Maze029

An 8080 Assembly Language

Maze Game from a Rat's Eye View

By

Lloyd Johnson

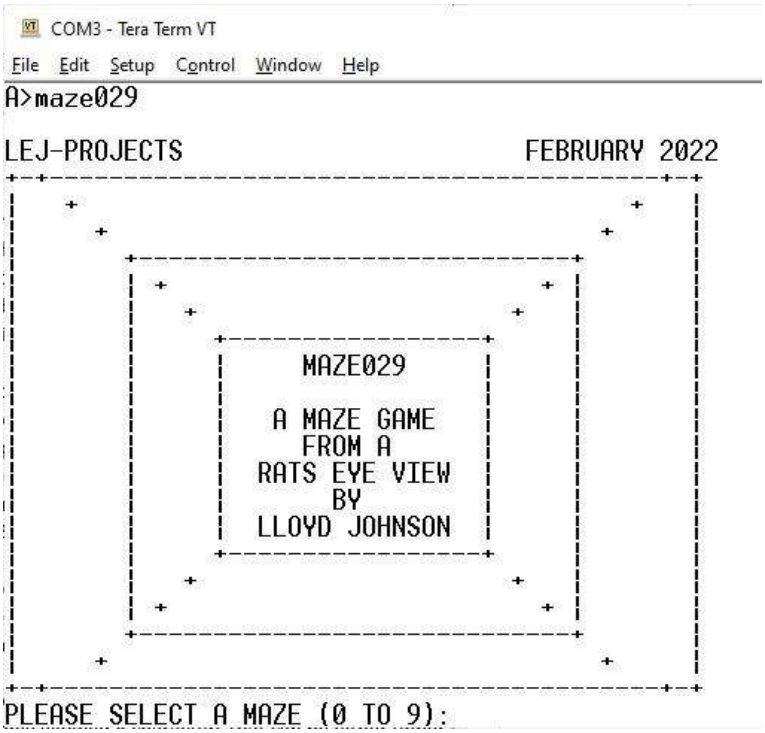
Introduction

I've always been disappointed that by the time I could afford a personal computer, they no longer had the front panel lights and switches. Early in my career I had worked with computers that had front panel lights and switches. I never used the lights and switches much but thought they would be fun to use for debugging programs. I retired from the Boeing company in August 2020 and became interested in Arduinos. I was thinking of a project that would emulate an old computer with front panel lights and switches. I didn't have to look far to see that it was already done and done well. I'm talking about the Arduino emulation of the Altair 8800. I purchased one from www.adwaterandstir.com and another smaller one from eBay from a user named dmanpdi.

I dabbled in assembly language for various processors over the years, but I have never attempted to write any large programs. I decided to learn 8080 assembly language and CP/M and write a maze game for my Altair 8800 emulation. In this document, I will tell you about the game, how to load it, how to play it and how to modify it. This document and the associated files for this game have been uploaded to www.github.com/LEJ-Projects/Maze029. This includes the source code, an Intel hex file and an executable (.COM) of this game. There are also a few tools I developed for creating the mazes, converting .hex files to a .com file, dumping the .com file, and converting the dump output of a .com file back to a .hex file. You do not need these tools to load or run the game, but they might come in handy if you want to update the mazes or make other modifications. I will describe them in more detail later in this document.

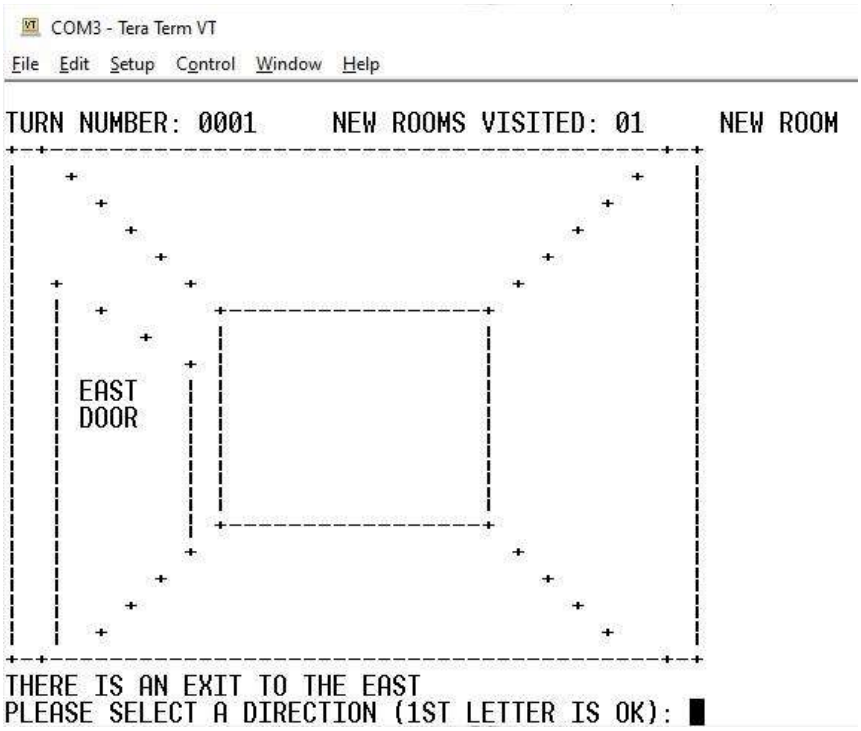
Game Description

Maze029 is a maze traversal game. Although the mazes are only 10x10, they are difficult to solve since you are only given a rat's eye view of the current room you are in. This game was written in 8080 assembly language on an Altair 8800 Arduino emulation. It should run on any CP/M system. The game is launched by typing maze029 at the CP/M prompt. This will result in the following splash page:



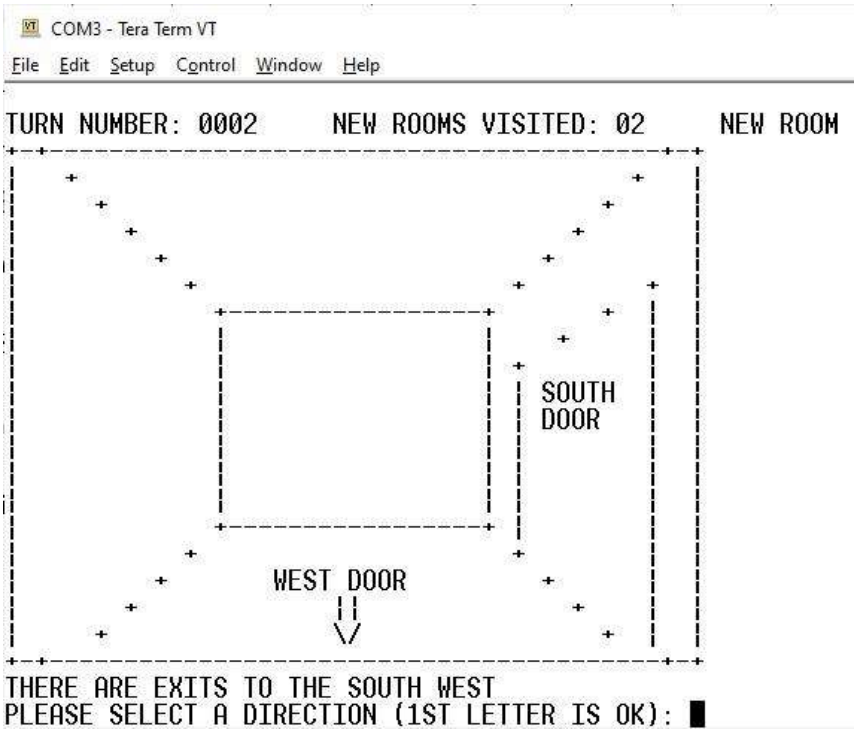
The program contains ten mazes that are numbered 0 to 9. The name, maze029 should be pronounced as "maze zero-two-nine" because there are mazes numbered 0 to 9. All mazes consist of 100 rooms and are arranged as 10 x 10. The mazes were generated by mazegen.exe and copied into the assembly language source code for maze029. Mazegen.exe is a cpp program I wrote to randomly generate 10x10 mazes. A detailed description of Mazegen and its algorithm will be described later in this document. Mazegen was designed such that the maze entrance is somewhere on the northern edge of the maze and the exit is somewhere on the southern edge.

When a maze is selected, the initial room is displayed from a rat's point of view – hence the description, "Rat's Eye View". All you will see is the available exits for the current room. In the following screen capture, there is a door on the left labeled "EAST DOOR". Other than that door, there are no other visible exits. The side of a room (left, right, straight ahead or straight back) where a door is identified depends on the direction you are facing. Initially the player is facing south so the door on the left goes east.



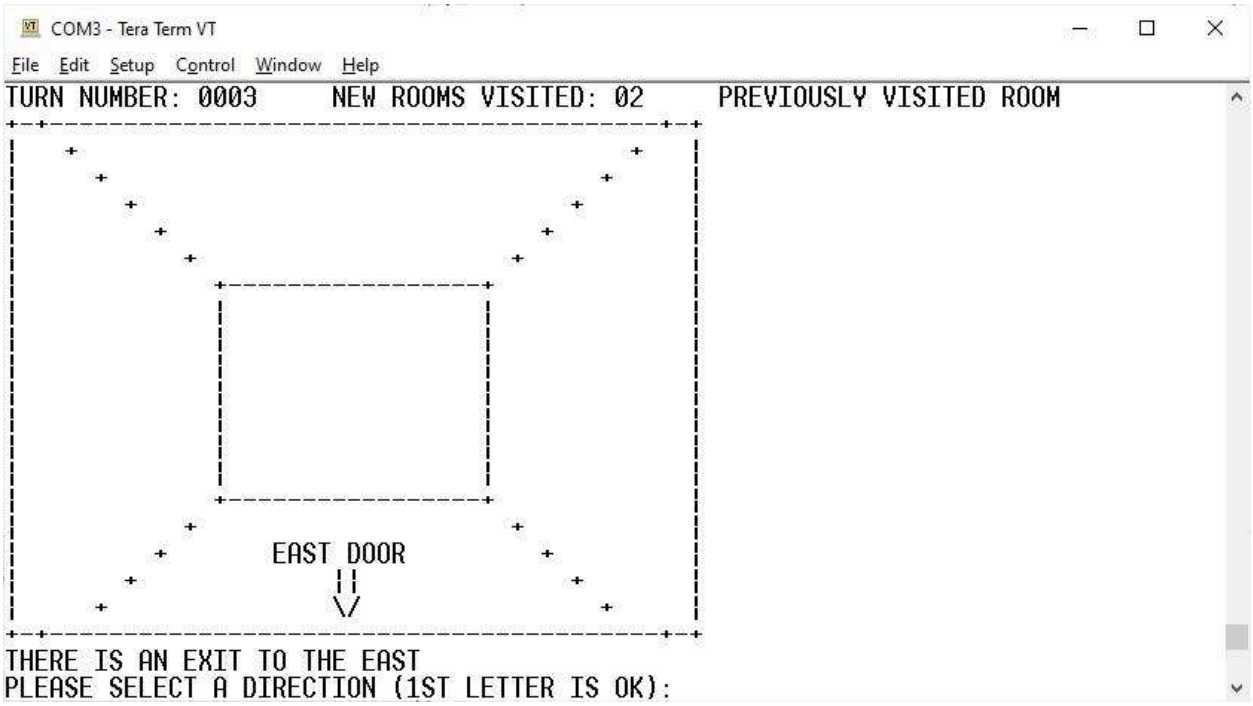
The starting room is on the northern edge of the maze. This is the only time the player will be facing south when he is on the northern edge of the maze. This is because the direction the player faces will always be the direction last chosen on the previous turn. Since this is the first turn and a previous direction has not yet been chosen, the initial orientation defaults to facing south.

The next display shows what the player sees when the east door is chosen by entering the letter, E. (In this case, there was no other choice to make.)



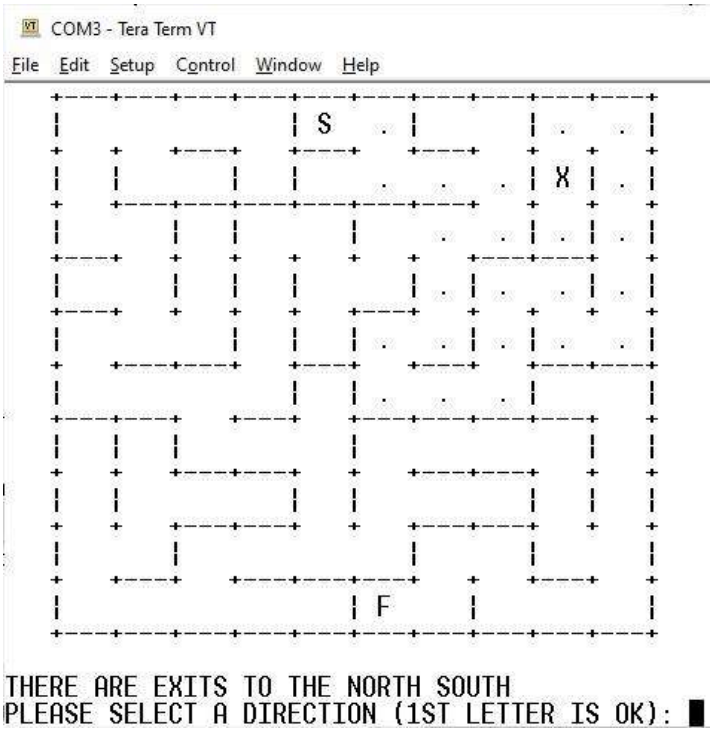
In the above display, there is now a south door on the right and a west door leading back. Since east was the last direction chosen, the player is now facing east. Except for the first turn, there will always be a door leading back.

In the top line of this display, the turn number is displayed along with the number of new rooms that were visited. On the far right of the top line, it identifies this as a new room. In the previous display (turn 1), the room was also identified as new. In this example, if the letter W is entered, the player is returned to the room where he was on the previous turn. In this case it would be the initial room when the game was started. This is displayed as follows:



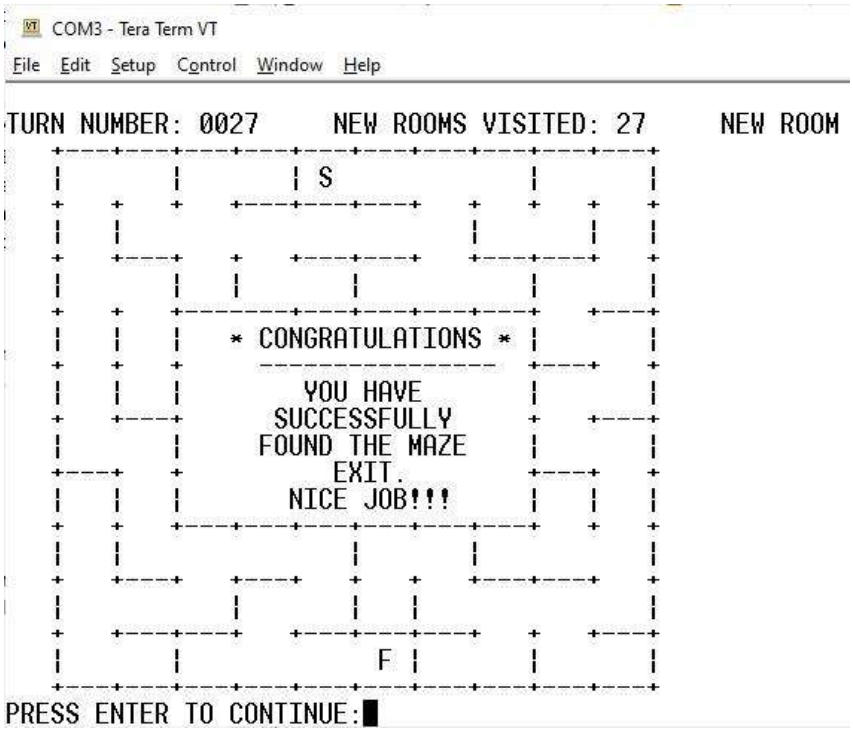
Now the top line identifies this room as a previously visited room. The only time a previously visited room can be entered from a new room is if the direction chosen was opposite of the previously chosen direction. In other words, the only way to go to a previously visited room from a new room is to go back. In turn 2, the room was a new room and had a west door leading back. By selecting this direction, a previously visited room was entered from a new room. Any other door chosen from a new room will always lead to another new room. This is so the maze will not double back on itself. The room displayed above in turn 3 is the same room displayed in turn 1. It does not look the same. In turn 1, the east door was on the left and in turn 3, the east door is pointing back. Again, it is all a matter of orientation The fact that the same room can look different depending on the direction used to enter the room makes solving the maze with just a rat's eye view very challenging. Perhaps, it is too challenging.

After wandering around the maze and getting hopelessly lost, there is always a temptation to give up and walk away. So that will not happen, I developed a cheat feature. The cheat feature consists of entering the letter, P instead of a direction. The following is an example of where this cheat feature was used while traversing a different maze. As noted in the display to the left, there is now an overview of the maze (bird's eye view) with the letter S indicating the start, the letter, F indicating the finish, and the letter X indicating the present position. The display also indicates the rooms that have been previously visited with a period (as opposed to a space). The period indicates the bread crumb trail that was left behind as the maze is traversed. The bread crumb information was also available in the other display since the program tells you in the upper corner of the previous display if it is a new room or previously visited room. Continual use of the cheat feature makes solving the maze trivial and should only be used in situations where you are hopelessly lost.



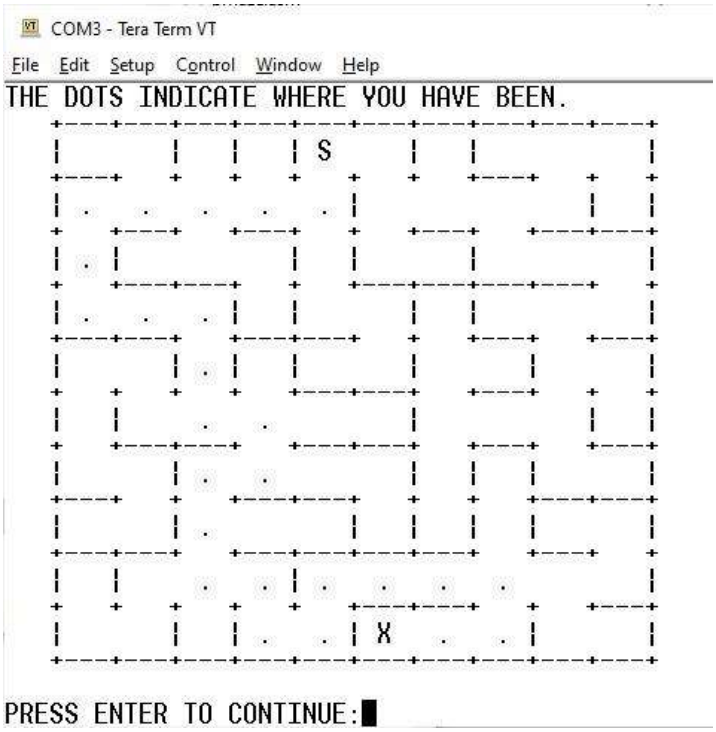
Incidentally, the best way to solve the maze is to always look for new rooms to enter. Eventually, one of them will be the exit.

When the maze exit is located (presumably without too much cheating), the following congratulations message is displayed:



You might notice that the turn number is identical to new rooms visited. That is because I knew this maze and went directly to the exit. Once this message is displayed, the turn number and new room visited count should be noted before pressing enter. Once enter is pressed, this page scrolls away and the information is lost forever. (Unless, of course, you are using a terminal program that allows you to scroll back in the buffer.)

When enter is pressed, the bird's eye view display is presented. This is the same display you can get during the game if you use the cheating feature and enter the letter, P instead of a direction. This is a record of where all you went when exploring the maze.



That is the game. The other sections in this document will describe loading the game to a CP/M environment, the maze generation program and some of the other tools that were developed for generating the program, MAZE029.ASM.

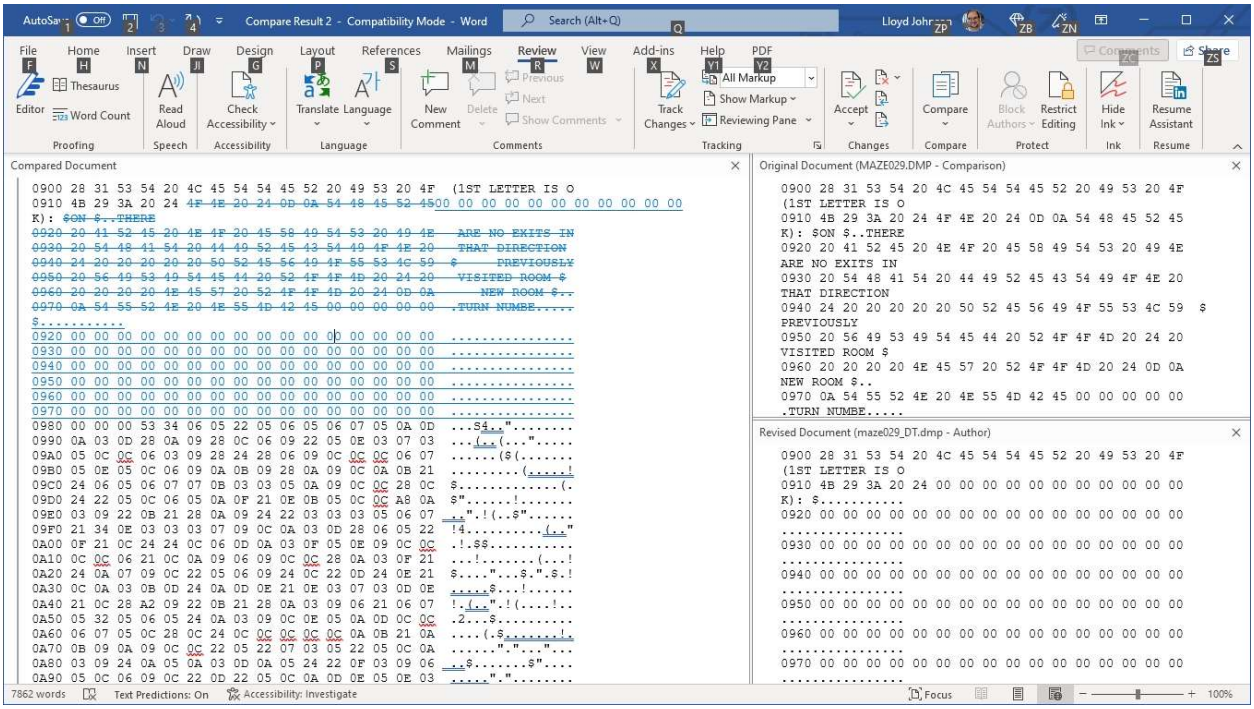
Load Discussion

Load instructions are simple if all I had to say is put the files on your CP/M machine and type MAZE029 to launch the game. However, I have not yet come up with a method of directly transferring binary files to my CP/M machine and so I believe this warrants more discussion.

The source file, MAZE029.ASM, the hex file, MAZE029.HEX and the executable file maze029.com are available on the GitHub page. From any one of these, you should be able to load the game to your C/PM machine and execute it.

The simplest of these would be to transfer maze029.com file to your machine using a binary file transfer protocol to do the transfer. I have not done this, but it is something I might explore later. I was hoping to find a PC based utility that would allow me to insert or remove files from the file on the SD card that represents the Altair's disk drive. My search for such a tool has not been successful and I suspect that developing one is beyond my skills. The file, maze029.com that is provided in the GitHub repository was created from MAZE029.HEX using the program, hex2com.exe that I developed. The source and executable for hex2com (hex to com) is included in the GitHub repository. For this section, I'm using a convention of upper-case names for files that originated from my CP/M machine and lower-case names for files that originated from my Windows desktop machine. The file, maze029.com is lower case to identify that it was not created on my CP/M machine.

I verified the file, maze029.com provided in the GitHub repository by dumping its contents and comparing it to the output from the CP/M DUMP command performed on MAZE029.COM. The files were identical except for the areas of memory corresponding to the assembly language define storage pseudo-op (DS). The data in that uninitialized area of memory contained garbage in the .COM file produced on the CP/M machine and contained zeroed out data in the .com file produced by the hex2com.exe program. A portion of the dump file comparison is illustrated in the following screen capture:



The program, fil_dmp.exe was a program I developed many years ago as a throw away program. I retrieved it and made some minor modifications such that it produces the same output as the CP/M DUMP command. The source and executable for fil_dmp is included in the repository.

So, if you have a reliable method of transferring a .com file to your CP/M machine, the file, maze029.com should work just fine for you. As another test, I used another program I developed called dmp2hex.exe (dump to hex) to convert the dump output of maze029.com to a hex file, maze029.hex. I did not include this version of maze029.hex in the repository. I transferred the maze029.hex created on my Windows desktop machine to my CP/M machine, executed the CP/M LOAD command on it and I had an executable version of MAZE029.COM that ran just fine. I consider this further verification of the maze029.com file in the GitHub repository.

What is probably the best method of getting maze029.com to your CP/M machine is to transfer the file, MAZE029.HEX by copying and pasting it to the terminal program you use to communicate with your CP/M machine. An editor should first be opened on the CP/M machine to receive the file. I use the CP/M program, Word Master for creating the file. When the file is

open and ready to receive data, the contents of MAZE029.HEX on my desktop machine is copied and pasted to the terminal program communicating with my CP/M machine. The file is saved on the CP/M machine, as MAZE029.HEX. It can be converted to an executable .COM file with the following command:

A> LOAD MAZE029

A reason to like this method is because each line of the .HEX file has a checksum and any data corruption of MAZE029.HEX that occurred during the transfer would be flagged by the LOAD program. If there are no errors, you should have an executable file that will launch the game when you enter the command:

A>MAZE029

And of course, you can transfer the assembly language source program and reassemble it. If this is done, the only file needed to be transferred is MAZE029.ASM. The current version of MAZE029.ASM has over 1400 lines and requires approximately 60k of disk space. If you have good method of transferring the file, MAZE029.ASM to your CP/M machine, I'd recommend you use it. I have not been successful transferring the entire file at one time. I instead divided the file up into 4 parts. The parts are identified as m029sp1.asm, m029sp2.asm, m029sp3.asm, and m029sp4.asm. The files are combined on my CP/M machine and saved as MAZE029.ASM using the following PIP command:

A>PIP MAZE029.ASM= m029sp1.asm, m029sp2.asm, m029sp3.asm, m029sp4.asm

The only reason you would want to transfer the source and assemble it instead of just transferring the hex file is if you plan to modify the source, (perhaps to replace some of the mazes or maybe fix a bug). In general, it is easier to do editing on smaller files and combine them into one file then to edit the main source file due to its size. Once the source code MAZE029.ASM is on your CP/M machine and modified to your satisfaction, assemble it with the following command:

A>ASM MAZE029

This will produce the files MAZE029.HEX and MAZE029.PRN. MAZE029.HEX can be converted to an executable .COM file with the following command:

A> LOAD MAZE029

Yu should now have an executable file that will launch the game when you enter the command:

A>MAZE029

This concludes my discussion on loading the file. Hopefully this insight will provide a method that work for you and you are able to successfully load and run the game.

Maze Algorithm Discussion

Initially, I went to the internet and found an on-line maze generator that would make a maze to any dimension I wanted. I selected 10x10. I printed out the maze and took a straight edge and drew horizontal and vertical lines such that there was a square around the rooms. I then assigned a number to each of the rooms based on the exits to the room. Each direction was given a value as follows:

Direction	Value
North	8
South	4
East	2
West	1

The value for the room is equal to sum of the door values. Computer people will quickly see that the lower nibble has a bit assigned for each direction. I found that doing this manually for 100 rooms was time consuming and error prone. After I completed it, I setup ten DB (define byte) statements with ten bytes per statement to represent the maze as follows:

```
DB   06H,07H,03H,21H,06H,31H,06H,05H,06H,05H
DB   0CH,28H,06H,05H,0CH,06H,0DH,0CH,28H,0CH
DB   0CH,06H,09H,0CH,0AH,09H,28H,0CH,06H,09H
DB   0AH,09H,06H,09H,06H,03H,05H,0CH,0EH,05H
DB   06H,03H,09H,06H,0DH,06H,09H,0CH,28H,0CH
DB   0AH,05H,06H,09H,28H,0AH,03H,0DH,06H,0DH
DB   06H,09H,0CH,24H,06H,03H,05H,0CH,0CH,28H
DB   0AH,03H,09H,0AH,09H,06H,0FH,09H,0AH,05H
DB   22H,07H,03H,07H,03H,09H,0AH,03H,05H,0CH
DB   0A2H,0BH,21H,0AH,03H,03H,03H,21H,0AH,09H
```

I wrote a short program to validate the maze by verifying that for each door leading from the room that there is a door in the room that the door leads to going back. For example, the highlighted room above has a value of 06H which means there is a door leading south and a door leading east. The room to the south has a value of 0CH which means it has a door leading north. The room to the east has a value of 07H which includes a door leading west.

I also used the bits in the upper nibble to indicate a previously visited room, the starting room, the finish room and if the room has only one exit. Knowing if there is only one exit determines if the message reads: “THERE IS AN EXIT TO THE” or “THERE ARE EXITS TO THE”. It saved a little bit of code having this information defined with the maze.

The bits for each of these bytes are defined as follows:

Description	Upper Nibble				Lower Nibble			
Bit Position	7	6	5	4	3	2	1	0
Definition if set	Maze Exit	Previously Visited Room	Room Only	Maze Entrance	North Door	South Door	East Door	West Door

			Has 1 Exit					

After researching some maze algorithms on-line, I decided to develop my own algorithm. I developed my own because I had trouble understanding the existing algorithms. I modified my program for validating the maze to now generate the maze. I left the validation code in place since it was useful for debug. The name of this program is mazegen. I included both the source and executable of this program.

My algorithm for generating a maze consists of starting with a maze of 100 room where each room has no doors. I randomly pick an entrance room and an exit room such that the entrance is somewhere on the northern side of the maze the exit is on the southern side.

There are two phases to the maze generation. The first phase is to create a path from the entrance to the exit. The second phase is to take any rooms that do not have doors and connect them to the main path. During phase 1, the upper nibble of the byte remains clear and the lower nibble contains door information. During phase 2, the upper nibble is used temporarily to store door information for the newly created path. Once the path is connected to the main path, data in the upper nibble of each room parameter is transferred to the lower nibble.

Phase 1 starts with the current room which initially is the maze entrance that was previously determined. During phase 1, a door will be added to the current room. The door will be added to a wall chosen randomly from the walls of the room that are available to receive doors. A wall can receive a door if there already isn't a door on that wall and if there is a room on the other side of the wall and the room on the other side of the wall does not have any doors on any of its walls.

Once a wall has been selected for a door, the current room is updated to indicate a door on that wall exists. The room on the other side of that door will become the new current room and the new room will be updated to add a door to connect to the previous room. The new room becomes the current room and the new current room is tested to see if it is the maze exit. If this is the maze exit, then phase 1 is complete and maze generation continues with phase 2.

If the current room is not the maze exit, then another door will be added to one of the available walls in the current room, the room the door leads to becomes the current room and the process is repeated until the maze exit is found.

In the event, that the new current room does not have any walls that are available to receive a door, then a new current room is selected. This is done by preparing a list of available room and selecting one of them to be the new current room. Only rooms that already have doors in them are available to be selected. Once a new current room is selected, a new door will be added to one of its available walls and again the room that the added door leads to will be the new current room. The new current room which will be tested for the maze exit and the same actions will be taken as the previous maze exit test repeating as necessary until the maze exit is found.

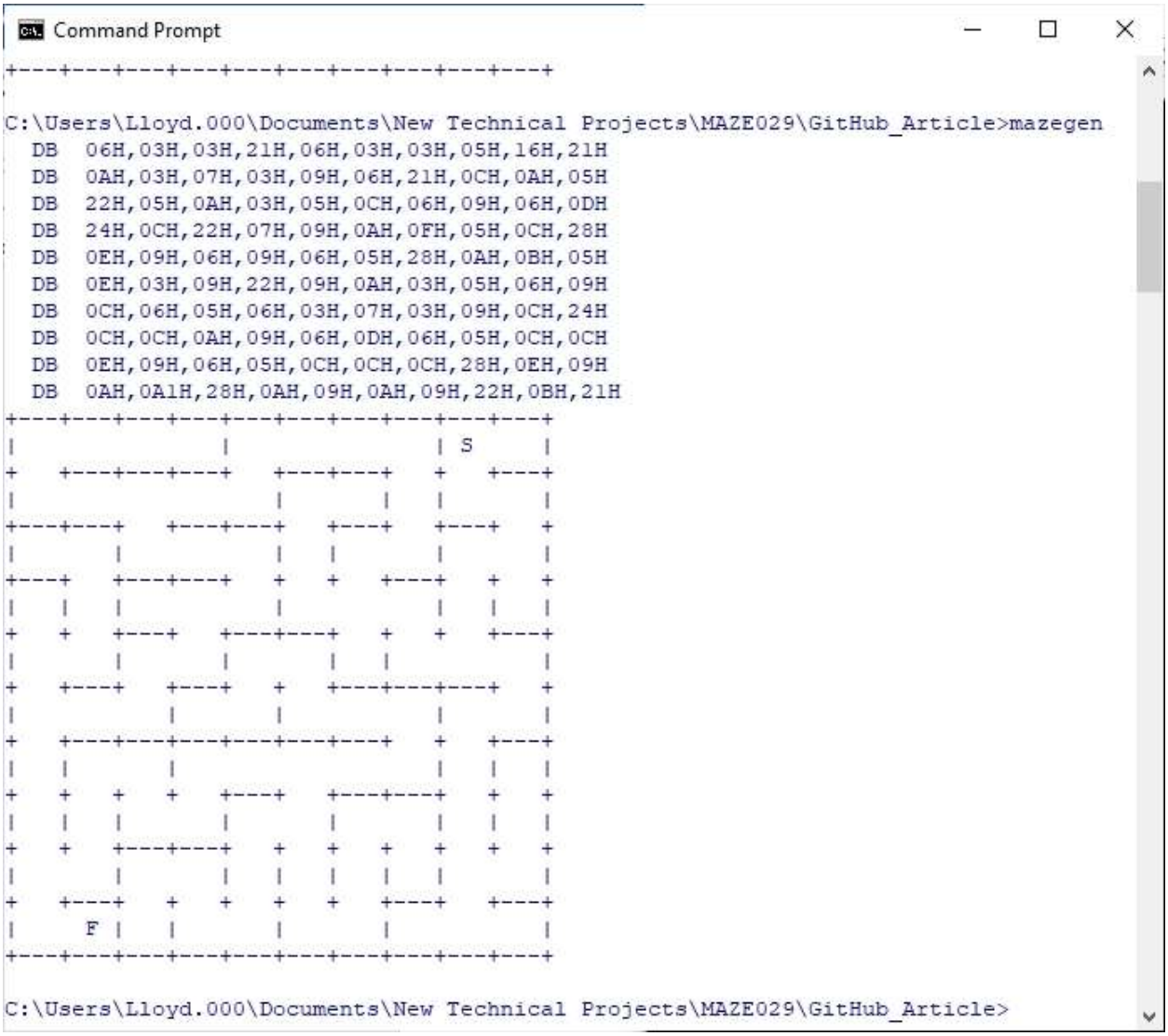
Phase 2 consists of creating a list of the rooms with no doors and selecting one of these as the current room. Key elements of phase 2 are as follows:

- Randomly pick a room with no exits.
- Randomly add an exit to the room such that it does not go outside the maze or connect to its own path.
- Make the room on other side of the added door the new current room.
- Keep adding rooms to the new path (using the upper nibble of the room byte) until a door is added that connects to the main path. During this phase any wall that leads to the main exit is not available to have an added door.
- If there are no available walls to add a door, randomly pick a room on the new path as the new current room and try adding a door to an available wall for that room.
- If a connection is made to the main path, the new path is added to the main path by transferring the data from the upper nibble to the lower nibble. The upper nibble will be cleared to prepare it for use last.
- Prepare a new list of rooms with no doors. If any rooms with no doors still exist, repeat the process by randomly picking one.
- If there are no rooms left that do not have any doors, then phase 2 is complete.

Once Phase 2 is complete, some cleanup work is performed. This cleanup consists of updating the appropriate bit in the upper nibble of the starting room to identify it as the starting room. I also left the room validation code active to verify each room is consistent with its neighbors and that no doors leave the maze. I also make sure there are no rooms that have no doors. (That would happen only if there was a coding error in Phase 2.) Also, during the validation process, any rooms that have only one exit are identified by updating the appropriate bit in the upper nibble.

Finally, the maze is printed. It is printed first as a set of 8080 assembly language DB (define byte) statements. It is next printed as a bird's eye view.

The program Mazegen is executed from a command prompt as follows:



The DB statements can be copied and pasted into the MAZE029.ASM replacing one of the existing mazes in the data area of the program. If the edit is performed carefully, there should be no problem assembling and loading the modified program.

Useful Tools

I have included the source and executable for three tools that were useful in developing this program. They are:

1. File Dump (fil_dmp)
2. Hex to Com (hex2com)
3. Dump to Hex (dmp2hex)

1. File Dump (fil_dmp)

The file dump program (fil_dmp) was something I wrote back in 1997 shortly after I started working on a NASA contract for Boeing. The program was meant to be a quick piece of throwaway code that would allow me to peek inside data files (LIF files) that would eventually be uploaded to the ISS (International Space Station) US computers (which were predominantly 386 computers). I have used fil_dmp for various things over the years so it never really was thrown away. I made some minor changes to it such that the output is identical to the output of the CP/M DUMP command. As discussed earlier in the document, this enabled me to compare the contents of a .COM file on my CP/M machine with a .com on my Windows desktop machine without transferring either of the binary files.

The fil_dmp program may also be a convenient means of moving .COM files to your CP/M machine if you do not have a means of doing binary file transfers. The output from the fil_dmp program (with the .com file as the input) can be used as the input to the dmp2hex program that is described later in this section. The output of the dmp2hex program is a hex file that can be transferred to the CP/M machine and converted to a .COM file using the CP/M LOAD command.

2. Hex to Com (hex2com)

The hex2com program will convert a file that is in Intel hex format into a .com program. I expected to find such a program on the internet and I did. Unfortunately, there were problems with each one I downloaded. Rather than debug and make someone else's program work, I decided to write my own. The format of the Intel hex file is simple and is described adequately in numerous places on the internet. I discovered the hard way that you must pay attention to the address supplied in the hex file. The hex files produced by the CP/M ASM command will leave gaps in the addresses when the define storage assembly language pseudo-op is encountered. The hex2com program will fill these gaps with zero data. I was initially caught off guard when I was comparing the maze029.com file I generated on a Windows desktop machine to the .COM produced by the CP/M LOAD command. I did the comparison by comparing dumps of each file. The data in the uninitialized memory on the CP/M .COM file resembled data that I had in the data section of the MAZE029 program. As it turns out, this data was just garbage. I mistook it for real data and was totally baffled how it came to be in the .COM file when it was missing from the .HEX file. Once I looked up the address in the MAZE029.PRN file, I saw that it was a define storage statement and data was garbage left in uninitialized memory.

The discovery of random garbage data in the uninitialized data brings out an interesting observation. If the same hex file is converted to a .com file using the same CP/M LOAD command on different occasions, the resulting .COM files may end up with different checksums depending on the garbage data that was left in uninitialized memory. This bothers me - but I'll get over it. Getting back on track…

The hex2com program will pause and report any checksum errors that may exist in the hex file. It will still perform the conversion in the event someone wants to modify a few parameters in the hex file but does not want to go to the trouble of computing a new checksum. This can be handy since the CP/M LOAD command will not accept checksum errors.

3. Dump to Hex (dmp2hex)

The dump to hex program will take the output of either the fil_dmp program or the CP/M DUMP command and convert it to a hex file. The program has no way of knowing if the data it is converting is uninitialized data or actual executable code. This means that the garbage data found in the uninitialized data in the .COM file will be put in the resulting .hex file. Because of this, the hex file will be a bit larger than the hex file created by the CP/M ASM command. By generating a hex file from the dump output, when the CP/M LOAD command operates on it, the resulting .COM will be identical to the original .COM file including the garbage in the uninitialized data area.

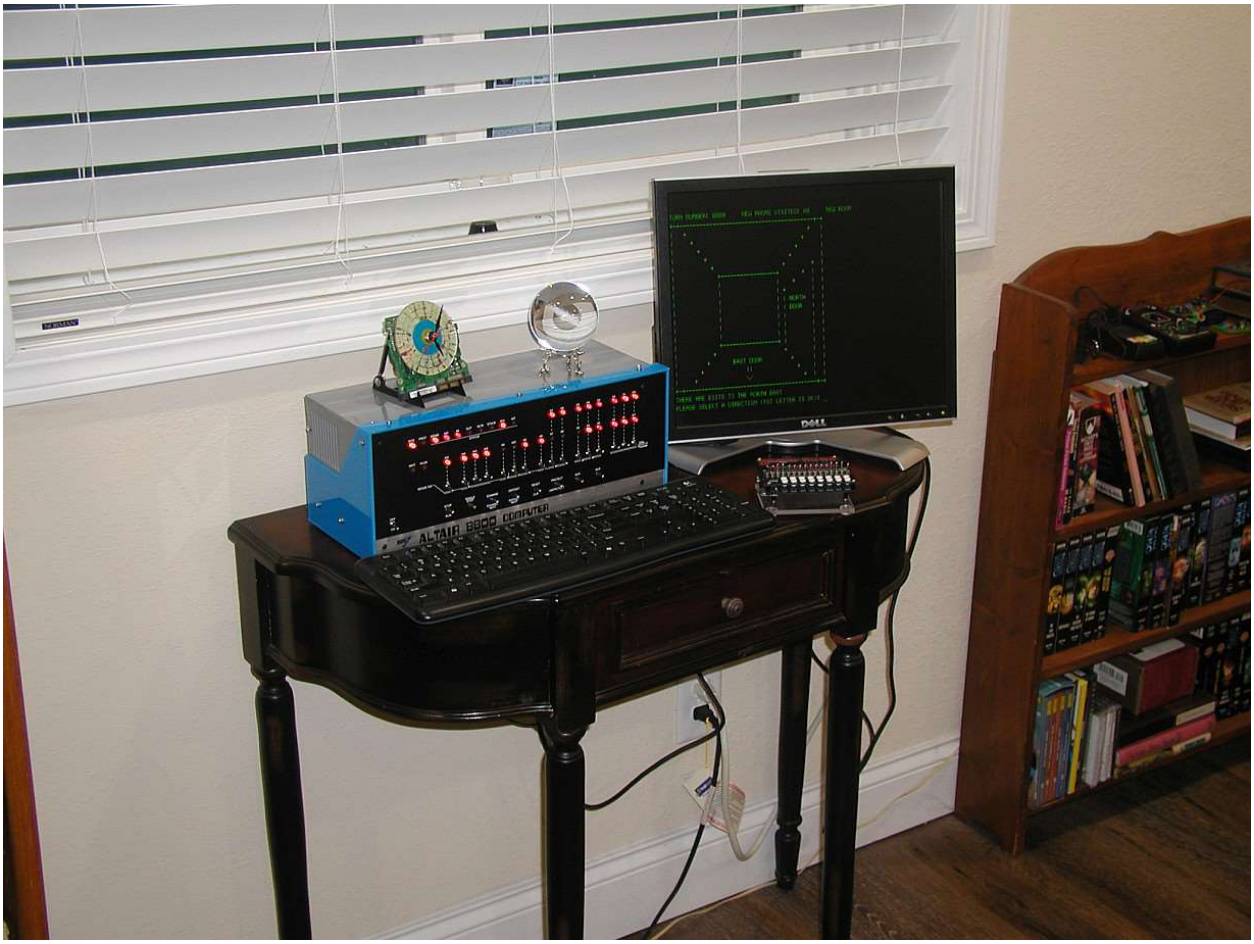
As stated earlier, this might be a good way of transferring other .com files that are acquired from the internet to your CP/M machine by using the fil_dmp program to dump the .com and the dmp2hex to convert the dump output to a hex file. The hex file can be transferred as a text file to the CP/M machine and then converted to a .COM file with the CP/M LOAD command. If I do very much of that, I would be inclined to write (or find) a new tool that would take the .com file directly to a .hex.

One of the observations I made while examining the dump output of a .com file is there is no address information. It is just the data. The column on the far left of a dump output is put there by the CP/M DUMP tool (and was added as one of my modifications to my fil_dmp program). That number is simply a byte count presented in 4 hexadecimal digits.

CP/M wants to load all programs to 0100H. The dmp2hex program assumes you want to also, so the address will always start at 0100 in the first record of the hex file and will increment accordingly for each record thereafter.

Conclusion

It is probably appropriate to end this document with some concluding remarks. There is nothing really profound to add here other than I had a blast writing this game as well as playing it from time to time when looking for some mindless entertainment. I did achieve my goal of using the front panel lights and switches for debug. This proved to be very helpful and somewhat of a unique experience. So far, the only fan of Maze029 is me. My youngest grandson does not want to play it and calls it tiresome.


ALTAIR 8800 COMPUTER
DELL