自主課題研究レポート
3 年 64 番小野祐貴

研究テーマについて

ゲームボーイアドバンス用ソフトウエアの設計

任天堂から 2001 年 3 月 21 日に発売された、
ゲーム機器の「ゲームボーイアドバンス（以下 GBA)」で実際に動作出来るプログラムを作成した。

GBA のアーキテクチャは以下の通り

ARM ARM7TDMI 32bit RISC CPU, 16.78MHz, 32bit

内部メモリ

BIOS ROM　　16 KBytes
作業用 RAM　288 KBytes (32K in-chip + 256K on-board)
VRAM　　96 KBytes
 背景モード 0,1,2
  06000000-0600FFFF　64 KBytes タイル、マップ用
  06010000-06017FFF　32 KBytes スプライト用
 背景モード 3
  06000000-06013FFF　80 KBytes ビットマップ用（使用するのは 75 KBytes)
  06014000-06017FFF　16 KBytes スプライト用
 背景モード 4,5
  06000000-06009FFF　40 KBytes フレーム 0 ビットマップ用
  0600A000-06013FFF　40 KBytes フレーム 1 ビットマップ用
  06014000-06017FFF　16 KBytes スプライト用
OAM　　1 KByte (128 OBJs 3x16bit, 32 OBJ-Rotation/Scalings 4x16bit)
  07000000-070003FF　1 KByte OAM スプライト用
Palette RAM　1 KByte (256 色 背景用、256 色 スプライト用)
  05000000-050001FF - 背景パレット（512 Bytes, 256 色）
  05000200-050003FF - スプライトパレット（512 Bytes, 256 色）

開発環境について
devkitPro　(gcc ARM 用コンパイラセット)
VisualBoyAdvance　V1.7.2　(GBA エミュレータ環境)
Eclipse　CDT　(Eclipse C 言語開発用　プラグイン)
Java　(素材を GBA 用に変換するプログラムに使用)
Adobe Flash CS4 (軌道設定に使用)
Microsoft Expression Design 3　(素材作成に使用)

プログラム仕様・開発方法について

C 言語で開発を行った (gcc でコンパイル可能な言語では別の言語 (アセンブラ・C++) でも開発は可能)
上記の各種レジスタにアクセスする方法

```
short* oam = (short *) 0x07000000;
oam[10]=0;
```

と変数宣言することで、0x07000000 のレジスタのポインタにアクセスができる。
oam[10]とすることで、　0x07000000+2*10=0x07000020
ちなみに、GBA の内部バスは 16 ビット(2 バイト) で構成されているため、最小単位の 1 バイト(8 ビット) で内部メモリにアクセスしようとしても。　次の 1 バイト分もアクセスされてしまう。
つまり、

```
char* oam = (char *) 0x07000000;
oam[0]=1;

//0x07000000 -> 1
//0x07000001 -> 1
```

上記のエミュレータで確認すると、2 バイト分「1」となっていることがわかる。　よってレジスタには 2 バイトごとにアクセスする必要がある。

内部レジスタは、1 ビットごとに機能が割り当てられており
各種レジスタの詳しい仕様については、以下などを参照

ＧＢＡメモ
http://m-niwa.hp.infoseek.co.jp/gba/gba_tech.html

GBATEK
http://nocash.emubase.de/gbatek.htm

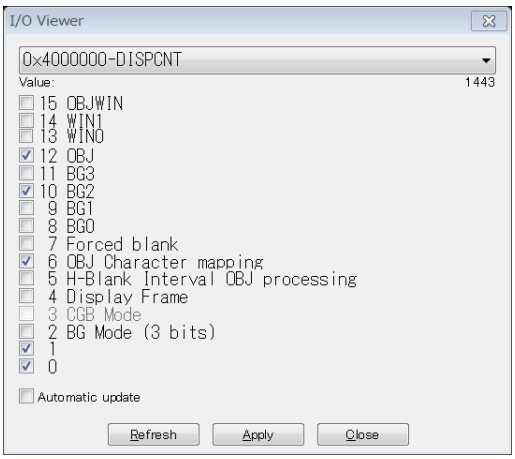
#define gba_reg(p) *((volatile hword*) p) // access to conrtol gba_regs

というマクロを作成することによって、上記のプログラムは
gba_reg(0x07000000)=0

と簡潔に記述することができる。

0x04000000 ～0x04000300 は I/O レジスタと呼ばれる。
各種 設定レジスタが用意されている。　ここで、背景モードやスプライトの設定・タイマーなどを行うことができる。

VisualBoyAdvance　ではプログラム実行時に、tools->I/O viewer で確認することができる。（図 1）
図 1. VisualBoyAdvance 上でレジスタの確認

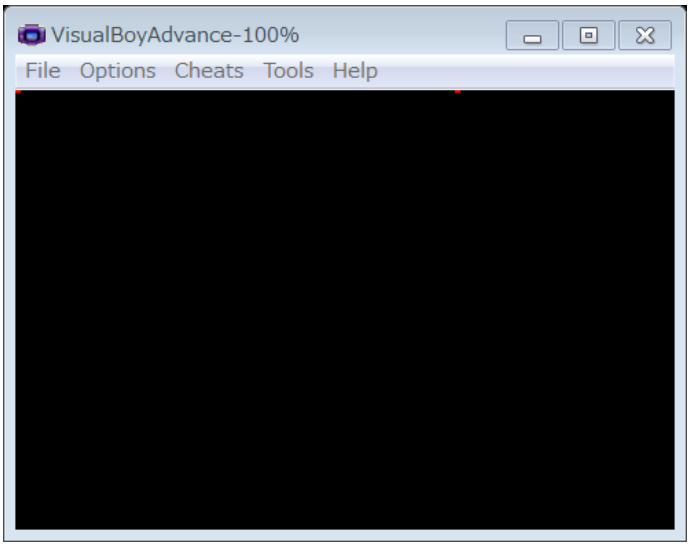


レジスタなどのわかりやすい例としては

gba_reg(LCD_CTL) = LCD_BG2 | LCD_MODE3 | LCD_OBJ_MAP1D | LCD_OBJ;
 // Mode3, BG2 1D 方式 表示
short* vram = (short *) 0x06000000;
vram[0]=31;
vram[1]=31;
vram[160]=31;
vram[160+1]=31;

LCD_CTL は　0x04000000 で、VisualBoyAdvance 上では、
図 1 のように DISPCNT となっている、LCD_BG2=1<<10, LCD_MODE3=3, LCD_OBJ_MAP1D=1<<6, LCD_OBJ=1<<12 の定数である。
これによって、背景モード 3、BG2 を使用する、1D 方式、キャラクタマッピングモードということになる。
背景モード 3 は、背景の大きさが 240*160 となっている。

Mode3 は、カラーモードが 32768 色 15 ビット　0BBBBB GGGGG RRRRR の並びになっており
31 =0000000000011111 (2) なので　左上に 4 マス赤色が、　表示される。(図 2)
図 2. 左上 4 マスドットが赤色に設定した図



スプライトについて
上記の背景にドット打ちだけでも、簡単な表示は満足できるが、ゲームのような高速描画を必要とするものは　オブジェクト（スプライト）と別の機構を用いることにより、高速かつ簡単にオブジェクトを管理ができる。

使用できるスプライトの種類は、以下の 12 種類の大きさで最大 32768 色中 256 色を同時に使用できる。

8x8　　16x8　　8x16
16x16　　32x8　　8x32
32x32　　32x16　　16x32
64x64　　64x32　　32x64

デバッグ表示

GBA エミュレータでは、MappyPrint と呼ばれる方法で実行ログを出力することができる。
（参考：GBA homebrew 日記　http://d.hatena.ne.jp/akkera102/20090504）
こういう関数を作る。処理内容はインラインアセンブラになってっていて、関数の引数 でOxff0000の命令を実行するというものである。

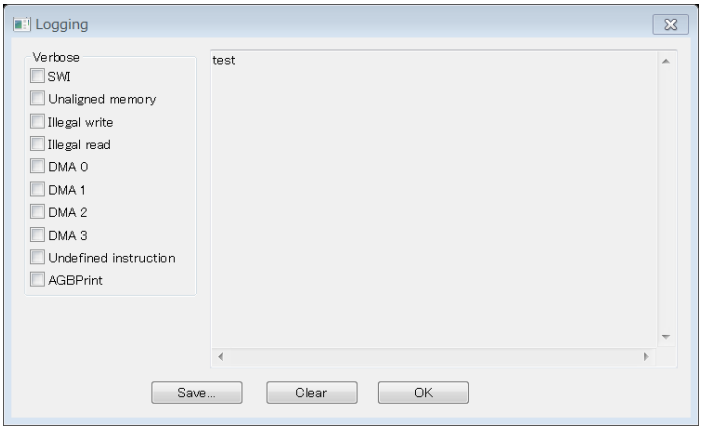
```
#include "gba.h"

void MappyPrint(char* s) {
        asm volatile("mov r0, %0;"
                     "swi 0xff0000;"
                     : // no ouput
                     : "r" (s)
                     : "r0");
}
int main() {

        MappyPrint("test");
        while(1);
}
```

上のコードでデバッグ表示ができる。（ただし、GBAエミュレータのバージョンが1.7.2でないと上手くいかないようです）

GBAエミュレータ上の(Tools->Logging)　ウィンドウ



数値を文字列に変換する関数を作ることで、プログラム中の変数デバッグも可能になる。

動画の表示まで

興味本位で動画を表示できるのではないかと思って、挑戦した。
その途中までの大まかの過程をたどると。
タイマー
割り込み
DMA
サウンド
画像表示
がある。
タイマーについて
タイマーは 4 種類使うことができる。
0x04000100 タイマー 0 カウンタ
0x04000102 タイマー 0 コントロール
0x04000104 タイマー 1 カウンタ
0x04000106 タイマー 1 コントロール
0x04000108 タイマー 2 カウンタ
0x0400010A タイマー 2 コントロール
0x0400010C タイマー 3 カウンタ
0x0400010E タイマー 3 コントロール

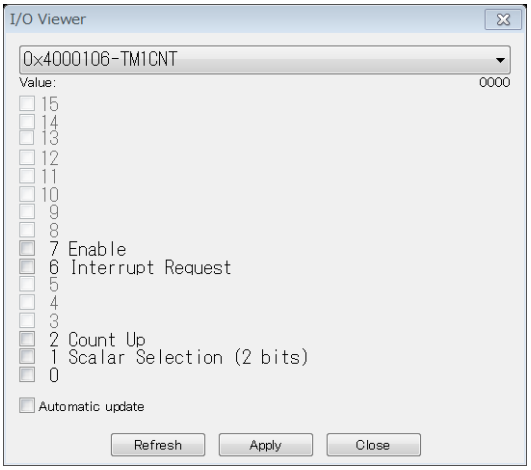
カウンタは、16bit 使うことができて、
コントロールの 0-1 ビットの Scalar Selection によって 0-F ,1-F/64 ,2-F/256, 3-F/1024 F=16.78MHz のタイマーひとつに付き、最大で 61 マイクロ秒間隔 16bit カウンタで約 4 秒を計測できる。
2 ビット目の Count Up を指定すると、タイマー1 ならタイマー0 がオーバーフローした際の周期でカウントができる。よって、最大でタイマー4 つで 32bit カウントタイマーまでつくれる。

6Bit 目の Interrupt Request を 1 にすると、割り込みがかかり、時間ごとに割り込みハンドラに割り込みが入る。(後述)

7bit 目の Enable を 1 にすると、タイマーがスタートされる。
カウンタのレジスタには、カウント中の値を見ることができる。
カウンタに値を直接入れると、その値からカウントを開始する。
その後、オーバーフローをすると、入れた値からカウントを再開するという仕様である。
これを用いれば、細かな周期も計測ができる。

つまり計算式としては　F/1 の設定なら、16.78MHz=(16777216Hz)
gba_reg(TIMER_CNT0)=65536-16777216/freq;　(freq:　整数型　256 以上の数値)
とすれば、任意の周波数のカウントが出来る。
ただし、256Hz 以下の場合は、この式では表せないので先程の分周比を変えるなどの処理を擦る必要がある。

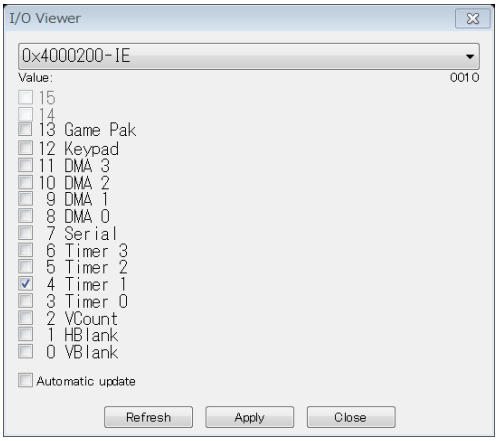
図 3。サウンドコントロールの図



割り込みについて
GBA では、割り込みを実装するときは
内部メモリの 0x03007FFC 番地に　呼び出す関数（アドレス）を指定することで割り込みができる。

```
int main(){
(*(int*) 0x03007FFC) = (int) InterruptProcess;
}
```

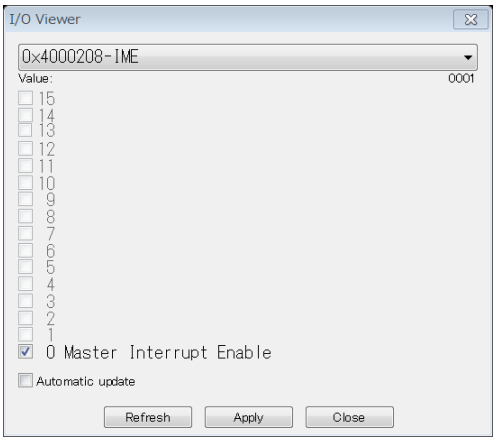
上のコードだけで、割り込み時には、InterruptProcess の関数が実行されるということになる。
図 4。IE レジスタの図



タイマーで実際に割り込みを行うには、図 4 のように
gba_reg(0x04000200)|=INT_TIMER0;

(INT_TIMER0:=1<<3)
として、割り込みを登録する必要がある。

さらに Interrupt Master Enable (0x0400208)のレジスタも 1 にする必要がある。
gba_reg(0x04000208)=1;
図 5。IME レジスタの図



実際の関数を述べると
一度、IME のレジスタを 0 にして重複の割り込みの実行を避けている。
gba_reg(INT_IF); は、割り込みされた例外の種類を得ることができます。
これをもって割り込みの種類を分けていきます。
INT_IF:=0x04000202

```
void InterruptProcess(void) {
        int flag = 0;

        gba_reg(INT_IME) = 0;
        flag = gba_reg(INT_IF);
        if (flag & INT_TIMER0) {
                /* Timer0 の処理
                  */
        }
        if (flag & INT_DMA1) {

                /*    DMA1 の処理                    */
        }
        if (flag & INT_KEY) {
                /* Key の処理  */

        }
```

```
        gba_reg(INT_IF) = flag;
        gba_reg(INT_IME) = 1;
}
```

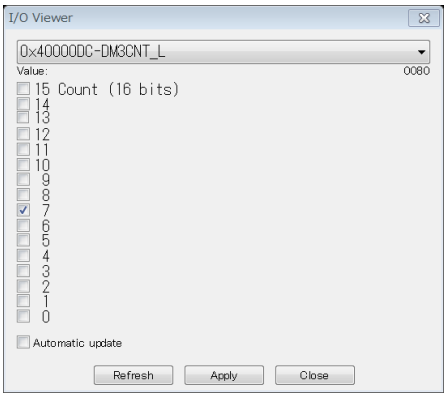
DMA について

GBA には、データを高速で転送する DMA という仕組みがある。
DMA には 4 つあるが、ここでは DMA1 を使っている。
FIFO バッファとリンクできるのは DMA1,DMA2 だけである。
これをつかうのは、簡単で
以下の 4 行のみで良い

```
#define  gba_reg32(p)      *((volatile int*) p)          // access to conrtol gba_regs

gba_reg32(0x040000BC) = (int) src;
gba_reg32(0x040000C4) = (int) dst;
gba_reg(0x040000DC) = length / 2;
gba_reg(0x040000C6) = 0x8000;
```

32 ビット長でアクセスするため、gba_reg32(p)を新たに定義した。
図 6. 0x040000DC レジスタの図

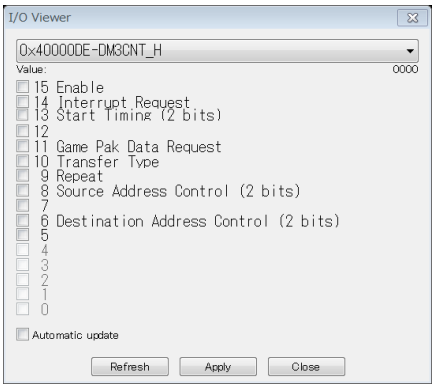


0x040000D4 にはコピー元のアドレス
0x040000D8 にはコピー先のアドレス
0x040000DC は、コピーする大きさを指定するが 16bit(2 バイト)事にコピーされるため、2 バイトで割った値を設定する。
0x040000DE　の 15 ビット目を 1 にした瞬間に DMA が開始される。自分でコピーするよりも何倍も早

くコピーが完了する。
14 ビット目を 1 にすると、DMA の完了後に先程の割り込みが発生する。
12-13 ビット目の Start Timing（開始時間）は、サウンドに重要なので後述する。

5-6 ビットは 転送先の転送ごとのポインタ（0=増加, 1=減少, 2=固定, 3=増加/リロード）
7-8 ビットは 転送元の転送ごとのポインタ（0=増加, 1=減少, 2=固定, 3=使用しない）
これらの 2 つは、0 を指定すると 0x040000D4 などのアドレスが増加して、連続で呼び出したときに次のデータをすぐに転送できる。　2 を指定すると転送先は固定することが可能になる。

図 7. 0x040000DE レジスタの図



サウンド再生について

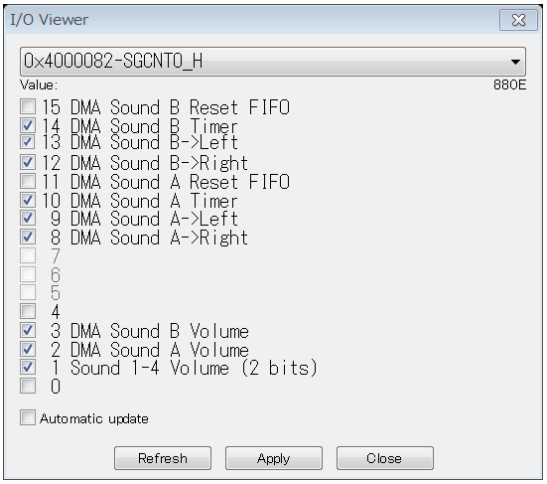
GBA のサウンド再生について行った。

GBA 上でサウンドを鳴らすときは、
レジスタ 0x040000A0（A チャンネル), 0x040000A4（B チャンネル）にそれぞれ PCM データを書き込むことでサウンドとして再生される。FIFO バッファである。

上記の DMA・割り込みを用いる。
FIFO バッファが、実際に音を成らす PWM にデータを送るタイミングは
0x04000082 のレジスタのそれぞれ 10,14 ビット目の値が、
0 なら Timer0 のオーバーフローのタイミング
1 なら Timer1 のオーバーフローのタイミングとなる。
図 7 では、Timer1 のタイミングとなる。

8，12 は、A,B のチャンネルを右に振るのか、9,13 は左に振るかである。
チャンネル A のみで、左右に振ると　モノラル音源

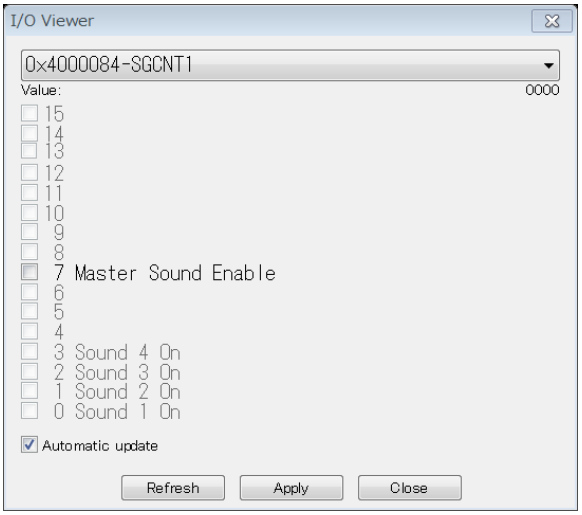
A,B をそれぞれ　左・右に振るとステレオも可能である。
図 7. 0x04000082 レジスタの図



さてここで、PCM データというのは、GBA では符号付き 8 ビット整数型となり。
音の高さをデジタル値で表したものになる。

127　最大
|
0　　無音
|
-128　最大

のこぎり波とすると
char wave_data[] = { 0, 127,0,-128};
のように表す。
最後に
0x04000084 レジスタの 7 ビット目を 1 にするとサウンドが有効になる。
図 8. 0x04000084 レジスタの図



ここまでで、音はならせるかと思うので

```
char wave_data[] = { 0, 127, 0, -127 };
//サウンドデータの転送
void data_copy() {
	gba_reg32(0x040000BC) = (int) &wave_data;//コピー元
	gba_reg32(0x040000C0) = (int) 0x040000A0;//コピー先
	gba_reg(0x040000C4) = 2;
	gba_reg(0x040000C6) = 0xF540;

}
void InterruptProcess(void) {

	int flag = 0;
	gba_reg( INT_IME) = 0;
	flag = gba_reg(INT_IF);
	if (flag & INT_DMA1) {
//DMA の転送後
		data_copy();

	}
	gba_reg( INT_IF) = flag;
	gba_reg( INT_IME) = 1;
}
int main() {

	(*(int*) 0x03007FFC) = (int) InterruptProcess;
	short freq = 440;
```

```
        data_copy();
        gba_reg(0x04000084) = 0x80;
        gba_reg(0x04000082) = 0x3308 | 0x4000;

        //タイマーの設定
        gba_reg( 0x04000100 ) = 65536 - 16777216 / freq;
        gba_reg( 0x04000102 ) = 1 << 7;
        //割り込み許可
        gba_reg(INT_IE) |= INT_DMA1;


        gba_reg(INT_IME) = 1;

        while(1);
}
```

Wave(RIFF)から GBA 用 Wave への変換

Wave フォーマットの解説によりますと、(参考 http://www.kk.iij4u.or.jp/~kondo/wave/)
ヘッダー部分の処理は、本項とはあまり関係なので省略する。(変換スクリプト作成するなら考慮する必要が有るかも)
通常、データが 2C=44 バイト目から始まり、ビット長が符号 2 バイトの時(16bit サウンド)リトルエンディアンとなっている、
で、32767(7FFF)～ -32768(8000) の値で　0 が消音である。
GBA では 8bit サウンドであり、256(FF)～0 (00)の値をとり、128 (80)が消音となる。
よって変換には、　元データを　8 ビット右シフトして+128 加算したら GBA サウンドデータとして使える。
元データ列を D[n]とすると
Result[n/2]=((D[n]+D[n+1]*256)>>8)+128;

そのサンプルデータ列を、変数配列として書き出しコンパイル時に含め、先程の wave_data として、元データのサンプル周波数を freq に設定すれば (デフォルトでは、44100) すれば GBA 上で　音楽再生が可能。(44100 はデータ数が多いので、間引いて 22050Hz としてもいいかもしれないが。)

映像表示部

映像表示部は、背景モードでフレームごとに全ピクセル書き換えることで実現できる。
一般的な動画ファイルは、音声と画像の配置が交互になっているがここではわかりやすいように動画、映像部分の連続となっているとする。

映像元のフレームごとのビットマップがあるのであれば、
（参考：ビットマップフォーマット：http://www.kk.iij4u.or.jp/~kondo/bmp/）
ヘッダー部分は省略することにして
データ部分が 1 バイトごと R,G,B　と並んでいると、1 ピクセル分のデータは
元データ列をそれぞれ、D[n],D[n+1]、D[n+2]とすると

```
int R = (D[n]& 0xFF) >> 3;
int G = (D[n+1] & 0xFF) >> 3;
int B = (D[n+2] & 0xFF) >> 3;
data= (short) ((R << 10) + (G << 5) + B);
```

とできる。つまり　240*160 だと、1 フレーム 240*160*2 バイト必要になる。
フレーム分のデータ　1 分 30fps 想定なら　240*160*2*30 バイト分必要になる。
30fps=(0.03Hz)は　F/1024 モードで 1/12 で表せるので

```
void Movie_Set() {
	u16* vram = (u16*) VRAM;//0x06000000;
	//DMA転送準備
	gba_reg32(0x040000D4) = (unsigned int) (movie_pointer + 240 * 160* movie_count * 2);
	gba_reg(0x040000DC) = 240 * (160) * 2 / 2;
	gba_reg(0x040000DE) = 0x8000;
	gba_reg( TIMER_CTL3) = (1<<7) | 3;
	gba_reg( TIMER_CNT3) = 65536 – (16777216/1024*30);

	gba_reg( TIMER_CTL3) = TIMER_ENABLE | TIMER_CASCADE | TIMER_INTREN;
	gba_reg(INT_IE) |= INT_TIMER3;
	gba_reg(INT_IME) |= 1;

}
```

ここでタイマーコントロール　で　1<<7 はタイマーEnale ビット　3 は 1/24 スケールと言うことである。
movie_pointer は、動画ファイルの先頭ポインタ、movie_count は再生してるフレーム数。
また、動画は映像部分の連続となっているとする

先程の音声部分のコードについかすることにより

```
void InterruptProcess(void) {
        int flag = 0;

        gba_reg(INT_IME) = 0;
        flag = gba_reg(INT_IF);
        int_flag = flag;
        if (flag & INT_DMA1) {
                //DMA の転送後
                data_copy();

        }
        if (flag & INT_TIMER3) {
        //映像部分処理
                gba_reg32(0x040000D4) = (unsigned int) (movie_pointer + 240 * (160)
                                * movie_count * 2);
                gba_reg32(0x040000D8) = (int) vram;
                gba_reg(0x040000DC) = 240 * (160) * 2 / 2;
                gba_reg(0x040000DE) = 0x8000;
        }
        gba_reg(INT_IF) = flag;
        gba_reg(INT_IME) = 1;
}

int main(void) {
        ～～省略～～
        //動画処理の初期化
                        Movie_Set();
}
```

BIOS 関数を使う

参考:http://nocash.emubase.de/gbatek.htm#biosfunctions

例えば、BIOS Arithmetic Functions—＞SWI 0Ah (GBA) - ArcTan2

を使うとする。

```
static int aTan(int *x, int *y)
//------------------------------------------------------------------------------
/*
 アークタンジェント（arc tangent）を計算する。
 */
{
        register int result;
        __asm ("SWI      0x0A<<16¥nmov %0,r0¥n" :  "=r"(result) :: "r1", "r2", "r3");
        return result;

}
```

SWI は BIOS 関数呼び出し　0x0A　<<16 は　通常モードの 0x0A 命令を実行する。=r(result)は結果が帰ってくる。ポインタとして渡してる x は r1,y は r2 として渡される。

通常は 0x0A の部分を変えるだけでよくて同じく sqrt（0ｘ08）を使うとき

```
short sqrt(int x)
//------------------------------------------------------------------------------
/*
平方根を計算する。
 */
{
        register int result;
        __asm ("SWI      0x08<<16¥nmov %0,r0¥n" : "=r"(result) :: "r1", "r2", "r3");
        return result;

}
```

上のようなものでよい。

ArcTan は参考ページにより、r0　0000h-FFFFh for 0<=THETA<2PI となっているので

戻り値に 0x10000 でわって　360 をかけることによって　角度が得られるようになる。

つまり 360 * aTan(x, y) / 0x10000　で角度になる。

BIOS 関数を使ったサンプル

```
#define ATOI_SIZE 15

void MappyPrint(char* s) {
        asm volatile("mov r0, %0;"
                     "swi 0xff0000;"
```

```
                        : // no ouput
                        : "r" (s)
                        : "r0");
}
static int aTan(int *x, int *y)
//--------------------------------------------------------------------------------
/*
 アークタンジェント（arc tangent）を計算する。
 */
{
        register int result;
        __asm ("SWI     0x0A<<16¥nmov %0,r0¥n" : "=r"(result) :: "r1", "r2", "r3");
        return result;

}

short sqrt(int x)
//--------------------------------------------------------------------------------
/*
 平方根を計算する。
 */
{
        register int result;
        __asm ("SWI     0x08<<16¥nmov %0,r0¥n" : "=r"(result) :: "r1", "r2", "r3");
        return result;

}

char* itoa(int k, char *st) {
        char* start = st;
        if (k == 0) {
                *st = '0';
                return st;
        }
        if (k < 0) {
                *st = '-';
                st++;
                k = -k;
```

```
        }
        int i = 1;
        while ((i *= 10) <= k) {
        }
        i /= 10;
        while (i > 0) {
                *st = '0' + (k / i);
                st++;
                k = k % i;
                i /= 10;
        }
        return start;
}
void trace(int k) {
        char str[ATOI_SIZE + 1] = { 0 };
        itoa(k, str);
        MappyPrint(str);
        MappyPrint("¥n");
}

int main() {
        int x = 100, y = 100;
        trace(x);
        trace(sqrt(x));
        trace(360 * aTan(x, y) / 0x10000);
        while (1)
                ;
}
```

実行結果