Laboratory of Embedded Systems

Project Report

Master's degree in:

Embedded Computing Systems

Academic year 2015/2016

Submitted by:

Elena Lucherini

Scuola Superiore Sant'Anna, July 24, 2016.

Contents

1 Introduction

The system taken into consideration is a game similar to popular mobile game "Candy Crush". Its name is ***Square Smash*** and it can be played in Single Player or Versus (two players) mode. This chapter contains a brief description of the system.

1.1 General description

The game has three modes:

- **Startup Mode:** when the system is powered on, it jumps into Startup Mode. From this mode, it is possible to start playing either in Single Player or in Versus mode, by tapping one of the buttons accordingly.
- **Single Player Mode:** By tapping on button Single Player, the system shifts into Single Player Mode. A randomly generated grid filled with squares of different colors appears. The game timer is started (it is currently set to 60 seconds) and the score is printed on the top right corner. The user can
  - select rows/columns of squares of the same color by tapping on the squares. When three or more squares are selected, it is possible to 'smash' them by tapping the *Smash* button, appearing on the right. The smashed squares are simply replaced with new, randomly generated, squares. The score is increased by 1 point for each smashed square.
  - tap on the *Randomize* button. This will replace the entire current grid with a new, randomly generated one - useful when there are no moves left on the current grid (that is, when there are no rows/columns of three or more squares of the same color). The Randomize button is disabled after three times it is selected.

  The game ends when the timer expires, or when there are no more moves available in the grid and the Randomize button is no longer available. A message appears with the final score.
- **Versus Mode:** By tapping on the Versus button in Startup Mode, the systems goes into Versus Mode. Each player can play on their own device according to the mode of operation described for Single Player Mode. At the end of the game, a message appears comparing the scores of the two players and determining the winner. As opposed to Single Player Mode, the opponent disconnecting from the game will cause the end of the game for both parties.

1.2 Implementation and hardware components

To implement the system, the following components were used:

- Two STM32F4Discovery boards
- Two LCD touch-screens
- An RS-232 cable

The system was implemented in C, using Erika Enterprise RTOS.

All communications needed between the two boards (for the Versus Mode) are implemented through serial communication.

2 Tasks

This chapter contains a description of the four tasks implementing the system and running on each board.

- **Touch:** this task handles the touch position on the screen and all related events. See Section 2.3.
- **Timer:** this task is in charge of the game timer. See Section 2.4.
- **End:** this task manages the end of the game and the next actions to take, depending on the game Mode and on how the game ended. See Section 2.5.
- **Disconnected:** this task detects if the opponent has disconnected in Versus Mode, forcing the game to end. See Section 2.6.

Each task handles the graphics to which it is related: the Touch task handles the graphics associated to the game, the Timer task the timer graphics, and the End task the messages shown at the end of the game. The Disconnected task does not manage any graphics, as the End task already takes care of that.

2.1 Shared variables

The tasks communicate with one another by means of a set of shared global variables. Please note that the variables are not protected by mutex resources, as pre-emption is disabled. Thus, each task will always leave the shared variables in a consistent state before another task uses it.

Three variables manage the game modes and the end of the game:

1. `start` is a boolean whose value is set to `true` if the system is in Startup Mode.
2. `versus` is a boolean whose value is set to `true` if the Versus Mode was selected; otherwise, the system is in Single Player Mode.
3. `end_game` is a boolean, set to `true` if the game has just ended.

The following variables are directly related to the game:

4. `play_timer` is an unsigned 8-bit integer which represents the seconds left before the timer expires.
5. `activate_timer` is a boolean whose value is set to `true` if the timer is on.
6. `randomize` is an integer whose value indicates how many times the user can still use the Randomize button during the game.

7. `comb` is a boolean whose value is set to `false` if there are no more valid combinations available of squares to smash in the current grid. In this case, the user cannot play any more moves.

2.2 Initialization

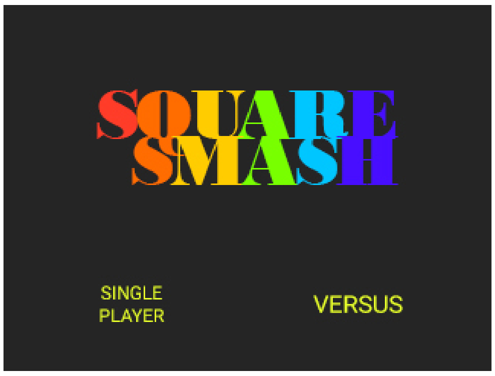


Figure 2.1: Startup Screen

In the main task, all structures associated to the OS are initialized, along with the LCD and the data related to the USART (see Section 2.5.1 for more details).

Next is a call to a function which initializes all the global shared variables associated to the game. The `init()` function also draws the Startup screen on the LCD, as shown in Fig. 2.1.

Finally, the three tasks are initialized and activated.

2.3 The Touch task

2.3.1 Description

The Touch task has a 100 ms period, chosen through trial and error.

First, the task determines whether the touch screen has been activated. If it has, it gets the position of the touch and takes actions depending on the current mode. The following roughly describes the actions taken by the task in each situation:

1. If the system is in Startup Mode (Fig. 2.1), the only 'responsive' areas are the ones corresponding to the two buttons, *Single Player* or *Versus*. If one of those is tapped, the game starts either in Single Player Mode or Versus Mode. The only difference between the two Modes at this stage is the value of the `versus` variable, which is set to `true` only if the Versus button was tapped. A grid of squares is then randomly generated, as shown in Fig. 2.2.

2. If the system is in Single Player or Versus Mode (Fig. 2.2), the responsive area is the grid of squares and the *Randomize* button.

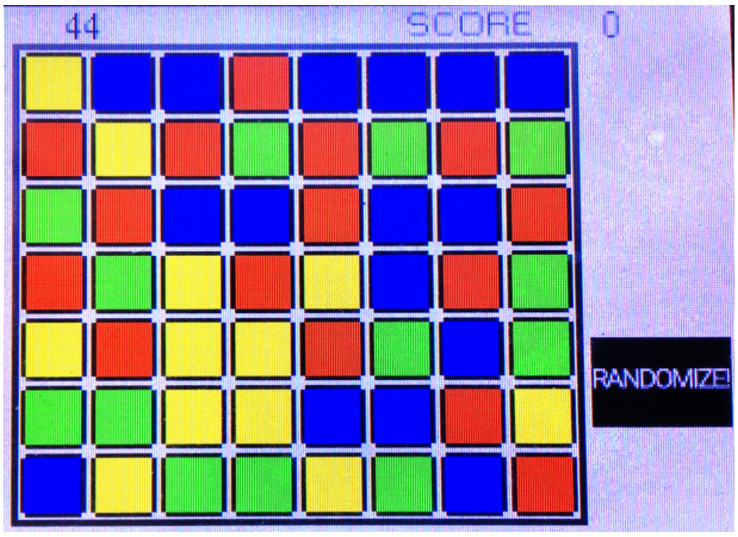


Figure 2.2: Play grid

- If the user taps a square (as mentioned in Chapter 1), it is selected - it can then be deselected by tapping on it again. When a square is selected, only adjacent squares of the same color can then be selected - any other attempt is ignored by the system.
- When three or more squares are selected, the Smash button appears. The selected squares are then smashed and replaced with new, randomly generated ones.
- If the user taps the Randomize button, shared variable `randomized` is decremented and a new randomly generated grid replaces the current grid. If `randomized` is equal to zero, the Randomize button disappears.

3. If the system is transitioning back from Single Player or Versus Mode to Startup Mode, the `init()` function is called, so that the variables are brought back to their initialization state.

The Mode is determined by using three shared variables: `start`, `end_game` and `activate_timer`. Namely:

- The system is in **Startup Mode** when `start = true`, `end_game = true` and `activate_timer = false`.
- The system is in **Single Player** or **Versus Mode** when `start = false`, `end_game = false` and `activate_timer = true`.
- Otherwise, the system is **transitioning** from Single Player/Versus Mode to Startup Mode.

2.3.2 Creating new grids

The grid is managed by two global variables that are only used by the Touch task. The first one, called `matrix`, is a two-dimensional array of unsigned 8-bit integers. Each element corresponds to a square and contains the code of its color (from 0 to 3, respectively:

blue, red, yellow and green). The second variable, `matrix_hl` is a two-dimensional array of booleans built as the first one, which keeps track of which squares are selected (*hl* = *highlighted*).

To create a new grid, the `generate_table()` function is called. Such function calls the `generate_square()` for each element of `matrix`. `generate_square()` assigns a random integer number from 0 to 3 (corresponding to one of four colors) and draws the square on the screen.

2.4 The Timer task

The Timer task has a 1 second period and manages all the time events. If the timer is on, it simply decrements the `play_timer` shared variable, which keeps track of the time until the timer is expired. Finally, it draws the value of `play_timer` on screen. Please note that the timer is on only when in Single Player or Versus Mode.

When the timer expires, when there are no more moves available (that is, when the game has come to an end), or when the opponent disconnects in Versus Mode, the timer is reset, `activate_timer` is set to `false` and `end_game` is set to `true`. If there are no more possible combinations and the timer has not expired, `comb` is then set to zero, signaling that the game has ended because there are no more available moves.

2.4.1 Checking for available combinations

This is done through the `check_comb()` function. For each element of `matrix` (described in Section 2.3.2), `check_comb()` checks if at least two adjacent squares are of the same color as the current element. When a combination is found, the function returns `true`; otherwise, it returns `false`.

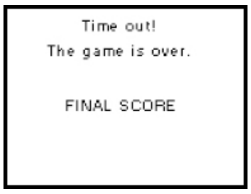
The function only checks the adjacent squares on the right and below the current square, as it starts from the zeroth element and, for each square, the ones on the left and on top has already been checked. Furthermore, the last two element of each row or column is not checked on its right or below, respectively, since it would not be possible to find a combination of at least *two* adjacent squares.

2.5 The End task

The period of the End task is set to 100ms and was found by trial and error. The task recognizes when the game is ending and manages the message to show depending on how the game ended.

Namely, it is possible to distinguish among the following combinations:

- End by timeout: if shared variable `comb` is not equal to zero, the game ended because the timer expired. A message is shown accordingly (see Figure 2.3).
- End by moves: if shared variable `comb` is equal to zero, the game ended because there are no more available combinations. A message is shown accordingly (see Figure 2.4).


Time out!
The game is over.

FINAL SCORE

Figure 2.3: Game over message: Timeout

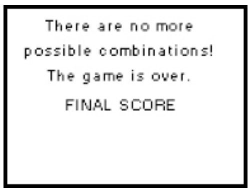

There are no more
possible combinations!
The game is over.
FINAL SCORE

Figure 2.4: Game over message: No more combinations

- End by disconnection: when the opponent disconnects from the game in Versus Mode (that is, the opponent resets their device). The message printed on screen is shown in Figure 2.5.

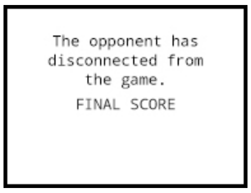

The opponent has
disconnected from
the game.
FINAL SCORE

Figure 2.5: The opponent has disconnected

In either case, the `end_game` variable is then reset to `false`. It is then sufficient to tap anywhere on the screen to go back to Startup Mode and start a new game.

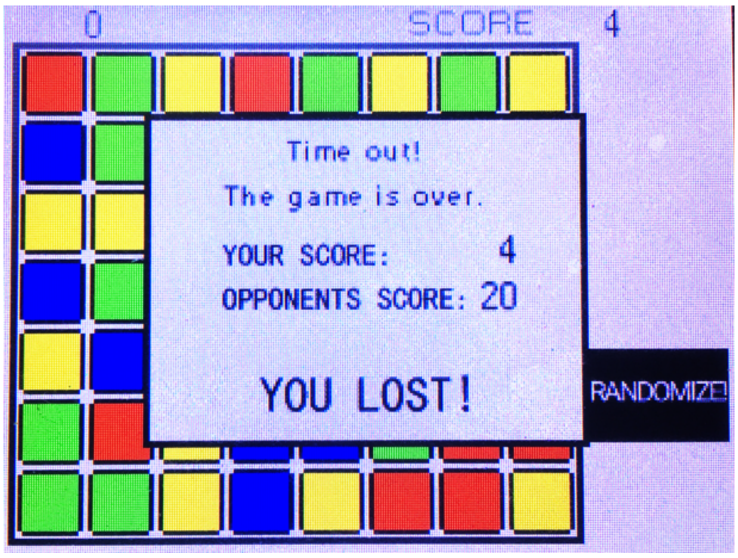
The end of the game is detected by checking three shared variables. The game ends if:

- `activate_timer` is `false`,
- `end_game` is `true`,
- `start` is `false`.

As mentioned in Section 2.4, the Timer task sets `activate_timer` to `false` and `end_game` to `true` when the game ends. Since this combination is verified at the initialization stage, too, it is necessary to perform an additional check on the `start` variable to make sure that the system is not in Startup Mode.

2.5.1 End in Versus Mode

In addition to the actions described above, if a game is ending normally in Versus Mode the users must share their results. This is implemented by means of serial asynchronous communication.



Time out!
The game is over.
YOUR SCORE: 4
OPPONENTS SCORE: 20
YOU LOST!

Figure 2.6: Endgame in Versus Mode

Asynchronous serial communication As mentioned in Section 2.2, the structure to manage USART communication is initialized in the main function. Both receive and trasmit are enabled, and the word length is set to 8 bits (scores are always much smaller than 255). Asynchronous communication was preferred to synchronous communication, as the former is simpler to implement and the amount of data to share is minimal.

When a player completes the game (Player 1), their system sends the score to the other player (Player 2). Player 1 then waits for Player 2 to finish - that is, they wait until Player 1's system has received Player 2's score. When the exchange has been completed, each player's system compares the two scores and determines the winner locally. A message is printed on screen, as shown in Figure 2.6.

As opposed to Single Player Mode, the `end_game` variable is not reset to `false` until said exchange has been completed.

End by disconnection If the opponent resets their device in Versus Mode, the player is notified and the game ends. No action is taken on the part of the user who restarts the device, as the game can normally continue after the event. Also, note that no error is given when the serial cable has been disconnected, as it suffices to connect it again to continue with the game regularly. For more details, see Section 2.6.

2.6 The Disconnected task

The Disconnected task is clocked at the same period as the End task (100 ms). The goal of this task is to check whether the opponent is still playing or their device has restarted (this is the only way to leave the game when the system is powered on).

Among other operations, the `init()` function (see Section 2.2) also sends an 8-bit integer to the opponent with value -2 (*disconnected value*).

The Disconnected task ignores any data received when the game has not started yet and takes no action when the game starts in Single Player Mode. If the system is in Versus Mode, the task checks for received data and, if it receives the disconnected value, it sets an additional shared global boolean variable, `pl2_disconnected`, to `true`. The game can then end, with the operations described in the previous Sections.

Please note that the -2 value has been chosen because the End task ignores it, which is the only other task to manage received data from serial communication. The End task expects to receive the opponent's score and ignores values that are smaller or equal to -1 (the initialization value of the variable storing the opponent's score).

3 Issues and solutions

3.1 Touchscreen issues

The LED touch screen employed is not capable of providing a high level of accuracy when detecting the touch position. Therefore, the game, because of its nature, may occasionally present glitches.

An example of such problems is observable when a square is tapped but an adjacent one gets selected instead. To minimize the occurrences of this issue, the dimensions of the squares were adjusted so that they be large enough for a mostly accurate detection of the touch position; and small enough to keep the game interesting and not too easy.

The periodic task managing the touch events, 'Touch', is also tuned for the touch screen to be responsive at its best. It may also occur that a square that gets tapped to be selected still appears to be not selected. This is because the period of the task managing the touch events, 'Touch', is too short. The task ends up detecting two touches: one to select the square and a second one to deselect it. However, increasing the period would lead to a loss in the accuracy of detection: fewer touches would be detected. The period chosen, 100 ms, is a trade off between the two cases.

3.2 The combination issue

As mentioned in the previous chapters, there are two reasons for the game to end in Single Player Mode: the game timer expires, or there are no more possible combinations of squares. A first version of the specifications of the game required that the game end only because of the timer expiring: in that case, whenever a group of squares got smashed, it would have been necessary to replace the grid so that a possible combination be available at any moment.

Such a scenario is more complex than the one adopted (see Section 2.3.2): with the solution presented in this paper, it suffices to replace the smashed squares with randomly generated ones and then check for available combinations whenever a move is carried to completion. The Randomize button, also part of the solution, is meant to make the game last longer if no combination is detected (or the user just can't see it).

3.3 Multiplayer issues

- The play grids are randomly generated (see Section 2.3.2) and should be the same for both players to guarantee a fair game. A possible solution is to implement a master-slave behavior, where the master generates the grid and then shares it with the slave through synchronous communication. A more simple solution, which is the one adopted, is based on an assumption: the seed of the pseudo-random generator must be the same on both devices, so that the behavior of said generator will also be the same. This is guaranteed by the Erika Operating System and no further coding is required.

- As all communication between the two party is carried asynchronously, the system is not capable of detecting whether the opponent has disconnected from the game. This part has been implemented separately to guarantee more coherence in the gameplay. For more details, see Section 2.6.

- With more than one task managing the asynchronous communication (see Sections 2.5 and 2.6), the send and receive flags must be reset to maintain a correct communication. In this implementation, the flags are cleared each time data is sent or received, respectively. However, this is not sufficient to guarantee a correct communication after a few games have been played without restarting: in some cases, an old score might persist, while the latest one might be accidentally discarded. The solution used here is to clear the receive flag an additional time, whenever a new game has been initialized.