**B31DD Mini-Project Report**

*Yifei Jing*

[*yj10@hw.ac.uk*](mailto:yj10@hw.ac.uk)

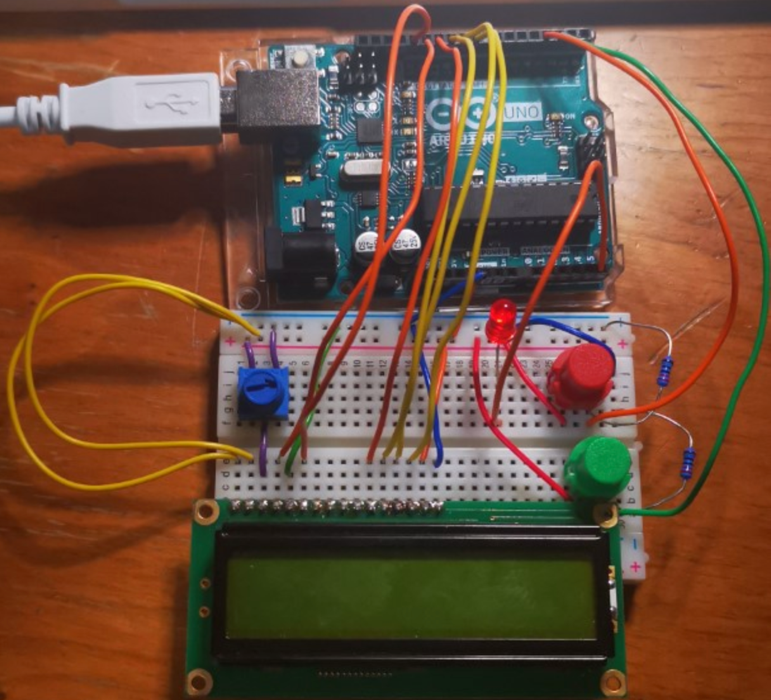
*H00263865*

*1. Abstract*

This report records the implementation of a game system with flappy bird on an ATmega328p based Arduino Uno board. The components used and mark related issues are mentioned in the *Description* section. The details of the design of the system will be covered in the *Design* section. The diagram of the circuit will be demonstrated in the *Circuit* section. The evaluation of the system will be discussed in the *Performance* section. Finally, miscellaneous issues will be mentioned in the *Discussion* section.

*2. Description*

The components used in this project include an LCD screen, buttons, a potential meter, some resistors and wires. The final step should be a game system contains a [*flappy bird*](https://en.wikipedia.org/wiki/Flappy_Bird) game. This project will not use any third-party libraries except the formal AVR compile tool chain and AVR’s specific chip (ATmaga328p) definition libraries. As it is advised from the supervisor, some real objects which are hard to find or too expensive to buy can be replaced by components at hand, with the essence of concentration to the perfect ideas and pretty designs on embedded system, the author finally chose to use one led to indicate a sound generator, which can produce a ding-dong when the bird is passing an obstacle successfully, just the same as what the mobile version has featured.



*3. Design*

*3.1 Framework Design*

Actually, the final completed implementation should not be simply called a game, however, a game system, as the design of the game can be separated out from the hardware part. That is to say, the game is completely independent of the platform, as the game code files contain nothing about the specific hardware usage. The author first implemented the hardware layer using the provided AVR library (the definitions on registers, peripherals on board), then abstracted interfaces for the upper layers, so that the upper layer can use the resources from the hardware on chip without knowing what the specific hardware is. After that, a multitask real time system can be built to provide the thread abstract for the hardware layer. Owing to the complexity and tediousness of the implementation on display, that part of code is isolated as a graphics part (It is always the case that the implementations on I/O are much more difficult than those of strategies to the threads for a system). Therefore, the structure of the system can be demonstrated as the chart below.

Figure 1

The AVR library provides various of peripherals for the hardware layer, then the hardware layer provides APIs which are appropriate to software development.

There are a deal of advantages for this structure. First, the system can be replanted to any other embedded system, as it is just the bottom layer (AVR layer in this project) has to be changed, we can implement the same set of APIs in the hardware layer to build the operating system. Then the code of the game in the software layer can be reused without any edits (As the code only cares about the logic of the game). Second, the system is extensible. Because we can develop software without caring about the details of hardware, we can make other applications based on the APIs form the hardware layer. Third, the system is easy to be maintained. It is still possible to get on implementing this project after a few months, as the structure is quite clear for us to review it.

*3.2 Game Control Flow*

In the next part, I will introduce the design of this system from the top to the bottom, as it is the way I used to implement this system. The reason is that we do not always know what kinds of APIs we will want for the lower layers, but we definitely know what the software, running the main functions of the system, should be like (the flappy bird game in this project). From my perspective, the game should contain three states: begin, playing, end. In the begin state, there should be a menu for the player to know the game and what to do to start the game. In the playing state, there should be a bird and several obstacles on the screen for the players to interact and control the bird. In the end state, the score of this turn should be displayed, and possibly, the history record should also be displayed. Then we can declare the functions for the whole game process.

A close up of a sign

Description automatically generated

As indicated in *flappybird.h* this three functions will control the process of the game. We now consider the details of each states. In the begin state, the information needs to be displayed on the screen, then the only thing to do is to wait for a signal which indicates the game start (A button pressed). That’s all for the begin state.

A screenshot of a cell phone

Description automatically generated

Then in the playing state, the bird and obstacles should be displayed on the screen, the bird should move forward in a steady pace, and when the bird cannot pass the obstacle, the game end, otherwise, the game will continue.

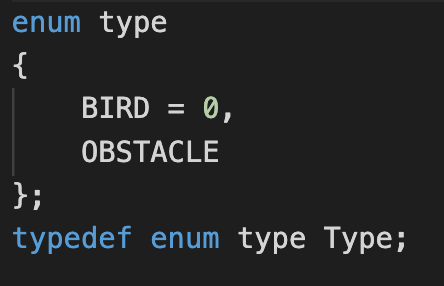
*3.3 Game Data Structure Design*

It is time to think how we can track the bird and the obstacles. They have positions, they have sizes. Actually, they are the same thing just in different forms. Thus, we define a structure for them.

A screenshot of a cell phone

Description automatically generated

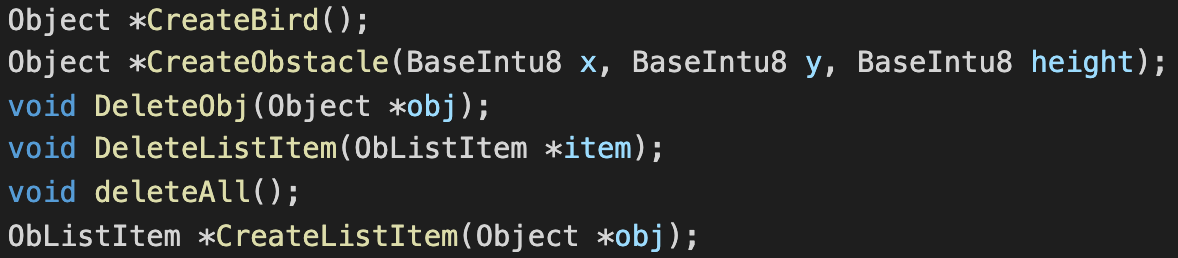
I choose to define a general type called object, then the type will indicate the bird or the obstacle.



Now, we need to find a strategy to organize these objects. There is only one bird, so the bird can be managed isolated. The problem is on the obstacles, which will disappear on left of the screen and new obstacles will appear on the right of the screen. They are not permanent as the bird; thus, some dynamic strategy should be used. Here we use a bi-directional list. The used obstacles are removed from the head and added from the tail, so that the program can manage the obstacles dynamically.

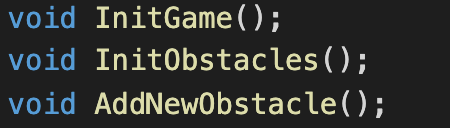
A screenshot of a cell phone

Description automatically generated



*3.4 Operations On Objects And Details Of Process Strategies*

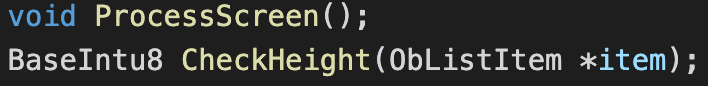
The APIs designed in the header file make it possible to control the playing state efficiently under the hood. Each time an obstacle runs to the left to the screen, we call the delete functions and we add a new obstacle using the create functions. Then here comes another issue, the profile of each obstacle should either be randomly chose or with predefined maps. The advantage for the random strategy is that it is easy to implement and can extend infinitely. The map is adequate for increasing difficulty as the game proceeds and there are no outliers. To make code simple, I selected the random strategy and defined the function below.



We now consider the bird, it is created at the start of the game and deleted at the end of the game. Thus, we can just create it with the obstacles, with x position always 0, and y at the mid of the screen.

We define that the bird will move forward (actually the obstacles move backward) and drop a little on every time slot. Thus, a function should take charge of this with complete case analysis (bird passes obstacle, bird drops to the bottom, bird collides obstacle, etc). Then, there will be a function update the screen based on the current state of each object. This function will use the APIs provided by the graphics.





The CheckHeight function will be called when an obstacle reaches x = 0, it will compare the bird position then judge whether the game should continue.

In the end state, the action is similar to that of the begin state. Display the score and the history score in EEPROM and wait for signal to exit or again.

After implementing the game, we can have a sense what need to be provided from the lower layer.

*3.5 Hardware Layer Design*

We first need to be able read the buttons, so that the player can control the bird and interact with the machine. The strategy here is called message dispatching. The hardware layer reads the button and sends a signal to the software layer, then the game can analyse the signal and found which button is pressed.

A close up of text on a white background

Description automatically generated

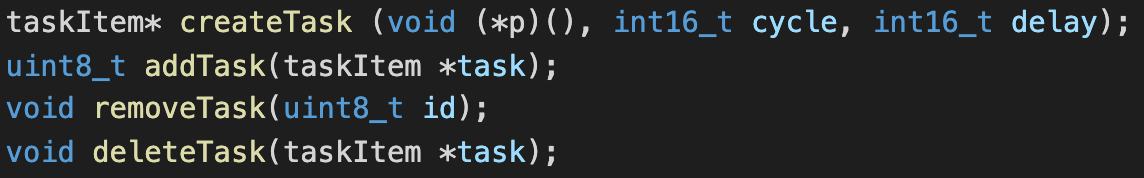
The six buttons in the system are designed in the form that they can be combined using an OR operation, so that multiple buttons pressed can be detected and transformed.

*3.6 Task Switching Design*

Secondly, to make the game proceed step by step, we need the timer in the hardware to put everything into the agenda, like the screen update, the detect of the button. Here is the problem, we need a strategy for the game in the software layer to access to the system and add a routine to be scheduled. Here I consider making a new data structure, a task structure, which contains the action needs to be performed, the cycles need to be run, the time delayed after each call.

A black sign with white text

Description automatically generated



Excellent! The only thing left now is to let the system check the task list periodically, then perform the action when the delay is decreased to zero. Additionally, the tasks can be divided into two groups, resident and temporary, as the screen update task should be run whenever the game is proceeding, and other tasks like shift the strings on the lcd will run only once.

Finally, three resident tasks are loaded into the system during startup. The update task will handle the processing of the game both under the hood and on the surface, that is to say, first call the process(), then call the processScreen(). The cycle attribute of all resident tasks is written -1 to indicate the task will run forever. The delay of update is set by the variable gameSpeed to be the most comfortable pace for the user to interact with the game. This variable can be changed through the function setGameSpeed(). The subtitleshifter task is used for the menu, as the lcd is too small to display all the information in one scene, so that periodic shift is need. Similarly, the delay of 300 is from the experiments, as the characters can neither move so fast that people will feel difficult to read, nor move too slow to wait. The buttonPressTask will deliver signal from the hardware layer to the software layer. The delay is defined by a variable called FPS, which means the frequency of checking button signals. If the value is set to 10, it will check 10 times with equally interval per second. This value is also set to be changed by users dynamically through the call setFPS(), which means that it can be changed by some sensors or just ADC value if components are accessible. three values can all be variables to give better experience of the game (Additional checks need to be implemented to guarantee the value is in the possible range, as the delay cannot be smaller than 0, or too small value). Another contemporary task is the led, which is turn on when the bird passes one obstacle, this task only run once, so the cycle is set to 1.

A close up of a sign

Description automatically generated

*3.7 Graphics Design*

It is time to talk about the graphics, which connects the abstract instances of game objects to the images on the screen. After reading datasheet of the lcd, it is depressing to say that the support of drawing things on the lcd is quite poor. Because it only has 8 memory locations to define graphics by the user. If the display space (what we write to the IO space) is using values of the characters (each character is composed of 5\*8 pixels from the data stored in the character ROM or the graphic RAM), then only one location will be enough (we can use the storage of board to store the graphic and load on to that graphic location dynamically, then write it to the IO space by commands), actually, the reason that it has 8 locations should have implied us that the strategy is not by copy value, by the address of the location to that IO space (It is a direct projection!). To verify this, direct one location to the IO space, then change the content of that location, we can find the picture on the screen has changed. Because of this limit, the max number if obstacles on the screen is only 3 (2 locations for each obstacle as they are vertically placed on the screen). We use the first 2 locations to draw the bird, then other locations are used for the obstacles.

A picture containing object, gauge, device

Description automatically generated



The bird defined above will have a height of 4, and the obstacle is drawn dynamically by its height in the buffer. This sounds easy, as we just write the buffered data to the locations, then write them to specific positions on the screen (Follow the instructions on the data sheet to use graphic memories and write the graph onto an individual position). What if the bird is at the same x position of one obstacle? (at this time, the bird is either passing an obstacle, or running into one obstacle) That is to say, we cannot separate the drawing of bird and obstacle into two parts at this time (As the new drawn obstacle will cover the originally drawn bird). We need to define the interface in the graphics to select to take the bird into consideration while the x meets.



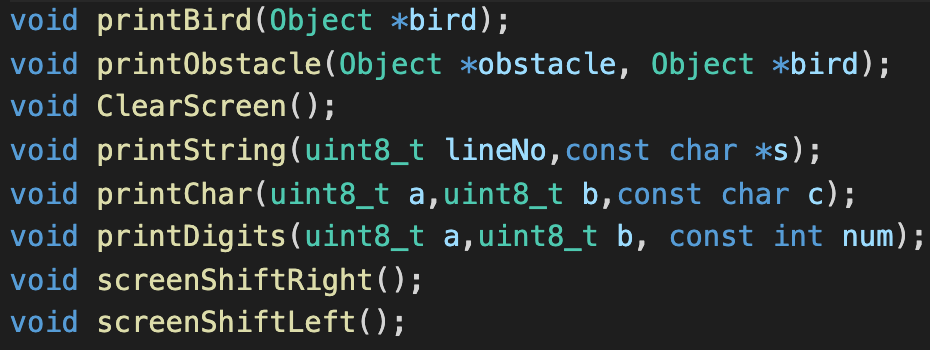
The private method define here uses the profile of one obstacle and the bird, an additional parameter called shadow will determine whether to consider the bird.

Then the exposed interface can be defined as:

A screen shot of a smart phone

Description automatically generated

Now, a set of interfaces can be declared in the header file to be used in the game.



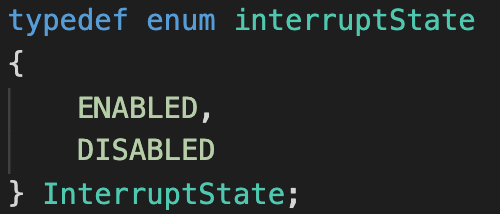
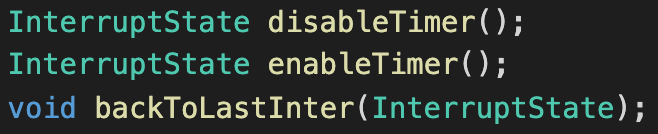
The printDigits() function is used to display the score at the end of the game, and the screen shift functions are just wrappers.

*3.8 Enhanced Persistency Strategy Design*

Until now, we have not talked anything about how to combine them and hook them into control flows to make the game process without crack. The crux now is that, though we have been able to create everything in the game, we are still far from completing the game. It is widely known that a rocket needs so many components, that it needs the group of the most intelligent people to build it, thought the core of a racket is only the fire engine which provides it with kinetic. The additional components are just to keep safe! That is the case that the core function of a system is always a very small part code, and other tedious code are added just to guarantee that the system will run without errors. The issue is that the more cases the system needs to consider, the more insurance code needs, thus the more the complexity of the coupling of the system is.

To elaborate what I mean, assume the action of starting game. The system should convert from the start menu to the game scene once the button is pressed. This seems quite easy, as we have implemented all of the APIs to achieve each step of game: the respective onPressed function will change the game state to playing, and call gameStart() to initiate all the objects for the game, then the resident task will update the game periodically. That’s all fine enough just through thinking the procedures. At this time, we have been putting every brick we need to build a layer of a skyscraper. However, after compiling the project and flashing it onto the board, we find the lcd screen can always corrupt into convulsion. This is because the lcd is a low speed component, it cannot be interrupt during a command cycle! The Onpressed task will frequently be executed and it is likely to interrupt current processing screen (More generally, all of the I/O devices). One very simple solution is to disable the interrupt while dealing with the lcd screen and enable it when return from it. It seems to work from a first view, as to disable the interrupt during update is the demand, however, to enable the interrupt when return is not the demand! The above strategy will enable interrupt whenever the update routine is finished. That’s part of the issue I have just discussed, to make the racket (system) possible to work without problems, we have to add many additionally components we have not considered, and it is always that fixing one problem will make another new problem. The game can proceed without convulsion of the screen, though, the speed of the game will be undetermined, as the update function enable the interrupt during the interrupt function (remember we use the timer interrupt to deal with the task processing), this will enable multiple task processor to execute the tasks. The bird may fly two or more steps in one time slot, and the screen shift may shift rather quickly. If the system has run enough long time, it will possibly corrupt because of stack overflow.

An available solution is to use a local variable to store the current interrupt state at the beginning of the function, disable the interrupt, run the function, restore the previous interrupt state before exit. This inspires to design a set of APIs in the hardware layer to fulfill the demand.



The new strategy will be added to each time-essential function to protect the processing state. The disable and enable call will return the interrupt state before the call, so it is possible to return to the previous interrupt state using the backToLastInter() call.

The new format of the code will look like the below one, where the functions of the method are wrapped by the interrupt charging calls.

A screen shot of a computer

Description automatically generated

*3.9 Debug And Test System Design*

The system has been developed using many interesting strategies to enable persistency and concurrency. As the system grows larger and larger, it can be rather difficult to figure out what introduces the problem when new bugs have been found during using. Actually, all of the discussed implantations are not originally designed to be so smart, however, it is optimized and updated step by step (The reader can check the git logs to see the new features of each version tagged with commits and the problems fixed in each commit.). To make it easier to find the problem, it is better to implement a debug system, which is to record each step during runtime, so that it is possible to track the trace to find the problem.

The debug system begins by define a new macro in the utils.h header, then we add the additional debug information into each function as written below.



A close up of text on a black background

Description automatically generated

The above code is part of process(). The system uses the serial port to transmit record to the PC, thus we can check the printed message to find what is really going on.

The debug message may look like the one below, the message tells everything executing in the kernel, so that it is quite easy to find the problem.

A close up of a screen

Description automatically generated

*3.10 Main Function Isolation*

So far, all of the main features of the system have been discussed. To start the system and the game, call the initialization functions in the main function.

A screen shot of a social media post

Description automatically generated

*4. Circuit*

As this is a game machine, there are mainly two buttons and one screen to be used.

The port definitions can be found in the pinout.h under the root directory, and the submitted sample is connected as figure 2 indicated.

A circuit board

Description automatically generated

Figure 2

*5. Performance*

This game should have used better display component, as the LCD is quite slow in communication and reaction with only two lines available. It can be shown from the debug system, also theoretically speaking, most of the CPU time is wasted on waiting for the screen to complete flash, which makes the system reacting to the button pressed stubborn. It would be better if the chip at least has one DMA, though.

Anyway, the system has been programmed to be rather stable, even the gamer pressed the button randomly or permanently. That is to say, all of the cases have been analysed and carefully implemented. Because the system is based on multitasking, permanently pressing on button will not cause the system hang on (say, the bird may stay at the top of the screen, if the button A is pressed permanently. Because the system detects A is pressed, it will always run the A pressed function permanently. The switching tasks guarantee that the bird will still drop periodically even A is permanently pressed).

The performance is largely determined by the FPS setting by user, and also the GameSpeed value. If the user feels the game speed is too slow, he can change the GameSpeed to a smaller value, and vice versa. If the user feels that the system frequently fails to react to the button, he can set higher FPS. Except that, all of the settings have been implemented to be transparent to the user.

Furthermore, if it is a product, additional buttons or sensors should be added to enable changing the speed and FPS without compiling the code. The system has been implemented to use two potential meters to change the two values. It is just because the space on the bread board is limited and I only have one potential meter which has been used to coordinate the contrast of the lcd screen.

A black sign with white text

Description automatically generatedA clock on the top of a sign

Description automatically generatedA screenshot of a cell phone

Description automatically generated

The above code shows the solution to change the FPS and GameSpeed using system tasks.

*6. Discussion*

*6.1 Task Switch Strategies*

The task strategy is called the cooperative tasking, which means all the tasks need to be scheduled in a defined streamline. One task in the tail needs to wait until the former tasks have all finished. That is why it is called cooperative. Another strategy is called pre-emptive tasking, in this strategy, all tasks are provided with equally sliced time slot, and they need to be managed by priority to compete for the kernel to execute them. In this system, there are not so many tasks, and all of the tasks need to run sequentially (The update needs always to be run first). Thus, the cooperative kernel is selected.

*6.2 VCS*

The project uses a VCS, called git, to develop and control the version, as sometimes it is not sure whether one implementation is right. The simplest way is to use two pointers: master and develop, where master is the stable version, and develop is the developing or test version with new features, after the develop is tested to be stable, then merge it to the master. That is to say, the develop pointer is always to the front of the master pointer. Each commit is done with specific problem recorded and the tags are recording specific feature. We can check the log by running “git log –pretty=oneline”.

A close up of a device

Description automatically generated

This will show simplified information like above. Actually, if the read is familiar to git, he can track the whole procedure of developing this system by starting from v1.0 or see what each feature is implemented by run git diff through different tags.

*6.3 Files Organization*

The organization of the project is using three subdirectories: src has all the source code of the system and the game, include has the headers of the source codes in the src, output has the compiled binary and hex files. In the root directory, there are only the main.c file and the pinout.h file, because only these two files are expected edited by the users (The Makefile is not expected to be edited by users, but it is demanded to run make in the root directory).

A screen shot of a smart phone

Description automatically generated

The above figure shows all files in this project.

*6.4 One More Interesting Status*

Thinking about adding something looks cool to the game to make it more engrossing, a new status is added as the God. As it has been discussed, three status (begin, play, end defined in flappybird.h) are used in the game to control the current processing state. Actually, the new status should be a child of the playing status, as it only valid while playing the game. In this mode the user need not to do anything but to watch the play the game automatically: this is a god player who never lose and never gets tired. The idea is got from analysing people’s interest on watching highly skilled players on TV or Internet. What is the most important is that the user will never need to struggle to flush the highest score, but the god player will complete the task for him. As there are only two buttons on the board, currently, and I don’t want to add new buttons to just a new function, I use the combined the value of A and B to start the god player (remember we have define the values of the six buttons with special values so that any combined pressing event can be detected from dispatching on the signal), and the user can interrupt the god player by pressing either button to get the control of the bird.

After the assumption of the god mode, it is time to consider the strategy of the god. Though the god never lose, he is quite omniscient that he should let the user watching the game feel amazing about his ability. Say, if the bird is directly flying through each obstacle, the user will feel embarrassed for not having such ability. The crux is that the god should play like a normal player who will constricted to the rules of the game (the gravity, the action to loft). This function should actually be described as an AI, then. Additionally, this function makes the implementation of the god status more interesting. In fact, this is also a case analysis, but this time, the machine will interact with its own settings. The god can be adventurous to always choose the lowest height to pass the obstacles or be conservative to keep at the safe level of height. Both of the modes are implemented at the end, checkout the version 3.0.1 and 3.0.2 for the adventurous god and conservative god, respectively. Actually, the final version of the code is not using either type of god at all, as there is a hybrid strategy. The bird needs to not touch the upper bound of the obstacle and not fall on the lower bound, so that the two condition can all be included into decision making.

*6.5 Increase Difficulty For The Game*

The game is still too simple if the bird just flies in a constant speed. As what is the most different from the one on the mobile phone is that there should be accelerations not only on the drop but also on loft condition for the bird. The continuous operations on drop or loft will influence on the bird with higher speed. This will be a very challengeable feature for the game. The gamer will not only press the button for a moment to keep the bird at the safest level, but he must take the upper bound into consideration as the bird is more likely to face on the upper bound. Additionally, with the score increases, which indicates the gamer has passed more and more obstacles, the game speed will increase to provide more challenge.

After this change to the game, the indestructible god can now fail the game, as it is even impossible for a sophisticated gamer to make the decision when to press the button or wait.

*7. Result*

A complete solution has been provided for a flappy bird game in this project. The expected functions and frameworks are all implemented from scratch without any third-source libraries (Even the LCD library has been re-written to improve both efficiency and persistency). The main idea behind all of the designs is abstraction, which means both data and procedure should be designed under strict abstraction, so that they can compose into modules in different layers. As the modules have been classified previously to enable specific concept, they can be flexibly embedded into the system or decoupled away from the system. As each component of the system only completes one specific function, it is convenient for later sustainment. The software layer can be implant in other platform as it does not contain any code relating to the hardware and the hardware layer can be packaged for this specific chip, so that the project can directly start from the logic design when using this chip again.

However, there are also some defaults for this system. The task switching implementation is not the most efficient one, and it is just a cooperative kernel which cannot response quickly under some condition. Originally, the kernel is designed to be a pre-emptive kernel, but code is not robust enough to avoid corruption. Farther research should be done on the pre-emptive implementation. The system was embedded with a Lisp style language, called klisp, which is a subset of MIT-Scheme, that the system can be programmed from its own REPL during runtime. However, after implementing core features, the real problem occurred to be writing the parser in the REPL. C language is well-known for really hard experience on dealing with strings, thus introduced quite a deal of difficulties. It was expected to cost a large amount of time writing and debugging the parser, as I am not going to rely on any third-party libraries. Because of the precious time, I had to removed it in the end, however, I will continue to develop it after the project.

*8. Reference*

*[1]* [*ATmega328P\_Datasheet*](http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf)

*[2]* [*LCD\_hd44780u\_Datasheet*](http://fab.cba.mit.edu/classes/863.06/11.13/44780.pdf)

*[3] Abelson, H., Sussman, G. and Sussman, J. (1984). Structure and interpretation of computer programs. Cambridge, Mass.: MIT Press.*

*[4] Remzi H.Arpaci-Dusseau and Andrea C.Arpaci-Dusseau (2018). Operating Systems: Three Easy Pieces.*