N（なんて） D（どうでもいい） S（作文だろうか）

N（名前）D（ですよ）S（さびたコイル）

1. はじめに

ここでは、NDS（任天堂 DS）上で動くプログラムの作り方を説明しています。実機で動作させるためには、別途マジコンが必要です。

2. 開発環境の導入

まずは開発環境の導入です。NDS の CPU、ARM で動くプログラムをコンパイルするために、gcc ベースのコンパイラを使います。

http://www.devkitpro.org/ で左メニューの Downloads から Windows Installer をダウンロードし、インストールします。無事インストールできたら、環境変数「DEVKITARM」「DEVKITPRO」と、Path に「c:¥devkitPro¥msys¥bin;c:¥devkitpro¥devkitarm¥bin;」が通っていることを確認（インストール場所デフォルト時）。もし Vista なら、さらに手動で「c:¥devkitPro¥msys¥1.0.11¥bin;c:¥devkitPro¥devkitARM¥libexec¥gcc¥arm-eabi¥4.1.1;c:¥devkitPro¥devkitARM¥arm-eabi¥bin;」を追加。これで devkitARM のインストールは完了です。

次にコンパイルするごとに実機に送っていたら面倒なのでパソコン上での実行用のエミュレータを用意します。自分は no$gba を使っています。http://nocash.emubase.de/あとは.nds ファイルを関連付けしておけばいいでしょう。

ここまで揃ったら、試しに c:¥devkitpro¥examples¥nds¥ の適当なフォルダに DOS で移動し、make とコマンドを打ってみましょう。そうするとコンパイルされ、.nds というファイルができるので、これを先ほどのエミュレータで開けば、おそらくは動作すると思います。

3. Makefile を作る

サンプルがコンパイルできたところで、いよいよ自分で作っていきます。まず作業用ディレクトリを決め、Makefile を作ります。そこに以下のように書きます。

```
NDSLIB_INCLUDE=$(DEVKITPRO)/libnds/include
NDSLIB_LIB=$(DEVKITPRO)/libnds/lib
all: XXXXX.nds.gba
```

XXXXX は出来上がるファイル名です。

```
arm9_main.o: arm9_main.cpp main.h

arm-eabi-gcc -MMD -MP -MF -main.d -g -Wall -O2 -mcpu=arm9tdmi -mtune=arm
9tdmi -fomit-frame-pointer -ffast-math -mthumb-interwork -I$(NDSLIB_INC
LUDE) -DARM9 -c arm9_main.cpp -o arm9_main.o
```

ここは arm9_main~と arm-eabi~の二行で書いてください。

```
arm9.elf: arm9_main.o

arm-eabi-gcc -g -mthumb-interwork -mno-fpu -specs=ds_arm9.specs arm9_ma
in.o -L$(NDSLIB_LIB)  -lnds9 -lm -o arm9.elf
```

ここも。

```
arm9.bin: arm9.elf
   arm-eabi-objcopy -O binary arm9.elf arm9.bin
Dpypy.nds: arm9.bin
   ndstool -c XXXXX.nds -9 arm9.bin
XXXXX.nds.gba: XXXXX.nds
   dsbuild XXXXX.nds -o XXXXX.nds.gba
clean:
   rm -f *.bin
   rm -f *.elf
   rm -f *.o
   rm -f *.nds
   rm -f *.gba
   rm -f *~
```

こんな感じです。NDS には ARM7 と ARM9 という 2 つの CPU があり、グラフィックなどは ARM9、音やタッチセンサーなどは ARM7 で処理しています。今はまだ ARM9 しか使わないので ARM9 だけを書きます。あとは同じディレクトリに arm9_main.cpp を作り、そこに C 言語感覚でコードを書いていきます。

4. BackGround

Makefileもできたので解説に移ります。NDSプログラミングでは、特定のアドレスに値を書き込むことにより、ハードウェア的に処理してくれるものがかなりあります。まずDSの画面はバックグラウンドとスプライトの表示で構成されていて、ここではバックグラウンドについて説明します。

バックグラウンド（以下 BG）には BG0~BG3 の 4 枚のレイヤがあり、モードを指定することでそれぞれを何に使うかを予め決めておく必要があります。

モード	BG0	BG1	BG2	BG3
0	Text/3D	Text	Text	Text
1	Text/3D	Text	Text	Affine
2	Text/3D	Text	Affine	Affine
3	Text/3D	Text	Text	Extended
4	Text/3D	Text	Affine	Extended
5	Text/3D	Text	Extended	Extended
6	3D	-	Large	-

モードを指定するのは videoSetMode()を使います。引数には、画面のモードとアクティブにする BG を指定します。MODE_n_2D でモード n に設定、DISPLAY_BGn_ACTIVE で BG n を有効化です。

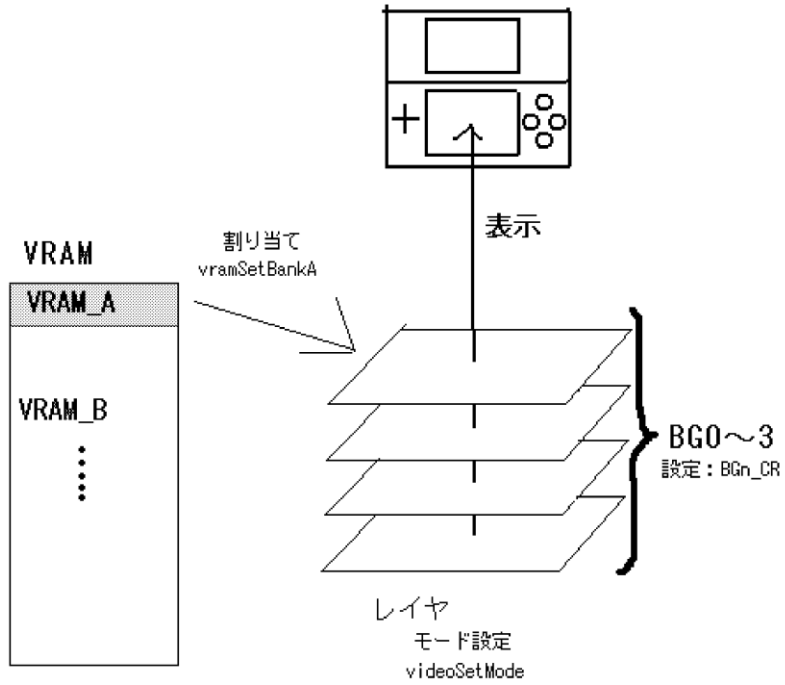
（例 1）
```
videoSetMode(MODE_0_2D | DISPLAY_BG0_ACTIVE);
```

（例 2）
```
videoSetMode(MODE_5_2D | DISPLAY_BG2_ACTIVE | DISPLAY_BG3_ACTIVE);
```

さらに、VRAM を割り当てる必要があります。プログラムによって使う BG やスプライトの容量に差があるので、どこにどれぐらい VRAM を割り当てるか指定できるようになっています。VRAM は A~I の 10 区画に予め分けられています。指定には `vramSetBankX()` を使います。X には設定したい VRAM の記号を、引数には何に使うかを指定します。VRAM_X_YYY で VRAM_X を YYY に使うということになります。

（例）
```
vramSetBankA(VRAM_A_MAIN_BG);
```

大まかに書くとこのようになってます。



BG を使う設定ができたら、次に BG 自体の設定をします。BGn_CR に BGn の設定を書き込みます。

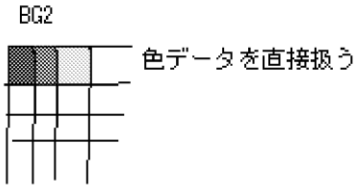
(例 1) `BG2_CR = BG_BMP8_256x256 | BG_PRIORITY(2);`

(例 2) `BG3_CR = BG_BMP16_512x512 | BG_BMP_BASE(16) | BG_PRIORITY(1);`

BG_BMPn_AxB で n 色 A*B サイズに設定します。BG には 8 ビットパレットモードと 16 ビットカラーモードがあり、前者はパレットエリアに書かれた色を番号で指定するもので、15 ビットカラーで同時に 256 色まで使えます。1 ドットにつき 1 バイト使います。後者は 15 ビットカラーを VRAM に直接入力し、その色が出力されます。同時に発色する制限はありません。1 ドットにつき 2 バイト使えます。

8ビットモード

16ビットモード



こう書いてみると、8 ビットパレットモードの利点が無いように見えますが、かなりあります。まず、各 VRAM の容量と割り当てられるモードの表は下記のようになっています。

	容量	BG	スプライト
VRAM_A	128KB	MAIN	MAIN
VRAM_B	128KB	MAIN	MAIN
VRAM_C	128KB	MAIN/SUB	-
VRAM_D	128KB	MAIN	SUB
VRAM_E	64KB	MAIN-	MAIN-
VRAM_F	16KB	MAIN-	MAIN
VRAM_G	16KB	MAIN-	MAIN
VRAM_H	32KB	SUB-	-
VRAM_I	16KB	SUB-	SUB-

スプライトは後述するとして、NDS は上画面と下画面、それぞれ MAIN と SUB に分かれています。MAIN や SUB の後の「-」はアドレスを指定できないことを意味します。どういうことかというと、MAIN_BG 用の VRAM は 0x6000000~0x6080000 で、512KB あり、これはメイン VRAM4 つ分になりますね。なので、これら全てを使う場合、その VRAM がどこから始まるか(0x6040000 など)指定しなければなりません。しかし「-」がついている VRAM はその指定ができず、先頭の 0x6000000 からになってしまうということです。ちなみにメイン VRAM4 つ全てを 1 枚の BG に使うと、8 ビットモードだと 1024*512、16 ビットモードで 512*512 使えます。今回は 8 ビットで 256*256 なので、ちょうど VRAM_A だけで足りるということです。

BG_PRIORITY(n)ではその BG の優先度を n に設定します。優先度は 0~3 の 4 つで、この値が小さいほど前面に表示されます。

BG_BMP_BASE(n)では、その BG のデータの先頭アドレスを指定します。1 につき 0x800 バイト進みます。BG_BMP_RAM(n)で n にここで使った値を指定するとこれがそのまま先頭のアドレスになるので、簡単にアクセスすることができます。

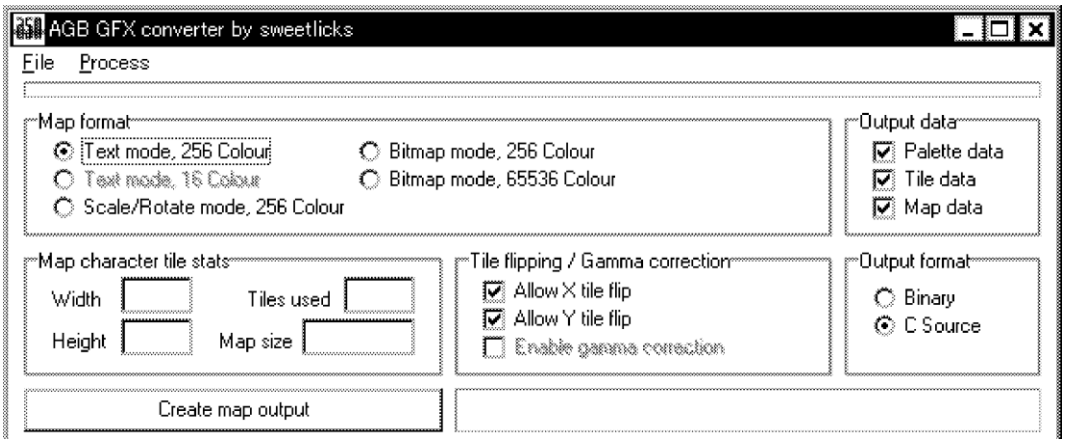
最後に大きさなどの設定をします。これは Extended な BG2 と BG3 限定です。
BGn_CX と BGn_CY で BGn の位置を設定します。画面左上を(0,0)とし、左上が正の方向です。値は 8 ビット上位にシフトしてから入れます。

BGn_XDX, BGn_XDY, BGn_YDX, BGn_YDY では回転と伸縮を設定します。<trig_lut.h> をインクルードし、以下のように書き込みます。

```
s16 s = SIN[angle & 0x1FF] >> 4;
s16 c = COS[angle & 0x1FF] >> 4;
BGn_XDX = ( c * scaleX ) >> 8;
BGn_XDY = (-s * scaleX ) >> 8;
BGn_YDX = ( s * scaleY ) >> 8;
BGn_YDY = ( c * scaleY ) >> 8;
```

angle が角度(一周 512)、scaleX と scaleY がそれぞれ X,Y の伸縮を表します。256 が標準で、増えると縮み、減ると伸びます。

長かったですが、以上で BG の設定は終わりです。後はデータを転送するだけです。まずは表示させたい 256*256、256 色ビットマップの bmp ファイルを用意してください。変換には AGBFXConverter を使います。http://www.gbadev.org/ の Tools->Graphics->GBA GFX converter です。ついでに Graphics にある bmp2spr もダウンロードしておきましょう。起動すると



こんな感じになるので、File->OpenBitmap から用意した画像を読み込み、右上の Tile data、真ん中の Allow-二つのチェックを外し、BitmapMode, 256 Colour をチェックしたら Create ボタンを押します。すると画像と同じディレクトリに画像名.c というファイルができるので、拡張子を.h に変え、開いて <agbtyoe.h> のインクルードを削除し、arm9_main.cpp からそのファイルをインクルードします。あとはそこに書かれているデータを然るべき場所に転送するだけです。転送には dmaCopy を使います。

（例）
```
dmaCopy(gazou_Palette, BG_PALETTE, 512);
dmaCopy(gazou_Map, BG_BMP_RAM(16),256*256);
```

ちなみに BG は 8 ビットパレットモードの 256x256、BG_BMP_BASE(16)を指定したときのものです。16 ビットの場合は変換のときのチェックを Bitmap mode, 65536 Colour にし、上の行のパレットの転送を無くし、下の行の 256*256 を 256*256*2 にするだけです（16 ビットモードでは 1 ドットにつき 2 バイト使うため）。これで上画面に用意した画像が表示されるはずです。

5. Double Screen

DS です。4 章をやってみて思いませんか？下画面にも表示させたいと。せっかくのダブルスクリーンなんですからね。ということで、ここでは両画面表示について説明していきます。まず前回のプログラムでは MAIN の方を使っていました。しかし MAIN/SUB と上/下画面の区別は別物なので、下画面をメインスクリーンにすることもできます。lcdSwp();と書くだけです。これを書くことで、好きなタイミングで上下の画面を入れ替えられます。

さて本題に入りましょう。MAIN の設定では、まず画面モードの設定をしました。同様の事をするには videoSetModeSub()を使います。

（例）
```
videoSetMode(MODE_0_2D | DISPLAY_BG0_ACTIVE);
```

引数はMAINのときと同じように指定すればOKです。

次に vram の割り当てです。これは vramSetBankX の引数の MAIN を SUB に置き換えるだけです。

（例）
```
vramSetBankA(VRAM_C_SUB_BG);
```

BGn_CR も BGn_CX などの各種 Extended 設定も、SUB_BGn_CR や SUB_BGn_CX など前に SUB_をつけるだけですので難しくはないと思います。

最後に画像を転送する場所ですが、これも BG_PALETTE を BG_PALETTE_SUB に、BG_BMP_RAM(n)を BG_BMP_RAM_SUB(n)にすればいけます。

こんな感じで基本的にはアドレスを指定する際にどこかに SUB の文字を追加するだけで SUB 画面も使えるようになります。

6. スプライト

次にスプライトの説明をします。スプライトとは、画像をハードウェアでバックグラウンドと合成

し画面に表示するもので、これを使った画像表示をすることで、画像の位置や角度を変更するときに一々データを転送したり難しい計算をしなくてよくなるなどの利点があります。NDS では、スプライトのデータ用のメモリ領域があり、そこに一見無秩序に入れられているデータを、OAM というパラメータを使って一つ一つのスプライト画像としてきりわけていきます。OAM にはキャラクタ番号というものが保存されており、スプライトデータ領域の先頭からこの番号*16 バイト進めたところが、その画像の先頭となります。他に OAM には画像の大きさやスプライトの表示する座標、優先度や有効/無効の設定などが入っています。ちなみに OAM は MAIN と SUB でそれぞれ 128 個ずつ使えるので、スプライトは基本的には一画面 128 個までということです。画像の大きさは最大 64*64 です。

まずは画面をスプライト表示に対応させます。videoSetMode の引数に DISPLAY_SPR_ACTIVE と DISPLAY_SPR_1D と DISPLAY_SPR_1D_BMP を追加で指定します。さらに前記の表でスプライトに対応している vram をスプライトに割り当てます。

(例) `vramSetBankE(VRAM_E_MAIN_SPRITE);`

次に OAM の宣言をします。型は libnds で定義されている SpriteEntry です。

(例 1) `SpriteEntry sprites[128];`

(例 2) `SpriteEntry sprites_sub[128];`

さらにスプライトの回転角度を便利に変更するために、pSpriteRotation を宣言します。これは MAIN と SUB それぞれ 32 個使えます。つまり回転伸縮できるスプライトは同時に 32 状態までということですね。後で解説しますが同じ Rotation 番号を指定すればそれらのスプライトの回転伸縮を同調させることができます。

(例) `pSriteRotation spriteRotations = (pSpriteRotation)sprites;`

このようにキャストした SpriteEntry へのポインタを代入するのを忘れないでください。

しかしこのように勝手に変数を宣言しただけでは、OAM に適用はされません。dmaCopy で OAM または OAM_SUB にコピーする必要があります。また OAM にはキャッシュ機能がついており、ただ sprites などをコピーしただけでは変更が適用されないとこがあります。これを防ぐためにキャッシュを消去するマクロを作っておきましょう。消去後に SpriteEntry の内容を OAM に転送するようにすると便利です。キャッシュの消去には DC_FlushRange または DC_FlushAll を使います。

N(なんて)D(どうでもいい)S(作文だろうか)

```
(例) #define updateOAM() DC_FlushRange(sprites, 128*sizeof(SpriteEntry)); ¥
        dmaCopy(sprites, OAM, 128*sizeof(SpriteEntry)); ¥
        dmaCopy(sprites_sub, OAM_SUB, 128*sizeof(SpriteEntry))
```

SpriteEntry には attribute[4]という変数があり、pSpriteRotation には hdx, hdy, vdx, vdy という変数があります。前者は座標や優先度や有効/無効などの情報、後者は回転伸縮の情報が入っています。回転伸縮の値の指定は BG のときと同じ感覚でできます。Attribute の主な値は以下のようになっています。

attribute[0]	y 座標		
	モード	ATTR0_DIABLED	無効
		ATTR0_NORMAL	通常
		ATTR0_ROTSCALE	回転(90 の倍数度のみ)
		ATTR0_ROTSCALE_DOUBLE	回転
	カラーモード	ATTR0_COLOR_16	パレット 16 個使用
		ATTR0_COLOR_256	パレット 256 個全使用

attribute[1]	x 座標		
	サイズ	ATTR1_SIZE_n	8,16,32,64 のみ
	回転伸縮データ	ATTR1_ROTDATA(n)	SpriteRotation の添え字

attribute[2]	キャラクタ番号		16 バイトにつき 1 進む
	パレットオフセット	ATTR2_PALETTE(n)	16 パレットモード時
	優先度	ATTR2_PRIORITY(n)	BG の優先度と同じ形式

```
(例) sprites[0].attribute[0] = ATTR0_ROTSCALE_DOUBLE | ATTR0_COLOR_16 | 100;
     sprites[0].attribute[1] = ATTR1_SIZE_16 | ATTR1_ROTDATA(3) | 20;
     sprites[0].attribute[2] = ATTR2_PALETTE(16) | ATTR2_PRIORITY(2) | 8;
```

順に解説していくと、まず座標は画像左上の画面上での座標になります。モードでROTSCALE というのは SpriteRotation を通しての回転が可能ですが、有効領域が元の画像のサイズ分しかないので 45° とか回転すると端が表示されません。ROTSCALE_DOUBLE だと有効領域が 2 倍になり、自由に回転できますが、位置も 2 倍のサイズの画像があると思って指定しなければいけないことに注意です。ATTR0_COLOR では使うパレットの個数を指定します。スプライトはパレットモードしかありません。パレットには 256 色ありますがここから 16 色のみ使うという設定ができるということです。これにより 1 ドットが 4 ビットで表せ、BG での 8 ビットパレットモードと同じような利点があります。

ATTR1_SIZE_n ではスプライトの画像サイズを設定します。ここからわかるように、画像は 8*8、16*16、32*32、64*64 しか使えません。ATTR1_ROTDATA(n)では attribute[0]で ROTSCALE(DOUBLE)にした場合、その回転伸縮情報を適用する pSpriteRotation の添え字です。例をつなげて見るとこの場合 sprites[0]の回転伸縮情報は spriteRotations[3]に格納されていることになります。

ATTR2_PALETTE(n)では、ATTR0_COLOR_16 を指定した場合に、その 16 個の先頭のパレット番号を設定します。ATTR2_PRIORITY(n)は、BG のときに解説した優先度と同じ形式で設定でき、BG との互換性があります。同優先度では BG よりスプライトの方が優先されます。

それでは実際にスプライトを使えるようにしてみましょう。まずは初期化をします。

```
void InitSprite(void)
{
 int i,j;
 DC_FlushAll();
 for(i=0;i<128;i++){
  sprites[i].attribute[0] = ATTR0_DISABLED;
  sprites_sub[i].attribute[0] = ATTR0_DISABLED;
  for(j=1;j<4;j++){
   sprites[i].attribute[j] = 0;
   sprites_sub[i].attribute[j] = 0;
  }
 }
 for(i=0;i<32;i++){
  spriteRotations[i].hdx=256;
```

```
        spriteRotations[i].hdy=0;
        spriteRotations[i].vdx=0;
        spriteRotations[i].vdy=256;
        （省略 sub も同様）
      }
    }
```

こんな感じです。

設定ができたら、次は画像を用意します。変換には bmp2spr を使います。先ほどダウンロードしておいたものです。これはコマンドプロンプト専用で、コマンドは以下の様に打ちます

```
bmp2spr 元画像名.bmp 出力名.h
```

これで出力名.hが作成されます。あとはこれをarm9_main.cppからインクルードすれば画像の用意は完了です。ちなみにこの変換方法だと256全パレットモードになります。あとは画像を転送するだけです。今回もdmaCopyを使います。スプライト用パレットはSPRITE_PALETTEとSPRITE_PALETTE_SUB、データはSPRITE_GFX+キャラクタ番号*16です。前述の例で言うと、

```
（例）dmaCopy(sprgazouPalette,SPRITE_PALETTE,512);
    dmaCopy(sprgazouData, SPRITE_GFX+ 8*16, 16*16);
```

特殊な使い方をしないなら、ファイル中にある変数、画像名 Palette を Palette にリネームして InitSprite で渡しておいたほうが便利です。

```
(例)void InitSprite(void){
        …
        dmaCopy(Palette, SPRITE_PALETTE, 512);
        dmaCopy(Palette, SPRITE_PALETTE_SUB, 512);
    }
```

これで（20,100）にsprgazouを表示させることができます。

最後に、あると便利な関数の一例を紹介します。

```
    void MoveSprite(SpriteEntry *spriteEntry, u16 x, u16 y)
    {
     spriteEntry->attribute[1] &= 0xFE00;//座標エリア下位 9 ビットを初期化
     spriteEntry->attribute[1] |= (x & 0x01FF);//x の値が大きすぎたときのために切り
```

詰める

```
 spriteEntry->attribute[0] &= 0xFF00;//y 座標は 8 ビット
 spriteEntry->attribute[0] |= (x & 0x00FF);
}
void RotateSprite(SpriteRotation *spriteRotation, u16 angle)
{
 s16 s = -SIN[angle & 0x1FF] >> 4;
 s16 c = COS[angle & 0x1FF] >> 4;
 spriteRotation->hdx = c;
 spriteRotation->hdy = -s;
 spriteRotation->vdx = s;
 spriteRotation->vdy = c;
}
```

7. キー入力

今までさんざん画像出力を解説してきましたが、何か足りないと感じませんか？そう、それはユーザーからの入力です。まあ他にもいろいろと足りないんですけど。ここでは XY を除いたキー入力の判定方法について解説していきます。なぜ除くかというと、XY キーは特殊で ARM7 でしか感知できないからです。

キーが押されるとREG_KEYINPUTが更新され、押されているキーに応じた値になります。

ビット	キー	定義
0	A	KEY_A
1	B	KEY_B
2	Select	KEY_SELECT
3	Start	KEY_START
4	右	KEY_RIGHT
5	左	KEY_LEFT
6	上	KEY_UP
7	下	KEY_DOWN
8	R	KEY_R
9	L	KEY_L

押されている間は 0、押されてないとき 1 となり直観的でないのでビット反転して使いましょう。

（例）
```
u16 Keys = ~(REG_KEYINPUT);
if(Key & KEY_A)
…
```

これだけですが、これをただソースに書いただけでは起動した後の一瞬の間だけキー入力判定をするだけになってしまいます。ですので while でループさせてチェックすることにします。

```
while(1)
{
  （先程の例）
}
```

しかしこのようにすると、PC で組んだことのある人ならわかると思いますが、CPU 使用率が 100%になってしまいます。これではとても使えません。PC では Sleep(1)などを使って 1 ループ 1/1000 秒に制限していました。NDS では垂直同期をとります。垂直同期というのは、画面の垂直復帰に割り込んで処理をすることで、NDS では 1/60 秒に一回、垂直復帰が行われます。この垂直復帰が起こったときに特定のフラグを ON にするなどして while 内で感知すれば、画面のリフレッシュレート 1/60 秒に合わせて while が 1 ループするようになるのです。

まずは垂直復帰割り込みの設定をします。詳しくはわかっていなくて申し訳ありませんが、以下のように書いてください。

```
void InitInterruptHandler(void)
{
 IME = 0;
 IRQ_HANDLER = yarukoto;//割り込んだとき関数 yarukoto()を実行する
 IE = IRQ_VBLANK;//垂直復帰のときに割り込む
 IF=~0;
 DISP_SR = DISP_VBLANK_IRQ;
 IME = 1;
}
```

while ループ前に InitInterruptHandler()を実行しておけば、垂直復帰に割り込んで yarukoto()が実行されます。前述した割り込みフラグ云々ですが、もっと適した関数があります。swiWaitForVBlank()です。これは垂直復帰割り込みがされたことを変数に通知するまで停

止しているという関数です。通知は

```
VBLANK_INTR_WAIT_FLAGS |= IRQ_VBLANK;
IF |= IRQ_VBLANK;
```

これで通知できます。これらを yarukoto()内に書いておけばいいですね。あとは while 内の最後にでも swiWaitForVBLANK();と書いておけば 1/60 秒周期でループしてくれます。もっとも、while 内の処理が遅くて次の垂直復帰に間に合わなかった場合は、次の垂直復帰を待つのでその倍かかったりしますが。

8. おわりに

長々と書いてきましたが、NDS プログラミングはまだまだ奥が深いです。今回は使ってなかった arm7 や、音を鳴らす方法、GBA スロットへのアクセス、BG やスプライトの透過処理など、ここで語るにはあまりに複雑なことが多々あります。そこで、解説のページを作ろうかと思いましたが、そうこうしているうちに DS プログラミングを取り扱うサイトが増えてきました。また自身が多忙なため書く暇がないです。おそらく今後何かあるとすれば次回のパ研の部誌、電脳 2008 に書くと思います。