NSMBW ASM & REing

1 PPC Assembly 1
1.1 Basics 1
1.2 ASM Instructions 1
1.3 Stack Save/Restore 1
1.4 Calling Functions in ASM 1
2 Ghidra 2
2.1 Search Functions 3
2.2 Assembly View 4
2.3 Decompilation 5
2.4 Pre Comments 6
2.5 Decompiled Headers 7
2.6 Function Graph 9
2.7 In- and Outgoing Calls 10
2.8 Function Call Graph 10
2.9 Function Signatures 11
2.10 Understanding the "Decompilation" 13
2.11 Search for Strings 14
2.12 Finding References to an Address (e.g. for a String) 15
2.13 Go To Address 15
3 Symbols 16
3.1 CodeWarrior 16
3.2 Clang 16
3.3 Filt 16
3.4 ExportSymbolsScript 16
4 Dolphin 17
4.1 OSReport 17
4.2 Break Points 18
4.3 Registers 19
4.4 Dolphin Memory Engine 19

Discord

1 PPC Assembly

1.1 Basics

https://docs.google.com/document/d/1y2jUmJn7aoXo1FohaU6gx1vJ8cVc1tg1FEjbgNf4tUY/mobilebasic
https://babbage.cs.qc.cuny.edu/IEEE-754/

1.2 ASM Instructions

https://mkwii.com/showthread.php?tid=863
https://wiibrew.org/wiki/Assembler_Tutorial
http://wiki.tockdom.com/wiki/Compiler
https://fail0verflow.com/media/files/ppc_750cl.pdf

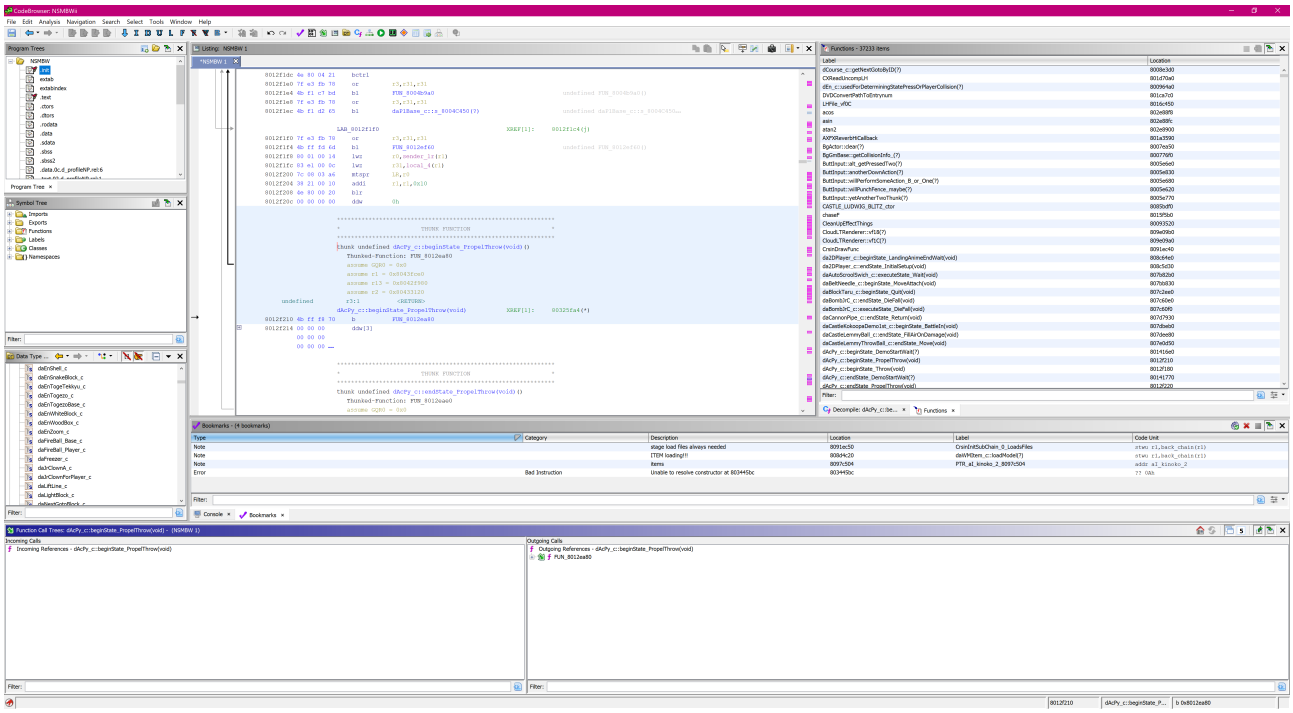
1.3 Stack Save/Restore

https://mkwii.com/showthread.php?tid=1108

1.4 Calling Functions in ASM

https://mkwii.com/showthread.php?tid=1052

2 Ghidra



https://ghidra-sre.org/
https://github.com/aldelaro5/ghidra-gekko-broadway-lang

2.1 Search Functions

Functions - 53 items (of 37233)

Label	Location
daEnKuriboBase_c::callsDeathByYoshiFumi(?)	80031940
daEnKuriboBase_c::endState_DieOther(?)	80032780
daEnKuriboBase_c::amIOnQuickSand(?)	800319a0
daEnKuriboBase_c::beginState_DieOther(?)	800326f0
daEnKuriboBase_c::beginState_TrplnJump(?)	80032570
daEnKuriboBase_c::beginState_Turn(?)	80032370
daEnKuriboBase_c::beginState_Walk(?)	800320c0
daEnKuriboBase_c::calledAlsoBy3DSetup(?)	80031360
daEnKuriboBase_c::calledAtEndOfOnCreate(?)	800312d0
daEnKuriboBase_c::calledByExecute(?)	800315a0
daEnKuriboBase_c::calledOnSpriteCollision(?)	80031780
daEnKuriboBase_c::chooseZPosition(?)	80031b30
daEnKuriboBase_c::drawModel(?)	800315e0
daEnKuriboBase_c::endState_TrplnJump(?)	800325b0
daEnKuriboBase_c::endState_Turn(?)	800323e0
daEnKuriboBase_c::endState_Walk(?)	80032160
daEnKuriboBase_c::executeState_DieOther(?)	80032790
daEnKuriboBase_c::executeState_TrplnJump(?)	800325c0
daEnKuriboBase_c::executeState_Turn(?)	800323f0
daEnKuriboBase_c::executeState_Walk(?)	80032170
daEnKuriboBase_c::freeze(?)	80032860
daEnKuriboBase_c::hitByJumping(?)	80031950
daEnKuriboBase_c::hitBySpinning(?)	80031870
daEnKuriboBase_c::hurtPlayer(?)	80031ab0
daEnKuriboBase_c::isFrozen(?)	80032800
daEnKuriboBase_c::isOnEnLiftRemoconTrpln(?)	80032080
daEnKuriboBase_c::loadModel(?)	80031370
daEnKuriboBase_c::makeQuicksandEffect(?)	80031a30
daEnKuriboBase_c::onCreate(?)	80031210
daEnKuriboBase_c::onDelete(?)	80031460
daEnKuriboBase_c::onDraw(?)	800315b0
daEnKuriboBase_c::onExecute(?)	80031470
daEnKuriboBase_c::playEnemyDownSound2(?)	800328c0
daEnKuriboBase_c::playerCollision(?)	80031790
daEnKuriboBase_c::playWalkAnim(?)	80031df0
daEnKuriboBase_c::setMyChainlinkMode(?)	80031cc0
daEnKuriboBase_c::setup3DStuff(?)	800312e0
daEnKuriboBase_c::setupModelMatrix(?)	800315f0

Filter: daEnKuribo

Decompile: dAcPy_c::be... Functions

In this view you can search for functions that were named by someone already.
It is recommended to follow the same scheme when adding new function names as was used so far.
Quite a lot of the classes have Japanese names such as daEn**Kuribo**Base_c for Goombas.

2.2 Assembly View

```
80a85280 94 21 ff f0     stwu       r1,back_chain(r1)
80a85284 7c 08 02 a6     mfspr      r0,LR
80a85288 3c 80 80 b0     lis        r4,offset aFlya                          = "flyA"
80a8528c 38 a0 00 00     li         r5,0x0
80a85290 90 01 00 14     stw        r0,sender_lr(r1)
80a85294 38 84 2b 6c     addi       r4,r4,offset aFlya                       = "flyA"
80a85298 93 e1 00 0c     stw        r31,local_4(r1)
80a8529c 3f e0 80 ad     lis        r31,offset DWORD_ARRAY_80ad39f0          =
80a852a0 3b ff 39 f0     addi       r31,r31,offset DWORD_ARRAY_80ad39f0      =
80a852a4 93 c1 00 08     stw        r30,local_8(r1)
80a852a8 7c 7e 1b 78     or         r30,r3,r3
80a852ac c0 3f 00 48     lfs        f1,0x48(r31)
80a852b0 4b fe e8 31     bl         daEnShell_c::setAnimation(void)          undefined daEnShell_c::setAnimat...
80a852b4 3c 80 80 b0     lis        r4,offset aWing_pata                     = "wing_pata"
80a852b8 c0 3f 00 48     lfs        f1,0x48(r31)
80a852bc 7f c3 f3 78     or         r3,r30,r30
80a852c0 38 a0 00 00     li         r5,0x0
80a852c4 38 84 2b 1c     addi       r4,r4,offset aWing_pata                  = "wing_pata"
80a852c8 4b ff ff 49     bl         daEnPata_c::changeAnimation(void)        undefined daEnPata_c::changeAnim...
80a852cc 80 1e 00 04     lwz        r0,0x4(r30)
80a852d0 38 60 00 04     li         r3,0x4
80a852d4 90 7e 07 7c     stw        r3,0x77c(r30)
80a852d8 54 00 c7 ff     rlwinm.    r0,r0,0x18,0x1f,0x1f                     nyb10 & 1
80a852dc 41 82 00 34     beq        nyb10m1is0
80a852e0 88 7e 0a 94     lbz        r3,0xa94(r30)
80a852e4 38 a0 00 80     li         r5,0x80
80a852e8 c0 3f 00 4c     lfs        f1,0x4c(r31)
80a852ec 38 00 00 01     li         r0,0x1
80a852f0 54 64 10 3a     rlwinm     r4,r3,0x2,0x0,0x1d
80a852f4 38 7f 00 10     addi       r3,r31,offset FLOAT_80ad3a00             = 1.0
80a852f8 7c 03 24 2e     lfsx       f0,r3,r4
80a852fc b0 be 0a 96     sth        r5,0xa96(r30)
80a85300 d0 3e 00 ec     stfs       f1,0xec(r30)
80a85304 d0 1e 01 18     stfs       f0,0x118(r30)
80a85308 90 1e 03 60     stw        r0,0x360(r30)
80a8530c 48 00 00 2c     b          LAB_80a85338

                    nyb10m1is0                                   XREF[1]:     80a852dc(j)
80a85310 88 9e 0a 94     lbz        r4,0xa94(r30)
```

This view shows what the game looks like internally — just some big amount of numbers representing equally confusing mnemonics (processor instructions).
Once you start to understand ppc asm it actually becomes relatively easy to understand though since a single instruction can only do really basic stuff with the exception of *rlwinm* and maybe some others.

I suggest starting learning asm following this and this. Understanding asm is essential when the decompiler puts out some unreadable stuff, which sadly happens quite a lot.
If you want to learn how to understand a function of a class that already was at least partly reverse engineered refer to this part of this document.

2.3 Decompilation

Decompile: daEnPata_c::endState_UDfly(void) - (NSMBW 1)

```
1
2  /* WARNING: This function may have set the stack pointer */
3
4  void daEnPata_c::endState_UDfly(void)(int param_1)
5
6  {
7    float fVar1;
8    undefined4 unaff_r30;
9    undefined4 unaff_r31;
10   undefined4 in_LR;
11   dword sender_lr;
12   dword back_chain;
13   dword local_8;
14   dword local_4;
15
16   puRam8043fcd0 = &_STACK_BASE_;
17   uRam8043fcd8 = unaff_r30;
18   uRam8043fcdc = unaff_r31;
19   uRam8043fce4 = in_LR;
20   daEnShell_c::setAnimation(void)(0x3ff0000000000000,param_1,"flyA",0);
21                    /* comment code */
22   daEnPata_c::changeAnimation(void)(0x3ff0000000000000,param_1,"wing_pata",0);
23   *(undefined4 *)(param_1 + 0x77c) = 4;
24   if ((*(uint *)(param_1 + 4) >> 8 & 1) == 0) {
25     *(undefined2 *)(param_1 + 0xa96) = 0x20;
26     fVar1 = (&FLOAT_80ad3a00)[*(byte *)(param_1 + 0xa94)];
27     *(float *)(param_1 + 0xec) = fVar1;
28     *(float *)(param_1 + 0x118) = fVar1;
29     *(undefined4 *)(param_1 + 0x360) = 1;
30   }
31   else {
32     fVar1 = (&FLOAT_80ad3a00)[*(byte *)(param_1 + 0xa94)];
33     *(undefined2 *)(param_1 + 0xa96) = 0x80;
34     *(float *)(param_1 + 0xec) = 0.00000000;
35     *(float *)(param_1 + 0x118) = fVar1;
36     *(undefined4 *)(param_1 + 0x360) = 1;
37   }
38   return;
39 }
40
```

Decompile: daEnPata_c:... × Functions ×

In this view you can see the decompiled C++ code of a function of a sprite. You can improve the decompilation by adding comments and other ways I sadly am not all that knowledgeable about.

In some cases this view is easy to understand and in others it's just confusing. Always take a look at both this and the assembly view if you are trying to understand something.

2.4 Pre Comments

Decompile: daEnPata_c::endState_UDfly(void) - (NSMBW 1)

```
/* WARNING: This function may have set the stack pointer */

void daEnPata_c::endState_UDfly(void)(int param_1)

{
  float fVar1;
  undefined4 unaff_r30;
  undefined4 unaff_r31;
  undefined4 in_LR;
  dword sender_lr;
  dword back_chain;
  dword local_8;
  dword local_4;

  puRam8043fcd0 = &_STACK_BASE_;
  uRam8043fcd8 = unaff_r30;
  uRam8043fcdc = unaff_r31;
  uRam8043fce4 = in_LR;
  daEnShell_c::setAnimation(void)(0x3ff0000000000000,param_1,"flyA",0);
  daEnPata_c::changeAnimation(void)(0x3ff0000000000000,param_1,"wing_pata
  *(undefined4 *)(param_1 + 0x77c) = 4;
  if ((*(uint *)(param_1 + 4) >> 8 & 1) == 0) {
    *(undefined2 *)(param_1 + 0xa96) = 0x20;
    fVar1 = (&FLOAT_80ad3a00)[*(byte *)(param_1 + 0xa94)];
    *(float *)(param_1 + 0xec) = fVar1;
    *(float *)(param_1 + 0x118) = fVar1;
    *(undefined4 *)(param_1 + 0x360) = 1;
  }
  else {
    fVar1 = (&FLOAT_80ad3a00)[*(byte *)(p
    *(undefined2 *)(param_1 + 0xa96) = 0x80;
    *(float *)(param_1 + 0xec) = 0.00000000;
    *(float *)(param_1 + 0x118) = fVar1;
    *(undefined4 *)(param_1 + 0x360) = 1;
  }
  return;
}
```

Edit Function Signature
Commit Params/Return P
Commit Locals
Copy Ctrl+C
Comments
Find... Ctrl+F
References
Properties

Set Plate Comment...
Set Pre Comment...
Set... Semicolon

Decompile: daEnPata_c:... | Functions

Set Comment(s) at Address 80a852c8

EOL Comment | Pre Comment | Post Comment | Plate Comment | Repeatable Comment

comment code

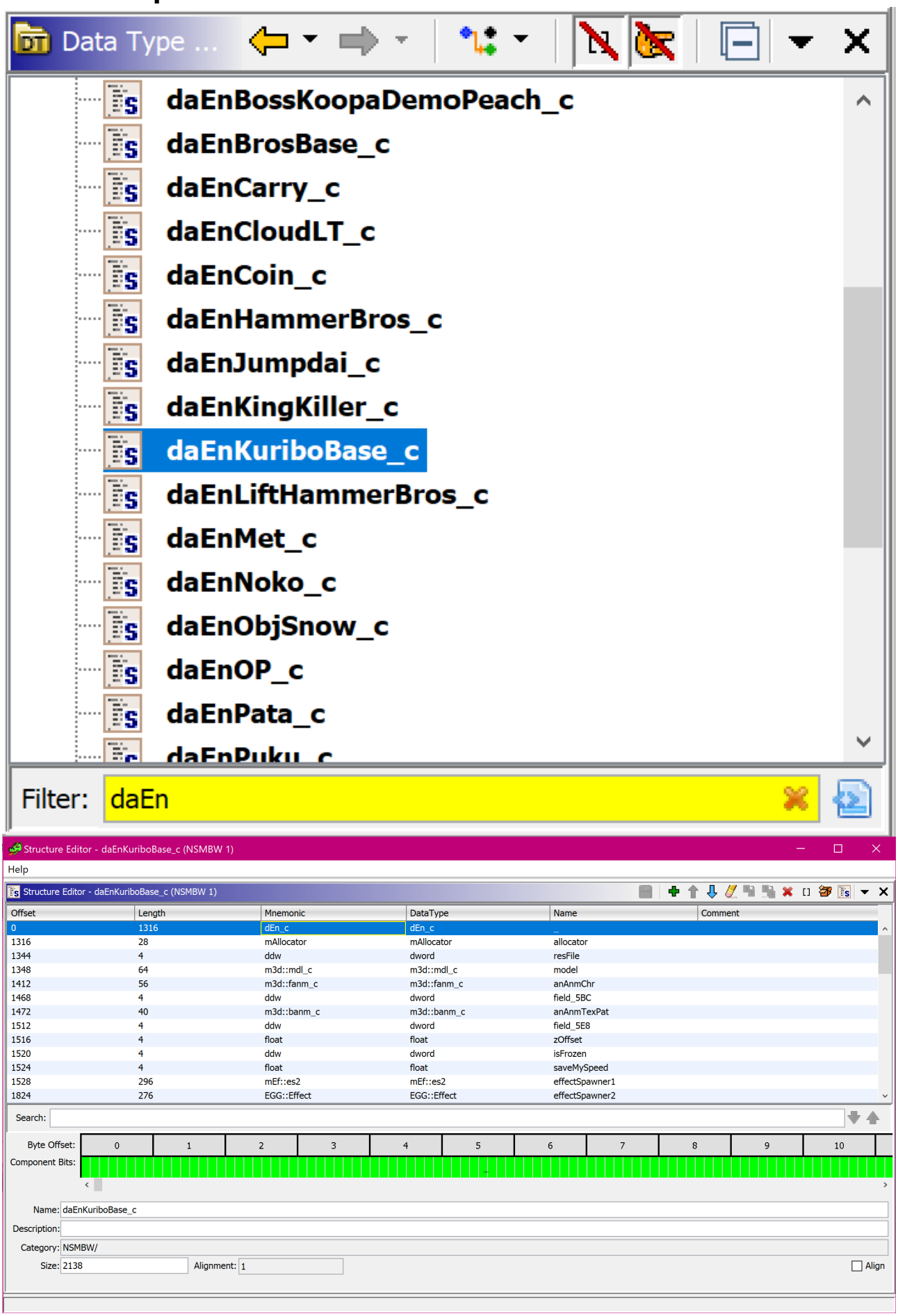
Add Annotation | Address

Enter accepts comment

OK | Apply | Dismiss

You can add multiple types of comments, but usually you should go with pre comments as they will show up in the decompilation. Comments help you keep an overview and help others to understand what you already understood if you ever where to share your Ghidra project.

2.5 Decompiled Headers



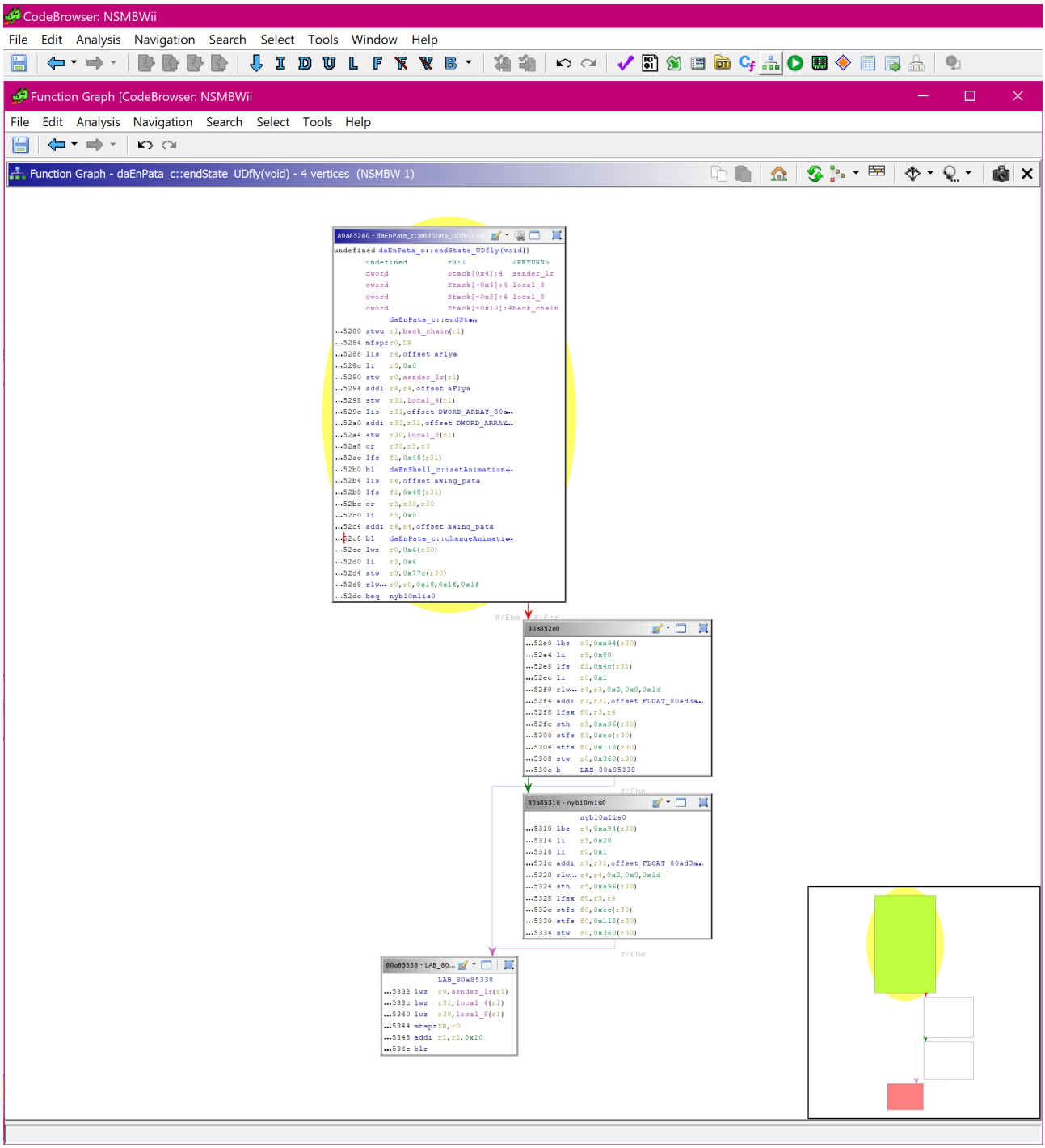
In this view you can find decompiled structs and classes (as structs). The second picture shows the class daEnKuriboBase_c.
It's first DataType is dEn_c which means this struct consists of a dEn_c object as well as some other stuff. This can be translated to the following C++ class:

```
class daEnKuriboBase_c : public dEn_c {
public:
    mAllocator allocator;
    dword resFile;
    m3d::mdl_c model;
    m3d::fanm_c anAnmChr;
    ...
};
```

Basically you can understand a class as a struct with functions. Inheritance happens by adding the struct of the inherited class at the start of the inheriting struct. This also provides the inheriting class with all functions of the inherited class.

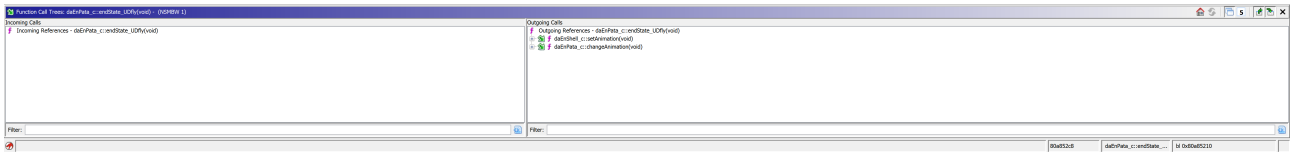
This view is quite helpful in some cases as can be seen here.

2.6 Function Graph



If you press the button on the right of the first picture the window in the second picture opens. This view shows how different parts of a function are connected, for example for if branches and loops. This way more complex functions become easier to understand in asm.

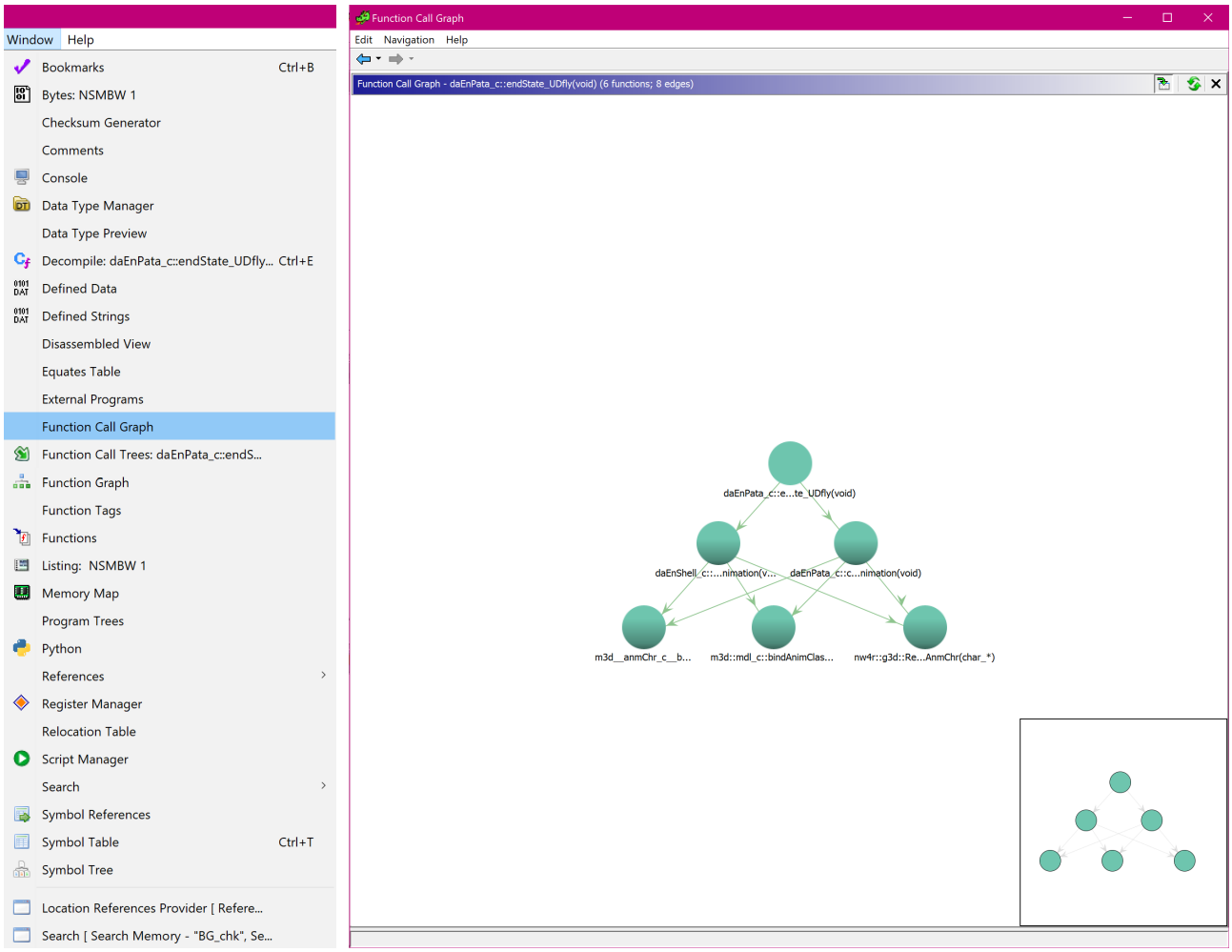
2.7 In- and Outgoing Calls



If you want to know which functions call a certain function, you can go to the "certain function“ and go through the incoming calls list to the left.
If you want to know what other functions a certain function calls you can take a look at the outgoing functions to the right.

The view probably can be found at the bottom of the main window. Otherwise you should be able to find it in the taskbar under "Window".

2.8 Function Call Graph



If you want to see how functions are connected the function call graph provides a good overview. You can right click vertices to expand incoming and outgoing function calls.

2.9 Function Signatures

Decompile: daEnPata_c::endState_UDfly(void) - (NSMBW 1)

```
/* WARNING: This function may have set the stack pointer */

void daEnPata_c::endState_UDfly(void)(int param_1)

{
  float fVar1;
  undefined4 unaff_r30;
  undefined4 unaff_r31;
  undefined4 in_LR;
  dword sender_lr;
  dword back_chain;
  dword local_8;
  dword local_4;

  puRam8043fcd0 = &_STACK_BASE_;
  uRam8043fcd8 = unaff_r30;
  uRam8043fcdc = unaff_r31;
  uRam8043fce4 = in_LR;
  daEnShell_c::setAnimation(void)(0x3ff0000000000000,param_1,"flyA",0);
  daEnPata_c::changeAnimation(void)(0x3ff0000000000000,param_1,"wing_pata",0);
  *(undefined4 *)(param_1 + 0x77c) = 4;
  if ((*(uint *)(param_1 + 4) >> 8 & 1) == 0) {
    *(undefined2 *)(param_1 + 0xa96) = 0x20;
    fVar1 = (&FLOAT_80ad3a00)[*(byte *)(param_1 + 0xa94)];
    *(float *)(param_1 + 0xec) = fVar1;
    *(float *)(param_1 + 0x118) = fVar1;
    *(undefined4 *)(param_1 + 0x360) = 1;
  }
  else {
    fVar1 = (&FLOAT_80ad3a00)[*(byte *)(param_1 + 0xa94)];
    *(undefined2 *)(param_1 + 0xa96) = 0x80;
    *(float *)(param_1 + 0xec) = 0.00000000;
    *(float *)(param_1 + 0x118) = fVar1;
    *(undefined4 *)(param_1 + 0x360) = 1;
  }
  return;
}
```

- Edit Function Signature
- Commit Params/Return P
- Commit Locals
- Copy Ctrl+C
- Comments
- Find... Ctrl+F
- References
- Properties

Edit Function at 80a85280

void daEnPata_c::endState_UDfly(void) (int param_1)

Function Name: daEnPata_c::endState_UDfly(void)

Calling Convention: __stdcall

Function Attributes: Varargs, In Line, No Return, Use Custom Storage

Function Variables

Index	Datatype	Name	Storage
	void	<RETURN>	<VOID>
1	int	param_1	r3:4

Call Fixup: -NONE-

OK Cancel

A function can be passed several values but it can return only one value. Usually the first passed value gets stored in *r3*, the second in *r4* and so on, floats get stored in *f1* to *f8* though. You can call functions with pure assembly if you understand how to store the values to be passed in the correct register before calling the function with a *bl* instruction, here an example:

```
lbz r4, 0(r3)          # load the byte of the array we just selected —> settings value
li r3, 0x107           # Actor ID of Bouncing Wendy Ring
addi r5, r31, 0xAC     # Position —> r31 stores the address of the Koopa, 0xAC is the
                       # offset to it's pos vector
li r6, 0               # Rotation
lbz r7, 0x38F(r31)     # Layer
bl CreateActor
```

You can find the entire asm code here.

If a function returns a value, it does so by storing the result in *r3* (and sometimes *r4* as well) for non-float values and in *f1* for floats before calling *blr* ("return"). An example can be found here.

You can find more information about register conventions here.

2.10 Understanding the "Decompilation"

Decompile: daEnPata_c::endState_UDfly(void) - (NSMBW 1)

```
/* WARNING: This function may have set the stack pointer */

void daEnPata_c::endState_UDfly(void)(int param_1)

{
  float fVar1;
  undefined4 unaff_r30;
  undefined4 unaff_r31;
  undefined4 in_LR;
  dword sender_lr;
  dword back_chain;
  dword local_8;
  dword local_4;

  puRam8043fcd0 = &_STACK_BASE_;
  uRam8043fcd8 = unaff_r30;
  uRam8043fcdc = unaff_r31;
  uRam8043fce4 = in_LR;
  daEnShell_c::setAnimation(void)(0x3ff0000000000000,param_1,"flyA",0);
  daEnPata_c::changeAnimation(void)(0x3ff0000000000000,param_1,"wing_pata",0);
  *(undefined4 *)(param_1 + 0x77c) = 4;
  if ((*(uint *)(param_1 + 4) >> 8 & 1) == 0) {
    *(undefined2 *)(param_1 + 0xa96) = 0x20;
    fVar1 = (&FLOAT_80ad3a00)[*(byte *)(param_1 + 0xa94)];
    *(float *)(param_1 + 0xec) = fVar1;
    *(float *)(param_1 + 0x118) = fVar1;
    *(undefined4 *)(param_1 + 0x360) = 1;
  }
  else {
    fVar1 = (&FLOAT_80ad3a00)[*(byte *)(param_1 + 0xa94)];
    *(undefined2 *)(param_1 + 0xa96) = 0x80;
    *(float *)(param_1 + 0xec) = 0.00000000;
    *(float *)(param_1 + 0x118) = fVar1;
    *(undefined4 *)(param_1 + 0x360) = 1;
  }
  return;
}
```

Calculator — Programmer

EC

HEX	EC
DEC	236
OCT	354
BIN	1110 1100

QWORD MS

Bitwise Bit Shift

A << >> CE ⌫
B ( ) % ÷
C 7 8 9 ×
D 4 5 6 −
E 1 2 3 +
F +/− 0 . =

Structure Editor [daEnPata_c, daEnNoko_c, daEnShell_c, dEn_c, dActorMultiState_c, dStageActor_c, dActor_c]

Structure Editor - dActor_c (NSMBW 1)

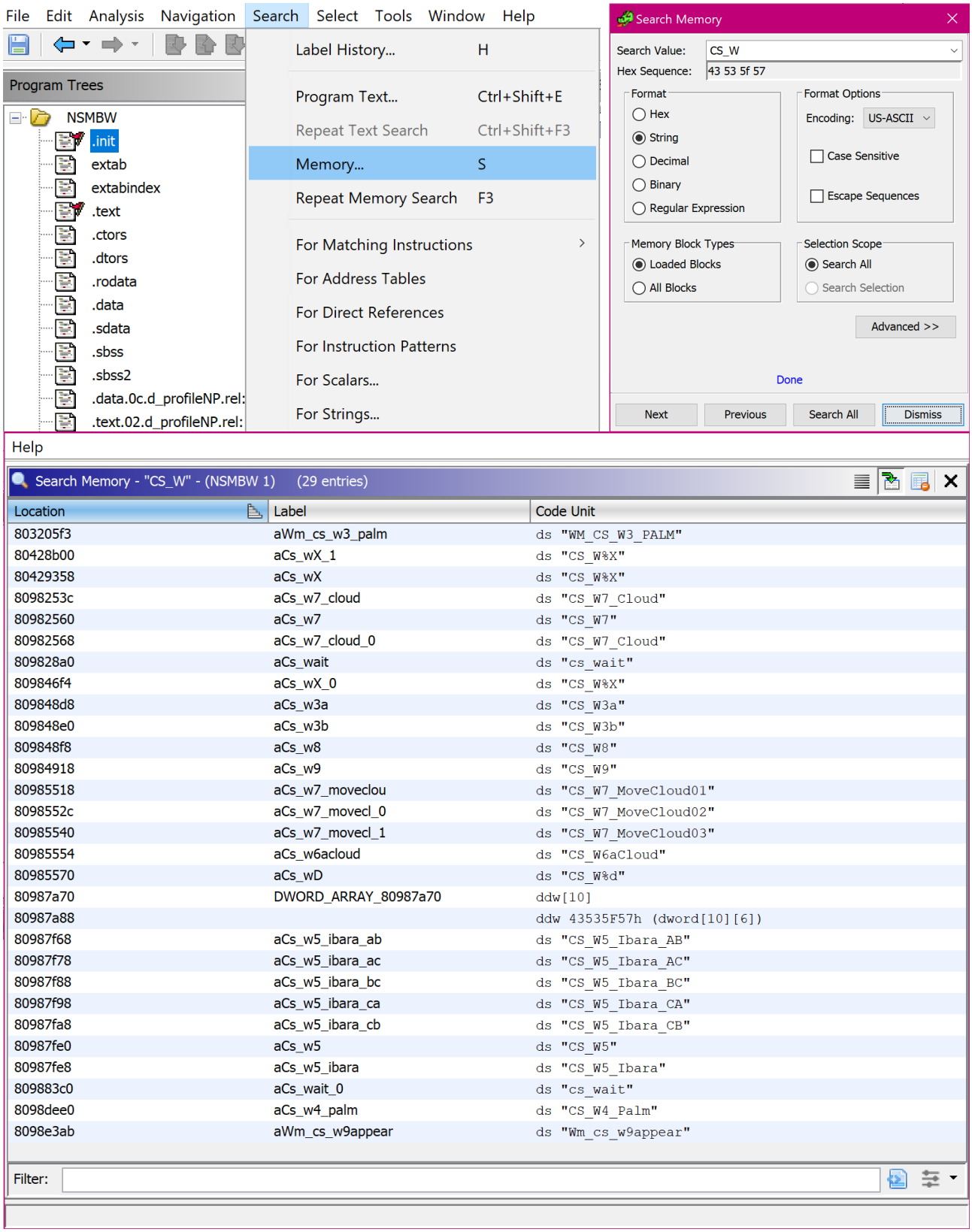
Offset	Length	Mnemonic	DataType	Name	Comment
170	1	??	undefined		
171	1	??	undefined		
172	12	VEC3	VEC3	pos	
184	12	VEC3	VEC3	last_pos	
196	12	VEC3	VEC3	pos_delta	
208	12	VEC3	VEC3	field_D0	
220	12	VEC3	VEC3	scale	
232	12	VEC3	VEC3	speed	
244	12	VEC3	VEC3	max_speed	
256	6	S16Vec	S16Vec	rot	
262	2	dw	word	field_106	

Byte Offset: 233 234 235 236 237 238 239 240 241 242 243
Component Bits: speed

Name: dActor_c
Description:
Category: NSMBW/
Size: 293 Alignment: 1 Align

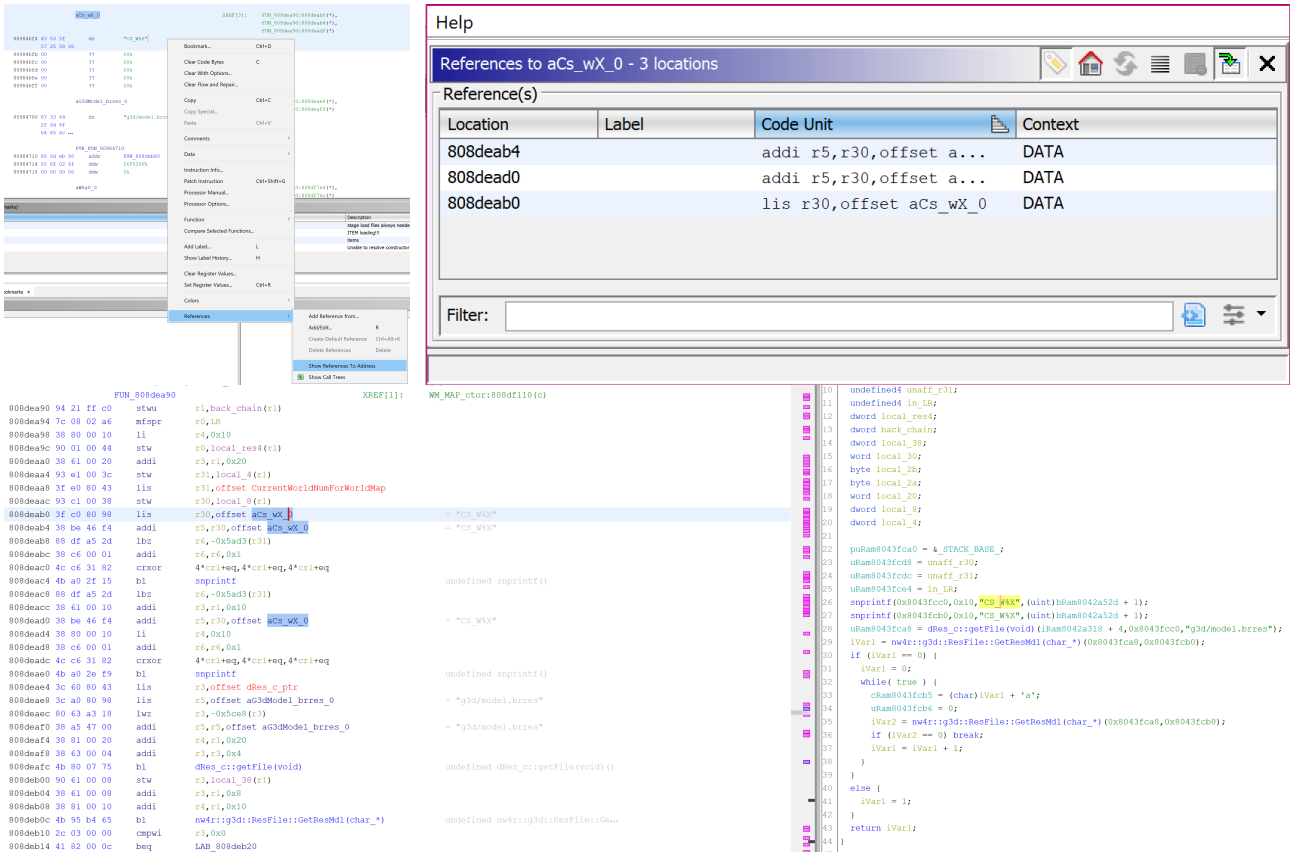
In the first picture you can see that the value at the address *param_1 + 0xEC* gets overwritten, indicating that *param_1* is a pointer to an object, which probably is of the type daEnPata_c in this case.
If you convert 0xEC to a decimal number you get 236. If you now open the "Structure Editor" with the daEnPata_c struct and open the first attribute by right-clicking it until you get to the offset 236 which is part of the inherited dActor_c class you can see that the value being overwritten is speed.y, since a VEC3 consists of 3 values with a length of 4 each.
At the bottom of the third picture you can see which classes the daEnPata_c class inherits (multilevel inheritance).

2.11 Search for Strings



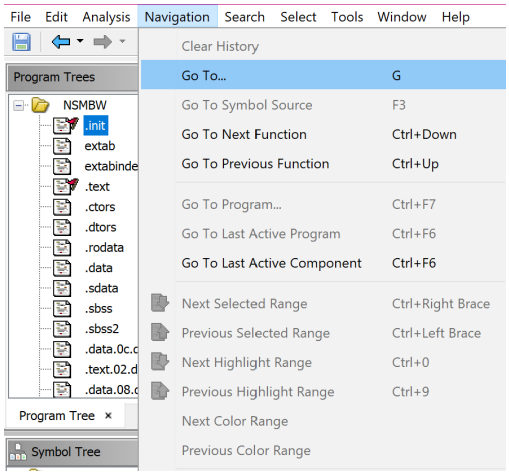
You can search for strings, e.g. names of .arc or .brres files (without the file extension). This way you can easily find code responsible for everything concerning the file you searched for. Usually functions belonging to the same class are right next to each other so you can easily find other functions belonging to the same class as the function which loads e.g. a 3d model.

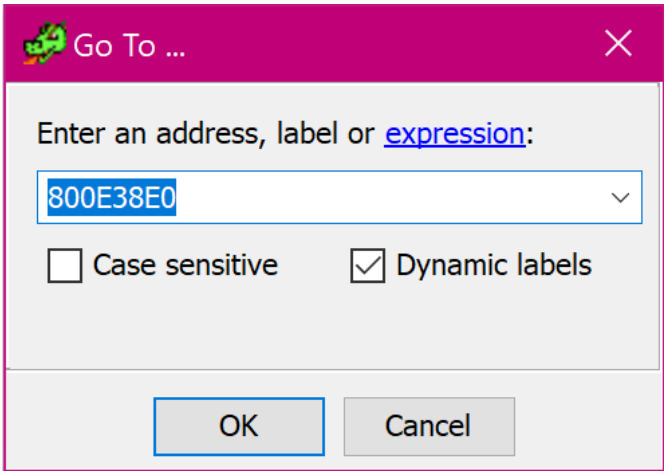
2.12 Finding References to an Address (e.g. for a String)



If you want to know where a specific address is referenced you can do so by following the first picture. With this you can find a function that uses a string found following this.

2.13 Go To Address





If you want to go to a specific address you can simply press *G* on your keyboard or follow the steps on the first picture. This is quite useful for jumping between different addresses.

3 Symbols

Symbols are mostly function names written in a way the used compiler uses internally — or at least this is how I understood it …
Symbols include the function name, class name if the function belongs to a class and also the data types of the passed values. It doesn't include the returned data type of the function though.

All symbols of functions from the original game are saved in a .x file in the Kamek folder.
If you want to call a custom C++ function from an asm code you have to create the symbol of your own function.
If you want to call a function of a reverse engineered class but the compiler says something about a function pointing at 0 you can copy the symbol from the error message and add it to the .x file together with its address found in Ghidra.

Since Newer based mods can be compiled using Clang or CodeWarrior there are two naming schemes (thanks at RedStoneMatt for explaining this to me):

3.1 CodeWarrior

functionName__XclassNameY:

- X is the number of letters in className (so if your class is named "yay" then X is 3)
- Y depends on what values are passed to the function
  - most of the time it starts with F and then what is after is v if it's a void or i if it's an int
  - if it has other parameters then it depends on those
- you might find some related stuff in hooks.py

3.2 Clang

_ZNXclassNameYfunctionNameZ

- X is the number of letter in className
- Y is the number of letter in functionName
- Z depends of what values are passed to the function
  - most of the time it starts with E the stuff after depends on the passed data types

3.3 Filt

Somewhere in your folder containing your compiler there should be a program called *filt*, for example */Kamek/tools/devkitPPC/bin/powerpc-eabi-c++filt.exe*.
You can convert symbols to actual function names using this tool.

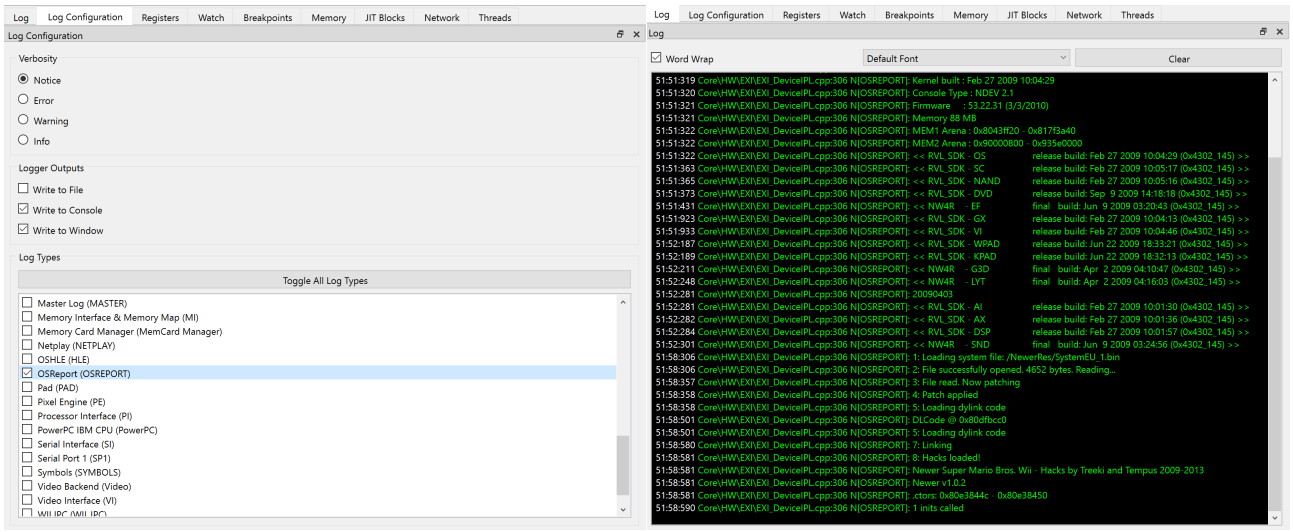
3.4 ExportSymbolsScript

RoadrunnerWMC wrote a Python script for Ghidra for exporting symbols which can be found here.

4 Dolphin

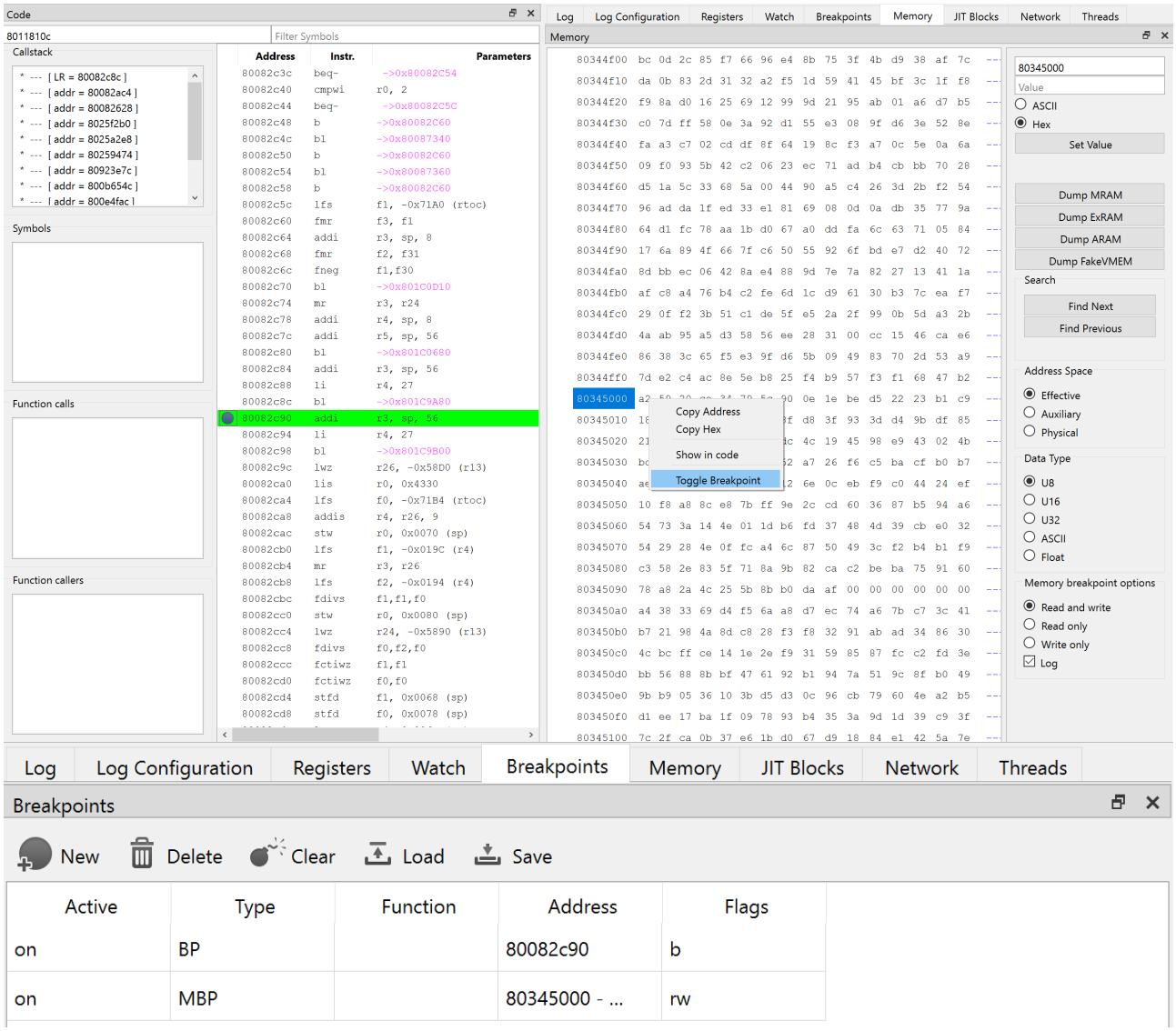
You can use Dolphin to debug your code and understand the internals of the game. You have to start Dolphin with the argument *-d* though.
On Windows you can create a shortcut to your *dolphin.exe* and add *-d* to the file path in the shortcut.
On Mac OS you can open Dolphin from terminal using *open Dolphin.app --args -d* inside the folder in which your Dolphin.app is.

4.1 OSReport



In your code you can call a function called OSReport. It basically works the same as printf.
You can see OSReports by activating them as shown in the first picture.

4.2 Break Points



You can set breakpoints in Dolphin to pause the game when an instruction at a specific address is executed (first picture) or the game reads or writes a value to a specific address (second picture).

In the Breakpoints tab you can save your current breakpoints and load the last saved breakpoints. This way you can quickly load the same breakpoints again after restarting the game.

4.3 Registers

Log | Log Configuration | Registers | Watch | Breakpoints | Memory | JIT Blocks | Network | Threads

Registers

r0	000b006c	f0	3ff0000000000000	c0c2c00000000000	IBAT0	800001ff00000002	SR0	00000000
r1	8043fa58	f1	0000000000000000	0000000000000000	IBAT1	0000000000000000	SR1	00000000
r2	80433360	f2	0000000000000000	4060000000000000	IBAT2	810000ff01000002	SR2	00000000
r3	8043fa90	f3	0000000000000000	bff0000000000000	IBAT3	0000000000000000	SR3	00000000
r4	cc008000	f4	0000000000000000	c069b30320000000	IBAT4	800001ff00000002	SR4	00000000
r5	cc010000	f5	3ff0000000000000	0000000000000000	IBAT5	c0001fff0000002a	SR5	00000000
r6	80429898	f6	3ff0000000000000	0000000000000000	IBAT6	810000ff01000002	SR6	00000000
r7	0000001e	f7	0000000000000000	bff3dfc000000000	IBAT7	e00001fee0000002	SR7	00000000
r8	00000010	f8	0000000000000000	3ff0000000000000	DBAT0	800001ff00000002	SR8	00000000
r9	810d6ac4	f9	0000000000000000	c06e000000000000	DBAT1	c0001fff0000002a	SR9	00000000
r10	0000000a	f10	0000000000000000	0000000000000000	DBAT2	810000ff01000002	SR10	00000000
r11	8043fb28	f11	3ff0000000000000	4079000000000000	DBAT3	e00001fee0000002	SR11	00000000
r12	80000090	f12	3ff0000000000000	0000000000000000	DBAT4	0000000000000000	SR12	00000000
r13	8042f980	f13	0000000000000000	c069b30320000000	DBAT5	0000000000000000	SR13	00000000
r14	81540b1c	f14	0000000000000000	bff0000000000000	DBAT6	0000000000000000	SR14	00000000
r15	00000000	f15	0000000000000000	4060000000000000	DBAT7	0000000000000000	SR15	00000000
r16	00000000	f16	0000000000000000	0000000000000000	TB	0091635b8ca15af3	GQR0	00000000
r17	810c3fa8	f17	0000000000000000	0000000000000000	PC	80082c90	GQR1	00000000
r18	00000001	f18	0000000000000000	0000000000000000	LR	80082c90	GQR2	00040004
r19	0000001e	f19	0000000000000000	0000000000000000	CTR	8024a410	GQR3	00050005
r20	00000010	f20	0000000000000000	0000000000000000	CR	22000988	GQR4	00060006
r21	810d6ac4	f21	0000000000000000	0000000000000000	XER	00000000	GQR5	00070007
r22	0000000a	f22	0000000000000000	0000000000000000	FPSCR	00000004	GQR6	05070507
r23	00000000	f23	0000000000000000	0000000000000000	MSR	0000b032	GQR7	08070807
r24	8043fb68	f24	0000000000000000	0000000000000000	SRR0	801a9cd4	HID0	0000c200
r25	810c3fa8	f25	0000000000000000	0000000000000000	SRR1	0000b032	HID1	80000000
r26	810d6ac4	f26	0000000000000000	0000000000000000	Exceptions	00000000	HID2	f00f0000
r27	00000010	f27	0000000000000000	0000000000000000	Int Mask	00004ff8	HID4	83900000
r28	0000001e	f28	c06e000000000000	c06e000000000000	Int Cause	00010000		
r29	00000000	f29	3ff3dfc000000000	3ff3dfc000000000	DSISR	00000000		
r30	0000000a	f30	3ff3dfc000000000	c071000000000000	DAR	00000000		
r31	80d25d24	f31	c06e000000000000	3ff0000000000000	Hash Mask	00000000		

When the game is paused — either by pressing the pause button or by reaching a breakpoint — you can see the current state of the different registers. Values that changed since the last pause are displayed in red.

4.4 Dolphin Memory Engine

https://github.com/aldelaro5/Dolphin-memory-engine