|  |  |  |
| --- | --- | --- |
| Pokémon Diamond and Pearl: Void Routing Guide | | |
|  |  |  |
|  | | |

# Table of Contents

1. Introduction
   1. The Void Glitch
2. Resources
   1. Scripts and Dumps
3. Entrance Points
   1. Tweaking Glitch
   2. Surf Glitch (JP exclusive)
4. Void Glitch Basics
   1. Traversing through RAM
   2. BSOD, crashes and softlocks
   3. Basic routing techniques
      1. Static Values
      2. Mapdata
      3. Matrix ID routing
      4. Chunk Dislocation
      5. Fake Sinnoh
5. Wrong Warps
   1. Main Wrong Warp
      1. The 3 Fake Sinnohs
   2. Overworld Wrong Warp
   3. Japanese Exlcusive Wrong Warp
   4. Spear Pillar Wrong Warps
   5. Other Wrong Warps
6. Battle Tower Void
   1. Routing to indoor areas
7. Tilewriting Basics
   1. Explanation
   2. Applications
   3. Manipulation
8. Ledgecancelling
   1. ASLR setup
   2. Consistent setups
   3. Chunk Storage
   4. Walk Through Walls
   5. Window Map Swapping
9. Advanced Tilewriting
10. Script Execution
11. Troubleshooting
12. Glossary

# Introduction

* 1. The void Glitch

The Void Glitch is the most infamous glitch in the gen 4 Pokémon games, Diamond and Pearl. By exploiting glitches it is possible to reach the out of bounds areas of the game. This area appears black[[1]](#footnote-1) and was therefore dubbed the Void. Through the years, this glitch has been researched thoroughly and has allowed players to capture unreleased event Pokémon like Darkrai, Shaymin, and almost a decade later even Arceus.

After the hunt for Arceus was over, the goal became to break open the gen 4 games as much as possible. Voidroutes became more optimized and new glitches continued to be found. In this paper I will attempt to explain how the Void behaves and how it can be exploited. The aim of this paper is to preserve the research the Voidmatrix and Hall Of Origin groups have conducted.

# Resources

2.1 Scripts, Dumps, and documentation

Since 2017 the research in gen 4 games has become increasingly more technical. As a result, scripts have been written by multiple members to assist in the finding of exploits. In the following section a list of useful scripts, dumps and similar resources are listed along with the authors of the works.

1. Routing Script for Dummies. / *RSFD.lua*   
   This script has combined all the most important resources from previous scripts and has added several new applications. It is compatible with DPPt and HGSS.  
     
   Author:   
   RETIRE  
   Co-Authors:  
   MKDasher: *Void.lua* and *Loadlines.lua*  
   Ganix: *VET.lua*  
   Martmist: RETIRE execution
2. DeSmuME  
     
   While this emulator is generally disliked because of its emulation of inaccessible ram as echo ram and other emulation issues, it is what we use for our routing.
3. *RNG.lua*   
     
   This script shows the possible encounters of wild Pokémon in the area. It also shows the stats of your party Pokémon and opponents.   
     
   Author: MKDasher
4. Mapdata Dump  
     
   This is a dump of NPC and Warprelated data that is loaded when entering a new Map ID. It is heavily utilized in Void routing.  
     
   Author: Ganix
5. Mapscripts Dump  
     
   This is a dump of mapscripts, and what their ID in runtime is. It is an essential document for understanding Script Execution.  
     
   Author: Ganix
6. RAM documentation  
     
   Author: MAP  
   Co-Authors:   
   Martmist  
   RETIRE  
   FlederKiari  
   Ganix
7. Script opcodes
8. Diamond Decompilation

# Entrance Points

* 1. [](https://www.youtube.com/embed/_-DShT1GLyY?feature=oembed)Tweaking Glitch

The tweaking glitch is the only currently known glitch that allows the player to enter the void without prior setup on all versions of Diamond and Pearl. It abuses the loading of new chunks in the game. At any given time, exactly 4 chunks should be loaded. By crossing the loadlines the game uses in quick succession, it is possible to misplace chunks.

You cross a loadline to write a chunk into a temporary buffer. You then cross more loadlines to delay that buffer from being written to the designated section in memory. After you stop moving the game will catch up and write the buffer to memory. However, it will overwrite the last loaded chunks as a result. In the video, you see the most commonly used tweak. If you slow the footage down, you can see the left area loaded correctly for a split second before being overwritten by the chunks that were still in the buffer.

This tweaking then allows you to move through this irregularly located chunk, which is called an Abyss. To fix this glitch, you simply must reload the chunks. This can be done by opening and closing menus, saving and resetting or crossing new loadlines. To go out of bounds, you want to move to where a door should be located under normal circumstances. You then open and close a menu to load the correct chunks, and you can enter the door from above. Doing this will load the player in the building, but will automatically move you 1 step down, into the out of bounds area called the Void.

* 1. Surf Glitch (JP exclusive)

The Surf Glitch is an oversight that is present in the Japanese Games but was patched out from the international releases. In the chunk data of all Japanese Elite 4 rooms of Diamond and Pearl, a single water tile is present on the bottom door. You cannot directly interact with it by pressing A. This is because the bottom door is an NPC, and it has priority over interacting with the tile it stands on. However, you can still open the Pokémon menu and select surf there, which will allow you to surf into the door and hop out of bounds.

There is one more entrance point for the Japanese game which is explained under   
5. Wrong Warps

# Void Glitch Basics

* 1. Traversing through RAM

The Void is essentially the game’s RAM. It interprets the game’s memory as Map IDs, and therefore any action you take directly influences the Void. By loading the correct data, we can manipulate what Map IDs appear, and therefore we can route to essentially anywhere in the game.

Every Map ID is a 2 byte value, between 0 and 65535. The valid Map IDs range from 0 to 558. Everything beyond that gets converted automatically to Map ID 3, which is Jubilife City.

Moving through the Void is like moving in an Excel grid, with a cell holding a 2 byte value called the Map ID. Every 32 steps you take, the game will read from a new cell. If you move left or right you’ll move a single cell or 2 bytes.

When it comes to moving up or down however, there is an additional variable. It will take into account the width of the loaded in bounds area, or Matrix, and divide it by 32. That variable is known as the Matrix Width. For example, most houses have a dimention of 32x32, or one cell. Therefore the Matrix Width will be 1. When you have a bigger loaded Matrix, for example Safari zone which is 128x160, the Matrix Width will be 4. For Sinnoh, by far the largest area with a size of 960x960 tiles, this will be 30.

The amount of bytes you travel vertically will be the Matrix Width multiplied by 2 bytes, since a Map ID is a 2 byte value. Every 32 steps up or down you would move 2 bytes, 8 bytes or 60 bytes (1 cell, 4 cells, or 30 cells vertically) depending on your Matrix Width. Other Widths also exist, but these serve as examples.

This means that ,in a house, going 1 cell down is the same as going 1 cell right. In a bigger area, going 1 cell down equals moving 4 cells right. In Sinnoh, going 1 cell down equals moving 30 cells right. Because of this, you should prioritize routing with a Sinnoh Matrix, because you'll move 30 times further in memory when moving vertically.

To change Matrix, you simply have to save and reset in any Map ID that is part of that Matrix. For Sinnoh, that includes any City or route, but for most other Matrixes, there is only 1 Map ID that can be used.

If this was a lot of information, luckily this is all visualized when using the *RSFD.lua* script. It will show the current position in RAM, and allows you to colorcode the Map IDs that are important to you.

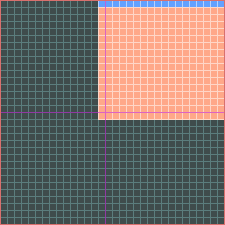
* 1. BSOD, crashes and softlocks

Before we start routing there are still some things we need to address. Most importantly, different causes for crashes and softlocks. A BSOD, or Black Screen of Death, is an effect that can permanently make a save file unplayable. Luckily, it is entirely preventable. A BSOD is caused by entering Map IDs known as ‘BSOD Maps’, which load 3D models that do not get removed when loading new Map IDs. If you attempt to load graphics after such a model has been loaded, the game will (usually) crash. The graphics are loaded when selecting a save file, therefore saving with these models loaded can make that save file unplayable. To prevent a BSOD from happening, simply do not enter any of the ‘BSOD Maps’. They are colored dark red in the *RSFD.lua*.

A loadline crash can occur when moving over a loadline in a void with a small Matrix, in specific directions. For example, moving left in a Pokétch Co. Void is perfectly fine, but moving right over a loadline will crash the game. This is not always easy to get around.

You also should look out for cutscenes. If an NPC should be moved that is not loaded, you crash. This can occur if a part of the cutscene moves you out of your Map ID, and the one you enter does not have enough NPCs to support the rest of the cutscene.

Softlocking can occur when the player moves into tiles that have a collision ID above 0x80. In the void, luckily, the 4 chunks that are always loaded only have a single one that doesn’t read 00’s. By default, it is the top right subchunk in a Cell.

* 1. Basic routing techniques

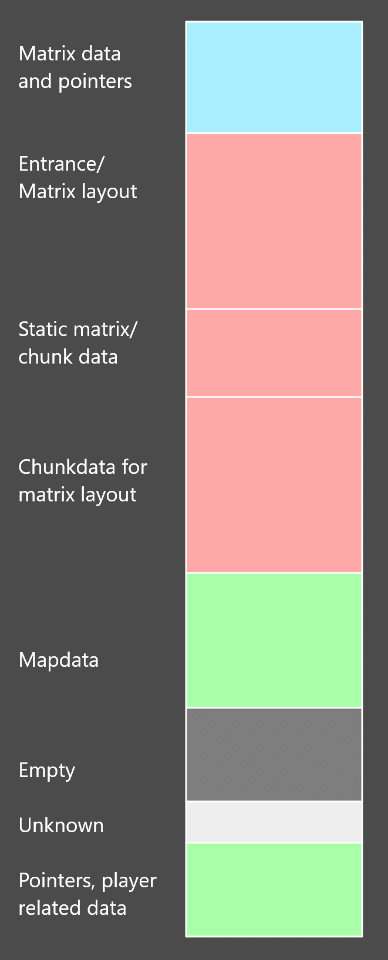
In the next section, we put this knowledge into play by writing some simple routes. For this, you will need the *RSFD.lua* as well as the *Mapdata.docx* dump.

In the first basic routes, we will be utilising static values. The game however does not only read Map IDs from memory, but also the chunks.

In any cell we will pass in the first memory section, the left and bottom section are walkable, but the top right chunk will be filled with other tiles. Later on in the guide, you will learn how to manipulate tiles. For now, simply avoid the chunk by moving on the left and bottom sides of cells.

The architecture of a chunk in the void

To toggle seeing the loadlines, press shift+L the while the *RSFD.lua* is active.   
To toggle seeing the sides of the cell, press shift+ K.

Earlier I mentioned that the void is RAM. That means we can read any value in the game as a Map ID, if we wanted to. But sometimes those values could take hours or even days of walking to reach. Next to that, you’d be walking through a ton of unrelated data that could end in softlocks or crashes.  
  
Instead, most routes will stay within the data sections closest to the entrance of the void. We will soon learn how to route using them, in progressive difficulty. I provided a visualization of these sections, which will be used as a reference throughout the guide.

Legend:

**Blue**: some data is manipulable, but it's mostly pointers.  
**Red**: the data in this section is static, doesn't change.[[2]](#footnote-2)  
**Green**: this section is fully or mostly manipulable.  
**Black/grey**: Empty or never used for routing.

* + 1. Static Values

To get the gist of things, we’ll first write a very simple route using values that won’t change by taking any actions, and simply require us to walk to them. A good example for this would be Map 516, which is the elevator in Lighthouse Vista, Sunyshore City. When you enter an elevator, the game moves you and automatically teleports you to where it should drop you off. This will also occur in the void, because this is handled by mapscripts. They are good as a first introduction to void routing.

In 99% of cases we start from the Pokétch Co. void entrance. This is an easy and commonly known entrance point and except for edge cases, it doesn’t matter at all where you enter the void because save resetting changes your Matrix anyway. And as mentioned before, we’d like to use the Matrix of Sinnoh, since it lets us move the maximum of 60 bytes through RAM when moving vertically.

So, the first thing we want to do is save in a Map ID that is a part of Sinnoh’s Matrix. Luckily, getting to Sinnoh’s Matrix is by far the easiest thing we can do. Because all Map IDs above 558 get error handled to be Jubilife City, which is part of Sinnoh. And RAM has plenty of high values. We’ll be using a part of a pointer in this case. Load up your *RSFD.lua*, we’re getting started.

Simply perform the following steps (from Pokétch Co.):

The first 3 steps are simply to get around the building. After that, we start moving left, through some Matrixrelated data such as the Width, height, ID… And eventually end at 415 W, which will be a Jubilife. There are more Jubilife left of us, but you should not use those because there is a small chance on Non-English European games that one of the pointers will return a BSOD Map, and permanently softlock the game. Use your bike in the Mystery Zone.

1 S  
17 W  
14 N  
  
415 W

save reset

Currently we are at the complete bottom of ‘Matrix and Pointers’, marked blue on the RAM Map. The static value 516 that we want to get to is located in Static Matrix/Chunk data, marked red. That means we’ll have to pass through the Matrix layout, which should be quite recognisable, as it is shaped like Sinnoh (since it actually is Sinnoh). If you press the ‘CENTER’ button followed by the arrow keys at the top left of the screen you can see what data is present in RAM. Left press will move you once, right click will move fast through RAM. Play around a bit, pressing Manual will reset it to your current position. Under your current position you should see Sinnoh’s Matrix.



If you press right till it stops you from going further and then go down, at some point you’ll see 2 yellow cells containing the values 516, our goal. Entering the Map from above would briefly push us into Jubilife City, loading the mapheader. This would then crash us during the warp. Instead, we will aim to enter the Map from the left.

So, we want to get to the row of data left of the 516. Create a savestate. Now, use the shift+directional keys to teleport an amount of steps. The steps can be changed by pressing J. Press W for walk through walls. Then you just teleport until you are left of 516, note down the x coordinate, which is from X = 65281 to X = 65312, and then go back to the earlier savestate and disable walk through walls.

Now, all we need to do is get to the X noted down. As I stated in [4.3 Basic Routing Techniques](#Basic_routing), the left and bottom side of the cell will be walkable. So we will simply move to the first tile of row we want, meaning we go to X = 65281, which is 161 E, in the Underground Map. I recommend making a savestate here. After that we go down until we’re at the bottom of the cell next to 516. We only need to go until we’re under the loadline, so press shift + L to see those. This means we’ll go 1329 S. Since We’re at most left side of our current cell, the next cell will be exactly 32 steps right of us, so we go 32 E to enter the cell containing 516. Since the elevator plays a cutscene, the last step won’t be counted on the stepcounter. When writing routes, it’s therefore usefull to say 31 E, 1 E (warp) or something similar. this avoids confusion.

Now our first basic route has been created. The final route goes as follows:

Being consistent in our notations and giving additional information on specific special events such as warping and cutscenes can help people use your routes. The end product should always be easy to consume.

This route used static data, which meant all we have to do is move towards this data. But only a small portion of Map IDs are available in this manor. The next step is manipulating values in RAM. Since Map IDs read RAM, manipulating RAM lets us write Map IDs. With the right tools, we can anywhere.

1 S  
17 W  
14 N  
415 W

save reset

161 E  
1329 S  
31 E  
1 E (Warp)

* + 1. Mapdata

Mapdata is a commonly used section of memory to manipulate. It is the closest section of memory in the void that can be fully manipulated. In this section of memory, the game loads NPCs, warps, triggers and their corresponding data when you enter a new Map ID. Things such as sprite IDs, flags, events, x and y coordinates are all loaded here.

This data does not get cleared when you enter a new Map ID, it will simply load in the NPC’s and warps of the new Map ID. Remember, the Void is RAM, so you can read this data as Map IDs. And by doing that, you will load even more NPC’s and Warps. This results in a chain of writing data followed by interpreting this data back as Map IDs.

First, to reach this section of memory you want to follow these steps from the Pokétch Co. entrance.

You might notice that while going 2097 S, you go through what looks like Sinnoh. Well, that is because you are 30 cells left of where Sinnoh is. If you remember from earlier, moving 1 cell down is the same as going 30 cells right. This also means that if you go 30 cells left, you are at the same place in RAM as if you just went 1 cell up. So you are simply reading from the section of RAM that is used for Sinnoh.

Under that, you go through a different Sinnoh, but this time it’s the chunkdata of Sinnoh. After that we reach Mapdata.

1 S  
17 W  
14 N  
446 W

save reset

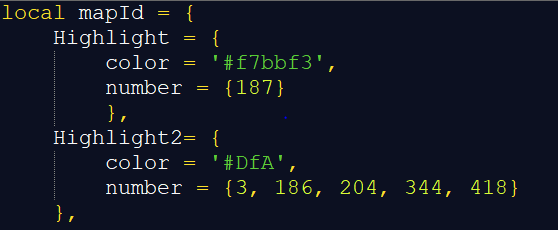
2111 S  
147 W  
320 S

What we will do now is route to a Map ID. As an example, we will be routing to Map ID 187, the Hall of Fame. Similarly to elevators, the Hall Of Fame is a Map that uses Mapscripts to automatically load a cutscene. Since the Hall Of Fame puts you into your house in Twinleaf Town after saving, this should be an easy route too.

First, we need to find out what Map ID(s) load 187 into memory, as a warp, coordinate, or flag of some sort. For this, open *Mapdata.docx* and press CTRL + F, then put ‘187’ with a space in front and after it. If you select ‘Headers’ you will find the Map IDs that load that value into memory. They should be highlighted yellow.

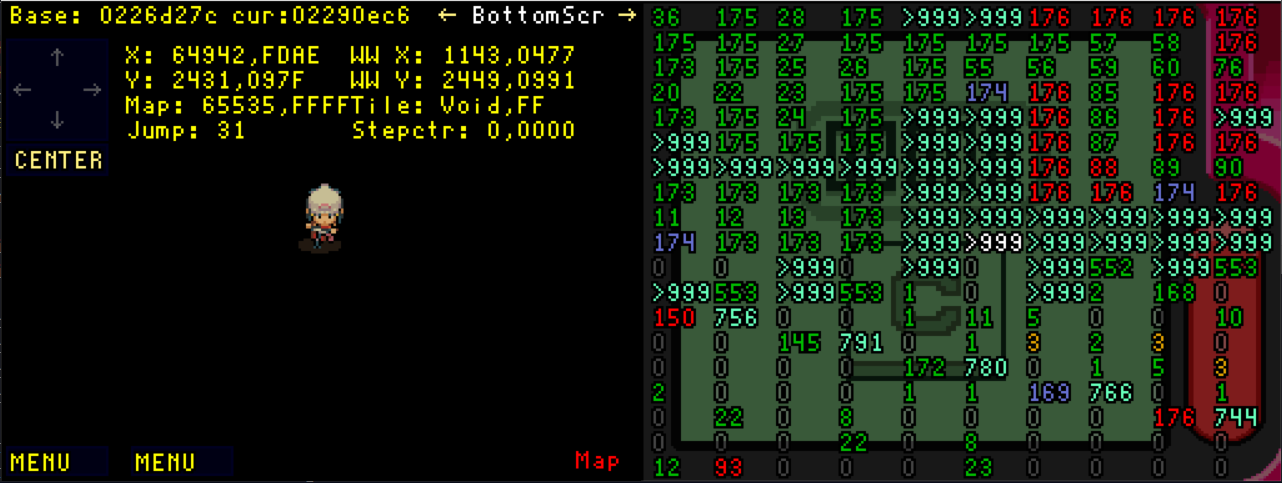
The Map IDs that should pop up are 3, 186, 204, 344 and 418 . What you do next is you open *RSFD.lua* in any prefered text editor and insert 187 as ‘Highlight’ and the other Map IDs as ‘Highlight 2’. This will change the colors in which they’re displayed in the Lua script.

This is how the Map IDs should be implemented in the script.   
  
You can also edit the colors. By default, Highlight is pink. Highlight 2 is orange.



With this set up, save and it should automatically update in DeSmuME.   
One important thing to note is that Map ID 3 is in our list. Since all Map IDs above 558 are considered to be 3, they will also work for our purposes.

This should currently be displayed on your sceen.



All Map IDs colored lightblue are considered Jubilife City, Map ID 3. Starting one cell below us, we’ll be in Mapdata. That means that, since we are in Map ID 3, our desired Map ID should already be loaded in there. You can look for a pink 187, it is located on the left side.

Just like with elevators, the Hall Of Fame Mapscript pushes you up before the game finishes (The walk towards the Hall Of Fame machine). The game would once again crash if we were inside a different Map ID if this occurred, so we’ll enter slightly below from the side. First, we create a save state and name it Mapdata. After that, we go left until we’re one cell to the right of 187. We want to be at the left side of the loadline, so we go 320 W. Now you just go south until we are in the cell next to 187. On our way down 161 S we go through Map ID 15. When this occurs, data in Mapdata changes and 187 is gone. There is a map 3 that is loaded one cell to the right of us, so we could write 187 again by entering that. Simply go 17 S and 19 E to enter it. As we can see, the 187 get’s written once again. But clearly, now our route is no longer optimal, we’re adding 19 steps E and an additional 19 W to go back.

If we go back to our savestate, we can optimise our strategy. Instead of going down when we are 1 cell right of Map ID 187, we’ll go down when we are 2 cells right of it. From our savestate, instead of going 320 W, we go 288 W. Under us is only a Map ID 3 and a Map ID 8. Map ID 3 will just write 187 again, and Map ID 8 won’t overwrite enough data to reach our Map ID 187. We go 242 S and nothing significant changed. From here, we go 45 W, and then 1 W to enter the Hall Of Fame.

It is important to remember that sometimes a Map ID can load enough data to overwrite the values we need for our route. You will need to learn how to manage savestates and getting around Map IDs efficiently. The higher up your desired value is, the more potential Map IDs can overwrite it. A selection of Map IDs can always be used, such as Mystery Zones which barely load any data.

This was still a pretty straight forward route because 187 is easily accessible. Jubilife City is everywhere due to the error handler. However, a lot of values require chaining. This technique can be a lengthy process. You write down every Map ID that writes the Map ID you want to reach. Then, you also search which Map IDs write those, until we get a Map ID we can easily access in the void. The best example of this is the route to capture Arceus. To capture Arceus we will exploit a trick known as the RETIRE trick. By entering Pal Park, we get the Pal Park menu, and the RETIRE option will execute 4th script in runtime. In the Hall Of Origin, the 4th script in runtime is the battle with Arceus. This is purely a coincidence, and interestingly, they put the battle script in runtime twice, which is never seen elsewhere.

So, in order to make a route to Arceus we need to enter Pal Park (Map ID 251) to get the menu and Hall Of Origin (Map ID 510) to run the script. We will look for a chain, Map IDs that when entered consecutively will write our desired Map IDs. We need to find a low value, as those are commonly found. I highlight the chosen values in green so you can follow the chain I chose. These chains can be made by referencing *Mapdata.docx* like we did for Hall Of Fame. Usually multiple chains are possible, but avoid BSOD Map IDs, which I highlighted red.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ch1 | 251 | 393 | 392 |  | Ch2 | 510 | 316 | 45 |  |  |  |  |  | 306 | 392 |
| 251 | 393 | 251 | 2 |  | 510 | 316 | 45 | 33 | 60 | 244 | 318 | 383 | 528 | 45 | 2 |
|  |  | 392 | 354 |  |  |  | 65 | 35 | 61 | 245 | 326 | 395 | 529 | 65 | 354 |
|  |  |  | 383 |  |  |  | 314 | 45 | 64 | 246 | 344 | 466 | 530 | 305 | 383 |
|  |  |  | 393 |  |  |  | 315 | 46 | 155 | 253 | 347 | 521 | 531 | 306 | 393 |
|  |  |  | 394 |  |  |  | 385 | 47 | 156 | 254 | 353 | 522 | 532 | 307 | 394 |
|  |  |  | 400 |  |  |  |  | 48 | 198 | 256 | 356 | 523 |  | 350 | 400 |
|  |  |  |  |  |  |  |  | 50 | 205 | 258 | 362 | 524 |  | 392 |  |
|  |  |  |  |  |  |  |  | 54 | 210 | 259 | 363 | 525 |  | 403 |  |
|  |  |  |  |  |  |  |  | 58 | 211 | 284 | 380 | 526 |  |  |  |
|  |  |  |  |  |  |  |  | 59 | 218 | 306 | 382 | 527 |  |  |  |

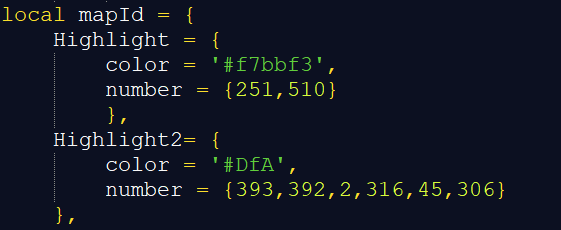
Chain for Map 251

Chain for Map 510

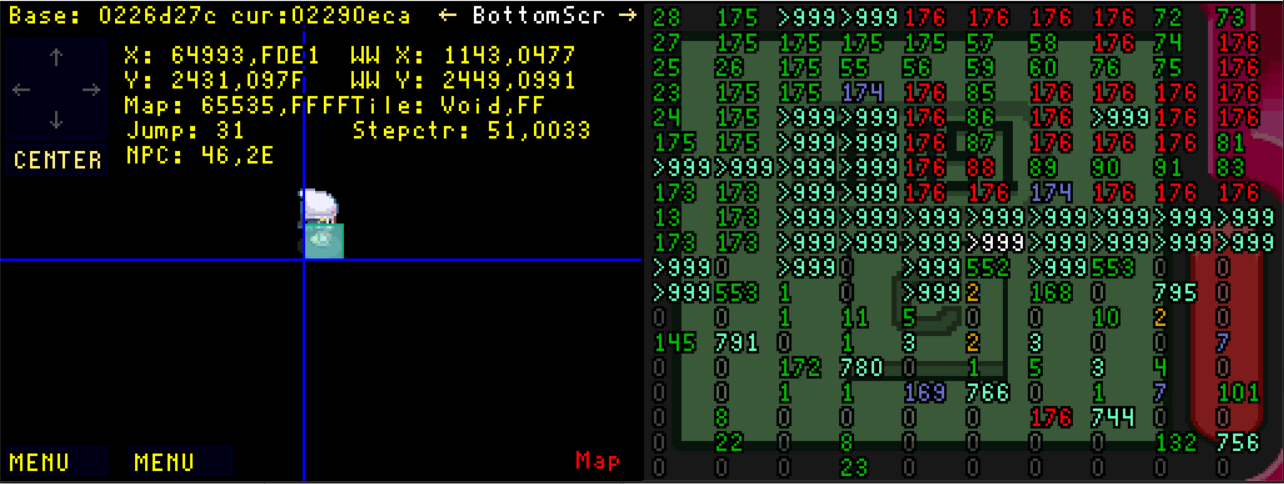
There is a common denominator between the two chains in this case, being Map ID 392. It is possible to load both 251 and 510 in one chain with this. 392 can be easily written by Map ID 2, and lower values are always easy to access. I’d like to note that this route has been optimised to death, it’s just important to understand the process behind it to create your own routes.

To summarise:  
Map ID 251 is written by 393, which is written by 392, which is written by 2.  
Map ID 510 is written by 316, which is written by 45, which is written by 306, which is written by 392, which is written by 2. Insert these values into *RSFD.lua.*

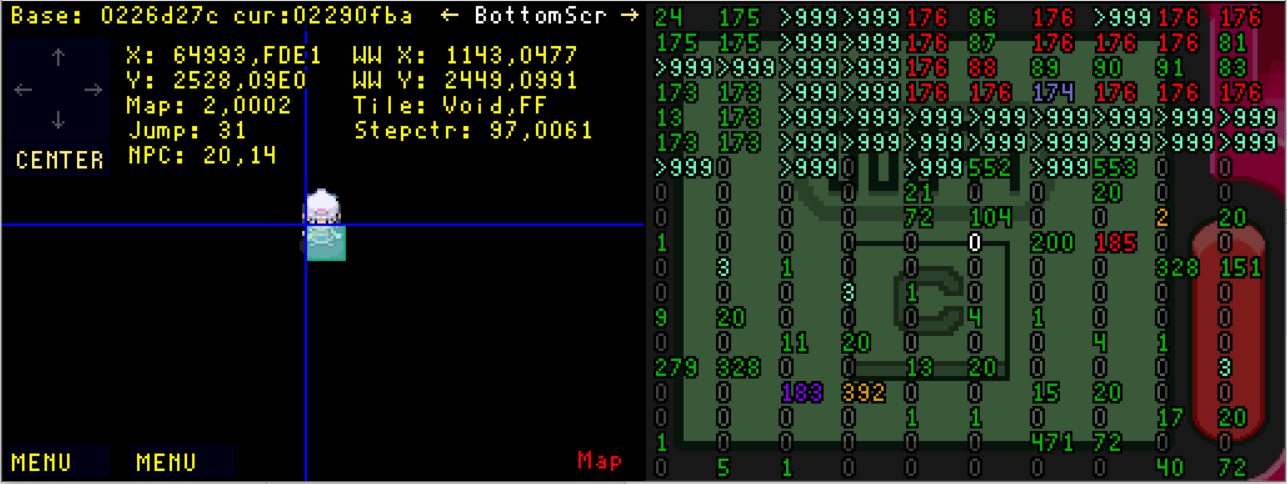
This is how the Map IDs should be implemented in the script.   
  
You can also edit the colors. By default, Highlight is pink. Highlight 2 is orange.



Now, we’ll go back to our savestate that we called Mapdata, or [follow the route mentioned to get there](#MapdataRoute). Our first goal is to get to Map ID 2, of which there are 2 located slightly to the right of us. The first one is so high up that it will be overwritten by the 552 above it, but the onder under that will remain. If we press shift + K we see the sides of a cell, showing us when we’ll enter a new one. We move 51 E and see our current cell is alligned with the Map 2 under us.



We go down and enter the second Map ID 2, since the first one gets overwritten while going down. At 97 S Map ID we enter Map ID 2, it overwrites itself and dissapears. This is known as a selfoverwriting Map ID, and will be mentioned later. We also see that 392 gets loaded slightly left under us.



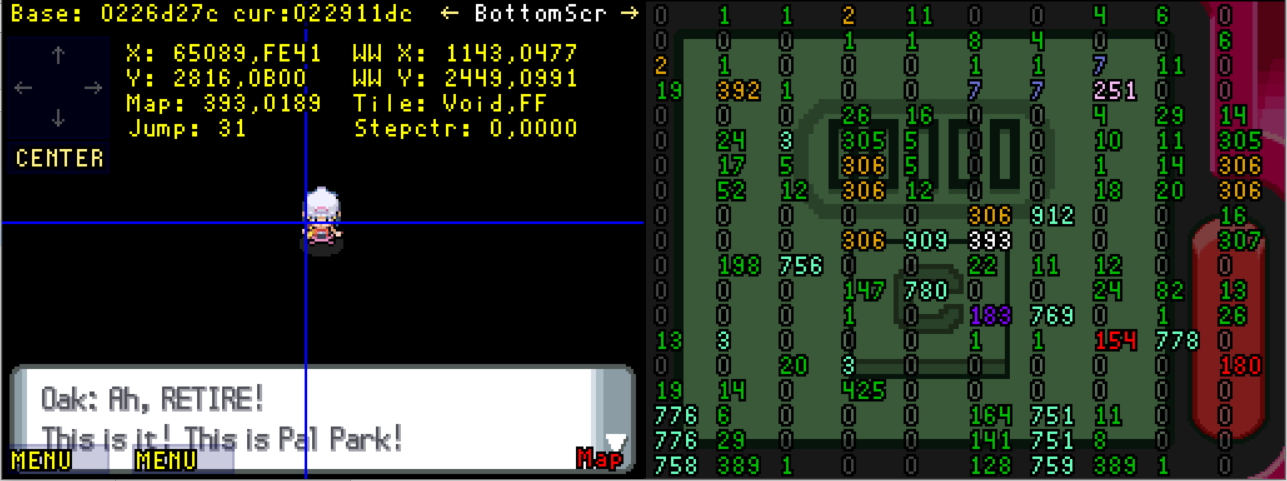
If we want to enter Map ID 392, we have to do that via the left side or bottom side, due to [how chunks are structured](#Basic_routing) in this part of memory. We will enter it from the right side, under the loadline. We head 209 S. Instead of writing 97 S then 209 S we combine these steps, and it becomes 306 S.

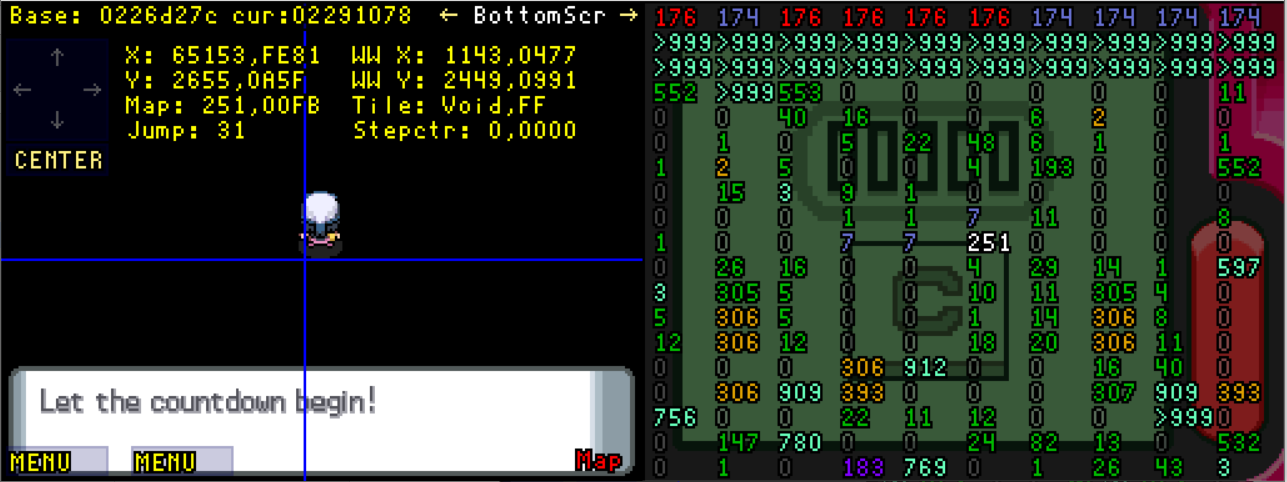


Now we simply go 1 cell + 1 tile left, 33 W. This loads Map IDs 306 and 393 right of us, which we will both need for our route. The Map ID 393 can be seen by entering Manual mode, it is one cell to the right of the centered screen.



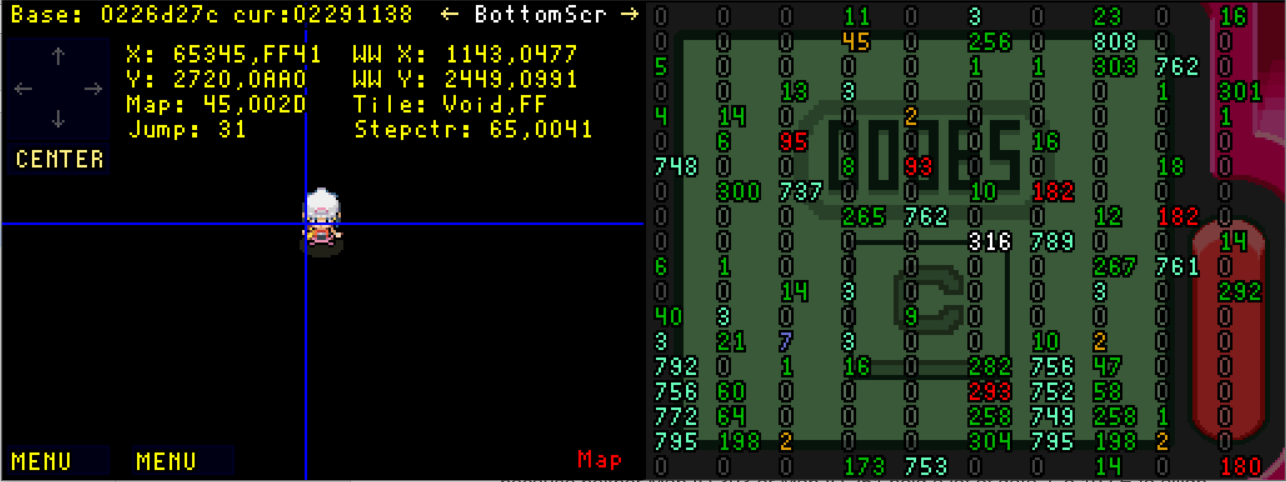
From here we want to go down and then right to enter these Map IDs. We can’t go straight down here, since we are at the right section our cell. If we go 1 E we will be at the left side of the next cell, which will let us go down. After that we go 78 S, until the bottom of the cell at the same height of one of the 306’s that is loaded. We will enter the 306 directly right of us, because the one below us would waste precious steps. We go 128 E to enter the Map ID. This wrote a lot of data, but luckily not enough to overwrite our Map ID 393. By entering Manual mode we can see Map ID 45 is written a bit more to the right of us. We enter the Map ID 393 under us by going 1 S. If you have not entered Pal Park’s hub area before, it plays a cutscene.



This cutscene does move the player one tile E, so in our route we need to mention that. We want to go to the catching area of Pal Park, which the pink Map ID 251. We go 17 S, and then need to go 64 E or 63 E if you had the cutscene play. After that, go 177 N. We are now located 1 tile below Pal Park. We go 1 N, which we note independently because it plays a cutscene when we enter the Map ID, which means the step counter won’t register it.   


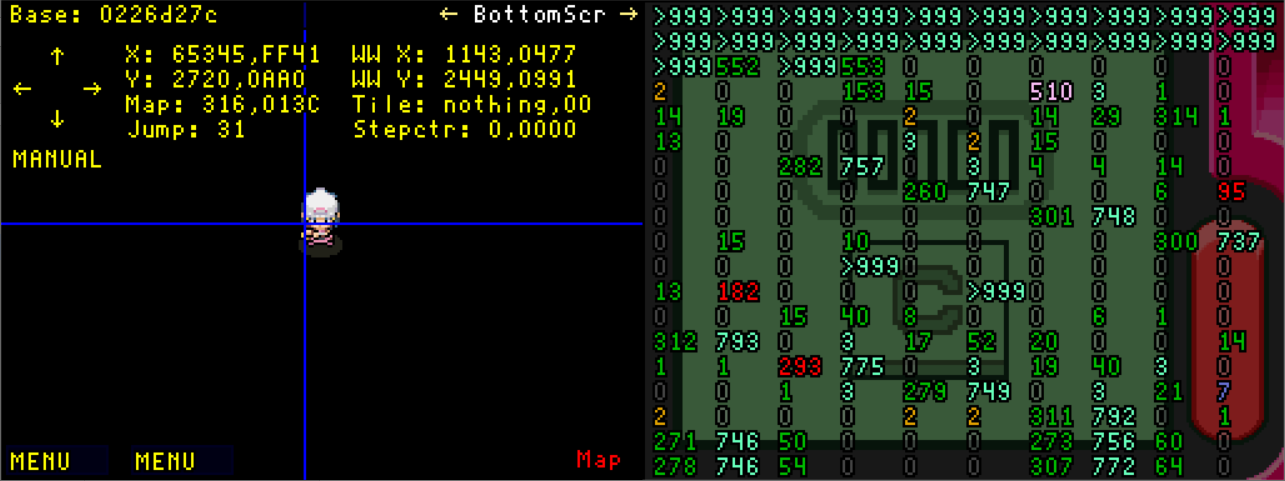
After the cutscene, if you open the menu you will notice the Pal Park menu is enabled. We have completed the first chain, and our second chain still works.

If you enter Manual mode and check right, the Map ID 45 is still in memory. That is because neither Map ID 393 or Map ID 251 hold a lot of data. Go 192 E to allign yourself with the Map ID 45. We go 65 S we enter the Map ID 45.



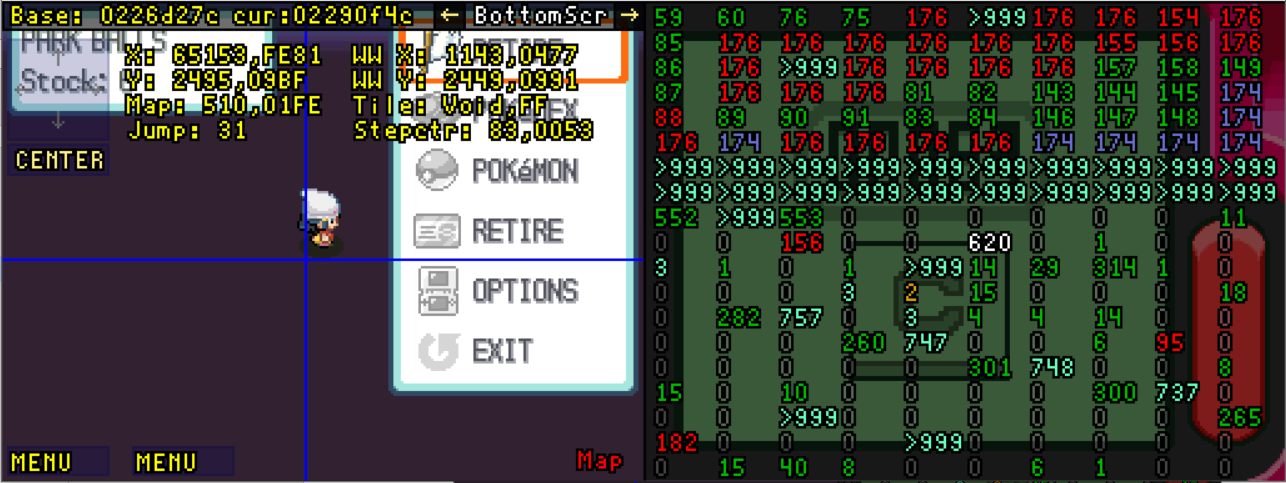
As you might notice, this is a selfoverwriting Map ID. In fact, it wrote the Map ID 316 directly at the same cell we’re standing in. Before we enter Map ID 316, I’d like to familiarize you with the concept of ‘freeing Mapdata’. Since Map ID 510 will be written high up, a lot of Map IDs can overwrite it. If we write it now, we will need to go around a huge bulk of data, only entering Mystery Zones to reach it. But, there is a way to get a ‘shortcut’. If we enter the Map ID 12 which is located above us, it will overwrite the Mapdata in such a way that it’ll be easier to enter the Map 510 later. I will show a comparison after we do that. Go 1 N, 64 E, 64 W and 1 S again.

On the left we see Mapdata when freed, on the right without being freed.



On the first display, there is now a path we can take through Mystery Zones at the left side. On the right display we see a Map ID 754 which I colored purple, that would definitely overwrite the Map ID 510. This is only an optimisation and not necessary, as it’s possible to go even more left and around the data as well.

From here, we go 17 S so we’re under the loadline, and then go through the Mystery Zones until we enter the Map ID 510. We go 179 W, followed by 64 N, 64 W, 32 N , 96 W. this will land us in the gap we just created by freeing Mapdata. Then we continue: 96 N, 64 E, 50 N and to end off, 83 E. This will make us enter Map ID 510.



As you may notice, this is a selfoverwriting Map ID. That was to be expected, since it was loaded so high in RAM. If we press the RETIRE function on our menu now, we can call the script for the battle against Arceus. Since the script moves you up, we will leave our Map ID 510, go through a Jubilife City, that also overwrites itself with a Mystery Zone. If we capture this Arceus, it will therefore have its location be the Mystery Zone.



After the battle, we still need to remove the menu. Luckily, if we use the RETIRE option in Pal Park (Map ID 251), it will warp us to the Pal Park hub and remove the menu. Since at this point you should understand the process, I won’t explain every detail of it. It is essentially just re-doing the chain we did earlier but pressing RETIRE in Pal Park once there.

The final route goes as follows:

64 E  
64 W  
18 S  
179 W

64 N  
64 W  
32 N  
96 W

96 N  
64 E  
50 N  
83 E

RETIRE

160 W  
244 S  
33 W  
1 E  
78 S

128 E  
1 S cutscene  
11 S  
64 E

177 N  
1 N Pal Park

RETIRE

1 S  
17 W  
14 N  
446 W

save reset

2111 S  
147 W  
320 S

51 E  
97 S  
306 S  
33 W  
1 E  
78 S

128 E  
1 S (cutscene)  
17 S

if cutscene:  
63 E

If no cutscene  
64 E

177 N  
1 N Pal Park

192 E  
65 S  
1 N

In this route you learned the concept of chaining, which requires quite a bit of search work. By entering select Map IDs, we wrote data to write another Map ID, which eventually lead to us reaching Pal Park and capturing Arceus. In section [10. Script Execution](#_Script_Execution) I will explain more about the inner workings of the RETIRE option and how to exploit it even further.

Try routing towards different Map IDs to get better at this. The Map ID of an area can be found in *Mapdata.docx*. As example, if you search for Pokétch Co. it will show Map IDs 8, 9 and 10.

If you feel like you’re proficient in routing with Mapdata, you can go to the next step. In the next step we will learn a second very commonly used RAM manipulation to get to Map IDs. It is known as Matrix ID chaining, and can be incredibly fast, taking less than a hundred steps in some cases.

The issue with Matrix ID chaining is that unlike Map ID chaining, the amount of accessible values is low. It uses the ID of your Matrix, which is not the same as the Map ID. The reason for this is duplicates. For example, all Poké Marts share Matrix ID 122.

* + 1. Matrix ID routing

Next to Mapdata, Matrix IDs are the most commonly used way of writing Map IDs. Save resetting in a Map ID changes your Matrix to match the building you saved in. But since a lot of buildings have the same interior, they use the same Matrix IDs. All Poké Marts, Poké Centra, small houses.

The reason this data section is useful is because of how close it is to where we enter the void. It is only 2 cells away. If you open *Matrix.docx* you find that for Pokétch Co. (Map ID 8) the Matrix ID is 141. That is the Map ID of Veilstone Store’s 5th floor.

|  |  |  |  |
| --- | --- | --- | --- |
| Ch1 | 187 | 176 | 141 |
| 187 | 176 | 141 | 8 |

One important thing to note is that you must have save and reset the game at least once before using this data. If you haven’t, there will be flag set in front of the value, causing it to become a Jubilife City. Earlier, we made a route to the Hall Of Fame (Map ID 187). It used Mapdata, which required quite some steps. But it is actually possible to optimise the route if we use Matrix ID chains instead. If we search *Matrix.docx* we find this chain to the Hall Of Fame.

The only chain that is possible in this case leads us right to Map ID 8, Pokétch Co. The way you should interpret this data is as follows, if I have the Matrix loaded of Map ID 8, the Matrix ID will be 141. If I then save and reset in Map ID 141, the Matrix ID will become 176, and if I save reset in that Map ID the Matrix ID will be 187. Since Pokétch Co. is an easy entrance point to the void, Matrix ID 141 will be loaded just by entering the void there. With this knowledge, let’s create a route to the hall of fame. Follow these steps from the Pokétch Co. void entrance:

Remember we need to have saved once before performing this route, to make sure that the Matrix ID will appear as expected.

When we are at 63 W, we’ll be at the Matrix ID, currently Map ID 141. By save resetting it changes to Matrix ID 176. Map ID 176 is actually a BSOD Map, but this is one of the few instances in which it is safe to save in it. The exact reason for why it doesn’t crash here is unknown at the time of writing this document. After resetting and going 1 W, we load the Hall Of Fame as Map ID. This is the shortest known route.

1 S  
17 W  
14 N  
63 W

save reset

1 W

save reset

1 W

Just like when working with Mapdata, multiple chains are possible. You can also first route to a section of the chain in Mapdata, save reset and then continue with Matrix ID routing. This can sometimes save a little time.

The routes we have performed so far only go to a value in RAM that we read as a Map ID. The next step is routing towards the physical Matrix of this Map ID. This way we can interact with NPCs, perform sequence breaks, get to areas earlier than intended and way more.

In the next step we’ll learn how to enter the Matrices of indoor areas. Not every indoor building can be entered without additional exploits that we’ll cover later.

* + 1. Chunk Dislocation

Chunk dislocation is a glitch that can be triggered under numerous conditions. When you walk around, the game calculates where to load new chunks when moving over loadlines. If you open a menu or enter a warp, the game also does this. But the methods for loading these areas are not the same between these two instances.

There is an error in the code to load areas after closing a menu that can cause you to order the chunks in a way that isn’t intended. You could think of it like tweaking by simply opening a menu. I spent a lot of time implementing a Chunk Viewer in the *RSFD.lua* that calculates what chunks are currently loaded and in which of them you currently are. To shift between the Memory viewer for the void to this Chunk Viewer (and other menu’s), press CTRL + V. I will say that when out of bounds, it may make mistakes with showing which chunk you’re currently in. In that case you can press V and up and down arrows until it matches the chunk you’re in. As an example, we’ll get inside the Great Marsh. The Great Marsh has trees all around it, which makes it seem impossible to enter, but with Chunk Dislocation we can move the Great Marsh chunks we can usually walk in to the trees, letting us go through them.

Follow these steps from the Pokétch Co. void entrance:

This route will get you into Map ID 509, one of the 6 Map IDs used in the Great Marsh. It is listed as a BSOD danger Map ID since it loads the train as a Gimmick Model, but luckily this one is entirely harmless.

Once we save reset in it, we load the Matrix of Great Marsh. The Matrix is always loaded at (0,0) to ((Mapwidth\*32)-1,(MapHeight\*32)-1).   
In this case that is from (0,0) to (127, 159)

From here, I’ll explain how we’ll abuse chunk dislocation to enter the Matrix.

1 S  
16 E

Bike  
462 N  
save reset

128 E  
64 N  
  
Get off bike and Register Town map

save reset

Now we want to move the chunks that are usually positioned under the trees to be moved inside the trees. This can be done by the simple act of refreshing graphics at a negative Y coordinate right above the Matrix. We can also move chunks one to the left when we refresh graphics at a negative X coordinate. And you can combine both by refreshing with a negative X and Y coordinate left above the Matrix, in which case chunks get moves left and up. For our purposes, only a negative Y coordinate is required.

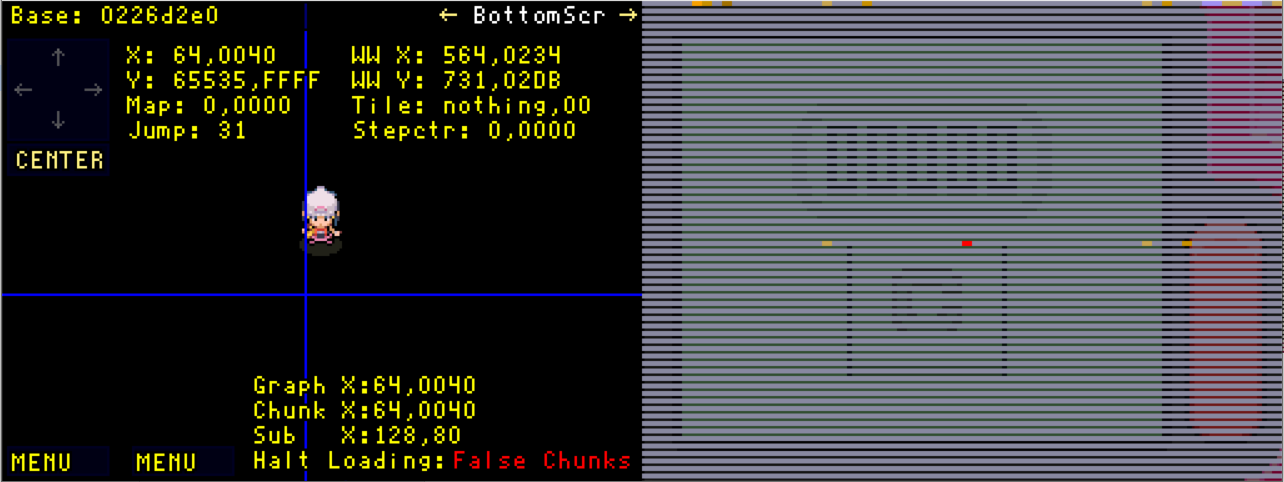
So the first thing we want to do is get right above the Matrix, at Y = -1. The script shows this as Y = 65535. (In reality, live coordinates are 32 bit values, but I truncate them for voiding purposes) We’ll be dislocating the chunks in the middle of the Matrix, which means we want to be around X = 64.

Once we’re here your coordinates should be (64,65535). We are currently in a Mystery Zone, which prevents you opening the menu.

96 W

511

We registered a Town Map before saving. Being in a Mystery Zone prevents opening menu’s, but not using Key Items. And this is what we will now abuse to enter the Great Marsh. Shift to the Chunkviewer if you haven’t already with CTRL + V, currently it will return an entirely black area with 4 lines with some tiles in them.



If we use our Key Item, the Town Map, it will change immediately. It will now load the chunks of Great Marsh, but due to being at negative coordinates we’ll be dislocating the chunks. It’s important to note that what you see is not the same as the actual tile data that is loaded. Graphical models are separate from the tiles themselves. The only affect on movement models have is blocking you if the height between areas is too large.

The screen should look like this: 

If we move in any direction except up the game will put us inside the chunks that are currently displayed at the bottom. This concept is known as chunk dislocation. If we take the following steps we can go through these chunks until we load new ones.

We go through the chunk and avoid the lighter green colored tiles, which are Mud tiles with grass on them. We also avoid the brown tiles, which are Mud tiles. These tiles would put you stuck in them until you wiggle, and this could make it difficult to route. When we went over the loadline at 8 S, it loaded new chunks.

2 S

11 W  
7 S  
2 E  
10 S

Since calculating chunks by walking is separate, it will actually load the correct chunks under us. Causing us to duplicate the chunks.



We simple will keep going until we are at the chunk under us, at which point we can reload graphics to correctly load all the chunks around us. Simply go 1 S over the ledge that is marked orange, then go 12 S. We enter the Great Marsh Map ID, and NPCs get loaded around us.



Since we loaded everything in the Mystery Zone tileset and we dislocated chunks everything looks off. Simply open a menu or use Town Map to fix this.



We have now entered the Great Marsh without paying. As a side effect, we can encounter Pokémon here in the normal battle style, and we don’t have to worry about running out of steps. This surely makes it perfect for shinyhunters who enjoy using glitches!

Now, this is a prime example of how to use Chunk Dislocation. In areas that have multiple chunks, there usually is always a way to get inside using these methods. But when it comes to Matrices that are only one chunk in size, it isn’t always possible to get inside. I’ll go over some examples of Matrices with this limitation that you can still enter using this method.

The thing that all the Map IDs I’ll show have in common is that they have some opening inside a wall. This could be a door or some sort of oversight. In our first example we’ll enter Veilstone Store. It is the Matrix ID of Pokétch Co., so we can make this into a very short route

Follow these steps from the Pokétch Co. void entrance:

Here we the Matrix ID, but above us rather than left of us. In this case it saves a small amount of steps due to the fact that the elevator is positioned to the right of the Veilstone Store.

We refresh graphics, which is done by opening any menu that covers the entire screen. Pokédex, Pokémon, Bag, Options, Key items such as Town Map or Journal, battles… It also occurs when you save reset. The reason we went 62 south until 65534 is due to there being tiles being read from RAM at 65535, which will be explained later in tilewriting.

If you check the chunkviewer, you see we are inside the chunks of the building when refreshing graphics, but when going down move to a different chunk. We then refresh graphics to be back in the correct chunk. The tile under us is the elevator, which has no collision. We can now enter it.

1 S  
16 E  
78 N  
1 W  
save reset  
62 S  
16 W  
1 S  
Refresh graphics

Hold down  
Refresh Graphics

Hold down

1 N

In this case we use the fact that the elevator inside the wall has no collision to enter   
 the Matrix. Because we had 2 tiles we could in any direction before we get blocked,   
 we were able to barely reach the elevator.

A second example of using chunk dislocation to enter Matrices with this method is   
 this short route to Giratina.

1 S   
16 E   
Bike   
430 N   
21 W   
save reset   
300 S   
save reset  
115 S   
Refresh Graphics Hold down   
Refresh Graphics  
Find room

This route doesn’t use any manipulations, it simply the first part of pointers. This part is always the same, unless you refresh graphics in a different Map ID along the way.

We enter one of the rooms via the top gate. After that you still have to find the correct room for Giratina. You can save to prevent having to restart the puzzle because of RNG.

Now that we’ve seen how to use Chunk Dislocation to enter matrices, I will show how to use it to encounter wild Pokémon while still inside the void. We will move chunks with grass/cave tiles/water… into the void, and walk around in them to encounter Pokémon.

Before we create these sorts of route, I should mention how encounters are generated. We can not just get any pokémon, we only can get those that are programmed to be inside certain areas. The game loads encountertables by entering a Map ID, so as long as we can get encounter tiles in the correct Map ID, we can encounter the Pokémon in its encounter table. I told you about the [layout of chunks](#Basic_routing) in a specific section of RAM we used for voiding. For example when we are (Map Width \* 32) tiles left of the Matrix. Here we can read chunks from the top right section of the cell. But, since the game calculates moving over loadlines differently from refreshing graphics, sometimes we can enter a chunk only when graphic reloading, and other times we can only enter it when walking inside. In our first example, we can simply enter the chunk with grass, but Chunk Dislocation will overwrite our chunk after battle. You’ll learn how to deal with this right away, as it’s quite simple.

Follow these steps from the Pokétch Co. void entrance:

You might notice I use different steps to get to 415 W. This method lets you get bike first, then uses Chunk Dislocation to let us move left, by moving Pokétch Co. left by (Map Width \* 32), or 32 steps, right next to itself. Since there is empty space under Pokétch Co., we can move under it.

After that we go east until we’re right above Map ID 403. We go down until we’re inside it, particularly next to where the chunk will have grass. We save next to the loadline, because it’s easy to get stuck if you mess up. Once we go 1 E, grass suddenly appears. Move to the chunkviewer, it will display our position incorrectly, sadly, but you can press C + down twice to fix this.

Once inside the grass, you can go up and down to encounter a Pokémon. But what’s important is what occurs after the encounter, as Chunk Dislocation will occur.

And this Chunk Dislocation puts us inside a wall, but luckily, the chunk next to us is considered void, so we can go 1 W again, then move 1 E. Since moving is calculated differently, it will load grass again. In this case Chunk dislocation is more trouble, but in other cases it allows for extra encounter spots.

If you would accidently go east after a battle, you’d be softlocked in this case. This is why we saved.

1 S  
17 W  
18 S

bike

1 N  
refresh graphics

415 W

save reset

161 E  
264 S  
13 E

save

1 E   
move up + down

after battle:  
1 W  
1 E  
move +and down  
repeat from battle

fly/teleport away

In the next example we will use the Chunk Dislocation to get both grass tiles and water tiles in the same spot. The chunk when loading by walking will contain grass, and after the refreshing of graphics by the battle there will be water.

Follow these steps from the Pokétch Co. void entrance:

This route is a very specific instance where we can both enter grass and water at the same spot depending on how we load the area. I’d like to note it’s possible to enter any tile next to a loadline with walking, including water. So it’s not necessary to use Chunk Dislocation, and there may even be spots where both the loading algorithms put you inside a chunk with water at the same spot.

So far we’ve been performing this in Black Sinnohs. Black Sinnoh is the name of 960x960 areas directly left and right of the original Sinnoh, because of their ability to let us enter chunks in them. These are actually not the only places you can get chunks to load. With ledgecancelling and Fake Sinnohs, which we’ll go over soon, it is also possible to get these effects. With ledgecancelling we’ll even acquire full control over the positioning of chunks, which will be a very powerful tool.

In the next and last route about Chunk Dislocation we’ll be using Chunk Dislocation to get encounters in an indoor Map ID, a cave to be exact. We’ll get the encounters of Wayward Cave’s secret entrance while not directly inside the Map ID.

1 S  
17 W  
18 S  
Bike  
refresh graphics

415 W

save reset

321 E  
518 S

13 E  
save  
1 E (grass)  
move up + down

after battle:

1 S (water)  
Wiggle around

after battle:

1 W   
1 E (grass)  
move up + down  
  
fly/teleport away

If you’re unable to get grass to appear at the border of a cell (so you can leave by   
 moving to the next row) you can always try to use the Black Sinnoh located at the  
 right side of Sinnoh’s Matrix. If that doesn’t work either, you can still just route to the   
 real area (with techniques we’ll see soon). There is also the possibility to use  
 tilewriting or chunk storage, which are a bit more advanced to pull off.

In the route to Wayward Cave’s secret area we’ll be accessubg a chunk that is left of   
 the matrix (this technique applies to all matrices), but we’ll have to deal with chunk  
 dislocation after battles again.

Follow these steps from the Pokétch Co. void entrance:

Once again I use static memory I know will always return Map ID 285, Wayward Cave. When making routes that use Mapdata, you’ll have to move all the way back to 0,0 yourself.

In this instance, walking gives us a chunk left of the real Wayward Cave. But when the battle ends, the game moves an unwalkable void chunk there, which would softlocks you.

If you remember from before, at y = 65535 tiles can appear. In fact, this happens at the top of each cell when at the left or right side of a Matrix. This occurs left of Matrices. Unlike chunks being loaded in Black Sinnoh, this effect extends along the y axis until it is positive 32,767. This is very useful, as we can actually manipulate these tiles, which we’ll learn later.

For now we’ll use a part of them that will always have 0 as collision, meaning they’re walkable space. If we encounter Pokémon at the bottom of the cell at y = 31, we can then go 1 S to the cell under us. The top of that cell will be tiles we can walk through, allowing us to go back over the loadline and load chunks by walking. This will return the cave chunk, and the cycle can be done indefinitely.

1 S  
16 E

Bike

462 N

save reset

192 W  
64 N

save reset

143 E  
543 S

left until wall  
move left/right  
  
after encounter:  
1 S  
8 E  
1 N  
left until wall  
move left/right  
repeat

Escape rope

These are the major ways of using and avoiding chunk dislocation to encounter Pokémon. Later we’ll also see how we can trigger chunk dislocation anywhere, using a technique known as ledgecancelling. We’ll be going over Fake Sinnoh’s first, and then immediately after learn Wrong Warping. With Wrong Warping we can significantly shorten routes. It also gives us access to previously impossible glitches.

I will be covering 3 major Wrong Warps. More exist, but are practically unusable for normal routes, and take more time to set up.

* + 1. Fake Sinnoh

Fake Sinnohs are more commonly known, due to them being used a lot in the older routes before the void was understood. Fake Sinnohs are a culmination of two mechanics. One of those is the visual aspect, the other the saving aspect. We’ll cover the visual aspect first. Fake Sinnohs appear if you go 65536 steps in any direction from the Matrix. In these spots the visual aspect of the Matrix will be loaded, the Mapmodels. They only appear when walking, not when reloading graphics, similarly to chunk dislocation. These models do not have collision, except for height. When the game detects a great height different between two areas it prevents you from walking. Luckily, by graphic reloading we can unload any areas that would have height differences. You can perfectly walk through buildings and walls in other cases. If you want to go to any Fake Sinnoh fast press J and let the jumpvalue go under 0, it will roll over to 65536. If you teleport in any direction you’ll end up at the same spot, but in a Fake Sinnoh.

The reason this occurs is because of the extreme precision of the camera in this game. The camera uses a 32-bit x and y coordinate. However, the camera can determine a location up to 1/65536th of a tile. This is way more precise than you’d ever see ingame. Since 32-bit is simply 65536², we only need to take 65536 steps in any direction to loop the camera.

This alone is however not incredibly useful, if it weren’t for how the game stores coordinates when saving. When walking around the game uses a 32-bit value for both x and y coordinates. When saving this is truncated to a 16-bit value. This is essentially applying a modulo 65536 to our live coordinate, meaning we end up in the original Matrix after saving.

For example, if your x coordinate would be x = 0x0002 0111 it would become  
x = 0x0000 0111 after saving, putting us at the Matrix. As long as our Map ID is part of Sinnoh’s outdoor Matrix, this lets us easily get inside. If we simply walk to a Fake Sinnoh to do this it would take 65536 steps, and if the Map ID doesn’t match with the area we want to save, we would need to take another 65536. This takes hours, and there would be tons of unpredictable data you are going through along the way. This is how old void routes worked, but obviously these practises should never be used anymore.

Instead, we will immediately learn of several methods to reach a Fake Sinnoh nearly instantanious, using Wrong Warps.

# Wrong Warps

Wrong warps are a common glitch in many games, including the Zelda, Mario, and many other franchises. The general premise of these glitches is causing the game to warp you to somewhere you’re not intended to. And in Diamond and Pearl there are plenty to talk about.

* 1. Main Wrong Warp

The main Wrong Warp is possible on all versions of Diamond and Pearl. The Wrong Warp is triggered by abusing a check with the NPC’s that let you enter the Union Room. The Union Room is used to let the player trade with other people, and there is one in each Pokémon Center. In order to let the player appear at the correct Pokémon center after leaving the Union Room the game checks your Map ID and coordinates when talking to the lady behind the desk. If the check succeeds it overwrites some warprelated variables with your Map ID and X and Y coordinates. If the player leaves the room or resets the game it will be teleported to these positions instead.

The same addresses used for this warp are also used by countless others. Elevators, the safari gate, the Battle Tower, the Hall Of Fame and even the Explorer Kit all initialize them for their own warps.

If the player would manage to bypass the check and stop it from overwriting the variables you could therefore manipulate where you warp to. The Explorer Kit specifically is extremely flexible, as it sets your current Map Id, X and Y coordinates by simply popping up the menu, you don’t even have to save or enter the Underground.

This can be achieved in two ways, one that only works on the Japanese versions and one that works on all versions. The first one is actually a huge oversight by the developers. They forgot to add the Pokémon league’s upper floor to the list of Map ID’s to check. This means that simply by talking to the lady behind the desk in the Japanese Versions and resetting the game will Wrong Warp you. The glitch was acknowledged by Nintendo, and they even provided an official fix that said to talk to the lady in the Contest Hall to fix any side effects (which we’ll cover later).

The second method abuses the Void Glitch and chunk dislocation. If we route ourselves to be behind the desk and talk to the lady from any side the checks will fail as our X/Y coordinates won’t match them. Afterwards resetting the game will Wrong Warp you.

There is a simple consistent route towards Map ID 7, which is the upper floor of the Pokémon center in Jubilife City. This one will be abused for all the Wrong Warps, but the route towards it won’t always be the same. This is because we want to use our Explorer Kit in different spots to warp to different sections in RAM.

* + 1. The 3 Fake Sinnohs

We can use the Main Wrongwarp to warp to all sorts of things as mentioned, but the Explorer Kit in particular is powerful. The reason for that is how coordinates are stored. When walking around your X and Y coordinates are stored as 32 bit signed integer values. This is represented as 0xFFFF FFFF in hexadecimal. However the game stores warp coordinates as 16 bit unsigned integer values. So they’re quite similar to how coordinates are stored when saving, except now they’re unsigned. The difference between signed and unsigned is wether or not a value can be negative or not. For example if you saved at x = 0xFFFF the game will consider this -1, so you’ll appear at 0xFFFF FFFF. When warping to x = 0xFFFF however the game will actually put you at x = 0x0000 FFFF. That’s one step away from a Fake Sinnoh.

So, now this is where Explorer Kit becomes powerful. When you use the Explorer Kit will write your current Map Id, x and y coordinates when the ‘Wireless Communications’ message pops up (It won’t if the item isn’t allowed to be used).  
If you used your Explorer kit at x = 0xFFFF FFFF, or -1, the game stores this as a 16 bit unsigned integer, or 0xFFFF. And as mentioned, this warps to x = 0x0000 FFFF.  
This means you warped a one full Fake Sinnoh along the horizontal axis. But, you can also warp along the vertical axis, or both. Of course it’s also possible to simply warp to neither, which will put you in the real Sinnoh.

Blue is position you use Explorer Kit, grey is the position it will warp you to later.

|  |  |  |
| --- | --- | --- |
|  | X = 0x0000 0000 | X = 0xFFFF FFFF |
| Y = 0x0000 0000 | (0x0000 0000,0x 0000 0000)  Real Sinnoh | (0x0000 FFFF,0x0000 0000)  Fake Sinnoh 1 along x axis |
| Y = 0xFFFF FFFF | (0x0000 0000, 0x0000 FFFF)  Fake Sinnoh 1 along y axis | (0x0000 FFFF, 0xFFFF FFFF)  Fake Sinnoh 1 along both axi |

Now let’s perform one of these and see where we end up. Let’s start with the Fake Sinnoh along the x axis. Note that we don’t need to be *exactly* at 0xFFFF FFFF. Anywhere remotely close to it will still bring us close enough to the Fake Sinnoh.

You can view the ‘Wrong Warp’ section of the RSFD script which shows where you’ll end up.

Follow these steps:

1 S  
17 W  
18 S  
Use Bicycle  
1 N

Graphic reload

415 W

Save + hard reset

384 E

Use Explorer Kit  
Press B on message

1 E  
511 N  
255 E

Save reset

214 W  
479 S

Graphic reload  
Hold down  
Graphic reload  
2 S

Talk to middle lady

We go through the chunk and avoid the lighter green colored tiles, which are Mud tiles with grass on them. We also avoid the brown tiles, which are Mud tiles. These tiles would put you stuck in them until you wiggle, and this could make it difficult to route. When we went over the loadline at 8 S, it loaded new chunks.

* 1. Overworld Wrong Warp
  2. Japanese Exlcusive Wrong Warp
  3. Spear Pillar Wrong Warps
  4. Other Wrong Warps

# Battle Tower Void

# Tilewriting Basics

# Ledgecancelling

# Advanced Tilewriting

# Script Execution

# Troubleshooting

# Glossary

|  |  |
| --- | --- |
| Abyss | A chunk that has been dislocated by performing a tweak or instant tweak. |
| ASLR | ASLR stands for Arbitrary Space Layout Randomization. This is a way to change the layout of RAM by adding a small offset. It is seeded by mac address, inputs, and the time of booting the game. It only changes when booting and soft resetting the game. |
| Battle Tower Void | When you warp or save and reset in any of the Battle Tower Map IDs, you will end up in a Battle Tower Void. When inside these voids, the game uses a different chunkloading system that has less boundchecks. You can also save in Mystery Zones in this void, but the step counter becomes disabled making routing in these voids problematic at times. |
| BSOD | BSOD stands for Black Screen of Death. This occurs when the player has travelled through a list of Map IDs (referred to as BSOD maps) that load maprelated 3D models (called Gimmicks internally) that do not get unloaded when loading a new Map ID. These models can cause a crash when the game attempts to apply textures by loading graphics. If the player saves and resets after this has occurred the game may crash each time you boot the game on a black screen, the origin of the name. |
| Buffer |  |
| Chaining |  |
| Cell | A cell is a 2-byte address which the game reads Map IDs from. The way the game moves you between cells is dependent on your Matrix Width. |
| Chunk |  |
| Chunk Storage |  |
| Ledgecancel | A Ledgecancel is a glitch that occurs when a player can walk/stand on a bike ramp and open the menu within a specific timeframe. It has many applications, such as chunkstoring and tilewriting. |
| Map ID | Every Map ID is its own ingame area. It is a 2byte value, meaning it can range from 0 to 65535. The used indexes range from 0 to 558, every index above that gets converted to being index 3 by an Error Handler. Map 3 is Jubilife City. |
| Matrix | The Matrix is the inbounds section of an area. It can be as small as a 32x32 area for houses up to 960x960 for Sinnoh’s outdoor section. When saving and resetting inside a Map ID, you will load the game with the inbounds area turning into the one of that Map ID. In some cases, such as Sinnoh, multiple Map IDs are part of the same Matrix Center, and therefore any of them can be used to load the same inbounds area. |
| Matrix Width | The Width of a Matrix center is a value used to determine your position in RAM. It is dependent on the size of your Matrix. The bigger the Matrix Center, the bigger the Matrix Width. And as a result, the more bytes you travel when going up or down. |
| Tilewriting |  |

1. This is not true if the camera loops or is desynced, in which case it shows a designated area. [↑](#footnote-ref-1)
2. It does change, but is the matrix or related to the matrix. Warping or save resetting can change it, but it will not change suddenly while walking around. [↑](#footnote-ref-2)