Stratagems Functional Overlay – a function library for WEIDU modding

# Introduction

Stratagems Functional Overlay (SFO) is a set of tools for Infinity Engine modding, built as a collection of functions written in WEIDU (the standard IE mod-installation tool).It has a slightly two-sided nature – it was written as a power tool for advanced applications, but in fact it might also be simpler than straight WEIDU coding relatively simple mods. (I say “might” because it’s always hard to guess what other people will find easy or difficult.) For that reason, this manual is also written at very variable levels. Chapter 1 (“Introductory SFO”) assumes no knowledge of WEIDU coding and only an elementary knowledge of the Infinity Engine file system (I assume you know how to browse the game files in Near Infinity and that you’ve played with editing files in DLTCEP). Chapter 2 (“Advanced SFO”) assumes a fairly sophisticated knowledge of both. Chapter 3 (“Library details”) is a (fairly) full rundown of SFO’s contents.

# Chapter 1: Introductory SFO

## Elementary WEIDU

WEIDU (created by Wesley Weimer, heavily developed by Valerio Biggiani (“TheBigg”) and maintained currently by Wisp) is the standard modding tool for writing and installing IE mods. To set it up:

* Download the latest version of WEIDU. The latest *stable* version is at <http://weidu.org/~thebigg/> ; the latest *beta* version is at <http://weidu.org/~thebigg/beta/>. In general you shouldn’t use beta versions unless you know what you’re doing; however, SFO uses features of WEIDU not present in the latest stable version (231) at time of writing, so if 232 hasn’t been released yet you’ll need to use the beta.
* Unzip the download and extract weidu.exe to the folder in which your IE game is installed. (I’ll be assuming this is BG2 for the most part; SFO also works with Icewind Dale and BG1, and with the Enhanced Editions of the games.) Then rename it to setup-mymod.exe. (Where “mymod” is the name of your actual mod.)
* Create a new folder, “mymod”, in your BG2 directory. In that folder, create a new text file and call it “setup-mymod.tp2”.
* Open up “setup-mymod.tp2” in a text editor (Notepad is fine; Context is better; lots of others exist and everyone has their own favourites). Enter the following text (it doesn’t matter exactly how long the lines of //////// are, or even if you put them in at all; they’re just there for readability):

BACKUP “stratagems\_external/mymod/backup”

AUTHOR “Myname (my email address)”

/////////////////////////////////////////////////////////////////////////////////////////////////////

BEGIN “Boring component”

/////////////////////////////////////////////////////////////////////////////////////////////////////

OUTER\_SPRINT boring “This is a boring string”

PRINT “The value of the boring string is ‘%boring%’”

//////////////////////////////////////////////////////////////////////////////////////////////////////

BEGIN “Even more boring component”

//////////////////////////////////////////////////////////////////////////////////////////////////////

PRINT “Here is some text”

If you now double-click on setup-mymod.exe, you will be offered the choice of installing the two boring components. If you do, you should see some text displayed.

Let’s look at the basic structure of the file. It starts with a block called the *preamble*, which specifies some stuff relevant to all components of the mod. In this case, the only information in the preamble is the folder used to back up the mod[[1]](#footnote-1) and the information to display to the user if the mod fails to install.

After that is a list of components, each starting with a BEGIN ~My component name~ (and each being ended, implicitly, either by the next BEGIN line or the end of the file). In the text following the BEGIN line you put the instructions required to install the component.

In this case, the instructions don’t do anything very interesting. All the first component is doing is defining a variable (‘boring’), specifying what the value of that variable is, and then printing it. It’s boring because it isn’t actually doing anything to *Baldur’s Gate* at all (and if you just want to program without any IE-specific applications, WEIDU is a lousy choice). The second component is even worse.

WEIDU comes equipped with a huge range of commands that could be put into the text of the component instead. They are exhaustively detailed at <http://www.weidu.org/~thebigg/README-WeiDU.html> . In programming terms the standard WEIDU commands are mostly “low level” – they let you insert or modify bytes in the files on the basis of how far into the file they occur, without much engagement with the high-level interpretation of a given file as (say) a spell or creature. In SFO modding, we largely eschew these commands in favour of high-level commands defined within SFO.

## Installing SFO

To install SFO and get it running:

* Download the latest version of SFO
* Unzip the download and put the folder “sfo” into your mymod folder. (If you intend to use the SSL scripting metalanguage, do the same for the folder “ssl”; if you don’t, or don’t know what that means, don’t bother.)
* Add the following code to your preamble (straight after the “AUTHOR” line in the tp2 file):

AUTO\_EVAL\_STRINGS

ALWAYS

OUTER\_SPRINT scsroot mymod

OUTER\_SET debug\_variable=0

OUTER\_SPRINT workspace “stratagems\_external/workspace”

INCLUDE “%scsroot%/sfo/install\_sfo.tpa”

END

You should replace “mymod” with the name of your mod, as usual. Other than that, don’t worry too much about what this means.

To check it’s installed correctly, add the following to one of the boring components in the TP2:

OUTER\_SPRINT list “Minsc Boo”

LAF return\_first\_entry STR\_VAR list RET entry=entry list=list END

PRINT “First entry is %entry% and second entry is %list%”

If you get “First entry is Minsc and second entry is Boo” you’ve done it right.

## Using SFO to install files

The simplest sort of mod (at least from a WEIDU point of view) consists of a bunch of files (with file extensions CRE, ITM, STO and the like) made using DLTCEP and a bunch of uncompiled scripts and dialogs (with extensions D and BAF) written as text files. (This guide doesn’t pretend to tell you how to use DLTCEP or how to write scripts and dialogs.)

Here’s how to install these files using SFO. Let’s suppose we’ve got some component of the mod which is intended to set up a quest (or whatever) involving an awesome frog. The biggest job is to make the files, but you’re on your own there.

Next, create a subfolder of your mymod folder, say “awesomefrog”, to hold all the files and scripts needed for the component (let’s say: frog1.cre, frog2.cre, frogswd.itm, froghost.baf, frog.baf, frog.d). Just put all the files and scripts into that folder.[[2]](#footnote-2)

Then, add the component’s code to setup-mymod.tp2:

BEGIN “Awesome Frog component”

OUTER\_SPRINT component\_loc awesomefrog

LAF install STR\_VAR files=”frog1.cre frog2.cre frogswd.itm froghost.baf frog.baf frog.d” END

The first line here just defines the component, as with regular WEIDU.

The second tells SFO which folder the component is located in.

The work is done by the third line. Let’s look at it step by step.

“LAF” stands for LAUNCH\_ACTION\_FUNCTION. Almost all SFO coding is done by “launching” (i.e., running) various functions. (There are also PATCH\_FUNCTIONS but we mostly won’t be needing them.) A function can be thought of as a black box to which we feed certain information and which does things based on that information.

The word after LAF is the name of the particular function that is to be run. In this case, it’s the “install” function, whose job it is to take a bunch of files and install them in the game.

“STR\_VAR” tells WEIDU that we’re ready to list the “string variables” of the function. These are (one part of) the information that the function needs.

After “STR\_VAR”, there is a list of the form “thing1=thing2”. The left-hand part is the *variable name*, which you can think of as a label on one slot of the black-box that is the function. The right-hand part is the *variable value*, which is the thing to put into the slot. Some functions need only one variable, some need many. “install” only needs one: a list of all the items to be installed. (Notice that we put quote marks around this list: that’s because it’s got more than one word in, and we need to tell WEIDU that all the items in the list (not just the first one) are to be assigned to the “files” variable.)

Finally, “END” tells WEIDU that we’ve finished specifying the function.

(This isn’t a full list of what calling a function involves; we’ll see other examples later.)

So much for how we write down the function. What does it do? It takes all the files in the list “files”, looks for them in “component\_loc” (i.e. in the awesomefrog folder, in this case), and installs them in the game. For most files this just means copying them over. For scripts (baf and d files[[3]](#footnote-3)) it means compiling them (and doing variable substitution if appropriate; don’t worry if you don’t know what that means).

If awesomefrog is a particularly large or complex component, you might want to organise its files, rather than dumping them all into the awesomefrog folder. You can do this in two ways:

1. Add a new string variable, “location”, that says which subfolder of awesomefrog contains the files, as in

LAF install STR\_VAR location=resources files=”frog1.cre frog2.cre frogswd.itm froghost.baf frog.baf frog.d” END

The install function will now look in the mymod/awesomefrog/resources folder, instead of in mymod/awesomefrog.

1. Define a “local location” in the TP2 code, like this:

OUTER\_SPRINT local\_loc firstbit

If you include this, SFO will act as if the component location is mymod/awesomefrog/firstbit, at least until you change the value of local\_loc.

In general, you’ll want to use the first method; the second is only really sensible for really large mod components that fall apart into discrete chunks.

Just for comparison, here’s (in the most straightforward way) how to install the files just in baseline WEIDU, without using SFO:

COMPILE froghost.baf EVALUATE\_BUFFER

COMPILE frog.baf EVALUATE\_BUFFER

COMPILE frog.d EVALUATE\_BUFFER

COPY\_EXISTING

“mymod/awesomefrog/frog1.cre” “override”

“mymod/awesomefrog/frog2.cre” “override”

“mymod/awesomefrog/frogswd.itm” “override”

So SFO is a bit quicker and easier to read, and also lets you change the location of folders without having to rewrite the code. (But ultimately it’s not a major improvement: that comes later.)

## Using SFO to edit files: STO files as an introduction

The real use of SFO is to make modifications to files (new or existing) during the install process, rather than just copying over files made elsewhere. The format for doing so is a bit abstract, so it’s helpful to explain it through a relatively simple example: STO files, which define stores (and bags of holding).

The STO file, like most IE data files, consists of two parts: the *header*, which contains information that all store files need, and the *entries,* the number of which vary from file to file. In both the header and the entries can be found *fields*, which store information about the in-game object represented by the file.

In store files, the header includes information like whether the store provides rooms, how much the rooms cost, what the cost is to identify an item, whether you can make temple donations at the store, what graphic (tavern/inn/temple/container) to use for the store, and whether the store buys stolen goods.

The possible entry types for stores are “drink” (each entry describes a drink for sale), “item” (each entry describes an item for sale), “cure” (each entry describes a [normally curative] spell that can be cast for a price), and “item\_purchased” (each entry describes a kind of item, e.g. swords or books, that the store will buy).

### 1.4.1 Simple editing

To edit stores, we use a function called (appropriately) edit\_store. Here’s a simple example of it in use:

LAF edit\_store STR\_VAR store=”doghma sartem01”

editstring=”sell\_markup=>200 identify\_cost=>50” END

The variable “store” just lists the files you want to edit (in this case, the temple of Oghma and the temple in Saradush). The variable “editstring” describes the edits you want to make: it’s a list of pairs, where the first pair is the field you want to edit and the second pair is the value you want to set it to. In this case, the command increases the markup of sold goods to 200% and halves the identify cost to 50. You can find lists of these standard commands in the library part of this manual. I’ll call pairs like this command=>argument pairs; a list of command=>argument pairs is a *patch*, a set of instructions on how to modify a file.

Before going any further, here are some useful variants on this example:

1. Suppose you don’t want to set the identify cost to 50 if it’s already lower than 50? Identify\_costLT=>50 sets the identify cost to 50 or to its current value, whichever is lower. identifycostGT works in the equivalent way. You can use this LT/GT syntax for most commands that set numerical values.
2. Suppose you have many patches to do. It can get a bit unreadable fitting them into the same editstring command. There are two ways to make things easier:
3. The simplest thing is just to define “editstring” somewhere else where you have more space, as in

OUTER\_SPRINT editstring “sell\_markup=>200 identify\_cost=>50”

LAF edit\_store STR\_VAR store=”doghma sartem01” editstring END

By leaving “editstring” not set to anything you’re telling WEIDU to use whatever value it’s already been set to (in this case, in the previous line).

1. For larger or more complex modifications you can shift from the “inline” way of defining patches (via editstring) to the alternative, “array” way of doing it. Here’s how:

ACTION\_CLEAR\_ARRAY patch\_data

ACTION\_DEFINE\_ASSOCIATIVE\_ARRAY patch\_data BEGIN

room\_peasant=>0

room\_merchant=>0

room\_noble=>1

room\_royal=>1

price\_noble=>50

price\_royal=>100

END

LAF edit\_store STR\_VAR store=binnkeep edits=patch\_data END

This code turns the inn into a very high-class establishment: only noble and royal rooms are available, and only for a pretty penny. You can replace “patch\_data” with whatever name you like; if it’s a name you are *sure* you’ve not used before you can leave out the ACTION\_CLEAR\_ARRAY line. Defining patches via arrays in this way is more cumbersome than the inline method but usually clearer; do it for modifications with more than a few commands. (In “organising your code”, below, I describe a way of making this format less cumbersome.)

1. Suppose you’re not sure if a given store is in the game or not? (This might apply if you’re modding content added by another mod.[[4]](#footnote-4)) Just add the argument “allow\_missing” to edit\_store, and set it to “1” (meaning “yes” or “true”), as in

LAF edit\_store INT\_VAR allow\_missing=1

STR\_VAR store=”dw#temog” editstring=”identify\_cost=>50” allow\_missing=yes END

This will patch dw#temog.sto if it’s present in the game, and skip it if not. (Normally, edit\_store will cause the installation of the component to fail if it can’t find a file it’s asked to find.)

Notice that allow\_missing is an example of an “integer variable”: a variable whose value is an integer, as opposed to the “string variables” we have been using so far. They are entered in the function in a slightly different way, as shown (not that any disaster will occur if you just put it in the “STR\_VAR” part.)

Incidentally, allow\_missing is also an example of an *optional variable*: you can set it if you like, and if you don’t then it’s set to a default value (“0”, in this case.)

### 1.4.2 Editing names and descriptions

Let’s talk about some more complex command=>argument pairs. Suppose you want to change the *name* of the store (say, to the “temple of sanctimonious sanctitiy”) You might imagine that something like name=>Temple of Sanctimonious Sanctity would do it, but it doesn’t, for two reasons.

The first is a bit shallow. The new name consists of four words, and so really needs to be enclosed in quotes so that WEIDU knows to keep the words together. But since the editstring itself is generally enclosed in quotes, this yields something like

editstring=”identify\_cost=>50 name=>”Temple of Sanctimonious Sanctity””, which is a mess.

You can get around this fairly straightforwardly. WEIDU provides two sorts of quotation mark: ~ and “. You could use ~ to enclose the name, or you could use the array input form (where the outer quotation marks aren’t needed).

The more serious problem is that names (and text in general) in BG2 aren’t stored in the file itself; they’re stored in one place, the Dialog file (called dialog.tlk). The reason for this is to facilitate translation: what gets called “Sword” in English needs to be “Epee” in French and “schwert” in German. By including in the “sword” file not the text itself but a number that identifies the text in the dialog file, the job of translating the game was reduced to translating one (admittedly huge) file.

For modding purposes, that means that you need to get any new names you want to use into the dialog file. SFO provides two methods for doing this:

1. The simplest method uses the “set\_” group of commands. The command=>argument pair

set\_name=>~Temple of Sanctimonious Sancitity~

for instance, adds “Temple of Sanctimonious Sanctity” to the dialog file and adds a reference to the dialog file into whatever store we’re patching.

1. The problem with the simplest method is that it’s language-specific. If you aspire to have your mod translated at some future time, you need to use a reference to a translation file instead. I won’t explain here how that works (look at the WEIDU documentation or any tutorial on dialog editing), but basically the English version of the dialog file will contain a line like

@442 = “Temple of Sanctimonious Sanctity”

You can now use the “say\_” group of commands to add that line to your file, referencing it by its number, as in

say\_name=>442

**Pro tip:** changing a mod to be translation-friendly after the fact is a pain. If you have any aspiration to make it translation-friendly at some later time, use a tra file right away.

Once in a while you will genuinely want to edit the string *number*. (For instance, maybe you are adding a string that is already in the game.) Then you can use the “\_string” group of commands. For instance, name\_string=>71106 sets the store’s name to “The Zephyr”.

### 1.4.3 Variants on “edit\_store”

The “edit\_store” function is ideal if you want to make modifications to a particular in-game file, but this isn’t the only circumstance in which you might want to apply patches. SFO supports five different formats for applying patches to files of a given type; the others, in the case of stores, are clone\_store, install\_store, edit\_all\_stores and make\_store.

1. clone\_store is used to make a modified copy of a store. (Maybe there’s another temple of Oghma you want to add elsewhere that charges less to identify things?) It works like this:

LAF clone\_store STR\_VAR store=”doghma=>dw#temog” editstring=”identify\_cost=>50” END

So the syntax is a list of pairs joined by arrows (in this case, only one such pair) – in each case, a copy of the first element in the pair is made and given the second element as its name, and then the patch is applied to the copy (leaving the original unchanged). This is a common way to build new resources in the game, using old ones as a starting point.

1. Suppose you want to apply this patch to all stores, not just these two. Just use the edit\_all\_stores function, not the edit\_store function (and drop the “store” argument). So

LAF edit\_all\_stores STR\_VAR editstring=”price\_royal=>100” END

raises the cost of staying the night in a royal suite to 100gp everywhere in the game.

1. Suppose you want to make a modified copy of a store? (Maybe there’s another temple of Oghma you want to add elsewhere that charges less to identify things?) Use clone\_store instead of edit\_store, like this:

LAF clone\_store STR\_VAR store=”doghma=>dw#temog” editstring=”identify\_cost=>50” END

This makes a copy of doghma.sto called dw#temog.sto and then patches the copy. (“dw#” is my modding prefix, which I use to make sure my files don’t overlap with other people’s. You should use your own; a list of available ones can be found at <http://forums.blackwyrmlair.net/index.php?showtopic=113> )

### 1.4.4 Adding and editing entries: drinks as an example

Modifying a file’s list of entries (in the case of stores: their drinks, cures and items for sale) is slightly trickier than modifying its header, because you need to say *which* entry you want to modify. SFO’s basic tool to handle this is called an *entry patch*: a patch that is executed on every entry of a given type matching some condition.

For instance, consider the drinks that a store sells. Each drink is specified by four pieces of information:

* “special\_rumor”, a rumor (i.e., a dialog file) associated specifically to that drink;
* “drinkname”, the actual name of the drink
* “drink\_price”, how much the drink costs
* “intoxication”, how drunk it gets you.

Suppose that you want to increase the price of Arabellan Wine in every store in the game to 50gp, and to make it super-intoxicating. (Don’t ask me why.) We can pick out Arabellan wine by its string number, 14760, and we can make the patch like this:

ACTION\_CLEAR\_ARRAY patch\_data

ACTION\_DEFINE\_ASSOCIATIVE\_ARRAY patch\_data BEGIN

match=>”drinkname\_string=14760”

drink\_price=>50

intoxication=>75

END

ACTION\_CLEAR\_ARRAY patch\_data

ACTION\_DEFINE\_ASSOCIATIVE\_ARRAY patch\_data BEGIN

patch\_drink=>patch\_data

END

LAF edit\_all\_stores STR\_VAR edits=patch\_data END

There are two patches here. The first is the entry patch: it looks like an ordinary patch but with one special command=>argument pair: match. The argument of “match” specifies a condition that must be met for the entry to be patched, in this case that the string naming the drink equals 14760.

So in this case, the idea is that SFO goes through all entries of the given type (drinks), and for each, it checks the “match” condition (the drink name string) and sees if it’s true (i.e., checks if the drinkname string is 14760, the string number for Arabellan wine). If it is, the other patches are applied to that entry.

What about the second patch? That’s the patch that actually gets applied: the instruction to apply an *entry* patch is itself one of the instructions (in this case, the only instruction) in an ordinary patch. (Leaving out this second step and just trying to apply the conditional patch directly to the file is in my experience an easy mistake to make.)

I’ve written the instructions out this way to show the structure, but it’s admittedly a bit cumbersome.[[5]](#footnote-5) We can make it more concise by using inline rather than array patches. For instance, instead of defining patch\_data we could just have used

LAF edit\_all\_stores STR\_VAR editstring=”patch\_drink=>patch\_data” END

Alternatively, you can make the conditional patch into an inline patch instead, like this:

ACTION\_CLEAR\_ARRAY patch\_data

ACTION\_DEFINE\_ASSOCIATIVE\_ARRAY patch\_data BEGIN

patch\_drink\_inline=> “match=>~drinkname\_string=14760~

drink\_price=>50

intoxication=>75”

END

LAF edit\_all\_stores STR\_VAR edits=patch\_data END

(Notice the nested quotation marks again.)

In principle you can even combine the two, though it can get hard to read.

Again, there are many variations:

1. What if you want to edit *all* entries of a given type? You want a match and check that together always return true. Just leave out the match command entirely, SFO assumes that you meant to edit all the patches.
2. What if you want to add a new entry? You can do so with “add\_[entry]” or “add\_[entry]\_inline”. This works basically the same as patch\_[entry], but the “match” command is checked against the header, not against any entry (and usually you won’t need it). Here’s code to add a teetotal option to a tavern:

ACTION\_CLEAR\_ARRAY patch\_data

ACTION\_DEFINE\_ASSOCIATIVE\_ARRAY patch\_data BEGIN

add\_drink\_inline=>~ drink\_price=>5

set\_drinkname=>”Non-Alcoholic Cocktail”

intoxication=>0~

END

LAF edit\_store STR\_VAR store=ambar01 edits=patch\_data END

By default, the new entry gets put at the top of the list of entries. If you want it at the end, just put “at\_end=>1” as one of your arguments in the add\_[entry] patch. If you want it a precise distance from the top, (say, below the first N entries) use “insert\_point=>N” instead.

1. Instead of just adding a new entry directly, you can “clone” (i.e. copy) an existing entry and then apply changes. This can be useful if you want to base the structure of the new one on the old one; it’s also a good way to make sure your entry appears in the right place in the list.

Cloning entries uses “clone\_[entry]” and “clone\_[entry]\_inline”; the block has exactly the same structure as for “patch\_[entry]” but now, if the conditions are met, and new block is created and the patch is applied to that block. Here’s code to make sure that every bar that stocks Arabellan Wine also stocks Brynnlaw Rotgut:

ACTION\_CLEAR\_ARRAY extra\_drink\_patch

ACTION\_DEFINE\_ASSOCIATIVE\_ARRAY extra\_drink\_patch BEGIN

type=>drink

match=>”drinkname\_string=14760”

set\_drinkname=>”Brynnlaw Rotgut”

drink\_price=>10

intoxication=>100

END

LAF edit\_all\_stores STR\_VAR editstring=”clone\_drink=>extra\_drink\_patch” END

By default, the new entry is put in immediately after the old one. If you want it immediately before, instead, put “clone\_above=>1” into the patch.

1. What about if you want to delete an entry altogether? You can use delete\_[entry] and delete\_[entry]\_inline, which work a bit like patch\_[entry] but just remove any entry satisfying the condition. Here’s code to remove Arabellan wine altogether from the game:

LAF edit\_all\_stores STR\_VAR

editstring=”delete\_drink\_inline=>~match=>”drinkname\_string=14760”~”

END

1. What if you want to check more than one thing at a time? You can use the special match value “combine\_checks”. This combines two or more matches: the matches you want to make, and their desired check values, are fed to SFO in via a command called “match\_parameter”. To illustrate, here’s code to delete any Arabellan Wine that costs 5gp:

ACTION\_CLEAR\_ARRAY delete\_wine\_patch

ACTION\_DEFINE\_ASSOCIATIVE\_ARRAY delete\_wine\_patch BEGIN

type=>drink

match=>combine\_checks

check=>1

match\_parameter=>”drinkname\_string=>14760 drink\_price=>5”

END

LAF edit\_all\_stores STR\_VAR editstring=”delete\_entries=>delete\_wine\_patch” END

SFO goes through the command=>argument pairs in match\_parameter. In each case it checks if the value of the command property equals the argument – is drinkname\_string 14760; does the drink cost 5gp? If the answer is yes in each case, the value of combine\_checks is 1; otherwise, it’s 0.

(This only scratches the surface of SFO’s conditional patching, but its full power requires a bit more faculty with WEIDU function design than I’m assuming here.)

### 1.4.5 Adding cure spells, the automatic spell-naming system, and repeated entries in arrays

The discussion above is pretty general, and applies to basically any entry type in the game. So you can (e.g.) patch the cure-spell files in a temple in just the same way. Here’s code to raise the base price of Cure Serious Wounds to 250gp in the Temple of Lathander:

ACTION\_CLEAR\_ARRAY cure\_patch

ACTION\_DEFINE\_ASSOCIATIVE\_ARRAY cure\_patch BEGIN

type=>cure

match=>cure

check=>sppr401

cure\_price=>250

END

LAF edit\_store STR\_VAR store=temlath editstring=”patch\_conditionally=>cure\_patch” END

Here the match is on the actual name of the spell file – in this case, Cure Serious Wounds, as you can check in Near Infinity, is sppr401.spl.

However, there’s a useful trick here that’s worth noting. Spell names like “sppr401” aren’t exactly easy to remember, so IE games give many of them (basically, all the ones with names like “sppr”, “spin”, or “spwi”) a more memorable name that lives in spell.ids – sppr401, for instance, is CLERIC\_CURE\_SERIOUS\_WOUNDS. When SFO is loaded, it creates a variable for each such memorable name and sets it equal to the spell filename. So any time you use “%CLERIC\_CURE\_SERIOUS\_WOUNDS%”, or ~%CLERIC\_CURE\_SERIOUS\_WOUNDS%~, in a command=>argument patch (or anywhere else, in fact), WEIDU will automatically substitute the correct filename. Here’s code to delete all raise dead spells from the game:

ACTION\_CLEAR\_ARRAY patch\_data

ACTION\_DEFINE\_ASSOCIATIVE\_ARRAY patch\_data BEGIN

delete\_entries\_inline=>”type=>cure match=>cure check=~%CLERIC\_RAISE\_DEAD%~”

delete\_entries\_inline’=>”type=>cure match=>cure check=~%CLERIC\_RESURRECTION%~”

END

LAF edit\_all\_stores STR\_VAR edits=patch\_data END

Why the ‘ sign after the second delete\_entries\_inline? Because of a niggling detail with array-mode patches: they can never contain the same command twice. (If they do, WEIDU just overwrites the argument in the first pair with the one in the second.) This didn’t matter until we started patching conditionally (there’s no point having the same command twice anyway in an ordinary patch). To work around it, you can put any number of ‘ symbols, or a ‘ symbol followed by any integer, after a command in an SFO patch, and it will be ignored. So use these to distinguish your commands in complex patches.

1. The particular choice here (stratagems\_external/mymod/backup) stores the backup information as a subfolder of a folder (“stratagems”) common to several of my mods and which is shareable with others. A more traditional alternative would be to choose mymod/backup, which stores the backup information as a subfolder of your own mod folder. I am inclined to think that it is better not to do this as it makes it easy to accidentally distribute a (potentially large) backup folder when you publish the mod, but it’s up to you. *Do not,* however, replace “stratagems\_external” with “mymod\_external”; we don’t need two folders per mod cluttering up the game directory! [↑](#footnote-ref-1)
2. It’s common for mods to be organised vertically rather than horizontally, with all files of a given type (e.g CRE) in one folder irrespective [↑](#footnote-ref-2)
3. And SSL files. [↑](#footnote-ref-3)
4. Warning: some people get annoyed if you mod their mods. This strikes me as hypocritical, but consider yourself warned in any case. (Annoyance is highest if you modify NPCs, which is perhaps understandable as that sort of mod is a bit like creative fiction.) [↑](#footnote-ref-4)
5. Although if you think that’s cumbersome, try doing it natively in WEIDU. [↑](#footnote-ref-5)