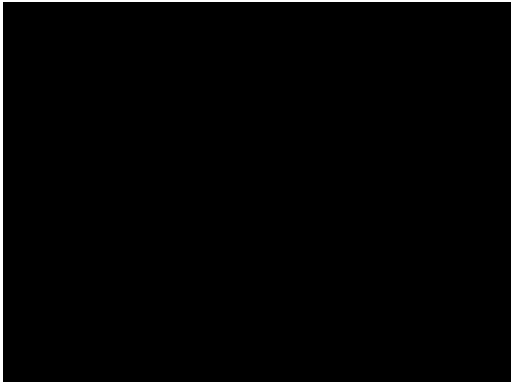
以下の例では、VBEモードを設定しています。



次の関数は、上記の割り込みを使用してVBEモードを設定します。

```
void vbe_set_mode (int mode) {...
    コールBIOS */*.
    INTR in, out;
    in.eax.val = 0x4F02;
    in.ebx.val = mode;
    io_services (0x10, &in, &out)です。
}
```

vbe_set_modeを呼び出すと、VBEで定義された任意のモードを設定することができ、次のような結果が得られます。



SuperVGA Mode 118h (1024x768) で動作するデモ

3.3. VBEディスプレイメモリ

VBEは、SuperVGAで採用されている2つの標準的な表示メモリの方式に対応しています。**リニアフレームバッファ (LFB) 」と「バンクスイッチング」だ**。VBEがサポートするすべてのビデオモードをこの2つで設定しようとしていることを思い出してください。これらは表示メモリへのアクセス方法の違いに過ぎません。この章では、両方のモードを扱う方法について説明します。

リニアフレームバッファ (LFB) モード

LFBモードでは、すべてのビデオメモリが**物理アドレス空間**にマッピングされ、通常は3GB~4GBの範囲の上位アドレスにマッピングされます (IA32アーキテクチャを想定)。ディスプレイメモリはC言語の配列のように直線的で、ポインタから読み書きするだけで、すべてのビデオメモリにアクセスできます。VBEでは、**INT 0x10 Function 4F01h**を呼び出して、現在のモードの**modeInfoBlock**構造を取得することで、このポインタを得ることができます。その時に表示するポインタは、ちょうど**modeInfoBlock.physBase**になります。



次のCコードは、16ビットの**赤-緑-青 (RGB)** のピクセルをプロットして表示します。

```
void pixel_16RGB (unsigned short color, unsigned short x, unsigned short y) {
    unsigned short* fb = (unsigned short*) _modeInfo.physBasePtr;
    unsigned short offset = x + y * (_modeInfo.bytesPerScanLine / 2);
    fb [offset] = カラー。
}
```

LFBモードのディスプレイメモリは、マッピングされているディスプレイメモリの量が多いため、プロテクトモードまたはロングモードからしかアクセスできません。リアルモードでは、高解像度モードのディスプレイメモリにアクセスするには、**バンクスイッチ**を使用するしかありません。

バンク切り替え

バンク切り替えを使用するモードでは、常に64KのディスプレイメモリがVGAメモリスペースの**0xa0000**の物理アドレスにマッピングされています。この64Kのディスプレイメモリのブロックを**バンク**と呼ぶ。ソフトウェアは一度に1つのバンクにしかアクセスできないため、ディスプレイメモリのすべてにアクセスすることはできない。**すべてのディスプレイメモリにアクセスするためには、必要に応じてバンクを切り替える必要があります。**

つまり、バンク0はディスプレイメモリの0~64Kバイト、バンク1はディスプレイメモリの64K~128Kバイトといった具合である。つまり、バンクを切り替えることで、ソフトウェアはこの64Kの "窓" を介して、どんなに高解像度のディスプレイメモリにもアクセスできるのだ。



次のコードは、8ビットのRGBピクセルをバンクスイッチングモードでプロットしています。

```
void pixel_8RGB (unsigned char color, unsigned short x, unsigned short y) {
    unsigned char* fb = (unsigned char*) 0xa0000;
    符号付き整数オフセット = x + (long)y * _modeInfo.bytesPerScanLine;
    vbe_bankSwitch (offset >> 16); fb [offset & 0xffff] =
    color;
}
```

バンク切り替えモードは、より多くの計算を必要とするため、一般的にはより遅くなります。つまり、画面上のピクセルの位置だけでなく、バンク内のオフセットや切り替えるバンクも計算しなければなりません。この計算は、ディスプレイメモリからの読み出し時にも同じです。しかし、バンク切り替えモードは、64Kのディスプレイメモリをマッピングするだけなので、リアルモードやv86モードでも使用することができます。

VBEでは、ソフトウェアは**INT 0x10 Function 0x4F05 - Display Window Control**を呼び出してバンクを切り替えることができました。また、**INT 0x10 Function 0x4F01 - Get VBE Mode Information**を呼び出し、**VbeModeInfo.WinFuncPtr**を呼び出すことで、（VBE仕様が推奨されているように）直接呼び出すこともできます。

INT 0x10 Function 4F05h - ディスプレイウィンドウコントロール

ディスプレイバンクの設定または取得を行います。**VbeModeInfo.WinFuncPtr**からも呼び出せます。

入力します。

- AX=4F05h
- BH=0（メモリウィンドウの設定）、1（メモリウィンドウの取得）
- BL=0（ウィンドウA）、1（ウィンドウB）
- DX=ウィンドウの粒度単位でのウィンドウ番号。BH=0（セット機能）の時のみ使用

出力します。

- AX=ステータス
- DX=グラニュラリティ単位のウィンドウ番号。BH=1（Get機能）の時のみ使用



次の関数は、上記の割り込みを使って基本的なバンク切り替えを行います。ウィンドウAとウィンドウBで割り込みを2回呼んでいることに注意してください。これは、規格によれば、VBEの実装によっては、読み取りウィンドウと書き込みウィンドウが別々になる可能性があるからです。

```
void vbe_bankSwitch (int bank) {.
    INTR イン、アウト。

    bank <=<= bankShift;
    in.eax.val=0x4F05;
    in.ebx.val = 0;
    in.edx.val =
    bankServices (0x10, &in, &out); /* コールBIOS *1
    in.eax.val=0x4F05;
    in.ebx.val = 1;
    in.edx.val =
    bankServices (0x10, &in, &out); /* コールBIOS *1
}
```

4. Bochs VBEインターフェース

Bochsエミュレータは、ファームウェアを呼び出さずにVBEモードを直接設定する代替方法を提供します。サポートは限られていますが、高解像度のグラフィックスを扱うためのセミポータブルな方法（互換性のあるBochsエミュレータのみが必要）を提供してくれます。



以下の関数は、BochsがサポートするあらゆるVBEモードの設定に使用できます。

```
#define VBE_DISPI_IOPORT_INDEX
0x01CE #define VBE_DISPI_IOPORT_DATA
0x01CF
#define VBE_DISPI_INDEX_ID 0x0
#define VBE_DISPI_INDEX_XRES 0x1
#define VBE_DISPI_INDEX_YRES 0x2
#define VBE_DISPI_INDEX_BPP 0x3
#define VBE_DISPI_INDEX_ENABLE 0x4
#define VBE_DISPI_INDEX_BANK_WIDTH 0x5
#define VBE_DISPI_INDEX_VIRT_HEIGHT 0x6
#define VBE_DISPI_INDEX_X_OFFSET 0x7
#define VBE_DISPI_INDEX_Y_OFFSET 0x8
#define VBE_DISPI_INDEX_Y_OFFSET 0x9

#define VBE_DISPI_DISABLED 0x00
#define VBE_DISPI_ENABLED 0x01
#define VBE_DISPI_GETCAPS 0x02
#define VBE_DISPI_8BIT_DAC 0x20
#define VBE_DISPI_LFB_ENABLED 0x40
#define VBE_DISPI_NOCLEARMEM 0x80

void BlBochsVbwrite (uint16_t index, uint16_t value) {.
    WRITE_PORT_USHORT(VBE_DISPI_IOPORT_INDEX, index);
    WRITE_PORT_USHORT(VBE_DISPI_IOPORT_DATA, value) となります。
}

void BlBochsSetMode (uint16_t xres, uint16_t yres, uint16_t bpp)
{ BlBochsVbwrite (VBE_DISPI_INDEX_ENABLE,
    VBE_DISPI_DISABLED);
    BlBochsVbwrite (VBE_DISPI_INDEX_XRES,
    xres);
    BlBochsVbwrite (VBE_DISPI_INDEX_YRES,
    yres);
    BlBochsVbwrite (VBE_DISPI_INDEX_BPP, bpp) となります。
}
```



```
}    bBochsVbeWrite (VBE_DISPI_INDEX_ENABLE, VBE_DISPI_ENABLED | VBE_DISPI_LFB_ENABLED);
```

標準的なスーパーVGAのハードウェアインターフェースは存在しないことを思い出してください。つまり、この方法はBochsに特化したものであり、他の環境やプラットフォームでは動作しない可能性があります。読者の皆様は、Bochsをメインのエミュレータとしてお使いになると思いますので、最高の互換性を保つために、この方法を採用します。また、この方法は最もシンプルな方法でもあります。

しかし、互換性を高めるためには、VBEのサービスを直接利用するために**仮想8086**モードを使用することをお勧めします。

5. デモ

今回の記事で公開を予定しているデモは複数あります。VGA BIOSデモ、VBE SuperVGAデモ、VGAハードウェアデモの3つです。

6. 結論

この章では、VGA BIOS、VGAハードウェア、SuperVGA、VESA BIOS Extensionsの紹介など、多くのことを説明しました。これで、後の章で必要となる低解像度と高解像度のグラフィックモードの切り替えに必要な資料をすべて網羅しました。次の数章では、実際のグラフィックレンダリングとパイプラインに焦点を当て、SuperVGAについても触れていきます。今回は、基本的なプリミティブとグラフィックレンダリング、そしてトランスフォームから始める予定です。

後のデモで使用する方法は、今回最後に紹介した「Bochs VBE Interface」です。これにより、読者の多くがBochsを使用することを前提に、後の記事で高解像度のLFBモードを使用することができます。つまり、どのような方法でディスプレイモードを設定しても、グラフィックスに焦点を当てたこれからの数回の記事に従うことができるのです。

次の機会まで。

～Mike () です。

OS開発シリーズ編集部

6. リソース

以下のリンクは、より直接的で正確な情報を提供するために参照されたものです。さらなる情報を得るために参照してください。

- [1] http://en.wikipedia.org/wiki/Video_Graphics_Array
- [2] <http://www.osdever.net/FreeVGA/vga/vga.htm>
- [3] http://wiki.osdev.org/VGA_Hardware
- [4] EGA、VGA、スーパーVGAカードのプログラマーズガイド（第3版）
- [5] VESA Bios Extensions (VBE) 2.0規格

[ホーム](#)



オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - マルチブート by Mike, 2010

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

このチュートリアルでは、マルチブート規格と、マルチブート・コンプライアント・オペレーティング・システムの開発方法について説明します。このシリーズでは、マルチブートの構造について説明しますが、マルチブート・コンプライアント・システムを作成するには、それ以上のことが必要です。あなたのシステムがマルチブート・コンプライアントであれば、どのマルチブート・コンプライアント・ブートローダでもロード可能になります。すごいでしょ？つまり、どのマルチブート・コンプライアント・ブートローダーでも、あなたのOSを起動できるということです。

マルチブート仕様

アブストラクト

マルチブート規格は、1995年に作成されたもので、**Free Software Foundation**が監修しています。マルチブートを可能にする標準的な方法を定義した書面による仕様を提供している。マルチブートとは、コンピュータシステムに複数のオペレーティングシステムやシステム環境をインストールして実行することができるものです。デュアルブートコンピュータシステムは、2つのオペレーティングシステムがインストールされているマルチブートの一例です。

マルチブートを可能にするのは、もうひとつの非公式なソフトウェアの仕掛けです。パーティショニングです。パーティショニングとは、1つの物理ディスク上に複数の論理ディスクが存在するような状態を作り出すことです。パーティショニングは、記憶媒体（通常はハードディスク）上のストレージを異なる用途に分けます。例えば、最初のパーティションはセクター0からセクター n までであり、NTFSフォーマットのWindows OSが入っています。他のパーティションには、データや他のオペレーティングシステムソフトウェアを含む、任意のファイルシステムを使用できます。

パーティション設定はソフトウェアのトリックなので、ブートローダがソフトウェアのパーティションテーブルを読み込んでこれらのパーティションを検出し、通常、起動するオペレーティングシステムが入っているパーティションのリストを表示できるかどうかにかかっています。これがブートメニューです。マルチブート仕様では、ブートローダがOSに制御を移すときのコンピュータの状態や、OSへのデータの受け渡し方法などを定義しています。

マルチブート規格は、マルチブートに対応していないディスクにも使用することができます。つまり、マルチブートのコンプライアントブートローダがフロッピーディスクから起動できる場合、フロッピーディスクのOSをそこから起動させることができるということです。

オペレーティングシステムイメージ

一般的なブートローダは、さまざまな種類のOSイメージを起動するように設定できます。一般的には、これはカーネルや他のOS読み込みプログラムです。マルチブート仕様では、イメージのフォーマットについての詳細は規定されていません。このため、フラットバイナリ、ELF、あるいはPEファイルなど、好きなフォーマットを使用することができます。

ただし、このファイルには、マルチブートヘッダーという追加のヘッダーが必要です。このヘッダーは、イメージの最初の8kのどこかに、ワード（32ビット）単位で配置されていなければなりません。マルチブートコンプライアントブートローダは、このヘッダーを見つけて情報を得ることができます。これにより、ブートローダはあなたのイメージをどのようにロードして実行するかを知ることができます。

ここでは、構造体のフォーマットを紹介します。

```
typedef struct _MULTIBOOT_INFO
{
    ...
    uint32_t flags;
    ...
}
```

```

uint32_t checksum;
uint32_t headerAddr; //全てオプション、フラグのビット16が設定されていれば設定される...
uint32_t loadAddr;
uint32_t loadEndAddr;
uint32_t bssEndAddr;
uint32_t entryPoint;
uint32_t modType; //全てオプション、フラグのビット2が設定されていれば設定される...
uint32_t width;
uint32_t 高さ;
uint32_t 深さ。

}multiboot_info, *pmultiboot_info;

```

パディングが追加されていないことを確認する必要があります。MSVCでは、上記の構造体宣言の周りに**#pragma pack(push,1)**と**#pragma pack(pop,1)**を追加することで、これを実現できます。

GRUBのようなマルチブート・コンプライアント・ブートローダでOSを起動するために必要な構造は、上記の通りです。メンバーを見てみましょう。

- **magic:** 常に0x1BADB002でなければなりません。
- のフラグを立てます。
 - ビット0
 - 0: すべてのブートモジュールとOSイメージは、ページ (4k) 境界に配置されなければなりません。
 - ビット1
 - 1: ブートローダはOSにメモリ情報を渡さなければならない。
 - ビット2
 - 1: ブートローダーはビデオモードテーブルをOSに渡さなければならない。
 - ビット16
 - 1: マルチブートヘッダのオフセット12~28が有効です。(つまり、マルチブートヘッダのメンバー header_addr から entry_addr が有効です)。このビットがセットされていると、ブートローダはイメージフォーマットを解析してそこから値を取得する代わりに、これらの値を使用します。マルチブートコンプライアントブートローダは、ロードできるELFやPEなどのネイティブ実行ファイルフォーマットのサポートを提供できます。
- **チェックサム。**これは、**magic**と**flag**に加えて、0の32ビット符号なしの合計でなければなりません。
- **headerAddr:** マルチブートヘッダのアドレス
- **loadAddr:** ロード先のベースアドレス
- **loadEndAddr:** ロード終了アドレス。0の場合、ブートローダはOSイメージファイルの終わりで見なします
- **bssEndAddr:** BSSセグメントの終わり。ブートローダはこのセグメントを無効にします。0の場合、BSSセグメントがないと判断されます
- **entryPoint:** エントリポイント関数のアドレス。そうです、OSのエントリーポイントです。
- **modType**です。
 - 0: リニアグラフィックモード
 - 1: EGA 標準テキストモード
 - その他はすべて予約済み
- **width:** ディスプレイの幅をテキストの列またはピクセルで指定します。0の場合、ブートローダは設定を想定していません
- **height:** ディスプレイの高さをテキストの列またはピクセルで表します。0の場合、ブートローダはdepthを設定しません。グラフィックスモードでのBPP(Bits Per Pixels)の数。0の場合、ブートローダは何も設定しません。

それがすべてです。ブートローダーは、2つの方法でOSをロードして実行することができます。1つは、実行可能なイメージフォーマット (ELFやPEなど) を読み込んで実行する方法、もう1つは、この構造体にある情報を利用する方法です。

ブートローダーは、イメージの中にあるこの構造を探します。そのため、この構造を記入して作成する必要があります。

マルチブートヘッダーの実装

マルチブート・ヘッダーをOSに組み込むには、さまざまな方法があります。ツールチェーンごとに異なるソリューションがあります。そのうちのいくつかを見てみましょう。

Visual C++ 2005、2008

これは、私が最近見つけて他のフォーラムに投稿したトリックです。これは、Microsoft Visual C++が提供す

る拡張機能を利用して、カーネル内のヘッダを定義するものです。

まず、構造体を宣言し、余分なパディングがないことを確認します。

あとは、`#pragma pack(push, 1)`で定義するだけです。このヘッダーは、ワード（32ビット）バウンダリで、カーネルの最初の8K以内に定義しなければならないことを覚えていますか？このトリックでは、構造体の適切な配置を保証するためにセクションアラインメント構造を使用します。IDEのリンカーオプションでセクションアラインメントを設定すると、ワードアラインメントになることが保証されます。あとは、新しいプログラムセクションを作って、その中に構造体を定義するだけです。

今すぐコードを見てみましょう。

```
uint32_t magic;
uint32_t flags;
uint32_t checksum;
uint32_t headerAddr;
uint32_t loadAddr;
#pragma section (.text)
__declspec(allocate(".text")) MULTIBOOT_INFO
MultibootInfo = {
    uint32_t entryPoint;
    uint32_t modType;
    uint32_t modHeaderMagic;
    uint32_t modHeaderFlags;
    uint32_t header;
    uint32_t depth;
    uint32_t checksum;
    uint32_t headerAddress;
    uint32_t loadAddress;
    uint32_t endAddress;
};
#pragma pack(pop);
```

これは動作しますが、問題があります。これは `.text` セクションに構造体を割り当てますが、どこに？マルチブートの仕様では、マルチブートの最初の8Kに配置することが要求されていますが、MSVCはまだ8K領域外に自由に配置するのを許します。これは問題になります。

この問題を解決するには、`MULTIBOOT_INFO`の命名規則を使う必要があります。MSVCで使われているセクションの命名規則は、`namesloc`の形式に従っています。`name`はセクションの名前で、`loc`はセクション内のどこを表すかを示す文字列です。セクション\$`a`が1番目、セクション\$`b`が2番目というように、英数字の順に並んでいます。つまり、`.text$0`を使えば、`.text`セグメントの先頭を表すことになります。しかし、もちろん、上記の`.text`を`.text$a`に置き換えるだけでいいです。

これをコードセグメントとして作成し、`.text`セクションにマージすることができます。

```
//!完全な例 #pragma
code_seg(". a$0")
__declspec(allocate(". a$0"))MULTIBOOT_INF
0_MultibootInfo = {

    multiboot_header_magic,
    multiboot_header_flags,
```

```

CHECKSUMです。
header_address,
loadbase,
0, //load end address
0, //bss end address
KeStartup
};

#pragma comment(linker, "/merge:.text=.a")

```

それがすべてです。**OADBASE**はカーネルのベースアドレス（例えば1MB）、**HEADER_ADDRESS**はマルチブートヘッダのアドレス（.textが常にオフセット0x400で始まるため、**LOADBASE+0x400**になります）、magicは**0x1BADB002**、flagsは**0x00010003**、チェックサムは**-(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)**です。

ここでは、その完全な例を紹介します。

```

#pragma pack (push, 1)

/**
 *   マルチブート構造
 */
typedef struct _MULTIBOOT_INFO {...

    uint32_t magic;
    uint32_t flags;
    uint32_t checksum;
    uint32_t headerAddr;
    uint32_t loadAddr;
    uint32_t loadEndAddr;
    uint32_t bssEndAddr;
    uint32_t entryPoint;
    uint32_t modType;
    uint32_t width;
    uint32_t height;
    uint32_t depth;

}multiboot_info, *pmultiboot_info;

#pragma pack (pop, 1)

/**
 *   カーネルエンタリー
 */
void KeStartup ( PMULTIBOOT_INFO* loaderBlock ) {。

    __halt () です。
}

// ! ローディングアド
// レス #define                                0x100000
LOADBASE
// ! ヘッダーのオフセットは常にこれにな
// ります #defineALIGN                        0x400
# defineHEADER_
ADDRESSLOADBASE+ALIGN
# define MULTIBOOT_HEADER_MAGIC                0x1BADB002
# define MULTIBOOT_HEADER_FLAGS                0x00010003
# define STACK_SIZE                            0x4000
# define CHECKSUM                              -(multiboot_header_magic + multiboot_header_flags)

#pragma code_seg(".a$0")
__declspec(allocate(".a$0"))MULTIB
OOT_INFO _MultibootInfo = {

    multiboot_header_magic,
    multiboot_header_flags,
    checksum,
    header_address,
    loadbase,
    0, //load end address
    0, //bss end address
    KeStartup
};

```

```
#pragma comment(linker, "/merge:.text=.a")
```

このカーネルが1MBのベースアドレスを持ち、Visual C++でコンパイルして有効なPE実行ファイルを作成したと仮定すると、これはあらゆるマルチブート・コンプライアント・ブートローダでブート可能なはずですが。

マシンの状態

ブートローダが我々のOSを実行するとき、レジスタは以下の値になっている必要があります。

- **EAX** - マジックナンバー。0x2BADB002でなければなりません。これはカーネルに、私たちのブートローダがマルチブート規格であることを示します。
- **EBX** - マルチブート情報構造体の物理アドレスが格納されています。
- **CS** - オフセットが`0`、リミットが`0xFFFFFFFF`の32ビットの読み取り/実行コード・セグメントでなければなりません。正確な値は未定義です。
- **DS,ES,FS,GS,SS** - 32ビットのリード/ライトデータセグメントでなければならず、オフセットは「0」、リミットは「0」です。0xFFFFFFFFとなります。正確な値はすべて不定です。
- **A20ゲート**が有効であること
- **CR0** - ビット31(PG)はクリア(ページング無効)、ビット0(PE)はセット(プロテクトモード有効)する必要があります。その他のビットは未定義

その他のレジスタはすべて未定義です。これらのほとんどは、既存のブートローダですで行われています。追加しなければならないのは、**EAX**レジスタと**EBX**の2つだけです。私たちにとって最も重要なものは**EBX**に格納されています。これは、マルチブート情報構造体の物理アドレスを格納します。それでは見てみましょう。

マルチブート情報構造

さて、ブートローダによってOSが起動したところで、次は何をするのでしょうか？マルチブート・コンプライアント・ブートローダは、OSに情報を提供する情報構造体も作成します。これらは、**EBX**レジスタの構造体へのポインタによって渡されます。

この構造体は、マルチブート仕様の中でも最も重要な構造体の一つです。この構造体の情報は、**EBX**レジスタからカーネルに渡されます。これにより、ブートローダがカーネルに情報を渡すための標準的な方法となります。

これはかなり大きな構造ですが、悪くはありません。これらのメンバーのすべてが必要なわけではありません。仕様では、オペレーティングシステムは、構造体のどのメンバーが存在し、どのメンバーが存在しないかを決定するために、**flags**メンバーを使用しなければならないとされています。

ここでは、構造全体のフォーマットを示します。マルチブートのヘッダー構造と同様に、パディングがないことを確認することをお勧めします。

```
typedef struct _MULTIBOOT_INFO {...
    uint32_t flags; uint32_t      必須
    memLower; uint32_t           //フラグのビット0が設定されている場合
    memUpper; uint32_t          //フラグのビット0が設定されている場合
    bootDevice; uint32_t        //フラグのビット1がセットされている場合
    commandLine; uint32_t       //フラグのビット2がセットされている場合
    moduleCount; uint32_t       //フラグのビット3がセットされている場合
    moduleAddress; uint32_t      //フラグのビット3がセットされている場合
    syms[4]; uint32_t           //フラグのビット4または5がセットされている場合
    memMapLength; uint32_t      合
    memMapAddress; uint32_t      //フラグのビット6がセットされている場合
    drivesLength; uint32_t       //フラグのビット6がセットされている場合
    drivesAddress; uint32_t      //フラグのビット7がセットされている場合
    configTable; uint32_t        //フラグのビット7がセットされている場合
    apmTable;                    //フラグのビット8がセットされている場合
    uint32_t                     //フラグのビット9がセットされている場合
    vbeControlInfo; uint32_t      //フラグのビット10がセットされている場合
    vbeModeInfo; uint32_t        //フラグのビット11がセットされている場合
    vbeMode; uint32_t           //フラグのビット12が設定されていれば、すべて
    vbeInterfaceSeg; uint32_t    のvbe_*が設定される
    vbeInterfaceOff;
    uint32_t vbeInterfaceLength;
} multiboot_info, *pmultiboot_info;
```


この構造は、見た目ほど複雑ではありません。**flags**メンバの対応するビットがセットされていれば、上に示したメンバが有効であることを意味します。このため、技術的には、**flags**が唯一の必須メンバであり、他のメンバはすべてオプションです。

ここにいるメンバを見てみましょう。

- **memLow, memUpper:**低い方のメモリと高い方のメモリの量（単位：KB）。下位メモリは0から、上位メモリは1MBから。
- **bootDevice.**ブートデバイス（後述）
- **commandLine:**カーネルのコマンドラインを含むC言語の文字列へのポインタ
- **moduleCount.**ブートローダーでロードされた追加ブートモジュールの数
- **moduleAddress:**最初のモジュール構造体のアドレス（後述）
- **syms:**シンボルテーブルの位置。以下を参照 **memMapLength:**シ
- ステムメモリマップのエントリ数 **memMapAddress:**メモリマッ
- プのアドレス
- **drivesLength, drivesAddress:** 以下参照
- **configTable:**BIOS ROMコンフィグテーブルのアドレス（GET CONFIGURATION BIOS INTコールから返される
- **apmTable:**アドバンストパワーマネージメント（APM）テーブルのアドレス
- **vbeControlInfo, VbeVbeModeInfo:Video Bios Extensions (VBE)** 構造体のアドレスです。
- **vbeMode:**VBEモード
- **vbeInterfaceSeg, vbeInterfaceOff, vbeInterfaceLength.** VBE 2.0のプロテクトモードのインターフェースにアクセスするために使用します。

本章ではVBEやAPMについては触れていませんので、ここでは割愛します。メモリーマップについては、[第17章](#)でシステムメモリーマップのフォーマットを含めて説明しました。

configTableのROM構成は、[BIOSINT0x15Function0xC0](#)から得られるテーブルです。

それがすべてです。この構造は悪くないですね :) **bootDevice**、**moduleAddress**、**syms**、**drivesLength**、**drivesAddress**など、まだ見ていないメンバがいくつかあります。これらを詳しく見ていきましょう。

ブートデバイス

bootDeviceメンバは、以下のフォーマットに従いま

- す。1ワード目：BIOSドライブ番号
- 2語目、3語目、4語目パーティション

BIOSのドライブ番号は、BIOSのINT 0x13サービスでドライブを表すために使用される番号です。その他の単語はパーティションを表します。単語2,3,4はパーティション1,2,3を表します。パーティション1はトップレベルのパーティションで、パーティション2はその中のサブパーティションです。未使用のパーティションは0xFFと表示されます。

moduleAddress

これは、最初のモジュール構造体へのポインタです。モジュール構造体のエントリは、以下のフォーマットに従います。

moduleStartと**moduleEnd**は、ロードされたモジュールの開始と終了のアドレスを含みます。string "はそのモジュールを表し、通常はコマンドライン名やパス名、または何もない場合は "0 "となります。

```
uint32_t moduleStart;
uint32_t moduleEnd;
char *string;
```

drivesLengthメンバには、すべてのドライブ構造体のサイズが含まれています。 **drivesAddress**には、最初のドライブ構造体へのポインタが含まれています。ドライブ構造体のエントリは、以下の形式を持ちます。

```
typedef struct _DRIVE_ENTRY
{
    ...
    uint32_t size;           //構造体のサイズ
};
```

```

uint8_t driveNumber;
uint8_t driveMode;
uint16_t driveCylinders;
uint8_t driveHeads;
uint8_t driveSectors;
uint8_t ports [0];           //任意の数の要素が可能
}drive_entry, *pdrive_entry;

```

それでは、各メンバーを詳しく見ていきましょう。

- **driveNumber**: BIOSで使用される番号
- **ドライブモード**。
 - 0: CHS
 - 1: LBA
- **ドライブシリンダー、ドライブヘッド、ドライブセクタ**。ドライブジオメトリ
- **ports**: ドライブへのアクセスにBIOSが使用するI/Oポート番号のリストを含み、0で終了します。

これがこの構造のすべてです。あと一人のメンバーを取り上げます。あの奇妙な**syms**のメンバーです。

シーム

本章の構造体では、**syms**メンバは**uint32_t syms[4]**と宣言されていますが、これは完全には正しくありません。実際にはいくつかのメンバーがあり、それらのメンバーに続くバイトを占有しています。

- syms[0] = uint32_t sym_num
- syms[1] = uint32_t sym_size
- syms[2] = uint32_t sym_addr
- syms[3] = uint32_t sym_shndx

仕様書には、OSイメージのフォーマットは何でもいい（ELF、PE、フラットバイナリなど）と書かれていますが、これはELFフォーマットに限った話です。技術的には、（カーネルのような）システムイメージは、それ自体を解析してシンボル情報を得ることができます。**sym_num**はELFセクションヘッダのシンボルエントリの数、**size**は各エントリのサイズ、**addr**はELFバイナリのシンボルテーブルのアドレスです。

結論

それがマルチブート規格のすべてです。技術的には、マルチブート・コンプライアント・ブートローダでシステムを起動するために、**マルチブート・ヘッダー**を適切に定義することだけが必要です。しかし、**マルチブート情報構造**を使って、通常はブート時に取得する情報を取得することができます。

シリーズでマルチブートをサポートしたい場合は、カーネルが仮想アドレスではなく**物理**アドレスでロードされるようにする必要があります。これは、マルチブート・コンプライアント・ブートローダがあなたのカーネルに制御を移すとき、ページングが無効になるからです。典型的なアドレスは**1MB**で、これは**GRUB**のような多くのブートローダでロードできます。もちろん、ページングを有効にして後から使用することもできます :)

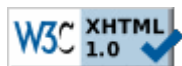
次の機会まで。

マイク

BrokenThorn Entertainment 社。現在、**「DoE」**と**「Neptune Operating System Software Suite」**を開発中。

ご質問やご意見がございましたらお気軽に**お問い合わせ**ください。

ホーム





オペレーティングシステム開発シリーズ

IA32機械語

by Mike, 2011

はじめに

この章では、**IA32機械語プログラミング**について説明します。ここで説明する内容は情報提供を目的としたものであり、基本的なオペレーティングシステムや実行ソフトウェアの開発には必要ありません。**IA32**（および**IA64**）の命令フォーマットを理解することは、不適切にアセンブルされた命令のデバッグや、**v8086**モードをサポートするために必要な**v86**モニター、命令のエミュレーション（特定のFPU命令のエミュレーションや、アセンブラ、エミュレータ、仮想マシンなどの開発時に必要）、デバッガやコンパイラなどの特定のシステムソフトウェアの開発に役立ちます。

この章では、新章の執筆に使用する新しいエディターをテストするためのもので、フォーマットの改善やスペルミスの解消に役立つはずですが、このテストが成功すれば、新章と旧章のすべてが新しいフォーマットに合わせて更新されます。もしエラーがあった場合は、ご意見をお寄せください。

機械語の概要

機械語は、**マシンコード**、**ネイティブコード**、**バイトコード**などとも呼ばれ、**中央処理装置（CPU）**で実行可能な生の命令とデータのセットです。機械語は、CPUが特定のバイトシーケンスを、タスクを実行するための「命令」として解釈することを可能にする。これらのタスクは、少量のデータをコピーしたり、演算したりといった非常に小さなものです。CPUの命令を表すバイト列を構築する行為は、**コーディング**と呼ばれる。**コーディング**の定義は、プログラミング言語の進化に伴って変化してきた。当初は、命令のバイト列を実際にコーディングすることを指していたが、現在では、**第2世代**、**第3世代**、**第4世代**のプログラミング言語によるさまざまな形態のプログラミングを指す。コンピュータ**プログラム**は**ソフトウェア**とも呼ばれ、ワープロやHalo®のプレイなど、複雑な作業を行うための機械コードとデータの集合体である。一般的なメディアでは、機械語は「1と0の連続」と解釈されることが多い。これは、ある程度は正確な表現です。

デジタル・ロジック

デジタルロジックとは、**論理ゲート**を利用して電子機器に判断をさせる電子工学の分野です。論理ゲートの例としては、**ANDゲート**、**ORゲート**、**NORゲート**、**NANDゲート**、**NOTゲート**、**XORゲート**などがある。これらのゲートは、2進法の演算を反映している。ANDゲートは2値のANDを、XORゲートは2値のXORを、というように。これらのゲートに意味を持たせるためには、何が**真**で何が**偽**なのかを理解するための規格を採用する必要があった。例えば、ANDゲートは、2つの入力と1つの出力がある場合のみ意味を持つ。2つの入力は2つの項目で、どちらかが偽であれば出力は偽、そうでなければ出力は真となります。電流の少ない線を**偽**、電流の多い線を**真**と定義するのが定番だ。これが2進法とデジタルロジックの関係である。2進法では、**0は偽**、**1は真**と表記されることが多い。機械語が2進数で表現されることが多いのは、CPUの命令解釈の仕組みや、RAMに格納して命令を取得する仕組みと密接に関係しているからだ。

プログラムの読み込み

プログラムは、オペレーティングシステム、エグゼキュティブ、またはファームウェアによってメモリにロードされます。**IA32**および**IA64**ファミリーのCPUは、**ROM（Read Only Memory）**や**RAM（Random Access Memory）**からプログラムを実行することができます。これは、ファームウェアとプログラムイメージが共有する**システムバス**と**物理アドレス空間（PAS）**によって実現されている。ファームウェアとプログラムイメージは、どちらもCPUコアによって直接実行されるため、同じ機械語のバイトコードを使用します。機械語は、ファームウェアが使用する**マイクロコード**（第7章参照）とは異なるが、実際のファームウェアは機械語であることに変わりはない。

アセンブリ言語

アセンブリ言語は、**第二世代のプログラミング言語**です。アセンブリ言語では、プログラマーがソフトウェアを開発する際に役立つ**ニーモニック**を使用して、明確に定義された言語でプログラムを書くことができます。例えば、「**MOV**」は、さまざまなアーキテクチャに対応した多くのアセンブリ言語で共通のニーモニックです。**MOV**は、データをソースからデスティネーションにコピーする命令を表します。また、**データ移動命令**の一例でもあります。

ニーモニックは、命令や命令形式に象徴的な名前を与え、各アセンブリ言語命令を単一の（場合によっては複数の可能性のある）機械語バイトシーケンスに変換できるようにした。アセンブリ言語の命令を機械語に変換するプログラムは、**アセンブラ**と呼ばれる。アセンブラは誤って**コンパイラ**と呼ばれることもある。

機械命令の概要

機械命令は、CPUの特定のタスクを実行する1バイトのシーケンスです。マシンインストラクションのセットは、以前は**マシンランゲージ**として定義されていた。機械語命令は、CPUメーカーが実装したすべてのCPU命令を記録した**命令セット**で定義されています。命令セットには、アセンブラ開発者が使用するための、示唆に富むアセンブリ言語のニーモニックも含まれているのが一般的です。命令セットはCPUの仕様書に記載されています。

CPUメーカーは、特定のCPUがサポートする機械命令を実装し、CPUが各命令をどのように解釈するかを決定します。これにより、CPUはメーカーが意図した機械語命令を「実行」することができます。しかし、CPUのハードウェアやファームウェアにバグがあると、有効な命令ではない命令をCPUが「実行」してしまうことがあります。これが**文書化されていない命令**です。アセンブラによっては、よく知られている文書化されていない命令にニーモニックを定義している場合があります。文書化されていない命令の中には、後に実際の命令として文書化されたものもあります。**IA32**の**LOADALL**命令のように、メーカーのテスト用として文書化されていない命令もあります（このバグはその後修正されました）。また、システムを停止させたり、CPUにダメージを与えたりするような悪い影響を与える命令もあります（これらは**HCF（Halt and Catch Fire）**命令として知られています）。

すべてのCPUアーキテクチャに対応した命令セットがあります。ソフトウェア業界の進化に伴い、命令セットにはある種の傾向が見られるようになりました。これらの傾向を理解することは、**IA32**の機械語を理解するのに役立ちます。

CISCとRISC

命令セットは、一般的に2つのカテゴリーに分類されます。**CISC(Complex Instruction Set Computing)**と**RISC(Reduced Instruction Set Computing)**です。RISCの例としては、PPCやARMのアーキテクチャがあります。CISCの例としては、IA32アーキテクチャがあります。RISCアーキテクチャは、CISCよりも単純な命令セット形式を採用しています。RISCアーキテクチャは通常、各命令に標準的なエンコーディングフォーマットを使用し、各命令が同じバイト数になるようにします。CISCアーキテクチャもまた、標準的なエンコーディングフォーマットに従いますが、可変長の命令が可能です。

操作コード

オペコード (OPCODE) とは、CPUが命令の種類を判断するために利用する1バイトの識別子です。例えば、MOV命令には、タイプ (MOV) などの命令に関する情報をCPUに知らせるオペコード識別子があります。多くの命令セットでは、1つの命令を別の命令と区別するためにオペコードを使用します。一部の命令は、**マルチバイトオペコード**や**拡張オペコード**を持つことがあります。これにより、命令セットの柔軟性が高まります。

アドレッシングモード

アドレスモードとは、CPUが**アドレス**を参照するための方法を定義するものです。アドレスは、アーキテクチャによって、仮想的なものと物理的なものがあります。**データ移動命令**などの命令は、データを取得するためにアドレスを参照する方法をCPUに伝える必要があります。例えば、多くのCPUは**ダイレクトアドレッシングモード**をサポートしており、特定のアドレスのデータを参照 (読み書き) するように命令がCPUに指示することができます。例えば、IA32のアセンブリ言語では

```
mov eax, dword [0xa0000].
```

この命令は、CPUに**直接アドレッシングモード**を使用して、現在のアドレス空間のアドレス0xa0000から読み取るように指示します。もうひとつの一般的なアドレッシングモードは**間接アドレッシング**で、ポインタを使ってデータを参照するようにCPUに指示することができます。例えば、IA32のアセンブリ言語では

```
mov eax, [ebp].
```

これにより、CPUはEBPレジスタに格納されているアドレスからEAXレジスタにデータを読み込むことになります。この他にも、アーキテクチャによって様々なアドレッシングモードが存在します。

IA32とIA64の命令コード

さて、ここからはIA32とIA64の機械語命令のエンコーディングを見ていきましょう。ここでは、スペースを節約するために、IA32とIA64の命令セットという意味でIA64を使用します。IA32はIA64のサブセットであり、IA32の命令セットの大部分を共有しています。IA64の命令セットは、CISCエンコーディングを実装しています。これは、各命令が特定のエンコーディング構造に従っており、長さが可変であることを意味する。IA32とIA64の命令は、1バイトから12バイトまでの大きさがあります。

登録コード

CPUは内部のレジスタを数値で識別している。多くのレジスタは同じコードを共有していますが、CPUは使用する命令や現在の動作モード (リアルモード、プロテクトモード、ロングモード) に応じて使用するレジスタを決定します。使用するレジスタを決定する際には、オペランドサイズオーバーライドプレフィックスも使用されます。このプレフィックスについては後述します。

レジスタコードは、命令がどのレジスタを操作するのかをCPUに知らせるために、命令のエンコーディングで使用されます。レジスタには以下のコードが使われています。

REX.r = 0

コード	0	1	2	3	4	5	6	7
REXなし	AL	CL	DL	BL	AH	CH	DH	BH
REG16	AX	CX	DX	BX	SP	BP	SI	DI
REG32	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
REG64	RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
MM	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
XMM	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
YMM	YMM0	YMM1	YMM2	YMM3	YMM4	YMM5	YMM6	YMM7
SSEG	ES	CS	SS	DS	ES	GS		
	CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
	DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7

例えば、**mov bx, 0x5**という命令では、BXのレジスタコードとして3を格納します。**mov ss, ax**という命令では、SSのレジスタコードとして2を、AXのレジスタコードとして0を格納する必要があります。命令によって使用するレジスタの種類が異なるため、同じコードの複数のレジスタを選択しなければならないという矛盾が生じることはありません。例えば、**mov REG16, IMM16**という命令は、常に16ビットの汎用レジスタをオペランドとして使用します。また、**movups xmm, xmm/m128**という命令は、常にXMMレジスタのみを使用します。

ロングモードでは、このリストにさらにレジスタが追加されます。上記のレジスタに対応するために、ロングモードでは、同じレジスタコードを使って他のレジスタを選択する命令を可能にする特別なフラグが設定されている。これが後に説明する**REXプレフィックスバイト**の**REX.r**フィールドである。このビットがセットされていると、レジスタテーブルは次のようになります。

REX.r=1

コード	0	1	2	3	4	5	6	7
REXなし	R8B	R9B	R10B	R11B	R12B	R13B	R14B	R15B
REG16	R8W	R9W	R10W	R11W	R12W	R13W	R14W	R15W
REG32	R8D	R9D	R10D	R11D	R12D	R13D	R14D	R15D
MM	R8	R9	R10	R11	R12	R13	R14	R15
XMM	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
YMM	XMM8	XMM9	XMM10	XMM11	XMM12	XMM13	XMM14	XMM15
SSEG	YMM8	YMM9	YMM10	YMM11	YMM12	YMM13	YMM14	YMM15
	ES	CS	SS	DS	FS	GS		
	CR8	CR9	CR10	CR11	CR12	CR13	CR14	CR15
	DR8	DR9	DR10	DR11	DR12	DR13	DR14	DR15

命令のエンコーディング

IA64の命令は、CPUの8085に由来する明確な構造を持っています。各命令は次のような形式になっています。

プレフィックス・バイト (0-4)	REXプレフィックス (1)	操作 (0-3)	Mod R/M (1)	SIB (1)	排気量 (0-4)	即時 (0-4)
-------------------	----------------	----------	-------------	---------	-----------	----------

コンパクトにするために、 () 内の数字はコンポーネントのバイト数です。0の数字は、そのバイトがオプションであることを示します。例えば、1つの命令の中で、**プレフィックス・バイト**は0から4バイトまであります。つまり、1つの命令には、0、1、2、3、4の**プレフィックス・バイト**が存在することになります。**REXプレフィックス**は、**IA64およびlongモードでのみ有効です**。唯一の必須フィールドは**オペレーションコード**です。他のフィールドはすべてオプションで、命令がそれらを必要とするかどうかには依存します。例えば、**INT (割り込み)** 命令では、操作コードと即値バイトが必要ですが、**MOV**命令では上記のフィールドをすべて使用する場合があります。

例として、INT命令をもう少し詳しく見てみましょう。INT命令には形があります。

INT imm8

ここで、imm8は8ビットの即値、INTは演算コード0xCDのニーモニックです。命令エンコードのフォーマットを知っていると、INT 5命令を次のようにエンコードすることができます。

0xCD 0x05

1バイト目の0xCDはオペレーションコードで、茶色で表示されています。プレフィックスバイトはオプションであり、INT5では必要ないため、必要ありません。Mod R/MとSIBのバイトも必要ありません。ディスプレイメントは**メモリアドレッシングモード**でのみ使用されるので、他に必要なフィールドは即値フィールドだけです。即時フィールドは、0〜4バイトのフィールドです。INT imm8という命令形式があるので、1バイトのフィールドとして使うことができます。この例の目的は、あるフィールドはオプションであり、必要とされないことを示すことです。また、これらのフィールドの順序は決して変わりません。例えば、上の例では、不要なフィールドを省略することにしましたが、演算コードフィールドが即値フィールドの前にあるという、フィールドの順序はそのままです。次のセクションでは、これらの各フィールドについて詳しく説明します。

プレフィックス・フィールド

プレフィックス・バイトは、命令がCPUに対してより多くの情報を与えることを可能にします。例えば、CPUにバスをロックさせたり、データ移動命令で別のセグメントレジスタを利用させたりすることが可能になります。これらのプレフィックスの多くは、アセンブリ言語のニーモニックを持っています。プレフィックス・バイトは4つのクラスに分類されています。**1つの命令で使用できるプレフィックス・バイトは、4つのクラスのそれぞれから最大1つです**。

クラス 1 プレフィックス

クラス 0xF0 LOCK プ

レフィックス

0xF2 REPNE, REPZ 接頭辞

0xF3 REP, REPZ, REPE 接頭辞

クラス2プレフィックス

0x2E CS セグメントオーバー

ライド 0x36 SS セグメントオ

ーバーライド 0x3E DS セグメ

ントオーバーライド 0x26 ES

セグメントオーバーライド

0x64 FS セグメントオーバー

ライド 0x65 GS セグメントオ

ーバーライド

クラス3のプレフィックス

0x66 オペランドサイズオーバーライド

クラス4のプレフィックス

0x67 アドレスサイズオーバーライド

ここでは、読者がIA32アセンブリ言語を知っていることを前提としているので、これらのプレフィックスについての詳細な説明は省略します。1つの機械語命令には、4つのクラスのうち、1つのプレフィックス・バイトしか設定できません。4つのクラスがあるため、1つの命令は0〜4個のプレフィックス・バイトを持つことができます。1つの命令が1つのクラスから2つ以上のプレフィックス・バイトを使おうとすると、CPUは無効な命令の例外を発生させます。

LOCKプレフィックス

LOCKをサポートしていない命令に **LOCK** プレフィックスを使用した場合、CPU は無効な命令の例外を発生させます。アセンブラの中には、無効な命令に LOCK を使用することをプログラマに警告せずに許すものがあります。そのため、ここでは、有効な命令のリストを紹介します。

LOCKプレフィックスは以下の命令でのみ使用可能です。ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, XOR。

オペランドサイズオーバーライド

オペランドサイズオーバーライドは、CPUが16ビットと32ビットのオペランドを選択できるようにするものです。アセンブラは通常、**bits16**や**use32**などの指示を使って間接的にプログラマが特定のオペランドサイズを選択できるようにしています。**IA32**と**IA64**の命令セットでは、レガシーの16ビットとネイティブの32ビットの2種類のオペランドサイズが用意されています。ネイティブサイズは、プロセッサの現在の動作モードに依存します。

動作モード	CS.d	REX.W	ネイティブ	オペランドのオーバーライド
リアルモード			16ビット	16ビット
V8086モード			16ビット	16ビット
プロテクトモード	0		16ビット	32ビット
プロテクトモード	1		32ビット	16ビット
ロングモード		0	32ビット	16ビット

例として、ADD AX/EAX, IMM16/IMM32命令を見てみましょう。この命令のオペレーションコードは0x05です。プロテクトモードのコードでは、CPUはデフォルトでこの命令をADD EAX, IMM32命令として解釈します。しかし、オペランドオーバーライドプレフィックスを使用することで、デフォルトの動作をオーバーライドし、16ビットの即値をコピーすることができます。アセンブリ言語では次のようにします。

```
add eax, 5 ; MOV EAX, IMM32
add ax, 5 ; MOV AX, IMM16
```

最初の指示では、組み立てを行います。

```
0 x 05 0 x 05 0 x 00 0 x 00 0 x 00
```

2回目の指示では、組み立てを行います。

```
0 x 66 0 x 05 0 x 05 0 x 00
```

この2つの命令の違いは、次の点だけです。(1) 最初の命令は32ビットの即値を使用し、2番目の命令は16ビットの即値を使用していること（これらは赤で表示）、(2) 2番目の命令はオペランドオーバーライド接頭辞を使用していること（これは黒で表示）。これは、16ビットのオペランド形式を使用するようにCPUに指示するものです。念のため、茶色の値はオペレーションコードです。

アドレスサイズオーバーライド

アドレスサイズオーバーライドのプレフィックスバイトは、オペランドオーバーライドのプレフィックスバイトとよく似ています。アセンブラでは、byte ptrやdword ptrなどのキーワードを使って、プログラマがアドレスサイズを選択できるようになっています。この機能はオペランドオーバーライドプレフィックスと非常に似ているため、目的は同じですがアドレスモードに適用されるため、説明を省略します。

動作モード	CS.d	REX.W	ネイティブ	アドレスオーバーライド
リアルモード			16ビット	16ビット
V8086モード			16ビット	16ビット
プロテクトモード	0		16ビット	32ビット
プロテクトモード	1		32ビット	16ビット
ロングモード		0	64ビット	32ビット
ロングモード		1	64ビット	32ビット

例えば、mov eax, [0xa000]という命令をプロテクトモードでアセンブルした場合、アドレスサイズのオーバーライドは必要ありません。アセンブラは0xa000を32ビットの変位として扱います。しかし、mov ax, word [0xa000]を使用した場合、アセンブラは16ビットのアドレス形式を選択するために、命令にアドレスサイズオーバーライドのプレフィックスを追加します。

REXプレフィックス

REXプレフィックスは、64ビット特有の機能を有効にします。以下のような形式になっています。



REX.wOperandのサイズ。0 : デフォルト、1 : 64ビット REX.rModRM.regの拡張子 REX.xSIB.インデックス拡張 REX.bModRM.rm拡張

プレフィックスオーダー

他のプレフィックスバイトと組み合わせて使用する場合、プレフィックスバイトの順序は重要ではありません。たとえば、マシンコードで0xF3 0x2Eを使用すると、REPとCSのオーバーライドを選択できます。また、0x2E 0xF3を使っても同じことができます。

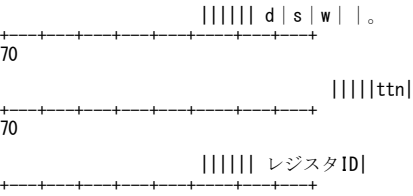
操作コード欄

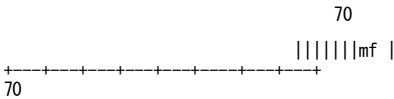
オペレーションコードフィールドは1~3バイトの長さになります。すべてのオペレーションコードはユニークで、使用する命令とそのオペランドを識別します。例えば、オペレーションコード0は、ADD REG/MEM8, REG8命令を示します。オペレーションコード1は、ADD REG/MEM16/MEM32, REG16/REG32命令を識別します。IA32およびIA64のCPUマニュアルに、各命令とそのオペレーションコードの概要が記載されています。

プライマリOpcode

プライマリオペコードは、すべての命令で必要とされる1バイトです。命令を識別するためのオペレーションコードフィールドのベースとなるものです。一次オペ

コードは、命令によって次のような形式があります。





PO.wオペランドサイズ
PO.sSign拡張
PO.dDirection
PO.ttn一部のFPU命令で
使用される PO.mfメモ
リフォーマット

セカンダリOPCコード

プライマリオペコードバイトが0xf0の場合、セカンダリオペコードバイトと呼ばれる別のバイトが続きます。セカンダリオペコードは、異なる命令を識別するもので、上記と同じ機能を持っています。これらは2バイトのオペコードとして扱われます。

OPCode拡張

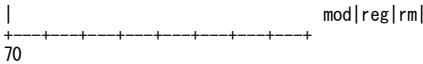
ある種の命令ファミリーは、同じオペコードを持ちながら、オペコードエクステンションと呼ばれる特別なフィールドによってのみ異なる。これは、Mod R/M.regフィールドに格納されている3ビットの拡張子です。Mod R/Mバイトについては、次のセクションで詳しく説明します。

マルチバイトOPCコード

プライマリオペコードフィールドの長さは1~3バイトです。ほとんどの命令はプライマリ・オペコード・フィールドの1バイトしか使用しませんが、中には3バイトすべてを使用するものもあります。これらの命令はすべて、セカンダリ・オペコード・バイト (0xf0) も使用します。

Mod R/Mフィールド

Mod R/M (Register/Memory) フィールドは、メモリまたはレジスタのオペランドを必要とする命令で使用されます。Mod R/Mフィールドのフォーマットは以下の通りです。



Mod R/Mフィールドは、CPUがリアルモード、プロテクトモード、ロングモードのいずれで動作しているかによって若干異なります。

リアル

モードProtected and Longモード

ModRM.mod	00 : 【メモリ 01 : 【メモリ+DISP8 】の場合 10 : 【メモリ+DISP16 】の場合 11: 登録	ModRM.mod	00 : 【メモリ 01 : 【メモリ+DISP8】の場合 10 : 【メモリ+DISP32】の場合 11: 登録
ModRM. regRegister codeModRM. rmIf ModRM.mod = 11: レジスタコー ド	000: [BX+SI]. 001: [BX+DI]. 010: [BP+SI]の場合 011: [BP+DI] 100: [SI]の場合 101: [DI]の場合 110: [BP]または[DISP16] ModRM.mod=0のと き 111: [BX]となります。	register code ModRM. rmIf ModRM.mod = 11: レジスタコー ド	000: [RAX]の場合 001: [RCX]の場合 010: [RDX]を使用して います。 011: [RBX]を使用して います。 100: [SIB]の場合 101: [rbp][disp32]. 110: [RSI]の場合 111: [RDI]の場合

ModRM.modフィールドは、Mod.rmフィールドと組み合わせてアドレッシングモードを決定します。例えば、**mov ax, [0xa000]**という命令は、（リアルモードでは）ModRM.mod = 0 (Memory)、ModRM.rm = 6 (DISP16) を使用します。ModRM.regにはAXのレジスタコードが入ります。**mov ax, [bx+0xa000]**を使用する場合、ModRM.modは2 (Memory+DISP16)、ModRM.rmは7となります。アセンブラは、0xa000がワードサイズの変位であることから、ここでは0xa000をDISP8ではなくDISP16として扱います。アドレスサイズオーバーライドプレフィックスを使用して、DISP8形式を選択することができます。

上記の表を見ると、間接アドレッシングに使えるレジスタはあまり多くないことが推測できます。例えば、**[BP]**は有効なアドレッシングモードではありませんが、アセンブラは**mov ax, [bp]**のような命令でうまく組み立てることができます。これらのアセンブラがよく使うトリックは、ModRM.mod = 1 (Memory+DISP8) と ModRM.rm = 6 (BP) を設定することです。つまり、アセンブラはこれを**[BP+DISP8]**アドレッシングモードに変換し、ディスプレイメントを0に設定します。そのため、**mov ax, [bp]**は**mov ax, [bp+0]**にアセンブルされます。

プロテクトモードとロングモードには、より多くの機能を提供するもう一つのアドレッシングモード、**[SIB]**が導入されています。SIBアドレッシングモードは、ModRM.rmモードと組み合わせることができます。例えば、プロテクトモードで**mov eax, [ebx+edi*2+0xa000]**は、ModRM.mod = 2 (Memory+DISP32)、ModRM.rm = 4 (SIBバイト) と変換されます。SIBバイトは、この命令で**edx+edi*2**をどのように抽出するかを示すもので、次のセクションで説明します。

一部の命令では、拡張オペコードフィールドを使用しています。拡張オペコードとは、命令を識別する際に**プライマリ・オペコード**と共に使用される識別子です。これは3ビットのフィールドで、これらの命令のMod R/M.regフィールドに格納されます。拡張オペコード・フィールドを使用する命令は、レジスタ・オペランドやメモリ・アドレッシング・モードを格納するために、ModRM.rmとModRM.modを使用する場合があります。

SIBフィールド

SIB (Scale Index Base) バイトは、Mod R/M.rm = 4で、CPUがプロテクトモードまたはロングモードの場合のみ、Mod R/Mバイトの後に続きます。このバイトは、IA32およびIA64アーキテクチャに追加のアドレッシングモードを提供します。SIBアドレッシングは、Mod R/Mアドレッシングと組み合わせ

ることで、幅広いアドレッシングモードを実現することができます。



SIB.Scale	00	: ファクター1
	01	: ファクター2
	10	: ファクター4
	11	: ファクター8
SIB.Index	標準のレジスタコードを使用	
	VSIBの場合、VRレジスタコードを使用	
	REX. x = 1の場合、64ビットのレジスタコードを使用	
	REX. x=1でVSIBの場合、VRレジスタコードを使用	
SIB.Base	標準のレジスタコードを使用	
	REX. b=1の場合、64ビットのレジスタコードを使用	

SIBバイトのレジスタフィールドの名前にもかかわらず、技術的にはどのようなレジスタコードも使用することができます。例えば、SIB.Baseにインデックスレジスタを入れることができます。

SIBバイトとMod R/Mを再び組み合わせて、どのように動作するかを説明します。先ほどの例では、**mov eax, [ebx+edi*2+0xa000]**とします。この例では、CPUはプロテクトモードで動作しているものとします。EAXは非メモリレジスタなので、Mod R/M.regに入ります。また、Mod R/M.mod = 2で[Memory+DISP32]を有効にし、Mod R/M.rm = 4で[SIBバイト]を有効にする必要があります。**EBX**はベースレジスタ、**EDI**はインデックスレジスタです。EBXのレジスタコードは3、EDIのレジスタコードは7です。これを利用して、SIB.index = 7、SIB.base = 3に設定します。スケールファクターの**2**はSIB.scaleに入ります。

これをまとめると、Mod R/Mのバイトは**10 000 100**のバイナリ、SIBのバイトは**10 111 011**のバイナリとなります。32ビットのディスプレイメントが**0xa000**、オペレーションコードが**0x89**であることから、例題の命令を次のように翻訳することができます。

0x89 0x84 0xbb 0x00 0xa0 0x00 0x00

これは**mov eax, [ebx+edi*2+0xa000]**の正しい変換になります。読みやすくするために、演算コードは**茶色**、Mod R/MとSIBのバイトは**赤色**、ディスプレイメントフィールドは**黒色**で表示しています。これは、命令の正確なフォーマットに従っていることに注意してください：最初に主オペコード、次にMod R/Mバイト、次にSIBバイト、そして変位バイトが続きます。

上の命令の変位バイトが奇妙に見える場合は、IA32およびIA64アーキテクチャが**リトルエンディアン**であることを考慮してください。

変位分野

ディスプレイメントフィールドは、Mod R/M.modがモード1(Memory+DISP8)またはモード2(Memory+DISP16またはMemory+DISP32)の場合のみ有効です。ディスプレイメントは、バイト、ワード、または、ワード値で、Mod R/MやSIBバイトと組み合わせて、アドレッシングモードにディスプレイメントを追加するために使用されます。ディスプレイメントフィールドは、常にMod R/MまたはSIBバイトの後に続きます。

直前のフィールド

即値フィールドは、命令がオペランドとして要求する場合にのみ有効です。命令は8、16、または32ビットの即値を必要とする場合があります。このフィールドは、命令の最後のフィールドとして存在する必要があります。16ビットと32ビットの両方の値を許容する命令では、オペランドオーバーライドサイズプレフィックスが存在するかどうかによって、このフィールドのサイズを決定します。

インストラクションテーブル

あるソフトウェアでは、命令の機械語翻訳を容易にするために、**命令ルックアップテーブル**が利用されています。このテーブルは、すべての命令のオペレーションコードとオペランドタイプを提供する汎用命令表を反映したものです。IA32のマニュアルやオンラインリファレンスなどを参考にして、テーブルを作成したり、機械語命令の作成に役立てたりします。これらのテーブルの設計はかなり異なります。これらのルックアップテーブルの読み方については、ドキュメントを読むことが重要です。

テーブルには、以下のような形で指示が表示されます。

0x10 | ADC | R/MEM8 | R8 | となっています。

これは、オペレーションコードが0x10のADC命令を表しています。R/MEM8が第1オペランド、R8が第2オペランドとなります。オペランドはテーブルのデザインによってさまざまな方法で表現されます。また、このテーブルには、命令セットに影響を与えるフラグ、命令が使用するオペコードバイトの形式（オペコードフィールドにレジスタIDを格納する場合など）、サポートされるプロセッサなどの追加情報が提示されることもあります。これらのテーブルは、サイズが非常に大きくなる場合がありますが、いずれも通常上記の形式で提示される基本的な情報を提供しています。

上記の「R/MEM8」は、第1オペランドが「レジスタ」または「8ビットのメモリロケーション」であることを意味しています。R8は、第2オペランドが8ビットのレジスタであることを意味しています。**命令がメモリオペランドを持つ場合は、Mod R/M（場合によってはSIBバイトも）が続く必要があります。**また、命令が**2つのレジスタオペランドを取る場合は、Mod R/Mバイトが続く必要があります。**Mod R/Mバイトは、Mod R/M.rmとMod R/M.regのメモリアドレスモード情報または両方のレジスタコードを格納します。CPUは、オペコードにより、レジスタコードが8ビットレジスタであることを認識します。**オペコードは、実行する命令だけでなく、その命令がどのようなオペランドを必要とするかをCPUに伝えます。**1つの命令は異なるタイプのオペランドを使用することができ、そのため同じ命令が複数のオペコードを占めることがあります。例えば、上記の命令形式ではオペコード0x10を使用しています。その他の形式としては、以下のようなものがありますが、これらに限定されるものではありません。

0x11	ADC	R/MEM16/MEM32	R16/REG32
0x12	ADC	R8	R/REG16/REG32
0x13	ADC	R16/REG32	R/MEM16/MEM32

命令が**REG16/REG32**のようなオペランドを使用する場合、**オペランドサイズオーバーライド**接頭辞があるかどうかと、現在のCPUの動作モード（プロテクトモードで動作しているか、リアルモードで動作しているか、ロングモードで動作しているか、など）に基づいて、使用するオペランドを推測する必要があります。例えば、**ADC ax, word ptr [0]**という命令をプロテクトモードで実行している場合（アセンブリ言語の用語で言うと**bits32**や**use32**命令を使用している場合）、この命令にはオペコード0x13を使用することができます。この命令は「**ADC REG16, MEM16/MEM32**」という形式であることがわかります。AXは第1オペランドで、16ビットのレジスタ（REG16）です。しかし、[0]とは何でしょうか？それを知るためには、アドレスサイズのオーバーライドがないことと、プロテクトモードであることを考慮します。アドレスサイズオーバーライドがないため、ネイティブサイズである32ビットのメモリアドレッシングを使用することになります。（詳細は、アドレスサイズオーバーライドのプレフィックスのセクションを参照してください。）これにより、**ADC REG16, MEM32**形式を選択することになります。（アドレス・サイズ・オーバーライドが存在する場合は、ADC REG16, MEM16形式を選択することになります。）

1つの命令が1つのレジスタ・オペランドしか持たない場合、オペランドがOPCode.regフィールドに格納されているかどうかを確認します。（一部の命令はスペースを節約するためにこのようにしており、**これがシングルバイト命令を可能にしています。**また、オペランドをOPCode.regとMod R/M.regまたはMod R/M.rmに格納するために2つのレジスタを利用する命令もあります。

この例を完成させるために、次のことを記します。OPCode 0x13、AXレジスタコード(0)、アドレッシングモードは[Memory]でディスプレイスメントは0。isがプロテクトモードであるため、32ビットのMod R/Mフォームを使用します。Mod R/M.reg = 0 (selecting AX)、Mod R/M.rm = 5 (DISP32)、Mod R/M.mod = 0 ([MEMORY])です。これにより、Mod R/Mの値は**00 000 101**の**バイナリになります**。SIB]モードを使用しないため、SIBバイトを使用する必要はありません。(SIBを使用する例としては、**mov eax, [ebx+edi*2+0xa000]**の逆アセンブルの例を参照してください)。)これらの情報をもとに、マシンコードを作成します。

0x66 0x13 0x05 0x00 0x00 0x00 0x00

OPCodeは**茶色**、Mod R/Mバイトは**赤色で表示されています**。ディスプレイスメントバイトはDISP32(Mod R/M.rmのため)なので、dwordでなければなりません。これは**黒**で識別されます。0x66は**オペランドサイズオーバーライドのプレフィックスで**、**青**で表示されています。REG32およびMEM32形式ではなく、REG16およびMEM16形式を選択するために、オペランドサイズオーバーライドのプレフィックスを使用します。プレフィックスを省略した場合は、以下のようになります。

0x13 0x05 0x00 0x00 0x00 0x00

プロテクトモードでは、これは**adc eax, dword ptr [0]**命令であり、望んでいたものではありませんでした。詳しくは、オペランドサイズオーバーライドの接頭辞の項をご覧ください。

ADC ax, word ptr [0] を **REP ADC ax, word ptr ES:[0]** にしたい場合は、ES オーバーライドプレフィックスと REP プレフィックスを使います。

0xf3 0x26 0x66 0x13 0x05 0x00 0x00 0x00

プレフィックスバイトの順序は重要ではありません。

リソース

以下のリソースは補助的な読み物として紹介されています。これらのリソースに対するサポートは行っておりませんのでご了承ください。

<http://ref.x86asm.net/>

IA32とIA64の命令表

<http://www.sandpile.org/>

命令フォーマットテーブル

http://wiki.osdev.org/X86-64_Instruction_Encoding

IA32とIA64の命令エンコーディング

結論

本章では、**IA32**および**IA64**アーキテクチャにおける機械語プログラミングと命令エンコーディングの概要を説明しました。本章では、デバッガやツールチェーンの開発を促進するために、新たな方法で資料を提供することを目的としています。また、本章は、特定の命令をエミュレートする際に、命令表を参考にすることができます。

ご質問やご意見がありましたらお聞かせください。

~Mike () です。

OS開発シリーズ編集部



オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - スキャンコード

by Mike, 2008

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

これは、すべてのスキャンコードを一覧にした資料です。keybordコントローラには、3つの定義されたスキャンコードセットがあります。

オリジナルXTスキャンコードセット

スキャンコードセット

KEY	MAKE	BREAK	-----	KEY	MAKE	BREAK	-----	KEY	MAKE	BREAK
A	1E	9E		9	0A	8A		[1A	9A
B	30	B0		`	29	89		INSERT	E0, 52	E0, D2
C	2E	AE		-	0C	8C		HOME	E0, 47	E0, 97
D	20	A0		=	0D	8D		PG UP	E0, 49	E0, C9
E	12	92		¥	2B	AB		DELETE	E0, 53	E0, D3
F	21	A1		BKSP	0E	8E		END	E0, 4F	E0, CF
G	22	A2		スペース	39	B9		PG DN	E0, 51	E0, D1
H	23	A3		TAB	0F	8F		U ARROW	E0, 48	E0, C8
I	17	97		CAPS	3A	BA		L ARROW	E0, 4B	E0, CB
J	24	A4		L SHFT	2A	AA		D ARROW	E0, 50	E0, D0
K	25	A5		L CTRL	1D	9D		R ARROW	E0, 4D	E0, CD
L	26	A6		L GUI	E0, 5B	E0, DB		NUM	45	C5
M	32	B2		L ALT	38	B8		KP /	E0, 35	E0, B5
N	31	B1		R SHFT	36	B6		KP *	37	B7
O	18	98		R CTRL	E0, 1D	E0, 9D		KP-	4A	CA
P	19	99		R GUI	E0, 5C	E0, DC		KP +	4E	CE
Q	10	90		R ALT	E0, 38	E0, B8		KP EN	E0, 1C	E0, 9C
R	13	93		APPS	E0, 5D	E0, DD		KP .	53	D3
S	1F	9F		ENTER	1C	9C		KP 0	52	D2
T	14	94		ESC	01	81		KP 1	4F	CF
U	16	96		F1	3B	BB		KP 2	50	D0
V	2F	AF		F2	3C	BC		KP 3	51	D1
W	11	91		F3	3D	BD		KP 4	4B	CB
X	2D	AD		F4	3E	BE		KP 5	4C	CC
Y	15	95		F5	3F	BF		KP 6	4D	CD
Z	2C	AC		F6	40	C0		KP 7	47	C7
0	0B	8B		F7	41	C1		KP 8	48	C8
1	02	82		F8	42	C2		KP 9	49	C9
2	03	83		F9	43	C3]	1B	9B
3	04	84		F10	44	C4		;	27	A7
4	05	85		F11	57	D7		'	28	A8
5	06	86		F12	58	D8		,	33	B3

6	07	87		PRNT SCRN	E0, 2A, E0, 37	E0, B7, E0, AA		.	34	B4
---	----	----	--	--------------	-------------------	-------------------	--	---	----	----

7	08	88		スクロ ール	46	C6		/	35	B5
8	09	89		PAUSE	E1, 1D, 45 E1, 9D, C5	-NONE				

ACPIスキャンコード

キー	コードを作る	ブレイクコード
パワー	E0, 5E	E0, DE
睡眠	E0, 5F	E0, DF
ウェイク	E0, 63	E0, E3

Windowsマルチメディアスキャンコード

キー	コードを作る	ブレイクコード
次のトラック	E0, 19	E0, 99
前のトラック	E0, 10	E0, 90
ストップ	E0, 24	E0, A4
再生/一時停止	E0, 22	E0, A2
ミュート	E0, 20	E0, A0
ボリュームアップ	E0, 30	E0, B0
ボリュームダウン	E0, 2E	E0, AE
メディア選択	E0, 6D	E0, ED
電子メール	E0, 6C	E0, EC
計算機	E0, 21	E0, A1
マイコンピュータ	E0, 6B	E0, EB
WWW検索	E0, 65	E0, E5
WWWホーム	E0, 32	E0, B2
WWW Back	E0, 6A	E0, EA
WWWフォワード	E0, 69	E0, E9
WWW停止	E0, 68	E0, E8
WWW リフレッ シュ	E0, 67	E0, E7
WWWお気に入り	E0, 66	E0, E6

最近のキーボードのデフォルトスキャンコードセット

スキャンコードセット

KEY	MAKE	BREAK	-----	KEY	MAKE	BREAK	-----	KEY	MAKE	BREAK
A	1C	F0, 1C		9	46	F0, 46		[54	F0, 54
B	32	F0, 32		`	0E	F0, 0E		INSERT	E0, 70	E0, F0, 70
C	21	F0, 21		-	4E	F0, 4E		HOME	E0, 6C	E0, F0, 6C
D	23	F0, 23		=	55	F0, 55		PG UP	E0, 7D	E0, F0, 7D
E	24	F0, 24		¥	5D	F0, 5D		DELETE	E0, 71	E0, F0, 71
F	2B	F0, 2B		BKSP	66	F0, 66		END	E0, 69	E0, F0, 69

G	34	F0, 34		スペース	29	F0, 29		PG DN	E0, 7A	E0, F0, 7A
H	33	F0, 33		TAB	0D	F0, 0D		U ARROW	E0, 75	E0, F0, 75
I	43	F0, 43		CAPS	58	F0, 58		L ARROW	E0, 6B	E0, F0, 6B
J	3B	F0, 3B		L SHFT	12	F0, 12		D ARROW	E0, 72	E0, F0, 72
K	42	F0, 42		L CTRL	14	F0, 14		R ARROW	E0, 74	E0, F0, 74
L	4B	F0, 4B		L GUI	E0, 1F	E0, F0, 1F		NUM	77	F0, 77
M	3A	F0, 3A		L ALT	11	F0, 11		KP /	E0, 4A	E0, F0, 4A
N	31	F0, 31		R SHFT	59	F0, 59		KP *	7C	F0, 7C
O	44	F0, 44		R CTRL	E0, 14	E0, F0, 14		KP -	7B	F0, 7B
P	4D	F0, 4D		R GUI	E0, 27	E0, F0, 27		KP +	79	F0, 79
Q	15	F0, 15		R ALT	E0, 11	E0, F0, 11		KP EN	E0, 5A	E0, F0, 5A
R	2D	F0, 2D		APPS	E0, 2F	E0, F0, 2F		KP .	71	F0, 71
S	1B	F0, 1B		ENTER	5A	F0, 5A		KP 0	70	F0, 70
T	2C	F0, 2C		ESC	76	F0, 76		KP 1	69	F0, 69
U	3C	F0, 3C		F1	05	F0, 05		KP 2	72	F0, 72
V	2A	F0, 2A		F2	06	F0, 06		KP 3	7A	F0, 7A
W	1D	F0, 1D		F3	04	F0, 04		KP 4	6B	F0, 6B
X	22	F0, 22		F4	0C	F0, 0C		KP 5	73	F0, 73
Y	35	F0, 35		F5	03	F0, 03		KP 6	74	F0, 74
Z	1A	F0, 1A		F6	0B	F0, 0B		KP 7	6C	F0, 6C
0	45	F0, 45		F7	83	F0, 83		KP 8	75	F0, 75
1	16	F0, 16		F8	0A	F0, 0A		KP 9	7D	F0, 7D
2	1E	F0, 1E		F9	01	F0, 01]	5B	F0, 5B
3	26	F0, 26		F10	09	F0, 09		;	4C	F0, 4C
4	25	F0, 25		F11	78	F0, 78		'	52	F0, 52
5	2E	F0, 2E		F12	07	F0, 07		,	41	F0, 41
6	36	F0, 36		PRNT SCRN	E0, 12, E0, 7C	E0, F0です 。7C, E0, F0, 12		.	49	F0, 49
7	3D	F0, 3D		スクロ ール	7E	F0, 7E		/	4A	F0, 4A
8	3E	F0, 3E		PAUSE	E1, 14, //, E1, F0, 14 です。 F0, 77	-NONE				

ACPIスキャンコード

キー	コードを作る	ブレイクコード
パワー	E0, 37	E0、F0、37
睡眠	E0、3F	E0、F0、3F
ウェイク	E0, 5E	E0、F0、5E

Windowsマルチメディアスキャンコード

キー	コードを作る	ブレイクコード
次のトラック	E0, 4D	E0、F0、4D
前のトラック	E0, 15	E0、F0、15
ストップ	E0, 3B	E0、F0、3B
再生/一時停止	E0, 34	E0, F0, 34
ミュート	E0, 23	E0、F0、23

ボリュームアップ	E0, 32	E0、F0、32
ボリュームダウン	E0, 21	E0、F0、21
メディア選択	E0, 50	E0、F0、50
電子メール	E0, 48	E0、F0、48
計算機	E0, 2B	E0、F0、2B
マイコンピュータ	E0, 40	E0、F0、40
WWW検索	E0, 10	E0、F0、10
WWWホーム	E0, 3A	E0、F0、3A
WWW Back	E0, 38	E0、F0、38
WWWフォワード	E0, 30	E0、F0、30
WWW停止	E0, 28	E0、F0、28
WWW リフレッシュ	E0, 20	E0、F0、20
WWWお気に入り	E0, 18	E0、F0、18

ATマザーボード用PS/2スキャンコードセット

KEY	MAKE	BREAK	-----	KEY	MAKE	BREAK	-----	KEY	MAKE	BREAK
A	1C	F0, 1C		9	46	F0, 46		[54	F0, 54
B	32	F0, 32		`	0E	F0, 0E		INSERT	67	F0, 67
C	21	F0, 21		-	4E	F0, 4E		HOME	6E	F0, 6E
D	23	F0, 23		=	55	F0, 55		PG UP	6F	F0, 6F
E	24	F0, 24		¥	5C	F0, 5C		DELETE	64	F0, 64
F	2B	F0, 2B		BKSP	66	F0, 66		END	65	F0, 65
G	34	F0, 34		スペース	29	F0, 29		PG DN	6D	F0, 6D
H	33	F0, 33		TAB	0D	F0, 0D		U ARROW	63	F0, 63
I	43	F0, 48		CAPS	14	F0, 14		L ARROW	61	F0, 61
J	3B	F0, 3B		L SHFT	12	F0, 12		D ARROW	60	F0, 60
K	42	F0, 42		L CTRL	11	F0, 11		R ARROW	6A	F0, 6A
L	4B	F0, 4B		L WIN	8B	F0, 8B		NUM	76	F0, 76
M	3A	F0, 3A		L ALT	19	F0, 19		KP /	4A	F0, 4A
N	31	F0, 31		R SHFT	59	F0, 59		KP *	7E	F0, 7E
O	44	F0, 44		R CTRL	58	F0, 58		KP-	4E	F0, 4E
P	4D	F0, 4D		R WIN	8C	F0, 8C		KP +	7C	F0, 7C
Q	15	F0, 15		R ALT	39	F0, 39		KP EN	79	F0, 79
R	2D	F0, 2D		APPS	8D	F0, 8D		KP .	71	F0, 71
S	1B	F0, 1B		ENTER	5A	F0, 5A		KP 0	70	F0, 70
T	2C	F0, 2C		ESC	08	F0, 08		KP 1	69	F0, 69
U	3C	F0, 3C		F1	07	F0, 07		KP 2	72	F0, 72
V	2A	F0, 2A		F2	0F	F0, 0F		KP 3	7A	F0, 7A
W	1D	F0, 1D		F3	17	F0, 17		KP 4	6B	F0, 6B
X	22	F0, 22		F4	1F	F0, 1F		KP 5	73	F0, 73
Y	35	F0, 35		F5	27	F0, 27		KP 6	74	F0, 74
Z	1A	F0, 1A		F6	2F	F0, 2F		KP 7	6C	F0, 6C
0	45	F0, 45		F7	37	F0, 37		KP 8	75	F0, 75
1	16	F0, 16		F8	3F	F0, 3F		KP 9	7D	F0, 7D
2	1E	F0, 1E		F9	47	F0, 47]	5B	F0, 5B

3	26	F0, 26		F10	4F	F0, 4F		;	4C	F0, 4C
4	25	F0, 25		F11	56	F0, 56		'	52	F0, 52
5	2E	F0, 2E		F12	5E	F0, 5E		,	41	F0, 41
6	36	F0, 36		PRNT SCRN	57	F0, 57		.	49	F0, 49
	3D	F0, 3D		スクロ ール	5F	F0, 5F		/	4A	F0, 4A
8	3E	F0, 3E		PAUSE	62	F0, 62				

結論

次の機会まで。

～Mike () です。

BrokenThorn Entertainment 社。現在、 EvolutionEngine とMicroOS OSを開発中。質問やコメントはお気軽にお問
い合わせください。