アセンブリ言語

ハードウェア・インタフェース（2）

情報工学系
権藤克彦

BIOSコール

BIOS（1）

- BIOS=Basic Input/Output System　　firmware
- BIOSはソフトウェア．ファームウェアとも呼ぶ.
- 通常，BIOSは（書き換え可能な）ROM中にある.
  - ROMは不揮発性（電源を切っても内容が消えない）.
- BIOSの主な機能.
  - ブートして，I/Oデバイスを（一時的に）初期化する.
  - BIOS起動画面などで，ブート時の設定を可能にする.
  - BIOSコールを提供する.

UEFIに移行中

宣伝😄

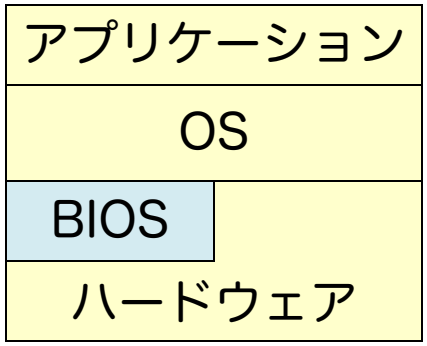
- ゼロからのOS自作入門
  - UEFI を使ってマイOSを作る本
  - 2021/3 発売
- 内田公太
  - 権藤研OB, サイボウズ・ラボ勤務

ゼロからのOS自作入門
内田公太

https://www.amazon.co.jp/dp/B08Z3MNR9J/

BIOS（2）

- BIOSはハードウェアとOSの間に位置する.
- OSは起動時にBIOSの機能を使う.
- OSはブート後はBIOSを介さずに，直接，ハードウェア（I/Oデバイス）とやりとりする.



BIOSコール

- ソフトウェア割り込みを使う手続きの一種.
  - I/Oデバイスとの入出力のためによく使う.
  - レジスタを引数として使う.
  - DOSファンクションコールとは別物.
- 呼び出し方は通常の関数呼び出しと大きく異なる.
  - 多くのBIOSコールは%axと%eflagsだけを変更する.
  - 一部のBIOSコールは返り値として他のレジスタも変更する.
- 通常，16ビット・リアルモードでのみ使用可能.
  - 通常，起動後のOSはBIOSコールを呼び出さない.
  - BIOSコールのコードは再入可能ではないから.

再入可能(reentrant)＝ある関数の実行中にその関数自身を再帰的，または非同期に呼び出しても問題が生じない関数の性質.

BIOSコールの例

テレタイプ端末(teletype)：
印刷式の端末．いわゆるtty（ダム）端末．

- ビデオサービス

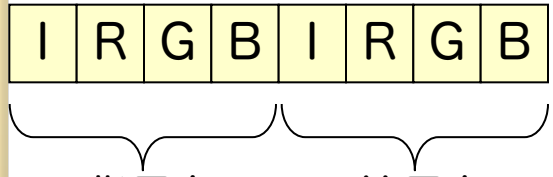
BIOSコールの引数

```
movb $0x0E, %ah
movb $'X,   %al
movb $0x02, %bl
int  $0x10
```

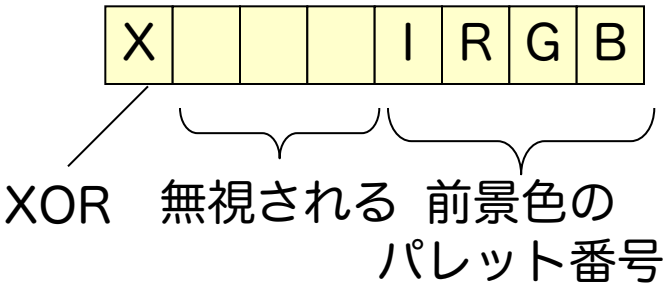
テレタイプ式文字書き込みコマンド．
書き込む文字を指定．
文字属性．

- 文字属性はモードごとに異なる．

モード0x03の属性バイト



モード0x12の属性バイト



I = 高輝度（intensity）

BIOSビデオサービス（1）

int $0x10：ビデオサービス	
%ah	説明
0x00	ビデオモードを設定
0x02	カーソル位置の変更
0x09	文字と属性を書き込む
0x0C	グラフィックの点を書く
0x0E	テレタイプ式文字書き込み

一部のみ

BIOSビデオサービス（2）

int $0x10：ビデオサービス	
%ah	0x00 （ビデオモードの設定）
%al	モードの値

%al	説明
0x03	80x25文字，16色（デフォルト）
0x11	640x480ドット，2色
0x12	640x480ドット，16色，80x30文字
0x72	640x480ドット，16色，80x25文字
0x73	80x25文字，16文字

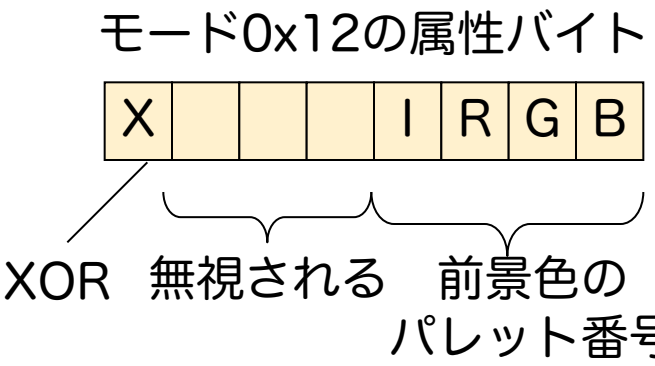
OADG BIOSリファレンス

%al	説明
0x00	40x25文字，白黒
0x01	40x25文字，カラー
0x02	80x25文字，白黒
0x03	80x25文字，カラー
0x04	320x200ドット，カラー
0x05	320x200ドット，白黒
0x06	640x200ドット，白黒
0x07	モノクロのみ

PhoenixBIOS 4.0 User's Manual

BIOSビデオサービス（3）

int $0x10：ビデオサービス	
%ah	0x0C （=1ドットの画素を書く）
%al	画素の色
%bh	0x00 （=画素を書くページ，0を設定）
%cx	何列目に書くか
%dx	何行目に書くか



BIOSビデオサービス（4）

int $0x10：ビデオサービス	
%ah	0x0E（=テレタイプ式文字書き込み）
%al	書き込む文字
%bl	前景色（グラフィックスモードのみ）

int $0x10：ビデオサービス	
%ah	0x02（=カーソル位置の変更）
%bh	0x00（=カーソル位置を変更するページ，0を設定）
%dl	文字単位で何列目か
%dh	文字単位で何行目か

int10-cursor

11-hard1.pptx の「16ビットのi386アセンブリコードからC関数を呼び出す」のスライドを参照

参照

（フロッピー）ディスケットサービス

int $0x13：ディスケットサービス	
%ah	0x02（=セクタを読み込む）
%dl	ドライブ番号（0〜3）
%dh	ヘッド番号（シリンダ番号）
%ch	トラック番号
%cl	セクタ番号
%al	読み込むセクタ数
%es:%bx	読み込み先のメモリアドレス

2x80x18x512=1.44MB

0〜1
0〜79
1〜18
} 3.5インチ, 1.44MB (MFM) の場合

返り値	
%eflagsのCF	1=エラー, 0=正常
%al	転送されたセクタ数
%ah	エラーコード

エラーコード（一部）	
0x00	エラー無し
0x01	不正なBIOSコマンド
0x03	書き込み保護エラー
0x06	メディア変更あり
0x09	DMA境界エラー
0x40	シークエラー
0x80	タイムアウト発生

ハードディスク入出力

- 基本はディスケットサービスと同じ.
- 異なる点：
  - %dl（ドライブ番号）のビット7を1に設定する.
  - %ah=0x02（セクタ読み込み）で，%clの0～5ビットにセクタ番号を，%clの6～7にはシリンダ番号10ビット中の上位2ビットをセットする.

詳細は，OADGテクニカル・リファレンス（DOS/V 技術解説編）を参照すること.
BIOSコールの説明あり.

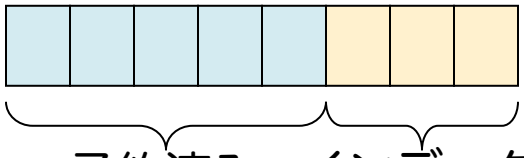
VGAディスプレイ

VGAのI/Oレジスタ

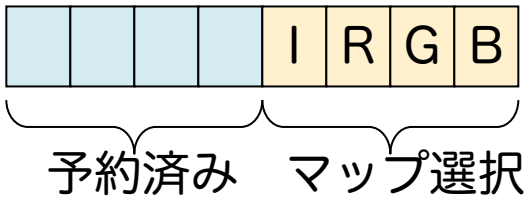
- たくさんあるが，BIOSコールで初期設定すれば，ほとんどは知らなくてOK.
- ここではシーケンサ・アドレス・レジスタとマップ・マスク・レジスタのみ扱う.

I/Oレジスタ	I/Oポート	インデックス
シーケンサ・アドレス・レジスタ	**0x03C4**	
リセット・レジスタ	0x03C5	0
クロッキングモード・レジスタ	0x03C5	1
マップ・マスク・レジスタ	**0x03C5**	**2**
文字マップ選択レジスタ	0x03C5	3
メモリモード・レジスタ	0x03C5	4

シーケンサ・アドレス・レジスタ



マップ・マスク・レジスタ



マップの選択例

マップは次ページで説明.

```
movw $0x03C4, %dx
movb $0x02,   %al
outb %al,     %dx
movw $0x03C5, %dx
movb $0x06,   %al
outb %al,     %dx
```

マップ・マスク・レジスタを選択.
（0x03C5に5つのレジスタがマップされてるので，この選択が必要．）

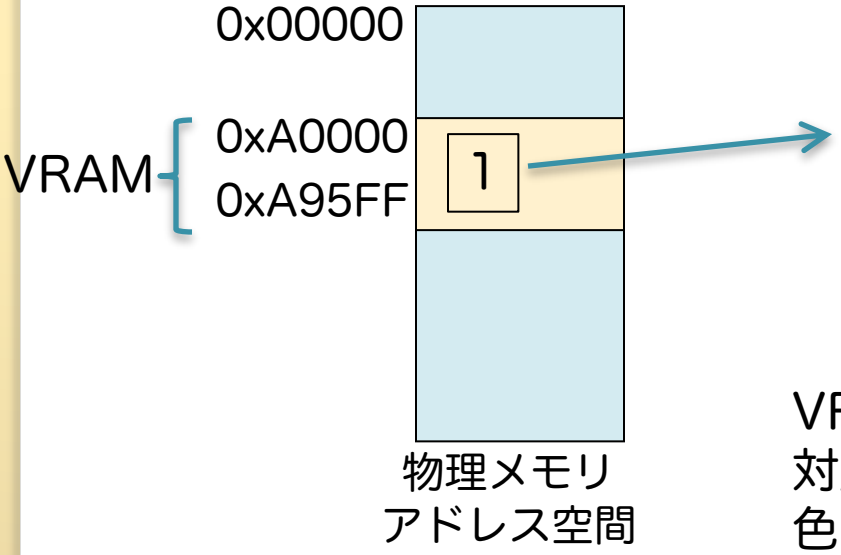
ここではマップ1（緑）とマップ2（赤）を選択している.

マップは同時に複数選択可能.

モード0x12のVRAMメモリマップ（1）

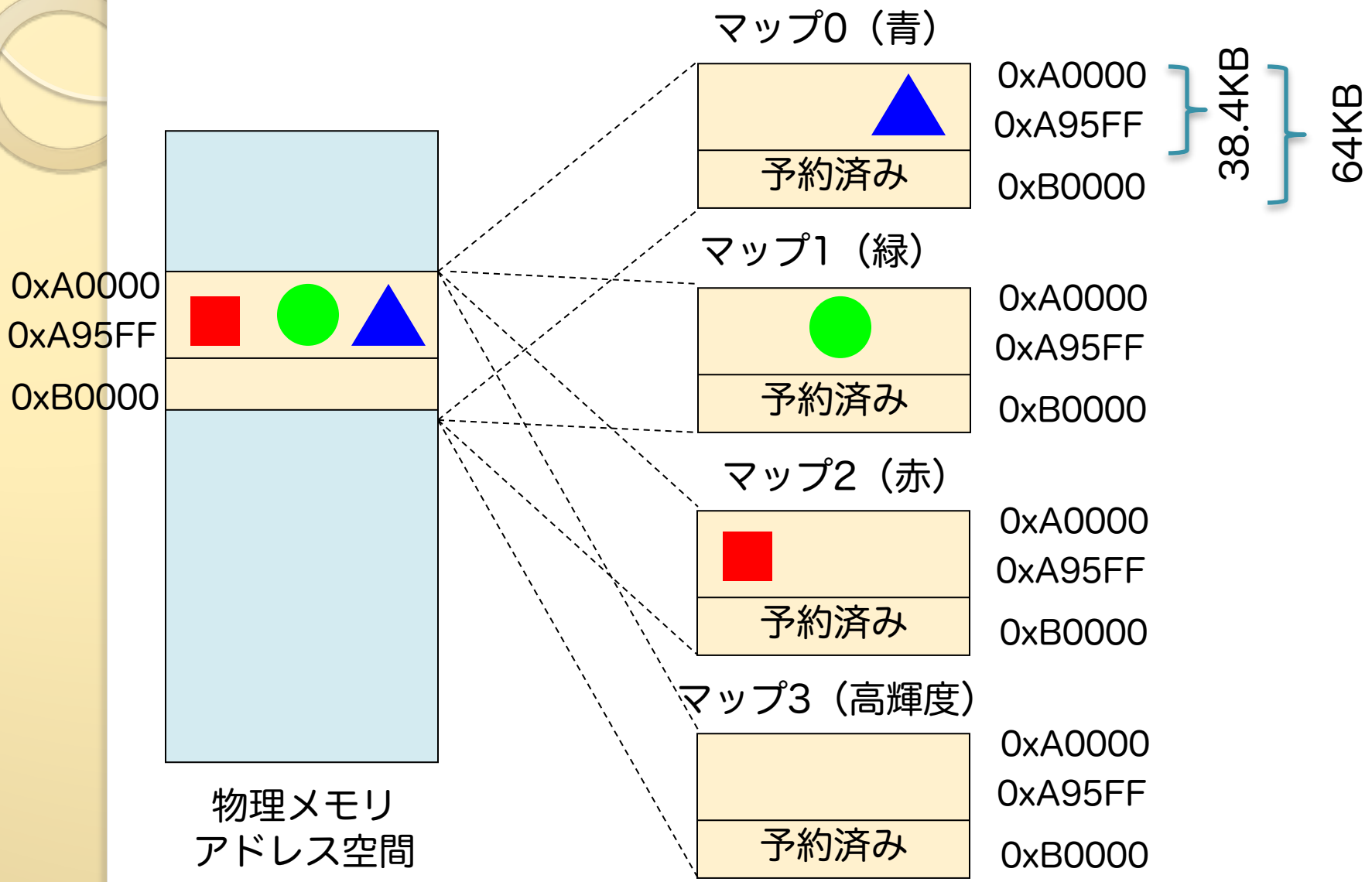
640*480/8=0x9600

- VRAMは0xA0000～0xA95FFの範囲にマップ.
- その範囲のビットを1にするとドットが光る.
- 1つのドットにマップ0～マップ3が対応.
  - このマップを選ぶことで，ドットの色を指定可能.



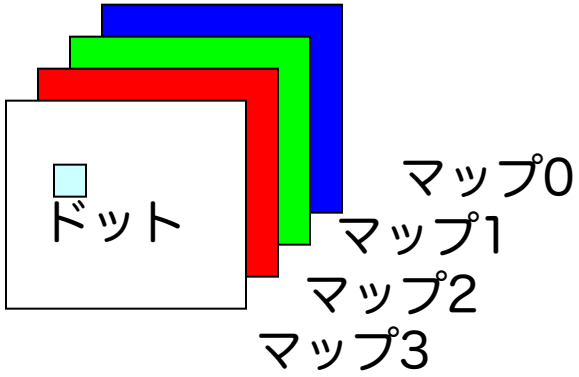
VRAM中のビットを1にすると
対応するディスプレイ上のドットが光る.
色はマップの選択で指定する.

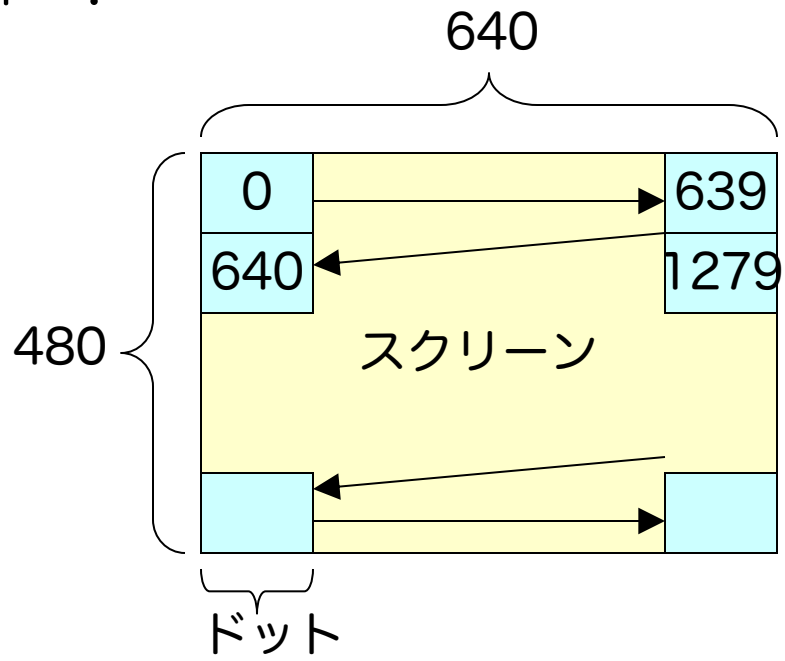
モード0x12のVRAMメモリマップ（2）



モード0x12のドット（画素，ピクセル）

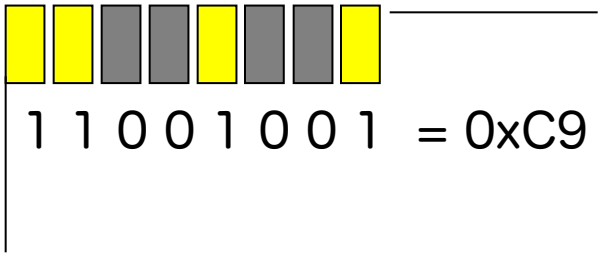
- 1ドットあたり4ビット（16色）
  - 4つのマップから1ビットずつで，1ドットを構成.
- ドットの順序.
  - 最初のドットはMSB.　（＝バイト内はビッグエンディアン）
  - 左から右へ，上から下へ.





モード0x12でドット描画例

```
.code16
.text
    ljmp $0x07c0, $1f
1:
    movb $0x00, %ah
    movb $0x12, %al
    int  $0x10
#
    movw $0x03C4, %dx
    movb $0x02, %al
    outb %al, %dx
    movw $0x03C5, %dx
    movb $0x0E, %al
    outb %al, %dx
#
    movw $0xA000, %ax
    movw %ax, %gs
    movb $0xC9, %gs:0x0000
#
2:  hlt; jmp 2b
.org 510
    .word 0xaa55
```



モード0x12を設定.

実行

マップ1とマップ2とマップ3を選択.
（赤＋緑＋高輝度＝明るい黄色）

VRAMの先頭バイトに11001001
を書き込み.

vram2

モード0x12で文字の描画例

非標準の

- ROMフォントの位置をBIOSコールで調べて使う.

int $0x10：ビデオサービス	
%ax	0x1130（フォント情報を得る）
%bh	ポインタ指示

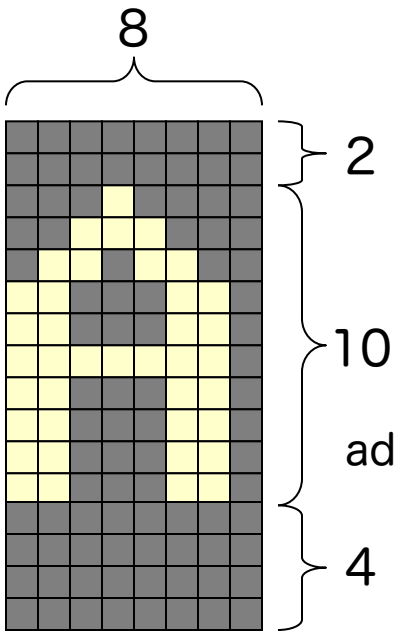
ポインタ指示	
6	ROM 8x16 フォント
7	ROM 9x16 代替フォント

返り値	
%es:%bp	フォントアドレス
%cx	文字あたりのバイト数
%dl	文字行数

640x480の場合，480/16=30.

```
movw $0x1130, %ax
movb $6,  %bh
int      $0x10
```

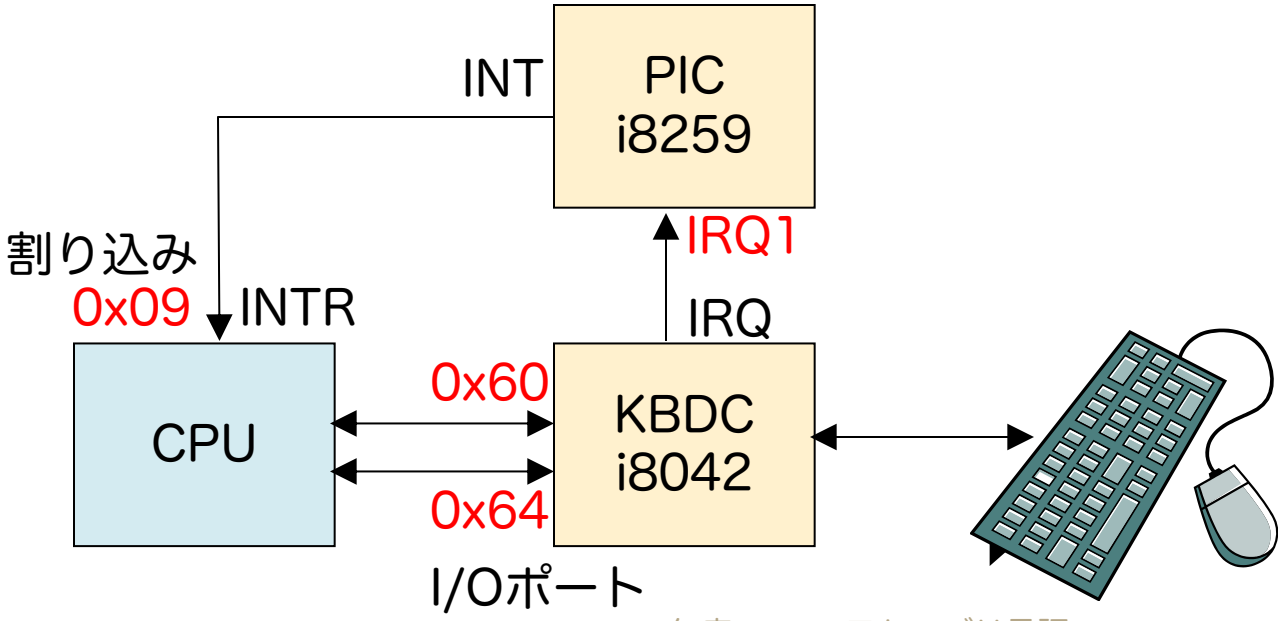
```
%es = 0xC000
%bp = 0x255A
%cx = 16
%dl  = 30
```



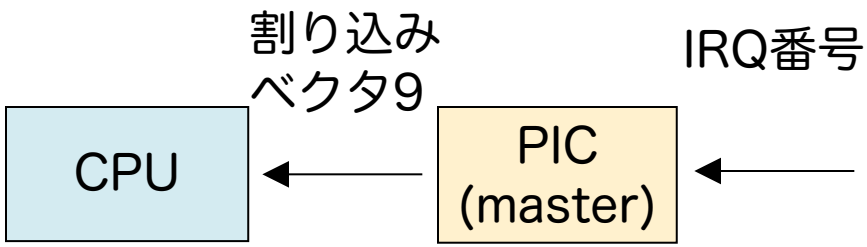
キーボード

キーボード，KBDC，PIC

- 2つのチップがキーボードを制御する．
  - i8042：キーボードコントローラ（KBDC）
    - ２つのI/Oポート（0x60，0x64）を持つ．
  - i8259：プログラム可能割り込みコントローラ（PIC）
    - 割り込みベクタ9でCPUに割り込み，キー入力をCPUに伝える．



PIC（1）



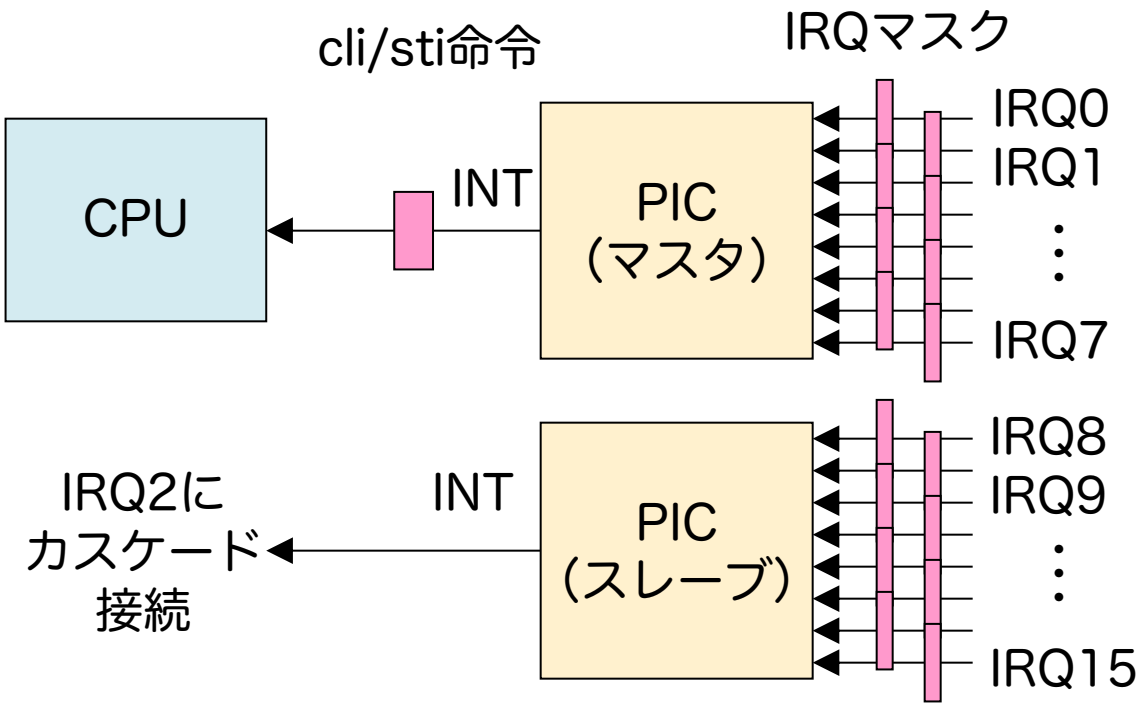
- PICはI/Oデバイスの割り込み要求をCPUに伝える.
  - その際，IRQ番号を割り込みベクタ番号に変換.

IRQ	INT	I/Oデバイス		IRQ	INT	I/Oデバイス
0	08	タイマー		8	70	リアルタイムクロック
1	09	キーボード		9	71	
2	0A	（PICスレーブに接続）		10	72	予約
3	0B	シリアルポート2		11	73	予約
4	0C	シリアルポート1		12	74	補助デバイス（マウス）
5	0D	パラレルポート		13	75	コプロセッサ
6	0E	フロッピーディスク		14	76	ハードディスク
7	0F	パラレルポート		15	77	予約

OADGテクニカル・リファレンス（ハードウェア）より抜粋.

PIC（2）

- PICのIRQマスクは割り込み毎に無効/有効を設定.
  - cli/sti命令は（NMI以外の）全割り込みを無効/有効にする.



複数のI/Oデバイスの割り込みをPICが集約してCPUに伝える.

IRQマスクはPICの一部.

PIC（3）

ここでは，IMRとOCW2だけ覚えればOK.
同じアドレスのI/Oレジスタの選択方法も
ここでは気にしなくてOK.

- PICのI/Oレジスタ
  - ICW1～ICW4：初期化コマンドワード.
  - OCW1～OCW3：オペレーションコマンドワード.
  - IRR：割り込み要求レジスタ（ペンディング中の割り込みを保持）.
  - ISR：割り込み中の割り込みを保持するレジスタ.
  - IMR：割り込みマスクレジスタ.

ICW1	w: 0x20 (0xA0)
OCW2	w: 0x20 (0xA0)
OCW3	w: 0x20 (0xA0)
IRR	r: 0x20 (0xA0)
ISR	r: 0x20 (0xA0)

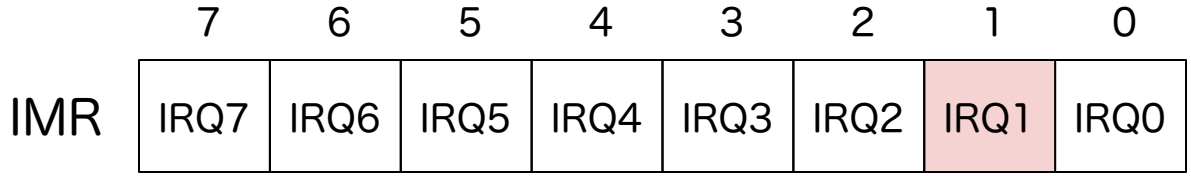
ICW2	w: 0x21 (0xA1)
ICW3	w: 0x21 (0xA1)
ICW4	w: 0x21 (0xA1)
OCW1 (IMR)	rw: 0x21 (0xA1)

OCW2 is selected if 0x20[4:3] = 0:0.
OCW3 is selected if 0x20[4:3] = 0:1.
ICW1 is selected if 0x20[4] = 1.

8259A PIC マスター（スレーブ）

PIC（4）

- IRQマスクの設定にはIMRを使う．




```
movw $0x21,  %dx
inb  %dx,    %al
orb  $0x02,  %al
outb %al,    %dx
```

PICマスターのIMRの
IRQ1を1（マスク）にする．

```
movw $0x21,  %dx
inb  %dx,    %al
andb $0xFD,  %al
outb %al,    %dx
```

PICマスターのIMRの
IRQ1を0（マスク解除）にする．

- 他のビットの元の値を保存するために，orやandが必要．
- マスクされた割り込みはIRRが保持する．マスク解除後にその割り込みが発生する．ただし2つ目以降は失われる．

PIC（5）

- 割り込み処理後，CPUはPICにEOIを送る必要あり．
  - EOI（割り込み終了） = end of interrupt
  - これを送ると，PICは次の割り込みをCPUに伝える．
  - CPUがPICスレーブに割り込まれた時は，PICマスターとPICスレーブの両方にEOIを送る．

```
movb  $0x20, %al
outb   %al, $0x20
```

PICマスターのOCW2（0x20）に
EOI（0x20）を送信．

0x20はnon-specific EOI.

キーボードのスキャンコード（1）

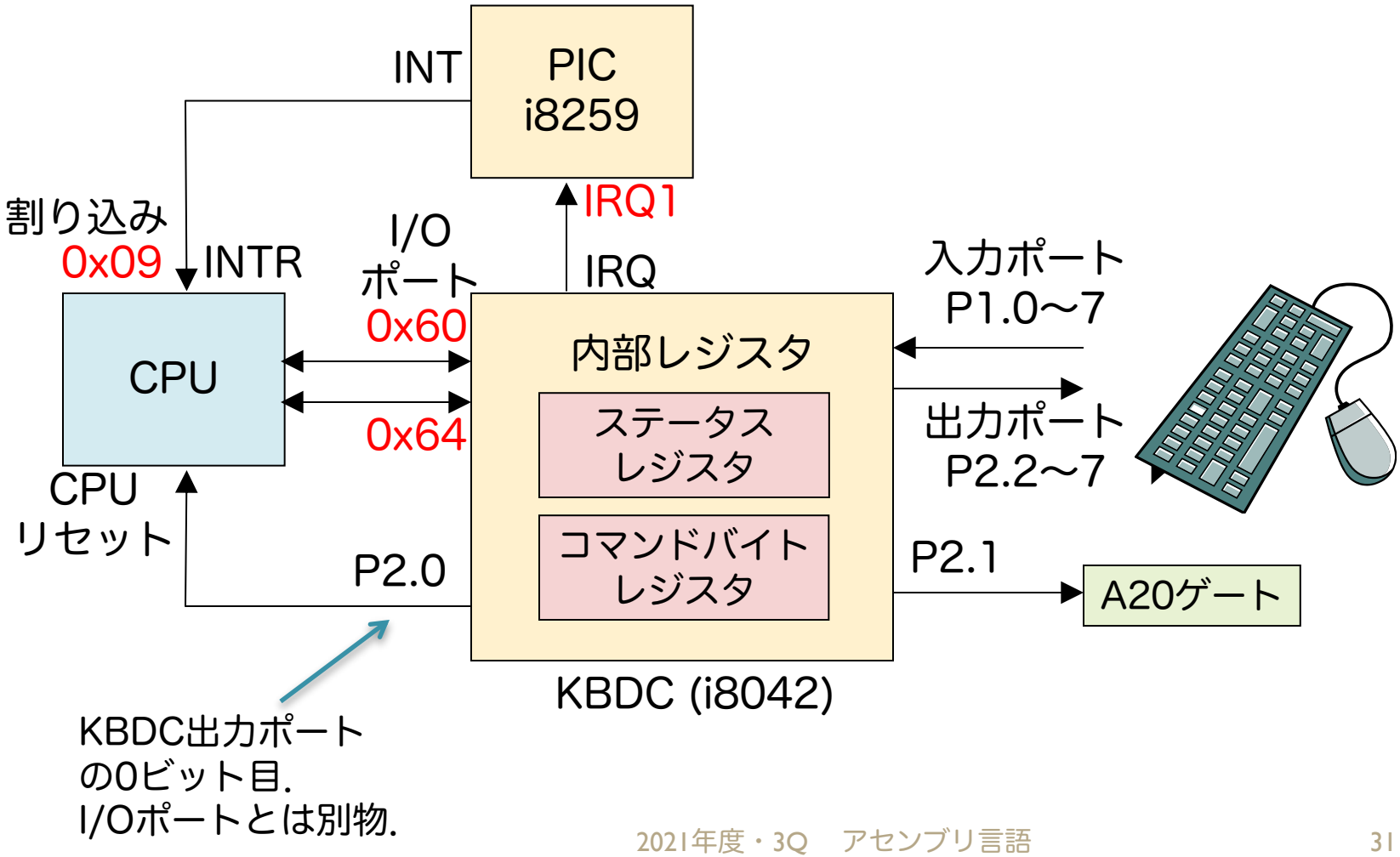
- スキャンコード（操作コード，scan code）
  - キー入力の際に，キーボードがCPUに送るコード（符号）．
  - ASCIIコードとは全くの別物．
  - 押す時（make-code）と離す時（break-code）でコードは別．
    - break-code = make-code + 128 （1バイト長のスキャンコードの場合）．
  - 多くのスキャンコードは1バイト長（拡張スキャンコードは別）．
  - 0x00=キーボードのバッファオーバーフロー．
  - 0xFF=キーエラー．
- キーボード毎にスキャンコードは異なる．
  - ここでは，101キーボードを仮定．
- 3つのスキャンコードセットがある．
  - 事実上のデフォルトはセット1．

キーボードのスキャンコード（2）

- セット1，0x01～0x46の make-code.

01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
esc	1!	2@	3#	4$	5%	6^	7&	8*	9(	0)	-_	=+	back space	tab	Q
11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20
W	E	R	T	Y	U	I	O	P	[{	]}	enter	ctrl	A	S	D
21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30
F	G	H	J	K	L	;:	' "	`~	left shift	¥|	Z	X	C	V	B
32	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	40
N	M	,<	.>	/?	right shift	print screen	alt	space	caps	F1	F2	F3	F4	F5	F6
41	42	43	44	45	46	・・・									
F7	F8	F9	F10	num	scrl	・・・									

KBDC（i8042）


INT
PIC
i8259
IRQ1
割り込み
0x09
INTR
I/O
ポート
0x60
0x64
IRQ
CPU
CPU
リセット
P2.0
内部レジスタ
ステータス
レジスタ
コマンドバイト
レジスタ
入力ポート
P1.0～7
出力ポート
P2.2～7
P2.1
A20ゲート
KBDC (i8042)
KBDC出力ポート
の0ビット目.
I/Oポートとは別物.

KBDCの2つのI/Oポート

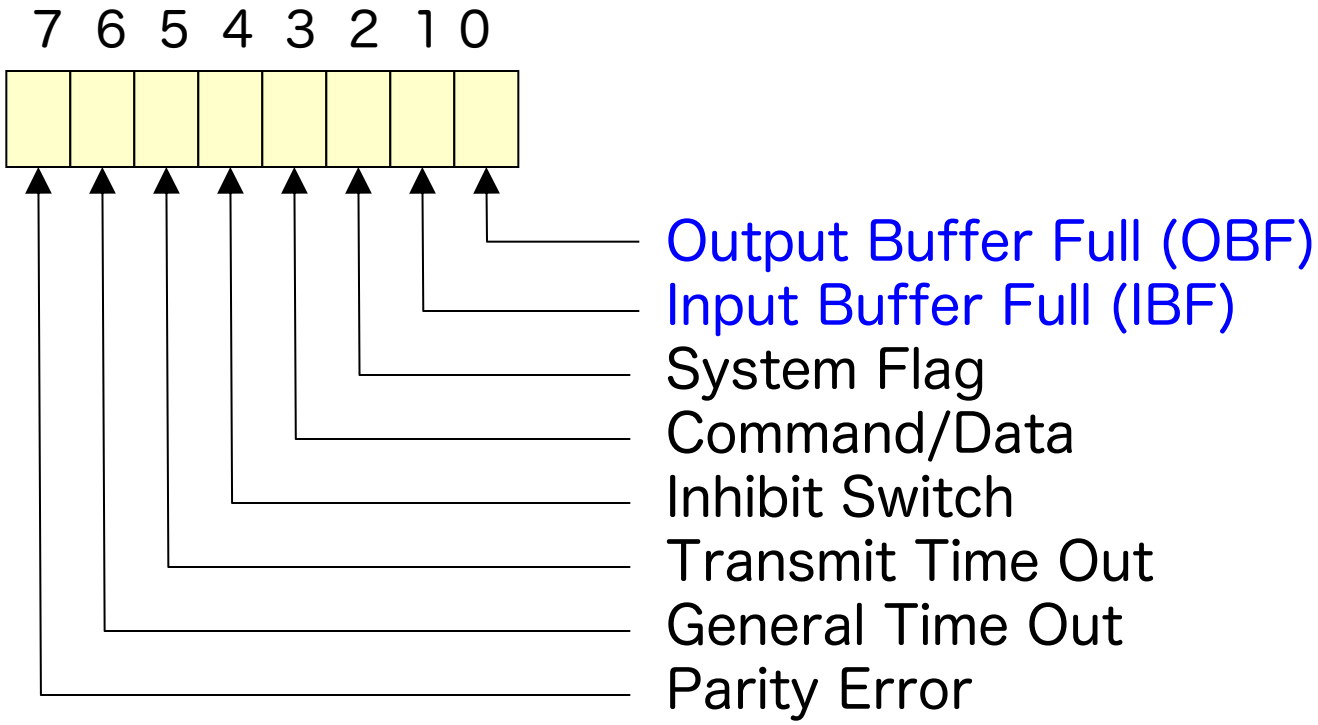
- 0x64と0x60の2つしかない.
  - アクセス方法や順番で意味が変わるので注意.

主な処理：

- ステータスレジスタの値を読む.
- キーボード（≠KBDC）へのコマンドを書き込む.
- KBDCへのコマンドを書き込む.
- KBDCのコマンドバイトレジスタを読み書きする.

ステータスレジスタ

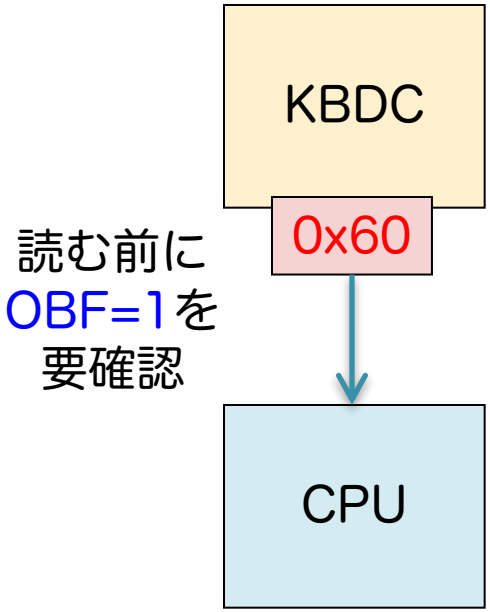
- I/Oポート0x64を読むとステータスレジスタ値を得る.
  - 0x64はいつでも読んで良い. （cf. 0x60の読み書き）
- ビット0のOBFとビット1のIBFが重要.



KBDCから見て出力なので，CPUから見ると入力になる．

OBF（出力バッファフル・フラグ）

- OBF=1は，I/Oポート0x60の出力バッファにデータが存在することを意味する．
  - そのデータをCPUが読むと，自動的にOBF=0になる．
- 0x60からの読み出し前にOBF=1の確認が必要．




```
    movw $0x64,  %dx
wait:
    inb     %dx,     %al
    test    $0x1,    %al
    jz wait
```

ステータス
レジスタを読む

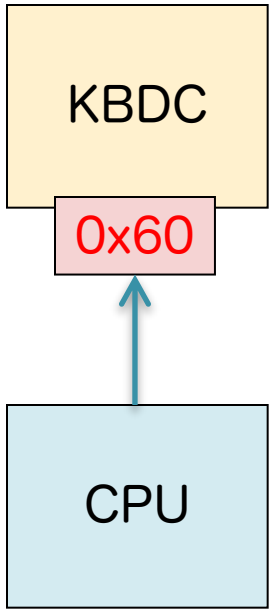
OBF=1になるまで待つコード．

busy-waitはCPUを浪費する点と，
タイムアウト処理がない点で悪いコード．

IBF（入力バッファフル・フラグ）

- IBF=1は，I/Oポート0x60の入力バッファにデータが存在することを意味する.
  - そのデータをKBDCが読むと，自動的にIBF=0になる.
- 0x60への書き込み前にIBF=0とOBF=0の確認が必要.

書く前にIBF=0とOBF=0を要確認




```
    movw $0x64,  %dx
wait:
    inb     %dx,     %al
    test    $0x3,    %al
    jnz wait
```

IBF=OBF=0になるまで待つコード.

busy-waitはCPUを浪費する点と，タイムアウト処理がない点で悪いコード.

キーボードへのコマンドを書き込む（1）

- 手順
  - IBF=OBF=0を確認後，0x60にコマンドを書く．
    - 引数があれば，さらにIBF=OBF=0を確認後，0x60に書く．
  - OBF=1を確認後，キーボードからの応答を0x60から読む．

0x64に書かずにいきなり0x60に書き込むと，
KBDCではなくキーボードへのコマンドと解釈される．

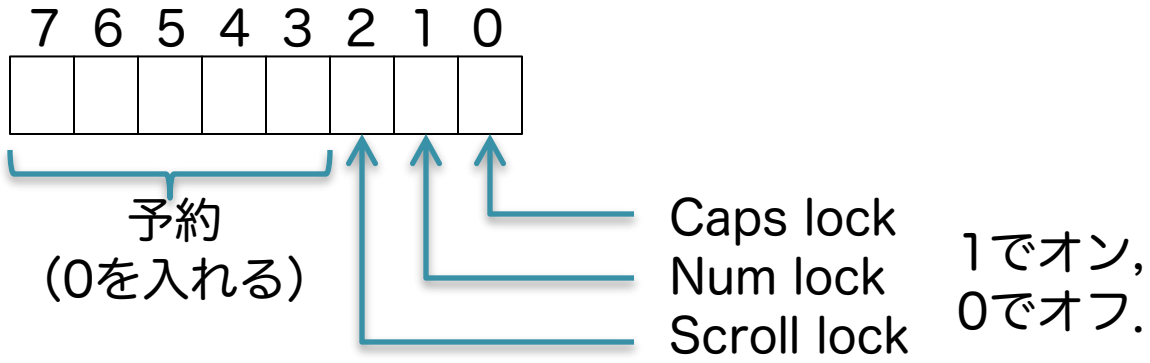
キーボードへの主なコマンド

コマンド	引数	説明
0xF4	なし	キーボードをイネーブル（有効化）．
0xFF	なし	キーボードをリセット．
0xED	1バイト	LEDの点滅．
0xF3	1バイト	オートリピート遅延や割合を設定．

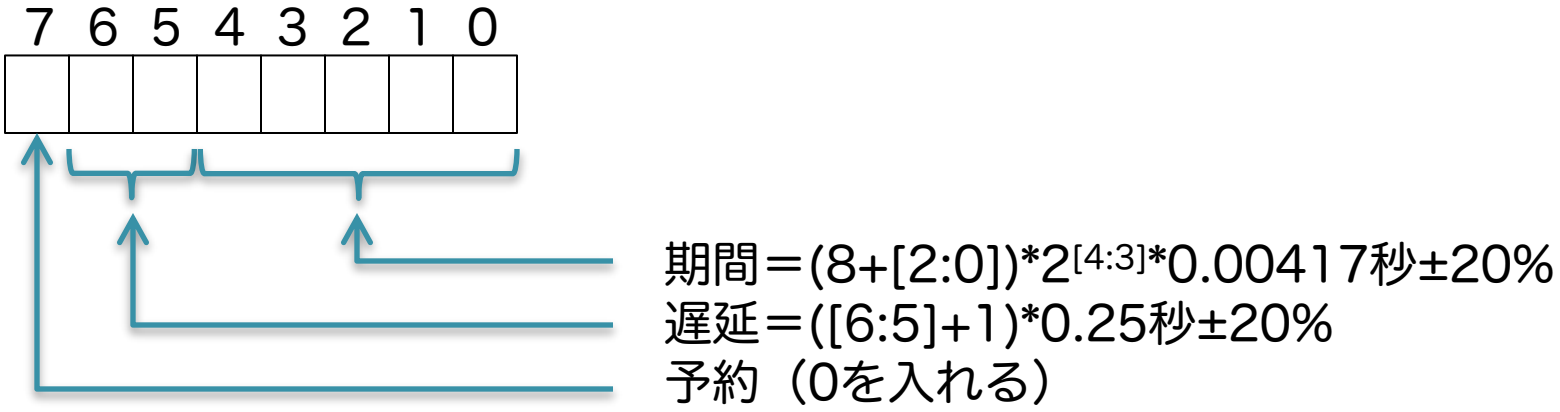
キーボードからの応答は0xFA(Ack)．
コマンドが0xEDや0xF3の場合，コマンドと引数に対して，それぞれAckを返す．

キーボードへのコマンドを書き込む（2）

コマンド0xED（LED明滅）の引数バイト



コマンド0xF3（オートリピート設定）の引数バイト



キーボードへのコマンドを書き込む（3）

例：NumロックLEDを点灯する.

```
.code16
.text
start:
    ljmp $0x07c0, $start2
start2:
    movw %cs,   %ax
    movb $0xED, %al
    movw $0x60, %dx
    outb %al,   %dx
    call wait_OBF_1
    movw  $0x60, %dx
    inb   %dx,   %al
    call wait_IBF_OBF_0
    movw $0x60, %dx
    movb $2,    %al
    outb %al,   %dx
    call wait_OBF_1
    movw  $0x60, %dx
    inb   %dx,   %al
    movb  $0x0E, %ah
    subb  $0x80, %al
    int   $0x10
exit: hlt; jmp exit
wait_IBF_OBF_0:
    movw $0x64, %dx
1:  inb  %dx,   %al
    test $3,    %al
    jnz  1b; ret
wait_OBF_1:
    movw $0x64, %dx
1:  inb  %dx,   %al
    test $1,    %al
    jz  1b; ret
.org 510
    .word 0xaa55
```

LED点滅コマンドを送信

ack (0xFA)を受け取る

引き数バイトを送信．numをオンに指定.

Ack (0xFA)を受け取る

Ack-0x80='zを画面に表示

IBF=OBF=0を待つ

OBF=1を待つ

実行

num-lock2

KBDCへのコマンドを書き込む（1）

- 手順
  - 0x64にKBDCへのコマンドを書く.
  - 2バイト目以降の引数や返り値は0x60で読み書きする.
    - OBF=1やIBF=OBF=0の確認をしてから.

0x64に書いてから，0x60に書き込むと，
キーボードではなくKBDCへのコマンド引数と解釈される

KBDCへの主なコマンド

コマンド	説明
0xAA	セルフテスト．成功すると0x55が返る.
0xAB	キーボードインタフェーステスト．成功すると0x00が返る.
0x20	コマンドバイトレジスタの値を読む.
0x60	コマンドバイトレジスタに値を書く.
0xD0	出力ポートの値を読む.
0xD1	出力ポートに値を書く（A20ゲートの値を設定できる）.

KBDCへのコマンドを書き込む（2）

セルフテスト(0xAA)のコマンドを実行.
0x55(='U)が返れば正常.

```
start:
    ljmp $0x07c0, $start2
start2:
    movw %cs,   %ax
    movb $0xAA, %al
    movw $0x64, %dx
    outb %al,   %dx
    call wait_OBF_1
    movw  $0x60, %dx
    inb   %dx,   %al
    movb  $0x0E, %ah
    int   $0x10
1:  hlt; jmp 1b
wait_OBF_1:
    movw $0x64, %dx
1:  inb  %dx,   %al
    test $1,    %al
    jz  1b; ret
.org 510
    .word 0xaa55
```

セルフテスト
コマンドを送信.

返り値を読む.
0x55なら成功

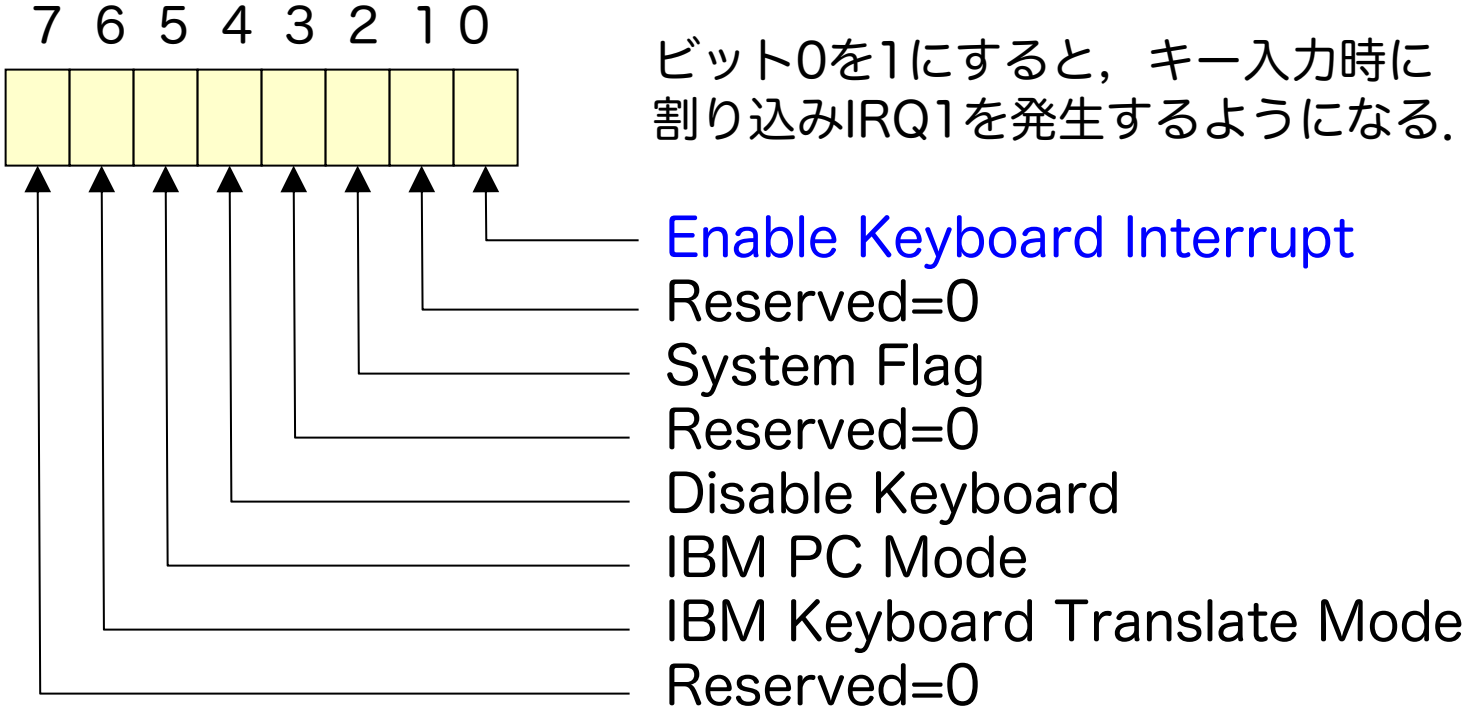
'U' = 0x55

実行

self-test

コマンドバイトレジスタ（1）

- KBDCへのコマンド0x20と0x60をI/Oポート0x64に書くと，コマンドバイトレジスタの読み書きを指示.
- 次にI/Oポート0x60から読み書きをしてコマンドバイトレジスタの値をやりとりする.



ビット0を1にすると，キー入力時に割り込みIRQ1を発生するようになる.

コマンドバイトレジスタの意味.

コマンドバイトレジスタ（2）

```
.code16
.text
start:
    ljmp $0x07c0, $start2
start2:
    movw %cs,   %ax
    movb $0x20, %al
    movw $0x64, %dx
    outb %al,   %dx
    call wait_OBF_1
    movw  $0x60, %dx
    inb   %dx,   %al
1:  hlt; jmp 1b
wait_OBF_1:
    movw $0x64, %dx
1:  inb  %dx,   %al
    test $1,    %al
    jz  1b; ret
.org 510
    .word 0xaa55
```

com-byte

コマンドバイトレジスタの
値を読む.

コマンドバイトレジスタの
読み込みを指示.

コマンドバイトレジスタの
値を%alに読む.

実行

bochsデバッガで%alの値を
調べると，0x61が入っていた.

コマンドバイトレジスタのビット0を0に設定して，
キーボード割り込みを無効にする．

コマンドバイトレジスタ（3）

実行

```
.code16
.text
start:
  ljmp $0x07c0, $start2
start2:
  movw %cs,   %ax
  call echo
  movb $0x20, %al
  movw $0x64, %dx
  outb %al,   %dx
  call wait_OBF_1
  movw  $0x60, %dx
  inb   %dx,   %al
  andb  $0xFE, %al
  movb  %al,   %cl
  movb $0x60, %al
  movw $0x64, %dx
  outb %al,   %dx
  call wait_IBF_OBF_0
```

ここではキー入力可能．

コマンドバイトレジスタの読み込みを指示．

コマンドバイトレジスタの値を読む．
ビット0だけを0に．

コマンドバイトレジスタの書き込みを指示．

```
  movb  %cl,   %al
  movw  $0x60, %dx
  outb  %al,   %dx
  call echo
1:  hlt; jmp 1b
wait_IBF_OBF_0:
  movw $0x64, %dx
1:  inb  %dx,   %al
  test $3,    %al
  jnz  1b; ret
wait_OBF_1:
  movw $0x64, %dx
1:  inb  %dx,   %al
  test $1,    %al
  jz  1b; ret
echo:
  movb $0x00, %ah
  int  $0x16
  movb $0x0E, %ah
  int  $0x10
  ret
.org 510
  .word 0xaa55
```

コマンドバイトレジスタに書き込む．

ここではキー入力不可．

キーボードから%alに1文字入力．

テレタイプ式文字書き込み．

com-byte2

キーボードからスキャンコードを受信（1）

```
.code16
.text
    ljmp $0x07c0, $start2
start2:
    movw %cs,     %ax
    movw %ax,     %ds
loop:
    call read_key
    jmp loop
read_key:
    call wait_OBF_1
    movw  $0x60, %dx
    inb   %dx,   %al
    test  $0x80, %al
    jnz   skip
    movzbw  %al,   %bx
    movb  keymap(%bx), %al
    movb  $0x0E, %ah
    int   $0x10
skip:
    ret
```

kbd-poll

```
wait_OBF_1:
    movw $0x64, %dx
1:  inb  %dx,   %al
    test $1,    %al
    jz  1b; ret
keymap:
    .byte 'X, '*, '1, '2, '3, '4
    .byte '5, '6, '7, '8, '9, '0
.org 510
    .word 0xaa55
```

0〜9までのキー入力を仮定.

busy-waitで待つ.
スキャンコードを
得る.

breakコード（キーを離す）は無視.
スキャンコードを
ASCIIコードに変換.

実行

何もコマンドを送っていない時に，
キー入力があると，OBF=1となり，
0x60からスキャンコードを読める.

ここでは起動時のPICの
設定をそのまま利用.

割り込みで（普通のやり方）

キーボードからスキャンコードを受信（2）

```
.code16
.text
  movw %cs,     %ax
  movw %ax,     %ds
  movw %ax,     %ss
  movl $0xFFF0, %esp
  movl $0xFFF0, %ebp
  movw $handler, %ax
  addw $0x7C00, %ax
  movw %ax,     4*9
  movw %cs,     4*9+2
  sti
1:  hlt; jmp 1b
handler:
  cli; pusha;
  call wait_OBF_1
  movw $0x60,   %dx
  inb %dx,      %al
  test $0x80,   %al
  jnz handler_exit
  movzbw %al,   %bx
  addw $0x7C00, %bx
```

9番の割り込み
ハンドラを設定.
割り込み許可.

割り込みを禁止して
レジスタを退避.

スキャンコードを
得る.

```
  movb keymap(%bx), %al
  movb $0x0E, %ah
  int  $0x10
handler_exit:
  movb $0x20, %al
  outb %al, $0x20
  popa; sti; iret
wait_OBF_1:
  movw $0x64,   %dx
1:  inb  %dx,     %al
  test $0x1,    %al
  jz 1b; ret
keymap:
  .byte 'X, '*, '1, '2, '3, '4
  .byte '5, '6, '7, '8, '9, '0
.org 510
  .word 0xaa55
```

PIC1にEOI
を送信.

kbd-intr

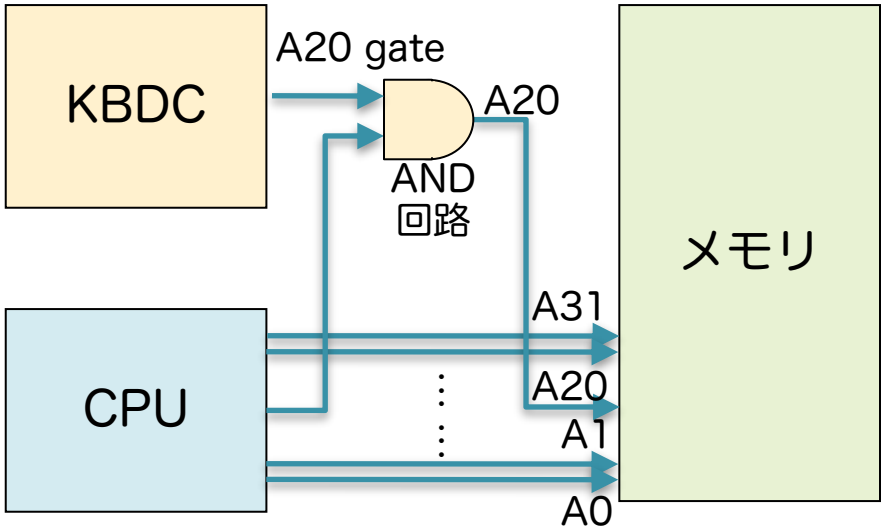
実行

何もコマンドを送っていない時に，
キー入力があると，割り込み番号9の
割り込みが発生して，
0x60からスキャンコードを読める.

8086の「1MBの壁」と
互換性を保つための仕組み.

参考：A20ゲート（1）

- A20＝i386の21番目のアドレスピン.
- A20ゲート＝A20を有効・無効に制御するピン.
  - なぜかKBCについている．起動後のデフォルト値はオフ(0).
  - A20ゲート=0だと，常にA20=0になる.
    - 1MB以上のメモリアクセスに支障.



参考：A20ゲート（2）

- OADGのマニュアルでは，デフォルトで，A20ゲートはオン（A20をマスクしない）とある．
- KBDCの出力ポートをコマンド0xD0と0xD1で読み書きして，A20ゲートをオン・オフする．
  - P2.1=1でオン，P2.1=0でオフ．
  - 他にも方法はある：BIOSコールやシステムポート(0x92)．

参考：A20ゲート（3）

実行

a20gate

```
.code16
.text
  ljmp $0x07c0, $start2
start2:
  call check_a20gate
  movw $0x64, %dx
  movb $0xD0, %al
  outb %al,   %dx
  call wait_OBF_1
  movw $0x60, %dx
  inb  %dx,   %al
  movb %al,   %bl
  movw $0x64, %dx
  movb $0xD1, %al
  outb %al,   %dx
  call wait_IBF_OBF_0
  movw $0x60, %dx
  movb %bl,   %al
# orb  $0x02, %al
  andb $0xFD, %al
  outb %al,   %dx
1:  hlt; jmp 1b
```

出力ポートの現在の値を%blに代入.

A20ゲート（P2.1）を変更して出力ポートに書く.

```
wait_IBF_OBF_0:
    movw $0x64, %dx
1:  inb  %dx,   %al
    test $3, %al; jnz  1b; ret
wait_OBF_1:
    movw $0x64, %dx
1:  inb  %dx,   %al
    test $1, %al; jz 1b; ret
check_a20gate:
    movw $0x0000, %ax
    movw %ax,     %ds
    movw $0xFFFF, %ax
    movw %ax,     %es
    movb $'C,     %al
    movw $0x1234, %ds:0x0000
    movw $0x5678, %es:0x0010
    movw %ds:0x0000, %bx
    cmpw $0x5678, %bx
    je   1f
    movb $'O,     %al
1:  movb $0x0E,   %ah
    int $0x10; ret
.org 510
    .word 0xaa55
```

0x000000に0x1234を書き，0x100000に0x5678を書き，値を比較してA20ゲートが有効か否かを調べる.

通常，ユーザ空間で in/out命令を実行できない理由.

参考：IOPL

- I/Oポートの特権レベル.
  - %eflagsのIOPLフィールド（２ビット）で設定.
  - I/Oポートは，数値的にCPL≦IOPLの時のみアクセス可能.
  - CPLは実行中のコードセグメントディスクリプタの特権レベル.
- IOPLフィールドは%popfd命令で設定する.
  - CPL=0の時のみ，IOPLフィールドを設定可能.
  - 特権命令ではないが，CPL≧1の時はIOPLを設定できない.