Programming in AFWSL

This will give you an idea on how to program in AFWSL. Certain major things will be covered, like messages, how to script out proper logic, adding people to towns, adding treasure chests, and shop creation. General rule: always run your script files under an emulator first, because if you script out the storyline wrong, you could cause an infinite loop or crash the calculator.

The storyline is embedded in an array that is 100x5 in size. Each row is one line of code. This was the easiest way to create scripting capabilities without creating an XML type engine. That would take up way too much coding. This is the fastest, easiest, and smallest way I could create an engine that could do "scripting."

Proper Header and ending creation.

There are 3 different types of storyline files: Shop, Town, and everything else. In order to decide which type of storyline file it is, every storyline file is started (and sometimes ended) with a numeric code to designate the file to a certain type.

For the shops, the starting number of the array must be 500. Shops do not need ending numbers. For Towns, the beginning number must be 100. The ending number must be 101. For everything else, the top number must be 2000. These storylines do not need ending numbers. Let's look at an example:

{2000,0,0,0,0}, - this is the proper header for all other storylines
{0,0,0,0,0}, - leave for a buffer, b/c it starts at 1
{warp,1,27,1,2}, - code
… lots of code...

{500,0,0,0,0}, -signals for the shop storyline
{0,0,0,0,0},// buffer
{501,64,20,200,503},
… lots of code...

{100,0,0,0,0}, - signals for the town storyline.
{0,0,0,0,0},// buffer
{add_sprite,11,13,guy,1},
… lots of code...
{101,0,0,0,0} – end of town storyline

If you notice, there is always a buffer after the designating code. This is there, because sometimes the storyline does not execute properly if you lack the buffer. So keep the buffer there to make sure the storyline executes properly. Also, only use one type of storyline per file. You could attain unexpected results if you do not. Besides, afw only accepts one type at a time.

Each map can have multiple storylines attached to it. However, there are 3 that are always pulled. They are: a storyline for a generic script, a storyline for initializing a town, and then a storyline for

initializing the extended warping system. It goes in that order, so you need to be careful of what you add to your world maps. As each one is pulled, the previous is erased. Keep this in mind. Also, up to 7 different scripts can be pulled when the 2nd key is pressed. Look at the 2nd key section for more information.

Logic creation:

This will teach you how to properly create and use the logic in AFWSL. In normal logic, you have "If blah==x, then do something." However, in AFWSL, there are a set of checking (chk) tags that will check various things and then return either true or false. Then there are if_true and if_false statements you put after the chk tags to execute the correct code depending on whether the chk tag returned true or false.

The "if" statements must be carefully created. The following is NOT stated in the AFWSL syntax guide, so make sure you pay attention here. Each "if" statement is linked to its own chk statement via a "linker". This prevents weird errors in the end. The linker is set up by using an ID in the 4th column of every chk statement and every if statement (including end ifs). Let's look at an example.

```
{chk_story_marker,2,0,0,1},
{if_true,0,0,0,1},
{set_message,1,4,34,0},
{set_message,2,4,36,0},
{end_if_true,0,0,0,1},
{if_false,0,0,0,1},
{end_if_false,0,0,0,1},

{chk_story_marker,2,1,0,2},
{if_true,0,0,0,2},
{set_message,1,4,34,0},
{set_message,2,4,36,0},
{end_if_true,0,0,0,2},
{if_false,0,0,0,2},
{end_if_false,0,0,0,2},
```

Look at the highlighted numbers. Each set of these numbers correspond to one set of if statements and one chk statement. This is how these things should be coded. As you use more logic, you need to use different ID numbers, otherwise, you could run into problems. General Rule: always use a different ID number for the chk and if statements.

Let's look at some examples of how to write logic the wrong way. In red will be the things, which are wrong.

```
{chk_story_marker,2,0,0,1},
{if_true,0,0,0,1},
{set_message,1,4,34,0},
{set_message,2,4,36,0},
```

{end_if_true,0,0,0,1},
The if_false statements were left off. You must always have both if_ture and if_false even if there is no code for both.
{chk_story_marker,2,1,0,1}, - you reused the same ID linker number.
{if_false,0,0,0,2},
{end_if_false,0,0,0,2},
{if_true,0,0,0,0}, - you forgot to add in the ID linker number
{set_message,1,4,34,0},
{set_message,2,4,36,0},
{end_if_true,0,0,0,2},

This is what will happen with the above errors: in the event that the first chk statement is false, then none of the rest of the code will be executed, because it will search through all of the array in attempt to find the corresponding (or linked) if_false statement. If the first chk statement is true, then the code will be executed. However, in both cases, nothing else will execute. Now if there is code after the 2nd if statement, then it will execute in the event it is not surrounded by an if statement.

Messages HowTo:

Messages have 2 different phases to them when you call them. Either they can wait for the 2nd button to be pressed (ie pause) or they can skip the button and continue on. If you decide to skip the pause then any code below the called message will be executed. This may cause problems if you do not allow time for people to read the message.

There exists reasons why you would want to skip the pause. The reason is for the asker tag. The asker tag will pull up a yes / no box below the message. The yes / no box will pause and wait for a choice to be made. If you skip the pause on the message and immediately call the asker, then the message will be displayed along with the yes / now box. However, if you decide to pause the message, then the yes / no box will not happen until after you press the 2nd key.

The pausing for messages is defined in the last argument when using the msg tag. The last argument has 1 of 2 phases: TRUE and FALSE. If true, then the message will pause. If false, then it will skip the pausing and continue down the code.

Here is an example of how to properly use FALSE:

{msg, 2, 64,FALSE,0} - call the message, but do not pause
{asker,0,0,0,1} - call the asker
{if_true,0,0,0,1} – if it is true then execute the code until you hit the end if; however, if it is false then find the if_false flag.
…
{end_if_true,0,0,0,1}
{if_false,0,0,0,1}
…
{end_if_false,0,0,0,1}

Both of the if_true and if_false must be there, otherwise you will run into problems. **Note: the asker is considered a chk statement.**

Here is an example on how to call a message and make it pause (ie wait for the 2nd key to be pressed)

```
{msg, 2, 64,TRUE,0}
```

Creating a Town Script:

When a map first loads, it will load a script. In this script, you will have the ability to do all sorts of things. When creating a town, the thing you want to do is to load all the people into the town and what they will say. To do this, you will have to use the add_sprite and add_message tags. The following code shows you how you should add the towns people with messages.

```
{100,0,0,0,0},
{0,0,0,0,0},
{add_sprite,46,35,sage,1}, - this will load the sage at (46,35) and give him the ID of 1
{add_sprite,6,35,fighter,2},
{add_sprite,6,6,guard,3},
{add_sprite,46,6,lady,4},
...
{set_message,1,4,34,0}, - this will give the message #34 out of the mess4 file to the towns person who
{set_message,2,4,36,0},       has the ID #1
{set_message,3,4,17,0}, - the rest of these have the same concept.
{set_message,4,4,20,0},
…
{101,0,0,0,0}};
```

Keep in mind that you can only add up to 60 towns people to the map. The ID #'s range from 0→59. If you notice, at the very fist line, there is {100,0,0,0,0}. This tells AFW what kind of storyline file it is. 100 stands for a town storyline. This exists, so the correct routines in AFW are called when any storyline file is loaded. 101 is at the end, because this signals the end of a town storyline.

If you want, you can add logic to the town script and make the messages change, based upon certain circumstances. See story5.h in the storyline editor for an example.

Keep in mind that a generic loading script is pulled before this one and a warper script is pulled after the town script. You can call the first script and have it execute and do its thing. After that, you can load the town script. The town script will place all of the initialized towns people into a separate array along with the messages they are supposed to say. This will erase the generic script when it happens, because when any other script is loaded, the previous is erased. If you need a warper script, then go ahead and create it. It will be called after the town script. Although the town script is erased, the mobs will still be there, because they are loaded into a separate array.

The Beginning Script:

When you choose new game, a script is called. This is script ID #2. In this script, you will initialize your character(s), set the starting map and coordinates, and begin the introduction to the game. The introduction should be a scene where you have people talking about the current situation. You should know how RPGs start. You just need to skip the CG and go straight to the talking.

During the initialization, you will need to do a few things in order to add the players. The very first character you add you will have to use the following tags: add_char, inc_max_hp, inc_max_mana, inc_str, inc_dex, inc_int, inc_def, set_level, add_ability, inc_skill, set_tnl. These tags will initialize the stats of the character. After you initialize your first character, you can use the "auto" ability for the add_char. If you decide to use the add_char's auto ability without having a character initialized, then you will run into stats of zero and / or divide by zero errors, which will crash your calculator. You have been warned.

After you have initialized your character(s), then you need to use the life tag to bring their hp out of 0. Then at the very end, do a full heal. This will fill up everyone's hp and mana. Now you are ready to go. Now you can use the set_map to set up your scene. Keep in mind that when set_map is pulled, so are the initialization storyline files with it. Below is an example of a beginning script.

```
{2000,0,0,0,0},
{0,0,0,0,0} – buffer.
{add_char,nick,0,0,0},// add a char
{inc_max_hp,nick,30,0,0},// set his hp to 30
{inc_max_mana,nick,30,0,0},// set his mana to 30
{inc_str,nick,5,0,0},//set str to 5

{inc_dex,nick,5,0,0},//set dex to 5
{inc_int,nick,5,0,0},// set int to 5
{inc_def,nick,5,0,0},//set def to 5
{set_level,nick,1,0,0},// set nick to level 1
{life,nick,30,0,0},// this will bring nick to life. all chars will start with 0 hp in the beg, so you will
// need to call this method to make sure they are alive.

{add_ability,nick,h2hc,0,0},// add hand to hand combat.
{inc_skill,nick,h2hc,45,0},// increase the skill to 45%
{full_heal,0,0,0,0},//full heal
{set_tnl,nick,35,0,0},// set the tnl to 35
{0,0,0,0,0},

{0,0,0,0,0},
{set_map,67,3,4,20},// set the map
{9,3,0,TRUE,0},// first message
```

Keep in mind that when you use the set_map tag, it will load the map along with its scripts. When a new script is loaded, the previous is erased. This means that if there is a script linked to the map, then it will overwrite the beginning script and start executing the other script(s). Since this is the case, you could run into errors, like your beginning script not fully executing. This is a limitation, but you can

use the limitations to your advantage.

For example, you can initialize all of your characters and then use the set_story_marker tag to set up a marker with a specific number in it. Then, when the map is loaded and the new storyline is loaded, have the new storyline check the story maker and see if it is that number. If it is, then execute your beginning sequence.

Creating shops:

Shops are unique in their own way. There is only one storyline file for them; although you could make more, I guess. They start with the number 500, followed by a buffer. After that, you must initialize the shops. You initialize the shops by using the number 501. This is the syntax for the shop initialization:
501, <xpos>,<ypos>,<map ID #>, <goto number XXX>
This means that if you are at coordinates (x,y) at mapX, then jump down the code and search through all the rows at column 0 for the number XXX, where XXX is some 3 digit number between 503 and 599. this will begin another set of initialization methods. Each of these methods are different, depending on which type of shop it is...

For armors, spells, skills, items, and weapons, the following syntax is used:
<number XXX>, <1st item >, <last item>, <shop type>

<1st item > and <last item> are item ID #'s of items you want to use. It specifies a range from the first item you want in the shop to the last item you want in the shop. You can put up to about 50 items in one shop.

<shop type> can be 1 of 8 different shops: armor_shop, weapon_shop, item_shop, spell_shop, inn, skill_shop, trainer, and 8 (This is the life shop. Use the number 8 or create a define for it.)

For the inn and life, the following syntax is used:

<number XXX>, <cost >, 0 , <shop type>

<cost> is how much it will cost to stay at the inn or how much it will cost to raise the dead. The number zero between <cost> and <shop type> must always be there.

For the trainer shop, the following syntax is used:

<number XXX>, 0,0,trainer,0

Look at the below code for examples:

```
{500,0,0,0,0},
{0,0,0,0,0},// buffer
//501 - if coord = x,y, and map, then goto pos xxx
{501,64,20,200,503},// total of 5 towns - armor
{501,14,20,200,504},// weapon
{501,64,5,200,505},// item
{501,14,5,200,506},// magic
{501,64,20,201,507},// inn
{501,14,20,201,508},// life
{501,64,5,201,509}, //trainer
{501,14,5,201,510},// skills


{503,1000,1005,armor_shop,0},// total of 5 towns - armor
{504,2000,2002,weapon_shop,0},// weapon
{505,3000,3003,item_shop,0},// item
{506,4000,4004,spell_shop,0},// magic
{507,30,0,inn,0},// inn
{508,30,0,8,0},// life
{509,0,0,trainer,0}, //trainer - only one trainer needed.
{510,5000,5002,skill_shop,0},// skills
```

Look at the different highlights. This is how they link together. You can fit up to about 8 shops in the storyline file this way. That was enough shops for me. Maybe you will need more. Unknown. In order to add items to the shops, you will need to specify a range of item ID #'s. You should create your item list in the order that it will be received in your towns. You cannot specify random numbers to be in your list – they must always be in a range, so make sure your items are in order, otherwise you will spend a lot of time redoing your inventory ID numbers to make it work the way you want it.

The trainer shop is a constant and never changes, so if you want, you only have to define trainer one time. Then just set the <number XXX> to always goto it.

This is one way to create shops. This is the way I used to create my shops. It was simple. When creating these guides, I realized that you could do other things if you wanted to. For example, you could add logic to this script and cause different sets of shops to load depending on the values of the inventory markers.

Using the way I showed you in the IDE, you are limited to only 5 shops per map. However, there is a way to expand that. You could use a psudocode like the following to add more shops:

```
chk_xy
        if true
                load_story (<the shops storyline>)
        end if true
        if false
        end if false
```

OR

```
chk_xy
        if true
                <define shops here>
        end if true
        if false
        end if false
```

You could stick this in the messages storyline file if you wish.

Treasure boxes:

Treasure boxes are created via the storyline. You will put the treasure boxes into the treasure box script, so when the 2nd button is pressed, the storyline will be pulled and it will run a check for whatever you put into it. Here is a basic psudocode on how to write the if statement for the treasure boxes:

```
check_xy
        if true
                check_treasure
                        if true
                                msg (treasure is empty)
                        end if true
                        if false
                                add item
                                msg (you get item <name>)
                        end if false
        end if true
        if false
                ** do nothing **
        end if false
```

You can repeat this as many times as you wish with only changing the (x,y) coordinates, which item to add and the item name. It is a fairly simple procedure. It requires 13 lines of code, so you are limited to about 8 or 9 treasure boxes per map. This should be ok, considering that you are only limited to 100 treasure boxes for the entire game.

Warper script:

A warper script is a special script I created that would extend the warping capabilities for a map. In the original map, you were limited to 18 different warp points. Although this may be fine for a general map, it is not ok for a major maze of warping to different locations. I created a maze like this and found myself without enough warp points, so I created the warper script to deal with this problem. This expanded my warp points from 18 to about 115.

The warper script is a script with only "warp" tags in it. It is the final storyline loaded when a map is loaded. AFW will check to see if you have one of these scripts linked to your map. If it does, then it will run through the loaded storyline array and see if you need to be transported anywhere.

Warning – If you have a warper script then don't use any of the scripts that are called when you press the 2nd key (defined in the 2nd key section). If you do, then whenever you press the 2nd key, you will call these scripts and erase your current warping script; thus, causing issues. A way to solve this is to use the load_story tag to reload the warper script at the end of one of these scripts.

The warper script uses code 2000 at the beginning of the file. See story6.h in the storyline editor for an example.

Reserved script numbers:

When creating your storyline files, there are certain file ID #'s, which are assigned to specific things and you cannot use for anything else. The file ID #1 is reserved for shops. File ID #2 is reserved for the "new game" script. File ID # 99 is reserved for the cheat script, so don't use these. (This does give you a clue on how to use the "cheat" option in the beginning.)

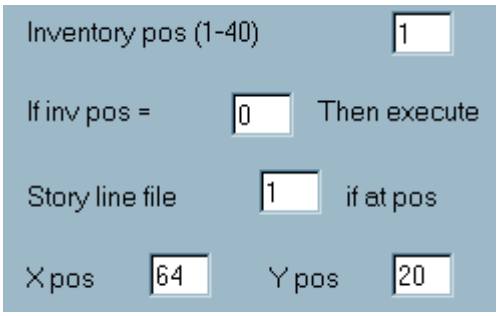
The cheat option (at the beginning screen):

The cheats are pretty simple: you create a storyline file with the ID #99. In this file, you add extra stats, gold, players, boat, airship, canoe, items, magics, skills, or whatever your little heart desires. I am considering creating an external file that will automatically create one of these storylines based upon what kind of "cheat code" is entered into it.

The main idea was that as you progressed through the game, major bosses and such would give you cheat codes. These cheat codes would give you different abilities, stats, or items. You can create something like this if you wish. Since you know the general idea behind cheat code creation, you can create your own external file and have it create its own storyline cheat file.

The 2nd key:

When the 2nd key is pressed, it runs a check for a few different storylines. First it will execute the treasure storyline (if it exists). Then it will execute the messages storyline (if it exists). Then it will execute the other storylines that are defined like this:



The treasure storyline is for treasure chests. The messages storyline is an extra set of messages that

towns people could say (assuming that they are stationary). This could also be used to aid in the game's plot. The other storylines that are defined like the picture above, are specifically for shop creation and for specific plot occurrences (or to cause plot occurrences).

Storyline creation (using the editor):

First, create your new header file for a story line. (Look at the storyX.h files in the storyline editor for how they should look. You can zero one of them out if you wish or copy and paste it into your own file.) Make sure the header file is a routine that updates the storyline array. Now code your storyline.

Go into the map_saver.c file and modify the "if (i==X){update_slX();}" statements or add to them. (X represents a number) You can turn this into a case statement if you wish. The "update_slX();" will be the name of the routine you created. The X in "if (i==X)" will be the file ID # of that file. After that is completed, then compile and run the program. At the prompt, type in the number you want for the ID and then it will export it. Now update your map to use that storyline and run afw to see if it works.

Good luck with the debugging.