Systems Programming

TP 5 –Sprites – Interruptions – Timers

Sprite – What is it?

- 2D Image made out of pixels!
- For the GBA, sprites sizes that are supported are
  - 8x8 pixels
  - 16x16 pixels
  - 32x32 pixels
  - 64x64 pixels
- Sprites are hardware accelerated thus really fast
- Animations can be created using sprites

Mario in Super Mario bros in the Nintendo Entertainment System was a 32x32 pixel sprite.

How to manage the sprites?

Three memory addresses:

- SpriteMEM: Information about the sprite.
- SpriteData: The pixels of the sprite.
- SpritePal: The color palette for the sprite.

Graphics mode for sprites – Tile Modes

- Mode 0:
  - Support for 4 backgrounds (bg0, bg1, bg2, bg3)
  - No scaling / rotation
- Mode 1:
  - Support of 3 backgrounds (bg0, bg1, bg2)
  - Scaling and rotation supported only in bg2
- Mode 2:
  - Support of 2 backgrounds (bg2, bg3)
  - Scaling and rotation supported on bg2 and bg3

Time to get your hands dirty…

- You can get away with using C for this TP.
- We will be using mode 2 for this TP

```
*(unsigned short*) 0x4000000 = 2 | 0x1000 | 0x40;
```

Enable sprites

Use a specific sprite layout map

Sprite Palette

- To access it, access the 0x5000200 address
- We will be using up to 256 different colors
- You can have 16 palettes of 16 colors or one palette with 256 colors. We will use the latter.

```
#define SpritePal ((unsigned short*)0x5000200)
```

Sprite Data

- To access it, access the 0x6010000 address
- This is where the sprite pixels are placed one after the other.
- IMPORTANT! Depending on the color depth used (see palette):
  - A byte contains 2 pixels if the color depth is 4bit (16 colors)
  - A byte contains 1 pixel if the color depth is 8bit (256 colors)

```
#define SpriteData ((unsigned short*)0x6010000)
```

Sprite Mem

- To access it, access the 0x7000000 address
- Contains information about:
  - Position of the sprite
  - Size
  - The used Palette
- It is possible to display up to 128 sprites in the GBA

```
#define SpriteMem ((unsigned short*)0x7000000)
```

SpriteMem – Whats happening in there?

- More or less it can be considered as an 128 entry array, where each entry contains four attributes of 16 bits each.
- GBA book has a really nice example where he uses this struct (check chapter 7):

```
typedef struct tagSprite
{
   unsigned short attribute0;
   unsigned short attribute1;
   unsigned short attribute2;
   unsigned short attribute3;
}Sprite
```

SpriteMem – Attribute 0

- Bit 15-14: Shape of the sprite:
  - 00 – Square
  - 01 – Horizontal rectangle
  - 10 – Vertical rectangle
  - 11 – not used
- Bit 13: palette type: 0 for 16 colors, 1 for 256 colors
- Bit 12: Mosaic effect: leave 0
- Bit 11-10: Transparency: leave 0
- Bit 9-8: Transformation: leave 0
- Bit 7-0: Y coordinate

SpriteMem – Attribute 1

- Bit 15-14: Size of the sprite – depending on shape:

Shape	Size	Dimensions
00	00	8x8
	01	16x16
	10	32x32
	11	64x64
01	00	16x8
	01	32x8
	10	32x16
	11	64x32
10	00	8x16
	01	8x32
	10	16x32
	11	32x64
11	-	-

- Bit 13-09: Transformation: leave 0
- Bit 13: Vertical flip : leave 0
- Bit 12: Horizontal flip : leave 0
- Bit 8-0: X coordinate

SpriteMem – Attribute 2

- Bit 15-12: The palette used (or not used if a single palette)
  - Remember if we are using single palette of 256 colors then we leave this a 0 (otherwise we specify the 16 color palette of the 16 palettes available…)
- Bit 11-10: Priority based on backgrounds: leave it 0
- Bit 9-0: offset sprite memory

Interrupts

What is an interrupt??

- A signal to the processor to stop what he is doing save all his registers and do something “else”.

- Or according to wiki:
  - A *hardware interrupt* causes the processor to save its state of execution and begin execution of an interrupt handler.
  - Software interrupts are usually implemented as instructions in the instruction set, which cause a context switch to an interrupt handler similar to a hardware interrupt.

Addresses you need

Interrupts are controlled through some special addresses:

- REG_IME = 0x4000208:The master interrupt register. Set to 1 if interrupts are enabled, 0 otherwise.
- REG_IE = 0x4000200: Informs GBA that a specific interrupt is used. (check the book at chapter 8 for the list of each bit). It's where u SET things…
- REG_IF = 0x4000202: Clone of REG_IE but… only one bit enabled each time based on the interrupt. Used for the callback function (or Interrupt Service Routine or ISR)
  REG_INTERRUPT = 0x3007FFC: address containing
  the address of the ISR

All are 16-bit except REG_INTERRUPT

List of interrupts

Bit	Description	Define MASKS
0	VB – Vertical Blank Interrupt	#define INT_VBLANK 0x0001
1	HB – Horizontal Blank Interrupt	#define INT_HBLANK 0x0002
2	VC – Vertical Scanline Interrupt	#define INT_VCOUNT 0x0004
3	T0 – Timer 0 interrupt	#define INT_TIMER0 0x0008
4	T1 – Timer 1interrupt	#define INT_TIMER1 0x0010
5	T2 – Timer 2 interrupt	#define INT_TIMER2 0x0020
6	T3 – Timer 3 interrupt	#define INT_TIMER3 0x0040
7	COM – Serial Communication interrupt	#define INT_COM 0x0080
8	DMA0 – DMA0 Finished Interrupt	#define INT_DMA0 0x0100
9	DMA1 – DMA1 Finished Interrupt	#define INT_DMA1 0x0200
10	DMA2 – DMA2 Finished Interrupt	#define INT_DMA2 0x0400
11	DMA3 – DMA3 Finished Interrupt	#define INT_DMA3 0x0800
12	BUTTON – Button Interrupt	#define INT_BUTTON 0x1000
13	CART – Game cartridge interrupt	#define INT_CART 0x2000
14-15	Unknown / unused	

How does these work?!?

1. Interrupt is triggered
2. CPU saves it’s state of all registers
3. Code execution is branched to the 0x3007FFC address
4. In that address you will put you own function / routine
5. We call that routine interrupt service routine – ISR
6. The ISR will handle all types of interrupts you want to support
7. ...Meaning that you will be checking which interrupt was triggered.

ISR or Interrupt Handler

```
void MyHandler()
{
    REG_IME = 0x00; //disable interrupts
    REG_IF_BACKUP = REG_IF; //backup the REG_IF
    if ( REG_IF & MASK1 )  // handling all the interrupts we are supporting with our handler
    {/*do stuff*/ }
    Else if ( REG_IF & MASK2 )
    {/*do stuff*/ }
    …
    Else if ( REG_IF && MASKX )
    { /*do stuff*/ }
    REG_IF = REG_IF_BACKUP;        //restoring the REG_IF will inform the CPU that the interrupt was
    indeed handled…
    // otherwise the handler will be executed again for the same interrupt
    //EXTRA CARE HERE…. THIS COMMAND MEANS NOTHING FOR A COMPILER,
    //THUS IT WILL BE OMMITED FOR OPTIMIZATIONS THUS… BAD THING. For
    //this reason we have to define REG_IF as VOLATILE in C.
    REG_IME = 0x01;     //enable interrupts again
}
```

Timers – Is this TP going to end?!?

- A timer is a counter which is 16-bit and is incremented at specific intervals.
- There are 4 timers in the GBA and they can be used separately.
- The timers are based on the system clock and thus they fit into specific frequencies.

Value	Frequency	Duration
00	16.78MHz clock	59.595 nanoseconds interval
01	64 cycles	7.6281 microseconds interval
10	256	15.256 microseconds interval
11	1024	61.025 microseconds interval

Timers – How to access them

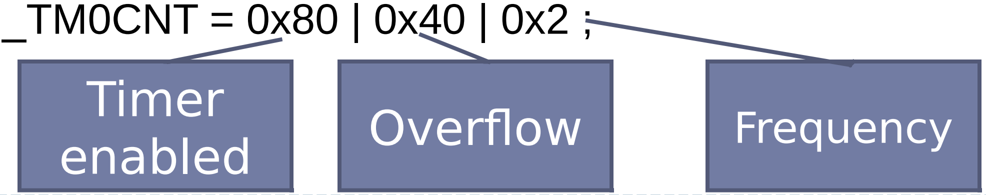
```
//useful defines for the frequencies available
    #define TIMER_FREQUENCY_SYSTEM 0x0
    #define TIMER_FREQUENCY_64 0x1
    #define TIMER_FREQUENCY_256 0x2
    #define TIMER_FREQUENCY_1024 0x3
//This is where you will be initializing the timers setting the status bits
    #define REG_TM0CNT *(volatile u16*)0x4000102
    #define REG_TM1CNT *(volatile u16*)0x4000106
    #define REG_TM2CNT *(volatile u16*)0x400010A
    #define REG_TM3CNT *(volatile u16*)0x400010E
//This is where you will be reading your timers from
    //#define REG_TM0D *(volatile u16*)0x4000100
    #define REG_TM1D *(volatile u16*)0x4000104
    #define REG_TM2D *(volatile u16*)0x4000108
    #define REG_TM3D *(volatile u16*)0x400010C
```

What to tinker from a REG_TMxCNT

Bit	Description
0-1	Frequency
2	Overflow from previous timer
3-5	Not used
6	Overflow generates interrupt
7	Timer enable
8-15	Not used

Example of setting a REG_TMxCNT:

REG_TM0CNT = 0x80 | 0x40 | 0x2 ;



Timer enabled

Overflow

Frequency

What to READ THOROUGHLY SERIOUSLY!!1!!1111!

GBA BOOK CHAPTER 7 AND 8!