CS 342302 Operating Systems
Fall Semester 2023
Prof. Pai H. Chou

Programming Project Checkpoint 5[1]

New for 2023! This is **not** the same checkpoint 5 as previous years.

What to submit: One zip file named <studentID>-`ppc5.zip` (replace <studentID> with your own student ID). It should contain:

- one PDF file named **ppc5.pdf** for Part 5 of this checkpoint. It should explain how your code works. Write your answers in **English**. Check your spelling and grammar. Include your name and student ID.
- (Part 1) Turn in the complete source files for the function to be compiled using SDCC and targets EdSim51, even if the file was already written.
  - `testlcd.c`, which
    - utilizes the LCD, keypad, buttonbar libraries and integrates them by your two-producers, one-consumer code from the previous checkpoint
  - `buttonlib.{c|h},` which
    - provide driver for the button bank
  - `keylib.{c|h},` which
    - provide driver for the keypad
  - `lcdlib.{c|h},` which
    - provide driver for the LCD module
  - `preemptive.{c|h},` which is also a required file to compile the project
    - `Makefile` for building the project.
- (Part 2) Turn in the complete source code for the dinosaur game you will make, the typescript for compiling your code using the updated `Makefile`, and your report as a PDF file.
  - `dino.c`
    - implements the dinosaur game as three threads. It initializes the system, peripheral devices, game, and runs the game.
  - `typescript`
    - a typescript for a clean make of the entire game.
  - `ppc5.pdf`
    - a writeup for this checkpoint, includes the screenshots and explanations as instructed below.

---

[1] A project programming checkpoint is a self-assessed assignment that does not count towards your assignment grade. You are not required to turn in anything; however, you are responsible for understanding the concepts for the subsequent lecture materials, assignments, and exams. Therefore, you are strongly urged to do it as a regular assignment and "grade yourself" according to the breakdown. You should be able to complete this entire checkpoint in one week, since plenty of hints are given; if not, you should ask for help and keep up, or else you will have a difficult time with the project. You will receive one score at the end of the semester for the programming project part of your grade.

For this programming project checkpoint, you are to first test the two-producer, one-consumer structure by replacing them with peripheral devices, and then part 2 uses some of the devices to make a small game.

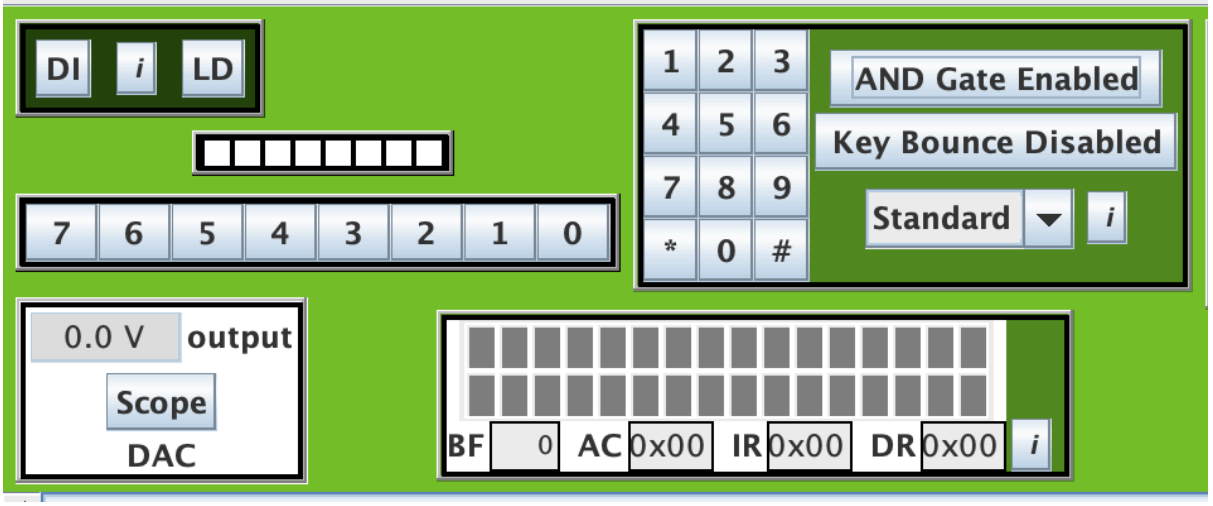
1. [30 points] `Peripheral devices`

Part 1 replace the two producers and the consumer with

(a) the button bank, shown as the row of [7][6][5][4][3][2][1][0] in the left part of screenshot below.

(b) the numeric keypad, in the upper right quadrant of the screenshot below. Be sure you click the top-right button so it shows [AND Gate **Enabled**]. (by default it is disabled).

(c) the LCD module, which is in the lower right. It can display 2 rows by 16 columns of text and some custom symbols.



To see how these peripherals are connected to the 8051 MCU, you can click on the [LD] button for the Logic Diagram. It is reproduced here for your convenience.

1.1 Button bank

Source files: (just for your own practice)

- `buttonlib.h`

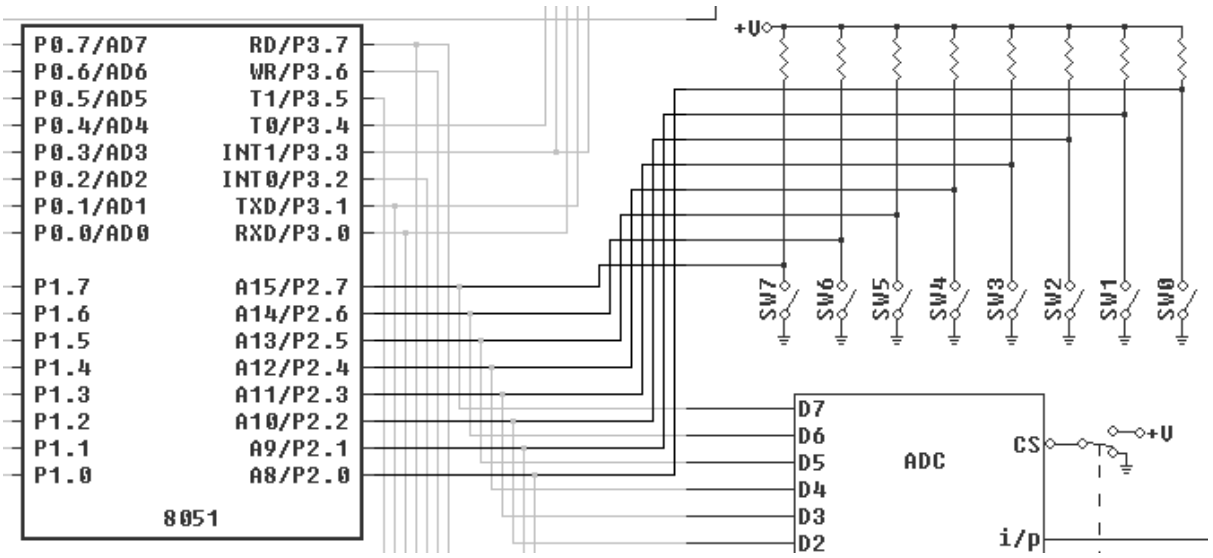
```
/*
 * This is the library for button bank.
 * it provides two functions:
 * one to check if any button is pressed,
 * and another to read the pressed button as ASCII; if multiple buttons
 * are pressed, then read the highest-priority button.
 */
```

```h
#ifndef __BUTTONLIB_H__
#define __BUTTONLIB_H__
char AnyButtonPressed(void);
char ButtonToChar(void);
#endif // __BUTTONLIB_H__
```

- buttonlib.c

```c
#include <8051.h>
#include "buttonlib.h"
/*
 * button library
 * for accessing the button bank in EdSim51
 * It is relatively easy because the buttons are tied to port P2.
 * When a button is pressed, it is 0; when released, it is 1.
 */
// returns true if any button is pressed. false if no button pressed.
char AnyButtonPressed(void) {
      return /* @@@ Your code here. returns true  if any bits of P2 is 0; false otherwise.. */;
}
// if one of the buttons is pressed, return the ASCII code for the
// highest number button pressed, while ignoring the second highest.
// but if none of the buttons is pressed, return the null '\0'
// character
char ButtonToChar(void) {
        if ((~P2) & 0x80) {
            return '7';
        } else if …
         /* @@@ Your Code here. depending on button state, return '6', '5', '4', … '0', or null if no key is pressed or if the combination is not valid.  */
        }
}
```

-

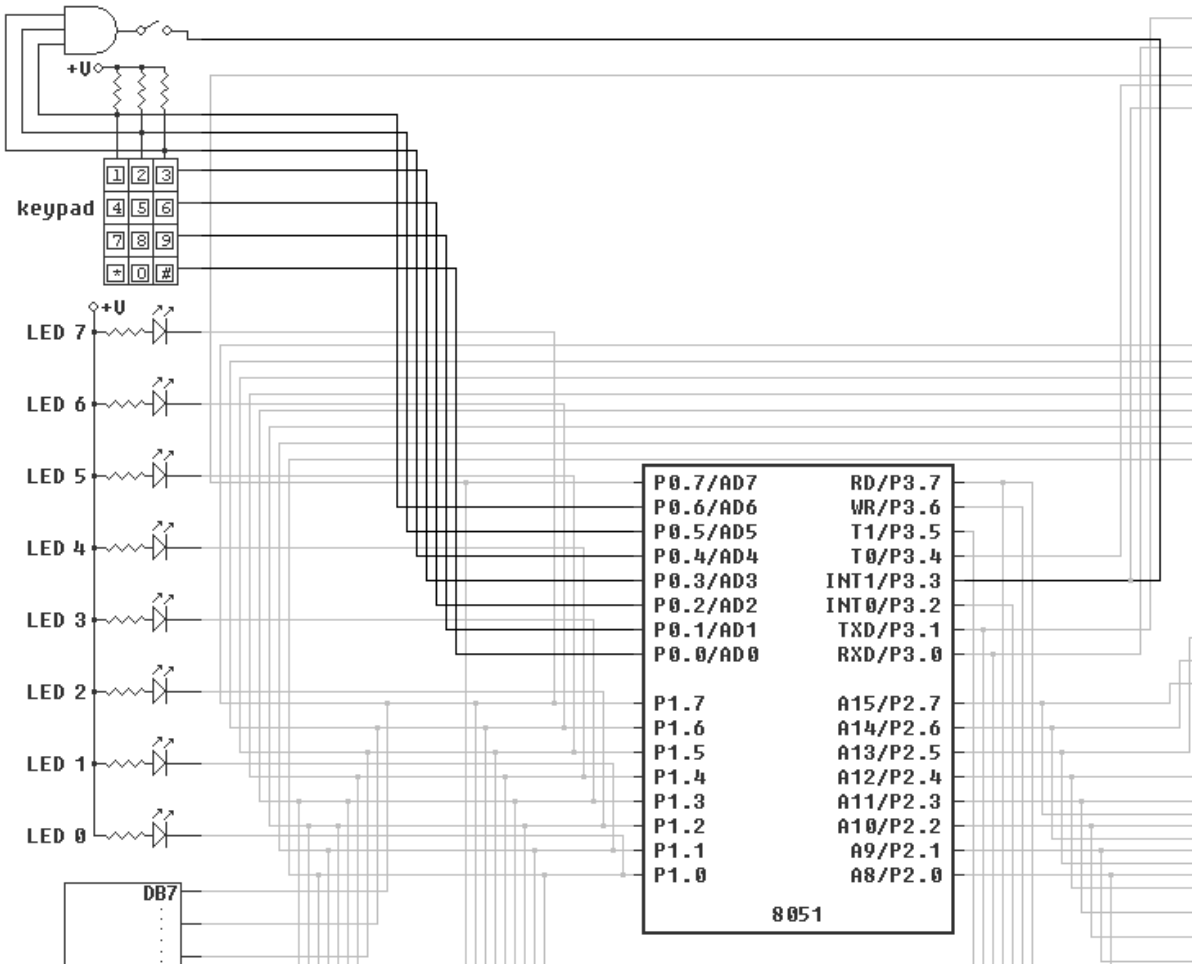


The button bank, also called the "switch bank" in EdSim51, consists of a bank of eight switches that connect a resistive pull-up side to the ground when pressed. By default, they are open (i.e., disconnected), so the pins of port `P2.0`, `P2.1`, … `P2.7` will read in a high (logical 1) value. If any button is pressed (i.e., the switch is "closed" or "connected"), then the corresponding I/O pin will read in a 0 value. So, you can use a single instruction to read in the state of the 8 switches together by reading `P2` using one instruction.

Note that in case multiple buttons are pressed, you can just return the highest numbered button; alternatively, you may also return the null character. It is your choice, and be sure to document accordingly. Since you will be writing your own game engine and can control how you press the buttons or keys, you just need to state your assumption for operation.

2.2 Keypad

A keypad is a matrix of these switches formed by connecting the columns together. In an *m*-column by *n*-row keypad, Instead of using *m* x *n* GPIO (general-purpose I/O) pins (as is the case with the button bank), it uses just *m*+*n* GPIO pins, plus one to detect if any key is pressed. So in this example, we have a numeric keypad with 12 keys (10 digits and the [*] [#] keys). Instead of using 12 GPIO pins, we need only 8 pins = 3 (columns) + 4 (rows) + one for "any pressed", which can fit in exactly one GPIO port. However, the hardware pin saving comes at the cost of extra work in software to scan one row at a time and check the columns



The rows [1][2][3] are connected to `P0.3` (output) when pressed;
The rows [4][5][6] are connected to `P0.2` (output) when pressed;
The rows [7][8][9] are connected to `P0.1` (output) when pressed;
The rows [*][0][#] are connected to `P0.0` (output) when pressed.

The value of column [1][4][7][*] can be read from pin `P0.6`. If any of these buttons is pressed and is connected to a 0 value, then the `P0.6` reads a 0; otherwise, it reads a 1. Similarly, the column [2][5][8][0] can be read from pin `P0.5`, and the column [3][6][9][#] can be read from pin `P0.4`.
To read from a GPIO port, you need to set it to 1 (pull-up).

(Hint) SDCC syntax for pin `P0.3` is `P0_3`, as defined in `#include <8051.h>`.

In addition, pin `P3.3` can read from the output of the AND gate such that if any button is pressed and connected to 0, then it can read a 0; otherwise, it reads a 1 value. Also note that `P3.3` can also be configured as the `INT1` (interrupt 1) line, so it can work as an interrupt instead of getting polled.

Source files: (all provided so you can just try running with your own code)

- keylib.h

```h
#ifndef __KEYLIB_H__
#define __KEYLIB_H__
void Init_Keypad(void);
char KeyToChar(void);
char AnyKeyPressed(void);
#endif // __KEYLIB_H__
```

- keylib.c

```c
/*
 * keylib.c
 * This is the library that works with the keypad.
 * it provides functions for
 * - initialize
 * - check any key pressed
 * - read the pressed key; in case of multiple, read highest priority
 */
#include <8051.h>
void Init_Keypad(void) {
	P3_3 = 1; // input mode from AND gate.
		// be sure to ENABLE the AND gate in Edsim51
	P0 = 0xf0; // configure column 3 bits (top) as input,
		// columns as pull-down.
		// Although we don't use the top bit for a column,
		// we set it to 1 to be safe.
}
/*
 * boolean to quickly check if any key is pressed
 */
char AnyKeyPressed(void) {
	P0 = 0xf0;  // set all rows to pull-down
	return !P3_3; // true if any button is connected to pull-down
}
/*
 * read the keypad, return ASCII character if pressed. in case of
 * multiple keys, read the highest priority one.  If no keys pressed,
 * return null character.
 */
char KeyToChar(void) {
	P0 = 0xf7; // test the top row
	if (P0 == 0xb7) { return '1'; }
	if (P0 == 0xd7) { return '2'; }
	if (P0 == 0xe7) { return '3'; }
	P0 = 0xfb; // test the next row
	if (P0 == 0xbb) { return '4';}
	if (P0 == 0xdb) { return '5'; }
	if (P0 == 0xeb) { return '6'; }
	P0 = 0xfd;  // test the 3rd row
	if (P0 == 0xbd) { return '7';}
```

```
        if (P0 == 0xdd) { return '8'; }
        if (P0 == 0xed) { return '9'; }
        P0 = 0xfe;  // test the last row
        if (P0 == 0xbe) { return '*'; }
        if (P0 == 0xde) { return '0'; }
        if (P0 == 0xee) { return '#'; }
        return 0;
}
```

-

2.3 LCD

The LCD module is more complex, since it has a controller and interprets commands. Here we provide the source code for you to try.

Source files: (all provided so you can just try running your own code)

- `lcdlib.h`

```
#ifndef __LCDLIB_H__
#define __LCDLIB_H__
#define CLEAR_DISPLAY 1
#define RETURN_HOME   2
#define DEC_CURSOR    4
#define INC_CURSOR    6
#define SHIFT_DISPLAY_RIGHT 5
#define SHIFT_DISPLAY_LEFT  7
#define DISPLAY_OFF_CURSOR_OFF 8
#define DISPLAY_OFF_CURSOR_ON  0xA
#define DISPLAY_ON_CURSOR_OFF  0xC
#define DISPLAY_ON_CURSOR_BLINK 0xE
#define SHIFT_CURSOR_LEFT  0x10
#define SHIFT_CURSOR_RIGHT 0x14
#define SHIFT_ENTIRE_DISPLAY_LEFT 0x18
#define SHIFT_ENTIRE_DISPLAY_RIGHT 0x1C
#define FORCE_CURSOR_LINE_1_HEAD 0x80
#define FORCE_CURSOR_LINE_2_HEAD 0xC0
#define DISP_2_LINE_5x7_FONT  0x38
void LCD_Init(void);  /* initialize the LCD module */
void LCD_IRWrite(char c); /* writes to the instruction register */
#define LCD_returnHome() \
       LCD_IRWrite(RETURN_HOME)
#define LCD_clearScreen() \
       LCD_IRWrite(CLEAR_DISPLAY)
#define LCD_entryModeSet(id, s) \
       LCD_IRWrite(0x4 | ((id) << 1) | (s))
#define LCD_displayOnOffControl(display, cursor, blinking) \
       LCD_IRWrite(0x8 | ((display)<<2) | ((cursor) <<1) | (blinking))
```

```
#define LCD_cursorOrDisplayShift(sc, rl) \
       LCD_IRWrite(0x10 | ((sc) << 3) | ((rl) << 2))
#define LCD_setDdRamAddress(addr) \
       LCD_IRWrite(0x80 | (addr))
#define LCD_setCgRamAddress(addr) \
    LCD_IRWrite(0x40 | (addr))
// row, column
// row = 0,1, column = 0..f
#define LCD_cursorGoTo(row, col)\
       LCD_setDdRamAddress(((row)*0x40+(col)))
void LCD_functionSet(void) ;
void LCD_write_char(char c) ;
void LCD_write_string(char *s);
unsigned char LCD_ready(void);
#endif // __LCDLIB_H__
```

- Usage:
  - call `LCD_Iniit()` to initialize the LCD module.
  - Pretty much every command can be expressed by calling `LCD_IRWrite()` with different parameters. So, we define commands as macros in the .h file.
  - Call `LCD_ready()` to check if the LCD is ready, since it is a slow device.
- `lcdlib.c`

```
#include <8051.h>
#include "lcdlib.h"
__data __at (0x3A) unsigned char lcd_ready;  /* @@@ change to a different
location if needed. It just needs a bit, no need to be a char. */
#define DB7 P1_7
#define DB6 P1_6
#define DB5 P1_5
#define DB4 P1_4
#define DB  P1
#define RS P1_3
#define E P1_2

void delay(unsigned char n) ;
#define DELAY_AMOUNT 40
unsigned char LCD_ready(void) {
    return lcd_ready;
}
void LCD_Init(void) {
       LCD_functionSet();
       LCD_entryModeSet(1, 1); /* increment and no shift */
       LCD_displayOnOffControl(1, 1, 1); /* display on, cursor on and
blinking on */
       lcd_ready = 1;
}
void LCD_IRWrite(char c) {
       lcd_ready = 0;
```

```
	DB = (c & 0xf0); // high nibble, keep RS low
	E = 1;  // pulse E
	E = 0;
	DB = (c << 4); // low nibble, keep RS low
	E = 1;
	E = 0;
	delay(DELAY_AMOUNT);
	lcd_ready = 1;
}
void LCD_functionSet(void) {
	lcd_ready = 0;
	// The high nibble for the function set is actually sent twice
	// because this is how 4-bit mode works for the HD44780 controller.
	DB = 0x20;  // DB<7:4> = 0010, <RS,E,x,x>=0
	E = 1;
	E = 0;
	delay(DELAY_AMOUNT);
	E = 1;
	E = 0;
	delay(DELAY_AMOUNT); // added, to ensure sufficient delay
	DB7 = 1; // 2-line model
	// DB6 defaults to 0 = 5x7, DB5, DB4 are don't-cares
	E = 1;
	E = 0;
	delay(DELAY_AMOUNT);
	lcd_ready = 1;
}
// ---------------------------------------------------------------------
void LCD_write_char(char c) {
	lcd_ready = 0;
	DB = (c & 0xf0) | 0x08; //; keep the RS
	RS = 1;
	E = 1;
	E = 0;
	DB = (c << 4) | 0x08; // keep the RS
	E = 1;
	E = 0;
	delay(DELAY_AMOUNT);
	lcd_ready = 1;
}
void LCD_write_string(char* str) {
	while (*str++) {
		LCD_write_char(*str);
	 }
}
void delay(unsigned char n) {
	{ __asm
	dhere:
		djnz dpl, dhere
	__endasm;
```

```
        }
}
```

- This implementation uses inlined assembly that decrements the DPL (register for the first parameter) to a delay function to achieve about 40ns of delay required for the LCD. If you are clearing the screen or put cursor home, the delay should be more like 1.5 ms.

2.4 `testlcd.c`

- You should write your own code where one producer reads from the button bank (by calling `ButtonToChar()`) and the other producer reads from the keypad (by calling `KeyToChar()`) and enqueue the ASCII code. They should keep track of whether the button or key has just made a transition from pressed to released and vice versa. It should just enqueue one character on each press, but not repeat enqueuing another character until all keys or all buttons have been released. However, keypads and button banks work independently.
- The consumer just obtains any key that has been enqueued and displays it to the LCD by calling the `LCD_write_char(char c)` function. Note that it cannot write another character or command (anything that calls `LCD_IRWrite()`) until `LCD_ready()` returns true.
- Adjust your `main()` program if necessary to run these three new threads instead.
- Update your `Makefile` as needed to compile your code. Here is an example:

```
#
# makefile for testing cooperative multithreading
#
CC = sdcc
CFLAGS = -c  --model-small
LDFLAGS =
LCD_OBJ = testlcd.rel preemptive.rel lcdlib.rel buttonlib.rel keylib.rel

all: testpreempt.hex testlcd.hex

testlcd.hex:   $(LCD_OBJ)
       $(CC) $(LDFLAGS) -o testlcd.hex $(LCD_OBJ)

clean:
       rm *.hex *.ihx *.lnk *.lst *.map *.mem *.rel *.rst *.sym
%.rel:  %.c    preemptive.h Makefile
       $(CC) $(CFLAGS) $<
```

- Note that `Makefile` indentation must use tabs (`'\t'`), not white space.

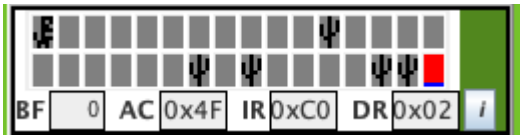
Be sure you can get your code working before moving on to the game.

2. [50 points] Dinosaur game

You are to implement a dinosaur game after your Part 1 is working..

How the game works:

- When the game first starts, the user types a digit (0-9), followed by the '#' key to set the level of difficulty, 0 being the easiest and 9 the most difficult. Be sure you release a key by pressing it again before pressing another key.
- You use the keypad as arrow keys to control a dinosaur character on the LCD. The dinosaur stays in the leftmost column. Pressing the [2] key moves it **up** a row, and [8] moves it **down** a row.

  in this screenshot, the dinosaur  is in the upper left corner, and there is one cactus on line 1, column 11;

  and four cacti on line 2, in columns 6, 8, 13, 14.

- Over time, cacti will be injected from the rightmost column (you may inject it from the 15th column to prevent the LCD screen from shifting right by the cursor), and they will shift to the left one column at a time after a delay, depending on the level of difficulty. If a cactus collides with the dinosaur (and kills it), then it is game over. However, by moving the dinosaur to a different row, you can avoid the collision and the dinosaur live. The score is incremented every time the dinosaur successfully dodges a cactus.
- When the game is over, refresh the display with "Game over" on the first row and the score in decimal on the second row.
- You may try this hex file to get a better idea of what you are going to implement. However, your demo should be built from your own source code.

Implementation hint:

You may follow our structure below, or you may have your own organization. In any case, it must use multiple threads rather than single-threaded code. The program needs to do the following steps:

- Initialization:
  - setup the threads and shared buffer(s)
  - initialize the devices
  - initialize the game state as global variables.
- spawn three threads:
  - keypad control: tracks the state of the keypad and produces keys

- rendering on the LCD: it repeatedly repaints the LCD to reflect the state of the game
- game control: it updates the state of the game by shifting the cacti, moving the dinosaur in response to keys, and checks if collision. If so, displays the score.

2.1 `keypad_ctrl` thread

This should be similar to the producer that tracks the button state for the keypad and writes the ASCII code to the shared buffer for the `game_ctrl` thread to interpret.

2.2 `render_task` thread

This thread is responsible for tracking commands from the `game_ctrl` thread and updates the displays content.

On initialization, it needs to define the bitmap images for the dinosaur and the cactus to the LCD's memory. You can use the following code:

```
// declare the bitmap data
const char dinosaur[] = {0x07, 0x05, 0x06, 0x07, 0x14, 0x17, 0x0E, 0x0A};
const char cactus[] = {0x04, 0x05, 0x15, 0x15, 0x16, 0x0C, 0x04, 0x04};
```

```
// to program the dinosaur and cactus bitmaps for the LCD to display
// make these two calls from your initialization function.
    LCD_set_symbol('\10', dinosaur); // bitmap for dinosaur starts at 0x10
    LCD_set_symbol('\20', cactus);   // bitmap for cactus starts at 0x20
```

```
// you can define the function with this type signature
void LCD_set_symbol(char code, const char symb[]) {
// or it could be defined as a macro.
// in any case, it takes the following calls:

    LCD_setCgRamAddress(code); // code is the character generation memory
              // (CG RAM address for the bitmap.
              //  write '\1' for the dinosaur, '\2' for the cactus.
    LCD_write_char(symb[0]);
    LCD_write_char(symb[1]);
    LCD_write_char(symb[2]);
    LCD_write_char(symb[3]);
    LCD_write_char(symb[4]);
    LCD_write_char(symb[5]);
    LCD_write_char(symb[6]);
    LCD_write_char(symb[7]);
   // you need to write all 8 bytes for each bitmap.
   // of course, you could do it in a loop or do pointer arithmetic
```

```
}
```

When the game starts, this `render_task` thread just repeatedly refreshes the display based on the `game_ctrl`'s state, which tracks where the dinosaur and the cacti are.

For rendering, you may use these display macros as needed:

- `LCD_returnHome()`
- `LCD_clearScreen()`
  ```
  /* note: this might not work if you don't delay long enough.  You may also
  just write blanks */
  ```
- `LCD_entryModeSet(id, s)`
  ```
  /* id = 1 means increment by 1 every time, id = 0 means don't increment;
   * s = 0 means no shift, s=1 means shift. Normally we do
   * LCD_entryModeSet(1,1)
   */
  ```
- `LCD_displayOnOffControl(display, cursor, blinking)`
  ```
  /* whether display, cursor, blinking should be on or off.*/
  ```
- `LCD_cursorOrDisplayShift(sc, rl)`
  ```
  /* sc controls whether cursor should shift or display should shift when
   * writing, and rl controls right shift or left shift.
   */
  ```
- `LCD_setDdRamAddress(addr)`
  ```
  /* this writes the address pointer in the display data (DD) memory, so your
   * next write will go to that address.
   */
  ```
- `LCD_setCgRamAddress(addr)`
  ```
  /* this writes the address pointer in the character generation (CG) memory,
   * so your next write will go to that address.
   */
  ```
- `LCD_cursorGoTo(row, col)`
  ```
  /* moves the cursor to the row and column, 0 based.*/
  ```

2.3 `game_ctrl` thread

This thread can be responsible for

- initializing the game state, including the score, the level of difficulty, dinosaur's row, the "`map`" of current cacti and their positions, etc.
- adding cacti from the right and shifting them to the left each time
- moving the dinosaur on every valid key press
- update the score when successfully avoiding a collision
- end the game and display the score when the game is over.

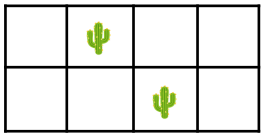
Questions:

- What data type do you use for the `map`? Hint: you don't have a lot of memory! (and you don't need a lot) There are 16 x 2 = 32 positions where a cactus may be present.

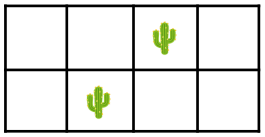
However, we are very memory constrained and don't have 32 bytes. You should consider using bits instead of chars for the `map`.

- How do you generate a new cactus? It does not have to be a complicated policy. It can be just based on some regular interval, but it should be possible for the player to avoid collision. In other words, you shouldn't have
  - two cacti in the same column (flat wall),

  - or two cacti on different rows but only one column apart from each other, because that would be a 45-degree wall.



  -



  -

- The level of difficulty can be controlled based on the shifting speed.

2.4 Deal with race condition

What variables may have race conditions and why?

3. [20 points] Typescript and screenshots

Turn in a typescript showing compilation of your code [2 points], screen shots and explanations for all the parts above [18 points].

Appendix: Debugging Hints

A. Source-level Debugging

Without a source-level debugger, it would be helpful to show on a separate screen or print out the "`.rst`" file, which includes the code memory, machine code, assembly code, and the source code together. This way, when you step through the instructions in SDCC, you can tell what you are looking at. For example, the following is taken from my `buttonlib.rst` file after compiling and linking.

```
                                    284
;--------------------------------------------------------
                                    285 ;    buttonlib.c:12: char AnyButtonPressed(void) {
                                    286 ;    ---------------------------------------
                                    287 ;     function AnyButtonPressed
                                    288 ;    ---------------------------------------
      0006B8                        289 _AnyButtonPressed:
                           000007   290      ar7 = 0x07
                           000006   291      ar6 = 0x06
                           000005   292      ar5 = 0x05
                           000004   293      ar4 = 0x04
                           000003   294      ar3 = 0x03
                           000002   295      ar2 = 0x02
                           000001   296      ar1 = 0x01
                           000000   297      ar0 = 0x00
                                    298 ;    buttonlib.c:13: return (P2 != 0xff);
      0006B8 74 FF            [12]  299      mov     a,#0xff
      0006BA B5 A0 04         [24]  300      cjne    a,_P2,00103$
      0006BD 74 01            [12]  301      mov     a,#0x01
      0006BF 80 01            [24]  302      sjmp    00104$
      0006C1                        303 00103$:
      0006C1 E4               [12]  304      clr     a
      0006C2                        305 00104$:
      0006C2 B4 01 00         [24]  306      cjne    a,#0x01,00105$
      0006C5                        307 00105$:
      0006C5 E4               [12]  308      clr     a
      0006C6 33               [12]  309      rlc     a
                                    310 ;    buttonlib.c:14: }
      0006C7 F5 82            [12]  311      mov     dpl,a
      0006C9 22               [24]  312      ret
                                    313
;--------------------------------------------------------
```

Figure A.1 Excerpt from the `buttonlib.rst` file (yours may vary)

The green highlights the code-memory address, the red highlights the original source code as assembly comments (starts with ";"), and the blue highlights the assembly code corresponding to each source code line above it.

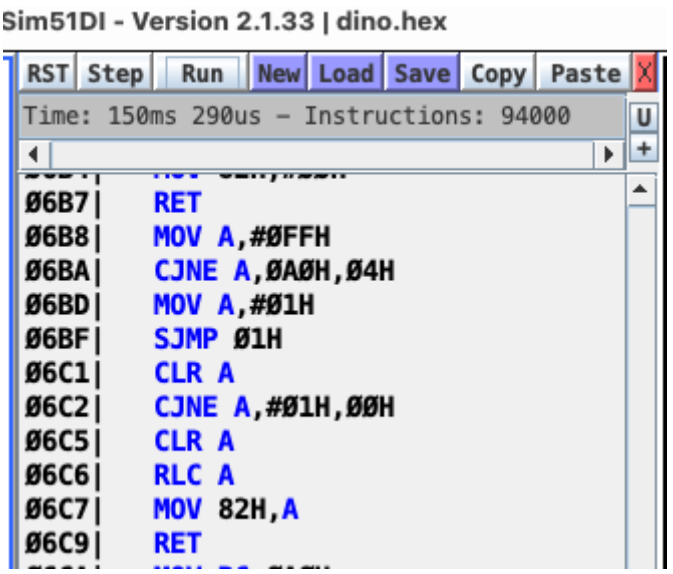


Figure A.2 Screenshot of the .hex file that shows the instructions corresponding to Fig. A.1

When the hex file is loaded into EdSim51, you can scroll down to the address and find that indeed address 06B8H has the instruction `MOV A, #0FFH`, which is Intel syntax for the same instruction `mov a, #0xff` in SDCC/Gnu syntax. This is a good way to mimic source-level debugging.

B. Absolute Registers (AR)

The most common issue may be with SDCC's use of "AR" (absolute registers) such as `ar7`, `ar6`, … In fact, in Figure A.1, you can see

```
000007   290      ar7 = 0x07
000006   291      ar6 = 0x06
```

which is just symbolic definitions for the name `ar7` to be (RAM, direct) address 0x7, `ar6` to be direct address 0x6, etc., when used in an instruction. Note that just having `ar` registers does NOT mean your code fails, but sometimes they may cause your code to behave strangely.

SDCC normally allocates a register (`r7`, `r6`, etc) for temporary or even auto-local variables and has no problems with `MOV` or `ADD` instructions, since they support many combinations of addressing modes. However, certain instructions are more restricted, specifically ANL ("AND logical" instruction), are much more restrictive and require direct addressing for the left source and target. So, SDCC may end up loading some value into `r7` but then uses it as `ar7` in an `ANL` instruction. This would be fine if we weren't switching register banks, but if we do, then the active bank's `r7` might not correspond to `ar7`. (if you use bank 0 then `r7` is the same as `ar7`) Also, **`switch`** statements sometimes also uses `ar` registers.

You could try to rewrite your C code (that is not running as thread 0) to avoid **`switch`** statements, auto-local variables, or logical/bitwise AND instructions that writes back to itself, especially in a temporary or auto-local. Of course, they do not always use ar registers, and you can inspect the `.rst` files to find out.

Two way to avoid auto-local are to

- allocate a global variable statically;
- reuse a parameter as a local variable. Since SDCC's convention is to use `DPL`, `DPH`, etc as the first and second **`char`** parameters (`DPTR` for 16-bit **`int`**), and both have direct addresses and are always safely preserved and restored during context switching, they may be a quick way to avoid `ar` registers.