

O'KRABS™



BFBB

Decompilation

REVERSE ENGINEERING GUIDE

**MP, Square, Seil
& Friends**

Foreword by Shift

Contents

Introduction	4
About this Book	4
What you will learn	4
Prerequisite Knowledge	4
What you need to know	4
What you don't need to know	5
What is Decompilation?	5
Git and GitHub Fundamentals for Open-Source Contributions	8
What is Git?	8
What is GitHub?	8
Setting Up Git	8
Forking a Repository on GitHub	8
Creating a Pull Request (PR)	8
Responding to PR Reviews	9
Github Command Quick Reference	9
Creating and Checking Out Branches	9
Making Changes and Committing	9
Pushing Changes to Your Fork	10
Integrating Upstream Changes into your Fork	10
BFBB Overview	11
Heavy Iron Studios	11
EvilEngine	11
.HIP/.HOP Files	11
BFBB Decompilation Project Overview	11
Why Decompile BFBB?	11
Why Choose the GameCube Version?	11
How is this Possible?	11
Legality	11
Repository Layout	11
DTK	11
Source Code	12
Third party libraries	12
/gc/ - GameCube specific code	12
/x/ - Core game engine code	12
/Game/ - BFBB game code	12
Tools	12
Objdiff	12
What is Objdiff?	12
Getting Started	12
Ghidra	12
What is Ghidra?	12
Getting Started	12
Ghidra-isms	12
concat44	12
VS Code	12
Code Formatting	12
Assembly Concepts	12

Sections	12
.comm	12
.bss	12
.ctors	12
.data	12
.rodata	12
.sbss	12
.sbss2	12
.sdata	12
.sdata2	12
.text	12
C++ Low Level Concepts	12
Mangled Names	12
Virtual Tables	13
Heavy Iron C++ Code Standards & Patterns	13
Primitive Type Aliases	13
Multi-character Literals, or 'NTR0'	13
Compiler Flags	13
Optimization Level	13
Decompilation Guides	13
1. The Simplest Function: Nothing	13
Our First Function – NPCWidget::Reset()	14
2. Returning Values	17
3. A Function with Some Logic	17
4. A Function with a For Loop	17
5. A Function with a Switch Statement	17
6. A Large Function	17
Assembly Patterns	18
Early Returns	18
Ternary Operators	18
Modulus	18
Unrolled For Loops	18
Switch Statements	18

Introduction

About this Book

Welcome to the exciting world of video game decompilation! This book serves as an introduction to the BFBB Decompilation Project¹, which is an organized community effort to reverse engineer the source code of the 2003 platform game “SpongeBob SquarePants: Battle for Bikini Bottom” (BFBB) for Nintendo GameCube.

What you will learn

The main goal of this book is to act as a complete guide that can take someone (with the required prerequisite knowledge from zero understanding of decompilation and assembly code to being able to read, understand, and decompile BFBB’s C++ source code and corresponding assembly code. It also aims to answer any and all questions that you might have related to any aspect of the project.

So what does this mean for you?

If you read through this book you will learn about BFBB’s game engine, the decompilation process, and how C++ code translates to assembly. You will see examples of how real game functions are decompiled from nothing but bytes. You will deepen your understanding of computer science, You will gain the knowledge required to be able to create your own custom mods and builds of BFBB. And most excitingly, you will get an exclusive look behind the curtains at the source code of one of your favorite childhood games and see how the sausage is made.

Your reverse engineering effort will unearth and give new life to long lost-and-forgotten source code as if you were a digital archeologist. It is also likely that your newly decompiled code will run on hundreds of thousands if not millions of PCs as the source code will serve as the base for a highly desired native BFBB PC port (and to perhaps other platforms).

You have an opportunity to become a part of gaming history.

Prerequisite Knowledge

Reverse engineering is an advanced topic. The goal of this book is to break it down into digestible parts that make it easy to follow along and learn. However, the contents of this book are written in a way that assumes that the reader has a comfortable foundational understanding of programming and computer science.

What you need to know

Here are a list of things that you do need to know before reading this book.

1. A moderate familiarity with C++ and the language features that the BFBB codebase makes heavy use of:
 - Classes, Inheritance, and Object Oriented Programming
 - Polymorphism (Virtual Functions)
 - Pointers and References
 - Callback Functions
 - Casting
2. Have an enthusiasm for BFBB and/or reverse engineering

Note that a lack of experience with C++ can be made up for with a solid understanding of programming in general. What ultimately matters is that you have a solid understanding of how to write code.

¹<https://github.com/bfbbdecomp/bfbb>

It is unreasonable to expect to be able to understand or follow along with this book without having ever written code before. If you are interested in the topic of this book but have never written code, it is recommended to learn programming fundamentals, along with each one of the bullet points above and then come back with some experience.

What you don't need to know

Here are a list of things that you don't need to know beforehand. If you do know them, it's a large plus, but if you don't, don't worry. You will learn these things while reading this book:

1. How to write or read assembly language, be it PowerPC or any other type of instruction set
2. How to use Ghidra or other binary analysis/reverse engineering tools
3. Math, or anything related to 3D programming
4. Game development or game programming techniques
5. Version Control or how to use Git

“But how can we reverse engineer a 3D game without needing to know game programming or 3D math?”

— You, probably

Great question! We will explain this idea in more detail later, but the answer is surprisingly simple: The compiler will tell you if you're right or wrong.

For now think of it like this:

Imagine you have the formula $x + 1 = 4$. There are an infinite amount of numbers you can substitute for x , but there is only one correct answer. You don't have to know anything about the number 4, or why the number 4 is important in this context, or about x , or why we are adding instead of dividing, or what the formula means. You just have to solve for x and that's it. When you realize that $x = 3$ you're done. You can forget about it and move on.

The decompilation process is similar. Generally speaking there is only one way to write code that will compile to the same assembly output. You don't have to even know what the code is trying to do, you just have to replicate the logic.

What is Decompilation?

Let's take a moment to define some foundational terms that will help us understand what decompilation is and also be useful going forward:

Machine Code Computer code consisting of machine language instructions, which are used to directly control a computer's central processing unit (CPU).

Assembly Language A low-level programming language with a very strong correspondence between the instructions in the language and the architecture's machine code instructions.

High Level Language A programming language with strong abstraction from the details of the computer. It may use natural language, making the process of developing a program simpler and more understandable than when using a lower-level language.

Compiler A computer program that translates computer code written in one programming language (the source language) into another language (the target language).

- The name “compiler” is primarily used for programs that translate source code from a high-level programming language to a low-level programming language (e.g. assembly language, object code, or machine code) to create an executable program.

Decompilation A type of reverse-engineering that performs the opposite operations of a compiler.

Type A description of a set of values and a set of allowed operations on those values

Statically Typed Language A language is statically typed if the type of a variable is known at compile time.

- The main advantage here is that all kinds of checking can be done by the compiler, and therefore a lot of trivial bugs are caught at a very early stage.
- Examples: C, C++, Java, Rust, Go, Scala

Dynamically Typed Language A compiler or an interpreter assigns a type to all the variables at run-time. The type of a variable is decided based on its value.

- Programs written using dynamically typed languages are more flexible but will run even if they contain errors.
- Examples: Perl, Ruby, Python, PHP, JavaScript, Erlang

When most people talk about programming today, they are almost always referring to high level programming languages. Programming languages come in all shapes and sizes, and each one offers a different level of capabilities and abstraction. Some are dynamically typed, some are statically typed. Some are interpreted, some are compiled. Each programming language varies in ease of use, performance, safety, and other factors.

Battle for Bikini Bottom is written in the C++ programming language. In later chapters we will discuss how this fact was discovered and what implications it has on the decompilation project.

C++ is a statically typed, compiled programming language that has been around since 1985. It is a very common choice for building game engines because it generates extremely fast machine code while at the same time being expressive and generally straightforward to program in.

The C++ compilation process involves a series of steps which translate the original human written source code into what ultimately becomes an executable file. Refer to Figure 1 for an illustration of this process.

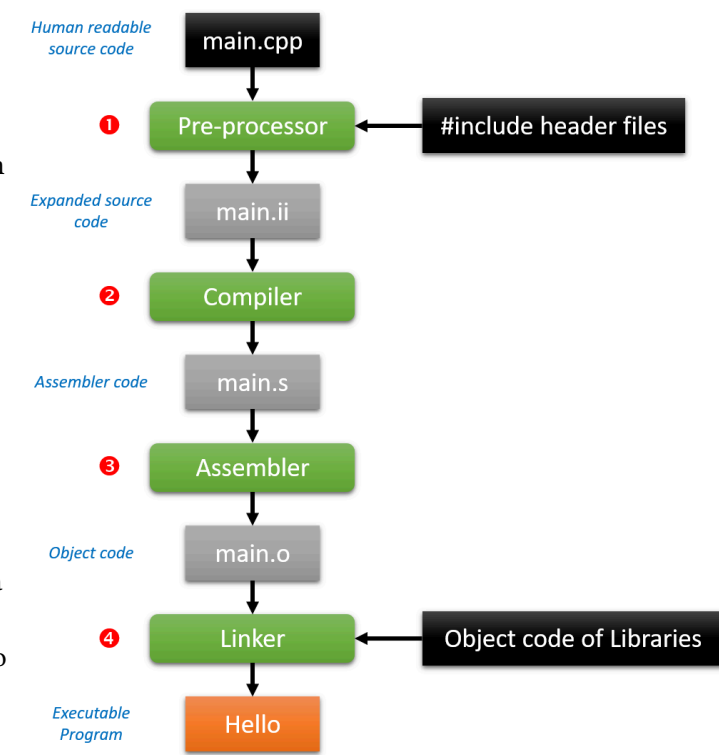


Figure 1: The C++ compilation process

If the compilation process is a series of steps from $A \rightarrow B$, then the decompilation process is simply the same process in reverse from $B \rightarrow A$. and it looks like this:

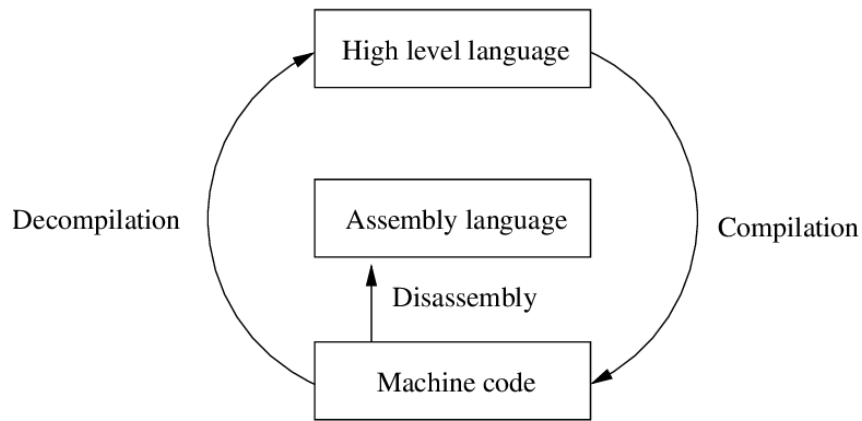


Figure 2: The decompilation process

It's important to understand that going in reverse is not something that can be done automatically in the same way that source code is compiled to machine code. Decompilation requires reverse engineering the machine code to understand the intent, and rewriting code at a high level which matches the same logic.

The decompilation process is like trying to deduce the original ingredients used to make a cake, except the only thing you have to work with is the cake which has already been baked.

Git and GitHub Fundamentals for Open-Source Contributions

What is Git?

Git is an open-source version control system commonly used to make it easier for developers to introduce changes to a project without stepping on each others' toes. Git will track changes to files across time in the form of **commits**.

A commit contains:

- Code changes (referred to as the “commit diff”)
- Author/Timestamp information
- A developer-written message describing the code changes

Commits are made on **branches**. Branches allow developers to compartmentalize code changes by particular features, bug fixes, documentation updates, etc. Branches can be merged into other branches, allowing for developers to work on things at the same time while minimizing conflicts created by simultaneous work.

All of these branches live inside of a Git **repository**, which is the collection of all branches and commits relevant to a Git project.

What is GitHub?

Github, not to be confused with Git itself, is a Git repository hosting service commonly used by open-source projects, as Github provides free hosting for public and private repositories alike. It is leveraged by Decomp Toolkit Template projects to run Github Actions, which can be used to automate things like progress website updates and create Discord notifications when people contribute to a project.

Setting Up Git

[Here is a link to a guide](#) hosted by Github is a great place to start for setting up and configuring Git. Follow it and you'll be able to set up Git locally on your machine!

Forking a Repository on GitHub

[Here is a link to a guide](#) provided by GitHub which is helpful for understanding repository forks, why they're useful, and how to work with them depending on your operating system (Windows, Mac, Linux).

Check out the guide for step-by-steps instructions on how to:

- Create a Github repository fork
- Clone that repository fork
- Configure your fork to be able to pull changes from the “upstream” original repository

Creating a Pull Request (PR)

A pull request is a method by which you can propose the merging of your code changes into the original repository from your fork.

[Here is a link to a guide](#) hosted by GitHub which will show you how to create a pull request once you've pushed changes to your remote fork. It will take you step-by-step through the pull request creation process.

Your pull request should contain:

- A descriptive summary of your proposed changes contained in the PR
- A helpful description which goes into detail about what changes you made and why you did them as needed

Responding to PR Reviews

Once you've opened a pull request, you're not done yet! There's still some steps that will happen before your code is merged into the original project repository:

- A maintainer will review your pull request
- The maintainers may post comments to discuss your changes with you
- The maintainers may request code changes they would like to see to better align your code with project standards

Engage with the project maintainers on your pull request, pushing changes to your code as needed until all feedback is addressed. Don't worry about updating your pull request on GitHub - pushing changes to your fork branch will automatically update the pull request for you.

Once all of the feedback has been addressed, the maintainers will merge your changes into the project repository. Congratulations - you're now a contributor to open source, and hopefully the newest contributor to the BfBB Decomp project!

Github Command Quick Reference

This section is intended to serve as a quick reference for some common Git workflows that you will use while working on your project. The documentation here is provided for the Git command line interface - you will have to reference the particular documentation for your GUI client (Github Desktop, etc.) to determine how to perform these actions using your GUI of choice.

Creating and Checking Out Branches

Branches are used to isolate development of particular features or bugs into compartmentalized units.

To checkout a new branch `my_branch` using the `main` branch as a base:

```
git checkout main
git checkout -b my_branch
```

To checkout an existing branch `my_branch`:

```
git checkout my_branch
```

To delete a branch `my_branch` (**Note: This will delete any unmerged changes for good! Be careful!**):

```
git branch -D my_branch
```

Making Changes and Committing

Files must be staged before the changes contained in those files can be committed.

To stage an individual file:

```
git add path/to/my/file
```

To stage all tracked files:

```
git add -u
git status
```

`git status` is optional but it can be a good sanity check to make sure that the files you intended to stage are staged and no files you did not intend to stage are staged.

To commit staged changes:

```
git commit -m "<commit message>"
```

A good commit message should:

- Contain a message summary no longer than 72 characters
- Describe what changes are made by the commit
- Explain any decisions or limitations contained in the changes (if the commit contains complex changes)

A good example commit for the BfBB Decomp project follows the format:

<decompiled filename>: <change to file>

Some examples of good commit messages:

zCamera: Matches for zCameraFlyRestoreBackup and zCameraRewardUpdate functions

zNPCGoalAmbient: zNPCGoalJellyBumped Near 100% Match

iTRC: Data Updates and Matches for Font Rendering Functions

Pushing Changes to Your Fork

To push changes from your local repository to your remote fork repository and configure tracking for that branch:

```
git push -u origin my_branch
```

To push changes from your local repository to your remote fork repository after configuring tracking:

```
git push
```

Integrating Upstream Changes into your Fork

When the upstream main branch is updated, and you want to integrate those changes into your working branch my_branch by rebase (**recommended**):

```
git stash
git checkout main
git pull
git checkout my_branch
git rebase main
git stash apply
```

When the upstream main branch is updated, and you want to integrate those changes into your working branch my_branch by merge (**not** recommended):

```
git stash
git checkout main
git pull
git checkout my_branch
git merge main
git stash apply
```

BFBB Overview

Heavy Iron Studios

EvilEngine

.HIP/.HOP Files

BFBB Decompilation Project Overview

Why Decompile BFBB?

Why Choose the GameCube Version?

How is this Possible?

Legality

Repository Layout

DTK

Repository Layout

Source Code

Third party libraries

/gc/ - GameCube specific code

/x/ - Core game engine code

/Game/ - BFBB game code

Tools

Objdiff

What is Objdiff?

Getting Started

Ghidra

What is Ghidra?

Getting Started

Ghidra-isms

concat44

VS Code

Code Formatting

Assembly Concepts

Sections

.comm

.bss

.ctors

.data

.rodata

.sbss

.sbss2

.sdata

.sdata2

.text

C++ Low Level Concepts

Mangled Names

Virtual Tables

Heavy Iron C++ Code Standards & Patterns

Primitive Type Aliases

Multi-character Literals, or ‘NTR0’

Compiler Flags

Optimization Level

Decompilation Guides

Welcome to the exciting part of the book! We finally get to get our hands dirty and start decompiling some code. This chapter is going to walk you through the process of decompiling 6 functions. They are going to start off as easy as possible and slowly increase in complexity. Each one will teach you something new about how the MetroWerks PowerPC compiler translates C++ language features into assembly language.

1. The Simplest Function: Nothing

We’re going to start off by decompiling the simplest type of function in the game: A function that does nothing.

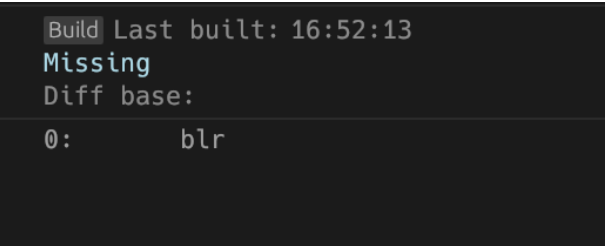
An empty function is a function that has no logic. It literally doesn’t do anything. In C++ a function that does nothing could look like this:

```
void DoNothing()  
{  
}
```

```
void DoNothing()  
{  
    return;  
}
```

Note that we can choose whether or not to write the `return;` keyword. It doesn’t make a difference whether we write it out or not.

Since the function is void, nothing is returned, so both of these functions are compiled down to a single `blr` assembly instruction:



```
Build Last built: 16:52:13  
Missing  
Diff base:  
0:      blr
```

What does the assembly instruction `blr` mean?

blr “Branch to Link Register”. Jumps to `lr`. Used to end a subroutine.

What is a Link Register?

A link register (LR for short) is a register which holds the address to return to when a subroutine call completes. This is more efficient than the more traditional scheme of storing return addresses on a call stack, sometimes called a machine stack. The link register does not require the writes and reads of the

memory containing the stack which can save a considerable percentage of execution time with repeated calls of small subroutines. The IBM POWER architecture, and its PowerPC and Power ISA successors, have a special-purpose link register, into which subroutine call instructions put the return address.²

So the `lr` register is special and holds the return address for the CPU to jump back to when the current subroutine finishes. This means that every function in the game is going to end with a `blr` instruction.

This example function `DoNothing` is only made up of one instruction. Since every PowerPC assembly instruction is 4 bytes in size, The total size of this function is 4 since it is just one `blr` instruction. Size 4 functions are the smallest functions you will find.

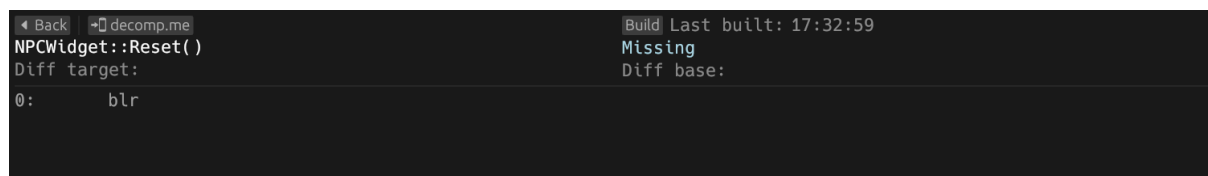
Fun fact: There are 318 functions in the game that do nothing. If we add up their size, we get a total size of $318 * 4 = 1272$ bytes. One other way of saying this is that 0.07% of the entire game's code does nothing!

Since we have a general idea of what an empty function is like, let's now decompile an empty function in BFBB.

Our First Function – `NPCWidget::Reset()`

We are going to look at the function `NPCWidget::Reset()`. It is a size 4 function located in the file `Game/zNPCSupport.cpp`.

If we open up `zNPCSupport` in `Objdiff` and click on `NPCWidget::Reset()`, we can see the original assembly on the left, and nothing on the right.



We can see that the original assembly on the left just has one instruction, a `blr` instruction like we expect. When the right side says “Missing”, this means that we have yet to define the function in our game source file. This is to be expected at this point because we haven't decompiled this function yet.

Let's go ahead and define this function in our C++ source file `zNPCSupport.cpp`:

```
void NPCWidget::Reset()
{
}
```

Great! We've defined our function. When we go back and look at `Objdiff`, it looks like there was an error compiling `zNPCSupport.cpp`:

```
### mwcceppc.exe Compiler:
#   File: src\SB\Game\zNPCSupport.cpp
# -----
#   25: void NPCWidget::Reset()
#   Error:      ^^^^^^^^^
#   undefined identifier 'NPCWidget'
#   Too many errors printed, aborting program
```

The error is pretty straight-forward. The compiler can't find any kind of definition for the type `NPCWidget`.

²https://en.wikipedia.org/wiki/Link_register

Classes and Member Functions

This function isn't *quite* as simple as our example DoNothing function we saw earlier. In this case, Reset() is actually a member function of the NPCWidget class, which is why we see the scope resolution operator, AKA double-colon (::) in NPCWidget::Reset().

We need to define the type of NPCWidget.

We can search all of the files in the BFBB repository and we can see that we haven't yet defined NPCWidget in any of our source headers. This means that we need to add it ourselves.

But how do we know what the type of NPCWidget is? To answer this question, we're going to reference the PS2 DWARF Data. If we search for NPCWidget in the PS2 DWARF data file, (defined in Section TODO) We can see the definition for the class here:

```
class NPCWidget {
    // total size: 0xC
public:
    enum en_NPC_UI_WIDGETS idxID; // offset 0x0, size 0x4
    class xBase * base_widge; // offset 0x4, size 0x4
    class zNPCCCommon * npc_ownerlock; // offset 0x8, size 0x4
};
```

We can simply copy and paste this into the accompanying header file zNPCSupport.h. After we do this, Objdiff thanks us for our effort by greeting us with a new error:

```
### mwcceppc.exe Compiler:
#   In: src\SB\Game\zNPCSupport.h
#   From: src\SB\Game\zNPCSupport.cpp
# -----
#   22: enum en_NPC_UI_WIDGETS idxID; // offset 0x0, size 0x4
#   Error:          ^^^^^
#   undefined identifier 'en_NPC_UI_WIDGETS'
#   Too many errors printed, aborting program
```

Again, the error message is pretty clear. We need to define en_NPC_UI_WIDGETS. Once again, we can't find this enum defined anywhere in our source header files yet, so it means that it hasn't been copied over yet. Let's do that now. Heading back over to the PS2 DWARF file, we can search for en_NPC_UI_WIDGETS and we find this enum:

```
enum en_NPC_UI_WIDGETS {
    NPC_WIDGE_TALK = 0,
    NPC_WIDGE_NOMORE = 1,
    NPC_WIDGE_FORCE = 2,
};
```

Once again, we want to copy this type over into our zNPCSupport.h file. We have to make sure that we put this definition before the definition of NPCWidget, as that class references the enum.

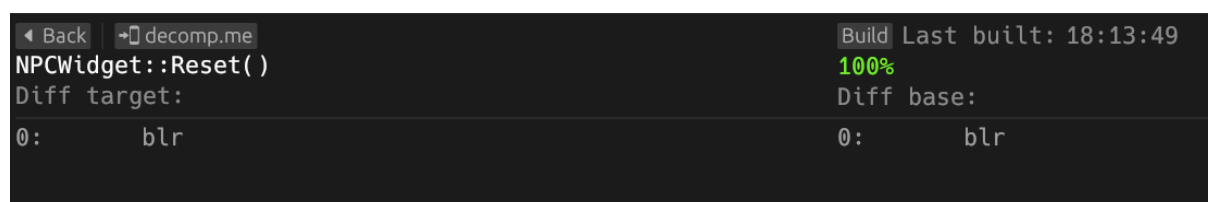
Back in Objdiff, unsurprisingly, we have another error:

```
### mwcceppc.exe Compiler:
#   File: src\SB\Game\zNPCSupport.cpp
#   -----
#   26: {
#   Error: ^
#   undefined identifier 'Reset'
#   Too many errors printed, aborting program
```

This one is simple though. It's just saying that the `Reset()` function isn't defined in `NPCWidget`. We can fix this by defining the `Reset` method with a `void` return type on our class:

```
class NPCWidget
{
    // total size: 0xC
public:
    enum en_NPC_UI_WIDGETS idxID; // offset 0x0, size 0x4
    class xBase* base_widge; // offset 0x4, size 0x4
    class zNPCCCommon* npc_ownerlock; // offset 0x8, size 0x4
    void Reset();
};
```

When we save this and look at `Objdiff` now, we finally have no more error messages. Not only that, but now the function is a 100% match!



We've done it! We have successfully decompiled our first function in *Battle For Bikini Bottom*. It's a humble little function, but we should be proud nonetheless.

Throughout this process, we have learned quite a few things:

1. How to use `Objdiff` to view and compare assembly code
2. What our first assembly instruction `blr` means and why it is generated
3. How to copy over the correct class and enum types from the PS2 DWARF data
4. How to define a new class member function

This all might have seemed like a lot of work just to decompile a function that does literally nothing.

At this point you may have a few questions, such as

1. Is it always this much trouble to simply get a function that does nothing to compile?

The answer to this is that it depends. It depends largely on the function that you're decompiling and a combination of factors. If the function is a member of a class, and that class is not defined in a header file yet, then you will have to define it like we did in this example. In this case, we also had to recursively define the properties which were included in the class we added. You can imagine that in some cases this could become a bit of work.

The same idea also applies for functions that do nothing but accept parameters whose types are not defined yet. Consider this example:

```
void xDebugAddTweak(const char*, xVec3*, const tweak_callback*, void*, U32)
{
}
```

This function accepts types such as `xVec3*`, and `tweak_callback*` which are non-primitive types. We already have these types included in the project header files, so solving the type errors would be as simple as including the appropriate headers:

```
#include "xVec3.h"
#include "zFX.h"
```


If we didn't already have those types defined in those header files, you would have to copy the types for these parameters from the PS2 DWARF again. This is needed to get the file to compile despite the fact that the function itself does nothing.

Then of course there are functions like `NPCWidget_Shutdown` which are not part of a class nor accept any parameters, so decompiling them is as simple as just writing:

```
void NPCWidget_Shutdown()
{
}
```

And that's that.

2. Why are functions that do nothing even in the game?

This is a commonly asked question for a good reason and there are multiple answers.

A common reason might be that the original game code had certain debugging code that might have been stripped out during the release build. There are at least 35 empty functions that contain the word "debug" that do nothing. `zNPCBPlankton::render_debug()` for example. You can imagine that the actual source code for this probably looked something like this:

```
void zNPCBPlankton::render_debug()
{
    #ifdef DEBUG
    // ...
    // A bunch of code to debug the plankton boss...
    // ...
    #endif
}
```

Naturally when releasing, none of that code would have been included, leaving us with empty functions. These functions are still included in the game and not optimized out by the compiler despite being empty in this case because they are called from other areas of the code. The compiler isn't smart enough to know that the function call itself wouldn't create side effects, so it keeps the function in the game despite it doing nothing in a release build.

Whew. This was a longer chapter than anticipated, but we have learned quite a lot. Whenever you're ready let's head on to the next example and start decompiling functions that have some actual code in them.

2. Returning Values

Hi mom

3. A Function with Some Logic

Hi mom

4. A Function with a For Loop

Hi mom

5. A Function with a Switch Statement

Hi mom

6. A Large Function

Hi mom

Assembly Patterns

Early Returns

Ternary Operators

Modulus

Unrolled For Loops

Switch Statements