Linux ブートアップ ソースコードの解析

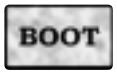
第6回 カーネルのセットアップを調べる —— setup.Sの解読(その2)

芳之内 弘

前回は、setup.Sのうち、PS/2マウスをチェクするところまでを調べました。今回は、これ以降のコードの働きを調べます。setup.Sの最後は、プロテクトモードに切り替えてhead.Sにジャンプします。setup.Sの概要やx86 CPUの32bitコードについては、前回説明しました。

今回は、プロテクトモードへの切り替えについて少し詳しく説明します。x86 CPUのプロテクトモードは、初めての人には少し難しいかもしれませんが、インテルは、8086からの上位互換性を保持するために、このようなCPUを考え出したのです。複雑怪奇なx86 CPUですが、20年近くも上位互換性を保持できていることは素晴らしいことです。

setup.SでコールされるBIOSのうち、video.Sで使われるもの以外については前回説明しましたので、そちらを参考にしてください。setup.Sでコールされるvideo.Sについては、次回以降で説明します。



GDTとGDTR

リアルモードでは、セグメントレジスタの値を0x10倍してオフセットアドレスに加えた値がアドレスです。しかしプロテクトモードでは、アドレスの計算方法が異なります。プロテクトモードでは、セグメントレジスタは、「セグメント記述子」を選択するセレクタになります。このセグメント記述子の集合が「ディスクリプタテーブル」です。このうち、すべてのタスクに共通なものが「GDT(Global Descripter Table)」です。

このセグメント記述子で示されるベースアドレスとオフセットを足したものが実アドレスとなります。このセグメント記述子は8bytes(すなわち64bit)で構成され、セグメントのリミット、ベースアドレス、属性を示します(表1)。ここでは、セグメン記述子の8bytesの並びを「B0、B1、B2、B3、B4、B5、B6、B7」として説明します。

これらは、表1にあるように、かなり複雑です。これは、80286の6bytesの記述子との互換性から、このように複雑になっているのです。80386から追加されたB6とB7の2bytesを使って拡張します。

このセグメント記述子は、必要な数だけ複数個作成されますので、セグメントセレクタは、GDTの中から、この記述子の先頭アドレスをポイント(指し示す)して選択します。従って、セグメントセレクタの値は常に8bytes刻みになります。CPUのGDTR(Global Descripter Table Register)には、このGDTの先頭アドレスと、GDTの長さをセットします。ソースコードの808行目から、実際のGDTが始まっていますので、これを用いて説明します(表2)。

リミット値は、セグメントのリミット値から"1"だけマイナスした値をセットします。セレクタが0x0018のリミットは、B6の下位ニブル、B1、B0から0xFFFFFとなります。このときのリミット値の計算は次のようになります。

```
0xFFFFF+1 = 0x100000(これはページ数になります)
0x100000 × 0x1000 = 0x100000000 = 4Gbytes
```

すなわち、セグメントのリミットは4Gbytesになります。

表1 セグメント記述子の構造

ビット数	意味	バイトの位置
20bit	セグメントのリミット値	B6の下位4bit、B1、B0
32bit	セグメントのベースアドレス	B7、B4、B3、B2
12bit	セグメントの属性	B6の上位4bit、B5

表2

セレクタ値	データ		!意味
0x0000	.word	0,0,0,0	!ダミーの記述子(使えない)
0x0008	.word	0,0,0,0	!今回は使わない2番目の記述子
0x0010	.word	0xFFFF	!リミット=0xFFFFF(0x100000*0x1000 = 4Gbytes)
	.word	0x0000	!ベースアドレス(0x0000)
	.word	0x9A00	!属性=0x9AC、ベースアドレス(0x00)
	.word	0x00CF	!ベースアドレス(00)、属性(0xC)、リミット(0xF)
			!リミット=B6の下位4bit、B1、B0=0xFFFFF
			!ベースアドレス=B7、B4、B3、B2=0x0000 0000
			!属性=B6の上位4bit(C)、B5=0xC、0x9A
			!(code read/exec、granularity=4096、386)
0x0018	.word	0xFFFF	!リミット=0xFFFFF(0x100000*0x1000 = 4Gbit)
	.word	0x0000	!ベースアドレス=0x00000000
	.word	0x9200	!属性=0x92、ベースアドレス(0x00)
	.word	0x00CF	!ベースアドレス(00)、属性(0xC)、リミット(0xF)

1ページは、この場合0x1000bytesです。

属性のビット構成の意味については、今回は説明を省略しますが、B6のbit 7(Gビット)が"1"のときは、リミット値は0x1000bytes(4Kbytes)単位になります。セレクタが0x0018のテーブルのB6は0xCFなので、bit 7は"1"で、1ページは0x1000B(4Kbytes)になります。

このGDTでは、CPUのコードセレクタ(ECS)が0x0010のときは、3番目の記述子が選択されて、ベースアドレスが"0"で長さ4Gbytesのコードセグメントが割り当てられます。データセレクタ(EDS)が0x0018だと、ベースアドレスが"0"で、長さ4Gbytesのデータセグメントが割り当てられます。

ディスクリプタテーブルには、このほかにタスク1つにつき1つずつ存在する「LDT」と、割り込みの設定に使う「IDT」があります。その構造はいずれもまったく同じです。さらに詳しく知りたい方は、記事末の参考文献(RESOURCE[1])を参照してください。

BOOT IDTとIDTR

リアルモードの割り込みルーチンの先頭アドレスは、ベクターテーブルにあります。プロテクトモードではIDTを用います。IDTには、最大で256個の割り込みゲートまたはトラップゲートが登録されます。CPUのIDTR(Interrupt Descripter Table Register)にこのIDTの先頭番地とその長さを格納します。

BOOT A20以上のアドレスバス

x86 CPUのリアルモードでは、アドレスバスはA0～A19を用いて0x00000～0xFFFFFの範囲の1Mbytesをアクセスします。従って、通常は1Mbytesを越えるアドレスは禁止されていて、A20以上は使用無効になっています。理由は、0xFFF0:0x1234(0x101134)のアドレスをアクセスしようとすると、アドレスバスは20bitしかないために、0x101134は0x0000:0x1134(0x01134)になってしまうからです。すなわち、16bitのセグメントと16bitのオフセットで0x1000000以上をアクセスしようとしたときにトラブルが起きるわけです。そこで、PC互換機では、A20以上のアドレスビットにどんな値が出力されても、常にゼロにする回路が挿入されています。逆に言えば、これを解除しないと32bitアドレスは使えないわけです。

このA20以上のアドレスバスを有効にする方法を次に説明します。PC互換機の0x60、0x64というI/Oアドレスは、キーボードを制御するポートですが、このポートを使ってA20の有効/無効を制御します。このポートはキーボードコントローラ8042のポートです。

①0x64ポートを入力したときは、ステータスレジスタの値を読み込みます。その値の意味は表3の通りです。

②制御コマンドを出力するときは2つの段階を踏みます。

- 1回目の出力
  ポート0x64にデータ0xD1を出力し、次回に0x60にコマンドを出力することを知らせます。
- 2回目の出力
  ポート0x60に制御コマンドを出力します。このコマンドの意味を表4に示します。

表3 ステータスレジスタの構成

ビット	役割	意味
bit 0	出力バッファア	0:空、1:フル
bit 1	入力バッファ	0:空、1:フル
bit 2	システムフラグ	0:電源ONでリセット、1:セルフテストOK
bit 3	コマンド/データ	0:ポートアドレス0x60、1:ポートアドレス0x64
bit 4	制御スイッチ	0:キーボード制御、1:制御なし
bit 5	タイムアウト送信	0:エラーなし、1:キーボード終了せず
bit 6	タイムアウト受信	0:エラーなし、1:キーボード終了せず
bit 7	パリティエラー	0:奇数(エラーなし)、1:偶数

③そして実際の制御は以下のようになります。
- 8042の送／受信バッファが空になるまで待つ
- 次に0x64ポートに0xD1を出力
- そして0x60ポートに「bit 1＝1」のデータを出力

実際のコードは、ソースコードの548行目～554行目で実行されています。

BOOT リアルモードからプロテクトモードへ

setup.Sの最後は、リアルモードからプロテクトモードに切り替えてlinux/arch/i386/boot/compressed/head.Sの先頭にジャンプします。ここでは、このプロテクトモードへの切り替えについて説明します。処理の流れは、以下のようになります。

①CPUのGDTRとIDTRに初期値を転送します。
②CPUの命令パイプラインの内容をクリアするため、JMP命令を1つ実行します。
③次に、CPUのコントロールレジスタ「CR0」のbit 0（ProtectEnableビット）を‘1’にセットします。

この直後に、CPUはプロテクトモードに移行します。プロテクトモードでは、アドレスの計算方法がリアルモードとは異なり、セグメントレジスタはセグメントセレクタになります。このプロテクトモードに移行直後の命令は、JMP命令でなければなりません。このプロテクトモードでの最初のJMP命令は次のようにして作られます。

```
db    0x66    ;オペランドサイズプリフィクス*1
db    0xEA    ;これはjmpiのコード
dw    0x1000  ;プログラムの最初の番地のオフセット
dw    0x10    ;コードセレクタ
```

これは「jmpi 0x1000:0x10」と同じです。このコードを実行すると、コードセレクタ0x10で示すGDTのベースアドレスに0x1000を加えたアドレスの命令から実行が始まります。

表4

ビット	意味
bit 0	システムリセット
bit 1	ゲートA20（A20以上のアドレスバスを制御します）
bit 2、bit 3	予約
bit 4	出力バッファ空（0：バッファ空、1：バッファフル）
bit 5	入力バッファ空（0：バッファ空、1：バッファフル）
bit 6	キーボードクロック出力
bit 7	キーボードデータ出力

BOOT 割り込みの制御

PC互換機では、割り込みの制御のために、8259あるいはその互換LSIを2個用いて、ハードウェア割り込みを制御しています。8259は、8入力の割り込みに対して、各対応するベクタ番号を発信することができるLSIです。これを2個用い、15入力のハードウェア割り込みに対する割り込み番号を、CPUに発信しています。この2つのLSIのうち、1つ目の8259-1の3番目の入力には、2個目の8259-2の割り込み要求出力が接続されています。この使い方は「マスター／スレーブ方式」として定義され、CPUは1つ目のマスタの8259-1よりベクタ番号を受け取ります。この8259に対する初期化のコマンドは、1byteごと4回に分けてICW1、ICW2、ICW3、ICW4として書き込まれます。このコマンドの意味は[2]の参考文献を参照してください。

BOOT ソースコードの解読（その2）

今回は、マウスの検出以降の「no_psmouse:」からです。

R E S O U R C E

[1] x86 CPUのプロテクトモードに関する参考文献
「初めて読む486」（蒲池輝尚著／アスキー出版）
「80486の使い方」（W.B.スルヤント著／オーム社）
「Inter Architecture Software Developer's Manual Volume 3」（Intel）

[2] PCの割り込み処理LSI（8259）に関するもの
三菱半導体データブック（三菱電機）

[3] PC互換機のアーキテキュチャに関する参考文献
「The Programmer's PC Sourcebook」
（Thom Hogan著、SE編集部訳、Microsoft Press）

[4] PC互換機のシステムBIOSに関する参考文献
「Phenix BIOS 4.0 Rivision 6 User's Manual」（Phoenix Technologies LTD）
「BIOS Enhanced Disk Driver Specification version.3.0 Rev.0.8」（Phoenix Technologines LTD）

*1 前回説明したものです。

リスト

■APM BIOSを検出してAPM BIOS情報をセットする

現在のDS=0x9000([0x9000:0x40]=[0x9000:64]はsetup.SのAPM BIOS情報の先頭アドレス)

```
389         mov     [64],#0                 ! APM BIOS情報のクリア
390
391         mov     ax,#0x05300             ! APM BIOS実装のチェック機能
392         xor     bx,bx                   ! BX=0
393         int     0x15
394         jc      done_apm_bios           ! CY=1はAPM BIOS未実装なのでdone_pm_bios(442)へ
395
396         cmp     bx,#0x0504d             ! BX="PM"?
397         jne     done_apm_bios           ! Noでdone_apm_bios(442)へ
398
399         and     cx,#0x02                ! bit 1=1?(32bitモードサポート?)
400         je      done_apm_bios           ! ZF=1(サポートなし)でdome_apm_bios(442)へ
401
402         mov     ax,#0x05304             ! APMインターフェイスの切断機能
403         xor     bx,bx                   ! BX=0
404         int     0x15                    ! 実行する
405
406         mov     ax,#0x05303             ! APM 32bitモードの接続機能
407         xor     bx,bx                   ! BX=0
408         int     0x15                    ! 実行
409         jc      no_32_apm_bios          ! CY=1(エラー)でno_32_apm_bios(439)へ
410
411         mov     [66],ax                 ! [0x9000:66]=32bit APM BIOSセグメント
412         mov     [68],ebx                ! [0x9000:68]=32bit APM BIOSオフセット
413         mov     [72],cx                 ! [0x9000:72]=16bit APM BIOSセグメント
414         mov     [74],dx                 ! [0x9000:74]=16bit APM BIOSオフセット
415         mov     [78],esi                ! [0x9000:78]=APM BIOSコードセグメント長
416         mov     [82],di                 ! [0x9000:82]=APM BIOSデータセグメント長
417                                         !
```

■APM BIOS 32bitモードの接続のチェックを再実行する(フラグの修正を行う)

```
421         mov     ax,#0x05300             ! APM BIOS 実装のチェック
422         xor     bx,bx                   ! BX=0
423         int     0x15
424         jc      apm_disconnect          ! CY=1でエラー。apm_disconnect(433)へ
425
426         cmp     bx,#0x0504d             ! BX="PM"?
427         jne     apm_disconnect          ! Noでapm_disconnect(433)へ
428
429         mov     [64],ax                 ! APM BIOSバージョンをセット
430         mov     [76],cx                 ! APM BIOSフラグをセット
431         jmp     done_apm_bios           ! done_apm_bios(442)へ
432


433 apm_disconnect:
434         mov     ax,#0x05304             ! 切断機能
435         xor     bx,bx                   ! BX=0
436         int     0x15                    ! 実行
437         jmp     done_apm_bios           ! dnoe_apm_bios(442)へ


439 no_32_apm_bios:
440         and     [76], #0xfffd           ! bit 2=0(32bitサポートのクリア)
441
442 done_apm_bios:
443 #endif
444
```

リストの続き

■プロテクトモードへ移行の準備を行う

```
447         seg cs
448         cmp     realmode_swtch,#0          ! realmode_swtchにエントリポイントがセットされているときは、そのアドレスのサブルーチンをコール。
449         jz      rmodeswtch_normal          ! realmode_swtch=0なのでrmodeswtch_normal(435)へ
450         seg cs                             ! CS=0x9020
451         callf   far * realmode_swtch       ! サブルーチンをコール
452         jmp     rmodeswtch_end             ! rmodeswtch_end(456)へ
453         rmodeswtch_normal:
454         push    cs                         ! CS=0x9020を待避
455         call    default_switch             ! 割り込みをすべて禁止する
456         rmodeswtch_end:
457
```

■32bitコードがスタートするアドレスをセット

```
460         seg cs                             ! CS=0x9020
461         mov     eax,code32_start           ! EAX=カーネルのスタートアドレス
462         seg cs
463         mov     code32,eax                 ! code32=EAX=(0x1000あるいは0x100000)
```

■カーネルのサイズがBIG_KERNELかどうかをチェック

BIG_KERNELの場合は、1Mbytes以上にロードされているので移動させない。そうでない場合は、全体を0x1000から始まるアドレスへ移動させる。

```
468         seg cs                                  ! CS=0x9020
469         test    byte ptr loadflags,#LOADED_HIGH ! 1Mbytes以上にロード？
470         jz      do_move0                        ! Noなので移動させる。do_move0(474)へ
471                                                 ! Yesなので移動させない
472         jmp     end_move                        ! end_move(499)へ
473
474 do_move0:
475         mov     ax,#0x100                  ! AX=0x100(移動先のセグメント値)
476         mov     bp,cs                      ! BP= #SETUPSEG=0x9020
477         sub     bp,#DELTA_INITSEG          ! BP= #INITSEG=0x9000
478         seg cs
479         mov     bx,start_sys_seg           ! BX=カーネルのスタートセグメント
480         cld                                ! SI、DIは増加する方向で繰り返す
481 do_move:
482         mov     es,ax                      ! ES=AX=0x100(移動先のセグメント)
483         inc     ah                         ! AX=AX+0x100(AH=AH+1と同じ)
484         mov     ds,bx                      ! DS=BX(移動元のセグメント＝カーネルのスターセグメント)
485         add     bx,#0x100                  ! BX=BX+0x100(0x1000bytesごとにコピーする)
486         sub     di,di                      ! SI=0
487         sub     si,si                      ! DI=0
488         mov     cx,#0x800                  ! CX=0x800(0x800ワード＝0x1000bytes)
489         rep                                ! CXの初期値の回数だけ次の命令を繰り返す
490         movsw                              ! [DS:SI] -> [ES:DI](1ワードコピーする)
                                               ! SI=SI+2、DI=DI+2
491         cmp     bx,bp                      ! BX < BP？
```

[start_sys_seg:0]の領域のプログラムを[0x100:0]から始まるアドレスへ4096bytesごとコピーする。転送元のセグメントが0x9000-0x100＝0x8F00まで繰り返す。すなわち、[start_sys_seg:0]-[0x8f00:0xfff]までのプログラムを[0x1000]から始まるアドレスへ移動する。

```
495         jb      do_move                    ! BX < BP(0x9000)の間はdo_move(481)から繰り返す(BX=8F00まで)
```

[0x1000:0000]-[0x8F00:0xFFF]のプログラムを[0x100:000]-[0x8000:0xFFF]に移動させた。すなわち、0x10000-0x8FFFFのプログラムを0x1000-0x80FFFに移動したことになる。

リストの続き

■全体的に、0xF000bytesだけ下位アドレスへ移動

```
499 end_move:
500         mov     ax,cs                       !AX= CS
```

LILOで起動すると、コードセグメントは「#SETUPSEG」と等しくなりますが、他のローダでは異なる場合があるかもしれません。例えば「jmpi INITSEG:0x0x200」を実行してセットアップコードに入った場合は、CSの値は「0x9000」となります。このルーチンは、このような場合を想定していると思われます。

＜500行目〜541行目は省略＞

```
541 end_move_self:
```

■CPUのGDTRとIDTRへデータのロード

```
543         lidt    idt_48                      ! 行番号823の値をIDTRにロードします。
544         lgdt    gdt_48                      ! 行番号827の値をGDTRロードします。
```

「gdt_48 .word 0x800」は、GDTの長さが2048bytes、すなわち256のGDTエントリを作れることを意味します。「.word 512+gdt,0x9」は、GDTのベースアドレス(0x90000＋512＋GDTのオフセット)を意味します。LIDTとLGDTも、共に6bytesのデータをロードします。最初の2bytesはテーブルのリミット長さで、次の4bytesはテーブルのリニヤなベースアドレスです。

■アドレスバスA20以上を有効にします

```
547
548         call    empty_8042                  ! バッファが空になるまで待ちます。入力バッファが空でないときは読み
                                                  出して空にします。
549         mov     al,#0xD1                    ! 制御コマンドを書き込むことを知らせます。
550         out     #0x64,al                    ! ポートに出力します。
551         call    empty_8042                  ! バッファが空になるまで待ちます。
552         mov     al,#0xDF                    ! A20=On、システムリセットをセット。
553         out     #0x60,al                    ! 出力ポートにALを出力
554         call    empty_8042                  ! バッファが空まで待ちます。
555
```

■アドレスバスA20以上が有効かをテストします

```
561
562 #defineTEST_ADDR 0x7c
563
564         push    ds                          ! DSを待避
565         xor     ax,ax                       ! AX=0
566         mov     ds,ax                       ! DS=0
567         dec     ax                          ! AX= -1 = 0xFFFF
568         mov     gs,ax                       ! GS=0xFFFF(High Memory Area)
569         mov     bx,[TEST_ADDR]              ! BX=[0x7C]、一時BXに待避させる
570 a20_wait:
571         inc     ax                          ! AX=AX+1(0、1、2、3、4……)
572         mov     [TEST_ADDR],ax              ! [0x0：0x7C]=AX
573         seg     gs                          ! GS=0xFFFF
574         cmp     ax,[TEST_ADDR+0x10]         ! AX=[0xFFFF:0x8C]？
```

アドレス0xFFFF:0x8Cは、0xFFFF0+0x8C=0x10007Cになる。そこで [GS:0x8C]に書き込んだ値と[0x0:0x7C]に書き込んだ値が同じだと、アドレスバスA20以上が有効ではないことになる。従って、A20以上のアドレスバスが無効だと無限ループになる。

```
575         je      a20_wait
576         mov     [TEST_ADDR],bx              ! [0x7C]に元の値BXを戻す
577         pop     ds                          ! DSを戻す
578
```

■コプロセッサがある場合はリセットする

```
580
```

リストの続き

```
581         xor     ax,ax                   ! AX=0
582         out     #0xf0,al                ! ポート0xF0にALのデータを出力する。
583         call    delay                   ! 少し時間を待つ
584         out     #0xf1,al                ! ポート0xF1にALのデータを出力する。
585         call    delay                   ! 時間待ち
586
```

■割り込み制御LSIの設定を行います

```
593                                         ! 割り込みベクタ「0x08〜0x17」を「0x20〜0x2F」へ変更して整理します。
594
595         mov     al,#0x11                ! 8259-1、8259-2のICW1を設定
                                            ! 0x11の意味：8259はシングルではないのでICW4が必要(エッジトリガ割り込み)
596         out     #0x20,al                ! 8259-1に書き込む
597         call    delay                   ! 時間待ち(LSIの反応が遅いので調整する)
598         out     #0xA0,al                ! 8259-2にICW1を書き込む
599         call    delay
600         mov     al,#0x20                ! 8259-1、8259-2のICW2を設定
                                            ! 0x20の意味：T*=0010 0000(すなわち、最初のベクタタイプ＝0x20)
601         out     #0x21,al                ! 8259-1に書き込む
602         call    delay
603         mov     al,#0x28                ! 8259-2にICW2を設定
                                            ! 0x28の意味：T*=0010 1000(すなわち、最初のベクタタイプ＝0x28)
604         out     #0xA1,al                ! 8259-2へ書き込む
605         call    delay
606         mov     al,#0x04                ! 8259-1のICW3を設定
                                            ! 0x04の意味：0000 0100(3番目の割り込み入力にスレーブが接続されている)
607         out     #0x21,al                ! 8259-1へ書き込む
608         call    delay
609         mov     al,#0x02                ! 8259-2のICW3を設定
                                            ! 0x02の意味：8259-2はスレーブ
610         out     #0xA1,al                ! 8259-2へ書き込む
611         call    delay
612         mov     al,#0x01                ! 8259-1、8259-2のICW4を設定
                                            ! 0x01の意味：8259-1は8086モード(ベクタ番号を送信する)
613         out     #0x21,al                ! 8259-1へ書き込む
614         call    delay
615         out     #0xA1,al                ! 8259-2へ書き込む
616         call    delay
617         mov     al,#0xFF                ! 8259-2の割り込み入力をすべてマスクする
618         out     #0xA1,al
619         call    delay
620         mov     al,#0xFB                ! 8259-1の割り込みをIRQ2を除いて
621         out     #0x21,al                ! すべてマスクする。
```

8259-1のIRQ2はマスクされていませんが、8259-2がすべてマスクされていますので結局、すべてのハードウェア割り込みはマスクされて受け付けされません。

■これよりプロテクトモードに移行します

最後のジャンプアドレスは0x100000か0x1000です。ただし0x1000はローダが変更しているかもしれません

```
637         mov     ax,#1                   ! CPUのCR0の(PE)bit=1にセット
638         lmsw    ax                      ! 実行(これでプロテクトモードに移行)
639         jmp     flush_instr             ! 命令パイプランをクリアする
640 flush_instr:
641         xor     bx,bx                   ! BX=0(これはhead.Sへ渡すフラグ)
```

次の段階のプログラムはlinux/arch/i386/boot/compressed/head.Sです。1Mbytes以上の領域にカーネルをロードしているときは、「jmpi 0x100000,__KERNEL_CS」という32bitオペランド命令でジャンプします。しかし、まだCSをリロードしていないので、デフォルトサイズはいまだ16bitです。オペランドサイズプリフィックスの0x66を0xEAの前に置くことにより、jmpi命令のオペランドは32bitに変更されます。すなわちジャンプアドレスは 48bitのファーポインタになります。

リストの続き

```
651         db      0x66,0xea                   ! 0x66=オペランドサイズフリフィックス
652 code32:dd       0x1000                      ! ここの値は、行番号463で書き換え済み
653         dw      __KERNEL_CS                 ! KERNEL_CS=0x10(セレクタ=0x10のベースアドレスは0。従って0x100000
                                                  か0x1000へジャンプする)
```

■プロテクトモードへ移行する前の割り込み禁止処理

```
668 default_switch:
669         cli                                 ! 割り込みの禁止
670         mov     al,#0x80                    ! NMIを禁止する
671         out     #0x70,al
672         retf
673
```

サブルーチン「bootsect_helper」は、bootsect.Sがカーネルを1Mbytes以上の領域にロードするときに使用するルーチンです。bootsect.Sは、セカンダリローダを持っていないので、サイズ制限の446bytes以下に納めるために、そのコードの一部をsetup.Sに埋め込んでいます。今回はこの部分の解読は省略します。

```
679 bootsect_helper:
```

＜省略＞

■キーボードバッファを空にします

```
758
759 empty_8042:
760         push     ecx                        ! ECXを待避
761         mov      ecx,#0xFFFFFF              ! ループカウンタ=0xFFFFFFにセット
762
763 empty_8042_loop:
764         dec      ecx                        ! ECX=ECX - 1
765         jz       empty_8042_end_loop        ! ECXが0になるまで繰り返す
766                                             ! 上は時間待ちルーチンです。
767         call    delay                       ! わずかな時間稼ぎ
768         in      al,#0x64                    ! AL=8042のステータス
769         test    al,#1                       ! bit 0=1?(出力バッファにデータあり?)
770         jz      no_output                   ! ZF=1で空。no_output(774)へ
771         call    delay                       ! わずかな時間待ち
772         in      al,#0x60                    ! 空でないのでバッファを読み込む
773         jmp     empty_8042_loop             ! 空になるまで繰り返す
774 no_output:
775         test    al,#2                       ! bit 1=1?(入力バッファにデータあり?)
776         jnz     empty_8042_loop             ! Yesは空でないので最初から繰り返す
777 empty_8042_end_loop:                        ! すべて空でここへ来る
778          pop     ecx                        ! ECXを戻す
779         ret
780
```

■リアルタイムクロックを読み出す

これはVideo.Sが使用する。ALにバイナリの秒データがセットされる。

```
784 gettime:
785         push    cx                          ! CX
786         mov     ah,#0x02                    ! リアルタイムクロックの時刻読み取り
787         int     0x1a                        ! 実行
788         mov     al,dh                       ! AL=DH(BCDの秒データ)
789         and     al,#0x0f                    ! 下位秒数のみ取り出す
790         mov     ah,dh                       ! AH=DH
791         mov     cl,#0x04                    ! CL=4(シフト回数のセット)
792         shr     ah,cl                       ! AHを4回右シフトする(AHは上位の秒数)
```

リストの続き

```
                                        ! AHにはBCDで表された秒データの10の位、ALには1の位がセットされている。
 793        aad                         ! 2桁のBCDデータを2進数に変換する。
                                        ! AL=バイナリの秒データ
 794        pop     cx                  ! CXを戻す
 795        ret
 798
```

■LSIへの書き込みへの反応の遅れを調整するための時間待ち

```
 800 delay:
 801        .word   0x00eb              ! jmp $+2(次のアドレスへジャンプ)
 802        ret
 806
```

■GDT、GDTR、およびIDTR

```
 808 gdt:
 809        .word   0,0,0,0             ! ダミーの記述子(使えない)
 810
 811        .word   0,0,0,0             ! 今回は使わない
 812
 813        .word   0xFFFF              ! リミット=0xFFFFF(0x100000*0x1000 = 4Gbytes)
 814        .word   0x0000              ! ベースアドレス=0x00000000
 815        .word   0x9A00              ! 属性=0x92C(data read/write, granularity=4096, 386)
 816        .word   0x00CF
 817
 818        .word   0xFFFF              ! リミット=0xFFFFF(0x100000*0x1000 = 4Gbytes)
 819        .word   0x0000              ! ベースアドレス=0x00000000
 820        .word   0x9200              ! 属性=0x92C(data read/write, granularity=4096, 386)
 821        .word   0x00CF
 822
 823 idt_48:                           ! IDTRへロードする値
 824        .word   0                   ! IDTのリミット=0
 825        .word   0,0                 ! IDTベースアドレス=0L
 826
 827 gdt_48:                           ! GDTRへロードする値
 828        .word   0x800               ! GDTのリミット=2048、256のセグメント記述子
 829        .word   512+gdt,0x9         ! GDTベースアドレス=0X9xxxx
```

＜途中省略＞

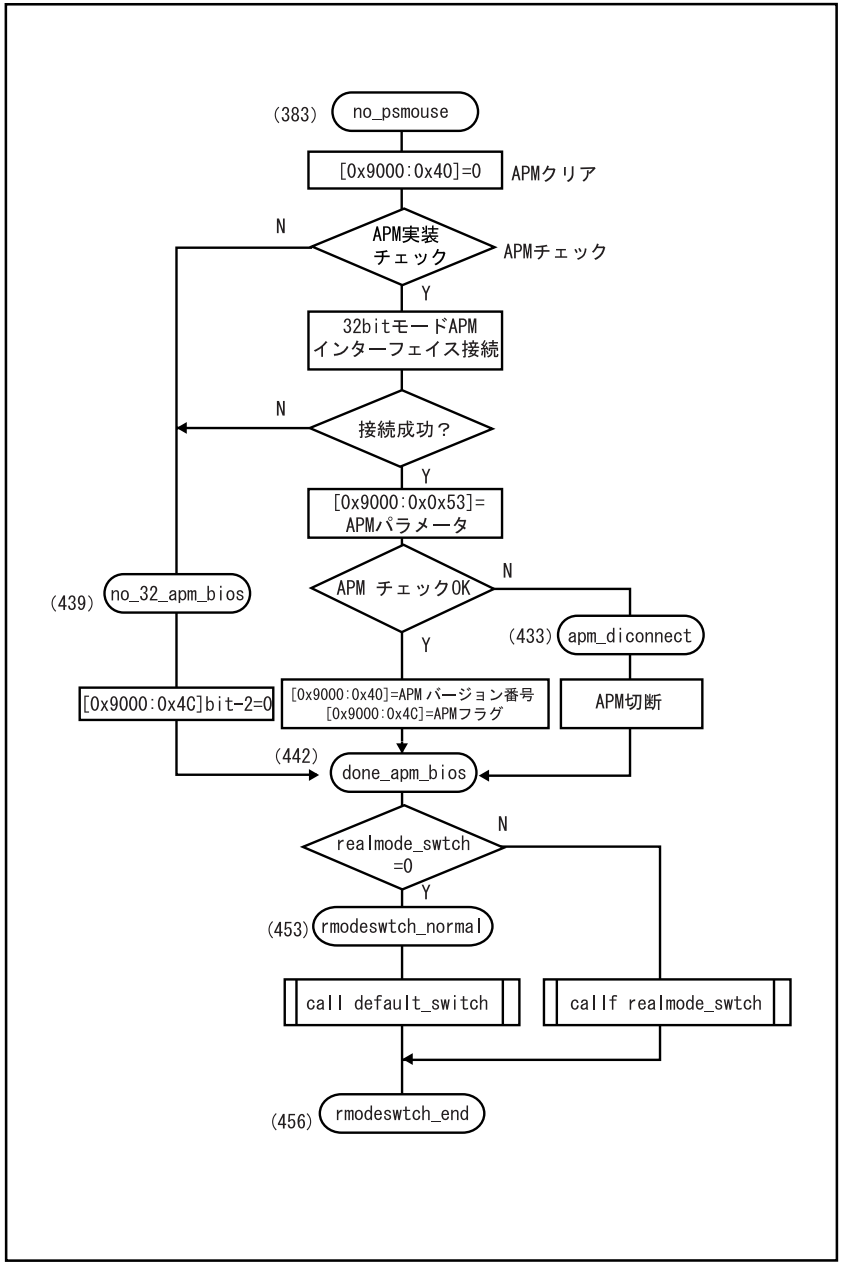
```
 834
 835 #include "video.S"                ! ビデオパラメータの設定ルーチンをここにインクルード。
 836                                    ! ここから後にコードが展開されます。
```

■setup.Sの読み込みをチェックするマジックナンバー

```
 841 setup_sig1:    .word   SIG1       ! 0xAA55
 842 setup_sig2:    .word   SIG2       ! 0x5A5A
 848
 849 modelist:
```

この後には、video.Sがビデオモードの設定で使われるテーブルを作ります。

フローチャート1



フローチャート2