

EarthBound Hacking 101

First Edition



Table of Contents

Introduction

I. CoilSnake

Chapter 1. Getting Started

Chapter 2. Your First Hack

Chapter 3. NPCs

Chapter 4. Playable Characters

Chapter 5. PSI Abilities

Chapter 6. Battle – Actions

Chapter 7. Battle – Enemies

Chapter 8. Battle – Backgrounds

Chapter 9. Items

Chapter 10. Stores

Chapter 11. The User Interface

Chapter 12. Music

Chapter 13. Miscellaneous

II. CCScript

III. Earthbound Music Editor

IV. PSI Animation Editor

V. Earthbound Save State Editor

Introduction

Greetings, and congratulations on your acquisition of this document! By doing so, you have not only ensured the renewal of hope for all of humanity, you have also saved an average of nine puppies from a horrible demise! On a related note, you have also taken the first step towards learning how to hack EarthBound.

This guide will hopefully instruct you in all the basic techniques required to get you up and hacking one of the greatest games in modern history. Beyond reasonable computer skills and a willingness to learn, there are no real prerequisites, so let's jump right in!

How can EarthBound be hacked? It's on a console!

EarthBound, like any other console game, can have its data extracted and manipulated on a computer with special tools that are able to grab its ROM (Read-Only Memory) from its cartridge, where all the game's programming and art is located. The ROM can then be passed through something called an emulator, which attempts to recreate an SNES environment so that you can play EarthBound on a personal computer.

The ownership of a ROM is legally-dubious at best, and distributing a ROM can frequently lead to legal action from the creators of the game. Since STARMEN.NET would rather avoid a lawsuit if possible, it does not distribute the EarthBound ROM, so you're on your own to find one (and in any case, a quick Google search should be enough to find one within a minute).

Once you have your ROM, you'll need to get an emulator. ZSNES is highly recommended, as it is cross-platform, possesses a plethora of features, and is easy to use. Just open your ROM with ZSNES and voilà – you can now play EarthBound on your computer.

Now that you have the game data on your computer, you can start modifying the game. To do so, you can either manually tweak the hexadecimal values (an explanation is beyond the scope of this manual, but there are many useful resources on the Internet) contained within the ROM (which is useful if your objective is to discover new features), or you can use some of the many tools that have been written over the years to automate the process considerably (particularly useful if you're less of a technical person and more of an artist-type). This manual will focus on some of these tools (keep reading...).



A screenshot of ZSNES' menu screen.

Well, okay, so what about Mother? And Mother 3?

For varying reasons, less efforts have been invested into hacking the games that preceded and followed EarthBound, namely Mother/EarthBound 0 and Mother 3. In the case of the former, this is mostly because it is much less interesting to work with the original Mother engine than with the EarthBound engine, since the latter offers a greater degree of customization. In the case of Mother 3, the architecture of its ROM makes it difficult to hack, so few efforts have been expanded toward doing so.

There are a few tools, however; check them out at [STARMEN.NET's PK Hack section](#), under "*Editing Tools for Mother 1 and Mother 3*". These tools will not be covered in this manual.

How is this manual organized?

This document has been split into several parts, each dealing with a specific tool used to hack EarthBound. These are the tools which will be covered, in order:

- **[CoilSnake](#)**: CoilSnake is a replacement for PK Hack (the original hacking software, also known as JHack), whose propensity towards ROM corruption was a severe limitation for hacking. Whereas PK Hack edited the ROM directly with custom editors, CoilSnake extracts the required data so that it can be edited with external programs. It can be used in conjunction with CCScript (see below). CoilSnake has both a CLI (Command-Line Interface) and a GUI (Graphical User Interface, written in Tkinter); this manual will cover both for each technique where possible. However, if you're just starting out, you might be more comfortable using the GUI.
- **[CCScript](#)**: the ultimate dialogue editor, CCScript is used to edit characters' and objects' dialogue with the player throughout EarthBound. It possesses a simple, human-readable and easy-to-use syntax for creating complex dialogs, with much more than just text. CoilSnake is able to bind these dialogues to the exported ROM automatically. While CCScript files can be created using Visual CCScript, this is not recommended as CoilSnake can handle CCScript compilation itself, when tied to an appropriate CCScript compiler.
- **[EbProjEdit](#)**: the best map editor beginning with Eb and ending with Edit, use EbProjEdit to graphically edit EarthBound map files generated with CoilSnake.
- **[EarthBound Music Editor](#)**: not only does this tool allow you to edit music, it also allows you to play back songs! Recommended for musicians bound to Earth.
- **[PSI Animation Editor](#)**: a really useful tool if you're looking for a way to animate new PSI effects in EarthBound.
- **[EarthBound Save State Editor](#)**: a useful tool when debugging to quickly skip ahead, this tool allows you to jump to another point in the game by modifying your save data.

What do I need to start hacking?

Operating system: CoilSnake and CCScript are cross-platform programs, and will work on Windows, Mac and Linux. EB Hack and EbProjEdit will run on any system with Java installed. The Earthbound

Music Editor, PSI Animation Editor and EarthBound Save State Editor are Windows-only programs, but using the Mono library on Wine, they could potentially be made to run on Linux and Mac.

Text editor: Apart from an appropriate operating system, you'll need various standard utilities to edit game files: a text editor to edit `.yaml` files, such as NotePad++ (Windows), TextMate (Mac), or Gedit (Linux). A word processor (such as Microsoft Word) is unlikely to work.

Image editor: Photoshop is the industry standard for image editing, and with good reason, but remains an extremely expensive program. If you can't afford it, there are many other programs available, such as [GraphicsGale](#) (free). *Warning:* while GIMP is an extremely popular open-source alternative to Photoshop, there is currently a bug in recent versions when saving indexed PNGs; it is therefore not recommended to use GIMP.

Dependencies: Individual programs might have third-party dependencies, or might even depend upon each other to work. Each tool's documentation will outline what is needed.

One human brain: Preferably not in a jar.

How complete is this manual?

Not completely complete. This First Edition deals only with CoilSnake and skims over some parts, but later editions will try to cover all the other important tools, while also providing appendices of reference for control codes, various values, and more!

But that is for another day.

I. CoilSnake



CoilSnake is perhaps the single most important tool in your hacking arsenal, as it allows you to decompile ROM files to an organized directory which will contain (almost) everything from the game, including PSI power lists, enemy statistics, artwork and more. Its purpose is to eventually supersede PK Hack/JHack through the progressive addition of new features.

The main advantage of CoilSnake over PKHack is that it does not directly modify the ROM file: this means that the chance that the ROM will get corrupted is a lot less significant. It also gives you more freedom over your hacking, letting you choose your own image editors, text editors, and so on, while preserving the human-readable [YAML](#) format for its text data files.

CoilSnake is written in Python and is therefore cross-platform. It requires **Python 2.7**, Python's **YAML extension**, and the **Python Image Library (PIL)**. If you wish to use its Graphical User Interface, you'll also need to have Python's **Tkinter extension**.

Additionally, if you wish to make use of CCScript within your hacking projects (which you probably should if you intend to write some dialogue), you'll have to obtain a CCScript compiler binary and specify its location within CoilSnake's preferences (see below for more instructions). For instructions on compiling CCScript for your system, see the appropriate section.

Note for non-Windows users: CoilSnake uses a pre-built binary blob for the `NativeComp` Python module; however, this binary will not work on other systems, so you'll have to compile your own. Scared? Don't be! `build_NativeComp.py` is a user-friendly Python script that will build the module for you. On *nix systems (like OSX or Linux), simply enter the following commands into the terminal (tested on Debian Testing):

```
cd path/to/coilsnake/modules/eb/  
python build_NativeComp.py build  
cp build/lib.*/NativeComp.so NativeComp.so
```

Once your CoilSnake installation is ready, you're ready to get started!

Official Thread: [CoilSnake v1.2: Based on a True Story](#)

Download Link (current version: 1.2): [CoilSnake_1.2.zip](#)

Source: <https://github.com/kij/CoilSnake>

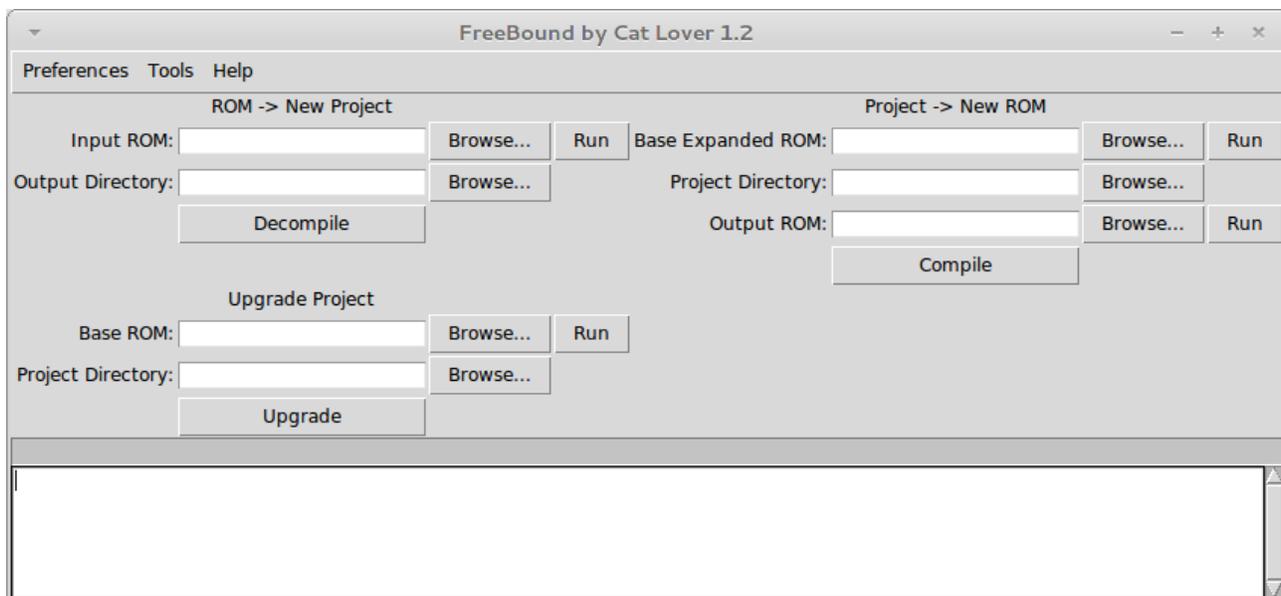
Chapter 1.

Getting Started

At a first glance, CoilSnake can seem pretty bare, but this bellies its powerful nature. For example, while the GUI doesn't seem to provide that many options, this is by design. Its true strength becomes apparent once you run the software a first time. Let's take a look at the GUI first, before moving on to the command-line options. In the next chapter, we'll actually make it do something.

1. CoilSnake's Graphical User Interface

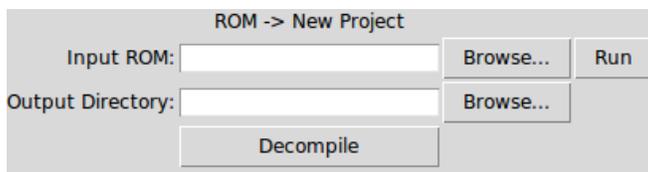
To start up the visual version of CoilSnake, run `CoilSnakeGUI.py` (either from the terminal or by double-clicking on it, depending on your system setup).



CoilSnake's GUI. Isn't Tkinter a beauty?

Even though it's not much to look at, it gets the job done, and it does it well. And while much of the interface appears self-explanatory, it never hurts to give more explanations.

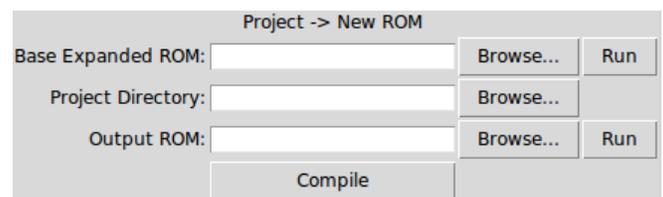
The main screen is the center of operations, and each section has a clearly different purpose.



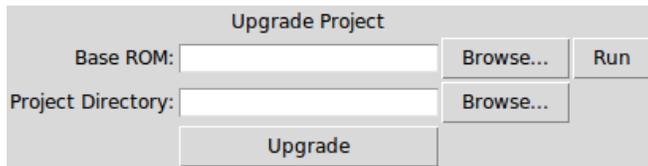
The source ROM, the directory where CoilSnake will place its files, and you hit Decompile. The progress of the operation will appear in the white textbox below.

The section to the right, **Project -> New ROM**, does the exact opposite: it takes a modified project directory (such as one exported using the previous section) and

The **ROM -> New Project** section is used to take an existing, vanilla ROM of Earthbound (modified/hacked ROMs will probably not work as well) and to decompile it to a directory on your hard drive. You specify the



compiles it into a new ROM, which you can then play to enjoy your hacks and modifications. You'll need to select a base ROM for this task – one which you have already expanded by using CoilSnake (see later on). Hit Compile and let CoilSnake do its magic – it shouldn't take too long.



Finally, the last section, **Upgrade Project**, is used to convert a CoilSnake project made with a previous version of CoilSnake (at the time of this writing, this would be versions 1.0 and 1.1 of CoilSnake) to the

latest format, while preserving existing data. You need to provide a Base ROM to serve as a reference, and the Project Directory to be upgraded by CoilSnake.

But wait! What if your project contains CCScript files? How will CoilSnake know how to compile your files? By not doing it, of course! Instead, it delegates the task to the CCScript compiler binary. To specify its location, navigate to **Preferences** → **CCScript Compiler Executable**, and open the file with the dialog box.

Similarly, if you'd like being able to test your ROM directly from CoilSnake, you can also set the emulator program to be used by an analogous method, using **Preferences** → **Emulator Executable**.

Additionally, if you like seeing more error messages, you can toggle the setting by navigating to **Preferences** → **Toggle Error Details** (by default, this is set to a lower setting).

An important step in the hacking of an EarthBound ROM is to expand its size to allow for content additions. This can be achieved through the GUI by using the menu entry **Tools** → **Expand ROM to 32MBit** (if you don't plan on a big hack) or **Tools** → **Expand ROM to 48MBit** (if your hack needs the extra space), and selecting the ROM to expand.

Additionally, in the **Tools** menu, you can choose to add or remove an SNES cartridge header to your ROM, although you probably won't need to do this in the course of your hacking.

This is about it for the GUI – there's not much more you can do here. So let's move on to the CLI.

2. CoilSnake's Command-Line Interface

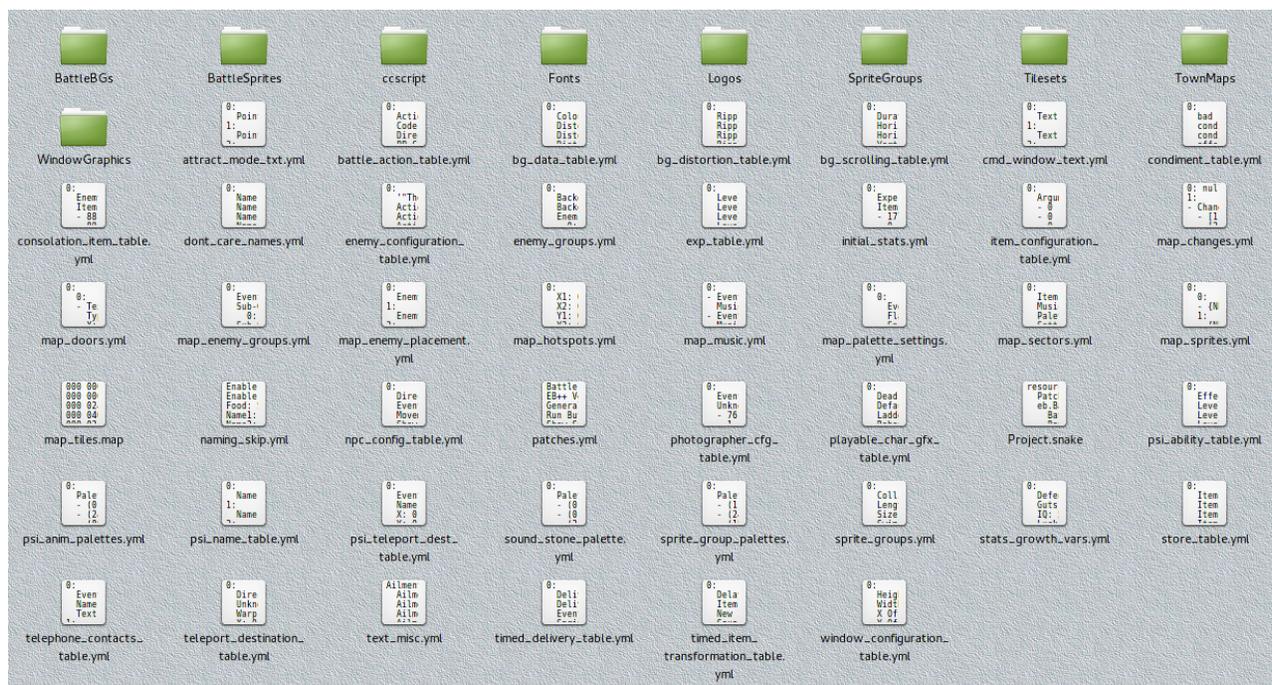
CoilSnake's command-line functionality is pretty much identical to the GUI's functionality, albeit with a few missing features (the ability to expand your ROM *is* available, but it requires using `tools/EbRomExpander.py` instead of `CoilSnake.py`; it will be covered in a later edition). You must start it either directly from the console (`./CoilSnake.py` on *nix operating systems) or by double-clicking it to open up the terminal/console (note: this might immediately close the console, depending on your OS and setup; an alternative is to open it with IDLE, the Python IDE). There are three different operations that can be performed (the syntax of the commands is for *nix systems, but it is usually analogous on other systems):

- `./CoilSnake.py -c ProjectDirectory BaseROM OutputROM [-ccc CCC]:` this command achieves the same result as the GUI's Compile option. The three arguments are paths to the locations of your project directory, base expanded ROM and output ROM, respectively. Remember to surround these paths with `"..."` if the paths contain spaces! There is a fourth, optional argument, which you can use if your project contains CCScript files that need to be compiled; appending it and providing the path argument will tell CoilSnake where it can find the CCScript compiler binary.
- `./CoilSnake.py -d ROM ProjectDirectory:` decompiles the ROM at the specified location to the specified project directory.
- `./CoilSnake.py -u BaseROM ProjectDirectory:` finally, this is the equivalent of the Upgrade option in the GUI, and takes the path of the base ROM and the folder to output to.

Now that you know how CoilSnake's interfaces work, let's look at the layout of the project directory.

3. The Project Directory

The typical appearance of a ROM decompiled with EarthBound is a great amount of folders and files, the latter being usually either in the `.yaml` or `.png` format.



The standard folder layout (on Linux Mint).

It may appear a bit overwhelming at first through the sheer amount of configurability available to you, but each file is logically named and located to make things easier to manage.

We will go into more detail on the subject of each individual file in the later chapters, but let's take a general look at them first.

- **The Project.snake file:** this is what makes this folder more than a collection of pretty images and gibberish: it provides CoilSnake with information on the location of every resource file there is, as

well as some metadata like the ROM type (EarthBound) and the CoilSnake version (1.0 → 1, 1.1 → 2, 1.2 → 3, and so on). You should never have to edit this file manually.

- **The .yaml files:** these are among the most numerous files, and for good reason. They contain the actual settings of the game, such as “Don’t Care” name options (when choosing names for your characters in EarthBound) in the `dont_care_names.yaml` file. YAML possesses a very self-explanatory syntax, but if you ever encounter something that confuses you, there is a wealth of online documentation available. You’ll also notice a great number of address code, either as `0x82ab` or `$ef8543` (for example): these are actually called *pointers* to data, indicating EarthBound where to look for the specified resource.
- **The BattleBG directory:** contains all the different types of battle backgrounds which should be stretched, deformed, etc. during battles.
- **The BattleSprites directory:** a list of enemy sprites used during battles.
- **The ccscript directory:** a place to put all your CCScript dialogue so that it can be compiled.
- **The Fonts directory:** the fonts used by EarthBound (including Saturnian, zoom!).
- **The Logos directory:** the logos displayed at the beginning of the game for the creators of EarthBound.
- **The SpriteGroups directory:** the sprites used to animate the characters on the world map (NPCs, player characters, enemies, etc.).
- **The Tilesets directory and map_tiles.map:** the files in Tilesets contain the tiles used for the maps and should not be manually edited. Likewise, `map_tiles.map` arranges those tiles into the maps used for the locations in EarthBound.
- **The TownMaps directory:** the small maps of towns displayed to the player.
- **The WindowGraphics directory:** the various GUI elements used to display the HUD, with the different flavors available to the player.

To modify the game, simply modify existing files to your liking. For now, however, there is no way to add new files (except for CCScript files, which must all be placed in the `ccscript` directory, where they will be automatically detected).

Conclusion

Now that you know how CoilSnake works and is organized, you’re (un)officially ready to write your first hack! Keep reading...

Chapter 2.

Your First Hack

The standard program most programmers write when learning a new language is Hello World, which aims to output "Hello World!" in some fashion to the user. Sounds exciting, right?

It isn't. It's boring.

So let's do a twist on that that's a bit more involved, but is hopefully a lot more rewarding.

1. Planning the Hack

A hack without a plan is like a train without a steering wheel: it works just fine. However, you might want some levers and other controls on that train, because unless derailing trains is your weekend hobby, you're not likely to enjoy the trip in the long-run, once you realize you don't know where you're going. So let's think a bit before making this hack.

Additional note: There a few good tips here on [an old tutorial for JHack](#) which might be of interest to you once you start fleshing out your own hacks; check it out!

EarthBound is a game, so it would make sense if you saw "Hello World" said by a NPC instead of just printed on-screen. So we'll need to make a character say this at some point in the game. Oh, but that's pretty easy – you just need to substitute some dialog at one point. And it's boring besides. Maybe the character could be in a place where he's not supposed to be? Say (spoiler alert!) Robot Ness in Ness' house at the beginning of the game? Yeah, that sounds a bit more interesting (or not; everyone's entitled to his opinion). Let's work with that.

So, what would we need to do? Well, the sprite already exists, so there's no need to create a new one. The map would have to be edited, though, so that Robot Ness can be placed somewhere in Ness' bedroom. But wait, since we can't create new NPCs (yup; that's a limitation currently), we'll have to use an existing NPC and modify it for our purposes. Maybe a more-or-less useless one could be used, like a present box (yes, a NPC CAN be an object!). And some dialogue should be tied to him; CCScript should get the job done. Alright, so we need:

- 1) To replace a mostly useless NPC (such as a present) with our custom one.
- 2) To add Robot Ness to Ness' bedroom using EbProjEdit.
- 3) To tie some dialogue to him with CCScript.

Doesn't sound that hard, right? So let's hop to it!

2. Replacing a NPC

All NPCs are configured in `npc_config_table.yml`, and are identified by their ID. Let's take at

look at 744:

```
744:
Direction: down
Event Flag: 0x274
Movement: 708
Show Sprite: when event flag set
Sprite: 195
Text Pointer 1: $c7db3f
Text Pointer 2: $0
Type: object
```

We need to change a few things about this (don't worry about the exact details, they will be covered later):

```
744:
Direction: down
Event Flag: 0x0
Movement: 605
Show Sprite: always
Sprite: 5
Text Pointer 1: robot.hello_world
Text Pointer 2: $0
Type: person
```

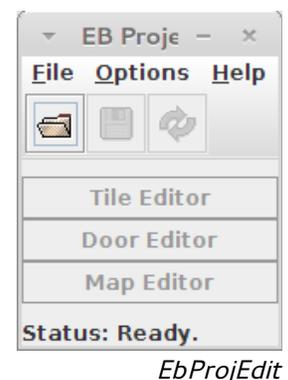
A few explanations are in order: since this sprite will always appear, we don't need to set an event flag to it (thus, `0x0` as an event flag and `Show Sprite: always`). Setting `605` as `Movement` will make the robot static until interacted with. `Sprite: 5` sets the sprite group #5 (Robot Ness) as the sprite animation group for Robot Ness. `Type: person` ensures that the correct interaction option is "Talk to". Finally, `Text Pointer 1: robot.hello_world` specifies that the text is located at the `hello_world` location in memory, which is a label we will define later on (in the `robot.ccs` file, thus the `robot` prefix) when writing CCScript.

Now that we have our NPC ready, let's place him in the appropriate location.

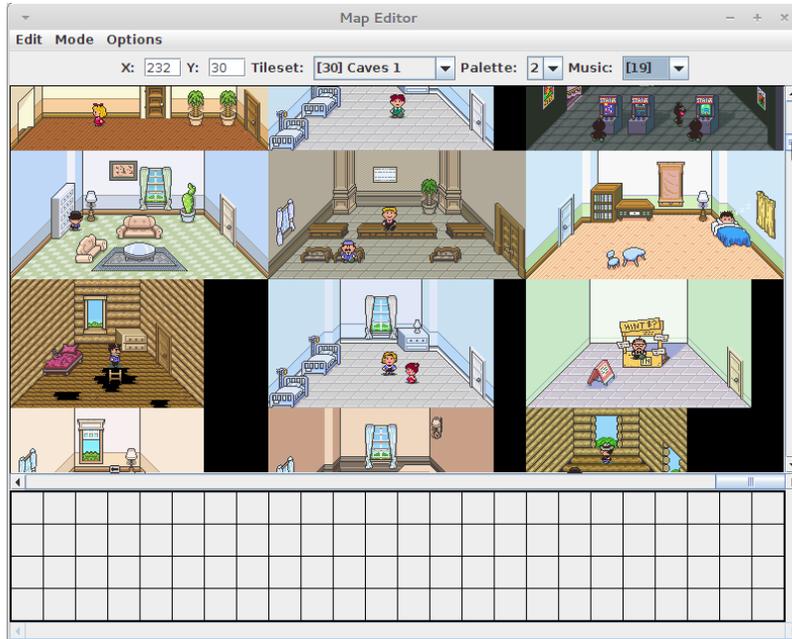
3. Editing the Map

You'll need EbProjEdit for this step, although for the purpose of this first hack, you won't need to be fully knowledgeable with it yet (read the tutorial on it later on if you lack confidence on your ability to follow step-by-step directions). Just download the `.jar` Java program specified and run it.

When you first open it, you should see a tiny, unassuming initial screen, pictured to the right. You can't really do anything yet with it, so click on the folder icon to open your `Project.snake` file. It will take a while to load all of its data, but it



will get there eventually; you'll know when it's ready when all three buttons light up. We don't want to edit existing tilesets or handle door mechanisms, so open up the Map Editor.



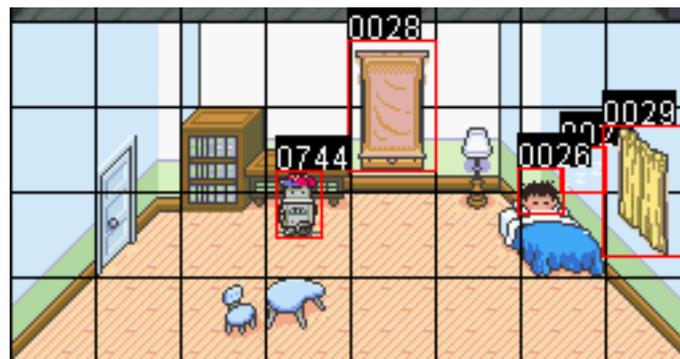
A new screen will pop up with a default map loaded. Since we want to edit Ness' bedroom at the beginning of the game, scroll over the map until you find it – it's somewhere around the top-right corner.

So now that that's done, you want to bring up the Sprite Edit mode – do so by hitting F2.

Then, right-click someplace within Ness' bedroom, and select "New NPC". By default, this is Ness, so let's select a different sprite by right-clicking on it and choosing "Switch NPC (0)". Type 744 into the box that pops up, 744 being the ID of the NPC we have replaced.

The Map Editor with Ness' bedroom visible.

Ness' bedroom should now look something like this:



Ness' Bedroom - now with 100% more robot!

4. Scripting Some Dialogue

For this step, you'll need a working CCScript compiler; obtaining one is described in the CCScript section. Then, you'll need to specify its location in CoilSnake's preferences before compiling. However, you don't need to know anything else about CCScript for this step.

CCScript files must all be placed in the `ccscript` of your CoilSnake project. Go ahead and create an empty `robot.ccs` file with a text editor of your choice, and write this text into it:

```
hello_world:
    "@???: Hello World!" end
```

This will effectively make our NPC (here called "???") say "Hello World!" to Ness, then stop the conversation. Neat, isn't it?

5. Compiling and Running

Looks like we're good to go. Load up CoilSnake, give him the location of your Project Directory and a path to a ROM file, either one that you don't mind getting overwritten or one to be created, and hit Compile. Then, open up your favorite emulator, and try out your work!



Clash of the titans.

And there you have it. Your first hack. Isn't he cute?

Of course, this is just the beginning, and there's a lot to learn yet...

Conclusion

You got to build your first hack for EarthBound and you got familiar with some of the tools you'll be using, as well as getting a glimpse at some of the techniques you'll be using time and again. From now on, you can read this manual in almost any order, since most chapters aren't dependent on others. You'll start learning all sorts of techniques one by one, until you're ready to make the Next Big Hack.

Is your heart beating incredibly fast?

Chapter 3.

NPCs

Non-Player Characters are a staple of many (if not all) RPGs, and EarthBound is obviously no exception. Whether they are as simple as a present box or as complex as Pokey Minch, almost all NPCs are handled in the same fashion by CoilSnake. You got a brief glimpse at how they work in the previous chapter, so let's go into more detail in here.

Before we begin, you have to remember (and this applies in almost every case) **that there is no way to add new objects directly**: each time you want to add a new object, be it a phone, a character, or anything else, you have to edit an existing entry and overwrite it with your new data. So before you start creating new things, make sure your hack can do away with at least one object which the player won't need to enjoy your hack.

List of files used:

- SpriteGroups/
- npc_config_table.yml
- sprite_groups.yml

1. NPC Sprites

All the NPC sprites CoilSnake can extract from EarthBound are located within the `SpriteGroups` directory.

Each *sprite group* is composed of 16 "slots" for sprites, some of them being unused (since not all NPCs use all of their possible animations). Each individual sprite represents a possible position for the NPC, such as sideways, forward-facing, and so on. Sprite groups are identified by their filename, when they are used in other files produced by CoilSnake; for example, `079.png` would be identified by `79`.

The sprite groups images can be modified using one of the programs suggested in the introduction of this manual.

Sprite groups also need to be configured so that the game engine knows how to animate characters correctly. Their properties are all defined in `sprite_groups.yml`, using a rather self-explanatory format (as you will soon find out, a lot of CoilSnake's project files are easy to understand):

```
1:
Collision Settings: [8, 8, 8, 8]
Length: 16
Size: 16x24
Swim Flags: [false, false, false, false, false, false, false, false,
false, false, false, false, false, false, false]
```

This is the definition for the group sprite 1 (Ness), as is indicated by the `1:` at the top. Let's take a

look at each individual property.

`Collision Settings` defines how EarthBound should handle collisions with this sprite group. Collisions are counted starting from the bottom center of the sprite. The first and third numbers define the width of the sprite on each side (so 8 and 8 means that the sprite is 16 pixels wide). Likewise, the second and fourth numbers indicate the height of the sprite (when it comes to collisions that is).

`Length` sets the number of available sprites within that sprite group (16 for 16 sprites... you get the idea).

`Size` specifies the pixel size of an individual sprite.

`Swim Flags` sets whether or not a sprite should be affected when entering a swim region (by partially submerging the sprite), by specifying a boolean (`true/false`) for each sprite, in left-to-right order.

However, configuration of sprite groups is obviously insufficient to place new NPCs, so additional steps are required.

2. NPC Configuration

The configuration associated with NPCs can be found in the `npc_config_table.yml` file. Each set defines a specific NPC (such as a Tenda bystander or an aggressive Shark), with appropriate behavior for each. Let's take a look at a definition for one of the non-aggressive bag-wielding ladies, of ID 82:

```
82:  
  Direction: right  
  Event Flag: 0x0  
  Movement: 12  
  Show Sprite: always  
  Sprite: 57  
  Text Pointer 1: $c72822  
  Text Pointer 2: $0  
  Type: person
```

Let's once more look at these properties one by one.

`Direction` indicates the initial direction the NPC should be facing (and thus which sprite to use initially). This can be `right`, `left`, `down` and `up`. This is mostly useful if your NPC will be static, as other types of movements (see below) will make it change its direction as time goes. Item boxes should always start with the `down` direction, to indicate their closed status.

`Event Flag` identifies the event to be used to trigger the appearance of this NPC, if necessary (example: `0x1a6`); if an event flag is not needed (for example, if the sprite must always be displayed), this should be the null flag: `0x0`. If the NPC is an `item` type, this flag indicates whether or not the item has been taken from the NPC.

`Movement` identifies the type of movement to be applied to the NPC; there are hundreds of options to choose from. In this case, `57` makes the lady walk very quickly to the bottom left, passing through

anything in her way, once the player gets close enough.

`Show Sprite` sets the condition for the NPC's visibility; in this case, since there is no `Event Flag`, the sprite should always be shown. Other options, to be used when an event flag is specified, are `when event flag set` and `when event flag unset`. For `item` NPCs, this should always be set to `always`.

`Sprite` specifies the sprite group to be used (as explained in the previous section).

`Text Pointer 1` takes a pointer to some dialogue to be displayed when the user interacts (by examining it or talking) with the NPC. This can either be hex code, or it could be a `CCScript` label.

`Text Pointer 2` takes a pointer to some dialogue to be displayed when an item is used on the NPC. If it is an `item`, this should point to the item to be obtained; `$100` will make it an empty box, any setting higher than that provides money to the player (for example, `$10A` gives the player 10\$).

`Type` specifies whether the NPC is a `person`, an `object` (such as an ATM machine) or `item` (such as a present box).

Notice: while it may seem like a NPC definition lends itself to re-usability, you should never have several instances of a NPC on your map, as this might cause instability.

Conclusion

Once your NPCs have been set and configured, you can use them from `EbProjEdit` to place them in the appropriate location; you can also assign them some dialogue using `CCScript` (see the `CCScript` tutorial).

Note however that NPCs do not include enemy characters (these are handled differently, as we will see in the following chapters), so make sure you don't assign them movements which will provoke an attack, as a general rule (there might be exceptions, of course, depending on your hack).

Chapter 4.

Playable Characters

What is an RPG without a character to control?

Probably either a very terrible or a very innovative game. But we digress.

Playable characters are handled differently from NPCs, and have different, more fleshed-out properties assigned to them. Customizing them can produce highly interesting results. However, like NPCs, their sprite groups are located in `SpriteGroups`.

List of files used:

- `dont_care_names.yml`
- `exp_table.yml`
- `initial_stats.yml`
- `naming_skip.yml`
- `playable_char_gfx_table.yml`
- `stats_growth_vars.yml`

1. Names

This is a minor feature, but it might interest you if you are doing a complete overhaul of EarthBound, for example: you can change the default names proposed to you (by the “Don’t Care” option) in the `dont_care_names.yml` file, where 0 is Ness, 1 is Paula, 2 is Jeff, and so on. A total of seven suggested names can be specified in total.

You can enforce names upon the user, if you so choose, through the use of the `naming_skip.yml` file. Setting `Enable Skip` to true skips directly to the game once a new game is started, using the names specified below; optionally, you can set `Enable Summary` to true if you want to show the player the names that are going to be set (if the user clicks “Nope” on the summary screen, he’ll just be taken back to the summary screen once more).

2. Leveling Up

There are four distinct files which control the way characters level up.

`exp_table.yml` sets the required amounts of experience required for each character to attain new levels – setting them all to 1, 2, 3, etc., for example, will make your characters level up quite quickly!

`stats_growth_vars.yml` sets the growth rates for each character when a new level is gained; these are used by the mathematical equations the engine uses to compute the new levels gained in each stat, as outlined in [this guide](#).

`initial_stats.yml` specifies the default state each character is in when he first joins the party. Each label is clearly-named (`Experience Points`, `Items Possessed`, `Level`, `Money`), with a fifth

property of unknown purpose (appropriately enough, labeled Unknown). The numbers specified in `Items Possessed` are defined in `item_configuration_table.yml`.

Finally, `playable_char_gfx_table.yml` is in various ways analogous in purpose to `npc_config_table.yml` in that it ties the characters with their sprites. `Dead Sprite Group` specifies the sprite group to use when one of the characters dies, and so on with all the other properties (once again, there is an Unknown property).

Conclusion

With the ability to completely customize the main playable characters, you can potentially completely remake the game, each character coming from a different place, having a different personality and even a different appearance! It's all up to you.

Chapter 5.

PSI Abilities

What would our heroes do without PSI abilities? Ness, Paula, Jeff, Poo... each has his own distinct set of PSI powers. And wouldn't you know it, CoilSnake lets you customize them... to some extent. For one thing, there is, as of this writing, no support for PSI animations (although such a feature is planned).

List of files used:

- `psi_ability_table.yml`
- `psi_anim_palettes.yml`
- `psi_name_table.yml`
- `psi_teleport_dest_table.yml`

1. Configuration

All PSI abilities are declared in `psi_ability_table.yml`, using the same standard format; let's look at the declaration for "Healing γ ", ID 29:

```
29:  
Effect: 38  
Level learned by Ness: 53  
Level learned by Paula: 0  
Level learned by Poo: 36  
PSI Menu position (X): 13  
PSI Menu position (Y): 1  
PSI Name: 8  
Strength: gamma  
Text Address: $ef5239  
Type: 2  
Usability Outside of Battle: usable
```

`Effect` specifies the action the PSI power should have. To modify the nature of this action, edit the associated entry in `battle_action_table.yml` (which we will cover in a later chapter).

`Level learned by ...` indicates the lowest level at which this power is attained.

`PSI Menu position (Y)` is the row on which the ability's name is written (going from 0 to 2), while `PSI Menu position (X)` is the horizontal location of the symbol for that level of power (α , β , γ , Ω , Σ).

`PSI Name` takes the ID of one of the names defined in `psi_name_table.yml`, such as ID 7 for "Lifeup " (BUG: there is currently a discrepancy between the ID specified in `psi_name_table.yml`, say 7, and the ID in `psi_ability_table.yml`, which would be 6; remember to add one to the ID

when editing a PSI ability).

Strength indicates the level of the power (alpha, beta, gamma, omega, sigma, or none (which you should never use)).

Text Address is either a pointer or a CCScript label to some text used to describe the PSI ability.

Type can be either 1 (Offense), 2 (Recover), 4 (Assist) or 8 (other), which is used to classify powers in the ability menus.

Usability Outside of Battle specifies whether or not the power can be used both in battle and out of battle (usable), usable only in battle (unusable), or a special power such as Teleport (other).

Notice: never edit the first or last entries in `psi_ability_table.yml`.

2. Teleportation Abilities

In the case of teleportation powers, additional configuration is required to specify the possible destinations; this is handled through `psi_teleport_dest_table.yml`. The syntax is simple as usual; let's look at the entry for Twoson for example:

```
2:  
  Event Flag: 0xd2  
  Name: Twoson  
  X: 176  
  Y: 820
```

Name is the displayed label for the location in the ability menu.

X and Y are the coordinates on the map in the 8x8 format; that is, they specify a 8x8 pixel-sized tile somewhere on the world map (as can be seen in the Map Editor under Options → Show Coordinates → Warp X, Y).

Event Flag specifies the flag required to be set in order for this destination to be displayed in the menu.

3. Animation

CoilSnake does not currently support PSI animation, despite the presence of the `psi_anim_palettes.yml` file.

Conclusion

You can now configure PSI abilities for each character, but how can you actually set the action it will have on your party or its enemies? Read the next chapter for information on... battle actions!

Chapter 6.

Battle – Actions

Like most RPGs, battles are an integral part of EarthBound, and what better way to fight than with yo-yos, baseball bats, and PSI Rockin? Of course, the properties of these actions have to be defined somewhere, and that's what we'll look at in this chapter.

List of files used:

- `battle_action_table.yml`

1. Configuring Battle Actions

The `battle_action_table.yml` file describes every single possible action that can be taken during a battle by either side, be it using a PSI ability, an item, or any of the miscellaneous abilities available. Let's take a look at the description for PSI Rockin:

```
12:  
Action type: psi  
Code Address: $c29568  
Direction: enemy  
PP Cost: 40  
Target: all  
Text Address: $ef8543
```

`Action type` defines whether it is a `psi` ability, an `item's` action, a physical attack (which can be a physical (affected by shields and defending) `attack` or a physical (unaffected by shields and defending) `attack`), an other action (such as Paula's Pray), or `nothing` (which you should never use, but exists nonetheless).

`Code Address` indicates the location of the assembly code to be executed when this action is triggered, whether through a pointer or a CCScript label (read the next section).

`Direction` indicates whether the action should happen to the `enemy` or to the `party`; the actual group referenced by this varies depending on who is using the action, whether it is one of the player characters or an enemy.

`PP Cost` is the amount of PSI Power which will be subtracted from the ability-user's statistics.

`Target` can either be `all` (to indicate all enemies or all party members), `one` (to indicate a specific character), `row` (to indicate a row of enemies), `random` (to randomly select an enemy) or `none`, for miscellaneous battle actions.

`Text Address` takes either a pointer or a CCScript label to the text to use as the action's text once it has been used.

2. Modifying Actions' Effects

Now while editing PSI Power costs and targets can sound like the most exciting past-time since eating sliced bread was invented, it is a lot more rewarding to actually create completely new actions that change the purpose of PSI Abilities – whether this is to heal more, to make enemies instantly die, or whatever other crazy effect your brain can come up with!

Lamentably, while CoilSnake makes a lot of things easy, it still requires the use of ASM (assembly code) to modify the effects of battle actions; SNES programming is beyond the scope of this manual, but if you feel daring enough to learn this dark and ancient voodoo, you might be interested to know that you can write this code from within CCScript (see the CCScript tutorial for information on this), then reference it from within `battle_action_table.yml`.

Conclusion

With the power to change battle actions, you can find new and innovative ways to let your characters annihilate Giygas' minions (or even Giygas himself, with the mighty Smelly Sock of Doom!). The next step is to define these battle enemies...

Chapter 7.

Battle – Enemies

EarthBound wouldn't be the same if it was deprived of its all of its unique enemies; fortunately, CoilSnake gives you a great degree of control over each and every one of them. From crows to Giygas, you can modify them to your heart's content... as long as Giygas is... H... A... P... P... Y...

List of files used:

- BattleSprites/
- enemy_configuration_table.yml
- enemy_groups.yml
- map_enemy_groups.yml

I. Battle Sprites

Anyone playing EarthBound must have noticed that the enemies you see in battle sometimes look nothing like what they look like on the world map. That's because they're drawn using completely different sprites, which can be found in the `BattleSprites/` directory, where they are classified by the number which identifies them. You can edit these sprites using one of the programs suggested in the introduction of this manual.

II. Enemy Statistics

It stands to reason that enemies should have some of the most detailed property blocks of any object in EarthBound, and CoilSnake being very reasonable, there is much to configure with enemies in `enemy_configuration_table.yml`. Each number in the file references a sprite number in the `BattleSprites/` directory. Each block is a whopping 47 lines long – that's a lot of properties, so let's look them over as always; as an example, we'll use the Armored Frog, ID 3:

```
3:
  "The" Flag': 1
Action 1: 4
Action 1 Argument: 0
Action 2: 4
Action 2 Argument: 0
Action 3: 4
Action 3 Argument: 0
Action 4: 95
Action 4 Argument: 0
Action Order: 0
Boss Flag: 0
```

```
Death Sound: normal
Death Text Pointer: $ef6d96
Defense: 108
Encounter Text Pointer: $ef78b8
Experience points: 1566
Final Action: 0
Final Action Argument: 0
Fire vulnerability: 100%
Flash vulnerability: 70%
Freeze vulnerability: 100%
Gender: neutral
Guts: 5
HP: 202
Hypnosis/Brainshock vulnerability: 50%
Initial Status: normal
Item Dropped: 7
Item Rarity: 4
Level: 22
Luck: 8
Max Call: 0
Mirror Success Rate: 50
Miss Rate: 1
Money: 77
Movement pattern: 21
Music: 98
Name: Armored Frog
Offense: 37
Overworld Sprite: 280
PP: 0
Paralysis vulnerability: 50%
Row: 1
Run Flag: 7
Speed: 7
Type: normal
Unknown: 2
```

Ye gods above! Let's examine that methodically.

' "The" Flag' sets whether the game should display a "the" before the enemy's name (1) or not (0) when appropriate.

Action 1, 2, 3, 4 indicate the action in `battle_action_table.yml` to execute; note that when an

enemy is executing this action, the `party` in the battle action entries means the enemy group, whereas `enemy` indicates the player characters. The associated `Argument` specifies an option to be passed to the action, which has a different meaning depending on the action (it could be an item number, or an enemy number if the enemy is calling for help). Optionally, `Final Action` can indicate an action to be executed by the enemy once it has been defeated, `Final Action Argument` being its argument.

`Action Order` provides the order used to run through each action:

- 0: random order,
- 1: random order where action 1 is favored with a 50% chance, then action 2 with a 25% chance, then actions 3 and 4, each with a 12.5% chance,
- 2: cycles through each action in order,
- 3: staggered order, which gives a 50% chance of choosing either action 1 or 2 the first turn, then a 50% chance of choosing either action 3 or 4 the next turn, and then it repeats.

`Boss Flag` can be set either to 0 to indicate a normal enemy or 1 to indicate a boss (which means you won't be able to run away from it).

`Death Sound` can either be `normal` for normal enemies or `boss` for bosses (which will make all other enemies disappear upon defeat); `Boss Flag` does not need to be set for this, however.

`Death Text Pointer` and `Encounter Text Pointer` respectively indicate the location of the text to be displayed once the enemy has been defeated and once he has been encountered.

`Defense`, `Guts`, `HP`, `Level`, `Luck`, `Offense`, `PP` and `Speed` represent the enemy's statistic levels, much like the player's levels. `Level` itself is used to determine whether the enemy should run away on sight or not.

`Experience points` specifies the amount of experience which should be split between party members.

`Fire/Flash/Freeze/Hypnosis/Brainshock/Paralysis vulnerability` describe how the enemy should be affected by specified attack or effect: if it is an attack, the percentage represents the damage that should be dealt to the enemy instead of the one it would normally receive; if it is a status effect, it is the chance that the effect will succeed.

`Gender` is used to tell EarthBound how the text should reference the enemy (as a male, female or neutral being).

`Initial Status` is the status the enemy should initially be affected with: `normal`, `asleep`, `cannot concentrate`, `feeling strange`, `psi shield alpha`, `psi shield beta`, `shield alpha`, or `shield beta`, each state being rather self-descriptive.

`Item Dropped` points to the item from `item_configuration_table.yml`, if any, using its ID. If an item is provided, its rarity must be specified with `Item Rarity`, which can take any value between 0 and 7. The actual chance is calculated with the following formula, with i being the specified value: $2^i/128$.

`Max Call` references the maximal number of enemies that can be called by the current enemy.

`Mirror Success Rate` represents the chance that Poo's Mirror ability will succeed using a

percentage (despite the absence of a % sign).

`Miss Rate` is the chance that the enemy will miss its attack upon the player character, with 0 being “always hit” and 16 being “always miss”.

`Money` is the amount of money the enemy will drop upon being defeated.

`Movement pattern` specifies the type of movement to use on the world map.

`Music` is the musical piece which should play during the battle, assuming this enemy is the one encountered on the world map.

`Name` is, unsurprisingly, the name of the enemy.

`Overworld Sprite` indicates the sprite from `SpriteGroups/` to use on the world map, using its ID.

`PP` is the amount of PSI power this enemy has when the battle begins.

`Row` indicates the row the enemy should appear in when the battle begins.

`Run Flag` specifies whether the player should be able to run from the enemy (7) or not (6).

`Type` describes the kind of enemy the player characters are faced with: is it a `normal` enemy? Is it made of `metal`? Is it an `insect`?

And finally, there is an additional `Unknown` property for each enemy. Will you be the first to discover what it does?

III. Enemy Groups

The groups of enemies encountered in battle are described in `enemy_groups.yml`, and allow you to control what types of enemies should be available in one group initially, what backgrounds should be displayed (see below for configuring these), and more. For example, let’s take the group that makes up the Titanic Ant boss:

```
450:
  Background 1: 170
  Background 2: 169
  Enemies:
    0: {Amount: 1, Enemy: 37}
    1: {Amount: 2, Enemy: 209}
  Fear event flag: 0
  Fear mode: run away if flag is unset
  Letterbox Size: 0
```

`Background 1/2` specify two possible animated background to layer when the battle begins, as defined in `bg_data_table.yml`. If only one is specified, it should have a color depth of 4; if two are specified, they should have color depths of 2.

`Enemies` is a list of enemies to add to the enemy group, with the following format: `0: {Amount: number of enemies, Enemy: ID of enemy}` , where the ID references the IDs of `enemy_configuration_table.yml`.

`Fear event flag` specifies the flag which will determine the enemy group's fleeing behavior, as set with the next property.

`Fear mode` describes the way the enemy group will behave according to the specified flag; it can either be run away if flag is unset or run away if flag is set.

`Letterbox Size` indicates the size of the black bars at the top and bottom of the screen when entering battle mode; it can go from 0 to 3.

However, defining the enemy groups is insufficient for them to be used; you also need to specify how they should be handled on the map in `map_enemy_groups.yml`; you can later use these definitions in EbProjEdit. Each *map group* is divided into two sub-groups, the first to be used before a flag is set, the second used after the flag is set; each sub-group randomly selects an enemy group to present to the player. For example, here is the definition for one of the groups found early on in Onett, initially made up of Spiteful Crows, later on made up of Starmen and Evil Eyes:

```
1:
  Event Flag: 0x84
  Sub-Group 1:
    0: {Enemy Group: 1, Probability: 2}
    1: {Enemy Group: 2, Probability: 3}
    2: {Enemy Group: 3, Probability: 3}
  Sub-Group 1 Rate: 5
  Sub-Group 2:
    0: {Enemy Group: 4, Probability: 2}
    1: {Enemy Group: 5, Probability: 2}
    2: {Enemy Group: 6, Probability: 2}
    3: {Enemy Group: 7, Probability: 2}
  Sub-Group 2 Rate: 8
```

This syntax should look pretty familiar to you by now.

`Event Flag` is the flag which, when set, specifies that `Sub-Group 2` should be used; when unset, `Sub-Group 1` will be presented to the user.

`Sub-Group 1` and `Sub-Group 2` list the enemy groups which could be selected from randomly (using its `Probability`), specified by their ID from `enemy_groups.yml`. The probabilities must add up to 8.

`Sub-Group 1/2 Rate` specifies the chance, expressed as a percentage without the percentage sign, that the group will spawn.

Conclusion

That was quite a bit of information, but at least you now know that your enemies are fully configurable. So, will you come up with the next Giygas?

Chapter 8.

Battle – Backgrounds

One of EarthBound's most noticeable features is that, unlike other RPGs which display animated enemies on static backgrounds, EarthBound displays a variety of shifting and twisting backgrounds behind the enemy sprites. CoilSnake allows you to customize your battles' backgrounds to your liking in a surprisingly easy manner, without even requiring manual animation.

List of files used:

- BattleBGs/
- bg_data_table.yml
- bg_distortion_table.yml
- bg_scrolling_table.yml

I. Background Images

The images are all stored in the BattleBGs/ directory, where their filename serves as an ID to identify them for CoilSnake. Each image is used as a base for the battle background, and depending on the battle, it can go from being hardly modified to being changed almost beyond recognition, using the settings which we will now study.

II. Configuring Backgrounds

Each entry in bg_data_table.yml describes the effects which should be applied to the image of the same ID in BattleBGs/. There are essentially three types of modifications which can be applied:

- **Distortion:** will bend and stretch the background image; EarthBound will cycle through each distortion specified.
- **Color:** the depth of color and the palette can both be customized.
- **Scrolling Movement:** will scroll the image in both vertical and horizontal directions, as specified; EarthBound will cycle through each scrolling movement specified.

Every property is self-descriptive in bg_data_table.yml, bg_distortion_table.yml and bg_scrolling_table.yml (assuming you understand how background animations are generated). A later edition might cover these in more detail, but for now know that the IDs specified in the Distortion fields reference the entries in bg_distortion_table.yml and that the IDs in the Scrolling Movement fields reference the entries in bg_scrolling_table.yml.

Conclusion

This chapter does not dwell in specifics for now; however, assuming you have a general grasp of how battle animations are generated, this should be enough to get you started.

Chapter 9.

Items

In our modern, materialistic and consumerist societies, where would we be without our precious items? EagleLand seems to have adopted a similar consumerist philosophy, which is why there is such a variety of items available, in and out of battle. And, wouldn't you know it, CoilSnake allows you to customize them too.

List of files used:

- `condiment_table.yml`
- `consolation_item_table.yml`
- `item_configuration_table.yml`
- `timed_delivery_table.yml`
- `timed_item_transformation_table.yml`

I. Configuring Items

The standard properties for items are all set in `item_configuration_table.yml`. Let's look at the entry for the Super plush bear.

```
3:
Argument:
- 17
- 1
- 1
- 0
Cost: 1198
Effect: 1
Help Text Pointer: $c53761
Misc Flags: 15
Name: Super plush bear
Type: 4
```

Again, a great deal of this should seem familiar.

`Argument` is a list of options the meaning of which varies depending on the effect of the item.

`Cost` is the monetary cost of this item in stores.

`Effect` is the ID of the action (from `battle_action_table.yml`) that this item should take.

`Help Text Pointer` accepts either a pointer or a CCScript label for the text to be displayed as a description for the item.

`Misc Flags` is for other flags which have varying purposes.

`Name` is the display name of this item.

Type indicates what kind of item it is:

- 0: a special type used for the Franklin badge,
- 4: teddy bears,
- 8: broken objects which Jeff can fix,
- 16: melee weapons,
- 17: ranged weapons (guns, yo-yos, beams...),
- 20: items that can be equipped on the body,
- 24: items that can be equipped on the arms,
- 28: items that can be equipped in the "Other" section,
- 32: food items,
- 36: beverages and capsules,
- 40: condiments,
- 44: the Large pizza,
- 48: items with special properties,
- 52: one-use items,
- 53: items which boost/lower statistics in battle,
- 56: miscellaneous items which don't fit elsewhere,
- 58: are for items which will only work if a sector property is set,
- 59: game items that aren't directly used.

Let's look at some of those items in more detail.

II. Condiments

Condiments get a special treatment, since they allow for "upgrading" of food items; the food items which can get upgraded are defined in `condiment_table.yml`. Each block corresponds to one item in `item_configuration_table.yml`; let's look at the one applied to Luxury jerky:

```
40:
  bad recover: 2
  condiment 1: 126
  condiment 2: 118
  effect: restore hp
  food: 245
  good recover: 100
  run time: 0
```

It might not be immediately obvious what all of these do, so let's take a look at them.

`food` is the ID of the food item defined in `item_configuration_table.yml`.

`condiment 1` and `condiment 2` indicate the two condiments' IDs from `item_configuration_table.yml` which can be applied to this item.

effect is the type of effect the condiment will have on the food item; this can either be restore hp, restore pp, restore hp/pp, increase random stat, increase iq, increase guts, increase speed, increase vitality, increase luck or no visible effect.

good recover is the statistical gain any character but Poo will receive.

bad recover is the statistical gain Poo will receive.

run time is duration for which the item will have an effect on the player (for Skip Sandwiches for example).

III. Special Items

There are two other files used in conjunction with items. The first is `timed_item_transformation_table.yml`, which is used for items like chicks, which eventually grow into chickens.

```
0:
  Delay: 50
  Item ID: 92
  New Item: 168
  Sound Effect: 0
  Sound Frequency: 0
```

Delay is the time it takes for the item to transform into the next item.

Item ID is the ID of the current item as defined in `item_configuration_table.yml`.

New Item is the ID of the item to transform into.

Sound Effect is the sound effect if any to play when this item is acquired. It shall repeated with the frequency set by Sound Frequency.

Additionally, the Li'l UFO and the Cute Li'l UFO have a special property whereby they might drop a random "consolation item". The `consolation_item_table.yml` file lists all the possible items they can drop. The file is completely self-explanatory: Enemy ID points to the appropriate UFO in `enemy_configuration_table.yml`, and the Item IDs are the ones defined in `item_configuration_table.yml`.

IV. Timed Deliveries

Escargo Express and Mach Pizza can both deliver items to you when you require it, but they do take a certain time to arrive; these times are defined in `timed_delivery_table.yml` as follows:

```
0:
  Delivery Failure Text Pointer: $c64cf8
  Delivery Success Text Pointer: $c64bbf
  Event Flag: 0xb4
  Sprite Group: 151
```

```
Timer: 180
```

```
Unknown:
```

- 6
- 0
- 15
- 0

```
Unknown2:
```

- 0
- 2
- 0
- 2

This is the definition for a delivery by Mach Pizza.

`Delivery Failure Text Pointer` indicates the text to be displayed when the delivery couldn't be made (because the area is unreachable), whereas `Delivery Success Text Pointer` indicates the text displayed upon a successful delivery of the requested item.

`Event Flag` is the flag which, once set, schedules the delivery, and is unset once the delivery is completed.

`Sprite Group` is the ID of the sprite group of the delivery person.

`Timer` is the time it takes for an item to be delivered.

`Unknown` and `Unknown2` are (obviously) unknown properties – can you figure them out?

Conclusion

Now that you know how to manage items, it's time to make sure the user can get them from stores... keep reading.

Chapter 10.

Stores

There are 65 stores in EarthBound, all defined in one file listing the items available for purchase in it.

List of files used:

- `store_table.yml`

The syntax for each block explains itself:

```
1:  
Item 1: 17  
Item 2: 18  
Item 3: 49  
Item 4: 74  
Item 5: 64  
Item 6: 0  
Item 7: 0
```

You can provide a total of 7 items defined in `item_configuration_table.yml`. And that's all there is to it.

Conclusion

That was easy. Let's move on to something a tad more involved now: the User Interface.

Chapter 11.

The User Interface

The User Interface is what allows the user to actually control the game, and is an obviously important component. CoilSnake provides some degree of control over it, mostly over the appearance of window elements and text labels.

List of files used:

- Fonts/
- Logos/
- WindowGraphics/
- cmd_window_text.yml
- text_misc.yml
- window_configuration_table.yml

I. Windows, Fonts and Logos

EarthBound uses windows to interact with the user; their appearance is defined in the Windows1_X.png files in WindowGraphics/, where the color palette is provided, along with the special images (such as "SMAAAASH!!"), the status icons, the HP/PP icons... Additionally, the Windows2_X.png files provide window borders and corners. The images 0 through 4 are used for the flavors listed in flavor_names.txt (in order), whereas 5 is used when a character is dead.

Notice: CoilSnake does not actually use anything but Windows1_0.png and Windows2_0.png for the layout of the windows; the other files are used purely to get the color for the other flavors.

The individual window elements are all put together into actual windows in window_configuration_table.yml, using the following format:

```
0:
  Height: 8
  Width: 13
  X Offset: 1
  Y Offset: 1
```

Height and Width define the height and width of your window (shocking, wouldn't you say?).

X Offset and Y Offset define the distance the window should be from the left and top border respectively.

All four numbers represent units of 8 pixels.

Another important element is the font library, which is available under `Fonts/`; each font is represented by two files: the first file (with the `.png` file extension) defines the actual font characters, whereas the `.yaml` file sets the individual width of each character in the font file. The 0th font is the standard font, the 1st font is the M. Saturn font, the 2nd font is the flyover font (used at the beginning of the game), the 3rd font is for HP and PP display and the 4th font is for window titles. Additionally, the credits font is monospace, and therefore there is no associated `.yaml` file.

Finally, CoilSnake also allows you to edit the logos displayed at the beginning of the game; they are all available in the `Logos/` directory, and can be edited like any other image file in EarthBound.

II. Interface Text

There are two files used to edit the interface text: `cmd_window_table.yaml` (for the main menu screen brought up with A) and `text_misc.yaml` for a great number of miscellaneous labels.

Conclusion

Modifying the interface might be useful for major changes to EarthBound (for example, a complete redesign of the menu interface). Looks aren't everything, though – sound matters too, which is why we'll look at music in the next chapter.

Chapter 12.

Music

While CoilSnake does not allow you to actually edit the music in EarthBound (a task better suited to other tools), it does allow you to select what music will be played and when.

List of files used:

- `map_music.yml`

Each block in `map_music.yml` represents a set of background music tracks which can be played back by EarthBound at certain locations; the IDs for the blocks are used in `map_sectors.yml` (which you should edit with EbProjEdit). Let's look at Map Music 35:

```
35:  
- Event Flag: 0x8216  
  Music: 87  
- Event Flag: 0x8049  
  Music: 93  
- Event Flag: 0x8217  
  Music: 77  
- Event Flag: 0x8047  
  Music: 129  
- Event Flag: 0x0  
  Music: 45
```

This is a list of possible background tracks; the one which is actually played will depend on the Event Flags which are currently set. EarthBound checks them in order: if it is set, play the specified Music (as listed in the Music Editor); otherwise, go to the next one. The last option should always have Event Flag: `0x0`, so that it is always true if all others are false.

Notice: `0x8_____` flags are the reverse of `0x_____` flags; if one is set, the other is unset.

Conclusion

These musical blocks are useful for re-usability, since several areas might play the same set of songs depending on the current state of the game.

Chapter 13.

Miscellaneous

But wait, there's more! There are many tiny, secondary features which add up for interesting effects, depending on what you want to do with your hack. Those that didn't fit into any specific category or that weren't big enough to warrant a whole chapter are explained here.

List of files used:

- `attract_mode_txt.yml`
- `patches.yml`
- `telephone_contacts_table.yml`
- `teleport_destination_table.yml`

I. Attract Mode

Attract Mode is the name given to the cutscenes shown if the player doesn't press any buttons after starting the game, and are used to attract the player (hence the name). It is controlled through `attract_mode_txt.yml`, which lists all the pointers to the cutscenes to play in order; these could be replaced with CCScript labels, for example, for custom attract mode cutscenes.

II. Patches

Over the years, a series of patches useful for EarthBound development have been developed by various people; CoilSnake can apply some of these patches automatically, using the `patches.yml` file. EB++, for example, which lets you store variables and perform arithmetic on them; switch it to enabled if you wish to make use of it.

III. Telephone Contacts

Over the course of his adventure, Ness will pick up a series of telephone contacts which he can call; these contacts are defined in `telephone_contacts_table.yml`, with a very simple syntax:

```
3:
Event Flag: 0xca
Name: Mach Pizza
Text Pointer: $c64a1c
```

This should be extremely familiar to you by now.

`Event Flag` is the flag which must be set for this contact to appear in the list.

`Name` is the name of the entry to be displayed.

`Text Pointer` points to the dialogue which should be initiated when the call is started.

IV. Teleport Destinations

Sometimes, EarthBound needs to teleport characters to another place (for example, when Ness and Paula are captured in the hotel); these teleportation options are defined in `teleport_destination_table.yml`, each operation being described as follows:

```
4:  
  Direction: 3  
  Unknown: 127  
  Warp Style: 1  
  X: 740  
  Y: 813
```

`Direction` is the direction the player characters should be facing when the teleportation is complete (you'll probably only use 1, 3, 5 or 7, which correspond to up, right, down and left respectively).

`Warp Style` indicates the way in which the teleportation should take place. There are 52 different warp styles (many of them used in Moonside), which will be covered in the appendix of a later edition.

`X` and `Y` are the coordinates (expressed in units of 8 pixels) to which the character will be warped.

And again, `Unknown` is a property of unknown meaning.

Notice: if you are familiar with CCScript, you might be interested to know that the teleport destinations correspond to the "warp" command in CCScript.

Conclusion

And these little tidbits conclude this presentation of CoilSnake. By now, you should have enough knowledge to make basic hacks on a variety of levels. However, this is just the beginning, as the true power of hacking derives not from a single tool, but from the conjoined use of many. Later editions of this manual will cover the other tools you can use for development.