Maze

Summary of the MAZE project. The MAZE project is worth 5% of the course grade.

There is 1 submission:

- *Maze.ans* is worth 5% of the course grade. The grade is based on correct answers, documentation, and both executable instructions *written* and instructions *executed*. See the calendar for the due date.
- If you need help from the staff:
  - submit your assembler source code to the ASK4HELP locker
  - send a note to the staff stating you have submitted code to ASK4HELP
  - explain the problem to be addressed
  - Assure your code is fully documented with header blocks and line comments. The staff may request to see your design.

If working on a team … create a single maze.ans file
that is submitted by both members of the team to their respective submit lockers.

You are going to write a subroutine named *nextval* that will guide a mouse through a maze.

The maze is a 15 row by 30 column array (each element is a byte) that can be traversed using a right or left turning mouse. These are both discussed further on in this document. The rules associated with moving the mouse are:

- You may only move one position at a time.
- You may only move into a location in the maze if the value of that location is blank (20h).
- You may not move into a location that is blocked (contains any value other than 20h).
- You may only move left, right, up, down.
- You may not move diagonally.

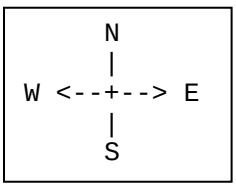
A position is given as ***y,x .*** The upper left square is 1,1 and the upper right square is 1,30 and the lower left square is 15,1 and the lower right square is 15,30.

The top and bottom borders of the maze (y=1, x=1...30 and y=15, x=1...30) will always be blocked.
The left and right borders of the maze (y=1...15, x=1 and y=1...15, x=30) will also always be blocked ***except*** that there will be one blank square for entry to the maze and one blank square for exit from the maze. Below is a sample maze.

1,1 1,30

X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
X															X														X
X															X														X
X		X	X	X											X								X	X	X				X
X				X											X						X				X				X
X				X							X				X			X	X	X	X				X				X
X				X	X	X	X			X	X	X						X							X				X
				X	X	X	X			X	X	X						X						X	X	X	X		
X				X							X							X	X	X	X				X				X
X				X											X						X				X				X
X				X											X										X				X
X		X	X	X											X								X	X	X				X
X															X														X
X															X														X
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

15,1 15,30



We will use map directions to indicate the direction of travel for the mouse.
The mouse will enter the maze on the west side (x=1).
In most cases the initial direction will be traveling east.
Its goal is to reach the empty location on the east side of the maze (x=30).

Specifications For The Nextval Subroutine

- Name your source file *nextval.asm* and label the entry point of your subroutine *nextval:*

- When your subroutine is called, *you make 1 move in the maze (East, South, West, North) and then return*.

The input parameters passed to nextval are 4 pointers that point to the data which is in the calling program's Data Segment.

- **si** contains the address of the current value of the x position. X is an unsigned byte in the range 1 to 30.
- **di** contains the address of the current value of the y position. Y is an unsigned byte in the range 1 to 15
- **bx** contains the address of the current value of the direction of travel of the mouse.
  The direction is an unsigned byte with these values: E=1 S=2 W=3 N=4
- **bp** contains the address of the maze (15 rows and 30 columns). Each element is a byte. The structure of the maze is a string of 450 bytes representing 15 rows each with 30 elements. An empty location has the value 20h. A blocked location contains any value other than 20h.

The output from nextval is the move of the mouse *one square* (East, South, West, North).

- Set the x and y values in the caller's data segment to the next location to which you want to move the mouse.
- Set the direction in the caller's data segment to the new direction of travel.
- All modified registers must be saved and restored to their original value.

Programming notes.

- No error checking is needed. You may assume that the maze and the values of x and y and direction are valid.
- There will always be at least one valid path through the maze.
- The outer perimeter of the maze (the borders) will always be blocked except for the entrance and exit.
- The driver will detect when your mouse has traversed the maze. You do not have to worry about that.
- The x and y and direction you select must be compatible. For example, let's assume that you are at location y=6, x=20 and the direction is east. You decide to move north to y=5, x=20. You will change y from 6 to 5. You must also change the direction from a 1 to a 4 indicating the mouse changed its direction of movement from east to north.
- Your mouse is not allowed to stay in a square. It must always move to a new square.
- Once the mouse leaves the starting square, it may not return to that starting square.
- Do not do any file I/O from your subroutine. Do not read data. Do not write messages.
- You may not modify the maze in any way.
- Your mouse must successfully traverse all legal mazes. This is defined as: linking your nextval.obj subroutine with our mazedrvr.obj; using our testmaze.bat procedure to perform the test; receiving the "Congratulations! Your mouse has traversed the maze." message.
- Design information provided is meant to only show functionality. Assembler efficiency is the responsibility of the coder.

The notes below are intended for the mischievous and devious among you.

- Do not keep any history information between the driver's calls to nextval.
  Every time nextval is called it must calculate its move based upon the new parameters passed to it.
- Code to the nextval specification as given on this page.
- Do not code anything specific to the driver.
  - Most of the time when you are first called, the mouse will be on the west side of the maze and the direction will be east. However, that is not part of the specification and may not be true. The driver may select an arbitrary position and request your next move.
  - Don't reverse engineer the driver code and find the calling program's data areas and remember them or even worse hard code them into your program.

Information About The Maze Data

The calling program defines all the variables. You access all data using the pointers passed to nextval and indirect addressing.

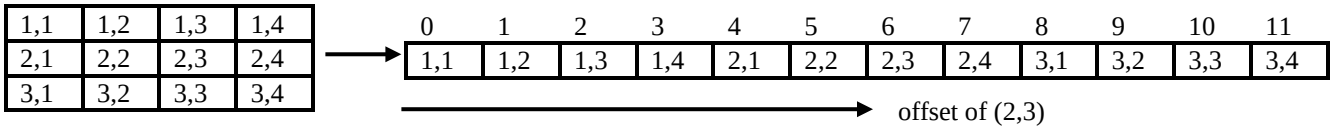
- The variable ***maze*** is defined as 15*30 bytes. This is a string of 450 bytes representing 15 rows each with 30 elements.

The maze *appears* as a two dimensional array. However, real memory is just a one-dimensional linear string of bytes.
So when you have an array location such as maze(y,x) you need to be able to locate its real location in linear memory.
The way this is done is to calculate the offset of the location into the maze. This assumes the first element is 1,1.

offset = (y-1) * number_of_columns + (x-1)

For our maze, with 30 columns, that offset is calculated as: offset = (y-1) * 30 + (x-1)

This example shows how data looks for a 3 row 4 column array.



To find a specific cell, you need to calculate the offset to that cell.
This is the formula: offset = (y-1) * 4 + (x-1)
For example for cell (2,3) the calculation is (2-1)*4 + (3-1) = 4 + 2 = 6

- The variables ***x*** and ***y*** and ***direction*** are unsigned bytes.

How To Select The Next Location

Both left and right turning mice are guaranteed to traverse a valid maze

A left turning mouse works as follows.
- At every location in the maze you try to turn left.
- If you cannot turn left then you try to go forward.
- If you cannot go forward then you try to turn right.
- If you cannot turn right then you go backwards.

In other words, the priority of movement is:
left, forward, right, backwards.

A right turning mouse works as follows:
- At every location in the maze you try to turn right.
- If you cannot turn right then you try to go forward.
- If you cannot go forward then you try to turn left.
- If you cannot turn left then you go backwards.

In other words, the priority of movement is:
right, forward, left, backwards.

Important Programming Notes.

This assignment focuses in indirect addressing.
The inputs to nextval are pointer registers.
They do not contain data but instead they point to the data.
In some cases you will need to use *data size override* and *segment override*.
For example:
- To access the current y value you would code : mov al,[di] ; load al with the y value of 1 to 15
- To set the direction of travel to East you would code: mov byte ptr [bx], 1 ; set the direction to East
- Remember that the bp register in brackets defaults to the stack segment. To access the maze, using the bp register, you will need to use a *data segment override*.

Files You Need

Retrieve the testing and grading files. All files are packed together in one self-extracting file named *unpack.exe*.
Go to the class homepage. Click on *Lockers With Program Grading System Files.* Click on the *MAZE* assignment.
Download the *unpack.exe* file and save it on your hard drive in the \P23X\MAZE directory.
Start DOSBox and execute unpack.exe to create the testing and grading files.

One of the files retrieved will be the model for your subroutines.
For MASM it is named *nextval.m* Rename *nextval.m* to be *nextval.asm* and then add your code to that file.
All source code must be in the single file named *nextval.asm*, otherwise we cannot count instructions written.

One of the files retrieved will be the main driver program for your subroutine. It is named *mazedrvr.obj*
Assemble and link your subroutine with the driver program *mazedrvr.obj* using these commands.

MASM
`ml /c /Zi /Fl nextval.asm`
`link /CO mazedrvr.obj nextval.obj`

Testing

The maze driver program will perform these functions.

- Read an input file that describes the maze, build and display the maze, display the mouse.
- Call your subroutine to request the next position to which the mouse should move.
- Move the mouse based on the values calculated by your subroutine.
- Check for successful completion of traversing the maze.

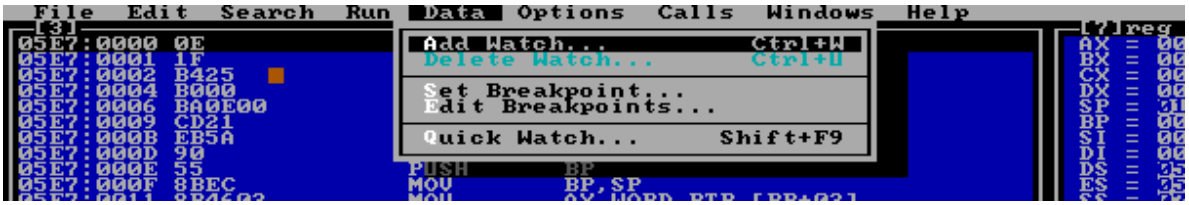
We provide test mazes that are named *maze.01* through *maze.06*.

Run the program by typing: ***testmaze maze.01***

We also provide a set of randomly created mazes. To run against these type: ***testmaze r***

You may need to use the debugger to isolate the cause of any defect uncovered by your tests.
After the debugger loads, you can set breakpoints at the entry of your nextval subroutine.

- To run under CodeView under DOSBox
  type: *cvset* ... Important - This copies the CodeView configuration file
  type: *cv mazedrvr < maze.01*
  When CV loads.
  - On the top row select "Data" then select "Set Breakpoint"



  - Under "Location:[...]" type *nextval* so it looks like Location:[nextval]"
  - Click ok
  - Pres F5 to go ... and CV stops at the entry to your nextval routine.
    This allows you skip over all the driver code and get to your code.
  - In the Memory Window you will see data at xxxx:yyyy Make sure that xxxx matches the value in the DS register. You should be able to just put the cursor on the xxxx field and set it to match the DS register.
  - Use F8 to step through your code.

Grading

Your program must successfully traverse all legal mazes.

If we test your submitted program against any legal maze and it fails to successfully traverse that maze, then your grade will be rescinded and you will need to fix your program and resubmit it. The date you submit the fixed program will then be the date used determine any early bonus or late penalty.

After you verify that it successfully traverses the sample mazes, run the grading program by typing:

gradmaze

Once nextval is working correctly, you may make additional grading runs to improve efficiency. If you execute the command *testmaze mark* all subsequent grading runs will be marked as only being done to improve efficiency.

The final grade will be based on:

- 50 points for getting the correct answers to the grading program test cases.
- 15 points for the number of executable instructions *written* to *correctly* complete this assignment.
- 15 points for the number of instructions *executed* in the grading run that had the *correct* answers.
- 20 points for documentation of a program that functions *correctly*.

Efficiency and documentation are only a concern after the code works totally correctly.
Therefore, efficiency and documentation grading will only occur after your code passes all the functional tests.

Submit Your Assignment

Electronically submit the file ***maze.ans*** created by the grading system.

It is your responsibility to assure:

- It contains two concatenated files. First the *results* file and then your source file *nextval.asm*.
- It contains the line: ++ Grade ++ *xxx* = Total grade generated by the Grading System.

Incorrect electronic submissions will result in your program not being graded and considered late.

This is a version of nextval in pseudocode that you may use to get started.
It is functionally correct but not optimized for efficiency.

```
// The term *direction is read as ... the value pointed to by the direction pointer

if (*direction == 1) goto testn;     // if direction is east  try to move north first
if (*direction == 2) goto teste;     // if direction is south try to move east  first
if (*direction == 3) goto tests;     // if direction is west  try to move south first
if (*direction == 4) goto testw;     // if direction is north try to move west  first


// Try the desired direction first. If it does not work move to the next direction
// The mouse must be able to move in one direction if the maze is legal

testn:                               // test the north square being empty
offset = (*y-1)*30 + (*x-1);         // calculate offset to current square
if (maze[offset-30] == ' ')          // if north square is empty
   {                                 // turn north by taking these actions
   *y = *y-1;                        // - decrement y
   *x = *x;                          // - leave x alone
   *direction = 4;                   // - set direction of travel to north
   return;                           // return
   }

// If we cannot move north, fall through and try east

teste:                               // test the east  square being empty
offset = (*y-1)*30 + (*x-1);         // calculate offset to current square
if (maze[offset+1] == ' ')           // if east  square is empty
   {                                 // turn east  by taking these actions
   *y = *y  ;                        // - leave y alone
   *x = *x+1;                        // - increment x
   *direction = 1;                   // - set direction of travel to east
   return;                           // return
   }

// If we cannot move east, fall through and try south

tests:                               // test the south square being empty
offset = (*y-1)*30 + (*x-1);         // calculate offset to current square
if (maze[offset+30] == ' ')          // if south square is empty
   {                                 // turn south by taking these actions
   *y = *y+1  ;                      // - increment y
   *x = *x;                          // - leave x alone
   *direction = 2;                   // - set direction of travel to south
   return;                           // return
   }

// If we cannot move south, fall through and try west

testw:                               // test the west  square being empty
offset = (*y-1)*30 + (*x-1);         // calculate offset to current square
if (maze[offset-1] == ' ')           // if west  square is empty
   {                                 // turn west  by taking these actions
   *y = *y  ;                        // - leave y alone
   *x = *x-1;                        // - decrement x
   *direction = 3;                   // - set direction of travel to west
   return;                           // return
   }

// If we cannot move west, fall through and try north

goto testn;
}
```