

# Wee Dig Dug: Documentations

Anand Balakrishnan  
anandbal@buffalo.edu

Amrit Pal Singh  
asingh42@buffalo.edu

May 6, 2017

## Contents

<b>1</b>	<b>User Guide</b>	<b>2</b>
1.1	Setup . . . . .	2
1.2	Instructions . . . . .	2
<b>2</b>	<b>Developer Guide</b>	<b>2</b>
<b>3</b>	<b>Design</b>	<b>2</b>
3.1	Controller . . . . .	3
3.2	Model . . . . .	3
3.2.1	Implementation . . . . .	3
3.2.2	Operations . . . . .	4
3.3	View . . . . .	4
3.3.1	Implementation . . . . .	4

# 1 User Guide

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.

## 1.1 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 30rows  $\times$  30columns.
3. Just enjoy the game.

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

## 1.2 Instructions

You are the character **Dug** and your job is to :

1. Dig through as much of the sand as you can.
2. Kill all enemies.
3. **You must kill all the enemies withing 2 minutes or else you die.**

You can move **Dug** using the W,A,S,D keys which correspond to UP,DOWN,LEFT,RIGHT respectively and shoot your bullet using SPACEBAR. You can use the 5th Momentary Push Button to PAUSE the game whenever you want.

# 2 Developer Guide

## 3 Design

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.

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

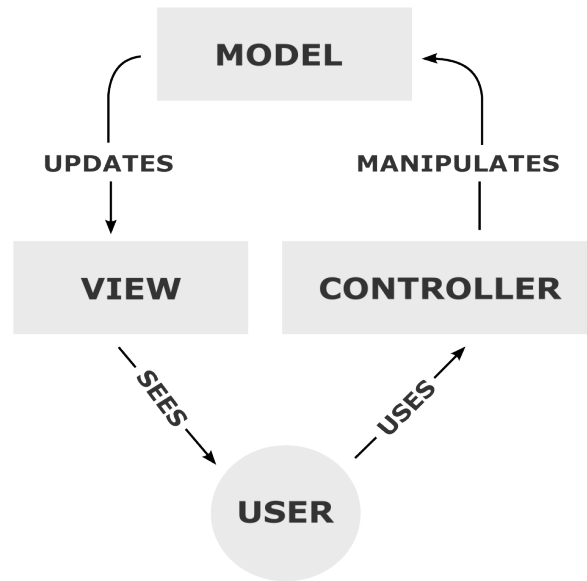


Figure 1: The components of the framework interacting with each other (courtesy: Wikipedia)

### 3.1 Controller

**FILES:** `controller.s`

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 sub-routines exposed by the **Model**.

### 3.2 Model

**FILES:** `model.s`, `collisions.s`

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**.

#### 3.2.1 Implementation

The **Model** consists of an “array” (created using the `FILL` directive) of size  $19 \times 15$  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`. There are also statically 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

### 3.2.2 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.

## 3.3 View

**FILES:** `gui.s`, `peripherals.s`

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.

### 3.3.1 Implementation