Advanced Lab 6

Interrupts, Timers, Sound and GBA BIOS

Andrew Wilder

Topics

- Interrupts
    - Handler
    - Timers
- Sound
    - Legacy
    - Streaming
- BIOS Functions

TONC

Most of the information in these slides is from TONC. If you want more information about any of the flags, definitions, or sample code, check it out:

http://www.coranac.com/tonc/text/

Interrupts

Polling:

- Teacher asks, “any questions?”
- waitForVBlank()
- Are we there yet?

  Are we there yet?

  Are we there yet?

Interrupt:

- Student raises their hand, teacher acknowledges
- Timers
- “Shut up, I’ll tell you when we get there”

Interrupts

Several events on the GBA can produce interrupts:

- VBlank (scan reaches line 160)
- HBlank (scan reaches row 240)
- VCount (scan reaches row 0)
- Timer overflows
- COM Port communication
- DMA completes
- Button presses
- Cartridge inserted/removed

Interrupts

Enable interrupts:

1. Point 0x3007FFC to your interrupt handler
2. Enable a type of interrupt (REG_IE)
3. Enable master interrupt flag (REG_IME)

Handle interrupts (in handler routine):

1. Disable interrupts (REG_IME)
2. Run the interrupt handler code
3. Enable interrupts (REG_IME)
4. Acknowledge interrupt (REG_IF), then return

Interrupts

Note:

Allowing the game to trigger an interrupt during the service of another interrupt is tricky, and writing a good handler to do so requires coding directly in ARM assembly, which is beyond the scope of this lab. If you wish to run complex systems which require nested interrupts to work, see TONC.

Interrupts

REG_IME:

Enable

FEDC BA98 7654 3210

Interrupts

REG_IE / REG_IF:

FEDC BA98 7654 3210

VBlank

HBlank

VCount

Timers (4)

COM Port

DMA Channels (4)

Keypad

Cartridge

Interrupts

```
#define REG_IE  *(volatile unsigned short*) 0x4000200
#define REG_IF  *(volatile unsigned short*) 0x4000202
#define REG_IME *(volatile unsigned short*) 0x4000208
#define IRQ_ENABLE    1
#define IRQ_VBLANK    (1 << 0)
#define IRQ_HBLANK    (1 << 1)
#define IRQ_VCOUNT    (1 << 2)
#define IRQ_TIMER(n)  (1 << (3 + (n))) /* 0, 1, 2 or 3 */
#define IRQ_COM       (1 << 7)
#define IRQ_DMA(n)    (1 << (8 + (n)))
#define IRQ_KEYPAD    (1 << 12)
#define IRQ_CARTRIDGE (1 << 13)
typedef void (*irqptr)(void);
#define REG_ISR_MAIN *(volatile irqptr*) 0x3007FFC
```

Interrupts

```
/* Set up interrupts */

void myHandler() {            // Handler should be void,
    …                         // with no args
}

void initInterrupts() {
    REG_ISR_MAIN = myHandler; // Point 0x3007FFC to handler

    REG_IE = IRQ_VBLANK       // Interrupt on VBlank and
           | IRQ_TIMER(0);    // Timer 0 overflow

    REG_IME = IRQ_ENABLE;     // Master interrupt enable
}
```

Interrupts

```
/* Basic (non-nested) interrupt handler */

REG_IME = 0;          // Disable interrupts in the handler
switch(REG_IF) {      // Switch on the interrupt raised
    case IRQ_VBLANK:
        …             // Code to execute on VBlank
        break;
    case IRQ_TIMER(0):
        …             // Code to execute on Timer 0 trigger
        break;
    default: break;
}
REG_IF = REG_IF;      // Acknowledge interrupt
REG_IME = IRQ_ENABLE; // Enable them again
```

Timers

Timers in the GBA are interrupt-based:

1. Configure the interrupt handler for the timer (REG_IE)
2. Set the number of ticks per trigger (REG_TMD)
3. Choose a timer, and set its frequency (REG_TMCNT)
4. Set it to raise an interrupt on overflow (REG_TMCNT)
5. Enable the timer (REG_TMCNT)

Timers

Frequency levels:

Level	Clock cycles	Frequency	Period
0	1	16.78 MHz	59.59 ns
1	64	262.21 kHz	3.815 µs
2	256	65.536 kHz	15.26 µs
3	1024	16.384 kHz	61.04 µs

Timers

REG_TMCNT:

FEDC BA98 7654 3210

Frequency Level (0-3)

Cascade Mode

IRQ Enable

Enable

Timers

REG_TMD:

This register should be set before enabling the timer. The way it works is that an event (IRQ, cascade) is triggered when the timer overflows. After that, it is reset to its initial value. Thus, if you want it to trigger after 100 ticks, set it to -100; it will increment each tick, and overflow after 100 ticks.

Timers

```
#define REG_TMD(n)    *(volatile unsigned short*) \
                      (0x4000100 + ((n) << 2))
#define REG_TMCNT(n) *(volatile unsigned short*) \
                      (0x4000102 + ((n) << 2))
#define TM_FREQ_1     0
#define TM_FREQ_64    1
#define TM_FREQ_256   2
#define TM_FREQ_1024 3
#define TM_CASCADE    (1 << 2)
#define TM_IRQ        (1 << 6)
#define TM_ENABLE     (1 << 7)
```

Sound

There are 2 types of sound on the GBA:

- Legacy sound

  Old Gameboy style of sound with two square channels and a noise channel

- Streaming sound

  Raw waveform read from a buffer as unsigned 8-bit PCM

Legacy Sound

Legacy sound consists of 4 channels:

1. Square with sweep
2. Square
3. Legacy waveform
4. Noise

Legacy waveform will not be covered because neither I nor TONC could figure it out. If you'd like to take a shot, see the in-depth GBA specification at:

http://problemkaputt.de/gbatek.htm#gbasoundchannel3waveoutput

Legacy Sound

Setting up legacy sound:

1. Set master sound enable (REG_SNDSTAT)
2. Set master volume (REG_SNDDSCNT)
3. Set L/R speaker volume (REG_SNDDMGCNT)
4. Enable L/R channels (REG_SNDDMGCNT)

Playing a sound:

1. Write control data (and sweep if channel 1)
2. Write frequency data (do this last)

Legacy Sound

Note:

For completeness, I have included the specification of the sound registers; however, you do not need to do these calculations by hand if you want to use legacy sounds in your game. I made **SquareDemo.gba** and **NoiseDemo.gba**, available under Resources > Game Share, which you can use to test and generate the composite values to write to the sound registers. You only need to construct a framework to play the sounds.

Legacy Sound

REG_SNDSTAT:

DMA channels playing (4)

Master sound enable

FEDC BA98 7654 3210

Legacy Sound

REG_SNDDSCNT:

FEDC BA98 7654 3210

DMG volume ratio (0-2)

DSA Volume (50%, 100%)

DSB Volume (50%, 100%)

DSA enable, left speaker

DSA enable, right speaker

Use TM0 / TM1 for DSA

DSA FIFO reset

DSB flags (same as 8 - B)

Legacy Sound

REG_SNDDMGCNT:

FEDC BA98 7654 3210

Left volume (0-7)

Right volume (0-7)

Left channels enable (4)

Right channels enable (4)

Legacy Sound

REG_SND1CNT and REG_SND2CNT:

FEDC BA98 7654 3210

Length (0-63)

Duty cycle (⅛, ¼, ½, ¾)

Envelope step time (0-7)

Envelope direction (dec/inc)

Envelope initial value (0-15)

Legacy Sound

REG_SND1FREQ and REG_SND2FREQ:

FEDC BA98 7654 3210

Frequency rate

Timed envelope flag

Play sound flag (reset)

Legacy Sound

REG_SND1SWEEP:

FEDC BA98 7654 3210

Sweep number (0-7)

Sweep direction (up/down)

Sweep speed (0-7)

Legacy Sound

REG_SND4CNT:

FEDC BA98 7654 3210

Sound length (0-63)

Envelope step time (0-7)

Envelope direction (dec/inc)

Envelope initial value (0-15)

Legacy Sound

REG_SND4FREQ:

FEDC BA98 7654 3210

Frequency dividing ratio

Counter width (15/7 bits)

Shift clock frequency (0-15)

Timed envelope flag

Play sound flag (reset)

Legacy Sound

```
#define REG_SNDDMGCNT *(volatile unsigned short*) 0x4000080
#define REG_SNDDSCNT  *(volatile unsigned short*) 0x4000082
#define REG_SNDSTAT   *(volatile unsigned short*) 0x4000084
#define REG_SND1SWP   *(volatile unsigned short*) 0x4000060
#define REG_SND1CNT   *(volatile unsigned short*) 0x4000062
#define REG_SND1FRQ   *(volatile unsigned short*) 0x4000064
#define REG_SND2CNT   *(volatile unsigned short*) 0x4000068
#define REG_SND2FRQ   *(volatile unsigned short*) 0x400006C
#define REG_SND4CNT   *(volatile unsigned short*) 0x4000078
#define REG_SND4FRQ   *(volatile unsigned short*) 0x400007C
#define MASTER_SND_EN (1 << 7)
#define DMG_VOL_25    0
#define DMG_VOL_50    1
#define DMG_VOL_100   2
```

Legacy Sound

```
#define LEFT_VOL(n)  (n)
#define RIGHT_VOL(n) ((n) << 4)
#define CHAN_EN_L(n) (1 << ((n) + 7))  /* n is 1-based! */
#define CHAN_EN_R(n) (1 << ((n) + 11))
#define DSA_VOL_50   (0 << 2)
#define DSA_VOL_100  (1 << 2)
#define DSB_VOL_50   (0 << 3)
#define DSB_VOL_100  (1 << 3)
#define DSA_EN_R     (1 << 8)
#define DSA_EN_L     (1 << 9)
#define DSA_TM_1     (1 << 10)
#define DSA_RST_FIFO (1 << 11)
#define DSB_EN_R     (1 << 12)
#define DSB_EN_L     (1 << 13)
#define DSB_TM_1     (1 << 14)
#define DSB_RST_FIFO (1 << 15)
```

Legacy Sound

Try building a legacy midi player! Just create a framework for writing SWP, CNT and FRQ values to the sound registers on timed intervals.

Streaming Sound

The GBA is also capable of streaming sound; the way it’s done is by using DMA to copy unsigned 8-bit PCM into a FIFO buffer, and trigger an interrupt to refill it when that buffer is empty. Time the completion of the music with the song length, by interrupting on and counting VBlanks.

You can also do stereo by simultaneously using the GBA’s 2 channels for streaming (DSA and DSB).

Streaming Sound

Set up streaming sound:

1. Convert sound to mono, 8-bit unsigned PCM WAV
2. Convert WAV with wav2c (Resources > GBA)
3. Set up interrupts and master sound flags
4. Configure DSA/DSB flags in REG_SNDDSCNT
5. Calculate number of cycles per sample for song len
6. Configure DMA (SRC = array, DST = FIFO buffer)
7. Interrupt on VBlank, counting VBlanks until the end of the song; terminate or loop music at end.

Streaming Sound

```
/* DMA and DISPSTAT defines, in case you don’t have them */
#define REG_DMASRC(n)         *(volatile unsigned int*) \
                              (0x40000B0 + ((n) * 12)
#define REG_DMADST(n)         *(volatile unsigned int*) \
                              (0x40000B4 + ((n) * 12)
#define REG_DMACNT(n)         *(volatile unsigned int*) \
                              (0x40000B8 + ((n) * 12)
#define DMA_REPEAT            (1 << 25)
#define DMA_32                (1 << 26)
#define START_ON_FIFO_EMPTY (3 << 28)
#define DMA_ON                (1 << 31)
#define REG_DISPSTAT          *(volatile unsigned short*) \
                              0x4000004
#define INT_VBLANK_ENABLE     (1 << 3)
```

Streaming Sound

```
REG_SNDSTAT = MASTER_SND_EN; /* Basic steps necessary for streaming sound */
REG_SNDDSCNT = DMG_VOL_100
             | DSA_EN_L
             | DSA_EN_R
             | DSA_RST_FIFO;
vblankcnt = (int)(59.77 * SONGLEN / SONGFREQ); // File-level variable
REG_TMD(0) = -((1 << 24) / SONGFREQ);
REG_TMCNT(0) = TM_ENABLE
             | TM_FREQ_1;
REG_DMASRC(1) = (int) songData;
REG_DMADST(1) = (int) REG_FIFO_A;
REG_DMACNT(1) = DMA_ON
              | START_ON_FIFO_EMPTY
              | DMA_32
              | DMA_REPEAT;
REG_IE |= IRQ_VBLANK;
REG_DISPSTAT |= INT_VBLANK_ENABLE;
REG_IME |= IRQ_ENABLE;
```

```
void vblankHandler() {
    REG_IME &= ~IRQ_ENABLE;
    if(!--vblankcnt) {
        REG_TMD(0) = 0;
        REG_TMCNT(0) = 0;
        initMusic(); // loop
    }
    REG_IF = REG_IF;
    REG_IME |= IRQ_ENABLE;
}
```

GBA BIOS

The GBA supports traps, in the form of software interrupts that call code in the GBA's BIOS. There are 42 such calls available, of which a small handful may be useful for general GBA games. These are called with the swi instruction, which can only be put into the code via assembly.

GBA BIOS

A few useful BIOS functions:

- 0x05: VBlankIntrWait
- 0x08: Sqrt
- 0x0A: Arctan

Full list: http://www.coranac.com/tonc/text/swi.htm

GBA BIOS

VBlankIntrWait:

swi 0x05

(no args)

return: void

Pauses the CPU until VBlank; saves significant power over waitForVBlank() when run on an actual GBA!

GBA BIOS

Sqrt:

swi 0x08

r0: n (unsigned int)

return: √(n)

Perform an integer square root.

GBA BIOS

Arctan:

- swi 0x0A
- r0: x (short)
- r1: y (short)
- return: arctan(y/x), from 0 (zero) to 0xFFFF (~2*pi)

Calculate the arctangent of the ratio of two numbers; very useful when you need to determine angles from ratios, such as in coordinate transformations.

GBA BIOS

```
/* asmfunc.s */
    .text
    .code 16
    .algin 2

    .global sqrt_
    .thumb_func
sqrt_:
    swi 0x08
    bx lr

    .global arctan
    .thumb_func
arctan:
    swi 0x0A
    bx lr
```

```
/* asmfunc.h */

// Because of a conflict with the math
// library’s sqrt() function, we have
// to name it something else, like “sqrt_”
#define sqrt(x) sqrt_(x)

extern unsigned short sqrt_(unsigned int x);
extern short arctan(short x, short y);
```

```
/* main.c */
#include “asmfunc.h”
…
int a = sqrt(64);  // Stores 8 in a
```