Contents

Version 0.4

Debug messages are required.

V0.2: Wall messages and incidental prints.

V0.3: Using diff on the output.

V.04: Two **Note:** marked with highlight. (debugs and what output does not own)

V.05 Graphics is finally here

Output in Lab 2

Output comes in two types, text and graphical. The setting of the file **`debug.h`** will turn on and turn off different kinds of output. Extremely few non-output routines will care about the output mode; they will call into the output system and it will do output in all modes that are live. **Do not have two different simulations.**

Please organize your code so that most output is done via functions in the file that holds the output code. This means moving one-line and two-line printf calls to a function. This will facilitate adding graphics calls. It will make writing the code outside your output file easier since it pushes off the details. Please read the document "Thoughts on Lab 2 and Single Purpose" to see how to provide entry points into the output system.

Note: The following messages do *not* belong in output:

- When main prints runtime
- When input prints the final value it got back from scanf that terminated the input

Text Output

Please carefully align your output to match what is given here. The graders will use the linux **diff** command to compare your text output with the reference text output.

Outside of main, all printf statements must be protected:

```
if(TEXT)printf("\n");
if(DEBUG)printf("\n");
```

Debug statements are always printf statements and have their own section below.

Text mode printing needs to be protected, but you can do it at a function call level:

```
void master_output(double et, unsigned char bits, double ball[])
{
        if(TEXT)master_text(et, bits, ball);
}
```

All functions that get called by master_text do not need to check TEXT before printing.

Tabular Output

The excerpt below was generated off the data file xperfect.pbd. This does not include various messages. This output corresponds to step 1 of "simulating one ball" in the simulation document. We see the starting condition and the starting time in the first line. Note that time has 6 digits to the right of the decimal and can has as many as 2 to the left of the decimal. In the table below, the first character of the heading lines up over the decimal point of the number below it. The underscores tell you how wide each field should be. Your output should not shift or waver as numbers change. Note that VY is wide enough to accommodate a number such as -100.0. VX is wide enough to accommodate numbers such as -75.0, the highest value we should see for it. The hex code for status is printed with upper case letters.

The output below uses a single space between items that are part of a group. It uses 4 spaces between groups.

```
ST    ___X_____ ___Y_____    ___VX____ ____VY____    ET= 0.000000
BF     -0.40000   0.00000      6.40000    2.94078

ST    ___X_____ ___Y_____    ___VX____ ____VY____    ET= 0.015625
BF     -0.30000   0.04021      6.40000    2.20559

ST    ___X_____ ___Y_____    ___VX____ ____VY____    ET= 0.031250
BF     -0.20000   0.06892      6.40000    1.47039

ST    ___X_____ ___Y_____    ___VX____ ____VY____    ET= 0.046875
BF     -0.10000   0.08616      6.40000    0.73520

ST    ___X_____ ___Y_____    ___VX____ ____VY____    ET= 0.062500
BF     -0.00000   0.09190      6.40000    0.00000

ST    ___X_____ ___Y_____    ___VX____ ____VY____    ET= 0.078125
BF      0.10000   0.08616      6.40000   -0.73520

ST    ___X_____ ___Y_____    ___VX____ ____VY____    ET= 0.093750
BF      0.20000   0.06892      6.40000   -1.47039

ST    ___X_____ ___Y_____    ___VX____ ____VY____    ET= 0.109375
BF      0.30000   0.04021      6.40000   -2.20559

ST    ___X_____ ___Y_____    ___VX____ ____VY____    ET= 0.125000
BF      0.40000   0.00000      6.40000   -2.94078
```

Messages

Various events will call for messages to be printed. Your code needs to produce all of the messages below. (The first two can help debug the input sequence.)

```
Loaded FF 7 ball at  -0.40000,   0.00000  24.67866 deg   7.04331 ips
Launched BF 7 ball at  -0.40000,   0.00000 at   6.40000,   2.94078
```

The above messages are the load & launch messages respectively. 7 is the ball color, which comes from the bitfield. The load message angle and velocity in polar coordinates and the launch message has VX and VY. The FF status code that was read in was converted to BF when the ball status was set to on the table.

```
Off table: BF      0.50000  -0.05169      6.40000   -3.67598
```

The off table message adds some characters in front of the same print used in the tabular output.

```
Right wall: BF     12.10485  48.08188     35.35534   33.88495
Upper wall: BF     11.89515  48.08188    -33.58757   32.19070
Left_ wall: BF    -12.03600  11.74872    -31.90819  -65.87057
```

Note that the left wall message has an extra character to get it to line up with the other wall messages. These messages use the data *before corrections are applied.* The regular printing will show the corrected data. The above six messages should have their code in the file with output code.

Incidental text output

When scanf fails to do all five conversions, print the return value (only when in text mode). This print can live in input code file and need not be done in the output code file. Do **not** hard code a -1 in this print. There will be data files where scanf returns other values.

```
Final scanf call returned -1
```

In text mode and in graphics mode, main will need to print the total runtime. The time will vary from run to run. Graphics mode runtimes should be quite close to each other because they are fixed to a particular frame rate.

```
Total runtime is 0.000916481 seconds
```

Debug Statements

Debug statements cannot be for required output. Debug statements can be turned off by manipulating the debug header file while still leaving on required text output. Debug statements do **not** count towards the ten-line limit. Put them in your code to find out what it is thinking and then turn them all off with a single change in the debug header file.

A more detailed handling of debug statements is given in "Doing labs in this class."

You are **required** to fully debug the bit field manipulation code. In other words, when DEBUG is true, that file should tell you everything about what is going on in those routines. The output should allow you to prove that they work. This might mean printing the parameters, possibly some intermediate values, and outputs in a way that lets you know it worked.

It is a great idea to spread debug messages anywhere you think bugs might be or in any code you don't fully trust.

Note: Unless you are debugging output code, debug messages go in the function they are debugging. Output doesn't want to know the details of various functions.

Checking your output using diff

Here is how to use the diff command to see how your output differs from the reference output. We run the lab, redirecting input from the xperfect.pbd file. We redirect the output to my.experfect.out. Then we see how that differs from the reference output using the diff command. They have different runtimes and that is fine.

```
[kirby.249@cse-sl1 lab2]$ lab2 < xperfect.pbd > my.xperfect.out
[kirby.249@cse-sl1 lab2]$ diff my.xperfect.out xperfect.output.txt
32c32
< Total runtime is 0.000163555 seconds
---
> Total runtime is 0.000132084 seconds
[kirby.249@cse-sl1 lab2]$
```

You may need to use the -w flag on diff so that it ignores whitespace differences:

```
diff -w my.xperfect.out xperfect.output.txt
```

Graphical Output

Graphics is a 2-point bonus in lab 2.

The reference code will have it.

Graphical output is achieved with the help of the pinball library. The basic sequence for graphics is as follows:

- At the beginning of the program, call **`pb_initialize`** and pay attention to the return value. Do this exactly one time. If it works, your code is allowed to do grahpics:
  - Each time your code has a new frame to draw, it calls **`pb_clear`**
    - All the different parts of your code that need to draw things in this frame make various **`pb_`** calls as many times as they need to get everything on the screen. In lab3 there will be more than one set of bits onscreen at a time, so there will be multiple **`pb_ball`** calls in each frame. The other call to make is to **`pb_time.`**
  - After all of the drawing calls have been made, **`pb_refresh`** outputs the new frame.
  - To give the humans time to see the frame (and to keep the IO system from optimizing everything away), a call to **`microsleep`** causes the code to pause.
- At the very end, if your code succeeded with **`pb_initialize`**, a call to **`pb_teardown`** is made to restore your screen.

All the functions mentioned above are in the libpb.h header file.

Only pb_initialize and pb_teardown may be called outside of the file output.c.

The Functions and the Constants

Here are the contents of the header file libpb.h and n2.h:

```
/* Neil Kirby */

int pb_ball(int color, double X, double Y);
void pb_beep();
void pb_clear();
void pb_flash();
```

```
int pb_initialize();
void pb_refresh();
void pb_status(const char *statstr);
void pb_teardown();
void pb_time(int milliseconds);/* Copyright 2022 Neil Kirby */

/* n2.h  Not for disclosure without permission */

int microsleep(unsigned int microseconds);
int millisleep(unsigned int milliseconds);
double now();
```

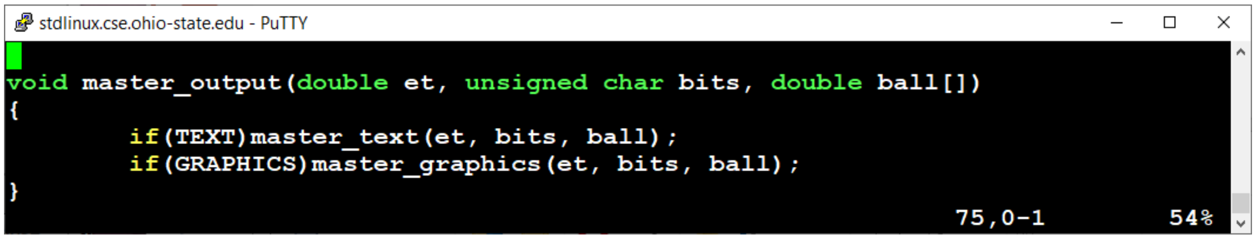
Here are the contents of the file constants.h:

```
#define DELTA_T (1.0/32.0)
# define M_PI           3.14159265358979323846  /* pi */
```

Framerate and DELTA_T

The file constants.h gives us the size of the time step in our simulation. Our simulation runs at 64 frames per second, so we wait 1.0/64.0 seconds between each frame. We use **`microsleep`** to wait for a time smaller than one second so we will need to multiply DELTA_T by 1,000,000 to turn seconds into micro-seconds.

Protecting Graphics Functions



stdlinux.cse.ohio-state.edu - PuTTY

```
void master_output(double et, unsigned char bits, double ball[])
{
        if(TEXT)master_text(et, bits, ball);
        if(GRAPHICS)master_graphics(et, bits, ball);
}
                                                    75,0-1         54%
```

Note the use of GRAPHICS, which is in debug.h, to prevent graphics routines from being called when not in graphics mode.

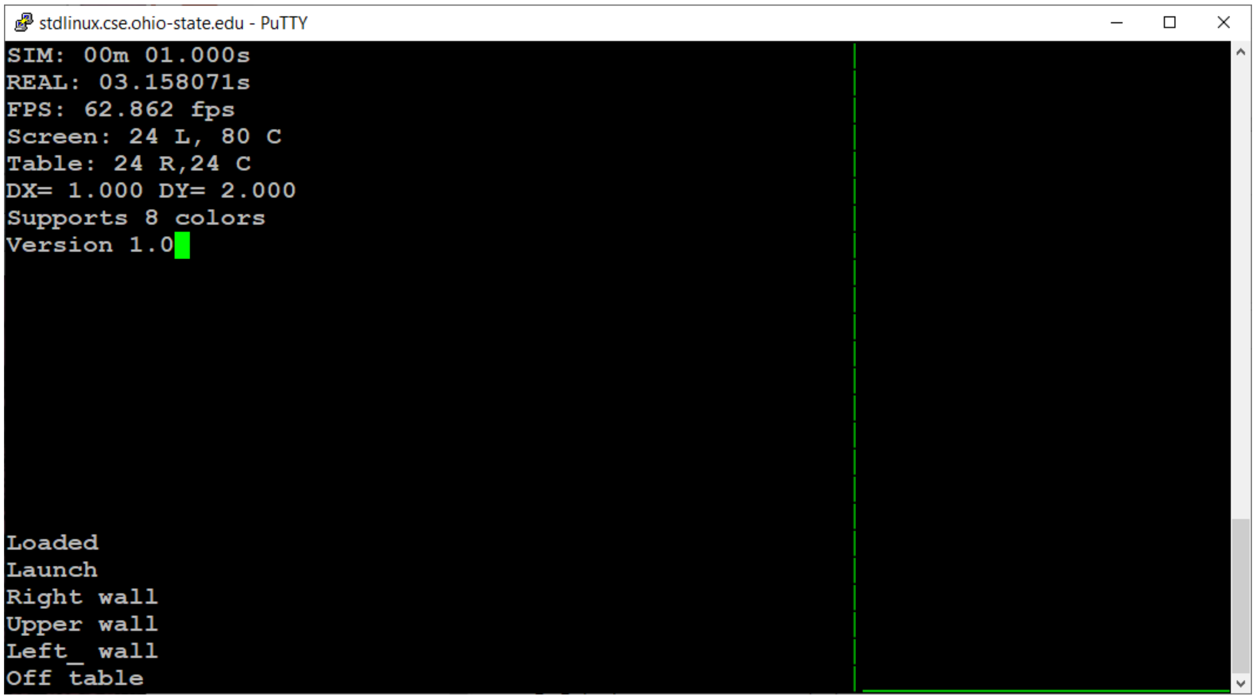
Consider the code below that belongs in main:

```
if( TEXT || ( GRAPHICS && pb_initialize()) )
```

This uses short circuit evaluation along with TEXT and GRAPHICS and it give a perfect line to decide if the simulation initialized cleanly. When in text mode it always succeeds. Only in graphics mode does it make the library call and that call determines if it succeeded.

Never presume that TEXT and GRAPHICS are the Boolean negation of each other. If you code needs to know about text mode, it should only look at TEXT. In later labs it may be possible that changes to debug.h make both true at the same time.

The Various Messages

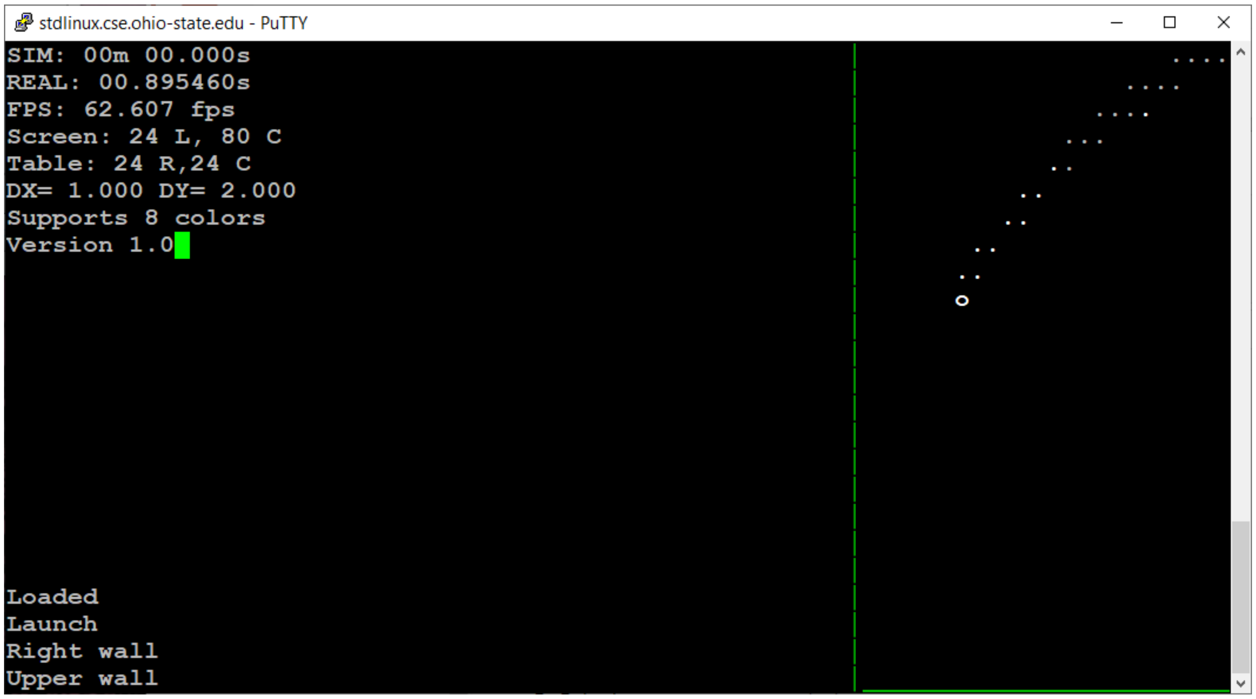


```
SIM: 00m 01.000s
REAL: 03.158071s
FPS: 62.862 fps
Screen: 24 L, 80 C
Table: 24 R,24 C
DX= 1.000 DY= 2.000
Supports 8 colors
Version 1.0









Loaded
Launch
Right wall
Upper wall
Left_ wall
Off table
```

The above screenshot is after the ball leaves the table. It shows the load, launch and wall messages in the lower left as well as the off table message. All of them are done with code such as:

```
pb_status("Right Wall");
```

The Ball In Play



```
SIM: 00m 00.000s
REAL: 00.895460s
FPS: 62.607 fps
Screen: 24 L, 80 C
Table: 24 R,24 C
DX= 1.000 DY= 2.000
Supports 8 colors
Version 1.0











Loaded
Launch
Right wall
Upper wall
```

In the above screenshot, the bright **`o`** is placed there with calls to **`pb_ball`**. The trail of dots following along are provided by the library to help you see the motion better.

The fps in the above capture shows various bits of telemetry generated by the library. Your code needs to call **`pb_time`** and pass it the elapsed time after doing a units and type conversions. Your code keep elapsed time in a double in seconds, the library wants an integer number of milliseconds. The factor of 1,000 used to convert seconds to milliseconds in *not* magic.

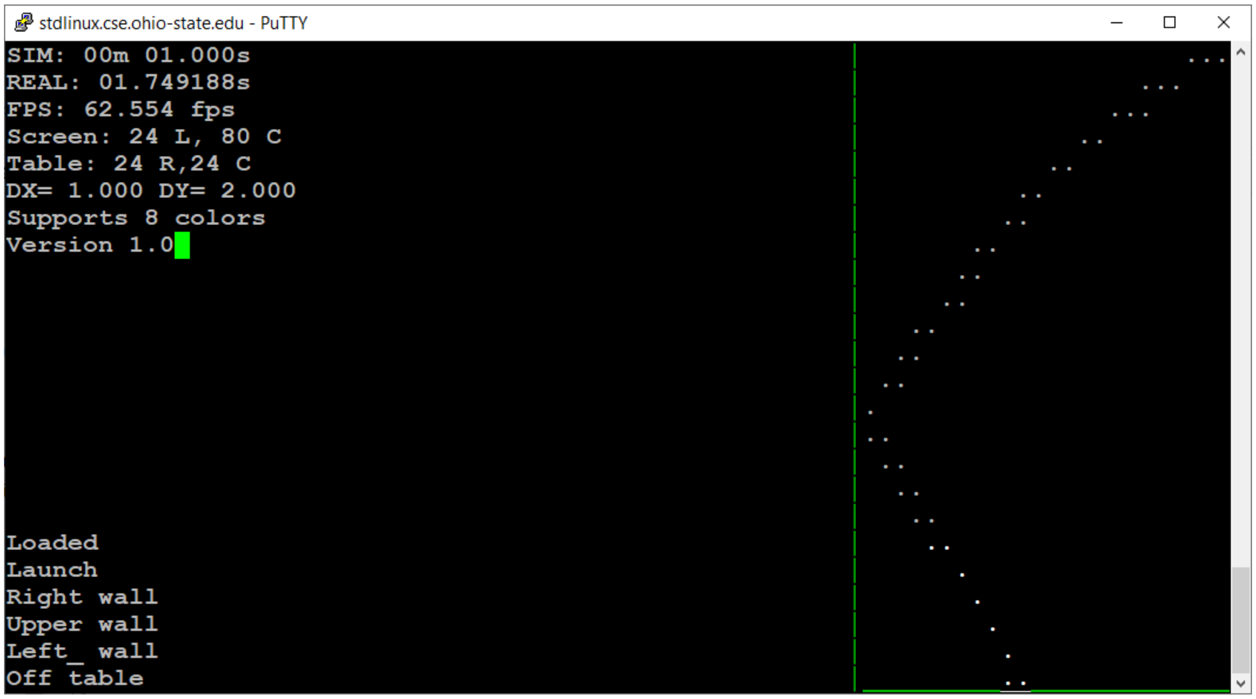
Once again, we employ **`pb_ball`** and **`pb_time`**, to draw everything we want. As always, graphical output starts with a call to **`pb_clear`** and ends with calls to **`pb_refresh`** and then **`microsleep`**.

Off table

When Y goes negative, the loop terminates and the off table output is called for.

- Use **`pb_status`** to give the off table message
- Create a loop that calls **pb_clear**, **pb_time**, and **pb_refresh** followed by **microsleep** for the normal sleep time. Run the loop until 4 seconds have passed this way. This allows the trail of dots of burn out on schedule.

An example is shown below:

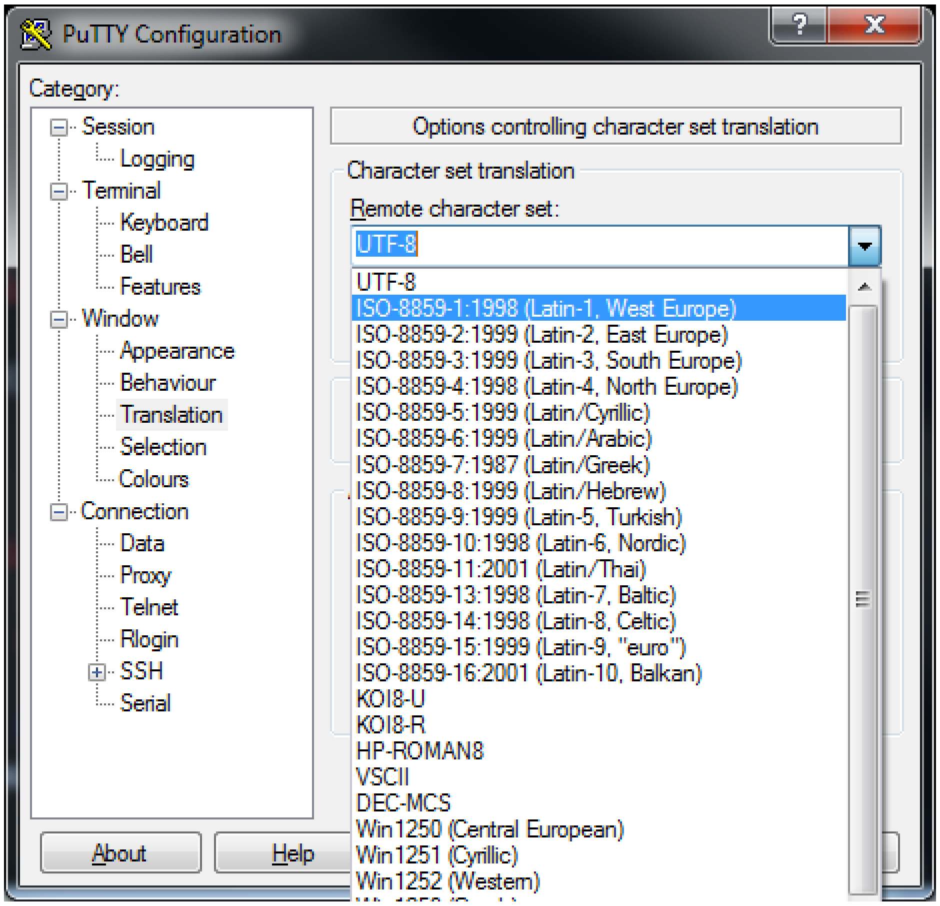


```
SIM: 00m 01.000s
REAL: 01.749188s
FPS: 62.554 fps
Screen: 24 L, 80 C
Table: 24 R,24 C
DX= 1.000 DY= 2.000
Supports 8 colors
Version 1.0







Loaded
Launch
Right wall
Upper wall
Left  wall
Off table
```

A guided tutorial for using the library

Get the zip file from piazza and unzip it to your folder. Take a look at the header file to know what capabilities the library offers. There are makefile samples in the text that follows. Almost all of the library calls start with pb_ in the function name. None of your functions should start with "pb_" unless you are using test code to be replaced.

If you are using PuTTY to connect, Go to the Window->Translation part of the configuration dialog and change from UTF-8 to Latin so that you get the right characters.


PuTTY Configuration
Category:
Session
Logging
Terminal
Keyboard
Bell
Features
Window
Appearance
Behaviour
Translation
Selection
Colours
Connection
Data
Proxy
Telnet
Rlogin
SSH
Serial
About
Help
Options controlling character set translation
Character set translation
UTF-8
ISO-8859-1:1998 (Latin-1, West Europe)
ISO-8859-2:1999 (Latin-2, East Europe)
ISO-8859-3:1999 (Latin-3, South Europe)
ISO-8859-4:1998 (Latin-4, North Europe)
ISO-8859-5:1999 (Latin/Cyrillic)
ISO-8859-6:1999 (Latin/Arabic)
ISO-8859-7:1987 (Latin/Greek)
ISO-8859-8:1999 (Latin/Hebrew)
ISO-8859-9:1999 (Latin-5, Turkish)
ISO-8859-10:1998 (Latin-6, Nordic)
ISO-8859-11:2001 (Latin/Thai)
ISO-8859-13:1998 (Latin-7, Baltic)
ISO-8859-14:1998 (Latin-8, Celtic)
ISO-8859-15:1999 (Latin-9, "euro")
ISO-8859-16:2001 (Latin-10, Balkan)
KOI8-U
KOI8-R
HP-ROMAN8
VSCII
DEC-MCS
Win1250 (Central European)
Win1251 (Cyrillic)
Win1252 (Western)

Remote character set: UTF-8

Step zero: Before calling anything, add to your makefile so that it brings along the two libraries and cleanly compiles. Here "p1" means "prototype one."

```
# A typical prototype.  But use a better name.
# Be sure to mark it as to be graded if that is the case

p1: p1.o
        #link everything together to get p1 the executable
        gcc -g -o $@ $^ -L. -lpb -lncurses
```

Note the items in the line in the makefile that links everything:

```
-L.
    means search . for libraries as well as the usual places.  Our
    libpb library is in . (the current folder)
```

```
-lpb
     means link in the libpb.a file – it needs to be in this folder

-lncurses
     means link in the new curses library (it will be found in the
     usual place that standard libraries are found)
```

Use this sample p1.c code:

```
#include "libpb.h"
#include "n2.h"
main() {}
```

If that fails to compile or link, fix that first before trying to touch the library. In other words, if you can't build, don't bother writing more C code.

Step one:

Now that you can compile against the library, start using it. Read the libpb.h file to find the functions that are available to you. See also Supplied Library section below here.

The first two functions to find are pb_initialize and pb_teardown. **If you call pb_initialize your code must check the return value!** (So put it in an if statement, because if it fails you aren't doing graphics.) After you make a successful call to pb_initialize, your code really needs to call pb_teardown before it exits or your terminal may be left in a bad state. (See the fix below if that happens.)

This line of code may be handy:

```
if( TEXT || ( GRAPHICS && pb_initialize()) )
```

By themselves, init and teardown don't do anything exciting. But we need them to get graphics going and to tear it down when done. So the next function on your list is pb_refresh. This function tells the system to output everything that has been drawn since the last refresh call. Your code will want to call it every time you want to make updates visible. Do all of your drawing for one update cycle before calling refresh. In lab 2 we only have a single call to do drawing since we only have one process to simulate.

So the first experiment is to use these functions:

```
pb_initialize()   /* only do the rest of the code if that returned a true value! */
pb_refresh()      /* do something visible */
getchar()         /* make it wait so we get to see it */
pb_teardown()     /* return the terminal to the normal state. */
```

You need to call getchar() to make the program wait on input from you. Press any key to go on. You will know it works when you see it take control of your screen.

```
pb_initialize();  /* get curses running */
```

If your terminal is left in a screwed up state blindly press enter and type the following command and press enter again:

```
stty sane
```

If your code crashes without a graceful exit, you will want to know that command. Another handy command is:

```
clear
```

Step two: Your next experiment is to get ready to draw something. Clone your experiment one code and add the following call just before the pb_refresh() call

```
pb_clear();        /* show a green field and sky */
```

We use this call to clear the drawing area, something we will need to do at the start of each simulation step before we output our newly updated process to its new position. If you forget to call it, we will draw over top of everything we have previously drawn. So in regular use, we will call pb_clear and then draw all of our balls. We still need to draw a ball.

Step three: Step two gave us a nice backdrop, but we want to show things! Clone your step two code. Just before the pb_refresh call add:

```
pb_ball(3, 7.0, 12.0);
```

Integration: About now you should either make any last prototypes needed to get comfortable with the clear / draw / refresh sequence needed to do the output you need. In the experiments we use getchar() to pause the program. In your lab 2 submission you will call microsleep with a value of deltaT converted to microseconds. When debugging, if you have to, you can use a larger number to slow things down or a smaller number to make them go faster.