Wee Dig Dug: Documentations

Anand Balakrishnan
anandbal@buffalo.edu

Amrit Pal Singh
asingh42@buffalo.edu

May 10, 2017

Contents

1 User Guide

1.1 Introduction

Welcome to **Wee Dig Dug**, a simplified, text-based version of the popular arcade game **Dig Dug** by Namco!. The following project was written in ARM Assembly for the LPC2138 Education Board, with the ARM7TDMI architecture.

1.2 Instructions

You are playing as Mr. Wee Dug, a glorious knight and miner (yes, it is an unconventional combination). You have been recruited by some villagers to kill a few beasts and you get paid depending on what kind of beast you kill.

- Kill a **FYGAR**, a species of vicious dragons, known to take a 100 knights to defeat a single one, and you get 100 points.
- Kill a **POOKA**, often mistaken for a cute, cuddly creature until it tries to eat you up, and you get 50 points.

To top it off, you will be locked up in an abandoned mine (abandoned because of these monsters obviously), and have 120s to kill all monsters, else you die (and don't ask me how you win, this is an arcade game). Best part is, you can mine the sand for 10 points per block, which is one of the perks of being a miner, right?

So, your objective is to earn as many points as you can within the 120 seconds. GOOD LUCK!

1.3 Setup

Before playing the game, please make sure of the following:

1. You are connected to the correct COM port on PuTTy, and at a baud rate of **115200 baud**.
2. Resize the console window to a minimum of 30 rows × 130 columns.
3. Be prepared to enjoy the game.

To start playing, just flash the code onto the LPC2138 board.

1.4 Controls

Keystroke	Action
W	Move UP
S	Move DOWN
A	Move LEFT
D	Move RIGHT
SPACEBAR	SHOOT a bullet
Momentary Push Button	PAUSE Game

1.5 Legend

Symbol	Character	Points
X	FYGARs	100
O	POOKAs	50
$\vee$, $\wedge$, <,>	DUG (YOU)	N/A
#	SAND	10
Z	WALL	N/A
EMPTY BLOCK	AIR	N/A

2 Design Overview

The game is designed based on the **Model-View-Controller (MVC)** architecture. This architecture is a common way of designing applications with a User Interface. In it, each of the following components is responsible for a particular task, and only that component is allowed to perform that task.

Model	This is the part of the framework responsible for maintainin the internal representation of the application. The **Model** in this game holds the location where sand is present, state of each sprite and state variables of the game.
View	The **View** is responsible for rendering the **Model** onto the GUI. It contains routines that display the board, the sprites and the sand on a console screen.
Controller	The **Controller** is responsible for handling user input and triggering changes in the **Model**. It contains the entry point for the game and interrupt handlers.

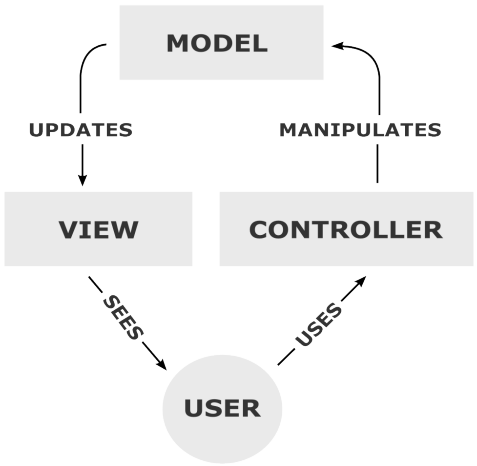

Figure 1: The components of the framework interacting with each other (courtsey: Wikipdia)

2.1 Division of Work

Task		Primary Developer/Author	
CONTROLLER		Anand Balakrishnan	(anandbal)
MODEL			
	model.s	Anand Balakrishnan	(anandbal)
	controller.s	Anand Balakrishnan	(anandbal)
VIEW			
	gui.s	Anand Balakrishnan	(anandbal)
	peripherals.s	Amrit Pal Singh	(asingh42)
DOCUMENTATION			
	User Guide	Anand Balakrishnan	(anandbal)
	Design Overview	Anand Balakrishnan	(anandbal)
	Controller	Anand Balakrishnan	(anandbal)
	Model	Anand Balakrishnan	(anandbal)
	View		
	GUI	Anand Balakrishnan	(anandbal)
	Peripherals	Amrit Pal Singh	(asingh42)

3 Controller Design

```
      FILES:  controller.s
WRITTEN BY:  Anand Balakrishnan (anandbal)
```

The **Controller** mainly contains interrupt handlers, and it is also the entry point for the game. In is, we do the following:

- Initialize timer and timer match registers for periodic interrupts.
- Listen for UART0 interrupt, read the keystrokes and perform the corresponding action.
- Listen for External Interrupt Button press and PAUSE the game.

The **Controller** is a relatively small component, responsible mainly for updating the **Model** via subroutines exposed by the **Model**. It triggers periodic updates to **Model** and receives user input, which it queues into **Model**. It's main components are below:

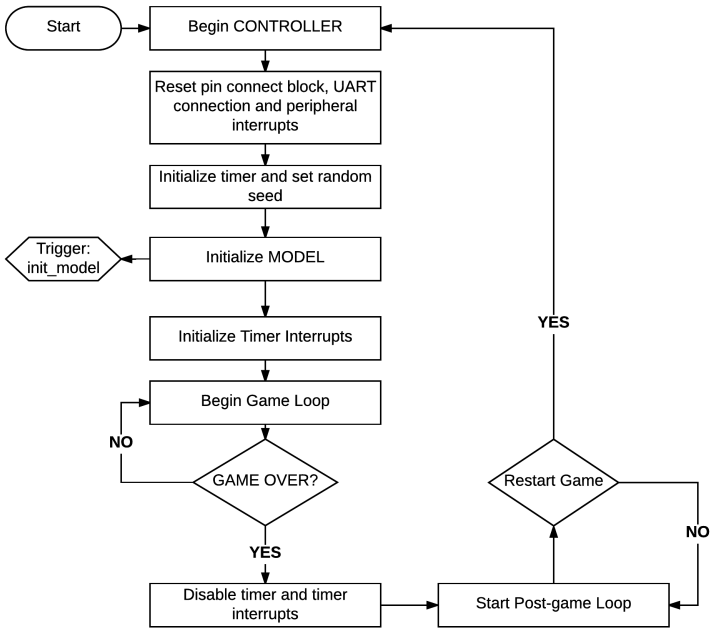

Figure 2: Initialization of **Controller** and entry point for program

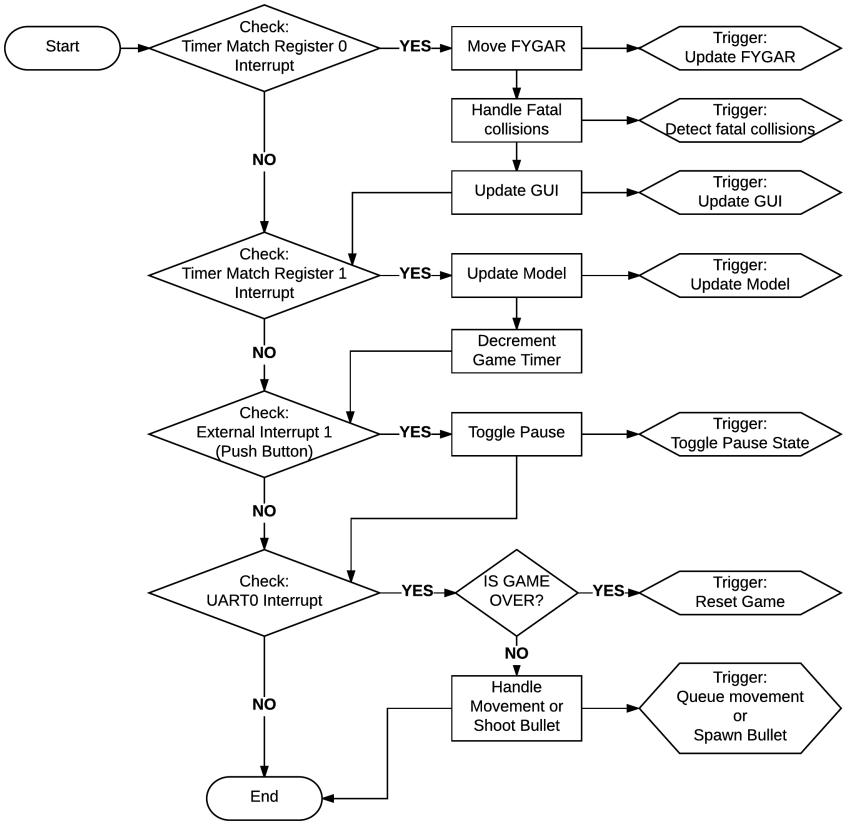

Figure 3: FIQ Handler to send periodic updates to **Model** and for listening to user input to control the game.

While the **Controller** is only a small part of the application, it is the origin of all triggers. It is the interface between the user and the application.

4 Model

FILES: `model.s`, `collisions.s`
WRITTEN BY: `Anand Balakrishnan (anandbal)`

The **Model** maintains the internal representation of the board and triggers **View** updates. It exposed routines that allows the **Controller** to trigger updates on the **Model**.

4.1 Implementation

The **Model** consists of an *"array"* (created using the `FILL` directive) of size 19×15 bytes, each byte representing a grain of sand.

The **Model** also consists of `DCD` tables to hold information of sprites. These tables are structured similar to a `struct`, see **Listing 1**. There are also staticaly defined regions of memory that keep track of the various states the game could possibly be in, for example, `PAUSE`, `GAME_OVER`. The **Model** is also responsible for keeping track of other variables of the game, such as, `LEVEL`, `HIGH_SCORE`, `CURRENT_SCORE` and `TIME`.

```
1 SPRITE
2   DCD X_POS ; Holds X coordinate of the sprite
3   DCD Y_POS ; Holds Y coordinate of the sprite
4   DCD LIVES ; Holds Number of lives the sprite has
5   DCD DIRECTION ; Code for direction the sprite is moving/facing
6   DCD OLD_X_POS ; Previous X coordinate of sprite
7   DCD OLD_Y_POS ; Previous Y coordinate of sprite
8   DCD ORIGINAL_X  ; Original X position (to reset when respawning)
9   DCD ORIGINAL_Y  ; Original Y position (to reset when respawning)
```

Listing 1: Structure for `SPRITE` data

4.2 Operations

Operations that are defined by the **Model** are:

- Initialize and reset model.
- Move sprites and update entire model.
- Handle and detect collisions.
- Get if sand exists at given (x,y) coordinate on the board.
- Clear sand at given coordinate (x,y).
- Toggle game states (`BEGIN_GAME`,`PAUSE`, `GAME_OVER`, `RUNNING`).
- Update individual sprites.
- Spawn sprites.

4.2.1 Initialize and Reset Model

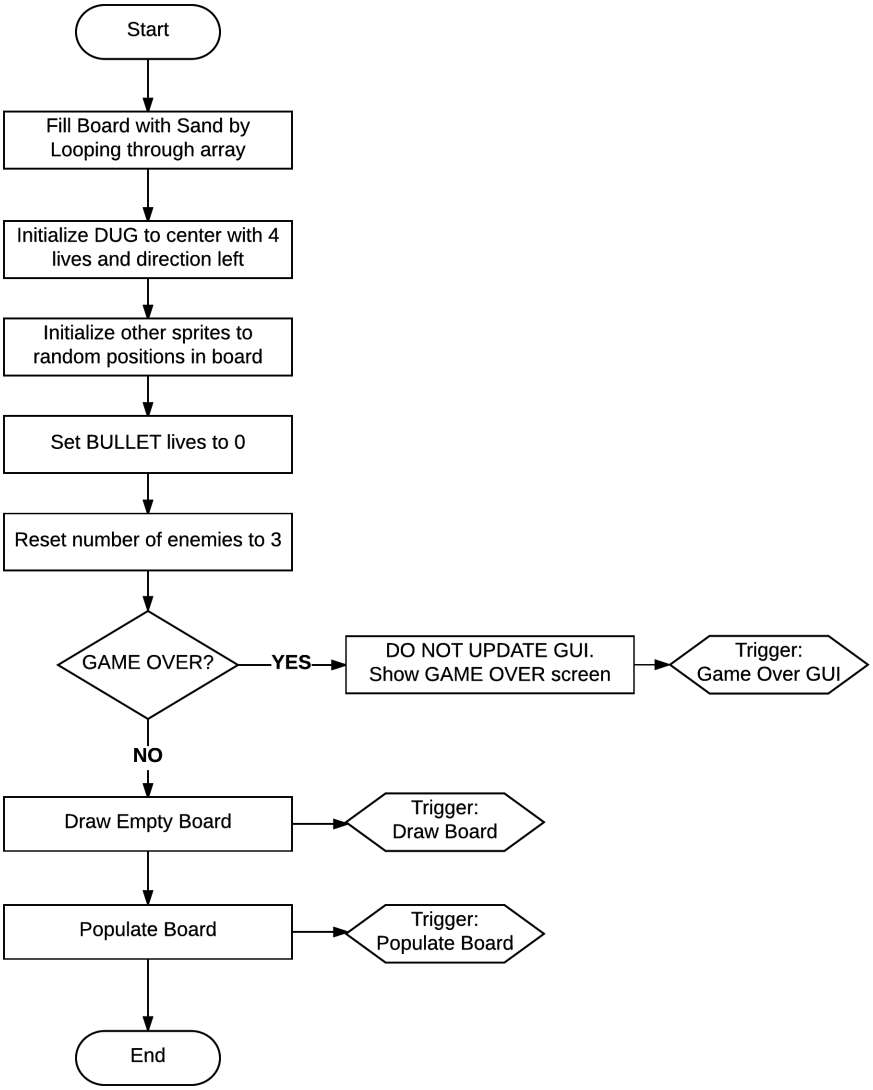

Figure 4: Reset Model Flowchart

Model is initialized by setting the `CURRENT SCORE` to `0` and `LEVEL` to `1`.

Then the **Model** is separately reset, that is, it resets the position of the sprites, refills the board with sand and prepares the game to be played. The **Model** is reset in 2 instances, when the game is initialized at boot up, and when the **Model** is in the `GAME OVER` state.

4.2.2 Game States and Representation

There are 4 variables that describe the **Model**'s state. These are:

`BEGIN_GAME`	True if we need the game to begin from the instructions screen.
`RUNNING_P`	This tells us whether the game is running or not. This is 1 when the game is running.
`PAUSE_GAME`	If the game is not running and this flag is up, the game is in the `PAUSED` state.
`GAME_OVER`	When the game is not running and this flag is up, the game is in the `GAME OVER` state.

For each state, model is updated differently. If game is not `RUNNING` the model isn't updated and the appropriate subroutine call is made to handle either `PAUSED` or `GAME OVER` state, where the GUI is updated to show the state.

4.2.3 Update Model and Control Sprites

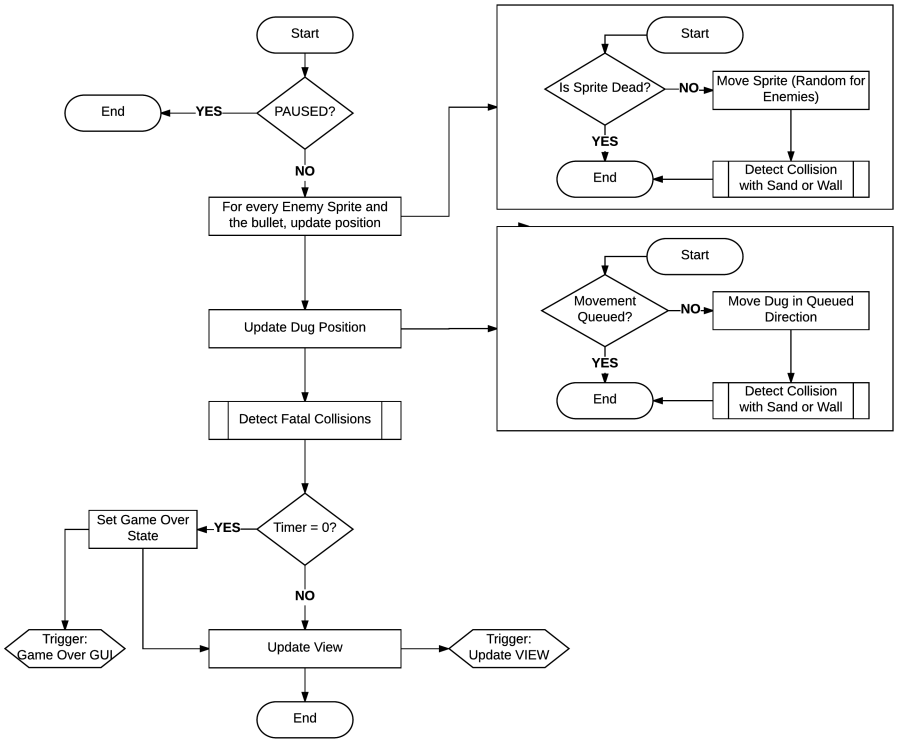

Figure 5: Update Model Flowsheet

4.2.4 Collision Detection

Collision detection is the main component of the **Model**. This is because there are 2 possible outcomes from each collisions, and because of the nature of the game, there are 2 ways collisions can happen.

But first, let us discuss the outcomes for each collision and how it differs depending on the involved sprites/objects.

Fatal Outcome

Fatal outcomes are those outcomes in which the victim sprite loses a life when it collides with another sprite or object. Depending on the victim sprite, the attacking sprite can differ. The following is a mapping from type of sprite to sprites/objects that are fatal to it, along with any other outcomes:

- Enemies – Bullet → Points for killing enemy
- Bullet – Sand, Wall, Enemy → Nothing
- Dug – Enemy → GAME OVER if Dug has 0 lives

Non-Fatal Outcome

Non-fatal outcomes are those in which the victim sprite does not lose a life during the collision. The outcomes differ based on the sprites involved in the collision. The following are all possible non-fatal collisions between two types of sprites/objects, with the outcome of the collision:

- Enemies – Enemies → Nothing
- Enemies – Wall/Sand → Enemy sprite chooses a random free path around it and heads along that.
- Dug – Wall → Nothing.
- Dug – Sand → Gain 10 points.

Now, let us discuss the types of collisions between moving sprites. These are 2 scenarios, **Type 1**, where the two sprites are on the same spot, or **Type 2**, where the sprites were right next to each other in the previous frame and pass each other in the next one. The second one is the more challenging type and occurs when the two sprites are on a head-on collision while being right next to each other. By the nature of fram updates, the sprites don't land on the same coordinate.

Type 1: Sprites arrive on the same Coordinate

This is the easy case, when the sprites arrive on the same spot. This is the more common case and can be detected by just checking if the two sprites have the same X and Y coordinate.

Type 2: Sprites do not arrive on the same Coordinate

This happens when sprites do not arrive on the same position when on a head-on collision course. This scenario can be shown easily with the following illustration:

Frame 1	>	.	.	<
Frame 2	.	>	<	.
Present Frame	.	<	>	.

Here, > and < are sprites moving towards each other

This can be detected by checking the following:

- `SPRITE 1`'s current position and `SPRITE 2`'s old position are same.
- `SPRITE 2`'s current position and `SPRITE 1`'s old position are same.
- If both of the above are `TRUE`, the collision is fatal.

5 View

5.1 Implementation

The **View** is responsible for rendering **Model** onto the GUI. It possessed routines that the **Model** uses to trigger updates to the GUI. This implementation was chosen as updates can be triggered as and when the **Model** is updates.

The **View** is holds many strings with ANSI escape sequences. Some of these stored strings are used to do the following:

- Change location of cursor (see **Listing 2**).
- Display game stats (time left, current refresh interval, level, high score, current score, etc).
- Print empty board (just walls).

```
1 ESC_cursor_position = 27,"["    ; Beginning of escape sequence
2 ESC_cursor_pos_line = "000"     ; This can be changed in code to change row
3 ESC_cursor_pos_sep  = ";"
4 ESC_cursor_pos_col  = "000"     ; This can be changed in code to change column
5 ESC_cursor_pos_cmd  = "f",0     ; End of sequence
```

Listing 2: Escape sequence to change cursor position

5.2 GUI Operations (*PuTTY* output)

```
                FILE:  gui.s
PRIMARY CONTRIBUTOR:  Anand Balakrishnan (anandbal)
```

The **View** also holds routines whose primary function is to use these strings and manipulate GUI. It exposes the following subroutines so as to allow the **Model** to trigger updates to GUI.

- `draw_empty_board`
- `populate_board`
- `update_board`
- `clear_sprite`

5.2.1 Board Initialization

The subroutines `draw_empty_board` and `populate_board` are responsible for displaying the game board before the game begins. The `draw_empty_board` subroutine does the following:

1. Output the string containing an empty board with the walls.
2. Output the initial game stats, game legend and controls.

The `populate_board` subroutine does the following:

1. For each sprite in **Model**, read the X and Y positions, and display it on the board.
2. For each coordinate on the board, check if the **Model** holds a block of sand at that coordinate and fill the board with sand accoding to the mode.

5.2.2 Update GUI

The subroutine responsible for updating the entire GUI is `update_board`. It is also a relatively small routine and does the following:

1. Check if `GAME OVER`. If so, display Game Over GUI. Else.
2. Erase all sprites by loading in their coordinates and printing a ‘ ’ at each coordinate.
3. For each sprite, check if the sprite is alive and print the appropriate GUI character at the sprite's current location.
4. Display game stats.

5.3 Peripheral Operations/Updates

The subroutine `update_peripherals`, updates the 7 segment display, the RGB LED, and monochrome LEDs. This is responsible for displaying the game's states on the hardware peripherals. It reads the 4 game state variables, `RUNNING_P`, `BEGIN_GAME`, `PAUSE_GAME` and `GAME_OVER`, and other variables like `LEVEL` and number of `LIVES` held by **Dug** to determine:

1. Color to display on the RGB LED.
2. Number of LEDs to illuminate in the standard single-color LEDs.
3. Number to display on the 7-segment display.

This routine is triggered periodically by **Model** by through `update_model`.