Documentation for Pokémon Battler

Edition one

Braedon Wooding

2015

Contents

[Manual Main Game 2](#_Toc431293979)

[Manual Coding 3](#_Toc431293980)

[Introduction 3](#_Toc431293981)

[How to code? 3](#_Toc431293982)

[Different Systems 3](#_Toc431293983)

[For example 3](#_Toc431293984)

# Manual Main Game

# Manual Coding

## Introduction

Coding is very, very simple and easy in Pokémon Builder Application due to it being just database editing. You can just copy the examples and edit to your needs or you can start from scratch. The coding aspect of the Pokémon Builder Application follows an argument based coding system in which you have elements called ‘values’ that can determine values using certain different ‘signs’ (for example getting the current ally Pokémon’s HP and then adding it to the current enemy Pokémon’s speed). This is your main resource in the coding aspect of the Pokémon Builder Application.

## What can you code?

Large aspects of the game were ‘coded’ in this new language taking its roots from XML and ‘argument based’ coding systems. You can code new functions for the computer to carry out such as applying a move’s effect or helping the AI to make the best decision. The entire AI core is built in a function based system using this new found language nicknamed: Project XML Coding. All the data (units, moves, items and other resources) are all loaded in and made using the Project XML Coding.

## How to code?

The coding is done in any xml editor or text editor program. There are quite a few really good ones with different interfaces. Some of the best are XML Notepad, Notepad++ and others. Each of the programs feature some sort of GUI that makes xml editing very simple and easy.

## Different Systems

To make coding more simple and easy the XML system is split up into multiple different sections. You have database editing, function creating and info editing.

### Database Editing

The easiest out of all the sections but the most tedious. Essentially you create a list of different units, moves, items and other resources that are available to the player and you can edit the already existing ones or you can create your own set. To go along with these resources you can also put in sprites and other images to use in your items, units and other resources.

#### Pokémon Database Editing

Each xml document that follows the ruleset of the Pokémon database editing system has a few rules and values it requires. Each ‘script’ can contain more than one Pokémon and is suggested that you put all your own creations under a single script. As usual don’t edit the standard ‘default’ since if the game is updated or something gets corrupted it will reset the default files, rather create your own and just turn of the default under the mods menu and turn on your own scripts.

As with all of this section each of the below values will be displayed with their name first then a colon then the type of value that is inserted then a brief description (I.E. Pokemon\_Types : Enum.PokemonType[] = The types of the Pokémon [I.E. Fire and Water; Poison, Steel and Grass; You can have how many types you want but it is suggested you have 1-2]).

There are quite a few values that you have access to:

* Pokemon\_Name : String = The Pokémon’s name that is displayed (I.E. Charmander).
* Pokemon\_Sprite\_Name : String = The name of the sprite or image that the Pokémon uses to display (I.E. Pokemon\_05).
* Pokemon\_Speed : Integer = The speed stat determines who goes first each turn as well as some other aspects.
* Pokemon\_MaxHP : Integer = The maximum hitpoints of the Pokémon. Once your current HP reaches 0 that Pokémon is dead.
* Pokemon\_Types : Enum.PokemonType[] = The types of the Pokémon (I.E. Fire and Water; Poison, Steel and Grass; You can have how many types you want but it is suggested you have 1-2). The different types that you have access to are located in the reference section.
* Pokemon\_Move\_Names : String[] = While you can have any number of moves you should stick to 1-4 with most having 4 due to errors if you have a larger amount than 4. Each value should just contain the name of the move. Moves can be made/edited using the Move Database Editing system explained in the section below.
* Pokemon\_Statuses : Enum.PokemonStatus [] = Same as Pokemon\_Types but you state the status effects the pokemon is under. This should be blank and left empty unless you want to start the pokemon with a ‘handicap’ or you add some effect that gives the pokemon a bonus to attack or whatever. The values are an enum and you can find the different values under the references section.

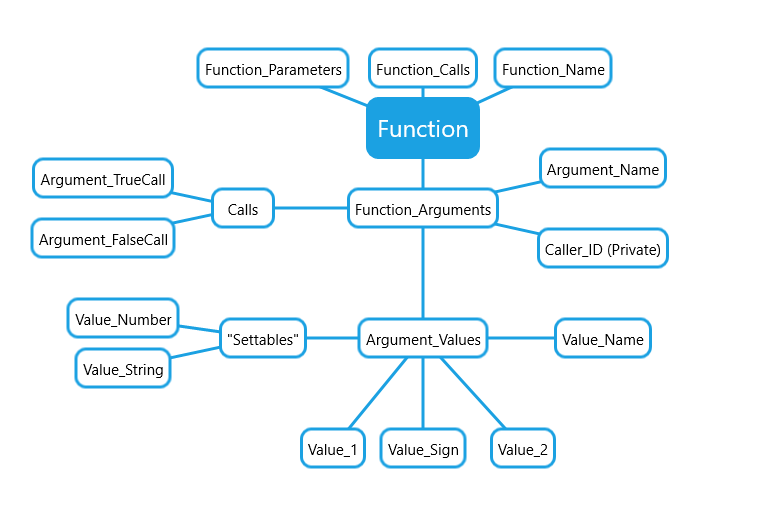
#### Move Database Editing

#### Item Database Editing

#### Sprite Inputting

### Function Creating

Function creating is the most impactful way for you to ‘mod’ the game. You can create various functions to complete tasks, such as applying the effects from a move or a status effect each turn or rather just increasing the AI’s ability to make decisions by making their ‘AI-Core’ improved. The syntax is rather simple but the concept is a little more complex to understand. Below is an image of how the mapping of the XML looks if drawn (it is suggested that you draw out your ideas before starting).



To help you understand below will be a ‘legend of sorts’:

* Function : Function = The handler of the function. It handles what argument to call and has a list of all the arguments and previous values as well as a dictionary (list of values). One by one it calls an argument (starting at 0) then calculates the value of each of that arguments values, till it reaches a Boolean value and then it will go to the next argument of value (Argument\_TrueCall or Argument\_FalseCall) dependent on the Boolean value. If the Argument Call it receives is negative then it exits the program but just before it calls the function from the ‘function\_Calls’ array corresponding to the negative number without the ‘negative’ value (-1 = Function\_Calls[1]).
* Function\_Name : String = Just the name of the function (This is just for developers to understand what a function does without checking the info.xml, No use in game).
* Function\_Calls : String[] = Dependent on negative values from arguments corresponding to a function call from this array (-1 = Function\_Calls[1]).
* Function\_Parameters : Object[] = XML doesn’t really like the Object class and often it’s better to avoid using parameters due to the computer not knowing if a value is of xType or yType (I.E. it doesn’t know if your ‘1’ is a float or integer). Still you can use it for some functions especially if they want strings.
* Function\_Arguments : Arguments are the hardest to understand since they are arbitrary, an argument always evaluates to a Boolean value (True/False). They will continue to go through their value array till they reach a Boolean value. The Argument\_Values array will be iterated through number basis (1, 2, 3, 4, 5 etc) and there is no way to change this. However if you want to just have an argument to carry out some actions (-10 hp, double speed so on), then you can just state a value that will always be true at the end to state you are ending the value. Such as NONE == NONE, which always evaluates to true.

### Info Editing

The info editing is easier than any database editing but isn’t used in game except to state what folders contain mods. You need this file for the system to recognise that that folder is a mod folder. In it you have access to a few values:

* Version\_Number : Float = This is what version your ‘mod’ is at and is just displayed so users can know if they have the most current version. Just a nice ‘quality of life’ feature.
* Mod\_Runnable : String = This is only used for Function Creating and states what the function is called that it should run. Essentially as stated in more detail in the learning syntax/reference section the system has a few ‘commands’ it requires such as AI\_Choose\_Action in which it calls the AI to choose an action. For each of these actions you can only have one mod that uses this runnable active at a time so if you are incorporating your own ‘function scripts’ then you need to make sure they don’t conflict. Also don’t name your ‘Function\_Name’ (under the function script) this runnable since that will confuse players and maybe you.
* Mod\_Name : String = Just the mod name to display. Players will see this as your mod name so name it something descriptive.
* Mod\_Active : Bool = If your mod is just for yourself you may want to set this to true since it determines if the game is going to use its value. If you are putting your mod online for other to use set Mod\_Active to false, since it will not force the mod to run and allow the user to choose. Even if you turn this on by default anyone can turn it off either in game or in the xml document.

## BAXML Syntax

### Introduction

Coding languages share a common thread with languages in the ‘real world’. This common thread is an idea of a syntax; a ruleset in which languages are distinguished by. The syntax for Project XML Coding (or BAXML – Beginners All-Purpose Extensible Mark-up Language) is simple and follows the rules of other coding languages such as JavaScript or C (and its predecessors). The system uses an enum based coding in which your choices are restricted to an enum (an enum is a datatype that consists of a set of named values called elements). There are two enums in BAXML; Enum.Signs and Enum.Values. The BAXML syntax is technically only used when ‘Function Creating’ and editing the ‘Value’ type (as stated in the Function Creating section). However creating ‘Values’ are what function creating is all about so knowing the BAXML syntax somewhat decently is required to be able to code a function at a ‘generic’ level.

### Enum.Signs

Enum.Signs is the simplest out of the two enums and it just tells the System (Computer) how to apply the two Enum.Values together (I.E. the ‘+’ sign tells the System to add the two values together). Enum.Signs has a few different categories below are the rules to distinguish the categories.

* Booleans: True or False Statement.
  + Any sign that if read with an If in the sentence it still makes sense. (I.E. X == Y is read as If X == Y).
  + Note almost all Booleans contain 2+ characters (Except ‘<’ and ‘>’) and they always contain at least one ‘=’ sign or one ‘>’ / ‘<’ sign (or both).
* Temporary Value Changer: They don’t change the ‘original’ value but still output a new value.
  + They don’t contain an ‘=’ sign and they are just a single character that is widely known for operation (I.E. ‘+’ or ‘\*’ are Temporary Value Changers (TempValue) whereas ‘+=’, ‘()’ and ‘NONE’ aren’t.
  + Note ‘~’ isn’t a TempValue and rather a ‘lookup’. Also ‘<’ and ‘>’ aren’t TempValues either.
* Value Changer: They change the ‘original’ value and output this new ‘original’ value.
  + They always contain a TempValue and then followed by a ‘=’ sign.
  + The only exception to this rule is the ‘Percentage Equal’ Value Changer which is ‘%/=’ since it Divides the two values then Times by 100 (as if you were to find the percentage of the two). This is just a shortcut and can be done normally under two values (I.E. Value 1: A /= B, 2: A \*= <Number> N.B. Number = 100).
* Special Values: Three signs each with a different meaning.
  + Don’t fit into any other category.
  + The lookup sign ‘~’: Tells the system that you are going to do a lookup (search through the database). For example: the command; ![CurrentAllyPokemon] ~ !P:[move0]. That tells the system that you want the Current Ally Pokemon and that you want to look up its first move (move0).
  + The command sign ‘()’: Tells the system you are executing a command and that you are going to use parameters if they are required (the second value is the parameter, you can put NULL/NONE/Run () if the command doesn’t require parameters).
  + NONE: Shouldn’t use this value ever, but it doesn’t touch the numbers whatsoever, no cases in which you may want to use this.

There are a few rules that are related to Enum.Signs.

* You can’t use any Value Changers on normal numbers (10 += 10 or 10 + 10)