**Loader Zone Details**

**Overview**

The world map is organized into 64 loader zones. There are several benefits to the loader zone design including speed, scalability, and flexibility.

This section will discuss the technical specifications of loaders zones, the benefits, and take a deep dive into the underlying mechanics which is useful for troubleshooting problems.

See Chart 20.0 for an illustration of the data flow associated with loader zones.

**Things I Frequently Forget**

1) ZONE.LOOKUP.LO doesn't usually begin on a page boundary even if .NO directive is used to position the arrays defined above if in offloaded\_variables2.ASM, which is typical. This is because ZONE\_TOOLS.INPUT\_BUFFER has a .BS size of $101. As a result ZONE.LOOKUP.LO will usually begin on $01 byte after the page boundary it might seem like it would start on.

2) When manually copying compressed zone data from AUX memory to ZONE\_TOOLS.INPUT\_BUFFER during troubleshooting, keep in mind that zones with a high compression ratio do not take many bytes AND that ZONE\_TOOLS.INPUT\_BUFFER will contain leftover data from earlier zone activities if the compressed data copied from aux memory takes up less than a full page (guaranteed if the zone has high compression).

Here is the most important point. A zone of all deep water will take 3 bytes: 00,8D,00. The first 00 means 256 tiles, and the 2nd 00 is the stop value. If you displayed the first 50 or even 16 values of ZONE\_TOOLS.INPUT\_BUFFER on screen, these first 3 bytes could blend in with the leftover data in ZONE\_TOOLS.INPUT\_BUFFER, which may look familiar since it is from other zones. As a result, it could lead to confusion, and a perception that the data expected is not there, which is a big waste of time.

**Technical Specifications**

Each of the 64 loader zones each with 16tiles x 16tiles. With the Tile\_ID for each tile using 1 byte of memory (uncompressed), each zone uses 256 bytes total, neatly fitting into 1 page of memory.

There are 9 loader zones stored (uncompressed) in main memory at all times, with the rest stored compressed in auxiliary memory. Which 9 zones are stored in main memory is based on the players location on the world map. This group of 9 zones in main memory is the regional map and the tile data for it is stored in RZONE.ARRAY.

The world zones are stored in auxiliary memory and the tile data in them is usually compressed, but not always. WZONE.COMPRESSION.FLAGS tracks the compression status of each world zone. The reason some world zones are not compressed is because they do not have enough strings of horizontally consecutive Tile\_IDs for the compression to pay off. Said another way, the result would be that the compressed data would take up more bytes of memory than the uncompressed data. On a side note, vertically (from the map perspective) consecutive Tile\_IDs are not compressed because to the computer the map is a continuous linear series of numbers. The assignment of those numbers to a rectangle map is a layer of abstraction.

**Where Are Those World Zones Anyway?**

An interesting challenge when creating this architecture was the fact that the uncompressed data was always a known size (256 bytes) and the compressed data is an unknown size. How do the zone functions that uncompress the data in auxiliary memory and copy it to main memory know where the start of each zone is in auxiliary memory? The answer is that a lookup table is created (ZONE.LOOKUP.LO/HO) by the routine that compresses the data (WORLD.COMPRESS2). Since WORLD.COMPRESS2 runs during game boot (GAME.LOADER1), before GAME.BIN is loaded into memory the lookup table has to be created in a temporary location. GAME.LOADER2 copies the lookup table from the temporary location to its permanent location because GAME.LOADER2 clobbers the temporary location by loading VARIABLES.BIN.

**Zone Transitions**

The routines for player movement (MOVE.NORTH/SOUTH/EAST/WEST) in Movement\_manager.asm monitor the players position within the regional map and if the player moves 16 tiles from the center of the regional map in any direction, that triggers a zone transition.

A zone transition results in tile data in RZONE.ARRAY being scrolled and 3 new loader zones being uncompressed from auxiliary memory and copied into the appropriate regional zone slot (0-8, representing the 9 zones in memory), based on the direction the player was moving when the zone transition occurred. After a zone transition, the player's RMAP is adjusted to the center of the regional map, the exact location on the world map the player was at before the zone transition (note a zone transition happens during the processing of a player move so technically there may be a 1 tile difference in the players location on the world map, depending on whether the players move was blocked).

Currently there is no code to handle a situation where a zone transition results in the player "moving off the map". However, it would be very straightforward to implement a wrap around world reflecting the geometric shape of a torus, which was a common wrap-around world shape in Apple II games.

**Benefits**

*Flexibility.* Loader zones allow the code to work with tiles in chunks the size of 1 page of memory, which results in more efficient code. Loader zones also provide a straight forward path to wrap around worlds as described in technical specifications, which are impractical if the tile data is stored in one large array.

*Scalability.* If the tile data were stored in one large array, at a certain world size the array would be too large to store in main memory. Without a system in place (like loader zones) for breaking the tile data into smaller chunks, offloading tile data to auxiliary, bank-switched memory or disk becomes problematic.

*Speed.* If tile data were stored in one large array, accessing the tile data would be slower. Tile data access occurs in the innermost loops of the graphics engine where speed is extremely important.

**Deep Under the Hood: Troubleshooting Guide**

**Introduction and Big Load of Nonsense**

The one challenge of loader zones is that it inserts several layers of abstraction into the code and remembering in detail how it all works becomes challenging. There is nothing worse than tooling around the ocean and suddenly encountering a bizarre unchartered landmass of mountains, grass, a jester (wtf?) and boats....lots of boats....dozens of boats....reminiscent of walking around on the Ultima surface world with the underworld disk in the drive.

Okay, there is something worse. Encountering a goofy cluster of tiles like that which aren't supposed to be on the map, and NOT having my loader zone troubleshooting guide handy.

I mean, you could consult the usual literature, but where will that get you? You could consult the Spates Catalog, but it's not even current, it hasn't been update since 1984 ffs. Yes, my loader zone troubleshooting guide is the best thing to have when a ghost island appears short of your own personal copy of Tobin's Spirit Guide.

**Getting On With It......**

See Chart 20.0 for an illustration of the data flow associated with loader zones.

If tiles appear on the screen that are not in the source map tile data (for me this is a spreadsheet map editor which compiles the Tile\_IDs into hex tables), then there are three general areas the root problem can be in:

1) A problem with the tile data at some point after it was loaded in into RZONE.ARRAY. In this case the RZONE.ARRAY would contain the correct tile data and SCREEN.TILE.DATA would contain incorrect tile data.

2) During game launch, tile data was uncompressed and copied from auxiliary memory incorrectly. (REGION.UNCOMPRESS.ALL)

2a) This problem is sometimes caused by incorrect values in ZONE.LOOKUP.LO/HO, which contains the starting auxiliary memory address for each zone.

3) During game boot, tile data was copied from main memory to auxiliary memory incorrectly. (WORLD.COMPRESS2)

4) During game boot, tile data was loaded from disk into main memory incorrectly (CUSTOM BOOT2). (*note: currently the tile data is stored in hex tables in the source code file. Custom boot2 is the boot stage at which the .bin file containing those hex tables is loaded from disk into memory. Eventually the tile data will be stored on disk compressed, possibly still has hex tables in the source code)*

Typically I either start by ruling out scenario 1 or 2 and work my down this list. The following are the key areas of code to check for each scenario.

**Troubleshooting** **Scenario 1**

Exit the game to the Apple Monitor (Q) and check the location in the RZONE.ARRAY associated with the map locations which had incorrect tile data on screen. Most likely a prerequisite step will be to change the definition of RZONE.ARRAY to start it at the beginning of a super page boundary (i.e $A000) or at least a page boundary, so that calculating the memory addresses associated with a particular row of tiles is easier using this chart <INSERT>.

If the tile data in RZONE.ARRAY is correct, check SCREEN.TILE.DATA. If that's correct as well then the problem likely has nothing to do with tile data is more likely related to the drawing routines. A variables might be getting clobbered by something overwriting it's memory address.

If the tile data in RZONE.ARRAY is not correct, then continue troubleshooting closer to the tile data's source (Scenario 2)

**Troubleshooting** **Scenario 2**

Check the tile data in RZONE.ARRAY immediately after the JSR to REGION.UNCOMPRESS.ALL during game launch (search for this routine in GAME\_LOOP.ASM)

If the tile data in RZONE.ARRAY is not correct, this rules out the possibility that the data was corrupted while the game was running, post-launch. In this event, check RZONE.ARRAY immediately before the JSR to REGION.UNCOMPRESS.ALL.

If the tile data in RZONE.ARRAY is not correct, then continue troubleshooting closer to the tile data's source (Scenario 3)

If the tile data in RZONE.ARRAY is correct, that points to a problem occurring when the tile data was uncompressed and copied from auxiliary memory which occurs in REGION.UNCOMPRESS.ALL. This routine contains a loop which does a JSR to ZONE\_TOOLS.UNCOMPRESS.SINGLE for each of the 9 regional map zones. I usually troubleshoot next in ZONE\_TOOLS.UNCOMPRESS.SINGLE.

The first thing ZONE\_TOOLS.UNCOMPRESS.SINGLE does is copy the compressed zone data from auxiliary memory to a buffer in main memory (still compressed). The buffer is ZONE\_TOOLS.INPUT\_BUFFER.

Put a break in just after that AUX memory copy and check the buffer to see if the compressed tile data is correct. If it's not correct, then investigate the close to the source of the tile data. (Scenario 3). However, before diving into Scenario 3, take a close look at the values in ZONE.LOOKUP.LO/HO (Scenario 2a below). Often times problems at this stage are due to incorrect values in the lookup tables, resulting in an incorrect starting memory address when ZONE\_TOOLS.UNCOMPRESS.SINGLE copies the compressed tile data from auxiliary memory to main memory.

**Scenario 2a**

As mentioned in the Technical Specifications, ZONE.LOOKUP.LO/HO is created when the tile data is compressed by WORLD.COMPRESS2 (GAME.LOADER1). It's stored in a temporary memory location and then copied to a permanent memory location by GAME.LOADER2.

Thus, a good next step is confirming whether the values in ZONE.LOOKUP.LO/HO remain the same in 3 locations:

\*In ZONE\_TOOLS.UNCOMPRESS.SINGLE, just before the auxiliary memory copy at the start of the routine.

\*in GAME.LOADER2, just after & before the lookup table is copies from its temporary memory location to its permanent memory location.

\*In COPY.ZONE.TABLE (GAME.LOADER1) just before the lookup table is copied to its temporary memory location.

If the values are different in one of these locations, investigate the change that's likely the source of the problem.

If the lookup table values are the same in all these locations then proceed to Scenario 3.

**Troubleshooting** **Scenario 3**

WORLD.COMPRESS2 contains a loop that calls ZONE\_TOOLS.COMPRESS.SINGLE2 for each zone, 64 tiles total.

ZONE\_TOOLS.COMPRESS.SINGLE2 uses a zero page pointer to designate it's input source as one of the hex tables in main memory (temporarily) that contains the uncompressed tile data for each zone. Eventually this will work differently and the compressed data will be read in from disk.

For now, verify the address in the zero page pointer, and confirm the data there. Then check the output buffer. This should be done for the zone where the unexpected tiles were detected. This can be done using a conditional break based on the value of Y-REG which is used as the loader zone counter. Note that Y-REG gets pushed to the stack at the start of ZONE\_TOOLS.COMPRESS.SINGLE2 and promptly initialized for a different use, so that must be taken into consideration when setting up the conditional break.

If the zero page pointer contains the correct memory address for the loader zone being examined, and the data at that memory address is not correct, then there may be a problem during the process of loading the tile data from disk to main memory.

**Troubleshooting** **Scenario 4**

Presently the tile data is loaded from disk by CUSTOM BOOT2 when LOADER.BIN is loaded at $1A00.

If the troubleshooting steps in Scenario 3 indicate the tile data is not arriving in main memory as expected, then the problem is likely either:

1) the hex tables in the source code contains incorrect data.

2) the wrong track and sector are being used to read in LOADER.BIN

#1 Check the values at the very start of a hex table and compare it to what is in memory. If it matches then continue to consider that the source code may contain incorrect data (check the map editor). If it doesn't match then the problem is more likely that the data isn't being read from disk correctly.

#2 Check the track/sector being used by CUSTOM BOOT2 when it calls MY.RWTS. Compare that to the track/sector where LOADER.BIN is stored using CiderPress. Check the .ORG statement in the source code for LOADER.BIN and compare it to the memory address used by AppleCommander when writing LOADER.BIN to the disk image.

One possible reason that the track/sector CUSTOM BOOT2 is using may be wrong is if LOADER.BIN increased in size. Since GAME.LOADER1 has a .NO directive at the end which allows the file to stay the same size as code is added, the only reason this should happen is if the .ORG statement at the start of the file was changed to make more room at the front end. However, when the size of a file on disk changes that will change the track/sector used by all files located on higher tracks/sectors. Accordingly, if the size of LOADER.BIN changed that would affect the size of GAME.BIN and VARIABLES.BIN, which would require changes to the tracks/sectors in GAME.LOADER1 to accommodate or the game should crash before displaying the first graphics screen. Thus, if making these changes doesn't sound familiar then most likely a previous troubleshoot step was missed or misinterpreted.

For more details on the how, when and where the files are read from disk see BootLoader.Code\_Walkthrough and BootLoader.Summary.

The following are instructions for verifying the track/sector used by a specific file on a disk image.

Open a disk image using CiderPress

Select Disk Sector Viewer from the Tools Menu

Click Open Current Archive

Click Open File

Enter the name of a file on the disk image (note: this is the name specified by AppleCommander. My convention was to drop the .BIN used in the file created by SBASM. So LOADER.BIN would be named LOADER on the disk image, for example)

The starting track and sector of the file appears in the upper right corner. click the "Read Next" button until the last sector is reached (the sector will stop incrementing when this happens). Viola! We now know the start/end track and sector of the file. Comparing the hex data on the ciderpress screen to the SBASM machine code output is when I first noticed the 4 byte header. The hex data on the ciderpress screen is exactly what RWTS reads in.