SDMM v0.1 Modders' Guide

Pherakki

Contents

1 Introduction **1**
1.1 Overview . . . 1
1.2 Additional UI Elements for Modders . . . 1
1.2.1 Extract Tab . . . 1

2 Game Asset Structure **2**
2.1 Asset Archives . . . 2
2.2 Main MDB1 Contents . . . 2
2.3 SFX MDB1 Contents . . . 5
2.4 BGM AFS2 Contents . . . 5
2.5 VFX AFS2 Contents . . . 6

3 Creating SDMM Mods **7**
3.1 Basic Mod Structure . . . 7
3.2 Adding and Editing Files . . . 8
3.2.1 CSV Files . . . 8
3.2.2 MBE Tables . . . 9
3.2.3 Script Files . . . 9
3.2.4 Model Files . . . 11
3.2.5 Including and Overwriting Vanilla Files . . . 12
3.3 Installing Data Outside of DSDBP . . . 13
3.4 Readying your Mod for Distribution . . . 13

4 SDMM Mod-Building Features **14**
4.1 Softcodes . . . 14
4.1.1 The Need for Softcodes . . . 14
4.1.2 Softcode Aliases . . . 15
4.1.3 Softcode List . . . 16
4.2 Mod Manager Variables . . . 20
4.2.1 What are Mod Manager Variables? . . . 20
4.2.2 Variable Types List . . . 20
4.2.3 Example Variables.txt . . . 21
4.3 Build Scripts . . . 22
4.3.1 Overview . . . 22
4.3.2 Basic Build Scripts . . . 22
4.3.3 Building Multiple Files with Build Patterns . . . 23
4.3.4 Build Script Variable Types List . . . 25

5 CYMIS **26**
5.1 Overview . . . 26
5.1.1 Components . . . 26
5.1.2 Flags and the Flag Table . . . 26
5.1.3 Important Files . . . 26
5.1.4 A Note on Aliases, Build Scripts, and Variables . . . 26
5.2 Specification . . . 27
5.2.1 Top Level . . . 27
5.2.2 Wizard Script . . . 27
5.3 Flag Types . . . 28
5.3.1 Flag . . . 28
5.3.2 HiddenFlag . . . 29
5.3.3 ChooseOne . . . 29
5.4 Install Script . . . 30
5.5 Condition Rules . . . 32
5.5.1 and . . . 32
5.5.2 or . . . 32
5.5.3 not . . . 32
5.6 Installation Rules . . . 32
5.6.1 copy . . . 32

1 Introduction

1.1 Overview

SimpleDSCSModManager (SDMM) is a poorly-named mod manager and mod-merging utility designed for the PC release of Digimon Story Cyber Sleuth: Complete Edition (DSCS). This guide is designed for people who want to create mods for DSCS, and will walk you through the features that the mod manager offers, as well as the basic mechanics of the asset structure. You are **highly recommended to read the users' guide first**, since it contains information on basic use of the mod manager and mod installation, as well as most of the UI.

Cyber Sleuth is not a straightforward game to mod. Like many games, it has a static asset database and no capability for merging files. If you attempt to add information to the game contained in a file, it will overwrite that portion of the game's database instead of adding it to the database. Therefore, in order to add or edit data, it much be patched in alongside the already-existing vanilla data. The goal of the mod manager is to handle as much of this automatically as possible, whilst minimising asset redistribution, and give mod authors as many tools as possible to edit the game's asset structure whilst maintaining maximum compatibility with other mods.

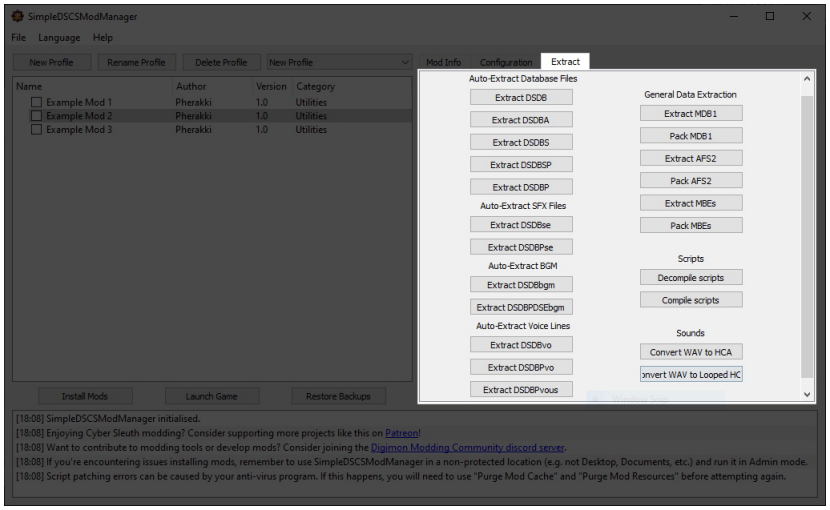
This article exists to introduce you to the patching features of the mod manager, and how to use them. This article will not tell you how to achieve specific things, such as how to edit or add a specific high-level feature. Additional articles dedicated to these specific topics will appear alongside the rest of the mod manager documentation over time. You should treat this document as a reference guide for the tools the mod manager can offer to you to achieve your goals, and refer to specific implementation articles outside of this document for guidance on how to add various features to your mods. The description of the asset structure in this document should, however, give you enough pieces to get started and figure things out for yourself.

I hope you like JSON files.

1.2 Additional UI Elements for Modders

1.2.1 Extract Tab

The Extract Tab allows you to extract various game assets, as well as manipulate them. The left-hand column is largely self-explanatory. The game has 12 asset archives, which are all listed in the left column and categorised according to their function. This is covered in detail in section 2. The right-hand column is for manual manipulation of the game data. The various asset types listed here will be covered in more data also in section 2. You are unlikely to need any of these buttons, with the exception of the HCA conversion buttons if you are adding sounds to the game.



2 Game Asset Structure

2.1 Asset Archives

The game data is stored in the **resources** folder of your game install. The most important files in here are the 12 MVGL archives. One thing to note is that “MVGL" is not a filetype. There are two kinds of archives in the resource folder: MDB1 and AFS2. MDB1 contains generic compressed data. AFS2 contains HCA sound files. There are five “main game data archives", listed here in order of increasing priority:

- **DSDB** - MDB1 archive. Contains the majority of the game data.
- **DSDBA** - MDB1 archive. Contains overwrite data, mostly censorship data such as the Sistermon Ciel model.
- **DSDBS** - MDB1 archive. Contains overwrite data for the PC version with an XBOX Controller.
- **DSDBSP** - MDB1 archive. Contains overwrite data for the PC version with a DualShock Controller.
- **DSDBP** - MDB1 archive. Contains more overwrite data, mostly data that was provided as DLCs for the Vita and PS4 versions. This is the highest-priority archive that will overwrite all others, and by default the mod manager installs data into this archive.

There are two sound effect archives:

- **DSDBse** - MDB1 archive. Contains SFX.
- **DSDBPse** - MDB1 archive. Contains more SFX.

There are two BGM archives:

- **DSDBbgm** - AFS2 archive. Contains the game soundtrack.
- **DSDBPDSEbgm** - AFS2 archive. Contains (very) small snippets of Digimon anime OST.

There are three voice effect archives:

- **DSDBvo** - AFS2 archive. Contains the majority of the game’s voicelines.
- **DSDBPvo** - AFS2 archive. Contains additional voicelines.
- **DSDBvous** - AFS2 archives. Contains censorship voicelines.

The **usm** files in the resources folder are video files.

2.2 Main MDB1 Contents

An important thing to note before diving into the MDB1 data is that the files inside it follow very strong naming conventions. That is to say, the game will look for assets with very specific filenames for certain game data records. This is both a blessing and a curse; there is in some sense less work to do since the game will automatically pick up and understand what certain files are supposed to be based on their filenames, but that automatic behaviour also means we must adhere to the filename restrictions the game places upon us. This is another reason for the internal complexity of the mod manager. We will cover current knowledge throughout this section.

An MDB1 archive can contain any of the following folders, in addition to model files that are not in any sub-folders:

- **data** - Contains the MBE archives that make up the game database, as well as some CSV files that also contribute to the database, in addition to a single image 'save_icon0.png'. The MBE archives are automatically unpacked to folders of CSV files if you tell the mod manager to unpack a specific archive (*i.e.* select an option in the left-hand column of the Extract Tab). The CSV files in the MBE archives have **UTF-8** encoding. The "loose" CSV files have **SHIFT-JIS** encoding. The "loose" CSV files are:
  - bgm.csv
  - debug_call_script.csv
  - debug_call_script_hm.csv
  - se.csv
  - soundtest_bgm.csv
  - soundtest_se.csv
  - soundtest_voice.csv
  - voice.csv
  - voice_us.csv
- **message** - Contains MBE archives that store the text for conversations. The MBE archives are automatically unpacked to folders of CSV files if you tell the mod manager to unpack a specific archive (*i.e.* select an option in the left-hand column of the Extract Tab). The CSV files in the MBE archives have UTF-8 encoding.
- **text** - Contains MBE archives that store various pieces of text, such as character names and item descriptions. The MBE archives are automatically unpacked to folders of CSV files if you tell the mod manager to unpack a specific archive (*i.e.* select an option in the left-hand column of the Extract Tab). The CSV files in the MBE archives have UTF-8 encoding.
- **images** - Contains all image files. For the PC version, these are DDS files with the file extension 'img'.
- **images_as** - Contains alternative images that are used when the language is set to Traditional Chinese.
- **images_de** - Contains alternative images that are used when the language is set to German.
- **images_kr** - Contains alternative images that are used when the language is set to Korean.
- **script64** - Contains script files that the game attaches to various game contexts, such as battles and maps. The scripts are compiled Squirrel 2.2.4 files. The scripts are automatically decompiled to text files if you tell the mod manager to unpack a specific archive (*i.e.* select an option in the left-hand column of the Extract Tab).
- **shaders** - Contains shaders used by the model files. The shaders are plaintext, and written in the Cg language, the language of a deprecated rendering system that is a predecessor-of-sorts to HLSL. The game uses the Cg rendering runtime in conjunction with OpenGL.

The files not in any folders are model files, with the exception of a single image file 'ui_title_logo_caution.img'. The models can be split into a few categories:

- **acc** - Accessory models.
- **cam** - Camera models, including those for attacks.
- **eff** - Effect files, including those for attacks.

- **chr** - Digimon/Enemy models.
- **pc** - Player Character models, including outfits.
- **npc** - Supporting cast models.
- **mob** - Generic NPC models.
- **dXXXX** - Dungeon map models, which are split into the same sub-types as the Town map models.
- **tXXXX** - Town map models, which are split into the following sub-types:
  - **tXXXXf** - The “Field" model. Contains the mesh that is rendered for the map.
  - **tXXXXc** - The “Collision" model. Contains invisible walls.
  - **tXXXXs** - The “Surface" model. Contains a copy of the floor.
  - **tXXXXp** - The “Position" model. Contains trigger meshes, trigger surfaces, NPC positions, spawn points, etc.
  - **tXXXX_cam** - The “Camera" model. Contains the camera for the map.
  - **tXXXX_hide** - The “Hide" model. These are meshes that can become transparent when the player enters a certain trigger, usually to prevent the mesh from obscuring the camera. Linked to trigger surfaces in the “Position" model.
- **ui** - Various bits of the user interface, which are all implemented as 3D models.

In addition to these, there are a number of other models that aren’t fully categorised.

The models themselves are split between nine filetypes:

- **name** - [Required] Contains bone names and material names.
- **skel** - [Required] Contains the “rest pose" of the model, which is equivalent to the first frame of the “base animation" for that model. A few models have different skels and first-frame-of-base-animations, but this is suspected to be a mistake on the part of the developer, and the game corrects for this regardless.
- **geom** - [Required] Contains the “bind pose" of the model, the materials, texture names, lights, and cameras.
- **anim** - [Required] Contains animation data.
- **detr** - [Optional] Unknown, seems to contain walk paths for NPCs in maps.
- **phys** - [Optional] Contains colliders for maps.
- **note** - [Optional] Unknown, seems to affect the rendering of text.
- **sprk** - [Optional] Unknown, seems to contain particle effects.
- **navi** - [Optional] Unknown, used in “autoran" models. May be another kind of walk path.

All files with the same filename are associated with the same model — *e.g.* ‘t0101.name‘ and ‘t0101.skel‘ would be parts of the same model. An exception is made for the Anim files - the file ‘t0101.anim‘ would be interpreted as the “base animation" for the model ‘t0101’ if the other files for that model also exist. This is an animation that is constantly playing, and for many models it contains no data. There are also some “overlay animations" for models. These are identified by suffixes being attached to the filename, *e.g.* for a model with a base animation called ‘chr003.anim‘, there might be an overlay animation called ‘chr003_ba01.anim‘. Many of these overlay animations have strict naming conventions, such that the game can automatically know where an animation file for a certain action should exist. Other suffixes are arbitrary. A few of the suffixes are:

- **ba01** - Battle Attack 1. The animation attached to the ‘basic attack’ function in battles.
- **ba02** - Battle Attack 2. The animation attached to the ‘skill’ function in battles.
- **bb01** - Battle Backwards 1. An animation for moving backwards.
- **bd01** - Battle Damage 1. The animation played when receiving damage.
- **bd02** - Battle Damage 2. The animation played when being knocked down.
- **bd03** - Battle Damage 3. The animation played when getting back up again.
- **bg01** - Battle Guard 1. The animation played when received damage while guarding.
- **bn01** - Battle Normal 1. The “rest" animation played during battles.
- **br01** - Battle Run 1. The “run" animation played in battles.
- **bs01** - Battle Special 1. The animation for the “Special Skill 1’.
- **bs02** - Battle Special 2. The animation for the “Special Skill 2’.
- **bv01** - Battle Victory 1. The animation played when a battle if won.
- **fe01** - Field Eat 1. The animation played when eating food in the DigiFarm.
- **fn01** - Field Normal 1. The animation played when “resting" in the field.
- **fr01** - Field Run 1. The “run" animation played in the field.
- **fw01** - Field Walk 1. The “walk" animation played in the field.
- **fw02** - Field Walk 2. A slower “walk" animation played in the field.

The game will fall back to alternatives if animations do not exist in some cases, for example the game will use ‘br01’ instead of ‘fr01’ if ‘fr01’ does not exist. For this reason, many **chr** models do not have most of the ‘field’ animations defined.

Other overlay animations seem to have names that are not strictly enforced by the game, but the developers still use a naming convention for.

- **e** - “Emote" animations used in conversations.
- **ev** - “Event" animations used for in-engine cutscenes, usually suffixed with a chapter and number.
- **fXX_mXX** - Face and Mouth animations. These change the facial expression.

2.3 SFX MDB1 Contents

The SFX MDB1 archives just contain a folder called “sound", which contains HCA files (although they may have the file extension ‘snds’).

2.4 BGM AFS2 Contents

The BGM AFS2 archives are standard AFS2 archives. The contents are extracted with filenames corresponding to their entry number in the AFS2, in **hexadecimal** (not decimal). The game finds these files using **bgm.csv**, which is in the **DSDB** archive. This csv gives each member of a AFS2 archive a name for internal use within the game. The ‘**cri_contents_id**’ column gives the file index, in **decimal**. For example, an entry in the csv with a cri_contents_id of **17** would correspond to the file **000011.hca**.

Note also that the name of the AFS2 archive the file belongs to is given. The game will dynamically load data from whatever name you put here for an AFS2. So, you can add new BGM tracks by creating a new AFS2 file and referring to the name of the AFS2 file in that column.

2.5 VFX AFS2 Contents

The VFX AFS2 archives are standard AFS2 archives. The contents are handled identically to the BGM files, except that the csv that assigns each track a name used by the game is **voice.csv** and **voice_us.csv**.

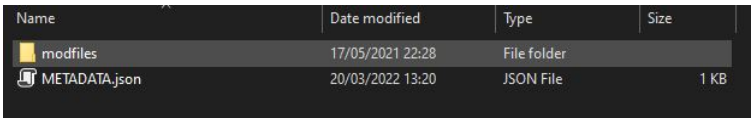
3 Creating SDMM Mods

3.1 Basic Mod Structure

Mods require two things at a minimum:

- A folder called **modfiles**
- A file called **METADATA.json** containing a JSON dictionary

The "modfiles" folder contains the mod data. **For now, we will just describe how to add data that goes in the 'Main Asset Data' archives**, *i.e.* data that will be inserted into **DSDBP**. We will describe how to add data that should go into other archives and loose files, in section 3.3. It should be structured like an unpacked game archive, with the "modfiles" folder in the place of the folder the unpacked game data would be in (i.e. "modfiles" might contain the folders "data/", "message/", "script64/" *etc*).



Name	Date modified	Type	Size
modfiles	17/05/2021 22:28	File folder	
METADATA.json	20/03/2022 13:20	JSON File	1 KB

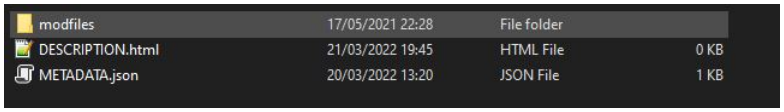
The **METADATA.json** can contain five main keys:

- Name
- Author
- Version
- Category
- Description

All of these should be strings (*i.e.* inside quotes), and additionally the "Description" field may be written in HTML.

```
{
    "Name": "Example Mod 1",
    "Author": "Pherakki",
    "Version": "1.0",
    "Category": "Utilities",
    "Description": "Short Description"
}
```

The "Description" entry field is very restrictive since the description must be all on a single line; if you want to have a more complex description making use of more HTML features, you can also add a '**DESCRIPTION.html**' file to your mod that will take priority over the Description member of the metadata file.



modfiles	17/05/2021 22:28	File folder	
DESCRIPTION.html	21/03/2022 19:45	HTML File	0 KB
METADATA.json	20/03/2022 13:20	JSON File	1 KB

3.2 Adding and Editing Files

The mod manager gives you several tools to edit the game data. The mod manager uses two core ideas to patch the game: **filetypes** and **rules**. The mod manager will scan your mod, and automatically identify certain files (such as csv files) as being specific filetypes. The mod manager will then use that file, in conjunction with a **rule**, to merge those files into the game data. Every filetype known to the mod manager has a default rule, meaning that you can just put a bunch of files into a mod and the mod manager will patch them into the game using the most commonly-desired method. You can change precisely how the mod manager will use your filetype by specifying different **rule**s, but we will see how to do this in section 4.3. For now, let's take a look at the various filetypes the mod manager will automatically recognise and patch into the game data. Note that, if a file is not recognised as any of these filetypes, the mod manager will copy the file into the game data "normally" which will cause it to take priority over a vanilla file with the same name, if one exists. In each of these sections, we will list the default rule that each filetype uses so we can refer back to these later.

This can get a bit complicated and difficult to describe in words. It's a good idea to download a few mods designed for SDMM, and cross-reference how they are set up and what data they contain with the descriptions given in this section. Remember also that this document is a reference guide! There are a huge number of possible things to mod in this game, and so instructions on how to mod a specific piece of the game are not included in this guide, lest it grow to an unmaintainable size and need extremely frequent updates. You can refer to separate articles on how to implement specific bits of functionality, which should be kept in the Documentation folder of the SDMM GitHub repository.

3.2.1 CSV Files

File Types

- **.csv Extension**
  To edit CSV files, add a csv file with the same path as the csv you want to edit. For example, if you want to edit **data/bgm.csv**, add a file called **bgm.csv** to a folder called **data** inside the **modfiles** directory of your mod. The first line of your mod CSV is ignored, so you can put the header of the original file in there to help keep track of what the columns are. In the following rows (which I will refer to as **records**), you can enter the data you wish to add to the game. Typically, the very first cell of the record is an 'ID' column. In some rare cases, the 'ID' is two or more cells. If the 'ID' of a record in your modded CSV is the same as one that already exists, your game data will be used to edit the vanilla game data. If it's a new ID, then it will make a new record.

  Default Rule: mberecord_merge
  The default behaviour is to replace (or add) the entire record corresponding to the ID specified in your file with the data contained in that row. If you leave a cell in the csv blank, the cell will act as if it is "transparent" and will not overwrite the original value that was in that cell with your data. You can use this to make surgical edits, such as editing a single value inside a record.

Rules

- **mberecord_merge**
  Replace (or add) the entire record corresponding to the ID specified in your file with the data contained in that row. If you leave a cell in the csv blank, the cell will act as if it is "transparent" and will not overwrite the original value that was in that cell with your data.

- **mberecord_overwrite**
  Replace (or add) the entire record corresponding to the ID specified in your file with the data contained in that row. This completely overwrites the entire record.

- **mberecord_append**
  Strips the padding data from the current record data, appends the data in your records on the

end, and re-pads with the padding characters. Default padding character is **0**. This can be changed by giving a padding character as a rule argument in a build script, *e.g.* [filepath, “mberecord_append", “-1"].

- **mberecord_remove**
  Strips the padding data from the current record data, removes any entries in your records from the current record data, and re-pads with the padding characters. Default padding character is **0**. This can be changed by giving a padding character as a rule argument in a build script, *e.g.* [filepath, “mberecord_remove", “-1"].

- **mbetable_overwrite**
  Completely replaces the csv with the version in the mod. Use this instead of the usual ‘overwrite’ rule to replace a csv file.

3.2.2 MBE Tables

File Types

- **.csv Extension**
  MBE tables are handled very similarly to .csv files, since the MBE data is unpacked to .csv files by DSCSTools (which the mod manager uses to unpack and re-pack the game data). The difference between MBE tables and csv files is that the mod manager will **recognise a file contained inside a folder ending with ‘.mbe’** as part of an MBE table. For example, **data/charname.mbe/Sheet.csv** will be recognised as an MBE table, whereas **data/charname/Sheet.csv** will be recognised as a csv file.

  Similarly to csv files, you need only include the data you wish to edit or add inside your MBE tables. An additional consideration in this case is that MBEs often contain multiple tables — they unpack to several csv files. You only need to include the files that you edit with your mod. The mod manager will take care of merging this in to the MBE alongside the other tables. Other than this, editing MBE tables is identical to editing csv files.

  Default rule: mberecord_merge

Rules
Rules are identical to the CSV rules.

3.2.3 Script Files

File Types

- **.txt Extension**
  Script files are identified as files with a **.txt** extension inside the **script64** folder. **Files outside of script64 will not be recognised as script files**. Any text inside your script file will be appended to the existing scripts. Due to the dynamic nature of the Squirrel programming language, if you create a function with the same name as a function that already exists, your function will overwrite that function.

  Default rule: squirrel_concat

- **.sqmod Extension**
  Sometimes, you don’t want to add or overwrite entire functions, but rather you want to extend pre-existing functions or replace function calls. To do this, you can use a ‘.sqmod’ file. This is a JSON file containing a list of dictionaries, each of which implements a Squirrel Mod rule. A list of these rules is given below. This list of rules will grow as the mod manager is updated.

- This rule adds code immediately before the first function definition in the script file.
  - **add_preamble**: A list of strings. Each string is a line of code to add to the function.
- This rule will add the code specified in the 'with' argument to the function assign to 'extend_func'.
  - **extend_func** : Assigned the name of the function you want to add code to.
  - with : A list of strings. Each string is a line of code to add to the function.
- This rule performs a straight find-and-replace of the text given to 'replace' with the text given to 'with'. **Can lead to very unexpected results if not used carefully**.
  - **replace**: The string to replace.
  - **with**: The string to fill in the replacements with.
- This rule will find the function call specified to 'replace_call' and replace it with the function call given to 'with'. Function arguments are identified by curly braces containing a hash alongside an argument ID number. These arguments can be re-used in the 'with' statement to re-arrange or adjust function arguments.
  - **replace_call**: Receives the function call to replace (see the example given at the bottom of the sqmod documentation)
  - with: Receives the edited function call.
- This rule functions identically to 'replace_call', but only works inside the functions named inside the 'funcs' argument.
  - **replace_call_in_funcs**: Receives the function call to replace (see the example given at the bottom of the sqmod documentation)
  - with: Receives the edited function call.
  - funcs: A list of strings. Each string is the name of a function that this find-and-replace will apply to.

Example sqmod file from the skip dialogue mod. globalSkipTextMode and globalSkipAnimMultiplier are variables added with a .txt script file.

```
[
    {
        "replace_call": "this.Talk.PlayAnimation({#0}, {#1}, {#2}, {#3})",
        "with": "this.Talk.PlayAnimation({#0}, {#1}, globalSkipTextMode ? ({#2}/
            globalSkipAnimMultiplier) : ({#2}), {#3})"
    },
    {
        "replace_call": "this.Vista.PlayAnimation({#0}, {#1}, {#2}, {#3})",
        "with": "this.Vista.PlayAnimation({#0}, {#1}, globalSkipTextMode ? ({#2}/
            globalSkipAnimMultiplier) : ({#2}), {#3})"
    }
]
```

Default rule: squirrel_modify

Rules

- **squirrel_concat**
  Appends the text in a file onto the end of the current source code.
- **squirrel_modify**
  Parses the file as a "sqmod" file, and applies those edits to the current source code following the sqmod scheme.

3.2.4 Model Files

File Types

- **.name Extension**
  A plain 'name' file. Overwrites the pre-existing file by default.

  Default rule: overwrite

- **.skel Extension**
  A plain 'skel' file. Overwrites the pre-existing file by default.

  Default rule: overwrite

- **.geom Extension**
  A plain 'name' geom. Overwrites the pre-existing file by default.

  Default rule: overwrite

- **.anim Extension**
  A plain 'anim' file. Overwrites the pre-existing file by default.

  Default rule: overwrite

- **.mdledit Extension**
  A '.mdledit' file allows you to make specific edits to model files. Currently, it is an extremely limited proof-of-concept that allows you to add new NPCs to maps only.

  The mdledit file is a JSON file consisting of a dictionary of mdledit rules. The dictionary can have repeated entries. Each rule is itself given a dictionary of parameters. A list of mdledit rules and their parameters are given below.

  - **editNPC**.
    - *id* - [Required]
      The NPC ID.
    - *position* - [Optional, Default: [0, 0, 0]]
      A list of 3 elements containing the [x, y, z] coordinates of the NPC in the map.
    - *rotation* - [Optional, Default: [1, 0, 0, 0]]
      A list of 4 elements containing a quaternion rotation, in WXYZ ordering, for the NPC.
    - *scale* - [Optional, Default: [1, 1, 1]]
      A list of 3 elements containing the [x, y, z] scales for the NPC.

Default rule: mdledit_name/skel/geom/anim

Rules

- **mdledit_name**
  Applies the mdledit editing scheme to name files.

- **mdledit_skel**
  Applies the mdledit editing scheme to skel files.

- **mdledit_geom**
  Applies the mdledit editing scheme to geom files.

- **mdledit_anim**
  Applies the mdledit editing scheme to anim files.

3.2.5 Including and Overwriting Vanilla Files

- **.request Extension**
  Sometimes, you may need to re-use a vanilla file inside your mod. If this is the case, you don't need to distribute this asset with your mod — just include a Request file in the location where you want that file to be. For example, if you want to include the file 'chr003.name', add the file 'chr003.name.request' to your mod as a blank text file. If you request the version of that file from a specific archive, put the name of that archive inside the Request file (*e.g.* if you want to include the version of 'chr783.name' from DSDB, write the letters 'DSDB' in the Request file).

  Default rule: request_file

- **Any other file** Any other file just gets given the "overwrite" rule, which copies the file over the vanilla file.
  Default rule: overwrite

Rules

- **request_file**
  Implements the 'request' system.

- **overwrite**
  Copies the file over the vanilla file.

3.3 Installing Data Outside of DSDBP

The vast majority of mods will want to install data into the DSDBP archive. For this, and backwards-compatibility reasons, the mod manager will install all data into DSDBP. However, some data, such as SFX and music, should go into their own archives. We can tell the mod manager to switch to a mode where it can install data to several locations by adding the line **"FormatVersion": 2** to the **METADATA.json**.

```
{
    "FormatVersion": 2,
    "Name": "Example Mod 1",
    "Author": "Pherakki",
    "Version": "1.0",
    "Category": "Utilities",
    "Description": "Short Description"
}
```

You can then install data to specific archives by **making a folder with the name of the archive** inside the **modfiles** folder, and putting all data to go into that archive inside that folder. For example, to recover the previous behaviour of putting all data into DSDBP, create a folder called **DSDBP** inside your **modfiles** folder and put all data inside it. The mod manager will automatically recognise the names of archives, and if a folder is not recognised as one, its contents will be installed as a folder rather than inside an archive.

You are recommended to keep all 'main game data archive' data inside DSDBP. For adding new AFS2 data, you may find it easiest to use the dynamic-archive-loading features of DSCS by packing your HCA data into the AFS2 archives with the mod manager yourself, including the entire AFS2 archive inside the **modfiles** folder, and referring to the archive inside the appropriate csv loader file (*e.g.* bgm.csv). Remember, **DSDBP is an overwrite archive**, so putting your edits to *e.g.* bgm.csv inside **DSDBP** will work fine since the mod manager will patch them into the game data appropriately for you.

You are highly discouraged from inserting new data into the **DSDB** archive. This is a very large archive that will take some time to re-build, as well as not having much spare room in the first place. You are best off sticking to DSDBP. Note that data packed into MDB1 archives will be compressed, and can actually hold many times more data than the 4 GiB limit would suggest. In general, you should only need to use the **"FormatVersion": 2** option if you're looking to install audio data or loose files. Specific articles that go into detail for adding data with new AFS2 archives should exist in the mod manager documentation if you want an in-depth example.

3.4 Readying your Mod for Distribution

The mod manager expects mods to either be contained inside **zip** files, or in a folder containing the **modfiles** and **METADATA.json** files. For distribution, you should pack your mod into a **zip** file. The mod manager will recognise zip files in two formats:

- The zip file contains the **modfiles**, **METADATA.json** *etc.* in the top level of the zip archive, *i.e.* unpacking the zip creates **modfiles**, **METADATA.json** in the same location as the zip file.
- The zip file contains a single folder, which itself contains the **modfiles**, **METADATA.json** *etc.* This folder must have the same name as the zip file itself. For example, you could have a zip file called **MyMod.zip** that contains a folder called **MyMod**, which itself contains the contents of your mod.

4 SDMM Mod-Building Features

4.1 Softcodes

4.1.1 The Need for Softcodes

You may have noticed that the IDs for records in the game database are hardcoded. These IDs appear in the MBE files, in model files, in script files, and even in filenames. For example, the file 'chr003.name' Uses the digimon ID '3' in the filename. This ID is hard-linked to various MBE records in the game by the game logic.

In some ways, this is good, since we have less work to do. However, since IDs have to be hardcoded in this way, what happens when two mods both want to add something new under a new ID, and accidently use the same ID? The two mods will conflict, and one will gain precedence over the other. This is not a good situation for the modular addition of data to the game, since two mods can very easily become incompatible in this way.

This critical issue is behind the introduction of **Softcodes**. Softcodes are tags that the mod manager can detect, and automatically fill in with a mod-manager-given ID. Several games designed for mods do similar things out-of-the-box, and give IDs to new records when mods are loaded by the game. Unfortunately, since DSCS is *not* designed for mods, we have to create a similar system ourselves.

The basic Softcode syntax is

[Category::Key]

where 'Category' is a particular collection of IDs, and 'Key' is that name of an ID in that collection. A list of all Categories is given in section 4.1.3.

Some softcodes have sub-categories. You can access sub-categories using a **|** character. Softcodes can have arbitrarily deep numbers of sub-categories, although in practice this does not ever exceed three.

This still only returns a single number. An example of a sub-category would be an enemy variant of a Digimon. The number you would get is the ID of the variant.

[Category::Key|SubCategory::Key2|SubSubCategory::Key3]

Some IDs are expected by the game to be in certain formats. Many softcodes define functions to assist with this — for example, padding a code to always be 3 digits in size. These functions are also given per category in 4.1.3. You can only call a function on the **final key of a softcode**, because this is the number that the softcode resolves to.

The function is called by adding an extra '::' after the key, followed by the name of the function, followed by an open parenthesis and a closed parenthesis.

[Category::Key|SubCategory::Key2|SubSubCategory::Key3::function()]

The most important function of softcodes is the ability to add new IDs using them. In order to use a softcode, simply write a softcode with a key attached to a category in one of your files, for example **[Item::MyNewItem]**. If the key **MyNewItem** does not exist, the mod manager will automatically assign it a new ID from the **Item** category. A nice feature of this system is that other mods will also be able to refer to these keys, and build on or edit other mods in this way.

As previously mentioned, some softcodes need to be used in the names of files. This will be covered in section 4.3.

Please note that a complete set of required softcodes is not included with v0.1 of SDMM. This version released with a number of softcodes that should be sufficient for the most common modding needs. Additional softcodes will be added over time, eventually covering the entire database. You can always hardcode other IDs for now, with the option of updating these to softcodes when the appropriate codes are ready.

4.1.2 Softcode Aliases

Softcodes can sometimes get very long. To aid with this, you can **alias** chunks of softcodes in order to make them shorter or more idiomatic. For example, instead of writing

[Dungeon_CS::d90|SubArea::d9002|NPC::npc_0008]

you might prefer to write

[ShibuyaNPCs::npc_0008]

You can do this by adding an **ALIASES.json** to you mod in the same place as the **METADATA.json**. The **ALIASES.json** is a JSON dict containing string : string pairs. The left string is the alias, and the right string is the original piece of the softcode you want to replace. Note that Aliases are expected to **end with a category**. For example, the Alias given above would be defined in a file as

```
{
    "ShibuyaNPCs" : "Dungeon_CS::d90|SubArea::d9002|NPC"
}
```

You can equivalently define it as

```
{
    "ShibuyaNPCs::" : "Dungeon_CS::d90|SubArea::d9002|NPC::"
}
```

if you prefer, since the mod manager will automatically pad the right-hand side of the aliases up to two ':' characters. This is so that the softcodes you write in your files will always be in the softcode format, *i.e.* **[Category::Key]**, **[Category::Key|SubCategory::Key2]**, **[Category::Key::function()]** *etc.*, so that the mod manager can easily find them. This restriction may be lifted in the future so that individual key values can be aliased, but it is not a priority.

Note that aliases are specific to *your mod only*, so they are merely a convenience for you alone. Other mods are not able to access the aliases you define, and you are not able to access the aliases of other mods.

4.1.3 Softcode List

- **BattleBGM**
  - Purpose: IDs for Battle BGM tracks.
  - Value: Return the ID without modifications.
  - Methods: None
  - Children: None
- **Digimon**
  - Purpose: IDs for Digimon and Battle Entities.
  - Value: Return the ID without modifications.
  - Methods:
    * 3ID : Pad ID to the left with 0s to a width of 3 characters.
      e.g. $1 \rightarrow 001$
    * 4ID : Pad ID to the left with 0s to a width of 3 characters, and add an extra 1 to the left.
      e.g. $1 \rightarrow 1001$
    * filename : Pad ID to the left with 0s to a width of 3 characters and add 'chr' to the left.
      e.g. $1 \rightarrow$ chr001
  - Children:
    * **Digimon→EnemyVariant**
- **Digimon→EnemyVariant**
  - Purpose: IDs for different statlines for enemy Battle Entities.
  - Value: Return the ID without modifications.
  - Methods: None
  - Children: None
- **Dungeon_CS**
  - Purpose: IDs Dungeon-type Areas.
  - Value: Pad ID to the left with 0s to a width of 2 characters, add d to the front of the ID.
    e.g. 1→d01
    e.g. 101→d101
  - Methods:
    * ID : Pad ID to the left with 0s to a width of 2 characters.
      e.g. 1→01
      e.g. 101→101
  - Children:
    * **Dungeon_CS→SubArea**
- **Dungeon_CS→SubArea**
  - Purpose: IDs for SubAreas of Dungeon-type Areas.
  - Value: Pad ID to the left with 0s to a width of 2 characters, add the parent value to the front of the ID.
    e.g. 1→ d0101
    e.g. 1→ d10101

- Methods:
  * ID: Pad ID to the left with 0s to a width of 2 characters.
    e.g. 1→ 01
  * MapID: Pad ID to the left with 0s to a width of 2 characters, add the parent ID to the front of the ID.
    e.g. 1→ 101
    e.g. 1→ 10101
  * battleFile: SubArea Value with 'b' on the end of the file.
    e.g. 1→ d0101b
    e.g. 1→ d10101b
  * collisionFile: SubArea Value with 'c' on the end of the file.
    e.g. 1→ d0101c
    e.g. 1→ d10101c
  * positionFile: SubArea Value with 'p' on the end of the file.
    e.g. 1→ d0101p
    e.g. 1→ d10101p
  * surfaceFile: SubArea Value with 's' on the end of the file.
    e.g. 1→ d0101s
    e.g. 1→ d10101s
  * fieldFile: SubArea Value with 'f' on the end of the file.
    e.g. 1→ d0101f
    e.g. 1→ d10101f
  * cameraFile: SubArea Value with '_cam' on the end of the file.
    e.g. 1→ d0101_cam
    e.g. 1→ d10101_cam
  * hideablesFile: SubArea Value with '_hide' on the end of the file.
    e.g. 1→ d0101_hide
    e.g. 1→ d10101_hide
- Children:
  * **Dungeon_CS→SubArea→NPC**

- **Dungeon_CS→SubArea→NPC**
  - Purpose: IDs for NPCs in the SubAreas of Dungeon-type Areas.
  - Value: Returns the ID without modifications.
  - Methods:
    * bone_name: Pads the ID to the left with 0s to a width of 4 characters, and add 'npc_' to the left.
      e.g. 1→npc_0001
  - Children: None

- **Dungeon_HM**
  - Identical definition to **Dungeon_CS**.

- **Dungeon_HM→SubArea**
  - Identical definition to **Dungeon_CS→SubArea**.

- **Dungeon_HM→SubArea→NPC**
  - Identical definition to **Dungeon_CS→SubArea→NPC**.

- **Field_CS**
  - Purpose: IDs for all Areas used in the master list of Areas for Cyber Sleuth.
  - Value: Returns the ID without modifications.
  - Methods: None
  - Children: None
- **Field_HM**
  - Purpose: IDs for all Areas used in the master list of Areas for Hacker's Memory.
  - Value: Returns the ID without modifications.
  - Methods: None
  - Children: None
- **Item**
  - Purpose: IDs for Items.
  - Value: Returns the ID without modifications.
  - Methods: None
  - Children: None
- **Shop**
  - Purpose: IDs for Shops.
  - Value: Returns the ID without modifications.
  - Methods: None
  - Children: None
- **ShopLimitLineup**
  - Purpose: IDs for LimitLineup entries for shops.
  - Value: Returns the ID without modifications.
  - Methods: None
  - Children: None
- **ShopLineup**
  - Purpose: IDs for Shop inventories.
  - Value: Returns the ID without modifications.
  - Methods: None
  - Children: None
- **ShopText**
  - Purpose: IDs for text displayed in shops.
  - Value: Returns the ID without modifications.
  - Methods: None
  - Children: None
- **Skill**

  - Purpose: IDs for Battle Skills.
  - Value: Returns the ID without modifications.
  - Methods: None
  - Children: None

- **Speakers**
  - Purpose: IDs for new Speakers in dialogues.
  - Value: Returns the ID without modifications.
  - Methods: None
  - Children: None

- **SupportSkill**
  - Purpose: IDs for Battle Support Skills.
  - Value: Returns the ID without modifications.
  - Methods: None
  - Children: None

- **Town_CS**
  - Identical definition to **Dungeon_CS**, except with the d padding character replaced with t.

- **Town_CS→SubArea**
  - Identical definition to **Dungeon_CS→SubArea**, except with the d padding character replaced with t.

- **Town_CS→SubArea→NPC**
  - Identical definition to **Dungeon_CS→SubArea→NPC**.

- **Town_HM**
  - Identical definition to **Dungeon_CS**, except with the d padding character replaced with t.

- **Town_HM→SubArea**
  - Identical definition to **Dungeon_CS→SubArea**, except with the d padding character replaced with t.

- **Town_HM→SubArea→NPC**
  - Identical definition to **Dungeon_CS→SubArea→NPC**.

4.2 Mod Manager Variables

4.2.1 What are Mod Manager Variables?

The mod manager can keep track of certain variables that mods can share. The mod manager doesn't contain any by default, but mods can easily define and set new variables. For example, a shop could be set up to sell a certain type of item only. This list of items could be appended to by other mods, allowing the inventory of this shop to change depending on what mods are installed.

Mod manager variables share the same syntax as softcodes.

To use Mod Manager Variables, create a file called **Variables.txt** in the same location as your **METADATA.json**. Define a new variable by writing the name of the list in the text file. On each new line, you can then edit the variable using an *operator*. The details how to edit these variables are given in section . Note that **you can use softcodes inside the Variables.txt!**

As an example, let's say we've defined a new **VarList** called **MyListOfItems**. We could use this in our mods by tagging it like **[VarList::MyListOfItems]**. This would replace this tag with comma-separated entries of the list, inside square brackets. We could call functions on this key if we wanted to insert the list in a different way.

4.2.2 Variable Types List

- **VarList**
  Description
  VarList is list, or array, or unique items. It can have items added or removed by mods. Duplicates are ignored. Attempting to remove an item that does not exist does nothing.

  Operations

  - ::
    Sets the default value of the list. Items are separated by ',', without any kind of brackets. The default is overwritten by any edits to the list, and the default provided by the mod loaded last sets the default value of the list. If your mod uses a VarList and you do not want it to be forced to contain starting information, you should provide a default value for it.
    Example. ::0,0,0
  - ++
    Adds an item to the list.
    Example. ++40
  - −−
    Removes an item from the list, if it exists.
    Example. −−40

  Functions

  - **splat**
    Returns the list as comma-separated values without any square brackets. This can be used to insert a list as multiple cells in a CSV file.
  - **splat_strings**
    Same as splat, but wraps each entry in quotes.
  - **as_list**
    Returns the list as comma-separated values in square brackets.
  - **as_list_strings**
    Same as as_list, but wraps each entry in quotes.

- **as_braced_list**
  Returns the list as comma-separated values in braces.
- **as_braced_list_strings**
  Same as as_braced_list, but wraps each entry in quotes.

4.2.3 Example Variables.txt

The following text would add and modify the two VarLists 'MyNewList' and 'AnotherNewList', as well as defining the list 'YetAnotherNewList'.

```
MyNewList
++10
++40
++20
--5

AnotherNewList
++10
++40
++20
--5

YetAnotherNewList
::0
```

4.3 Build Scripts

4.3.1 Overview

Sometimes, you need more fine control over exactly how your mod files should be built. **Build Scripts** assist in this regard. These are files that tell the mod manager how it should use your mod files. In the Build Script, you can specify which rules you want your mod files to use, specify which order you want multiple files to be built in if they share the same target, and use softcodes with filenames. The sections below will introduce these features.

4.3.2 Basic Build Scripts

The Build Script is a JSON dictionary, where the keys are **File Targets** and the values are a set of **Build Instructions**. A **File Target** is the filepath you want the mod manager to recognise a file as. For example, if you have a txt file located at **myscript.txt** in the **modfiles** folder and you want the mod manager to patch this file into **script64/t0101.txt**, then the **File Target** is **script64/t0101.txt**. A Build Script to do this would look like

```
{
    "script64/t0101.txt": "myscript.txt"
}
```

Note that if you're using the **"FormatVersion": 2** metadata option, these paths will still need to reflect the full path inside the modfiles folder, *e.g.*

```
{
    "DSDBP/script64/t0101.txt": "myscript.txt"
}
```

The right-hand-side element, **"myscript.txt"**, is the **Build Instruction**. Note that this is a contraction of the full syntax, which we will expand on as we explore more features.

We can tell the mod manager to merge a file using a specific **rule**. To do this, we turn the **Build Instruction** into a List and add the **rule** name as the second element of the list. For example, if we wanted to completely replace the vanilla script file instead of merging our new data into it (generally a bad idea!) we would do

```
{
    "script64/t0101.txt": ["myscript.txt", "overwrite"]
}
```

If the rule requires arguments, you can add these as additional elements in this list. If we want to just use the default rule, we can either use the previous syntax as before

```
{
    "script64/t0101.txt": "myscript.txt"
}
```

or simply not specify a rule,

```
{
    "script64/t0101.txt": ["myscript.txt"]
}
```

Maybe now we want to also apply **randomfiles/someedits.sqmod** to **script64/t0101.txt** too, and specifically we want to do this *after* we've merged in the additional data from "myscript.txt". In this case, we can add a build instruction after our current one by wrapping your syntax in another list:

```
{
    "script64/t0101.txt":
    [
        ["myscript.txt"],
        ["randomfiles/someedits.sqmod"]
    ]
}
```

If we want to swap the build order, we just swap these instructions over.

```
{
    "script64/t0101.txt":
    [
        ["randomfiles/someedits.sqmod"],
        ["myscript.txt"]
    ]
}
```

This collection of instructions is referred to as the **Build Steps**, and if we want to we can also wrap our current build steps in a dictionary that explicitly states this:

```
{
    "script64/t0101.txt":
    {
        "BuildSteps":
        [
            ["myscript.txt"],
            ["randomfiles/someedits.sqmod"]
        ]
    }
}
```

This dictionary syntax exists so that we can create **Build Patterns** to build multiple files with a single set of **BuildSteps**, as we will see in section 4.3.3.

The final thing that we shall mention is that we can also use **Softcodes** in the build script, *e.g.*

```
{
    "[Digimon::MyNewMon::filename()].name": "mynewmon.name"
}
```

4.3.3 Building Multiple Files with Build Patterns

You may have many files with similar names that you wish to build with similar Build Steps. In this situation, you can make use of **Build Patterns**. A **Build Pattern** will automatically generate a number of **Build Targets** based on what files it finds inside your mod. An example **Build Pattern** is

```
{
    "{0}[Digimon::MyNewMon::filename()]{1}":
    {
        "BuildSteps": "{0}mynewmon{1}",
        "Variables":
        [
            {"Regex": "(.*)mynewmon(.*)"}
```

```
        ]
    }
}
```

Let's start with the new **Variables** element of the build instructions. This is a list of **Variables** that will generate **Pattern Groups** that are used to generate the **File Targets** the **Build Pattern** will create. The **Variable** that we currently have is a **Regex** variable. This will scan all files in the mod, and look for any files that match the pattern. If a file matches the regex pattern, it will provide a **Pattern Group** for every regex group in the pattern—here we have two regex groups, the two (.*) elements of the pattern. To generate **File Targets**, we use the **Pattern Groups** to create the **File Targets**. The numbers wrapped in braces, *e.g.* **{0}**, are references to the **Pattern Groups**. **Pattern Groups** are created in the order that they are defined by the **Variables**. In this situation, **{0}** refers to the first **(.*)** in the regex pattern, and **{1}** refers to the second **(.*)** in the regex pattern.

Let's clarify this with an example. Let's say we have a mod with the following files:

- cam_mynewmon_bv01.name
- data/charname.mbe/Sheet1.csv
- a_directory/mynewmon.name

First, the Regex pattern will iterate over these files. It will match **cam_mynewmon_bv01.name** and **a_directory/mynewmon.name**. For the first match, the values of the two **Pattern Groups** it finds are **cam_** and **_bv01.name**, and for the second match it will find **a_directory/** and **.name**. For each of these two, the mod manager will then attempt to generate a **File Target**. The **File Target** is **{0}[Digimon::MyNewMon::filename()]{1}**. First, the mod manager will substitute for the softcode function call **[Digimon::MyNewMon::filename()]**, which might evaluate to in **chr001**. Next, the values of the **Pattern Groups** are substituted in. For our first match, this yields **cam_chr001_bv01.name**, and for the second we get **a_directory/chr001.name**. We do something similar for the **Pattern Groups** in the **BuildSteps**. At the end of this, the **Build Script** that uses this **Build Pattern** is equivalent to

```
{
    "cam_chr001_bv01.name": "cam_mynewmon_bv01.name",
    "a_directory/chr001.name": "a_directory/mynewmon.name"
}
```

which might seem a bit like overkill for this situation, but if you mod has many files with similar names, or you want to make a bunch of mods with similar build scripts and only want to update filenames in one place, this can be extremely useful.

Important note: if a **File Target** is created more than once by a **Build Pattern**, or the **File Target** is already built by another element of the **Build Script**, the mod manager will tell you that you've written an invalid build script.

4.3.4 Build Script Variable Types List

- **Regex**
  - Description: A Regex pattern used to identify which files should be used to generate Build Targets.
  - Parameters:
    - pattern: A Regex pattern used to identify which files should be used to generate Build Targets. Pattern Groups are provided by Regex Groups built into the pattern.
- **Range**
  - Description: Generate Build Targets using integers generated in a range.
  - Parameters:
    - start: The integer to start generating the range from.
    - stop: The integer to stop generating the range at.
    - step: The gap between integers in the range.

5 CYMIS

5.1 Overview

CYMIS (CYber sleuth Mod Installation Script) documents are json-based files that can generate customised mod versions from a set of input data. This is achieved by:

- Generating a set of boolean flags, intended to be done *via* an installation wizard;
- Generating the required mod files based on the values of the boolean flags.

The details of how a CYMIS document can be built to satisfy these requirements are given in section 5.2. This section focusses on the concepts the CYMIS format is built upon, rather than the specification details.

5.1.1 Components

A CYMIS implementation is made up of two components — a **wizard** and an **installer**. The job of the **wizard** is to generate a set of booleans. The job of the **installer** is to decide which files to include in the final mod build based on the values of those flags.

5.1.2 Flags and the Flag Table

A CYMIS implementation stores a list of defined flags in a **Flag Table**. These flags are stored under a name and are boolean-valued.
The two parts of a CYMIS implementation – the **wizard** and the **installer** – interact with the Flag Table in the following ways:

- The Wizard **creates** and **modifies the values** of flags;
- The Installer **reads** flags and **decides** what to do based on their values.

5.1.3 Important Files

There are three critical components to a CYMIS-ready mod:

- a “modfiles" folder,
- a “modoptions" folder,
- an INSTALL.json file.

The “modfiles" folder should be **empty**. It is the location in which the mod will be built.
The “modoptions" folder contains the data that the installer can install into the “modfiles" folder, based on the flags set in the CYMIS wizard.
The “INSTALL.json" file contains the CYMIS document. The information that must go into this file is defined in section 5.2.

5.1.4 A Note on Aliases, Build Scripts, and Variables

Currently, CYMIS documents are unable to build **ALIASES.json**, **BUILD.json**, and **VARIABLES.txt**. They will be able to in a future update. You may therefore have to release certain mods as multiple mods until this feature is complete.

5.2 Specification

5.2.1 Top Level

The top level of the CYMIS script must contain the following three labels:

- Version
- Wizard
- Install

“Version" states which CYMIS version the script should be interpreted as. If the script is written to satisfy the stipulations of this document, that version should be **0.1**.
“Wizard" is a list of pages that make up the installation wizard, specified in section 5.2.2. These pages contain boolean flags that are used to determine which mod files are used and/or generated.
“Install" is a list of possible routes the installation process could take, depending on which flags have been set in the wizard. This is specified in section 5.4.

The top level of a CYMIS document will therefore look like the following:

```
{
    "Version": 0.1
    "Wizard": [],
    "Install": []
}
```

5.2.2 Wizard Script

The Wizard is defined as a series of pages. Each page is defined by a dictionary containing the following items:

- Title
- Contents
- Flags

“Title" is the title of a page in the wizard.
“Contents" is the text that is displayed after the title, intended to describe what the current installer page does or what its purpose is. “Flags" is a list of flags that can be set on the page and will be displayed as UI elements. The definitions of these flags are given in section 5.3.

The contents of the “Wizard" item of the CYMIS document will therefore look like the following:

```
...
    "Wizard":
    [
        {
            "Title": "The First Page",
            "Contents": "This is the first page.",
            "Flags": []
        },
        {
            "Title": "The Second Page",
            "Contents": "This is the second page.",
```

```
            "Flags": []
        },
        ...
    ],
...
```

5.3 Flag Types

The CYMIS specification defines a number of flag types that can be used. Each flag type can itself potentially define more than a single flag, depending on the implementation. Each flag type must contain at a minimum a “Type" keyword, which defines which flag type it is, and some way of providing names for flags to be entered into the Flag Table.

5.3.1 Flag

The “Flag" flag type is intended to be implemented as a labelled checkbox, taking True or False values. The specification can contain the following items, with optional values in square brackets:

- Type
- Name
- Description
- [Default]

“Type" should be set to **“Flag"**.
“Name" is the name this flag is referred to by.
“Description" is the label attached to the checkbox.
“Default" defines whether the Flag begins in a True or False state. By default, it is False.

Some example flag definitions are given below.

```
{
    "Type": "Flag",
    "Name": "False Flag 1",
    "Description": "This Flag is False by default."
}
```

```
{
    "Type": "Flag",
    "Name": "False Flag 2",
    "Description": "This Flag is also False by default.",
    "Default": false
}
```

```
{
    "Type": "Flag",
    "Name": "True Flag 1",
    "Description": "This Flag is True by default.",
    "Default": true
}
```

5.3.2 HiddenFlag

The “HiddenFlag" flag type is not intended to be displayed in the UI. It should behave like a flag, but without the necessity for a “Description" item in the specification. The specification can contain the following items, with optional values in square brackets:

- Type
- Name
- [Default]

“Type" should be set to **“HiddenFlag"**.
“Name" is the name this flag is referred to by.
“Default" defines whether the Flag begins in a True or False state. By default, it is False.

Some example flag definitions are given below.

```
{
    "Type": "HiddenFlag",
    "Name": "False HiddenFlag 1",
}
```

```
{
    "Type": "HiddenFlag",
    "Name": "False HiddenFlag 2",
    "Default": false
}
```

```
{
    "Type": "Flag",
    "Name": "True HiddenFlag 1",
    "Default": true
}
```

5.3.3 ChooseOne

The “ChooseOne" flag type is intended to be displayed as a group of mutually exclusive checkable UI elements, such as a list of radio buttons. This flag type requires neither a Name nor Description, since it is a group of flags. However, the flags defined by this flag type should themselves each have a Name and Description. The specification can contain the following items, with optional values in square brackets:

- Type
- Flags
- [Default]

“Type" should be set to **“ChooseOne"**.
“Flags" is a list of flags provided by the ChooseOne group.
“Default" is the name of the flag that is selected at the start. By default, it is the first flag.

Each flag in the “Flags" item should contain the following items:

- Name
- Description

“Name" is the name this flag is referred to by.
“Description" is the label attached to the radio button.

```
{
    "Type": "ChooseOne",
    "Flags":
    [
        {
            "Name": "ExampleFlag1",
            "Description": "This is the first option."
        },
        {
            "Name": "ExampleFlag2",
            "Description": "This is the second option."
        },
        {
            "Name": "ExampleFlag3",
            "Description": "This is the third option."
        }
    ],
    "Default": "ExampleFlag2"
}
```

5.4 Install Script

The Install section of the CYMIS document describes how to build the data in the “modfiles" folder, given a set of possible build options and flags stating which paths should be followed. This follows a simple if-then format, where a single flag – or combinations thereof – are entered into the “if" section, and the build option is defined in the “then" section. Note that the “if" element is optional, and can be omitted if a certain build path should always be followed.

An “if" statement can be built from nested single-entry dictionaries that resolve down to flags and condition operations. The most basic “if" statement is a single flag:

```
"if": "Flag Name"
```

A more complicated “if" statement will contain a rule and an argument to be passed to the rule:

```
"if": {<rule name>: <rule argument>}
```

For example, checking if multiple flags are true:

```
"if": {"and": ["Flag 1", "Flag 2"]}
```

or if one flag is true and another is false:

```
"if": {"and": ["Flag 1", {"not": "Flag 2"}]}
```

Complex “if" statements can be built up in this manner. Any flag name can be replaced with a rule and argument in order to create arbitrarily deeply-nested conditions.

Build options, on the other hand, are simply a list of rules-and-arguments. The rules that can be used in a build option are detailed in section 5.6.
The contents of the “Install" item of the CYMIS document will therefore look like the following:

```
...
    "Install":
    [
        {
            "if": <condition>,
            "then": []
        },
        {
            "if": <condition>,
            "then": []
        },
        {
            "then": []
        },
        ...
    ],
...
```

5.5 Condition Rules

In the following section, the word “condition" is intended to mean either a **flag name** or a **condition rule plus its argument**, since both are valid values to pass to a condition rule.

5.5.1 and

The “and" rule takes a **list** of conditions and returns True if **all** the contained conditions are true.

```
"if": {"and": [<condition 1>, <condition 2>, <condition 3>, ..., <condition N>]}
```

5.5.2 or

The “or" rule takes a **list** of conditions and returns True if **any** the contained conditions are true.

```
"if": {"or": [<condition 1>, <condition 2>, <condition 3>, ..., <condition N>]}
```

5.5.3 not

The “not" rule takes a **single** condition and returns the opposite of the value of that condition.

```
"if": {"not": <condition>}
```

5.6 Installation Rules

Installation rules are individual entries in a “build path" that can be executed. They consist of a name for the rule, and a list of named arguments that get passed to the rule. Rules do not have a common set of arguments; each rule defines its own set. The only common factor between rule specifications is the “rule" keyword, which defines which rule is being selected.

5.6.1 copy

Copies an item from a location within the mod into the “modfiles" folder of a mod. The “copy" rule has two arguments:

- source
- destination

“source" is the item within the “modoptions" folder of a mod to be copied. It can be either a file or a directory. The path is separated by forward-slashes.
“destination" is the location within the “modfiles" folder of a mod the source is to be copied to. It can be either a file or a directory. The path is separated by forward-slashes.

An example use of the rule to copy a file at “modoptions/scripts/common_scripts.txt" to “modfiles/script64/t3004.txt" is given below.

```
{"rule": "copy": "source": "scripts/common_scripts.txt", "destination": "script64/t300
    4.txt"}
```