Manual and tutorial for the Script Generator.
By Lilac Soul.

Date: May 11th 2006.

Version of the Script Generator this pertains to:
Version 2.2 and 2.3.

NEW IN THIS DOCUMENT SINCE LAST RELEASE: No news since 2.2. Since 2.1: Major edits of instructions for OnActivateItem, OnAcquireItem, OnUnAcquireItem, OnPlayerEquipItem, and OnPlayerUnEquipItem scripts.

Index

Chapter 1. Introduction to the Toolset World

In order to understand a lot of the things scripting (and the Script Generator) let you do, it is beneficial to have a certain understanding of the Aurora Toolset. I suggest that you play around with it for a while, so that you are a little familiar with it before you throw yourself into the scripting fray. In this chapter, I am going to try and give you a rudimentary understanding of what objects are, especially considering the *tag* and *ResRef* aspects of objects. The description here will not be terribly profound, and will be relatively non-technical. I plan to tell you only what you need to know to use the NWN objects scripting wise.

Objects

In case you are familiar with other programming languages, you may already be thinking: "Aha, Object Oriented Programming." This is incorrect, so if you don't know what Object Oriented Programming is, you're not missing any points relating to the Aurora Toolset. An object is simply all the things that exist "physically" in the game (inasmuch as anything can exist physically when speaking about software). Thus, creatures, placeables, items, waypoints, sound objects, and store objects are objects. So are triggers and encounters, and also areas and the module itself are considered objects. PCs (Player Characters) are a type of creature, and are considered objects as well in the world of Neverwinter Nights.

A lot of the scripting you will be doing, in fact most of it, will have to do with objects in some way or another. When making an NPC say something, damaging a player that enters a trigger, or applying a visual effect to something, or even creating an object at runtime[1], you are making a script that interacts with an object. Thus, it is important to be able to identify the object you want to interact with. This can be done in a variety of ways, which will be detailed in the scripting chapter. For instance, you can identify the object that opens a chest fairly easily with the command GetLastOpener in the OnOpen event of the chest. Sometimes, however, you need to identify an object in other ways. You may need to simply locate the NPC called Bob, or you may need to find him on the palette so that you can create him. In the first case, you're locating an object that already exists in the game (he is positioned in an area somewhere). In the second case, he doesn't exist in the game (i.e. he isn't in any of the areas in the game), but only on the palette.

Since many objects in the game can have the same tag, there are various script commands to further narrow down the search. GetObjectByTag has an option to, instead of finding the first object with that tag, you can find the second, third, etc., object with that tag. Also, there are commands such as GetItemPossessedBy, which finds an object with a certain tag possessed by, for instance, a PC, and GetNearestObjectByTag, which gets the nearest object with a given tag.

Objects already in the game

All objects, whether creatures, waypoints, placeables, areas, etc., have certain characteristics. You can view these in the toolset. For instance, all objects have a name and a tag. The name is what the player sees when he interacts with the objects (e.g., the name of the NPC he is talking to or the name of the area being loaded), as well as what is displayed as the main-characteristic in the

[1] If you put, say, a creature in an area directly in the Toolset, this is designated design time. Putting a creature in an area via scripting, i.e. while the game is running, is called creating it at runtime.

Toolset. The tag of an object isn't something the player will notice, but it is the way, most of the time, you'll be, as a scripter, thinking of the objects. This will be detailed a little better later on, but if you run a script where you need to do something with, say, a guard, you'll often need to know the tag of that guard. If the tag of the guard is "big_guard"[2], you can use GetObjectByTag("big_guard") to identify him and have him do whatever you want (well, mostly, not sure if you can make him do the Macarena…). Thus, you'll often need to know the tag of your objects. You can always find this by placing a copy of the object in an area in the toolset, then right-click it to view its properties. It will usually be placed just under the object's name.

You can give a lot of different objects the same tag, and you can give a lot of seemingly identical objects different tags. It is usually best to use different tags for different objects. It will probably make sense for you to tag all your guards "guard". But perhaps there's a tough guard that you want to be able to single out via scripting, so you give him the unique tag of "big_guard". He could still have the name "Guard", just like all the other guards, and look exactly the same as the normal guards. That way, the player won't be able to tell which is the tough guard, but you'll be able to single him out easily via scripting.

Objects on the palette

It will often make sense to not put an object in an area at design time, i.e. directly in the Toolset. For instance, the first time a PC enters an area, you don't want any guards to be there, but the second time, if the PC enters after a warning, you might want to spawn in a guard. In order to be able to do this, you need to have the guard on your palette. Technically, he isn't really an object while on the palette, but that's just semantics.

Objects on the palette have names and tags, but you can't use those to identify such an object. For instance, it could easily be that you had three different types of guards, all with the tag "guard". There is a third characteristic that all objects have, and that is a ResRef, or resource reference. Technically, the ResRef is the name of the file that holds info about that particular object on the palette. Note that, on the palette, ResRefs are unique. This is necessary for at least two reasons: First, two files can't have the same name. Second, you'll need to be able to identify each object on the palette easily when scripting – you couldn't do this if more of them had the same ResRef.

When you create an object using one of the wizards (say, the Creature Wizard), you only get to choose its name. The tag is then constructed as the name without any spaces, and the ResRef is constructed as an all lowercase version of the tag, or the first 16 characters of that (the ResRef can only be 16 characters long). Thus, the object you create named "Big Bob Guard" will be given the tag "BigBobGuard" and the ResRef "bigbobguard". If an object already existed with the ResRef "bigbobguard", it will get a constructed ResRef like "bigbobguard001".

You can easily edit the object's tag, but not it's ResRef. The only way to decide the ResRef yourself is to right click the creature on the palette, press edit copy, and then edit it's ResRef. Note that this actually makes a new object on the palette – the first one will still exist with the old ResRef.

Though it is, to me, very clear when you must use the tag of an object and when you must use the ResRef, this is the cause of most scripters' problem. A quick solution is to make it a rule to

[2] Note that tags, unlike names, can't have spaces in them.

make the tag and ResRef identical whenever possible (if you want two different objects on the palette to have the same tag, you can do so, but you can't give them the same ResRef).

Note that just because two objects on the palette can't have the same ResRef, that doesn't mean that two objects in the game can't have the same ResRef. You can place various copies of the same palette-object around your game, and you can even edit those so that they are different. But they are created from the same ResRef blueprint nonetheless.

There are different ways of viewing an object's ResRef. The most reliable is to place a copy of the object in an area, right click it to view its properties, and then select the advanced tab. It will be called Blueprint ResRef. You can also click edit on the palette to view its properties. If you press edit copy, remember that you're actually creating a new object with a new ResRef, which is most likely not what you want. So press "edit", not "edit copy". For standard palette objects, there's no "edit" option, so you'll have to place a copy of it in the game. You can also use the Lexicon (http://www.nwnlexicon.com), which contains a listing of tags and ResRefs for standard palette objects, as well as the palettes for items, creatures, and placeables that are included in the Script Generator.

The major use of ResRefs, thus, is when creating an object at runtime from the palette. The commands CreateObject and CreateItemOnObject both require the ResRef, not the tag, of the object you want to create.

Summary

All objects have, among other characteristics, a name, a tag, and a ResRef. The name is what the player notices about the object. The tag is used for identifying an object, which already exists in the game. The ResRef is most commonly used for identifying an object on the palette, which you want to create via scripting. No two objects on the palette can have the same ResRef. Because tags and ResRefs are often confused by many people, it is a good idea to keep them identical whenever possible.

Chapter 2. Introduction to scripting

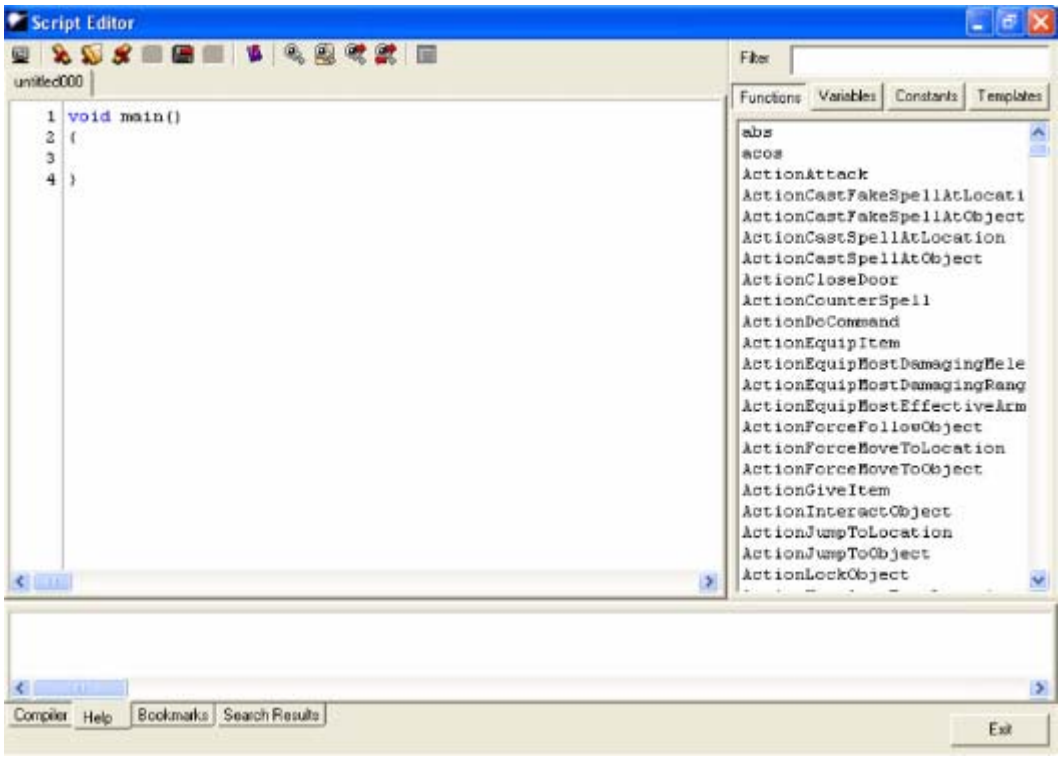
In this chapter, I will try to give you a basic understanding of what scripting is and does. This is solely intended to make you understand how to use the Script Generator, and as such it is very basic and rudimentary. It probably won't teach you how to script – there are other tutorials out there that attempt to do so. Here, however, I will teach you what a script is, what an event is, and give you a few examples of how you can use scripting to achieve your goals. Even if you don't know how to script, you'll still need to know what can be achieved via scripting in order to realize that, perhaps, the Script Generator can help you to do what you want.

What is a script?

One way to define a script is: *a type of computer code than can be directly executed by a program that understands the language in which the script is written.* Basically, when you create a script, you make a small "program" that Neverwinter Nights knows how to translate into something.

Anybody who knows Basic programming will be familiar with the overall flow of nwscript. It isn't really object oriented; you can't create classes, etc. Basically, with some exceptions, it's just a list of commands being executed one by one. The syntax and rules are borrowed from C, so if you know some C programming, the layout of the scripts won't be a surprise to you.

The Aurora Toolset comes with a script editor, but unfortunately the only script wizard included is for making scripts used in conversations, and even at that it isn't very versatile. To open up the script editor, look on the left side of the screen – there's an expandable tree structure, with one major note called scripts. Either right-click it and press new, or expand the node and right click an existing node and press edit to edit that particular script. This will open up a window that looks like this:

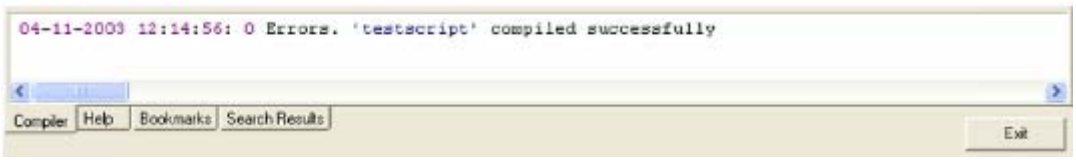


I will not go into details about the syntax of nwscript, or how the script editor works. But you will notice these lines:

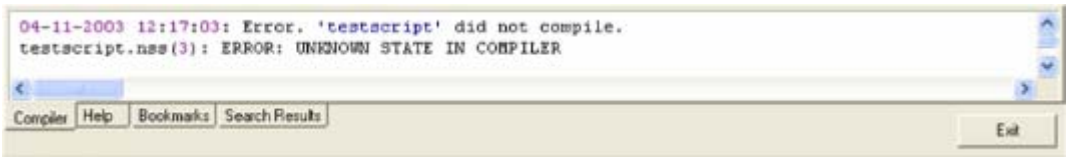
```
void main()
{

}
```

This is a normal script (when you use the Script Generator and it asks you what kind of script you want to make, all scripts but the text appears when (starting conditional) scripts will have the structure you see above. If you want to know what "void" means, I suggest you read the Lexicon, but basically, it is just the way that Neverwinter Nights knows that this is the main function. Whenever the script runs, it will start with the main function and then execute everything within the { }.

Actually, that isn't 100% true. Neverwinter Nights can't read the scripts the way we write them – first, in the Toolset, it needs to be compiled. Pressing the little computer screen icon near the top of the script editor will both save and compile the script. You can also press F7. If you have turned it on in the options on the main toolset screen's menu, the script will always get compiled whenever you save it – even if you just press the disk icon for saving. If there were no errors in the script, you will see something like this at the bottom of the script editor:



If, however, there was some error, you might see something like this (error message will vary depending on the type of error, of course):



The scripts the Script Generator make have been tested as thoroughly as possible for me – they should compile. If they don't, make sure you have copied EVERYTHING – sometimes, important stuff is placed at the top of the script, even above the lines about the script being made with the Script Generator. Also, make sure that you have deleted the initial

```
void main()
{

}
```

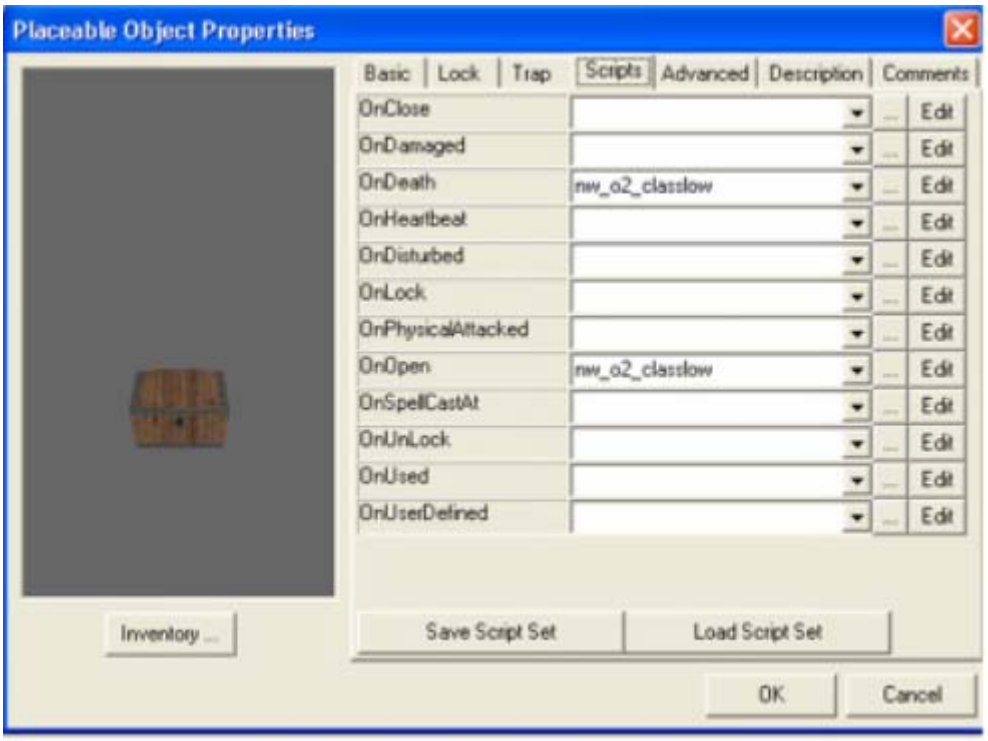
created when you opened the script editor. If the script still doesn't compile, either mail me at lilacsoul@gmail.com and / or post a question either at BioWare's scripting forum (http://nwn.bioware.com/forums/viewforum.html?forum=47) or my own scripting and bugs forums (http://lilacsoul.proboards13.com/index.cgi).

What is an event?

Many objects in the game have events attached to them. Basically, an event is where you can attach a script. The script in the event will then fire upon certain conditions. For instance, a chest has an OnOpen event. When somebody opens the chest, the script in its OnOpen event is executed.

For instance, you want something to happen when a PC opens a chest (perhaps you want to spawn in a creature at a waypoint near the chest that will guard the chest by trying to kill the intruder).

The first thing you'd then do is create a chest and place it in an area. Somewhere near it, you'd then put a waypoint (waypoints won't be visible to the player). Note the tag of the waypoint, and make sure it is unique. You then right click the chest and open up its properties. Click the scripts tab. You'll see something like this:



Note that there are already two scripts attached. The "nw_o2_classlow" script is attached both to the OnOpen event and the OnDeath event. That means that, if somebody (most likely the PC) opens it or bashes it (or kills it by other means, e.g. a spell), the script called "nw_o2_classlow" will run. This is a standard BioWare scripts that puts some low level, random treasure in the chest. Thus, the distribution of treasure can be relatively random. For this script, though, we want something else to happen. So delete the two places where it says "nw_o2_classlow" – you're not deleting the scripts, you're simply removing them from the object. You can then make the script that goes into the OnOpen event of the chest with the Script Generator (more information about that in the following chapter). When you have made the script that will spawn in the guard (but only the first time the chest is opened – again, this will be detailed in the next chapter), press the Edit button next to the OnOpen event. This will open up the script editor. Delete the void main and everything else, and paste in the script made with the Script Generator. Press the compile and save button, and choose a name. You may want to use certain naming conventions for your scripts so you can identify them more easily, but for now, just call it "chest_onopen". If the script compiles without problems, close

the script editor, and your new script will be attached to the OnOpen event of the chest, and will run when somebody opens the chest in game. You may want to make a second script, identical except working for OnDeath of the chest, in case the PC decides to destroy the chest rather than just open it. Or you could check the chest's plot box (under the Basic tab), which means that nobody will be able to destroy it. Ever. Unless the plot flag is switched off again, that is.

It is good to know which events exist and which ones don't. The Lexicon contains descriptions of all the events available for the various types of objects. You might want to download the Lexicon. The index for events can be found online at this link:
http://www.nwnlexicon.com/compiled/categoryevent.index.html.

The most important thing to note, perhaps, is that items don't have any events attached. Events such as when a PC activates or picks up an item are actually events of the module. This means that one event, containing one script, has to handle stuff for, potentially, more than one item. At the time of printing this, the Script Generator supports making of scripts for the module's OnActivateItem, OnAcquireItem, OnUnAcquireItem, OnPlayerEquipItem, and OnPlayerUnEquipItem events – that is the events which fire when a PC, respectively, activates an item (such as using the unique power of the Stone of Recall in the NWN original campaign), acquires an item, looses an item, equips an item, or unequips an item. When you start making such a script with the Script Generator, read the information in the form that pops up – it details an easy way to do this without having to combine scripts.

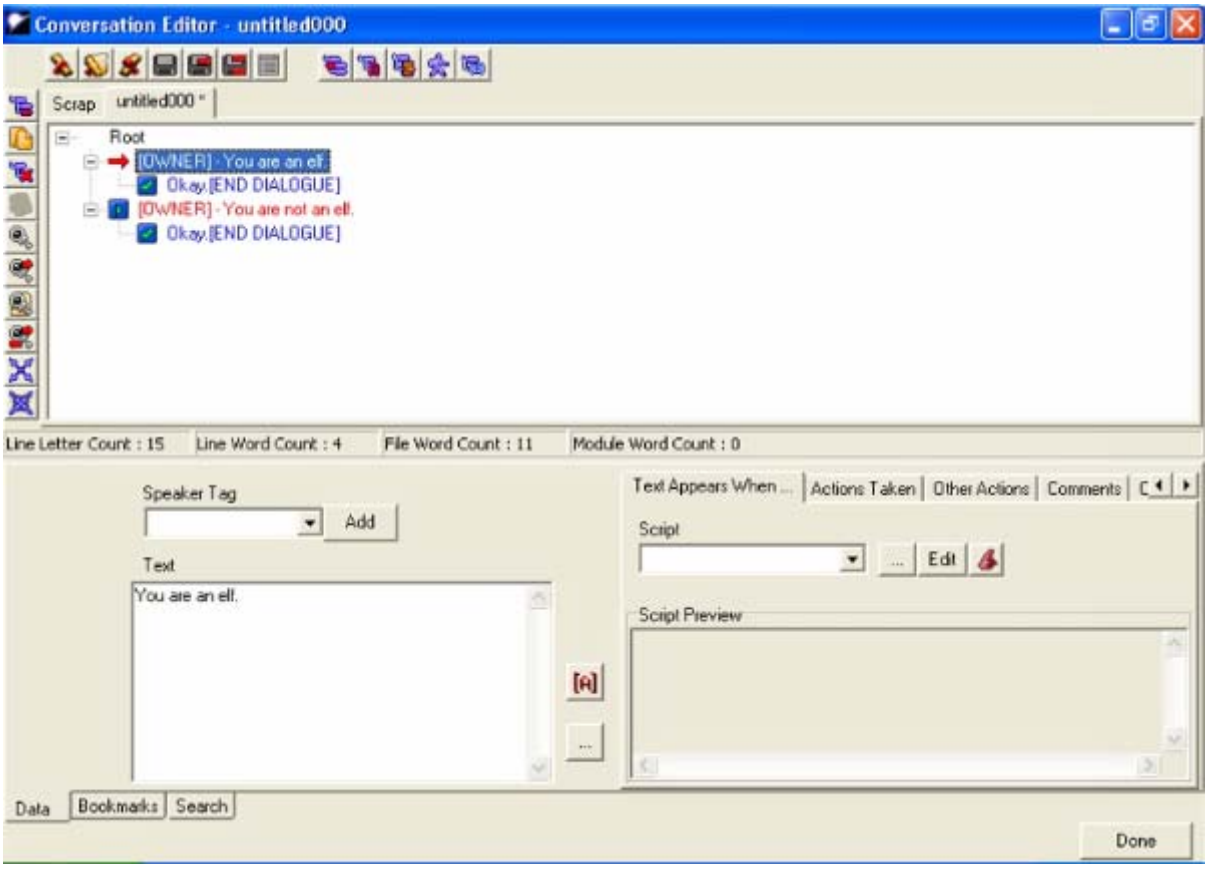
Chapter 3. Using the Script Generator

"Text appears when scripts"

There are basically two types of scripts. One type is used to determine when a certain line of text in a conversation line is spoken or not. This is called a "text appears when" script or a "starting conditional". It is different from other types of scripts because it needs to return either TRUE or FALSE to the conversation (TRUE meaning that the line should be spoken, FALSE that it shouldn't be). The other type can be called a normal script. In the Script Generator, there are certain other script types, like blacksmith scripts or OnActivateItem scripts that can be made. These are basically normal scripts, it's a distinction that pertains purely to the Script Generator.

Remember how we talked about the void main() line earlier? That line is common to all normal scripts, but "text appears when" scripts don't have that line. Instead, they have a like that says "int StartingConditional()".

Open up the conversation editor in the toolset. You will see a screen like this:



There are four places you can place scripts. Look in the bottom left. There's a tab called "Text appears when", you can put a script there. There's a tab called "Actions taken", you can also put a script there. And if you press the arrow to the right, there's a tab called "Current file" where you can put a script for when the conversation finishes normally and when it is aborted. Each line in the

conversation has its own “text appears when” and “Actions taken” tab or event (though they don’t all need to have a script put in them).

If you look at the screen, you’ll see that one of the lines in the conversation is highlighted. That means that any script we put into the “Text appears when” or “Actions taken” event has to do with that line and that line only. So if you put a script into that line’s “Text appears when” event, then change to another line, the “Text appears when” event of that line is still empty.

Say we put a script in the “Text appears when” event of the currently selected line, which we make with the Script Generator. For instance, we only want the PC to hear the NPC speak the “You are an elf” line if the PC is actually an elf. We can then open up the Script Generator and have it make that script for us – in this case, BioWare’s Script Wizard could do the same thing, however. Anyway, open up the Script Generator (if this is your first time using it, you may be asked a couple of questions before you can start making scripts). There’s a scroll box that says “Choose script type”. Select the “Text appears when (starting conditional)” option.

You’ll then be asked what you want to check for. The PC’s race is part of his or her stats, so select that. A box will pop up that asks you what you want to check for. You can select all the ones you want. In our case, we just want to check the “Race” box and press continue. A new box will pop up. Set it to be so that the PC must be elf. Remember, we only want the PC to see this line if he is an elf, so that’s what the PC must be. Click the okay button. Your script should now look like:

```
/*  Script generated by
Lilac Soul's NWN Script Generator, v. 1.5

For download info, please visit:
http://www.lilacsoul.revility.com    */

int StartingConditional()
{
object oPC = GetPCSpeaker();

if (GetRacialType(oPC) != RACIAL_TYPE_ELF) return FALSE;

return TRUE;
}
```

(Version number may vary). If we want, we could repeat our steps and make more conditions (for instance, if we only wanted the PC to get that line if he or she also had a certain item). But for now, we’re satisfied, so just press the “Copy script to clipboard” button. Go back to the toolset. Select the line we only want the PC to hear if he or she is an elf. Go to the “Text appears when” tab. Press edit. The script editor then pop ups. Erase everything in it and copy in the script from the clipboard. An easy way to do this is to place the cursor in the script editor window, press CTRL-A which will highlight it all, then press CTRL-V to paste in the script from the clipboard, overwriting all the highlighted stuff in the process (this may not work on all systems, I don’t know). Save and compile the script (F7 or the button in the top left corner), and close the script editor by pressing exit. The script is now attached to the “Text appears when” event of the first line in the conversation. Thus, if the PC is an elf, he’ll get the “You are an elf” greeting. If he is not an elf, the conversation will progress to the next NPC line, giving him “You are not an elf” greeting.

You can also put "Text appears when" scripts on PC lines. This works in exactly the same way – if the PC doesn't meet the conditions you set forth when you make the script, that line won't be available to the PC.

A good and common use for "Text appears when" lines is to keep track of a quest in a conversation. You may have lines like this (only first lines shows):

- Hello, would you do my quest?
- Have you done my quest?
- Thanks for finishing my quest!

You only want the PC to get the first line when he hasn't accepted the quest. The second line should only be available when he hasn't finished the quest.

To accomplish this, we need to work with local variables. A local variable is a variable stored on something. For instance, we can set the local variable "my_quest" on an NPC to 1. We can then check if that local variable is 1 or something else.

A local variable that has never been set has the value 0. This is useful for us. Can you see how? On the first line, we put a "Text appears when script" that checks if the local variable "my_quest" on the PC is 0. The PC then only gets the line if the variable is 0, i.e. has never been set. Once the PC accepts the quest, we then set the variable to 1 (see the normal script section on how to do this).

On the next line, we check if the variable is 1. And when the PC has done the quest, we set the variable to 2. There's no need to check if the variable is 2 on the last line, but you may want to do so just to make sure, in case you add more lines later on.

Normal scripts

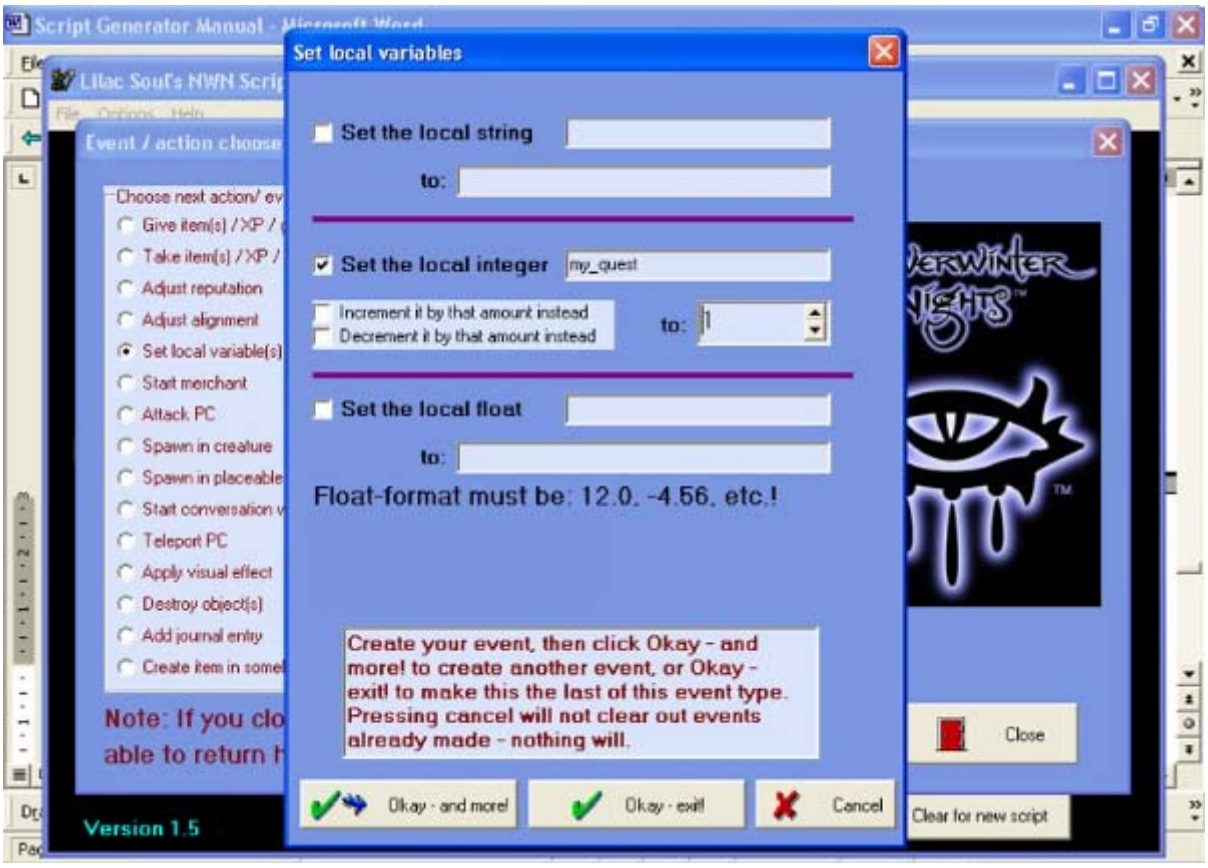
As has been mentioned, any script that isn't a "Text appears when" script (i.e. a starting conditional) is a normal script. In the Script Generator, blacksmith scripts, unique power item scripts (OnActivateItem scripts) and OnAcquire / OnUnacquire scripts aren't called normal scripts, so we won't talk about those just yet.

A normal script can be placed in a lot of places, unlike the "Text appears when" scripts we talked about earlier. You can use them in conversations on the "Actions taken" event, and on a lot of other events in the game, such as when a chest is opened, a creature is killed, or a trigger is entered.

To continue the example from a few lines up, we could make a script that sets the local variable "my_quest" to 1 on the PC on Actions taken of the first line (so the PC wouldn't get the "Hello, will you do my quest" greeting when he has already accepted the quest). It could also go on a line you've added to that line, most likely a PC (blue) line where the PC actually accepts the quest.

To make the script, start up the Script Generator. In the "Choose script type" combo box, choose "normal script". We now have to tell the Generator where the script is called from. In this case, it is called from a conversation. So choose that. If this is your first time using the Script Generator, you may see a box giving you some info about how to use the event window. I suggest you read it. Then, when you press close, you'll see the event chooser window. You'll note that there are a LOT of options you can choose. Hopefully, you'll notice that one of them is called "Set local

variable(s). Select that one and press “Script”. This will open up a new form. We want to set the variable “my_quest” to 1. 1 is an integer (no decimals), so choose the integer option, type in the name and adjust the spin edit to the correct value. When you’re done, it should look like this:



Pressing either of the okay buttons will script the lines for setting the local variable to 1. The “Okay – and more” button will clear out the form so you can set another variable. The “Okay – exit” button will close the form. So just press the “Okay – Exit” button. This will bring you back to the event chooser. There, press “Close”. This will bring you to the main form where you can view the created script. It should look like:

```
/*  Script generated by
Lilac Soul's NWN Script Generator, v. 1.5

For download info, please visit:
http://www.lilacsoul.revility.com    */

//Put this on action taken in the conversation editor
void main()
{

object oPC = GetPCSpeaker();

SetLocalInt(oPC, "my_quest", 1);

}
```

You then press the "Copy script to clipboard", go to the toolset and the conversation editor, highlight the line you want this script to fire off (e.g. the line where the PC accepts the quest), go to the "Actions taken" tab, press edit, clear out the stuff in the script editor that gets opened, and paste in the script made by the Generator. Save, compile and exit.

You should now be able to repeat those steps to make a script on the second NPC greeting line that checks if the variable is 1 (on "Text appears when") and sets it to 2 on "Actions taken" of the line where the PC declares (and proves, probably) that he has completed the quest.

How could you get the PC to prove that he has completed the quest? Well, you could put a script OnDeath of a creature that sets a local variable on the PC (say, the local variable "killed_bob"), and then check for that in a "Text appears when" script in the conversation. The PC only gets to say "I killed Bob" if he has that local variable set. You could also have Bob drop an item when he dies (put it in his inventory, make it droppable, make it non-pickpocket-able), and then check in the same way if the PC has that item in his inventory.

This is about as much help as I can give you on scripts without getting technical about how scripts work. If you still don't understand how scripts work or how to use the Script Generator, try playing around with the Toolset for a week, and ask questions on BioWare's scripting forum and BioWare's toolset forum. You may also try to look for some tutorials. Neverwinter Vault is a good place to start – it contains tons of Neverwinter Nights related material. You most likely downloaded the Script Generator from there!

Blacksmith scripts

Blacksmith scripts are fairly simple, though I do admit that the code produced by the Script Generator is messy looking at best.

Basically, the blacksmith scripts allow you to have a PC "craft" certain items by using other items. For instance, when the PC puts three different items in a container, then fires a spell at that container, a magic sword is created. If you are at all familiar with the other functionalities of the Script Generator, then you should have no problem using this function.

Chapter 4. Item related event scripts

IMPORTANT, UPDATED WITH VERSION 2.2 OF THE SCRIPT GENERATOR:

BioWare has implemented a new default way of handling the (now 5) module item-related events: OnActivateItem, OnAcquireItem, OnUnaqcuireItem, OnPlayerEquipItem and OnPlayerUnEquipItem.

Up to and including version 2.1 of the Script Generator, I recommended using one particular approach for making the scripts generated with the program work in game. This approach consisted of altering the system BioWare had set up with Hordes of the Underdark, and was thus not

compatible with BioWare's system. This is a problem, because BioWare's system has of course become the community standard.

I have been thinking a lot about what to do about this, and after asking around for advice on various forums (and thanks to everyone who helped me decide!), I decided to switch to an approach that **is** compatible with BioWare's system. However, the **new system** is not compatible with **the old system**, so you shouldn't switch in the middle of a module. Below, I will first give some information on when to use the old system and when to use the new one, and then I'll give instructions on how to set up each of these systems. If you have any questions, feel free to contact me with the contact information printed elsewhere in this manual.

Should you use the old system or the new system

Should you use the OLD SYSTEM or the NEW SYSTEM for these five events: OnActivateItem, OnAcquireItem, OnUnacquireItem, OnPlayerEquipItem, and OnPlayerUnequipItem?

It depends. The NEW SYSTEM is better because it is compatible with BioWare tagbased, which is the community standard for these events. The OLD SYSTEM is not compatible with BioWare's system, and therefore doesn't allow you to use most people systems in your module.

However, switching from the OLD SYSTEM to the NEW SYSTEM can be problematic. I don't recommend that you do that in the middle of a module, since the NEW SYSTEM is not backwards compatible with the OLD SYSTEM.

So if you have, in the module you are currently working on, already made OnActivateItem, OnAcquireItem, OnUnacquireItem, OnPlayerEquipItem, or OnPlayerUnequipItem script with the Script Generator version 2.1 or older, I DO NOT RECOMMEND switching to the new system. In that case, use the OLD SYSTEM instead and switch to this new system when you start a new module. The NEW SYSTEM takes a little more work to get working, but it is well worth it for being compatible with community standards.

Questions? Please email me at lilacsoul@gmail.com.

How to set up the new system

Here, I will describe THE NEW SYSTEM for making item event related scripts. It was introduced in the Script Generator in version 2.2, and while it does take a little more work to get working, the beauty of it is that it is 100% compatible with BioWare's tagbased system. However, it is NOT compatible with the system recommended up to and including version 2.1 of the Script Generator. So if you have, in the module you are currently working on, already made OnActivateItem, OnAcquireItem, OnUnacquireItem, OnPlayerEquipItem, or OnPlayerUnequipItem script with the Script Generator version 2.1 or older, I DO NOT RECOMMEND switching to this new system. In that case, use the OLD SYSTEM instead and switch to this new system when you start a new module.

This is not an introduction to tagbased scripting. It will not tell you exactly how BioWare's tagbased system works, nor will it teach you how to make this work on your own, without the Script Generator. However, it WILL tell you how to setup the item-event related scripts you make with the Script Generator to work flawlessly with BioWare's tagbased system.

First thing you must do is make sure you have these scripts in these events. They are standard BioWare scripts, and are automatically placed in these events when starting a new module in the Toolset:

OnAcquireItem: x2_mod_def_aqu
OnActivateItem: x2_mod_def_act
OnModuleLoad: x2_mod_def_load
OnPlayerEquipItem: x2_mod_def_equ
OnPlayerUnequipItem: x2_mod_def_unequ
OnUnacquireItem: x2_mod_def_unaqu

Then open x2_mod_def_load and make sure you have this line, without // in front making it green. You should have it as it is there by default, on line 89.
SetModuleSwitch(MODULE_SWITCH_ENABLE_TAGBASED_SCRIPTS, TRUE);

This means that BioWare's tagbased system is set up in your module. This is good. The next step, and this is very important **BECAUSE YOU MUST CREATE A SCRIPT LIKE THIS EVERY TIME YOU MAKE ITEMRELATED EVENTS FOR A NEW ITEM**:

Create this script and name it the same as the TAG of the item you're making an itemrelated event script for. So, if you're making an OnAcquireItem script for an item tagged "HATCHET", you must make a script called "hatchet". It should look like this:

```
---
#include "x2_inc_switches"
void main()
{
int nEvent =GetUserDefinedItemEventNumber();
switch (nEvent)
  {
  case X2_ITEM_EVENT_ACTIVATE: ExecuteScript("ac_"+GetTag(GetItemActivated()),
OBJECT_SELF); break;
  case X2_ITEM_EVENT_EQUIP: ExecuteScript("eq_"+GetTag(GetPCItemLastEquipped()),
OBJECT_SELF); break;
  case X2_ITEM_EVENT_UNEQUIP:
ExecuteScript("ue_"+GetTag(GetPCItemLastUnequipped()), OBJECT_SELF); break;
  case X2_ITEM_EVENT_ACQUIRE: ExecuteScript("aq_"+GetTag(GetModuleItemAcquired()),
OBJECT_SELF); break;
  case X2_ITEM_EVENT_UNACQUIRE: ExecuteScript("ua_"+GetTag(GetModuleItemLost()),
OBJECT_SELF); break;
  case X2_ITEM_EVENT_SPELLCAST_AT:
ExecuteScript("sp_"+GetTag(GetModuleItemLost()), OBJECT_SELF); break;
  case X2_ITEM_EVENT_ONHITCAST: ExecuteScript("on_"+GetTag(GetSpellCastItem()),
OBJECT_SELF); break;
  }
}
---
```

Note that if you have already made an OnAcquireItem script using this approach, if you later on decide to make a, for instance, OnUnacquireItem script for the same item, the above script will already exist with the correct name, and there's no reason to make it again.

The final thing to do is then to create the script with the Script Generator, and save it using the following naming scheme:

OnActivateItem: ac_+TAG
OnPlayerEquipItem: eq_+TAG
OnPlayerUnequipItem: ue_+TAG
OnPlayerAcquireItem: aq_+TAG
OnPlayerUnacquireItem: ua_+TAG

So, for the item tagged HATCHET, we could get ac_hatchet, eq_hatchet, ue_hatchet, aq_hatchet, and ua_hatchet. If the tag is so long that the script won't let you save it with the entire name, just save it with as many letters as you are allowed.

That's it! You can always find this information in the program as well.

How to set up the old system

Please note that the approach given here is what is considered THE OLD SYSTEM. It was recommended until version 2.1 of the Script Generator, and is provided here for backwards compatibility. It works just fine, but IS NOT COMPATIBLE with BioWare's tagbased system. I recommend you use THE OLD SYSTEM, i.e. the one described here, if you have used previous versions of the Script Generator to make scripts for activating, equipping, unequipping, acquiring, or un-acquiring items. However, if you have not in your current module used the Script Generator to make scripts for any of these events, I recommend that you use THE NEW SYSTEM instead. I DO NOT recommend switching from OLD to NEW system in the middle of building a module, as the new system is not backwards compatible with the old system.

OnActivateItem

The OLD SYSTEM works by having you put this small script OnActivateItem:

```
void main()
{
ExecuteScript(GetTag(GetItemActivated()),
OBJECT_SELF);
}
```

It will then run the script with the same name as the tag of the unique power item. If your item is tagged "fire_stone", the script called "fire_stone" will be run. These new scripts don't have to go anywhere, they just need to exist. If the PC activates an item for which no script exists, nothing will happen.

So if you make an OnActivateItem script with this program, just name it the same as the tag of the item. And then just put the small script posted a few lines up on OnActivateItem of the module properties, and you'll be ready to go.

OnAcquireItem and OnUnAcquireItem

The OLD SYSTEM works by having you remove the x2_mod_WHATEVER reference from all of these five events, listed under the module's properties: OnActivateItem, OnAcquireItem, OnUnacquireItem, OnPlayerEquipItem, and OnPlayerUnequipItem.

Then, when you make an OnAcquireItem script, call it "ac_"+tag. So, if the item is tagged myring", the script is called "ac_myring". Similarly, for OnUnacquireItem scripts, name the scripts uac_"+tag. So the OnUnacquireItem script for the item tagged "myring" would be "uac_myring".

In order for that suggestion to work, you then need this script OnAcquireItem:

```
void main()
{
ExecuteScript("ac_"+GetTag
(GetModuleItemAcquired()), OBJECT_SELF);
}
```

And this script OnUnacquireItem:

```
void main()
{
ExecuteScript("uac_"+GetTag
(GetModuleItemLost()), OBJECT_SELF);
}
```

OnPlayerEquipItem and OnPlayerUnEquipItem

The OLD SYSTEM works by having you remove the x2_mod_WHATEVER reference from all of these five events, listed under the module's properties: OnActivateItem, OnAcquireItem, OnUnacquireItem, OnPlayerEquipItem, and OnPlayerUnequipItem.

Then, when you make an OnPlayerEquipItem script, call it "eq_"+tag. So, if the item is tagged "myring", the script is called "eq_myring". Similarly, for OnPlayerUnEquipItem scripts, name the scripts "ueq_"+tag. So the OnUnEquipItem script for the item tagged "myring" would be “ueq_myring".

In order for that suggestion to work, you then need this script OnPlayerEquipItem:

```
void main()
{
ExecuteScript("eq_"+GetTag
(GetPCItemLastEquipped()), OBJECT_SELF);
}
```

And this script OnPlayerUnEquipItem:

```
void main()
{
ExecuteScript("ueq_"+GetTag
(GetPCItemLastUnequipped()), OBJECT_SELF);
}
```

What if I already have a script in one of these events?

This could potentially be a problem, yes. If your module already uses BioWare's tagbased system, then I recommend you go ahead using that, and use the Script Generator's **new system**. Read the caveat above, though. However, if you are using one of the very old systems, that go something like this:

```
if (TAG==”something”)
{
STUFF
}
else if (TAG==”somethingelse”)
{
OTHER STUFF
}
```

…and so on; if you have that sort of system set up, you can most likely get by with using this approach: Create a third script that you put in the event instead. It should look like this:

```
void main()
{
ExecuteScript(“name of script 1 you wanted on this event”, OBJECT_SELF);
ExecuteScript(“name of script 2 you wanted on this event”, OBJECT_SELF);
}
```

Where either script 1 or script 2 is the script recommended by the Script Generator (old or new system). Just make sure that none of the tags in the other script could cause a script to be executed from the tagbased system that the Script Generator uses (old approach or new approach). For instance, if script one uses the “if TAG” approach and checks for an item tagged “key”, and you also have a script tagged “ac_key”, then it could fire both the “if TAG” sequence and the “ac_key” script OnActivateItem. To avoid this problem, I recommend sticking with just one system inside a module – preferably the BioWare tagbased system, being compatible with the **new system** the Script Generator uses. Again, only use the **new system** if you've read all the instructions here and are sure it won't cause conflicts because you've already started using the **old system** in your module.

Chapter 5. Useful links

BioWare's homepage: http://nwn.bioware.com/

BioWare's Scripting Forum: http://nwn.bioware.com/forums/viewforum.html?forum=47

BioWare's Toolset forum: http://nwn.bioware.com/forums/viewforum.html?forum=46

Neverwinter Vault: http://nwvault.ign.com/

My Scripting Forum: http://lilacsoul.proboards13.com/index.cgi

Script Generator: http://nwvault.ign.com/View.php?view=Other.Detail&id=4683&id=625

The Lexicon: http://www.nwnlexicon.com/

The Lexicon (download version:) http://nwvault.ign.com/View.php?view=Other.Detail&id=736

My mail address: lilacsoul@gmail.com

If you have questions, suggestions, etc., about the Script Generator, please feel free to either mail them to me, post them on my forum, post them on the Script Generator sticky on the BioWare scripting forum, or post them as comments on the Script Generator download site. I read them all regularly. Please, however, do not mail me general scripting questions. I simply don't have time to answer all those questions. BioWare's scripting forum is a very good place to ask those questions.

Carsten Hjorthøj (Lilac Soul)
November 2005.