G12 Racer Game Design Manual

Written by:

- Mohammad Ghasembeigi (z3464208)
- Phillip Khorn (z3463638)
- Pearlie Zhang (z3460347)
- Savinka Wijeyeratne (z3433816)

Introduction

The purpose of this document is to explain the design choices made in writing the program code that implements the G12 Racer Game.

A variety of decisions had to be made in regards to control flow in which developers had to create a program will good control flow and efficient code. The core flow of the program is shown in the **system control flow** section.

Due to the limitations of the AVR board, controlling the memory used and allocated to each game variable was difficult and these problems and their solutions are discussed in the **data structures** section.

The state of the game was altered using various algorithms and techniques which are described in the **algorithms** section.

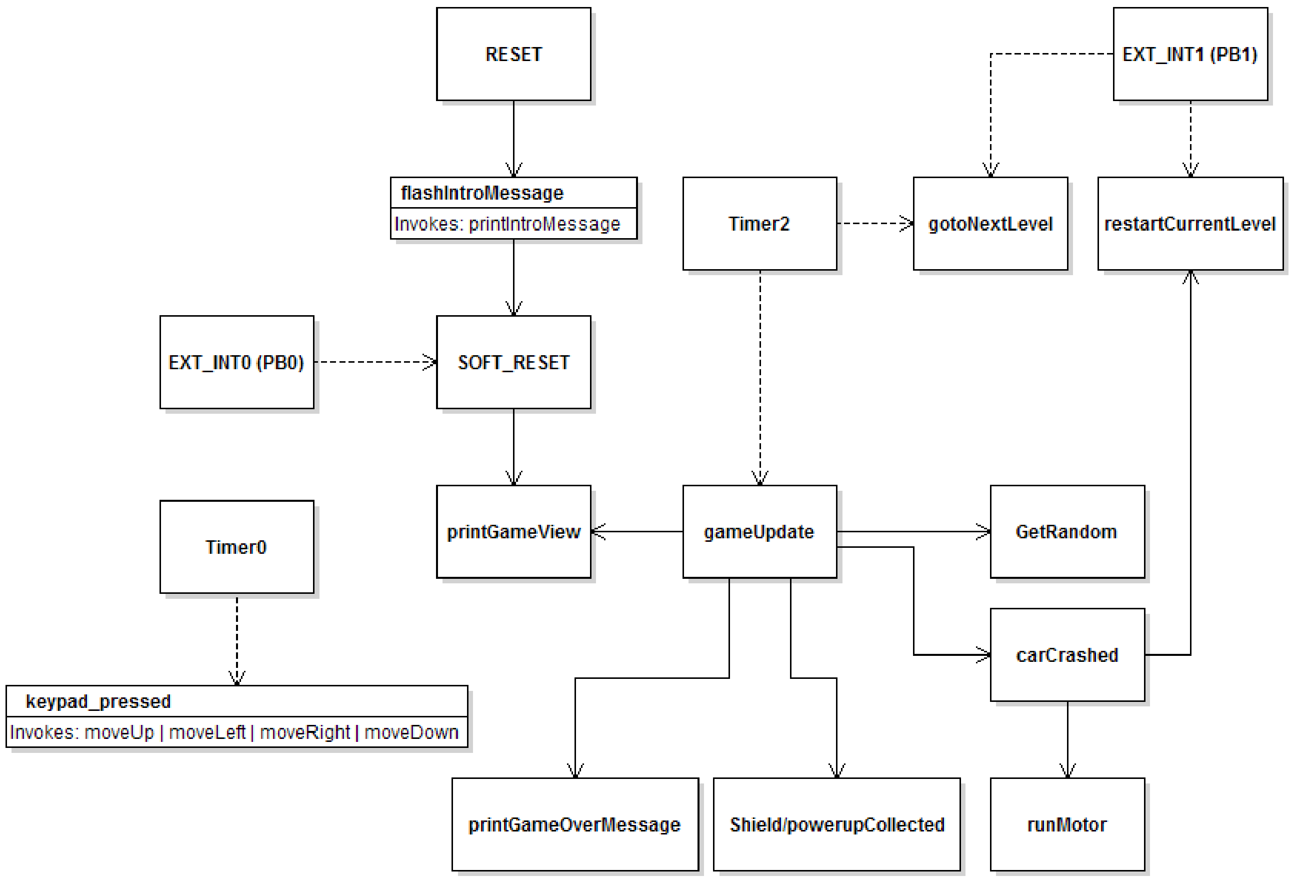
Module specifications documents each module in the final program code and its input and output as well as module calls. This section additionally documents control flow for all modules including minor/helper modules.

a) System Control Flow

The diagram below is a visual representation of the control flow of the game program.

Straight lines represent method calls and dotted lines represent method calls from an interrupt.

Minor methods were omitted from this chart as they are not required in describing the system control flow.



b) Data Structures

Registers

Register 16 (data_lo)
This is a register pair with Register 17. They are mostly used to temporarily store 16 bit numbers. It is used by the delay function to store the smaller numbers for the delay. It is used to poll the keypad, overflow counter for the time step, debouncing for the push buttons and updating the game, storing the randomly generated number and preparing the next level. data_lo's data is also used for output in the printing to the LED, LCD initialisation and printing.

Register 17 (data_hi)
This is a register pair with Register 16. They are mostly used to temporarily store 16 bit numbers. It is used by the delay function to store the higher number for the delay in the delay function, a temporary register in the Timer interrupts, push buttons, updating the game, storing data_lo's value in LED printing. data_hi's data is used for LCD printing and initialisation.

Register 18 (temp)
This is the general temp register. It is used for initialisation of stack, timers, external interrupts, counters, keypad, LED, LCD
It is used to poll the keypad, check if INT_1 has been double pressed in the required time, general time counter, debouncing for the push buttons, flashing the intro and end message, LCD printing storage, updating the game, storing the motor speed , restarting and preparing new level, time step and generating a random number.

Register 19 (level)
This register stores the level where the player currently is. It is initialised with the start level. It increments when the player reaches a new level, is printed out to the LCD, ensure the correct time step and used to add to the score.

Register 20 (score_lo)
This stores the lower number of the score. It is a pair with register 21. It is initially cleared. It is printed to the LCD and LED. It is updated in gameUpdate.

Register 21 (score_hi)
This stores the higher number of the score. It is a pair with register 20. It is initially cleared. It is printed to the LCD and LED. It is updated in gameUpdate.

Register 22 (car_top), Register 23 (car_bot)
This stores the position of the car if it is on the top line or bottom line. car_top is initialised with LINE_POS_1. It can me moved left,right,up or down through moveLeft, moveRight, moveUp and moveDown. It is used to check if the car has crashed into a powerup or an obstacle. It is also printed to the LCD.

Register 24 (obs_top), Register 25 (obs_bot)
This stores the position of the obstacles if it is on the top line or bottom line. car_top is initialised with 0. It is used to check if the car crashed into the obstacles and is left shifted every step. It is printed out to the LCD.

Register 26 (powerup)
This stores the position of the power_up. The front half is the top and the second half is the bottom. It cleared in initialisation. It is used to check to make sure that the power ups disappear in the front half and if the car has collected it. It is left shifted every step and printed to the LCD.

Register 27 (shield)

This stores the position of the shield The front half is the top and the second half is the bottom. It is used to check to make sure that the shields disappear in the front half and if the car has collected it. It is left shifted every step and printed to the LCD.

Register 28 (lives)
This register is used to store how many lives the player has left. It is initialised with START_NUM_LIVES (3). It is printed to the LCD and is decremented if the player crashed in gameUpdate.

Register 29 (mask)
This stores the mask for the keypad. It is initially stored with the column mask, used to check if any row or column is grounded.

Register 30 (col)
This is used to store the current column. It is initially cleared and incremented once it has been established that the current column was not pressed when checking the keypad.

Register 31 (row)
This is used to store the current row. It is initially cleared and incremented once it has been established that the current row was not pressed when checking the keypad.

Stacks

Because of the limited general purpose registers, our game uses stacks for additional temporary. Stacks are primarily used to store the current values of conflict registers. This allows us to keep our data whilst using the registers in another function. Registers are pushed onto the stack when we need the register and are popped once we have finished. It is initialised in RESET and used by most modules. It is important to note that not all registers could be pushed onto the stack so that they could be used as temporary variables. For example, pushing one of the score registers onto the stack in a longer module while Timer2 updated the game view on the LCD would cause the changed/temporary value of the score register to be displayed which is clearly unwanted behaviour.

Z Register

The Z register was used in some modules to point to addresses containing string constants in flash memory so that they could be written character by character. This is the ideal way to print out strings and was thus adopted. Note that the Z registers are actually defined to be used elsewhere and where thus pushed/popped off the stack when required.

String constants in flash

String Name	**String**	**Use**
GAME_OVER_STRING	GAME OVER	Used to print out to the LCD when a player died. It is on the top row
SCORE_STRING	SCORE:	Used to print out to the LCD when a player died. It is on the bottom row and describes their score
INTRO_TOP_STRING	G12 RACER V1.2	Used to print out to the LCD top line when the game is started initially. Is not repeated with INT_0 is pressed
INTRO_BOT_STRING	PREPARE TO PLAY!	Used to print out to the LCD bottom line when the game is started initially. Is not repeated with INT_0 is pressed

Variables in data memory

Some variables were stored in data memory as all general purpose registers were used. These variables were not constantly changed or used in displaying the game view on the LCD and it is thus reasonable to store them in data memory. Variables were accessed with **LDS** and **STS** when required.

All variables are stored in DSEG starting at address 0x100.

Label Name	Size (byte)	Use
RAND	4	It is used to store the random seed from timer 0 (TCNT0) and used in the number generator
Timer_Counter	1	It is cleared at initialisation. It stores the current number of interrupts invoked.
Timer_Overflow_Count	1	It is cleared at initialisation. It stores the current number of interrupt overflows.
Timer_Overflow_Per_Sec	1	It is cleared at initialisation. Stores the current time step.
Timer_Level_Counter	1	It is cleared at initialisation. It stores the current number of interrupts invoked.
Timer_Level_Overflow_Count	1	It is cleared at initialisation. It stores the current number of interrupt overflows. This is incremented when it has been one second (ie OVERFLOW_PER_SEC overflows).
Timer_Level_Sec_Count	1	It is cleared at initialisation. Incremented when one second is over. Used to control next level logic.
TIMER_INT1_Second_press	1	It is cleared at initialisation. Checks if there is double press for INT1.
TIMER_INT1_Overflow_Count	1	It is cleared at initialisation. Is incremented to until it is equal to OVERFLOW_FOR_DOUBLE_PRESS. After this the player opportunity to double press is finished.
IS_PLAYER_SHIELDED	1	It is cleared at initialisation. Shows if a player has a shield or not
MASK_STORE	1	It is used to store keypad masks to check and debounce keypad presses

c) Algorithms

Overview

The state of the game is maintained primarily by interrupts and general purpose registers. After a reset (soft or hard), all game state variables are set to their default or initialised values. This includes variables in data memory, and registers R16-R31. All external interrupts are also enabled and set to their default state.

After resetting, the program loop indefinitely and the game state is only altered via interrupts. These interrupts and modules invoked from the interrupts use different algorithms to perform required tasks. These algorithms are detailed below.

Shifting of Entities in Obstacles, Powerup and Shield Registers

At each time step, non-player entities are moved left 1 position using a logical left shift.

Obstacles are representing across two registers (as 16 positions may contain obstacles). A **lsl** on the top and bot register for obstacles allows us to move every obstacle left 1 position, remove the left most obstacles from the game and clear up the right most positions to allow a new entity to be spawned.

The case for score powerups and shields is however different. Powerups and shields disappear half way across the game screen and thus entities in both the top and bot 4 right most positions on the screen can be represented using a single register. We have the 4 MSB in a register representing the top row (4 right most positions in top row) and the 4LSB in the same register representing the bot row (4 left most positions in bot row).

As a result, we use **lsr** to move all these entities left 1 position. However, while the top rows last obstacle is automatically removed, we must clear bit 4 (as the previous bit 3 has moved into this position and is past the half way position). After clearing this bit, we can spawn a new entity at bit 0 or 4 depending if we want the entity in the top row or bottom row.

Printing the powerup and shield registers involves similar bit manipulation. We simply create a mask 0b11110000 or 0b00001111 and logcial **and** this with the register to focus on the bits of importance. We are then free to print the bits in the correct positions on the LCD.

Counting with Timers

Timer2 was used to count a variety of things in the game such as the time step and each level duration. Each timer worked by having 3 variables. A timer counter which counts the number of interrupts invoked. A timer overflow counter which counts the number of times the first counter had overflown. This value is compared to some overflow per duration constant or variable. If the two values are equal, then the necessary amount of time had passed. The overflow per duration is determined theoretically and can then be modified based on delays in interrupts.

Push Button Debouncing

Software debouncing was implemented on Push Button 1 and Push Button 2. After each external interrupt, a check is done to check that the button is pressed. If it is, a short delay occurs and the check if performed again. If the button is still pressed then the action is performed.

Whilst software debouncing is not perfect, this allows us to reduce the number of unintended presses to a very small number.

LED Bar Graph Bit Reverse

It was necessary to perform a bit reverse on registers due to the use of ports used for the LED. The process involves starting with a mask 0b1000000 which is shifted right each step using **lsr**. We then take the register and check the bit using a logical **and** with our mask. If the bit is not set, we do nothing and proceed to the next step. Otherwise if the bit is set, we add **128** to the result register. Before this addition, this result register is shifted right using **lsr**. Thus we compare bits 7-0 in order and add their result to the MSB of the result register before eventually shifting 7 times.

This effectively reverses the bits in the input register allowing our LED connections to be wired properly. Note that one could alternatively switch the cable pins on one end of the connections for the same effect. However, this algorithm allows portability for even larger LEDs, perhaps with 16 lights which would be capable of displaying the theoretical maximum score.

Random Number Generation

A random number generator is implemented which uses Timer0 and an interrupt counter for levels to get a seed. A series of operations involving shifts and rotations are performed on this seed to generate random numbers which are used to introduce an element of pseudo-randomness in the spawning of obstacles, score powerups and shields.

d) Module Specification

Address Label Names

Each module was given a clear and informative name to help developers understand the program code. Labels in these modules had the module name as a prefix to the label so that duplicate, similar and unclear labels were avoided.

Module Specification

This section describes each module (including interrupts), its input (preconditions) and its output (postconditions). Any module invokes are also listed. This information also exists at a lower level in the source code

RESET	**Invokes: SOFT_RESET** Reset the entire game by initialising the stack and displaying the game intro message. Then **SOFT_RESET** is invoked.
SOFT_RESET	Soft reset the game state. This initialises all game state variables and begins the game instantly in the initial game view.
Timer0	**Input**: N/A **Output:** N/A No prescalling timer. Polls the keypad each interrupt. This ensures a reactive move of the player is performed on keydown as opposed to an update each time step or a slower update within an iterative main function.
keypad_pressed	**Input:** Row and Col are set to the correct row and col corresponding to the key that was pressed. **Output:** N/A **Invokes: moveUp or moveLeft or moveRight or moveDown** Determines key pressed and invokes appropriate move module to move player (car).
Timer2	**Input**: N/A **Output:** N/A **Invokes: gameUpdate, gotoNextLevel** No prescalling timer. Manage timers in data memory by counting overflows. When the appropriate time step has passed, **gameUpdate** is invoked. Timer2 also handles the resetting of the double press INT1 time as well the 30 second level timer, after which **gotoNextLevel** is invoked.
EXT_INT0	**Invokes: SOFT_RESET** Debouncing implemented. On keypress, soft reset game state to restart game.
EXT_INT1	**Invokes: restartCurrentLevel, gotoNextLevel** Debouncing implemented. On single key press, current level is reset with **restartCurrentLevel** being invoked. On double key press (two presses in short duration of time), the game moves onto the next level with **gotoNextLevel** being invoked.
lcd_write_com	**Input:** data_lo contains command value to be written. Write a command to the LCD.
lcd_write_data_lo	**Input:** data_lo contains value to be written. Write an ASCII character to next position on LCD.
lcd_wait_busy	Read the LCD busy flag until it reads as not busy.

lcd_init	Initialisation function for LCD.
flashIntroMessage	**Input:** N/A **Output:** Output blank screen to LCD **Invokes: printIntroMessage** Invokes **printIntroMessage** then delays and then outputs a blank screen to LCD. This is repeated a particular number of times to create a flashing effect. Finally, **printIntroMessage** is invokes with a significantly longer delay at the end.
printIntroMessage	**Input:** N/A **Output:** Output two line introduction message to LCD. Displays the tile of the game as well as the version of the game on the first line of the LCD (“G12 RACER V1.2”) and a welcoming message on the second line of the LCD (“PREPARE TO PLAY!”).
printGameOverMessage	**Input:** N/A **Output:** Output two line game over message to LCD. Displays centred “GAME OVER” on the first line of the LCD and centred “SCORE: XXXXX” on the second line of the LCD. X represents either a space or a number representing the players final score
printGameView	**Input:** N/A **Output:** Output two line game interface to LCD. **Invokes**: **printNumToLCD** Query the game state variables and display game interface with latest game state information. This interface is detailed in depth in the **user manual.**
printNumToLCD	**Input:** Number must be stored in **data_lo**:**data_hi**, Number must be unsigned (ie 0 or positive), Number must be equal to or lower than than 65535, **temp** stores the number of digits to print, must be 0 < **temp** <= 5 **Output:** Output ASCII characters corresponding to numbers in pair register with prepadding of spaces. 5 digit ATOI function with prepadding (using spaces) feature.
gameUpdate	**Input**: N/A **Output:** All entities are moved 1 position to the left. Powerups past half way are removed. Points are added for obstacles that just left the game area. **Invokes**: **printGameView, printGameOverMessage , carCrashed, powerupCollected, shieldCollected** This subroutine will update the game state, moving all obstacles to their next location. It also handles the scoring as well as collisions of the player with obstacles or powerups or shields New elements to be added on LSB are determined by chance and a random number. A request to reprint the LCD is made at the end of this function once game state has been updated.
moveUp, moveLeft, moveRight, moveDown	**Input**: N/A **Output:** Player moves to new position if possible. **Invokes**: **printGameView, printGameOverMessage** Each module moves the car (player) in a particular direction as described in the module name. **printGameView** is then invokes or **printGameOverMessage** if the player crashes as a result of the move.
carCrashed	**Input**: N/A **Output:** Displays explosion symbol on screen and runs motor if crashed. Life is lost and **restartCurrentLevel** is invoked**.** **Invokes**: **printGameView, printGameOverMessage, runMotor, restartCurrentLevel** Called to check if car has crashed with an obstacle and performs the appropriate

	action.
runMotor	Runs the motor at specified speed (70rps) for a duration of specified speed (2 seconds).
powerupCollected	**Input**: N/A **Output:** Gives player bonus points if powerup collected. Removes powerup entity from game area. Called to check if power up has been collected. If so, powerup is removed and bonus points are added
shieldCollected	**Input**: N/A **Output:** Adds shield buff to player or gives player bonus points if shield collected. **IS_PLAYER_SHIELDED** set to correct value. Removes shield entity from game area. Called to check if shield has been collected. If so, shield is removed and shield buff is applied. Will also give bonus points to player if collecting shield while shield equipped.
restartCurrentLevel	**Input**: N/A **Output:** Level restarts, all entities removed from game area. Restarts the current level without restoring previous score.
gotoNextLevel	**Input**: N/A **Output:** Level is incremented, timer variables are reset, new level begins. **Invokes: restartCurrentLevel** Updates all variables required to prepare the next level of the game. That is, timestep is updated, timer variables are reset and **restartCurrentLevel** is invoked after **level** is incremented.
updateLEDScore	**Input**: **score_lo:score_hi** contain score to be displayed. **Output:** LED is updated with current score in binary. **Invokes: bitReverse** Updates the LED score based on current score in: **score_lo:score_hi**. Will display binary score from 0-1023 and then overflow for numbers after that. **bitReverse** is invoked due to the nature of ports connections utilised.
bitReverse	**Input**: **temp** with bit positions to reverse. **Output: data_lo** with reversed value of initial **temp** input Takes register in **temp** and reverses bit positions in register, storing result in **data_lo**. Values of **temp**, **data_lo** and **data_hi** are altered.
delay	**Input: data_lo:data_hi** contained low and high value of duration to delay for. **Output:** N/A Produce delay greater than ~1.08us***data_lo:data_hi**.
generateRandomNum	**Input:** N/A **Output: data_lo** contains generated random number. **Invokes: InitRandom, GetRandom** Invokes **InitRandom** followed by **GetRandom**
InitRandom	**Input:** N/A **Output: RAND, RAND+2** get TCNT0 value and **RAND+1, RAND+4** get Timer_Level_Overflow_Count value as seeds. Generate seed based on timer constants.
GetRandom	**Input:** Requires timer based seed to have been made and stored in RAND addresses by **InitRandom** **Output: data_lo** contains generated random number. Generate a random number between 0-255.