CS 2261: Media Device Architecture - Week 11

Overview

- Candy
- Interrupts
- Function Pointers
- Timers
- Sound Demo Using timers (details next class)

Interrupts

- An interrupt causes the CPU to stop what it is currently doing and execute another piece of code (process the interrupt)
- There are two kinds of interrupts
  - Software interrupts – used to call low-level code that is typically provided by the OS or BIOS – called “software interrupt” because the interrupt is initiated in software. These are also sometimes called traps or system calls.
  - Hardware interrupts – triggered by hardware events – used to let the CPU respond to asynchronous hardware events
- We’re only looking at hardware interrupts

Processing Interrupts

- The CPU saves the current value of registers
- The CPU starts executing code at a special location, the interrupt entry point
- When the interrupt code is finished, the CPU restores the registers and goes back to the code it was in the middle of working on when it was interrupted
- We'll only worry about processing one interrupt at a time (not processing interrupts while we're interrupted), because we disable interrupts while one is being handled.

GBA Interrupts

- You use `REG_IE` (interrupt enable register) - `0x04000200` to to tell the GBA which events you want to be interrupted by. The you read the bits of `REG_IF` - `0x04000202` (the interrupt fired register), to see which fired.

Bit	Description	Note
0	VB – vertical blank interrupt	Also requires REG_DISPSTAT bit 3
1	HB – horizontal blank interrupt	Also requires REG_DISPSTAT bit 4
2	VC – vertical scanline count interrupt	Also requires REG_DISPSTAT bit 5
3	T0 – timer 0 interrupt	Also requires REG_TM0CNT bit 6
4	T1 – timer 1 interrupt	Also requires REG_TM1CNT bit 6
5	T2 – timer 2 interrupt	Also requires REG_TM2CNT bit 6
6	T3 – timer 3 interrupt	Also requires REG_TM3CNT bit 6
7	COM – serial communication interrupt	// we're not using this at all
8	DMA0 – DMA0 finished interrupt	Also requires REG_DMA0CNT bit 30
9	DMA1 – DMA1 finished interrupt	Also requires REG_DMA1CNT bit 30
10	DMA2 – DMA2 finished interrupt	Also requires REG_DMA2CNT bit 30
11	DMA3 – DMA3 finished interrupt	Also requires REG_DMA3CNT bit 30
12	KEYPAD – keypad interrupt	Also requires REG_KEYCNT bit 14
13	CART – game cartridge interrupt	Raise when cartridge is REMOVED!
14-15	unused	

Steps for setting up an interrupt

1. Disable all interrupts
2. Set the interrupts you want to pay attention to
3. Set the subordinate interrupt control register for your specific interrupts
4. Set up the entry point for your interrupt handling routine
5. Re-enable interrupts

REG_IME

- The interrupt master enable register – the master switch for all interrupts
- When its value is 0, all interrupts are turned off
- When its value is 1, interrupts can happen (but only the ones you've specifically enabled)
- `#define REG_IME *(u16*)0x4000208`

REG_IE

- The interrupt enable register
- Set specific bits in this register to enable specific interrupts
- The meaning of the different bits is defined a couple of slides before
- `#define REG_IE *(u16*)0x4000200`

Specific interrupt control registers

- Each specific interrupt type has its own control register
- Display interrupts (VB, HB, VC) use the DISPSTAT (display status) register
- Each DMA channel as a REG_DMAxCNT (e.g. DMA0 control) register
- Each timer has a REG_TMxCNT (e.g. timer 0 control) register
- There are also control registers for the button (REG_KEYCNT) and serial communication interrupts

REG_DISPSTAT

- The REG_DISPSTAT register is used both to control display related interrupts and indicates display status

Bit	Description
0	VB – vertical blank is occurring
1	HB – horizontal blank is occurring
2	VC – vertical count reached
3	VBE – enables vblank interrupt
4	HBE – enables hblank interrupt
5	VCE – enables vcount interrupt
6-15	VCOUNT – vertical count value (0-159)

Interrupt Handler

- When an interrupt happens, the GBA starts executing the code whose address is stored at location 0x03007FFC (REG_INTERRUPT)
- That means that, 0x03007FFC is the location of a *pointer* that points to *code*

```
#define REG_INTERRUPT *(u32*)0x03007FFC
REG_INTERRUPT = (u32)interruptHandler;
```

Point to code?: Function Pointers

In C symbols often represent addresses

```
int a;
short b[10];
struct {
    int x;
    int y;
} point1;
// etc.
```

Function Pointers

```
int someFunction(int arg1, int arg2)
{
  return arg1 + arg2;
}
```

What is "someFunction"?

Function Pointers

- In C the name of a function is a symbol whose value is the memory address.
- We are "cheating" when we say:

```
#define REG_INTERRUPT *(u32*)0x03007FFC
REG_INTERRUPT = (u32)interruptHandler;
```

- It works and for our purposes, but if you are going to be a C programmer you should know there is a right way™.

Function Pointer Syntax

Want to store the address of a variable we use a pointer
(e.g. `int *ip;`)

- Same is true for holding the address of a function

```
int fi(void);   /* Function that returns an int */
int *fpi(void); /* Function that returns a pointer to an int */
int* fpi(void); /* What compiler sees */

int (*pfi)(void); /* Declaring pfi to be a pointer to a
                     function! */
```

More Syntax Examples

```
// General Function Pointer Syntax:
<return-value>(*variablename)(<parameter-list>)

// Examples:
int (*fp1) (int, float);
void (*fp2) (double, double);
int * (*fp3) (int *, char);

void (*fp4) ();
void (*fp4) (void); // adding void tells the compiler NO
params are allowed when calling the function; otherwise,
some compilers would allow calls without complaint
```

Function Pointers

```
// Let's use a typedef to make things readable
typedef void (*ihp_t)(void);

#define REG_INTERRUPT *((ihp_t *)0x03007FFC)
REG_INTERRUPT = interruptHandler;
```

C Interrupt Handler Function How-to

- Disable interrupts
- Check which device interrupted
  - look at REG_IF
- Handle the interrupt given the source
- Notify GBA that interrupt has been handled
- Re-Allow interrupts
- Function returns and processor goes back to normal execution
- No…
  - return values
  - parameters
  - need to worry about figuring out how to go back to the code/state you were at before the interrupt (C/the GBA handles this for you)

Useful Constants

```
//primary interrupt locations
#define REG_IME *(u16*)0x4000208
#define REG_IE *(u16*)0x4000200
#define REG_IF *(volatile u16*)0x4000202
#define REG_INTERRUPT *(u32*)0x3007FFC
#define REG_DISPSTAT *(u16*)0x4000004

//interrupt constants for turning them on
#define INT_VBLANK_ENABLE 1 << 3

//interrupt constants for checking which type of interrupt
  happened
#define INT_VB 1 << 0
#define INT_BUTTON 1 << 12
```

```
//interrupt constants for checking which type of interrupt happened
#define INT_VB     1 <<  0 // VB - vertical blank interrupt
#define INT_HB     1 <<  1  // HB - horizontal blank interrupt
#define INT_VC     1 <<  2 // VC - vertical scanline count interrupt
#define INT_T0     1 <<  3 // T0 - timer 0 interrupt
#define INT_T1     1 <<  4 // T1 - timer 1 interrupt
#define INT_T2     1 <<  5 // T2 - timer 2 interrupt
#define INT_T3     1 <<  6 // T3 - timer 3 interrupt
#define INT_COM    1 <<  7 // COM - serial communication interrupt
#define INT_DMA0   1 <<  8 // DMA0 - DMA0 finished interrupt
#define INT_DMA1   1 <<  9 // DMA1 - DMA1 finished interrupt
#define INT_DMA2   1 << 10 // DMA2 - DMA2 finished interrupt
#define INT_DMA3   1 << 11 // DMA3 - DMA3 finished interrupt
#define INT_BUTTON 1 << 12 // BUTTON - button interrupt
#define INT_CART   1 << 13 // CART - game cartridge interrupt
```

Setup

```
void setupInterrupts(void) {
  REG_IME = 0; //disable interrupts
  REG_INTERRUPT = (u32)interruptHandler; //set int handler
  enableVBlankInterrupt();
  REG_IME = 1; //enable interrupts
}
void enableVBlankInterrupt() {
 REG_IE |= INT_VB; // Enable vblank interrupt detection
 REG_DISPSTAT |= INT_VBLANK_ENABLE; // Make display generate
                                    // vblank interrupts
}
```

Interrupts for a Precise Update Rate

```
void interruptHandler(void) {
  REG_IME = 0; //disable interrupts
  // Check which event happened, and do something if
  // you care about it
  if (REG_IF == INT_VBLANK) {
    // A vblank happened, call appropriate response
    // function (one update)
    gameLoop();
  }
  REG_IF = REG_IF;  // Tell GBA that interrupt has
                    // been handled
  REG_IME = 1;      //enable interrupts
}
```

MyHandler (draw pixel on vertical blank interrupt)

```
void MyHandler() {
   u16 x, y; // declare 2 unsigned shorts for x,y location

   REG_IME = 0x00;              //disable interrupts

   sprintf(str, "%d", i++);    // Print out count of MyHandler calls
   Print(0, 0, str, 0xFFFF);

   if(REG_IF == INT_VBLANK)  { //look for vertical refresh interrupt
       x = rand() % 240;         //draw a random pixel
           y = rand() % 160;
       DrawPixel3(x, y, RGB(rand()%31, rand()%31, rand()%31));
   }

   REG_IF = REG_IF; // What's this line for?

   //enable interrupts
   REG_IME = 0x01;
}
```

REG_IF

- REG_IF has a bit set for each of the interrupts that have occurred and have not yet been handled
- You tell the hardware that you've handled an interrupt by setting the bit in REG_IF corresponding to the interrupt you've handled
- Assuming there's only one outstanding interrupt (we won't let ourselves be interrupted while handling an interrupt), this is equivalent to setting REG_IF to itself!
  - From the point of view of changing variable values, a = a does nothing
    - Make sure it's volatile, since this looks very odd to the compiler!
  - Setting REG_IF = REG_IF writes to REG_IF which clears the bit acknowledging the interrupt has been handled
  - **If you don't set REG_IF to itself, the interrupt handler will be called endlessly for the same event**

Food for Thought

- How would we use the button interrupt to look at button values?
  - Input via interrupts instead of polling.
- How could we use the vblank interrupt with page flipping?
- How could we use the hblank interrupt to display more than 256 colors on the screen at a time in mode 4?
  - What are the limitations of this approach?

Timers

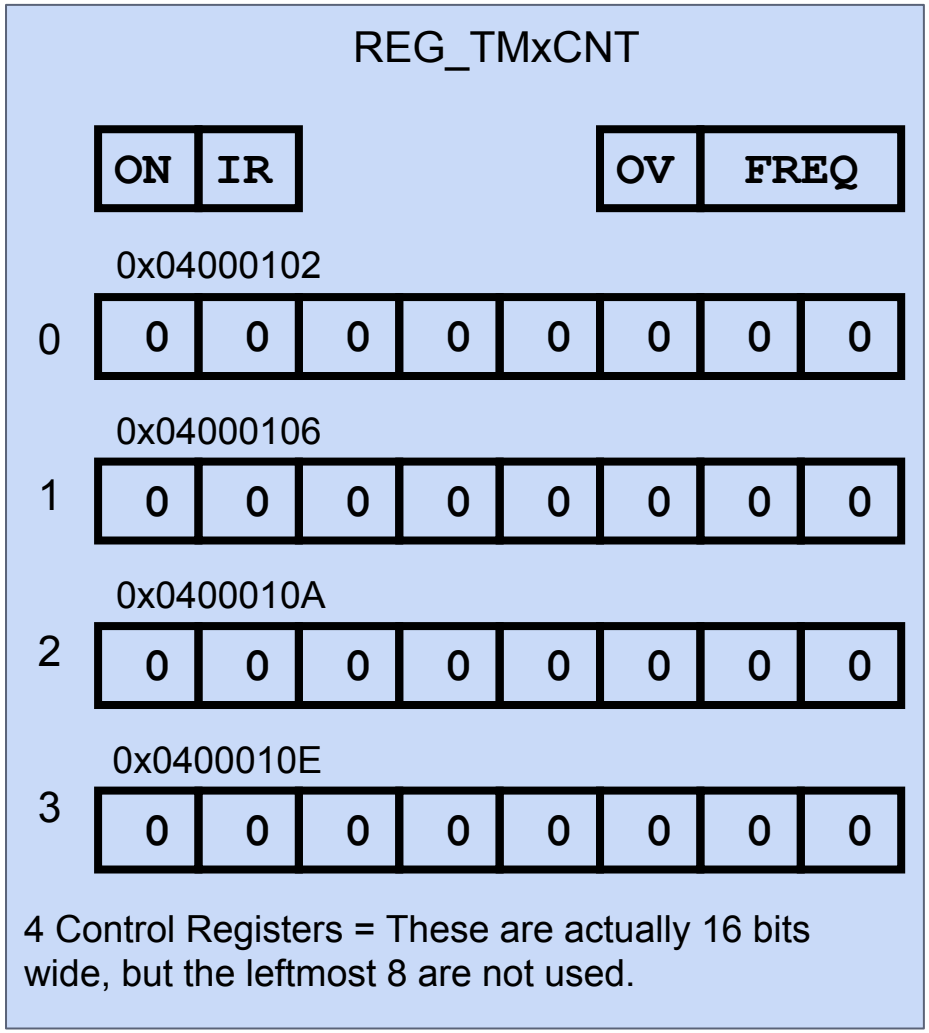
Common Practice

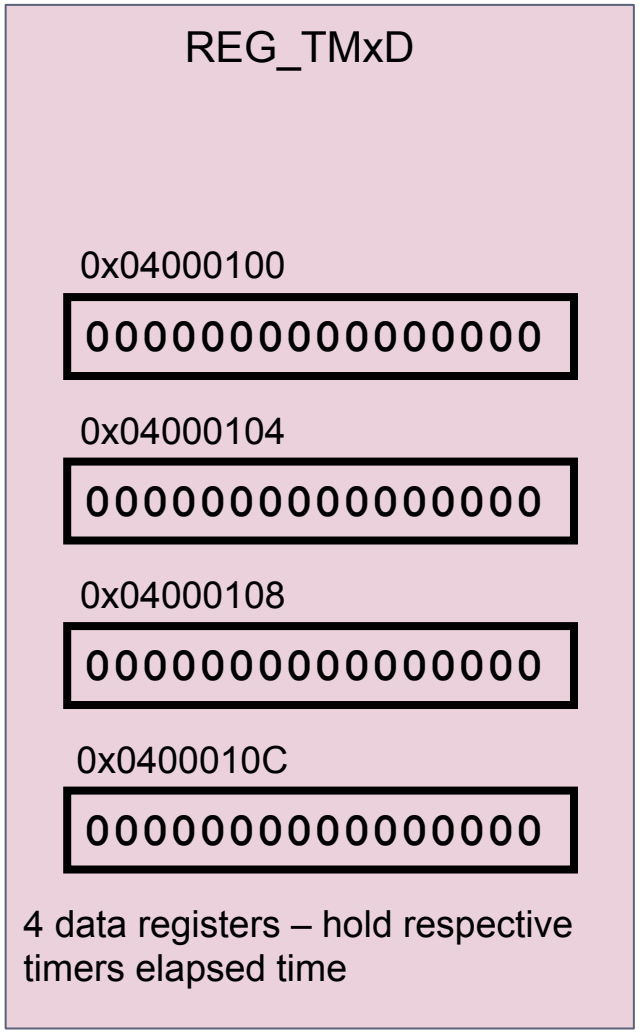
- Game timing is normally synchronized with the vertical blank.
- For certain applications such as precise DMA transfers involving sound, timers are used.
- Timers are also sometimes used in profiling studies, that is, determining how much time is spent in executing different portions of code.

GBA Timers

- Trigger an event after a duration, or keep track of the elapsed time since started
- 4 timers available
  - Each is a 16-bit register
  - Can be chained together to create even longer timers
- 4 different speeds/frequencies
  - When ticks should happen
  - Based on processor clock (more later)
  - Each timer can use any frequency

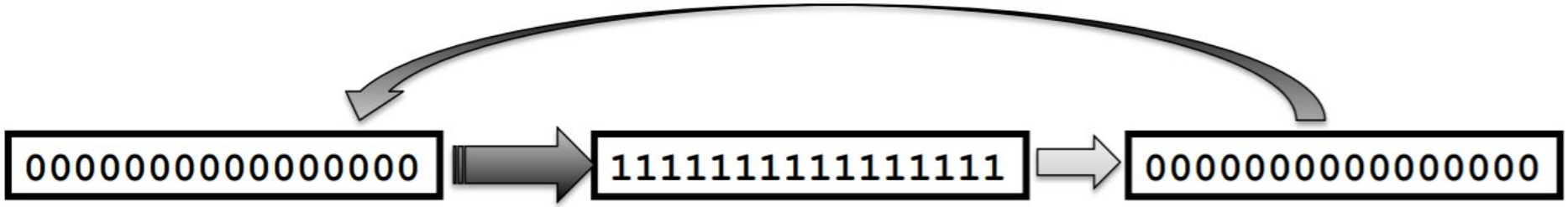
Timer Layout





Timer Registers

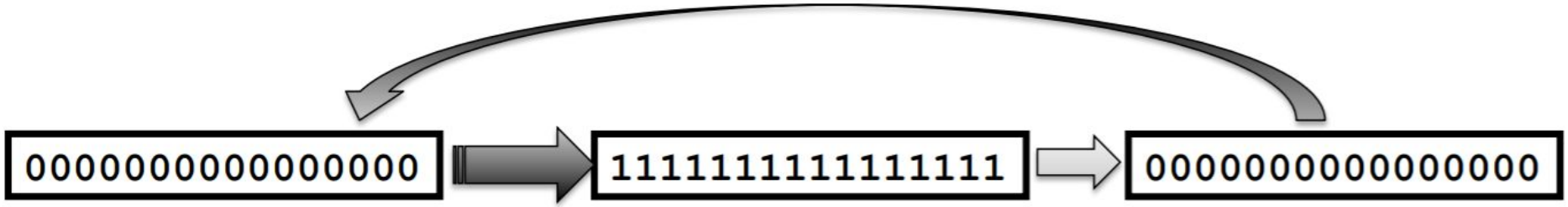
- Timer register holds number of elapsed timer ticks
  - 16 bits
  - Counts up until it hits all zeros (overflows)
    - Then resets to initial value



- How many ticks max?

Timer Registers (REG_TMxD)

- Timer register holds number of elapsed timer ticks
  - 16 bits
  - Counts up until it hits all zeros (overflows)
    - Then resets to initial value



- How many ticks max?
  - 0 -> 65535, then one more for back to 0 (so 2^16 total)
- To find the value of a timer, check the number stored in REG_TMxD
- Set REG_TMxD to provide an initial value (which resets every time it hits the end)

What If 2^16 ticks isn't enough?

- 16 bits is the limit in terms of ticks of a timer
- Suppose you want more ticks at a certain frequency than 16 bits can hold?
  - Chain timers together to create longer timers
  - e.g.
    - Chain two if you need a 32 (vs. 16) bit timer
    - Once one reaches its limit, start the next one
- Max is 64 (16 * 4) bit timer

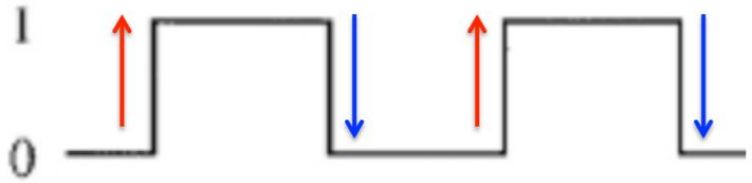
REG_TMxCNT

F E D C B A 9 8	7	6	5 4 3	2	1 0
-	En	I	-	CM	Fr

- Bits 0-1: Frequency
  - How often should it tick -- See next slides for details
- 2: Toggle overflow from previous timer ("Cascade Mode")
  - Causes this timer to tick when the previous timer (N-1)overflows
    - Means nothing for timer 0
  - `#define TM_CASCADE (1<<2)`
- 6: Generate interrupt when timer register overflows
  - `#define TM_IRQ (1<<6)`
    - (must also set bit in the interrupt enable register for timer)
- 7: Enable timer
  - `#define TM_ON (1<<7)`

Frequency?

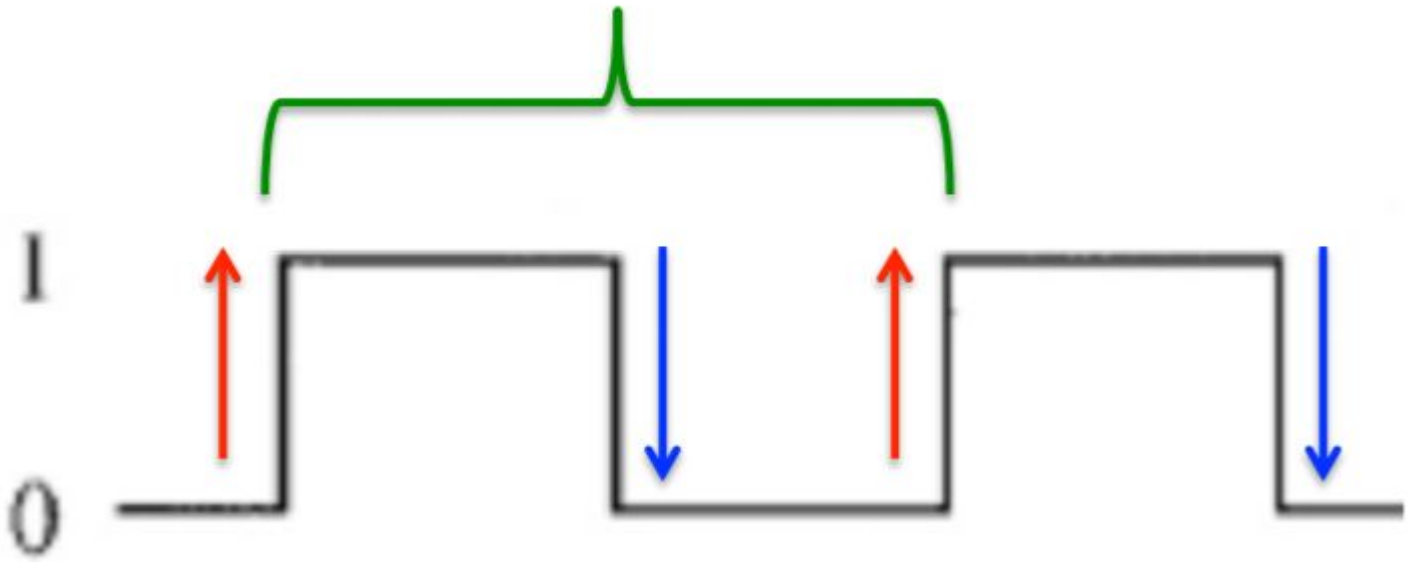
- Inner-workings of a processor are synchronized by an internal oscillator (or "clock").
- What's an oscillator?



  - Something that "oscillates" (alternates) between two opposite states at a fixed frequency
  - In electronic terms, it produces a sine wave or a square wave of voltage.
    - They convert DC current (AA batteries) to AC current.
    - Typically made using quartz crystal
    - They provide the base "system clock" used to keep everything in sync within a digital computer.

Clock Cycle (inverse of Frequency)

- Wavelength (in time) between two pulses:



Frequency is Measured in Hz (1/s)

- More pulses per second generally means faster processing
  - Though they can be very misleading as well, since some processors can do way more with each clock cycle than others.
- Speed is usually represented by clock cycles per second -- in megahertz (MHz) or gigahertz (GHz).
- A 1 MHz process performs 1,000,000 clock cycles per second.
- A 5 GHz processor performs 5,000,000,000

GBA vs Phone vs Core i7

- GBA Clock Speed:
  - 16,777,216 Hz (16.78 MHz)
  - (single core)
- Galaxy S9 (Snapdragon 845):
  - Up to 2.8 GHz
  - (8 cores)
- Core i7-9700K:
  - 3.60 - 4.90 GHz (base vs "turbo")
  - (8 cores)

WOW, phones are getting FAST!

Available GBA Timer Frequencies

Bits 0-1	# CPU cycles	Frequency	Period
00	1	2^24 Hz or 16,777,216 Hz	59.604ns
01	64	2^18 Hz or 262,144 Hz	3.811µs
10	256	2^16 Hz or 65,536 Hz	15.259µs
11	1024	2^14 Hz or 16,384 Hz	59.382µs

GBA Timer Frequencies

- Timer frequencies are distinct from the four timers
  - Any of the four timers can use any of the four frequencies
- In principle, by combining timers, almost any frequency can be generated on the GBA (at least up to the max given by the CPU itself (16.78 MHz)

Timer Frequency Selection

```
#define TM_FREQ_1 0
#define TM_FREQ_64 1
#define TM_FREQ_256 2
#define TM_FREQ_1024 3
```

Using Timers for Polling and/or Interrupts

- **Polling** - start a timer, then check the value in the timer register after doing some stuff
  - Find out how long a function takes to execute
  - Useful for development and optimization

- **Interrupts** – you are alerted by interrupt event whenever a certain amount of time has passed
  - Good for doing tasks at a precise time or repeating a task at even intervals

Setting Initial Timer Values

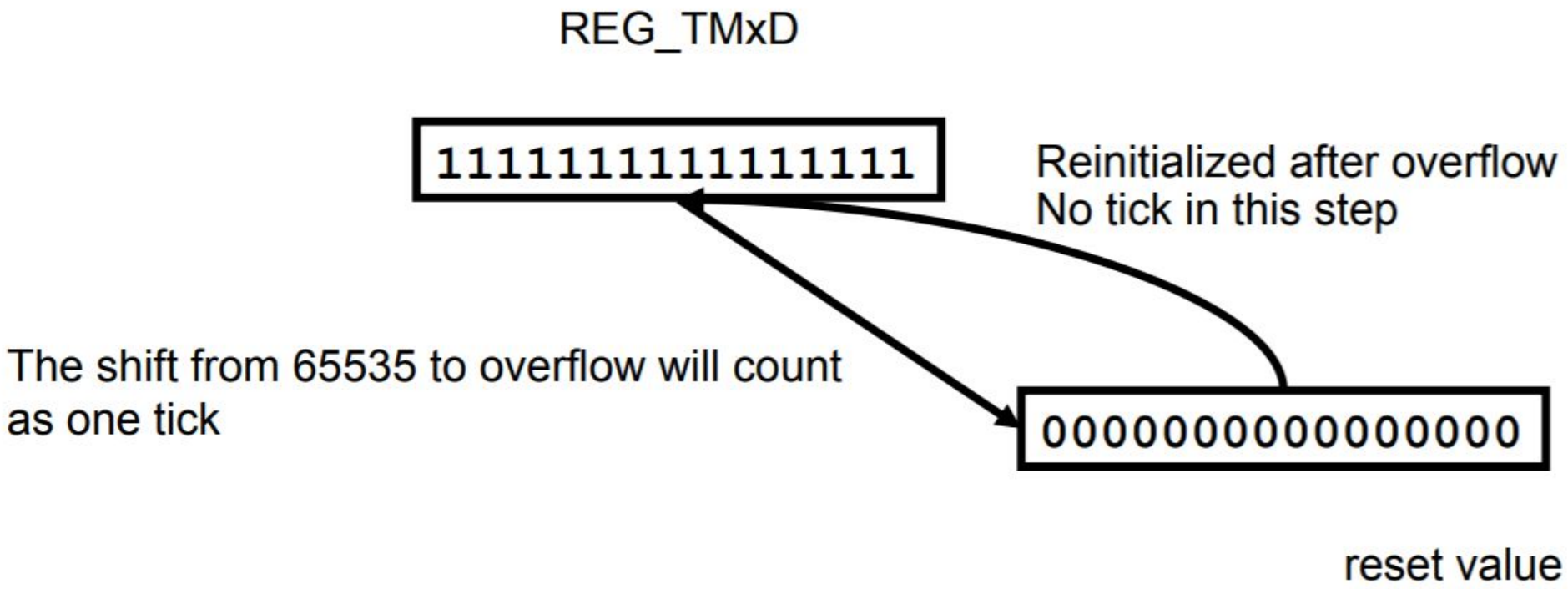
- Remember, times count up.
- Give timers a value other than zero to start counting from.
- Precisely control how long before the timer overflows.
- You can only initialize the value of a disabled timer.
- A timer initialized to a value will reset to *that value* (not zero) when it overflows.

Setting Timers

- Remember, timers count up.
- If you wanted to set a timer to go off after just one tick, what do you set its value to?

Setting Timers

- Remember, timers count up.
- If you wanted to set a timer to go off after just one tick, what do you set its value to?



Rules: Initial Timer Values

- Can only initialize the value of a disabled timer
- A timer initialized to a value will reset to that value (not zero) when it overflows
- Initial value is stored somewhere
- If you just let it start to 0 it will count up until it again reaches 0 (overflow).

Micro and Nanoseconds Those are REALLY SHORT

Bits 0-1	# CPU cycles	Frequency	Period
00	1	2^24 Hz or 16,777,216 Hz	59.604ns
01	64	2^18 Hz or 262,144 Hz	3.811µs
10	256	2^16 Hz or 65,536 Hz	15.259µs
11	1024	2^14 Hz or 16,384 Hz	59.382µs

Exercise:

- Set up a 1 Hz timer, that is it will 'tick' every second
- Which Frequency should we use?

We only get to count 65536 times! Which two work?

Bits 0-1	# CPU cycles	Frequency	Period
00	1	2^24 Hz or 16,777,216 Hz	59.604ns
01	64	2^18 Hz or 262,144 Hz	3.811µs
10	256	2^16 Hz or 65,536 Hz	15.259µs
11	1024	2^14 Hz or 16,384 Hz	59.382µs

1 second timer:

Frequency_256:

- If we set that to zero, we get to count exactly 65536 times!

Frequency_1024:

- If we set that to 65536 - (16384Hz / 1Hz)
- So just 65536 - 16384

```
void int_in_one_minute(void)
{
    REG_TM0CNT = 0; // Turn off timer 0
    REG_TM1CNT = 0; // Turn off timer 1
    REG_TM0D = -0x4000; // 1 sec
    REG_TM1D = -60; // 60 secs = 1 minute
    REG_TM0CNT = TM_FREQ_1024 | TM_ON;
    REG_TM1CNT = TM_IRQ | TM_CASCADE | TM_ON;
}
```

Addresses

```
#define REG_TM0CNT *(volatile u16*)0x4000102
#define REG_TM1CNT *(volatile u16*)0x4000106
#define REG_TM2CNT *(volatile u16*)0x400010A
#define REG_TM3CNT *(volatile u16*)0x400010E

#define REG_TM0D       *(volatile u16*)0x4000100
#define REG_TM1D       *(volatile u16*)0x4000104
#define REG_TM2D       *(volatile u16*)0x4000108
#define REG_TM3D       *(volatile u16*)0x400010C
```

- To find the value of a timer, check the number stored in REG_TMxD