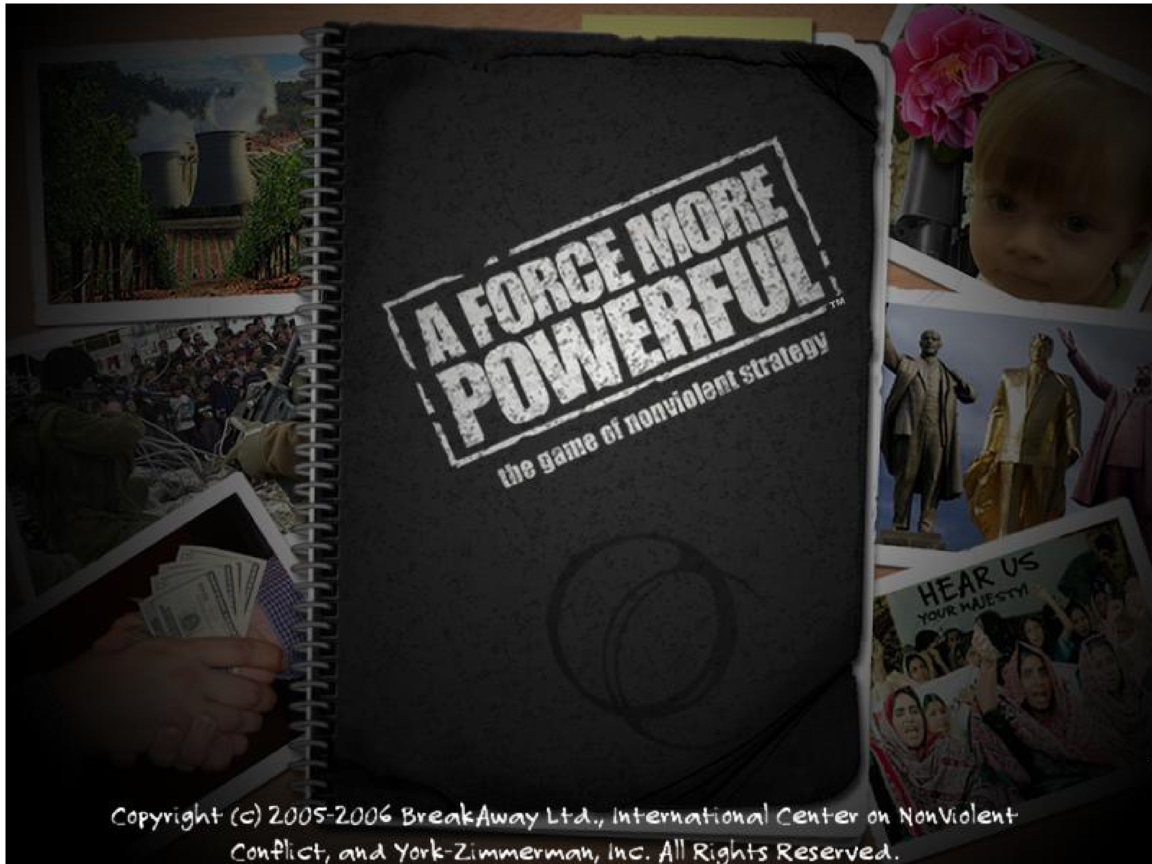
A Force More Powerful™



Scripting Guide

Version 1.0.0.5

1. INTRODUCTION 3
1.1 Script Color-Coding 3
2. SCRIPTS IN AFMP 4
2.1 Victory Condition & Objective Scripts 4
2.2 The AFMP.LUA Script 6
2.3 Events & Elections 10
3. CONCLUSION 12
4. APPENDIX 13
4.1 The AFMP Script Function Library 13
4.1.1 Introduction 13
4.1.2 Basic Functions 13
4.1.3 Travel Time 21
4.1.4 Difficulty Modifiers 22
4.1.5 The GlobalChange function 22
4.1.6 Victory Condition & Objective Handling 23
4.1.7 Handling Variables 24
4.1.8 Timers 25
4.1.9 Changing The AI Mode 26
4.1.10 Miscellaneous 26
4.2 Scenario-Editor Slider Values 28

IMPORTANT!!

Although *A Force More Powerful* aims to impart the values and techniques of strategic thinking for individuals participating in (or leading) nonviolent Movements against oppressive opponents, ***it does not guarantee real-world results***. Life is very complicated and circumstances in the real world are difficult (even in the best of times) to anticipate and can change rapidly. Thus, no training tool can simulate these circumstances fully. Please keep this in mind when playing *A Force More Powerful*, and know that the results you receive during play may not be the same results you receive if (and when) you apply these methods to a "real" Movement.

IMPORTANT UNINSTALL NOTE!

When you uninstall the game, you will lose all files and folders *except*:

- The "Save" folder if it contains any saved-game files;
- Any *new* folder under the current "Scenarios" folder;
- Any *new* folder under the current "CityTiles" folder;
- All files in these (only) folders.

Files saved in any folder created when the game was installed will be lost.

TECHNICAL SUPPORT

For technical assistance with *A Force More Powerful*, please go to the technical support forum at www.afmpgame.com.

© 2005-2006 BreakAway Ltd., International Center on Nonviolent Conflict, and York Zimmerman Inc. All Rights Reserved.

1. INTRODUCTION

The AFMP Scripting Guide explains how to create basic scripts for AFMP scenarios.

AFMP's scripts are in Lua, a simple scripting language. It is *strongly* recommended that users get a Lua editor to work in when creating Lua scripts for AFMP. A very good one is *B:Lua*, available at http://blua.sourceforge.net/; all of the scripts for AFMP were created using *B:Lua*, and they may be hard to read if B:Lua is not used to open and work with them. This guide will only briefly deal with Lua basics such as local variables, if-then statements, and functions—the best place to learn about Lua from the start is the Lua homepage at www.lua.org.

In particular, the sections on if-then statements (e.g., http://www.lua.org/pil/4.3.1.html) are particularly relevant. There is also a good resource at www.lua-users.org.

1.1 Script Color-Coding

An asterisk (*) below indicates that this color is used only inside the sample text boxes.

Brown: function names

Blue: numbers that the user enters

Dark Blue: Lua keywords*

Green: strings that the user enters*

Orange: logical operators*

Gray: comments*

2. SCRIPTS IN AFMP

Each scenario has three types of scripts associated with it:

1. **AFMP.LUA**—Every scenario requires exactly one AFMP.LUA script. We have provided an example AFMP.LUA script in your AFMP\Scripts directory, and an empty one in the AFMP\Editors directory.
2. **VICTORYCONDITIONX.LUA**—Every scenario requires *at least* one VictoryCondition script. They are numbered, with the number appearing right before the .LUA extension (e.g., VICTORYCONDITION1.LUA, VICTORYCONDITION2.LUA, etc.).
3. **OBJECTIVEX.LUA**—Objective scripts are optional, although they add a lot of flavor and opportunities for the player to create different strategies on successive play-throughs of the same scenario. They are numbered as well, in the same way VictoryCondition scripts are numbered.

2.1 Victory Condition & Objective Scripts

Here is an example of a VictoryCondition script (it is VICTORYCONDITION2.LUA from *The Weight of Tradition*):

```
--Constants

REGIME = 0;
MOVEMENT = 1;
UNALIGNED = 2;

function VictoryCondition()


-- Integrate Premier Shops

-- Must get Premier Shops Owners preferred policy on Ethnic Discrimination to >6 and their
-- Fear lower than 4

        if ((AFMP:GetPolicyPref("Premier Shops Owners", "group", "Ethnic Discrimination") > 6 ) and
        (AFMP:GetFear("Premier Shops Owners") < 4 ) ) then

                AFMP:CompleteVictoryCondition("Integrate Premier Shops");

        end

        return 0;
end

function Reward()

        AFMP:GlobalChange( "Gilradi Rights", "all", "enthusiasm", 1);
        AFMP:GlobalChange( "Gilradi Rights", "all", "fear", -1);
        AFMP:GlobalChange( "unaligned", "all", "fear", -1);

        return 0;
end
```

The script has three parts. The first is the definition of constants. Some of the AFMP Lua functions prefer to deal with numbers instead of words such as “REGIME” and “MOVEMENT”. Therefore, *these lines must appear at the top of every AFMP script*:

```
--Constants

REGIME = 0;
MOVEMENT = 1;
UNALIGNED = 2;
```

The second part is the definition of the Victory Condition: what triggers it, and the function that tells AFMP to mark it as completed:

```
function VictoryCondition()


-- Integrate Premier Shops

-- Must get Premier Shops Owners preferred policy on Ethnic Discrimination to >6 AND their fear to <
-- 4

       if ((AFMP:GetPolicyPref("Premier Shops Owners", "group", "Ethnic Discrimination") > 6 ) and
       (AFMP:GetFear("Premier Shops Owners") < 4 ) ) then

              AFMP:CompleteVictoryCondition("Integrate Premier Shops");

       end

       return 0;
end
```

This part of the Victory Condition always starts with "`function VictoryCondition`" and ends with the word "end", as above. Inside it, there is an "if-then" statement. Here, you can see that the `GetPolicyPref` function is used to check whether the Victory Condition should be triggered. (All functions in these examples are explained in the Appendix; refer there as necessary when using this Guide.) In this case, the Victory Condition is triggered if the Ethnic Discrimination policy preference of the Premier Shops Owners is more than 6, and their Fear is less than 4. (These numbers refer to the increments on the sliders in the Scenario Editor; see Section 4.2.) If you wanted to create a Victory Condition that checked these same numbers, you would copy everything between and including the "if" and the "then", and then change the name of the Group, the name of the policy preference, and the number to suit what you wanted. You could also remove the "and" and everything after it if you wanted to get rid of the check on the Group's Fear level. *B:Lua* will help you match up the parentheses properly. The statement at the end reading "`return 0`" is just an output statement. Since the VictoryCondition function's output is never checked—what matters is the other scripting contained in it—the return statement is just a formality. Your own scripts' return statements should exactly parallel the ones in the existing scenario scripts.

By far the best way to create your own Victory Conditions is to look at the scripts that already exist and copy over the sections they have like the one above—the part within the "`function VictoryCondition`" block.

The last part is the Reward section, which defines what happens when the player accomplishes the Victory Condition or Objective (note: he only gets the reward for an Objective if he completes it in the phase to which it was assigned).

```
function Reward()

        AFMP:GlobalChange( "Gilradi Rights", "all", "enthusiasm", 1);
        AFMP:GlobalChange( "Gilradi Rights", "all", "fear", -1);
        AFMP:GlobalChange( "unaligned", "all", "fear", -1);

        return 0;
end
```

Most Reward functions in AFMP use the `GlobalChange` function (see the Appendix for a description of how to use it).

Objectives use the same basic structure as Victory Conditions, except that their main block starts with "`function Objective`" and they use the `CompleteObjective` function instead of the `CompleteVictoryCondition` function.

2.2 The AFMP.LUA Script

The AFMP.LUA script is arguably the most important script for an AFMP scenario. *Every scenario must have one*. It runs before every game turn, and has four parts. The first part is the "Constants" section, which is just like the one described under "Victory Conditions and Objectives".

The second part is the main game script, which looks like this (this example is from the AFMP.LUA script for *We're Done With the War*):

```
function LuaMain()
        local err;
        err = 0;
        if (AFMP:CheckEventTimer("Election warning") == 1 ) then

                AFMP:DisplayMessage("ELECTIONWARN");

        end
        if (AFMP:CheckEventTimer("Legislative elections") == 1) then
        -- resolve local elections

                local BroadwaterResult = AFMP:HoldElectionRegion("Broadwater");
                local GrovefieldResult = AFMP:HoldElectionRegion("Grovefield");
                local CascadeResult = AFMP:HoldElectionRegion("Cascade");

                local OverallResult = BroadwaterResult + GrovefieldResult + CascadeResult;

                if (OverallResult > 1) then

                        AFMP:SetVariable("ELECTIONWIN", true);

                else

                        AFMP:DisplayMessage("ELECTIONLOSE");

                end
        end

        -- End the script
        return err;
end
```

The most important thing to remember about this script, called "`LuaMain`", is that it runs *every single game day*. That is why you see a lot of "if-then" statements in the `LuaMain` sections in our scripts: usually it doesn't make sense for an event to happen every day. (It does make sense to *check* every day to see if an event ought to happen, hence the if-then statements.)

In this example, `LuaMain` does the following things:

1. It sets a local variable called "err" and then sets it to zero. This should always be the first part of a `LuaMain` script, but you never have to change it—you can just copy and paste it and forget about it.
2. It checks an Event Timer (see Appendix for details about creating and checking Event Timers) to see if it has gone off, and if it has, displays a Scenario Message called "ELECTIONWARN".
3. It checks another Event Timer and, if it has gone off, runs a series of local elections using the `HoldElectionRegion` function, then checks to see how many Regions were won by the Movement. If the Movement won 2 of the 3 Regions, then a variable called "ELECTIONWIN" is set to "true" to signify that the player has won the elections. Otherwise, a message called "ELECTIONLOSE" is displayed.
4. It outputs the value of "err", which is used internally by the game. This line should always be the last line in a `LuaMain` function (apart from the "end" which closes off the function).

This is a good example of a simple election event. Each Region is tallied and then the outcome of the overall election is determined by who won more Regions (so it is akin to the U.S. electoral system, except that in this example all the Regions are equal regardless of population). To replicate this sort of event in your own scenario, you would want to copy all the lines from the first "if" to the *second to last* "end", inclusive, and paste them into your own AFMP.LUA script, and then change the names of the Regions to match the ones in your scenario. You would then want to add copies of the lines with the `HoldElectionRegion` function if your scenario has more Regions than this example, or take some out if fewer. You would also want to make sure that the timers are properly set—this is usually done in the "Start" section, which we look at now.

```
function Start()  -- Run Once at start of game

	AFMP:CreateVariable("ELECTIONWIN", "bool");
	AFMP:SetVariable("ELECTIONWIN", false);

	AFMP:SetEventTimer("Legislative elections", 365);

	AFMP:SetEventTimer("Election warning", 275);

	return 0;

end
```

This part of the AFMP.LUA script is just like the other, with the very important exception that it is only run once, at the very beginning of the game. Here is where global variables—variables that the game will need to "remember" from day to day—are created and initialized (given a starting value). Here is where timers are usually set as well. In this example, we see one variable that the game has to

remember, called “ELECTIONWIN”, and two timers. One is called “Legislative elections” and is set to go off one year (365 days) after the scenario starts; the other is called “Election warning” and goes off 275 days after the scenario starts. In the earlier example—the `LuaMain` function part—you can see that the timers are checked every turn (since the `LuaMain` function runs every turn) and certain actions are taken if they have gone off (for example, if the “Election warning” timer has gone off, the game simply displays a message informing the player of the upcoming elections).

Another part of the AFMP.LUA script is the “Regime Collapse” section:

```
function RegimeCollapse() -- Run when Regime base points are less than collapse
-- threshold

      AFMP:DisplayMessage("REGIMECOLLAPSE");
      AFMP:EndGame();

      return 0;

end
```

This function only runs when the Regime collapses (recall that the Regime collapses when the total number of Base Points it has from its Groups is lower than the Collapse Threshold). In this case, nothing happens except that the game ends (`AFMP:EndGame()`) and a message is displayed (called “REGIMECOLLAPSE”). Remember that if this message key—“REGIMECOLLAPSE”—does not appear in your scenario’s SCENARIOMESSAGES.CSV file, the game will generate an error (this is true of all messages that you tell the game to display using the scripting system—they all must have their keys in that file).

In this section you can do more elaborate things such as checking Groups’ support levels and statistics to see if the Regime collapsing is a good or a bad thing. For example, in the *Bringing Down a Dictator* scenario:

```
function RegimeCollapse()
-- Run when Regime base points are less than collapse threshold
      if ( (AFMP:GetSupportLevel("Planicies Infantry Division", 1, MOVEMENT)) > 3 and
      ( AFMP:GetSupportLevel("Montanha Infantry Division", 1, MOVEMENT) > 3 and
            ( AFMP:GetFear("Planicies Infantry Division") < 4 ) and
            ( AFMP:GetFear("Montanha Infantry Division") < 4 ) ) ) then
            AFMP:DisplayMessage("NOCOUP");
            AFMP:CompleteVictoryCondition("Collapse Regime without military coup");
            else
                  AFMP:DisplayMessage("COUP");
            end
      AFMP:EndGame();
      return 0;
end
```

Here, we see something more complicated. The script is checking the two army divisions' Legitimacy support levels for the Movement, as well as their Fear. If the support levels are low, or if Fear is too high, then the army divisions will mount a coup. One of the Victory Conditions for this scenario is to overthrow Kosanic, the dictator, without a coup, so that Victory Condition is completed only if the conditions for the army to stay out of the political struggle are met. Otherwise, a "Coup" message appears and the Victory Condition is not flagged as completed.

You will always want to have a `DisplayMessage` line in your `RegimeCollapse` function. You may not necessarily want more than that—often, the text of that message can convey what you want the player to know when the government collapses. The only time `RegimeCollapse` has to be more involved is when you want there to be multiple possible Regime collapses, as with the above example (coup vs. no coup).

The last part of the AFMP.LUA script is the `EndGame` function. It is very important to recognize the difference between the `AFMP:EndGame` function, which appears in the RegimeCollapse example below, and the simple EndGame function (note the lack of an "AFMP:" before its name). The `AFMP:EndGame` function simply tells the game to run the simple `EndGame` function, then stops the game. Here is an example of a simple `EndGame` function:

```
function EndGame() -- Run once at end of game
	if (AFMP:GetPoliticalOrgBasic("Gilradi Rights", "numViolentTactics") == 0)
then
		AFMP:CompleteVictoryCondition("Avoid the use of violence");
	end
	return 0;
end
```

This is the `EndGame` function from *The Weight of Tradition*. One of the Victory Conditions in that scenario is for the player to avoid the use of violence. Thus, when the game is over, this function runs, and checks to see how many violent Tactics have been performed by the Gilradi Rights Alliance (the Movement). If that number is zero, then the condition is met, and the Victory Condition is flagged as Complete. If not, then it will *not* be flagged as Complete, and will show up as Failure in the Evaluation Screen.

Usually the `EndGame` function is very simple and can be left empty or almost empty.

2.3 Events & Elections

The most important thing to learn about events in AFMP is how to make sure they don't repeat when you don't want them to. One way to do this, as explained before, is through the use of timers—since timers go off only once by default, an event controlled by a timer is guaranteed only to happen once.

Sometimes, though, you'll want an event whose timing is not certain—for example, the election in *Paradise Regained* isn't on a set schedule, but is only called when a particular condition is met:

```
if ( ( RegimeStability < 10 ) and (AFMP:GetVariable("ELECTIONSCALLED") == false )
) then

            AFMP:SetVariable("ELECTIONSCALLED", true);

            AFMP:DisplayMessage("ELECTIONSSOON");

            AFMP:SetEventTimer("Election", 30)

            -- Tell player if fear is high

            if (AFMP:GetFear( "West Talu") > 3 ) then
                  AFMP:DisplayMessage("WESTTALUFEAR");

            end

            if (AFMP:GetFear("Navilli Inner City") > 3 ) then
                  AFMP:DisplayMessage("NAVILLIFEAR");

            end
```

(Note that this example won't work if directly copy/pasted, because it's showing only the first part of this election routine—hence it's missing an "end" statement.)

The condition is whether the `RegimeStability` variable is less than 10 and the elections haven't already been called. If the condition is met, notice that the very first thing that happens is that the "ELECTIONSCALLED" variable is set to "true", so the next time through this routine, the condition will always not be met and everything inside the if-statement will be skipped. It is worth carefully looking over the Appendix section on "Variables" to learn exactly how they work and how they can be set up.

Note that the `AFMP:GetVariable` function is not used to get the value of `RegimeStability`. That is because `RegimeStability` is a local variable, calculated fresh every day, with its value forgotten from day to day. The `GetVariable, SetVariable`, and `CreateVariable` functions all create variables whose values are *remembered* by the game from day to day.

Another interesting thing about this function is that it sets a timer—obviously the player needs some warning time for the election, so it sets a timer for 30 days and displays a message to that effect, and then also provides an additional message if Fear is high (and Fear matters a lot in the election in

Paradise Regained). A separate section of the script will check to see if the timer has gone off, and if it has, run the actual election routine.

Other events work just like elections—in general, you will want to create at least one "control variable", which is devoted to controlling whether the event takes place. (The example above actually has two control variables: `RegimeStability` and ELECTIONSCALLED.)

Here is another example of an event (not an election):

```
    local AgitationTemp = AFMP:GetVariable("LASTAGITATION");

    AgitationTemp = AgitationTemp + 1;

    AFMP:SetVariable("LASTAGITATION", AgitationTemp );

    local AgitationVar = AFMP:GetRand(0,100);

    if ( AFMP:GetPolicyPref("Tim Murray", "character", "Luthanian
Independence") == 10 ) and
       (AgitationVar < 5 ) and
        (AFMP:GetVariable("LASTAGITATION") > 30 ) then

          AFMP:DisplayMessage("TIMAGITATION");

          AFMP:IncPolOrgMoney("Borosovan Occupation", -50 );

          AFMP:SetVariable("LASTAGITATION", 0 );

    end
```

This is taken from the `LuaMain` section of the AFMP.LUA script file for *Unwelcome Guests*. Note the several lines that begin with "local". In those lines, local variables—variables that are "forgotten" by the game every turn—are declared and given values. In this example, a local variable called `AgitationTemp` is being declared, then given a value equal to the game variable LASTAGITATION, then having 1 added to itself. Then, LASTAGITATION is being set to AgitationTemp's new value. The net result of the first three lines here is that LASTAGITATION's value is being increased by one.

In the fourth line, another local variable, `AgitationVar`, is being declared and then given a value equal to a random number between 0 and 100. Then, an event is triggered if the Tim Murray Character's policy preference on Luthanian Independence is equal to 10 AND the `AgitationVar`'s value (the one with the random number) is less than 5 *and* the value of LASTAGITATION is more than 30. Thus, this event has three control variables—one based on Tim Murray's policy preference information, one based on a random chance, and one based on LASTAGITATION.

LASTAGITATION increases by one every game day. Therefore, its purpose is to make sure this event doesn't happen too often (note that if the event occurs, LASTAGITATION becomes zero, so it can't happen again for at least another game month).

3. CONCLUSION

By far the best way to get scripts for AFMP up and running quickly is to copy over the ones from the scenarios that come with the game, and adapt and tweak them to match your custom scenario's needs. We have built in a good error-checking system that, when you test your scenarios, will tell you when an error has occurred in a Lua script. If that happens, don't panic—just read the explanation in the error message. Often, that will tell you exactly what's wrong. If it doesn't, go to the line that it refers to and make sure that:

1. the name of the function you're trying to use *exactly* matches a function listed in the Appendix;
2. you are putting in the correct number of parameters, in the correct format;
3. you have enough "end" statements to match up with all of your "if" statements

If checking these possible sources of error doesn't work, you can also get help with your scripts in the AFMP support forums; see www.afmpgame.com.

We wish you the best of luck with custom AFMP scenarios and scripts, and hope this Guide has been of some help.

4. APPENDIX

4.1 The AFMP Script Function Library

4.1.1 Introduction

This appendix lists the script functions supported by the AFMP game engine. You can use any of these functions in your AFMP.LUA, Victory Condition, and objective scripts.

To use a function in a script, remember that it must have the prefix "`AFMP:`" (including the colon) before its name. Each line of your script should have a semicolon at the end of the whole line (this is actually not required in the Lua language, but it helps a lot to make your scripts more readable).

The part of a function that is in parentheses is its *parameter list*. Some functions have empty parameter lists, e.g.:

```
AFMP:EndGame();
```

Others have one, two, three, or more parameters, e.g.:

```
AFMP:ChangeBuildingBasic("WBRK Radio Building", "IsDestroyed", true);
```

It is very important that each parameter be entered (or "passed in") in the correct order, with quotation marks when required. The description of each function will make it clear when you need to use quotation marks. (In general, you need them most of the time, unless you are passing in a value like a number or a true/false value.) Remember also that all names and values are case sensitive.

Tip: "IMF" stands for "Interactive Map Feature", which comprises buildings on city maps and all national-map icons.

4.1.2 Basic Functions

`ChangeBuildingBasic(name, attribute, payload)`

Use this function to change an IMF's Notes field, its destroyed status or its Economic Significance value.

PARAMETER #1: The IMF's name, in quotation marks.

PARAMETER #2: The name of what you want to change, in quotation marks. This must be either `"Description"` (if you want to change the Notes field), `"IsDestroyed"` (if you want to change whether the IMF has been destroyed), or `"EconomicSig"` (if you want to change the IMF's economic significance).

PARAMETER #3: What you want to change the thing named in parameter #2 to become. If you put in `"Description"`, this should be the text you want to appear in the Notes field (this will overwrite what's already there!), in quotation marks. If you put in `"IsDestroyed"`, this should be either `true` or `false` (no quotation marks!), depending on whether you want the building to be destroyed or not. If you put in

"EconomicSig", this should be a whole number from 0 to 10 (no quotation marks!), matching the new value for Economic Significance you want.

GetBuildingBasic(name, attribute)

This function simply returns (gives as output) the value of a building's Economic Significance, Destroyed status, or Description. It works just like the ChangeBuildingBasic function, except that Parameter #3 is missing (because you're not changing anything). Note that nothing will be done with the value unless you store it somewhere, like a local variable, so the best way to use this function is something like:

```
local DestroyedVar = AFMP:GetBuildingBasic("WBRK Radio Building", "IsDestroyed");
```

This will create a local variable called DestroyedVar, then store the value of WBRK Radio Building's destroyed status (which can be true or false) in that local variable. Then you can use it in an if-statement to trigger an event, or whatever you want.

Example:

```
local DestroyedVar = AFMP:GetBuildingBasic("WBRK Radio Building", "IsDestroyed");

if (DestroyedVar == false) then
        AFMP:ChangeBuildingBasic("WBRK Radio Building", "IsDestroyed", "true");

end
```

GetPoliticalOrgBasic(name, attribute)

This function lets you get and store the Notes, and several other attributes of an Alliance.

PARAMETER #1: The name of the Alliance you are interested in, in quotation marks.

PARAMETER #2: The name of the value you are interested in. This can be:

"Description": The Notes field associated with the Alliance

"Acronym": The Alliance's acronym

"IsGovernment": This has a value of true if the Alliance is the Regime, and false if it isn't.

"DomesticOpinionSensitivity", "InternOpinionSensitivity", "ViolenceTolerance", "ViolenceUnwillingness", "CommEfficiency", "CommSecurity", "MessageEffectiveness", "Decisionmakingstyle", "numViolentTactics": Each of these returns a number representing the Alliance's value for that attribute. "numViolentTactics" just returns the number of violent Tactics that Alliance has undertaken. "MessageEffectiveness" returns the Message Quality for the Alliance, reflecting how well that Alliance has done using the Craft Message tactic. "Decisionmakingstyle" returns a whole number from 0 to 4, where 0 is autocratic and 4 is consensus (and the others are in between).

Example:

```
local RegimeAcronym = AFMP:GetPoliticalOrgBasic("Kosanic Regime", "Acronym");
```

ChangeGroupBasic(name, attribute, payload)
GetGroupBasic(name, attribute)

These functions work just like their counterparts for Buildings (GetBuildingBasic and ChangeBuildingBasic), but are for Groups. The order of the parameters is the same, but what is allowed for Parameter #2 is different, as follows. Again, if using GetGroupBasic, you don't need Parameter #3.

"ShortName"

"Acronym"

"Description"

"ViolenceUnwillingness"

"ViolenceTolerance"

"GenPopInfluence"

"CivicDuty"

"EconWellBeing"

"Resources_Money": This refers to how much money per week the Group will give to its Alliance.

"Resources_Manpower": This refers to how many People per week the Group will give to its Alliance.

"RegimeBase": This refers to the number of Regime Base Points the Group contributes to the Regime's staying in power.

"TrainingOppSupport": This refers to the Support & Logistics training level for the Group.

"TrainingOppDirect": This refers to the Nonviolent Intervention training level for the Group.

"TrainingRegSecurity": This refers to the Repression training level for the Group.

"TrainingRegPolitical": This refers to the Political training level for the Group.

"Enthusiasm"

"Fear"

"Infiltrated": This is a true/false value; if true for a Regime Group, then the Group is infiltrated by the Movement. If true for a Movement Group, then the Group is infiltrated by the Regime. (Note that an infiltrated Regime Group, should it switch to the Movement side, will still be infiltrated, but the infiltrator will have no effect unless the Group switches back to the Regime later.)

"IsPillarOfSupport": This is a true/false value indicating whether the Group is a Pillar of Support for the Regime.

GetNeighborhoodBasic(name, attribute)
ChangeNeighborhoodBasic(name, attribute, payload)

These two functions will let you retrieve and change (respectively) the Enthusiasm, Fear, Economic Well-Being, Health, Literacy, and Percent of Population of a Neighborhood (only).

PARAMETER #1: The name of the Neighborhood you want to retrieve a value for, or whose value(s) you want to change, in quotation marks.

PARAMETER #2:

"Enthusiasm"

"Fear"

"EconomicWellBeing"

"Health"

"Literacy"

"PercentOfPopulation": This can be a decimal number.

PARAMETER #3 (ChangeNeighborhoodBasic only): Just put the new value you want in for Parameter #3. This will always be a whole number between 0 and 10, except for PercentOfPopulation which can be a decimal number between 0 and 100 (including 100 but not zero).

GetRegionBasic(name, attribute)

This function works just like GetNeighborhoodBasic, except that it only works on Regions (on the national map).

SetRegionSupport(name, alliance, attribute)
SetNeighborhoodSupport(name, alliance, attribute)

These two functions let you change the Support value for a Region and Neighborhood (respectively).

PARAMETER #1: The name of the Region or Neighborhood whose Support you want to alter. Use quotes!

PARAMETER #2: The name of the Alliance for whom the Support is to be changed. Use quotes!

PARAMETER #3: The new value for the Support. Must be a whole number from 0 to 10.

Example:

```
AFMP:SetRegionSupport("Great Talu", "Republic of Talutia", 2)
```

This command would set the support for Republic of Talutia in the Great Talu Region to 2.

SetRegionHealth(name, value)
SetRegionLiteracy(name, value)
SetRegionUnemployment(name, value)
SetRegionEconomicWellBeing(name, value)
SetRegionFear(name, value)
SetRegionEnthusiasm(name, value)
SetRegionPercentPopulation(name, value)

These functions mirror how ChangeNeighborhoodBasic works, but there is a separate function for each value you might want to change. The parameters work the same for each of them:

PARAMETER #1: The name of the Region whose value you want to change.

PARAMETER #2: The new value for the parameter (this will always be a whole number from 0 to 10, except for PercentPopulation which can be a decimal between 0 and 100).

Example:

```
AFMP:SetRegionPercentPopulation("Sodania", 7.8);
```

This command would set the Percent of Population in the Sodania Region to 7.8. Note that when using this parameter, you will end up with less than 100 as your total population in the scenario! You'll need to use this function several times to make sure that other Regions "take up the slack".

GetVictoryConditionBasic(text, attribute)

This function can be used to determine whether a Victory Condition has been completed, how highly the player prioritized it, and how highly the designer prioritized it.

PARAMETER #1: The name of the Victory Condition (the name that appears in the Strategic Estimate's Victory Condition list—this is entered in the editor).

PARAMETER #2: Must be one of the three following:

`"completed"`: If this is put it for parameter #2, then the function will return `true` if the Victory Condition has been completed, and `false` if not.

`"playerPriority"`: Returns the number representing the priority that the player assigned this Victory Condition in the Strategic Estimate (`0` if he didn't select it).

`"designerPriority"`: Same as above, but returns the priority the designer assigned this Victory Condition in the scenario editor.

Example:

```
PlayerDidSomething = AFMP:GetVictoryConditionBasic("Execute 2 successful Mass Protests",
"completed");
```

This would retrieve the true/false value representing whether the player has completed the Victory Condition "Execute 2 successful Mass Protests", and then store it in a local variable called "PlayerDidSomething".

SupportRegimePointsFromCollapse()

This function returns the number of Base Points that the Regime must lose before it collapses (so if the Collapse Threshold is 10, and the Regime currently has 15 base points, this function would return "5").

RegimeBasePoints()

This function returns the number of base points the Regime currently has.

SetStatusQuo(name, value)

This function changes the Status Quo policy setting to a specified value.

PARAMETER #1: The name of the policy you want to change. Use quotes!

PARAMETER #2: The value you want the policy in parameter #1 to have.

Example:

```
AFMP:SetStatusQuo("Corruption", 0)
```

This would change the Status Quo value for the Corruption policy to `0`.

Float GetVoteShare(name)

This function runs the AFMP internal election algorithm and tells you what the overall vote share for the whole country is, for a particular Alliance.

PARAMETER #1: The name of the Alliance (e.g., `"Kosanic Regime"`) that you want to know the vote share for.

CharacterIncSupportLevel(name, allianceName, SupportLevelType, amount)

This function changes one of the Support Levels for a Character in the game.

PARAMETER #1: The name of the Character whose support level you want to change.

PARAMETER #2: The name of the Alliance the Character in Parameter #1 is going to be supporting less or more.

PARAMETER #3: The type of support you want to change. Must be a number as follows:

1. Legitimacy
2. Ideology (note: changing this may have a very brief effect, since Ideology support is automatically recalculated every day. If you want to change a Character's Ideology support for an Alliance, it is best to change his/her policy preferences—see the `SetPolicyPref` function for more details).
3. Financial Gain
4. Whichever support level is currently the highest

PARAMETER #4: The amount by which you want the support level to change. Can be negative. Should never be more than `10` or less than `-10`.

Example:

```
AFMP:CharacterIncSupportLevel("Bob Namaramac", "BreakAway Games", "financialgain", 5);
```

This would increase the Bob Namaramac character's Financial Gain support for the BreakAway Games alliance by 5.

GroupIncSupportLevel(name, allianceName, SupportLevelType, amount)

This function works exactly like the `CharacterIncSupportLevel` function, except that Parameter #1 must be a Group's name rather than a Character's.

GroupGetSupportLevel(name, allianceName, SupportLevelType)
CharacterGetSupportLevel(name, allianceName, SupportLevelType)

These two functions work just like the previous two, except that they don't change the support level you pass in; instead, they just return it, so you can store it in a variable and use it for something else (like an if-then statement). The parameters are the same and work the same, except that Parameter #4 is not used.

GetPolicyPref(name, groupOrChar, policyName)
SetPolicyPref(name, groupOrChar, policyName, value)

These two functions are used to retrieve and set a policy preference for a Character or Group.

PARAMETER #1: The name of the Group or Character whose policy preference you want to retrieve or change.

PARAMETER #2: This should be `"group"` if the name in Parameter #1 is that of a Group, `"character"` if it is that of a Character.

PARAMETER #3: The name of the policy whose value you want to retrieve or change.

PARAMETER #4 (`SetPolicyPref` only): The new value you want for the policy preference.

Example:

```
AFMP:SetPolicyPref("Robert Namaramac", "character", "Corruption", 0)
```

This line would change the Robert Namaramac Character's preference on Corruption to `0`.

IncPolicyPref(name, groupOrChar, policyName, value)

This function works just like the previous two, except that Parameter #4 works slightly differently. Instead of changing the Character or Group's policy preference to a specified number, this function uses Parameter #4 to MODIFY the existing preference. So, if you want to increase or decrease Robert Namaramac's policy preference, you'd use this. The game will obey the rules about the possible ranges (so it's okay if you try to lower someone's Policy Preference below `0` or raise it above `10`; the game will just use `0` or `10` respectively).

Example:

```
AFMP:IncPolicyPref("Robert Namaramac", "character", "Corruption", -3)
```

This would lower Robert Namaramac's Corruption policy preference by 3.

IncPolOrgMoney(name, amount)

Use this function to give or take away money from an Alliance.

PARAMETER #1: The name of the Alliance to receive or lose money.

PARAMETER #2: The amount of money to give (or take away, if negative) from the Alliance named in Parameter #1. The game won't let an Alliance have negative money.

GetCharacterStatus(string charname)

This function is a way to determine the status of a Character.

PARAMETER #1: The Character's name, in quotes.

NOTE: This function returns an integer number. Each possible status is represented by a different number. Note that the "worse" statuses are the higher numbers; this means you can use a single if-then statement to see if a Character has a given status "or worse".

NONE = `0`

AVAILABLE = `1`

BUSY = `2`

TRAVEL = `3`

DETAINED = `4`

HOUSE_ARRESTED = `5`

ARRESTED = `6`

MISSING = `7`

OUT_OF_COUNTRY = `8`

MISSING_AND_DEAD = `9`

DEAD = `10`

`GetGroupAllegiance(string groupname, int AllianceID)`

This is a very useful function that will return `1` if the Group named in Parameter #1 is a member of the Alliance identified in Parameter #2, or `0` if not..

PARAMETER #1: The text name of the Group you are interested in, in quotes.

PARAMETER #2: The ID number of the Alliance you are interested in knowing whether the Group in Parameter #1 is a member of. The ID numbers should be used as below (if you have a third or fourth alliance in your scenario, those ID numbers will match their ordering in the list of Alliances in the scenario editor):

Regime = `0`

Movement = `1`

`GetCharacterAllegiance(string charactername, int AllianceID)`

This is like `GetGroupAllegiance` above, and uses the same rules and the same numbers for Parameter #2. Parameter #1 is the name of a Character rather than a Group (in quotes).

`GetStatusQuoPolicy(string PolicyName)`

This function lets you determine the current value of a Status Quo policy (very useful for triggering events, or for changing the game state without relying on the AI to do so “naturally”).

PARAMETER #1: The name of the Status Quo policy whose current level you want to know.

`GetEnthusiasm(string unitname)`

This function returns the Enthusiasm level of any Character or Group.

PARAMETER #1: The name of the Character or Group whose enthusiasm level you are interested in.

`GetFear(string unitname)`

This function returns the Fear level of any Character or Group.

PARAMETER #1: The name of the Character or Group whose fear level you are interested in.

`GetSupportLevel(string UnitName, int supportlevel, int AllianceID)`

This is a very useful function that lets you find out a particular unit’s Support Level for a particular Alliance. This works on Neighborhoods and Regions too.

PARAMETER #1: The name of a Character, Group, Neighborhood, or Region (use quotes!).

PARAMETER #2: The ID number of the support type you want (note that this will be treated as `1` no matter what if the name in Parameter #1 is a Neighborhood or Region):

Overall Support = `0`

Legitimacy Support = `1`

Ideology Support = `2`

FinancialGain Support = `3`

PARAMETER #3: The ID number of the Alliance the Support Level is for. Use `0` for the Regime, `1` for the Movement. If you have a third or fourth alliance in your scenario, their ID numbers will match their ordering in the list of Alliances in the scenario editor.

4.1.3 Travel Time

These three functions let you change the travel times on the national map in various ways. They are very good for modeling the effects of weather or security crackdowns on travel.

ChangeRegionTravelTime(regionName, payload, duration)

This function simply changes travel times for any routes terminating at IMFs in a given Region, for a given time.

PARAMETER #1: The name of the Region whose travel times you want to affect. Any routes that terminate at IMFs located in this Region will have their travel times altered when this function is used.

PARAMETER #2: The value by which you want to alter the travel time. This can be a decimal number and it can be negative (so you can decrease or increase the travel times).

PARAMETER #3: The number of days you want this change to last.

Example:

```
AFMP:ChangeRegionTravelTime("Sodania", -1, 4);
```

This use of the function would reduce all travel times for routes that have at least one endpoint in Sodania by one day; the change would persist for 4 days.

ChangeIMFTravelTime(IMFName, payload, duration)

This function is very similar to the previous one. Parameters #2 and #3 work exactly the same way. Parameter #1 is the name of an IMF instead of a Region, and the change in travel times will apply to any routes that have an endpoint at that IMF.

ChangeSingleRouteTravelTime(ConnectionName, payload, duration)

This function lets you change the travel time associated with a single route. Parameter #1 is just the name of the route (the name you gave the route when you created it in the national map editor). The other two parameters work the same way as the previous two functions.

4.1.4 Difficulty Modifiers

ChangeDifficultyLocal(int *payload*, int *duration*, string *tactictype*, string *placename*)

This function lets you make Tactics easier or harder based on their target. This is a good way to model politically or culturally important places: you can make certain Tactics easier when they are located at a particular building or IMF, representing that place's political or cultural significance.

PARAMETER #1: The amount by which you want to change the difficulty of the Tactics at the place you want to affect. If this number is negative, then Tactics will be easier; if positive, then they will be harder. A value of `1` or `-1` will have a small effect on difficulty; a value of `20` or `-20` will have a huge effect on difficulty.

PARAMETER #2: The number of days you want the alteration in difficulty to last.

PARAMETER #3: The type of Tactics to be affected by the change. This parameter must be: `"ATTACK"`, `"DEFEND"`, `"COMMUNICATE"`, `"BUILD STRENGTH"`, `"DENY"`, or `"ALL"`. You must use quotation marks!

PARAMETER #4: The name of the place that, if targeted by Tactics matching the type in Parameter #3, will have those Tactics' difficulty altered. This can be an IMF, Region, or Neighborhood. If you use a Region or Neighborhood, then any IMFs and Buildings in those areas will also be affected by the function. Use quote marks!

Example:

```
AFMP:ChangeDifficultyLocal( -5, 30, "ATTACK", "Star Square");
```

This will reduce the difficulty of all Attack Tactics targeted at Star Square by 5, for 30 days.

ChangeDifficultyGlobal(int payload, int duration, string tactictype)

This function works just like the previous one, but does not have Parameter #4 (the place name). That is because this function will alter the difficulty for all Tactics matching the type in Parameter #3 *regardless of target*. This is a very powerful function! It is a good way to model things like cultural events and holidays (for example you might want to make all Tactics harder to do on Mardi Gras, since everyone is out partying).

4.1.5 The GlobalChange function

GlobalChange(name, target, type, value, targetAlc)

This is a very important function. It is often used in the `Reward()` functions in `VictoryCondition` and `Objective` scripts. It is used to change Enthusiasm, Fear, or Support Levels for large numbers of Characters and Groups.

PARAMETER #1: The name of the Alliance to which the Characters or Groups whose values are going to be changed belong. Can also be `"unaligned"`, meaning that unaligned Groups or Characters can have their values changed as well.

PARAMETER #2: The type of units whose values are to be changed. This should be "groups", "characters", or "all". If "groups", then only Groups belonging to the Alliance in Parameter #1 will have their values changed. If "characters", then only Characters. If "all", then all members—Groups and Characters both—will have their values changed by the function.

PARAMETER #3: The type of value to be changed. Can be one of the following:

"fear": Changes the Fear value for the units described by Parameters #1 and #2.

"enthusiasm": Changes the enthusiasm value for the units described by Parameters #1 and #2.

"legitimacy": Changes the Legitimacy Support of each of the units described by Parameters #1 and #2. If this option is chosen for Parameter #3, then Parameter #5 must also be filled in, so the function will know which Legitimacy support value to change.

"financialgain": As above, but for Financial Gain support.

PARAMETER #4: The amount by which to change the value selected in Parameter #3. Can be negative.

PARAMETER #5: The Alliance for whom Legitimacy or Financial Gain support is to be changed.

Example:

```
AFMP:GlobalChange("unaligned", "all", "legitimacy", -3, "Kosanic Regime")
```

This will lower the Legitimacy support for the Kosanic Regime by 3 among all unaligned Characters and Groups.

Example:

```
AFMP:GlobalChange("Kosanic Regime", "characters", "financialgain", -2, "Democracy Now")
```

This will lower the Financial Gain support for Democracy Now by 2 among all Characters who belong to the Kosanic Regime.

Important: Make sure you get the order of these parameters right, as well as making sure that every name is correctly spelled! It is best to use copy and paste from the scenario editor so you are 100% sure the names in your script match the ones in your scenario file.

4.1.6 Victory Condition & Objective Handling

Important: In the game, Objectives are checked every game day *only if they are selected by the player, and only during the Phase the player assigned them to*. Victory Conditions, conversely, are checked every game day, provided they were selected. So, if there are script lines that need to be run every day no matter what, they should not go in a Victory Condition or Objective script. They should go in AFMP.LUA instead.

CompleteVictoryCondition(string VictoryConditionName)

Use this function in your VictoryCondition scripts to set the Victory Condition to "completed" status. Victory Conditions default to "failed".

PARAMETER #1: The exact name of the Victory Condition that you want to be set to "completed" status. It is best to just copy and paste the name from the scenario editor into your script, to avoid errors.

CompleteObjective(string ObjectiveName)

Use this function in your `Objective` scripts to set the Objective to `"completed"` status. Victory Conditions default to `"failed"`.

PARAMETER #1: The exact name of the Victory Condition that you want to be set to `"completed"` status. It is best to just copy and paste the name from the scenario editor into your script, to avoid errors.

4.1.7 Handling Variables

Important: This section deals with creating *global* variables—variables that are used from day to day, that the game engine "remembers". Local variables in Lua are very easy to create and use, but the game will discard them and their values at the end of each game day.

Tip: To create a local variable in Lua, just type the name of variable you want to create and assign it a value, like this:

```
LocalVariable = 42;
AnotherLocalVariable = AFMP:GetVariable("SOMEGLOBALVARIABLE");
LocalStringVariable = "Breakaway";
```

Sometimes, in the scenario scripts, the keyword "local" precedes variable declarations like those above. Ignore that; you don't have to use the "local" keyword when creating AFMP scripts.

CreateVariable(name, type)

This function creates new variables that are remembered from day to day by the game engine.

PARAMETER #1: The name of the variable you want to create, in quotation marks.

PARAMETER #2: The type of the variable. This can be either `"string"`, `"int"`, or `"bool"`. Use `"string"` if you want the variable to store a word or sentence. Use `"int"` if you want it to store a number. Use `"bool"` if you want it to store a `true` or `false` value. By far the most common use is `"bool"`. Remember to include the quotes!

Tip: In the AFMP scripts, the names of variables created with CreateVariable are always ALL CAPS. However, we also name messages that are used with **`DisplayMessage`** the same way. The best way to make sure you don't confuse them is to look at the Start section of each AFMP.LUA you are working with and note the list of variables (if any) that are created at scenario start.

SetVariable(name, value)

This function lets you store a value in a variable that you've previously created. Make sure that you don't try to store a string (e.g., `"A Force More Powerful"`) in a variable that only stores numbers.

PARAMETER #1: The name of the variable whose value you want to set.

PARAMETER #2: The value you want to set the variable to.

GetVariable(name)

This function retrieves a value previously stored in a variable. Just pass in the name of the variable whose value you want to access.

Example:

```
AFMP:CreateVariable("TEST", "bool");
AFMP:SetVariable("TEST", false);
LocalVar = AFMP:GetVariable("TEST");
```

This code fragment would create a variable that stores true/false values and name it "TEST", and set its value to false. Then, a local variable called "LocalVar" is created, and the value of "TEST" is retrieved and stored in it.

RemoveVariable(name)

This function lets you get rid of a variable you've already created. Just pass in the name of the variable to remove. This is usually unnecessary, though, unless you have created a *lot* of variables.

FlushVariable()

This function just uses the `RemoveVariable` function on all your variables.

4.1.8 Timers

CheckEventTimer(string TimerName)

Use this function to check whether an existing timer has gone off yet. It will return a value of `1` if the timer has gone off, and `0` if it hasn't.

PARAMETER #1: The name of the timer you want to check

SetEventTimer(string TimerName, int numDays, int repeatable(optional))

Use this function to create a new timer.

PARAMETER #1: The name the new timer is to have.

PARAMETER #2: The number of days to elapse before the timer goes off.

PARAMETER #3: Must be either `1` or `0`. If you put a value of `1` here, the timer will go off *every day* after the number of days in Parameter #2 has elapsed. If you put a value of `0` here, the timer will only go off once. The default is `0`; if you don't bother putting in Parameter #3 (as in most examples in the game), it will automatically be `0`.

4.1.9 Changing The AI Mode

`SetAI(name)`

This function lets you change the AI weighting table. There is a file in your AFMP "Scenarios" directory called AITACTICWEIGHTS.CSV. Its rows represent the Tactics in the game; the columns represent different AI styles. Each column header is a name that describes the AI style. Each other cell contains a number representing how important the Tactic in its row is for the style in its column. The higher the weight, the more likely the AI mode described by that column is to attempt the given Tactic.

> *Example:*
>
> The "EasyDoesIt" AI mode has a value of 0.25 for the Arrest tactic, whereas the Standard mode has a value of 1.0 for the Arrest tactic. Thus the Standard mode is four times more likely than EasyDoesIt to attempt the Arrest tactic. (The default weight is 1.)

PARAMETER #1: The name of the column representing the AI style you want the AI to adopt.

Tip: Remember that you should use *CSVed*, a free program used to edit comma-separated data files, when editing AITACTICWEIGHTS.CSV or any other .CSV file provided with *A Force More Powerful*.

4.1.10 Miscellaneous

`EndGame()`

Use this function to end the game after the current day is over. No parameters are needed.

`DisplayMessage(string MessageTag)`

Use this function to display a custom scenario message.

Parameter #1: The name of the message you want to display.

Important: The name *must exactly match* one of the names in the first column of SCENARIOMESSAGES.CSV. Otherwise the game will crash!

Tip: The SCENARIOMESSAGES.CSV file should be in all scenario folders (see the Editors Guide). It has several columns. The first one is the most important in terms of game stability: it contains the message's name (which is not the same as the message text or summary).

`AddTacticToQueue(string TacticName, string CharacterName, string TargetName, int option, int execdays, int prepdays)`

Use this function to force a Character to do a particular Tactic. The Tactic will go at the END of the Character's current Tactics Queue, so if the player or AI has already scheduled some Tactics for the Character, the Tactic added here will be attempted after all of those previously scheduled ones are resolved. This function is very useful for modeling things like out-of-control violent elements within a Movement, independent (unaligned) Groups that nonetheless undertake Tactics, and so forth. Examples of its use can be found in *The Weight of Tradition* and *Unwelcome Guests*. Note that Characters who are

in Alliances can have this function used on them; if that's the case, then they'll use their Alliance's money and people to try the Tactic! If they are unaligned, the Tactic is free.

PARAMETER #1: The name of the Tactic you want done, in quotes.

PARAMETER #2: The name of the Character who will coordinate the Tactic.

PARAMETER #3: The name of the Target. (If the Tactic does not require a target, put in `"NONE"`.)

PARAMETER #4: The number of the Option to be used. If the Tactic has no options, enter `0`. Otherwise, the numbers correspond to the order the Options appear in the Tactics interface (so `0` would be the first option on the list, `1` would be the second, and so forth).

PARAMETER #5: The number of execution days you want the Tactic to have (if this is not variable, then put in `0`).

PARAMETER #6: The number of preparation days you want the Tactic to have (if this is not variable, then put in `0`).

Example:

```
AFMP:AddTacticToQueue("Destroy Property", "Mad Max", "Movement HQ", 0,0,10);
```

This line would find the character "Mad Max" and add a Destroy Property Tactic to the end of his current tactics queue, targeted at the Movement HQ IMF. It would have 10 days of prep time and use the default value for execution time (which is not variable for this Tactic). The Tactic has no options, so `0` is entered for parameter #4.

GetRand(lowRange, highRange)
GetRand()

Use these functions to generate random numbers.

PARAMETER #1: The lower bound of the range you want the number to be within. This bound is inclusive.

PARAMETER #2: The upper bound of the range you want the number to be within. This bound is exclusive.

If you leave out both parameters, you'll get a totally random number, with no upper or lower limit.

Example:

```
local num = AFMP:GetRand(0,20);
```

This will generate a random number between `0` and `19`—the number can't be 20 because the upper bound is exclusive.

4.2 Scenario-Editor Slider Values

The 0–10 numbers used in the various functions represent the values set in the relevant slider controls in the Scenario Editor (e.g., for Policy Preferences). See the illustration below for a representation of how the values relate to the labels and colors.

Note too that when you place the cursor over a slider's pointer in the Scenario Editor, a "tooltip" showing the current value will appear. For example, this slider is set to a value of `2` ("No/none" would be `0`, and "Unrestricted" would be `10`):