Tutorial: Reverse Engineering and Debugging for Dragon Ball Z: Sparking Meteor / Budokai Tenkaichi 3 (Wii)

# Prerequisites

Basic C programming knowledge is required. While you don't need to be an expert, you should understand the following concepts:

* Variables and data types
* Logical operators
* Conditional statements
* Loops
* Functions
* Pointers

# Required Software

(Feel free to download them as you go along, as some of the steps might not be needed)

1. Ghidra 11.2.1 for code decompilation
2. Dolphin 5.0 for game emulation, extracting game file and improved debug mode
3. Dolphin Memory Engine <https://github.com/aldelaro5/dolphin-memory-engine>

# Background Information

**DOL Files**

A DOL file (.dol) is the main executable format used by Nintendo GameCube and Wii consoles. Named after the GameCube's development codename "Dolphin," these files function similarly to .exe files on Windows computers. DOL files contain the compiled game code and can be up to 7MB in size for Wii games. They serve as the primary program that runs the game, handling essential functions like loading game assets and managing memory. While these files can be modified for game alterations, they should always be backed up before any modification attempts as they are crucial to the game's operation.

**Assembly Language**

Assembly language is a low-level programming language that corresponds directly to machine code instructions. It typically results from compiling high-level code into platform-specific format.

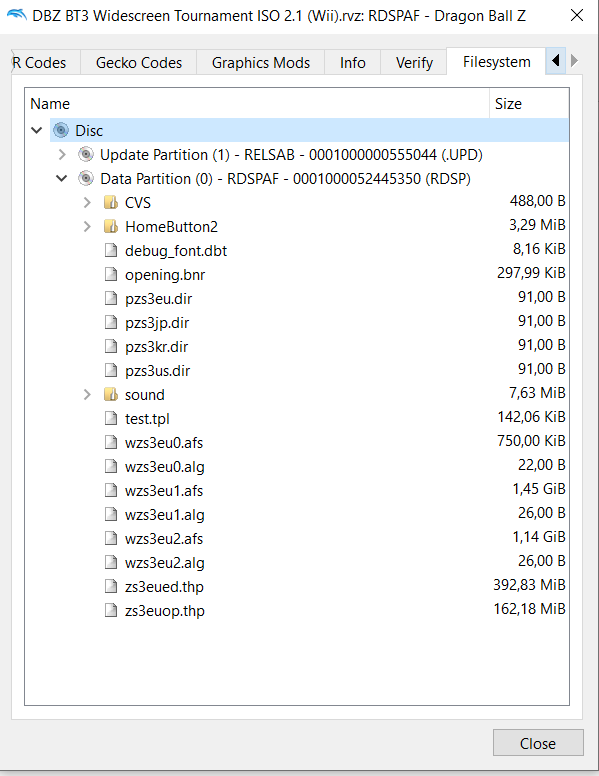
**BT3 Wii Assembly Language**

BT3 Wii uses PowerPC Gekko assembly language, which was standard for GameCube systems with IBM Gekko processors. The Wii's IBM Broadway processor maintains backward compatibility with this instruction set.

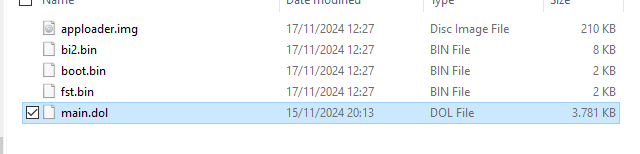
# Getting Started

## Extract the DOL File from the ISO

1. Select the game in the menu within the Dolphin browser, right-click the game and select “Properties”.
2. In the new window, go to the “Filesystem” tab (on the far right).

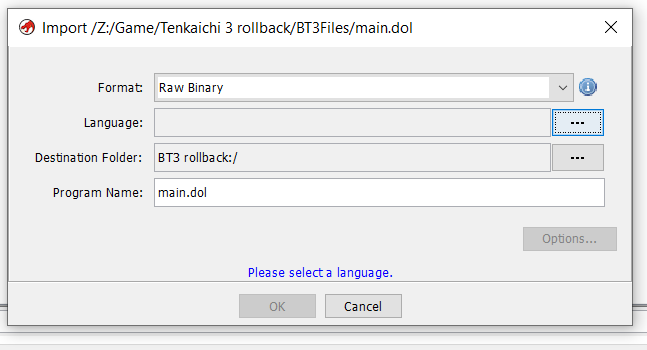


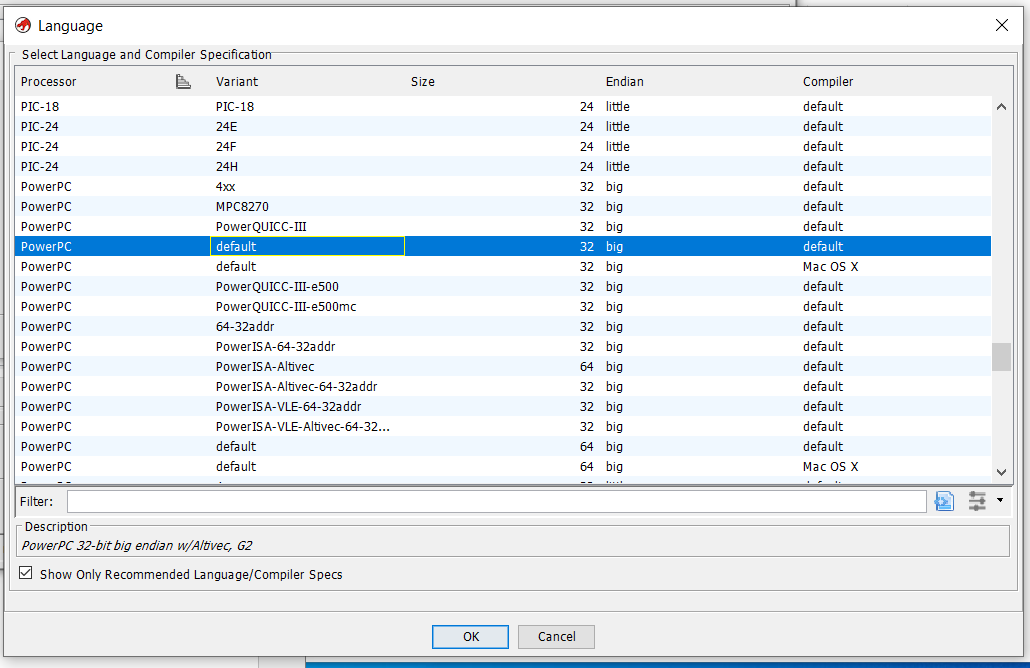
1. Right-click the Disc on top and select “Extract entire Disc…” and extract the content in a folder, this will be the game file that we will use to do stuff with. (Don’t extract it in the Git folder, this is your testing environment)
2. In the extracted folder under DATA/sys/ you will find “main.dol”, this is the game’s code that we’ll be working on. Make sure to always keep a backup of this when working on it.



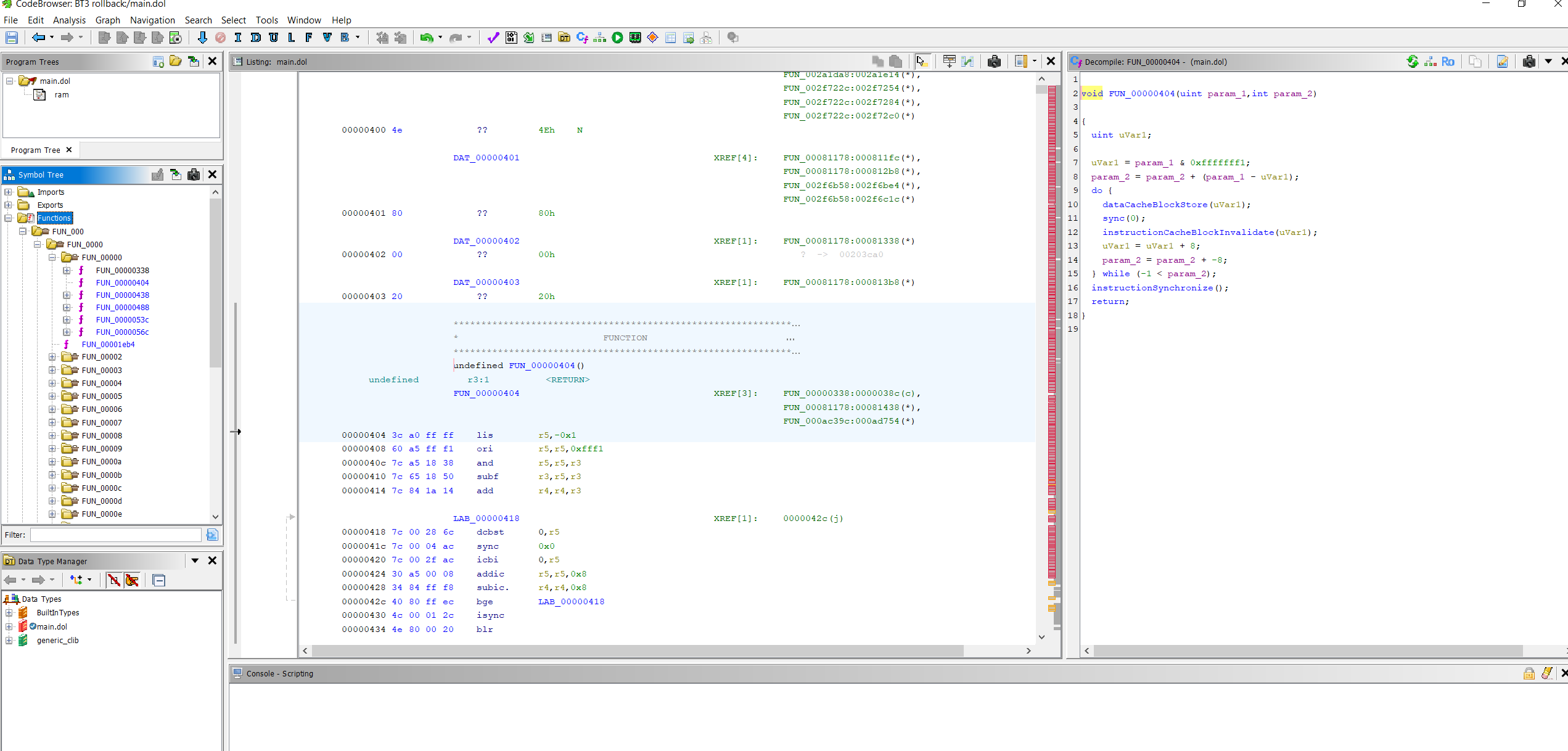
## Decompiling the DOL File

1. Copy the “main.dol” file (it’s already done, skip this unless you want to create another ghidra project)
2. Set up Ghidra: (see readme)
   * Create a new project (or open project and skip the rest unless you want to try it yourself, the Ghidra folder “BT3 rollback.gpr”)
   * Import the DOL file
   * Select PowerPC, 32, big Endian and default compiler for the language variant





* + Open main.dol
  + Run the initial analysis when prompted



Congratulations! You are now looking at the game’s code instructions

On the left under “Function” you can see a whole list of functions that Ghidra has auto decompiled to the right side.

Looks incomprehensible, this is where it begins. (or it is partially comprehensible if we made any progress)

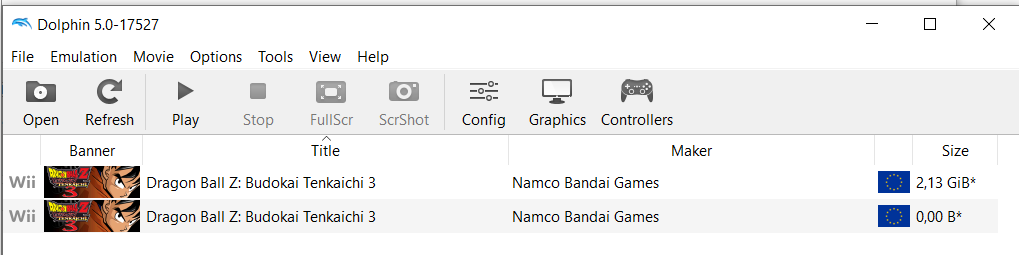
We will slowly find out and label what each of these things are and do and eventually understand enough of it to inject our own code.

# Setting up debugging environment

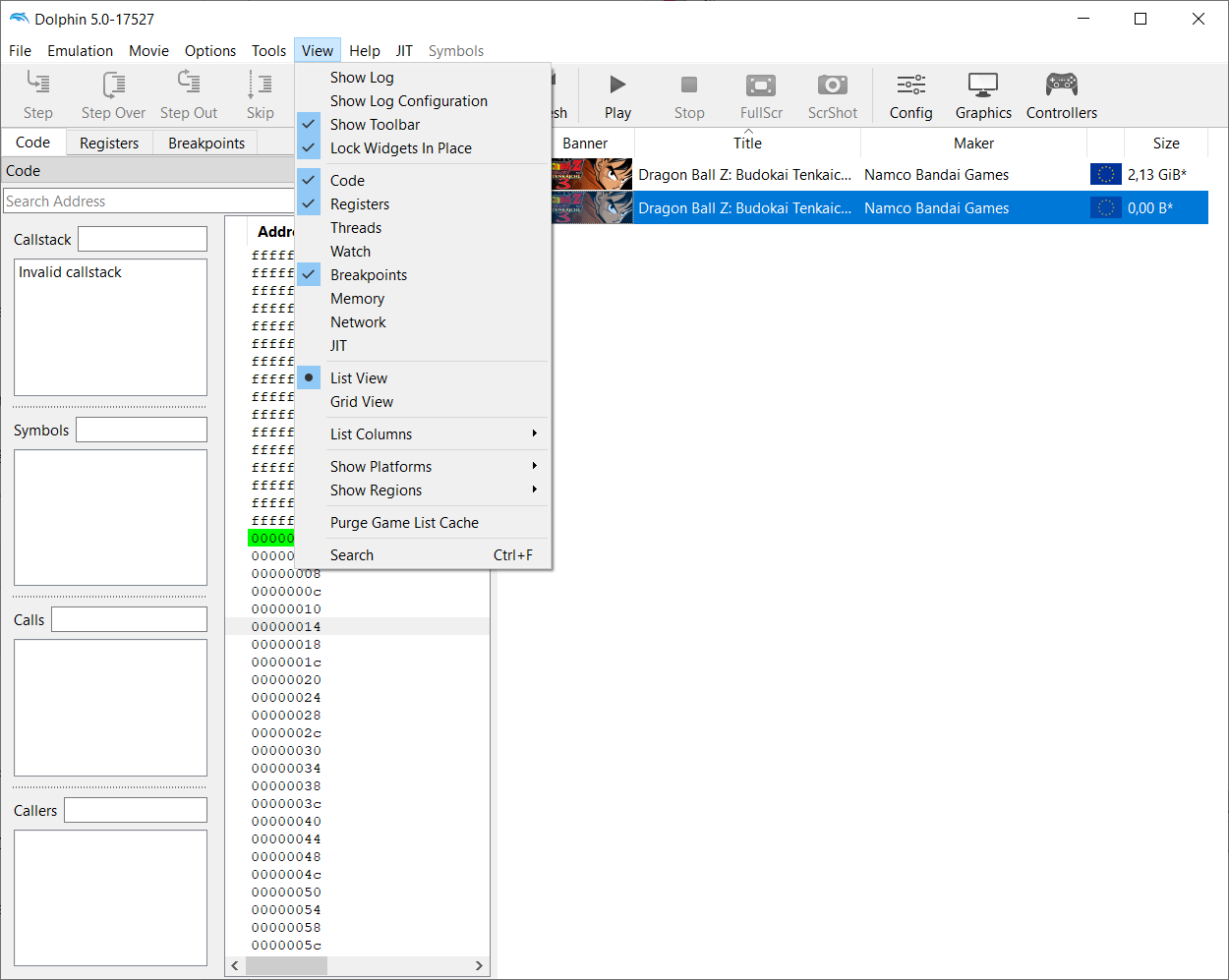
## Dolphin Debug mode

In order to test anything we need to be able to run the extracted game (main.dol) and any subsequent changes that we will make in debug view.

1. On Dolphin’s tool bar, select “Options>Configuration”.
2. Under the “Path” tab, select “Add” and add the folder where we extracted the game folder previously, make sure “Search Subfolders” is checked.
3. Dolphin should now recognize the extracted game that we will use to debug with, in this case the 0,00 B\* one.

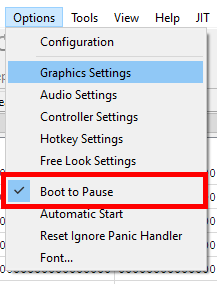


1. Once again on the tool bar, select “Options>Configuration”.
2. Under the “Interface” tab, check “Show Debugging UI”.
3. Then in “View” show all the following.



1. Now we have debug mode!

I recommend activating “Boot to Pause” under “Options” for cases in which you want to look for some code that is execute at the beginning without the game just executing the instruction before you have a chance of catching it.



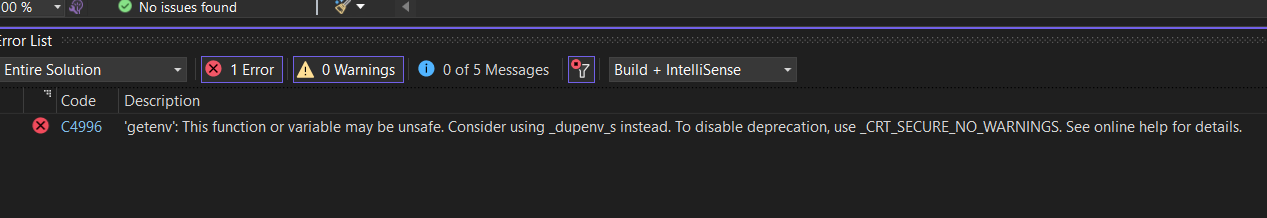
## Dolphin Memory Engine

Next we need to monitor the memory of the emulator, and because Dolphin doesn’t have a memory monitor natively we will get one externally and hook it to Dolphin.

1. Get it here: <https://github.com/aldelaro5/dolphin-memory-engine>

Clone the repo and whatnot, follow “How to Build”

Note: When building it in Windows through Visual Studio you will encounter a bug like so:



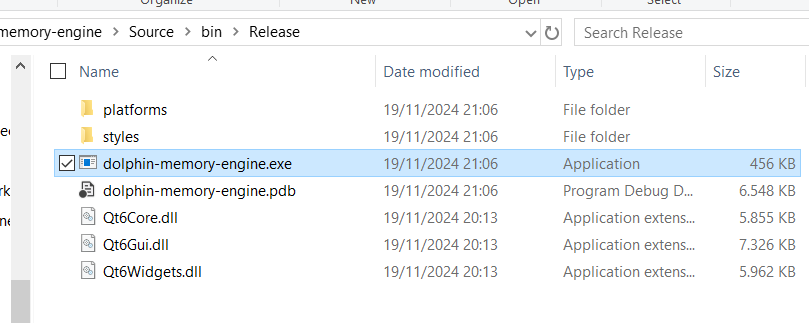
This can be forcibly ignored by adding

#define \_CRT\_SECURE\_NO\_WARNINGS

As shown at the beginning of the offending code.

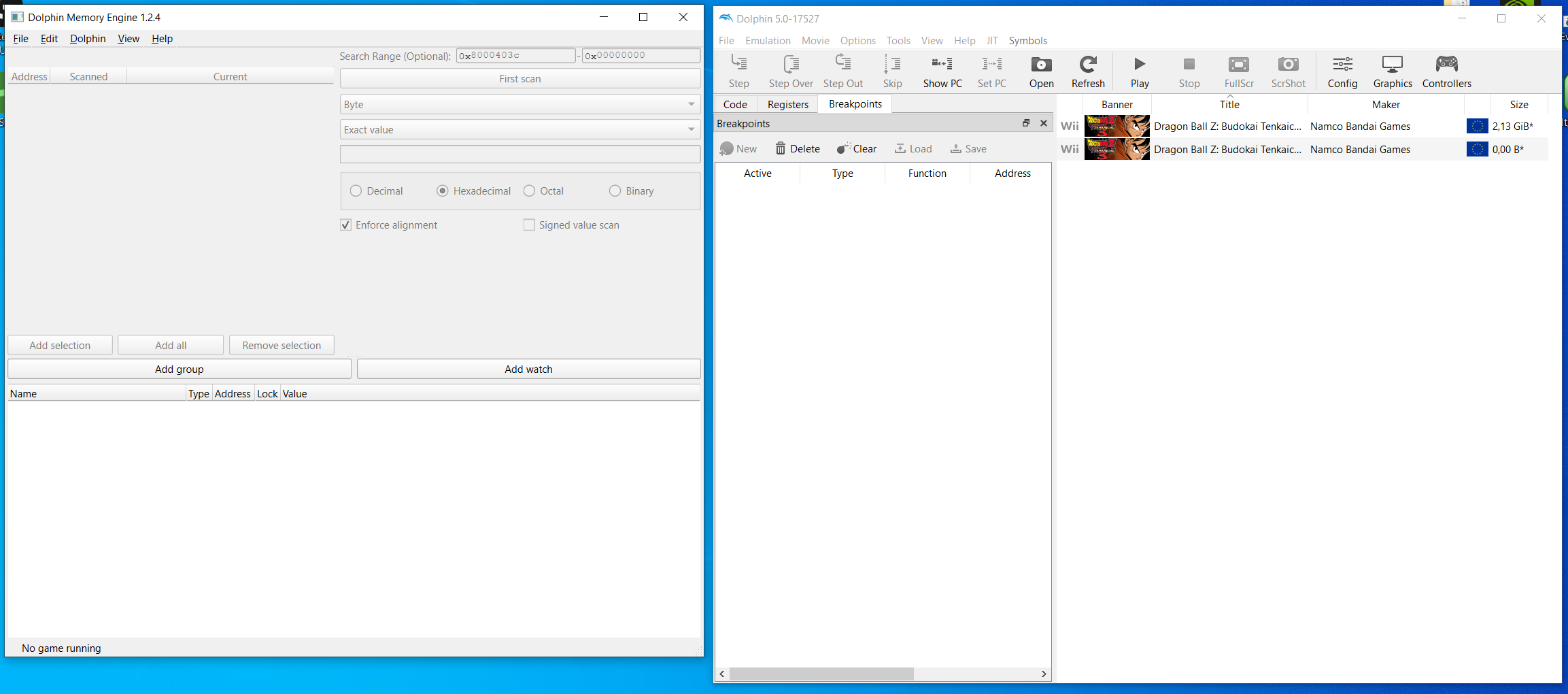
Make sure to build in release so you don’t suffer from slow performance.

You should be able to see the dolphin-memory-engine.exe under Source\bin\Release

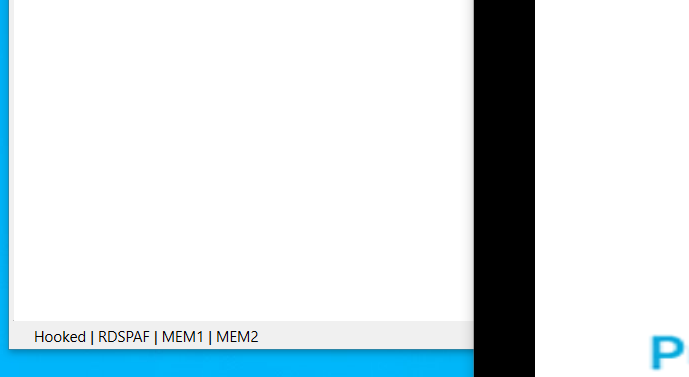


If you use Linux or OSX you probably know what you’re doing and how to fix them but just in case, feel free to mention any problem and add them here.

1. Run it and now combined with Dolphin debug mode, we now have our running environment:



1. Thanks to dolphin memory manager 1.2.4, it auto hooks itself when it detects a game running, as shown on the bottom left.



# Debugging and reverse engineering

## Overview:

The reverse engineering process combines three main tools working in tandem:

* Dolphin's debug menu for code execution control
* Save/Load states to quickly locate things
* Dolphin Memory Engine for memory monitoring
* Ghidra for code analysis and documentation

## Core workflow

1. Code Investigation

* Use Dolphin's debug menu to set breakpoints at key locations
* Monitor code execution through stepping and step-over commands
* Identify critical code paths during specific game events

1. Memory Analysis

* Track memory changes using Dolphin Memory Engine
* Label important memory addresses (e.g., menu selections, character states)
* Save memory watch groups (.dmw files) for future reference

1. Documentation

* Document findings in Ghidra through function renaming and commenting
* Map out key systems like:
  + Dragon net battle menu
  + Duel mode logic
  + Fighting game logic
  + Game rendering
  + Sound call system (rollback separation)
  + Input handling

This foundational understanding will be essential for future modifications, such as implementing rollback networking through a modified Dolphin build.

Note: All discoveries, even if seemingly unrelated, should be documented as they may prove valuable for understanding the game's architecture or avoiding redundant investigation later.

!!! will be continuing the document, currently making example by watching memory address in main menu to figure out: menu functions, target menu pointer address and hopefully some input handler function