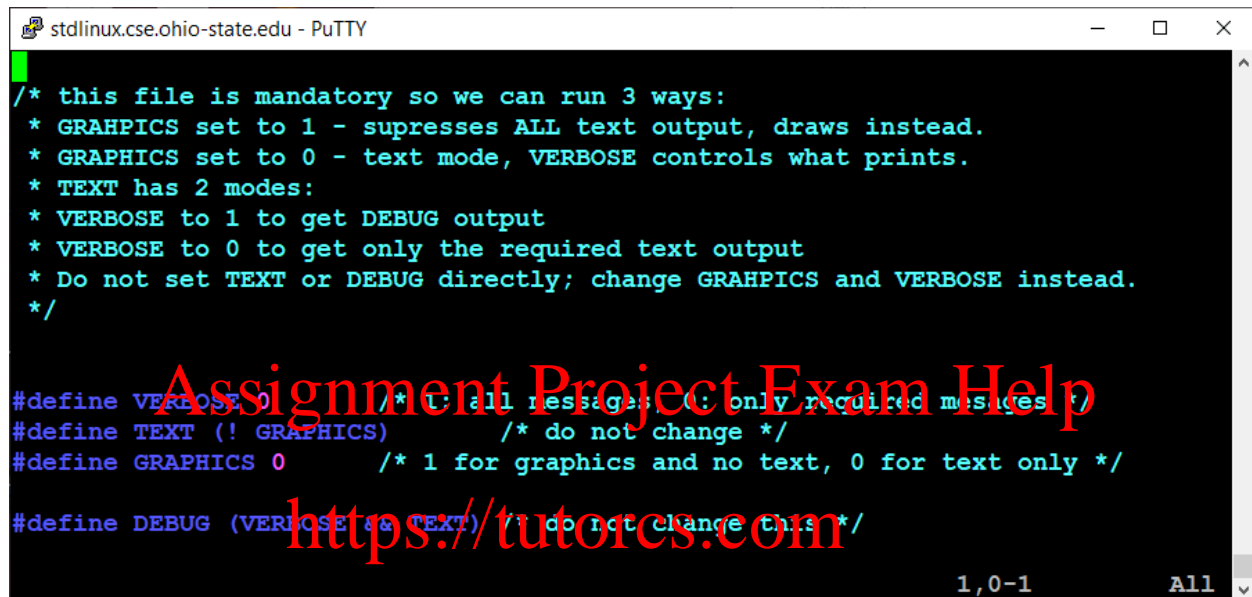


Output

Output comes in two types, text and graphical. The value of the GRAPHICS and VERBOSE defines in 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.** The `debug.h` file is on piazza but here is what is in it:



```
/* this file is mandatory so we can run 3 ways:
 * GRAPHICS set to 1 - supresses ALL text output, draws instead.
 * GRAPHICS set to 0 - text mode, VERBOSE controls what prints.
 * TEXT has 2 modes:
 * VERBOSE to 1 to get DEBUG output
 * VERBOSE to 0 to get only the required text output
 * Do not set TEXT or DEBUG directly; change GRAPHICS and VERBOSE instead.
 */

#define VERBOSE 0 /* 1: all messages, 0: only required messages */
#define TEXT (! GRAPHICS) /* do not change */
#define GRAPHICS 0 /* 1 for graphics and no text, 0 for text only */

#define DEBUG (VERBOSE && TEXT) /* do not change this */
```

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

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 shown above would not need to check TEXT before printing.

Your text output must be identical to the reference output (other than the numbers in the total runtime message) to receive full points. Use shell redirection and then the diff command to see. The two commands below show how this is done.

```
[kirby.249@cse-s12 lab2]$ lab2 < xA.mm > xA.myout
[kirby.249@cse-s12 lab2]$ diff xA.myout xA.text
1449c1449
< Total runtime is 0.001295805 seconds
---
> Total runtime is 0.002854347 seconds
[kirby.249@cse-s12 lab2]$
```

Tabular Output

Here are the first few lines of output for the xA.mm input file. You have this output as the xA.text file.

```
C  S      X      Y      D      ET= 0.000000
7  1      2.000000, 20.000000 120
C  S      X      Y      D      ET= 0.031250
7  1      2.027063, 19.984375 120
```

There are no tabs in this output, only spaces. Those spaces can come from spaces in the format strings or when the printed field is smaller than the minimum field width for a field. The first column is color, the second is speed. These are followed by the X and Y values, with the X and Y in the heading lined up over the decimal place in the numbers. The D column shows direction. The heading shows elapsed time. As the simulation runs, the decimals must remain lined up (all columns must remain on their spacing).

Wall Messages

When an object would hit the wall if it moved, a text output message is generated. Here is the first of two such messages in the xA.text output, shown using a small font to keep text from wrapping:

```
At elapsed time 11.000000:
Smile 7 at 11.4992161, 14.5156250 speed 1 direction 120 will hit the wall.
```

The number 7 is the color number, white is color 7.

Final output

After the last wall message, the program will attempt to read and fail to do so. At this point the final output will be generated. The last lines of the xA.text file are shown below in a small font to keep lines from wrapping.

```
At elapsed time 4.000000:  
Projectile 3 at 12.5312500, 13.1259234 speed 2 direction 210 will hit the wall.  
scanf returned -1  
Returning 0  
Total runtime is 0.002854347 seconds
```

The input code should ask the output code to deal with the final return value from scanf. The output code won't do anything in graphics mode, but it will print the message when in text mode. Do **not** hard code a -1 in this print. There will be data files where scanf returns other values.

The main function will always print the value it intends to return to the operating system since this message will come after any graphics teardown and the program can freely print to the console.

Finally, as the very last thing before main returns, main will compute the runtime (real time, not simulated time), and print it. The main given to you handles all of this appropriately.

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.

Graphical Output

In graphics mode, it is possible that initialization could fail. If so, the program should terminate gracefully and return a value of 1 to the operating system.

Here are two screenshots of the program running the xA.mm input file with a scale factor of 1. The first screenshot shows repeated calls to `midi_smile` to place the white smiley face on the maze. The scoreboard line shows the numeric data passed in with the call; this smile has a score of zero and is at (9.01, 15.95) and going in direction 120 degrees. The white dots in the maze are drawn by the library to show the path of motion. The simulation clock shows 8.093 seconds elapsed simulation time. This value is driven by your code calling the `midi_time` function. In lab 2, all score values will be zero.

```

stdlinux.cse.ohio-state.edu - PuTTY
SIM: 00m 08.093s
REAL: 08.231233s
FPS: 31.457 fps
Screen: 24 L, 80 C
Field: 22 R, 43 C
DX= 0.500 DY= 1.000
Supports 8 colors
7 status lines
Lib Midi Version 1.1

:) 0 9.01, 15.95 120

```

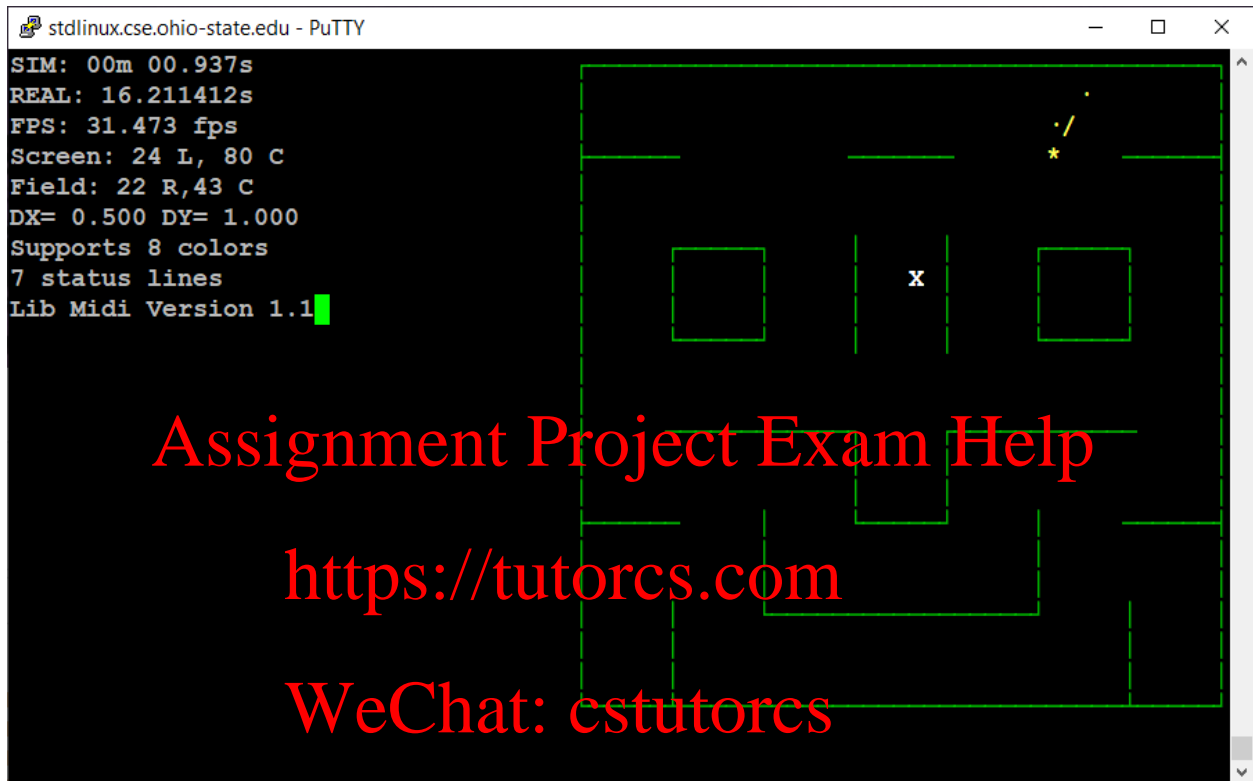
Assignment Project Exam Help
<https://tutorcs.com>
 WeChat: estutores

The second screenshot (shown below this text) shows a moving yellow projectile. This was done with calls to the `midi_projectile` function. The projectile is the `*` and the `/` roughly shows the direction it is traveling. There is also a trail of dots to help show the path. Projectiles do not generate a status line the way smile avatars do.

The white X is from the prior part of the data file and it shows where the white smile hit the wall. The X was placed via the `midi_x` function. When doing final output in graphics mode, the code will call that function if the object was a smile and it will not if the object was a projectile.

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

Your code should never output an object that conflicts with any of the walls. An error message will be printed if it attempts to do so.



Final output in graphics mode is different than text mode. If the object is a smile, place an X at the last valid place to the object. Then run a loop that does the clear, time, refresh, and wait using the normal `delta_T` time for the wait. That loop should run for 4 seconds. Don't change the simulated elapsed time in this loop, just output the simulated elapsed time as it was last update each time through the loop. This give the sim time for the trails to die out and to see how everything wound up.

Making Graphics Happen

Take a look at the Library document to see all of the functions available. You might put it up in a different browser window side by side with this one.

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

- At the beginning of the program, call `midi_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 `midi_clear`

- All the different parts of your code that need to draw things in this frame make various **midi_** calls as many times as they need to get everything on the screen. In lab3 there will be more than one thing onscreen at a time, so there will be multiple **midi_smile** and **midi_projectile** calls in each frame. The other call to make is to **midi_time**.
 - After all of the drawing calls have been made, **midi_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 **midi_initialize**, a call to **midi_teardown** is made to restore your screen.

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

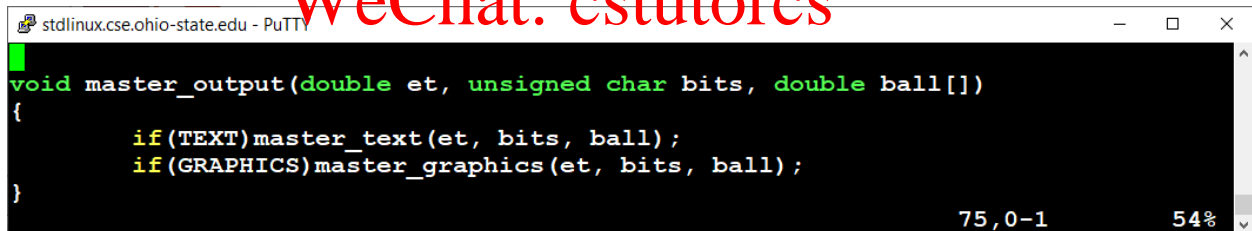
Only midi_initialize and midi_teardown may be called outside of the file output.c.

Framerate and DELTA_T

Our simulation runs at 32 frames per second, so we wait 1.0/32.0 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. The delta_T value must be passed to output functions that need it. It should be defined in the simulation code and passed as needed.

Protecting Graphics Functions

The code shown below cannot be used as it is for this lab2. Your master output will have a different set of parameters. Aside from the parameter lists, this is a *perfect* way to structure your main output function.



```

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 && midi_initialize() ) )
```

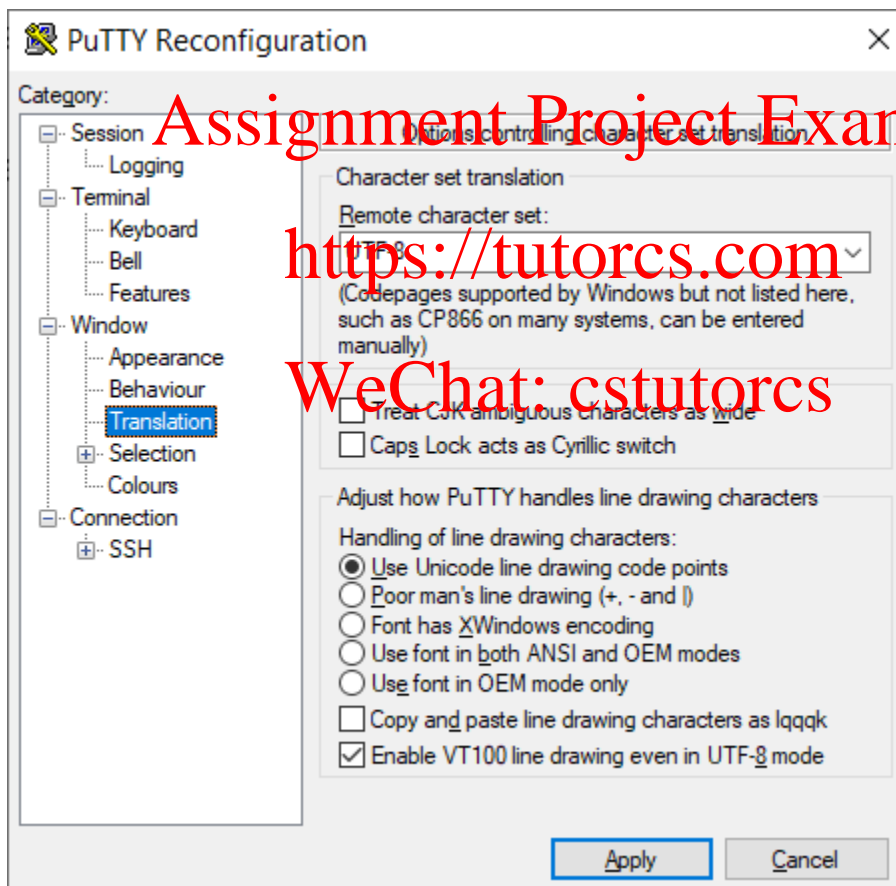
This uses short circuit evaluation along with TEXT and GRAPHICS and it give a perfect line to decide if the simulation initialized cleanly. When in text mode it always succeeds. Only in graphics mode does it make the library call and that call determines if it succeeded.

Never presume that TEXT and GRAPHICS are the Boolean negation of each other. If your 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*.

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 midi_ in the function name. None of your functions should start with “midi_” 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 check the box for VT100 line drawing so that you get the right characters. This is from the latest version of PuTTY, if yours doesn't have the checkbox, update your software. Other terminal emulators may not give you the pretty lines that PuTTY gives.



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. -lm -lmidi -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 libmidi library is in . (the current folder)

-lmidi
means link in the libmidi.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)

-lm
means link in the C math library (it will be found in the usual place that standard libraries are found)

Use this sample p1.c code

```
#include "mdid.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 midi_initialize and midi_teardown. **If you call midi_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 midi_initialize, your code really needs to call midi_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 && midi_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 midi_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:

```
midi_initialize() /* only do the rest of the code if that returned a true value! */
midi_refresh() /* do something visible */
getchar() /* make it wait so we get to see it */
midi_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.

```
midi_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 midi_refresh() call

```
midi_clear(); /* show the maze */
```

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 midi_clear and then draw all of our balls. We still need to draw a smile.

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

```
midi_smile(3, 6, 2.0, 20.0, 180.0);
midi_projectile(4, 2.0, 2.0, 270.0);
midi_X(7, 10.5, 16.5);
```

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.

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs