Exscriptor

X3 External Script Editor

USER MANUAL

Contents

Thanks to Egosoft for creating an incredible series of games

and also allowing them to be so easily modified!

1 Introduction

Welcome to **Exscriptor**, an External Script Editor for Egosoft's *X*-series of games. Currently Exscriptor supports:

- *X3: Albion Prelude*
- *X3: Terran Conflict*
- *X3: Reunion*
- *X2: The Threat*

With this program, you will be able to:

- Open existing X2/X3/TC scripts in both .XML and .PCK format
- Create new X2/X3/TC scripts (.XML)
- Compile the script to check for any syntax errors
- Save scripts as .XML or in simple .TXT format

In addition, because the editor uses the X2/X3/TC/AP data files, all the possible commands, wares, ships, stations and so forth will be available, including those present in mods or patches like XTM. The program even supports the use of data files in different languages (though the editor's interface is only in English). Since the X games offer a large number of commands and objects, these are visible in a list onscreen, so you do not have to remember them all.

Before using the program, however, please read the warranty and check the requirements.

1.1 Requirements

The editor requires Windows and the Microsoft .NET Framework 3.5 or better in order to run. You may have this already installed, but if you do not, you should be able to obtain the latest version from Microsoft's website (www.microsoft.com) or via Windows Update. The editor also requires X2 or X3 to be installed, since it needs access to the game's data files. When you run the program for the first time, it will ask you where to find the games. You can later change this by going to the Tools --> Options menu. It should work with X2 version 1.5, X3 version 2.5, X3TC version 2.7, and X3AP version 1.0 – earlier versions may or may not work correctly (and similarly, any future patches may not work either).

The program should easily run on any computer capable of running X3, but it can take a few moments to load up the data when the editor starts, so please be patient.

1.2 Warranty

Put simply, there isn't one. This program is provided 'as is', with no guarantee that it will work 100% of the time. Before using the program it is **strongly recommended** that you make a backup of your script directory, at the very least. The author accepts no responsibility if you happen to corrupt the game or break your computer somehow while using the program. If this possibility concerns you, then do not use the program.

1.3 Acknowledgements

This program would not be possible without the prior work of other people. Acknowledgements go to:

- doubleshadow for the X3 file system interface
  - (www.doubleshadow.wz.cz)
- the posters in this Egosoft forum thread, who started the process of deciphering the script file format:
  - http://forum.egosoft.com/viewtopic.php?t=89990
- the following Code Project articles, without which the interface would look much more like Notepad:
  - http://www.codeproject.com/KB/cs/shadyrichtext.aspx
  - http://www.codeproject.com/KB/edit/SyntaxHighlighting.aspx
  - http://www.codeproject.com/KB/cs/dandtutorial.aspx
  - http://www.codeproject.com/KB/cpp/RTFSynchronizedScrolling.aspx

Many thanks also to the following people at the Egosoft forums for helping solve various bugs or suggesting improvements in the Exscriptor:

- Blacky_BPG
- ThatGuyBob
- Erilaz
- Cebraio
- bunkerprivate
- SSwamp_Trooper
- Moonrat
- DesertEagle
- draffutt
- many, many others – see the About menu in the tool.

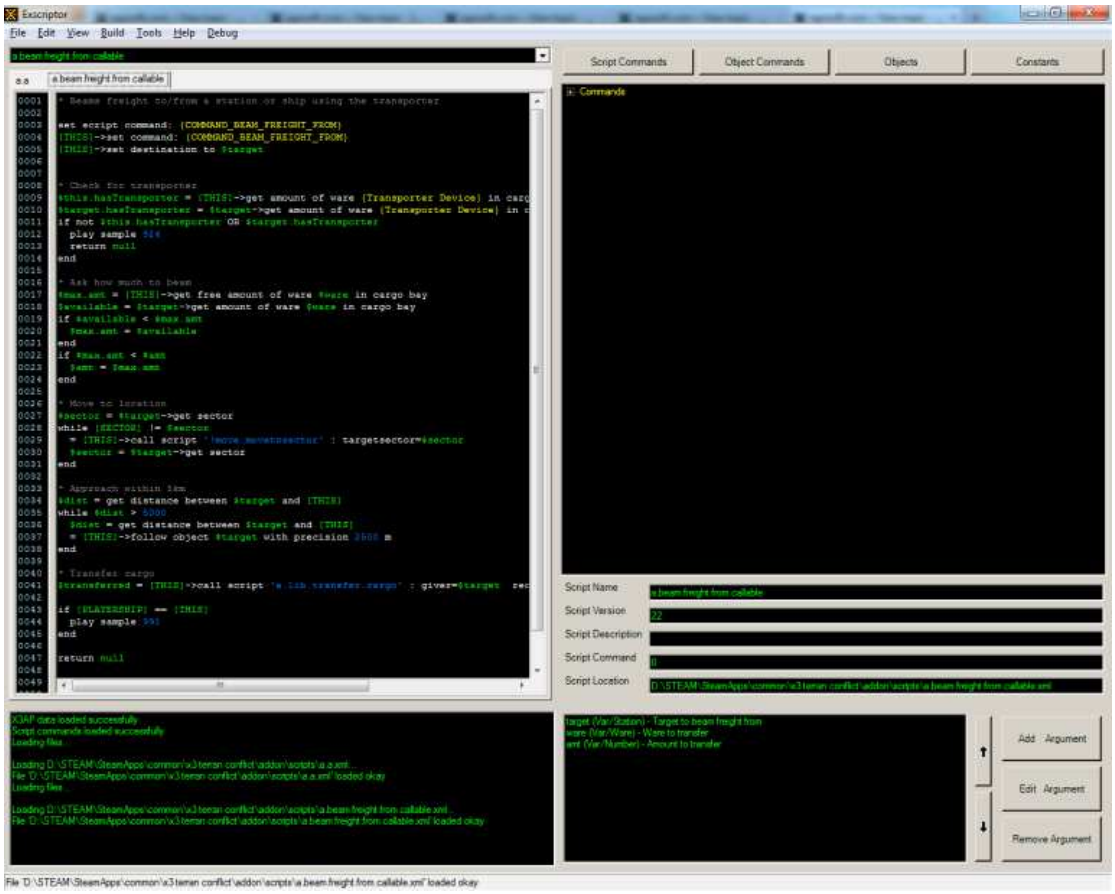
Special thanks to Shimrod for helping with the update to X3AP.

Extra special thanks go to Blacky_BPG for providing alternative hosting at the www.blackpanthergroup.de site!

2 Using the editor

2.1 Main User Interface

When the editor has started, you will see a screen like this:



This is the main interface of the editor. The large black area on the left is the Script Editor, where you can see and edit the code of the script. Below this is the Status Box, which will tell you if anything goes wrong. On the right is the Object List, where you can select script commands, objects (e.g. wares, ship types), and object commands. Above this are the buttons controlling the Object List and below it is the Script Metadata and, below that, the Script Arguments. At the far top left is the Menu.

2.1.1 Script Editor

The Script Editor is where most of the activity takes place. When you open a script, or if you are creating a new one, the code of the script will be displayed here, with the line numbers in the corresponding column on the left. By default, the code is highlighted in the same way it is in X3's internal script editor; you can turn this off in the options (Tools --> Options) if you wish. Most

of the usual text editing commands should apply: copy, paste, cut, undo, redo etc. These commands are also available in the "Edit" menu.

Right clicking in the Script Editor will open a small shortcut menu with six options: script commands, variables, constants, wares, ships, and stations. Clicking on one of these (or, in the case of script commands, one of the sub-options) will open a sort of auto-complete box that will allow you to select the command/object you want. You can either select it directly from the list, or start typing - as you do, the options should be narrowed down. For example, if you open the Ship menu and type "Argon Bus", all of the ships beginning with those letters (i.e. all variants of the Argon Buster) will be displayed. You can either click on them or just keep typing. When you are done, press Enter or click Accept and the selected object/command will appear in the Script Editor.

More details can be found in Section **2.2**.

2.1.2 Object List

The Object List on the right hand side of the interface is there to make it possible to write scripts without memorising all 800+ or so script commands, not to mention the hundreds or even thousands of wares, ships, stations, ship commands and so forth. It has three modes, selectable using the buttons above it. In the first mode, the List will show all the script commands available, sorted into the appropriate categories; the second mode shows all available ship/station commands, plus any user defined names for them; the third mode shows all objects - wares, ship types, that sort of thing. Double clicking on any of these will copy it into the Script Editor at the current cursor position.

More details can be found in Section **2.3**.

2.1.3 Status Box

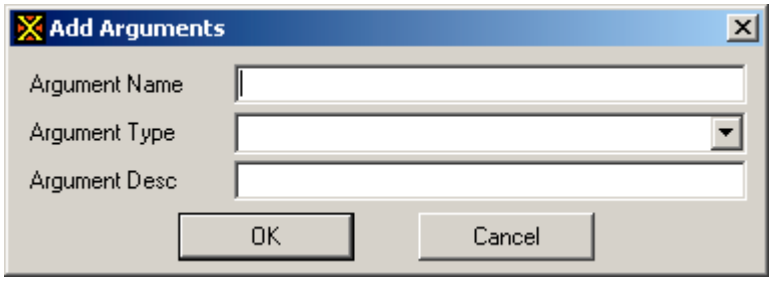
The Status Box is how the editor communicates with you. If it encounters any problems, then it will list them here. You are strongly advised to pay attention to this box, especially when compiling scripts or when the editor first loads the data from X3 (or X2) - there may well be unreadable files, conflicting object commands, or other problems.

2.1.4 Script Metadata

The Script Metadata is data about the script rather than the data in the script: its name, for example. You can alter this information as you wish; the name is the most important, as this determines how it will show up in the game (not the filename - so two separate files with the same script name will conflict). If your script is for a ship/station command, you will probably want to add it to the Script Command box; you can find the command names in the Object List, or you can also use the command number (if you know it).

2.1.5 Script Arguments

Here is where you can add arguments to your script – parameters that it needs to run. If you click the big **Add Argument** button, a new window appears:



where you can enter the name, select the type, and add a description. You can also remove arguments by selecting them in the list and clicking the **Remove Argument** button, or edit existing arguments with the **Edit Argument** button. You can also reorder them with the two arrow buttons. Once added, you can refer to an argument just as a variable, i.e. $argument, just like in the X3 ingame script editor.

2.1.6 Menu

The main Menu in the top left provides a number of extra commands:

- File…
  - New – creates a new script in a new tab
  - Open – opens an existing script (in XML, PCK, or TXT format)
  - Open Script Set - opens a set of multiple scripts at once (like a workspace, or project)
  - Close – closes the current script tab
  - Close All – like above, but for all script tabs
  - Save – saves a script if it has already been saved (otherwise it acts like Save As). This will not prompt for overwriting.
  - Save As – saves a script in either XML or TXT format. Saving in XML will automatically compile the script first. The Exscriptor can also make a backup when overwriting an existing file (a copy with the extension .bak) if the appropriate option is set.
  - Save All – as the name implies, saves all scripts. It will prompt you when necessary for names etc and will compile too.
  - Save Script Set - saves the set of currently opened scripts as a script set (like a workspace or project). Note that this **does not** save the scripts themselves, only a list of their names.
  - Quicksave as Text – as the name implies, it will save the current script to a text file named after the script
  - Exit – closes the program
- Edit…
  - Copy, Cut, Paste, Find, Replace, Undo, Redo, Select All, Goto Line
- View…
  - Hide Object List – you can hide the Object List if you so wish.
  - Compare with – loads the Comparator (see below).

    - Refresh from file – reloads script from disk
- Build…
    - Compile – compiles a script to check it for errors. Note that this does not save the script – no file is produced unless you use Save As.
    - Compile All – compiles all scripts, stopping if there are any errors.
    - Indent – automatically indents a script.
- Tools…
    - X2/X3/TC/AP Mode – this allows you to select whether Exscriptor is running in X2, X3, TC or AP mode. Note that this involves reloading all the data from the appropriate game and can take a few moments. See **X2 support**, **X3TC support,** and **X3AP support** later for more info.
    - Options – allows you to select the game directories, appearance of some of the interface, and also allows you to set other options such as enable/disable highlighting. (See **Options Menu** below for more info.)
    - Language files – allows you to select what language data files you wish to use. Note that not all will necessarily be available! This will reload the game data and may take a few moments; you will need to reload any opened scripts to see the changes, however.
    - Reload Game Data – reloads the game data files.
    - Conflict Viewer – shows all conflicts in data (see **Conflict Viewer** below for more details).
    - Find calling scripts – scans all scripts to see if any call the current script, and if so, gives you the option to open them.
    - Find called scripts – finds and optionally opens any scripts called by the current script[1].
    - Find called scripts (recursively) – finds and optionally opens any scripts called by the current script, plus any scripts called by those scripts, and so on. Will ignore .pck and ! prefixed scripts.
- Help – view the About screen or open up this manual.

When changing language, the user interface remains in English (as a consequence of my not speaking seven languages) but a different set of data files are loaded. However, if you have scripts that use objects or commands that are only available in certain languages, the external script editor will be unable to find them and compilation will likely fail.

Also, a note on saving/opening scripts: there are three formats available when opening scripts, but only two when saving – you cannot save in .PCK format. The .TXT format is readable by any text editor, e.g. Notepad, but includes the script metadata at the top of the file, like so:

[1] Note that if a script exists in both .pck and .xml forms, then the .pck form is opened.

```
test.script.1                  ← name
0                              ← command
Last mission for M0            ← description
194                            ← version
1                              ← number of arguments
name|value|9|name of ship      ← argument
---END-OF-METADATA---
```

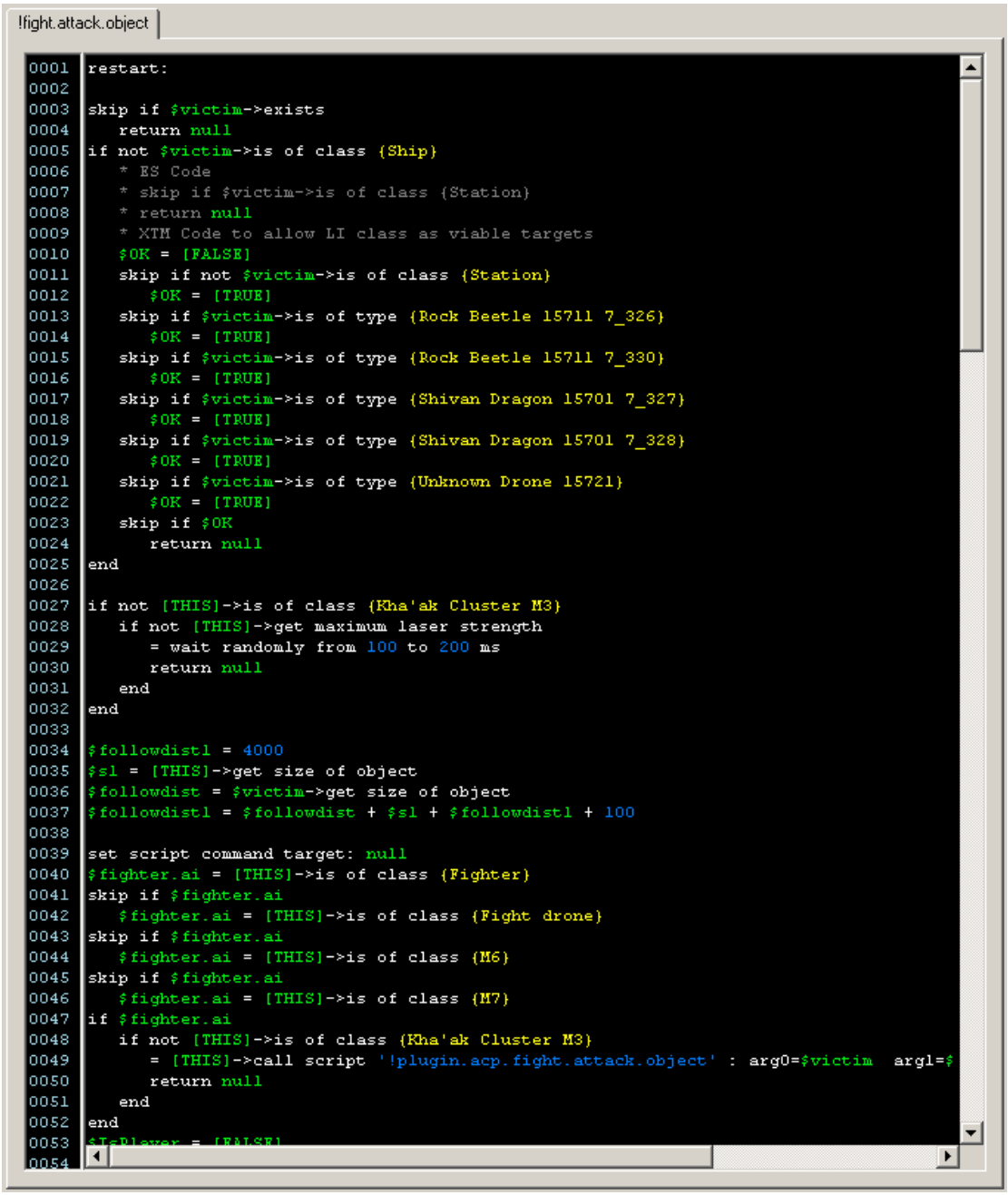
And the rest of the code then follows. The .TXT format is designed to allow you to write/edit scripts in other programs (should you so wish), but if you forget this first part, it will not load correctly.

2.2 Script Editor

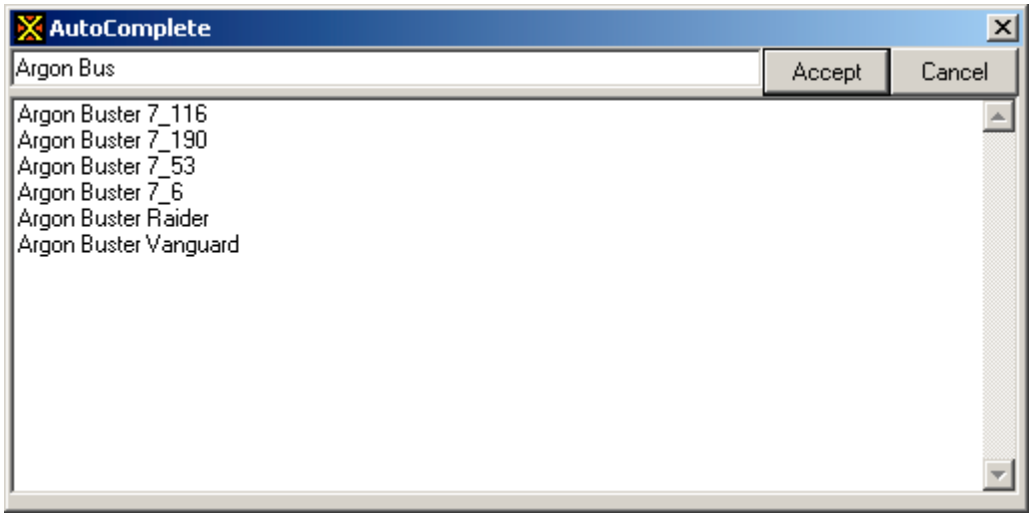
The Script Editor is where most of the activity takes place. When you open a script, or if you are creating a new one, the code of the script will be displayed here, with the line numbers in the corresponding column on the left. By default, the code is highlighted in the same way it is in X3's internal script editor; you can turn this off in the options (Tools --> Options) if you wish. Most of the usual text editing commands should apply: copy, paste, cut, undo, redo etc. These commands are also available in the "Edit" menu.

Multiple scripts can be opened at once, and each will be displayed in its own tab. Above the editor window is a drop-down menu of all the opened scripts, sorted by name, allowing you to quickly select another script if you have many tabs open.

After editing a script, you will notice an asterisk (*) appear next to the name of the script at the top of the tab; this lets you know it has been modified. If you try to close a modified script without saving first, it will ask you if you're sure.



Right clicking in the Script Editor will open a small shortcut menu with the usual editing commands plus several other options: script commands, variables, constants, wares, ships, and stations. Clicking on one of these (or, in the case of script commands, one of the sub-options) will open a sort of auto-complete box that will allow you to select the command/object you want. You can either select it directly from the list, or start typing - as you do, the options should be narrowed down. For example, if you open the Ship menu and type "Argon Bus", all of the ships beginning with those letters (i.e. all variants of the Argon Buster) will be displayed. You can either click on them or just keep typing. When you are done, press Enter or click Accept and the selected object/command will appear in the Script Editor.



Notice how some of the ships have numbers after their name; the reason for this is explained in more detail in the Object List section, but essentially the problem is that there are multiple Argon Busters in the game, all with the same name. To distinguish them, the editor adds some identifying numbers after these duplicates; the numbers are often either the maintype and subtype of the ware or the index into the language file, as these are always unique. If you want to add an Argon Buster or other duplicated entity to your script, you should check to make sure you are selecting the correct one. Note that you even have this problem in the ingame editor, except in that case you don't get any identifying numbers.

2.2.1 Syntax

Note that there are a few differences between the syntax used in the ingame script editor (ISE) and the one used in Exscriptor. This is a consequence of the nature of the scripting language, which is very difficult to parse. To make my life easier, the editor recognises several special types of text:

* Variables
* Constants
* Numbers
* Strings
* Literals
* Comments

Everything else is considered to be part of a script command.

As in the ISE, variables must start with a $ sign and can include letters, numbers, and dots. Constants must not include spaces and must be surrounded by [square brackets]. Both of these are highlighted in green, as in the editor. The special value null is also highlighted green. Numbers and strings are both highlighted in blue; numbers can include a negative sign but not a decimal point, since there are only integer numbers in X3 scripts. Strings must be surrounded by 'single apostrophes' and can include any characters except more apostrophes.

The most complex type of text is the Literal, a catch-all term for anything else; this can include object commands, ship types, wares, and other various special items. Because there are so many of these, to make them easier for the editor to recognise, they must be surrounded by { curly braces }. Literals are highlighted in yellow. The compiler will complain if you make up a non-existent Literal, telling you it has found an "unresolved literal" (i.e. it doesn't know what it is). In some rare cases, real Literals will be unresolved; this is almost always because the editor has been unable to load the data file where this Literal is stored. In this case, you should read the messages when the editor first loads to see if it has had problems loading any data files.

Comments are the final type of special text; these are lines that start with an * asterisk. Comments are coloured grey. Note that some comments have unusually strange behaviour in X3 scripts: if you have commented out a command in the ISE, then that command still exists, it is merely disabled. Early versions of the Exscriptor tried to handle these sorts of commented commands, then later versions disabled it (handling them just as text), and as of v1.023, Exscriptor handles them again, including conditional commands. However, they are not always handled perfectly so to avoid any problems, it is best if you do not use commented commands.

2.2.2 Errors

Note that the editor is fairly sensitive: missing out an "=", for example, can cause a command to go unrecognised. You can check to see if a script contains any problems by compiling it. You can do this in one of three ways: select "Compile" from the Build menu, press Ctrl-Shift-B, or save the script in .XML format, which will cause the editor to automatically attempt to compile it. Errors and warnings will appear in the Status Box beneath the Script Editor, and where possible, problem lines will be highlighted in red, like so:

!fight.attack.object*

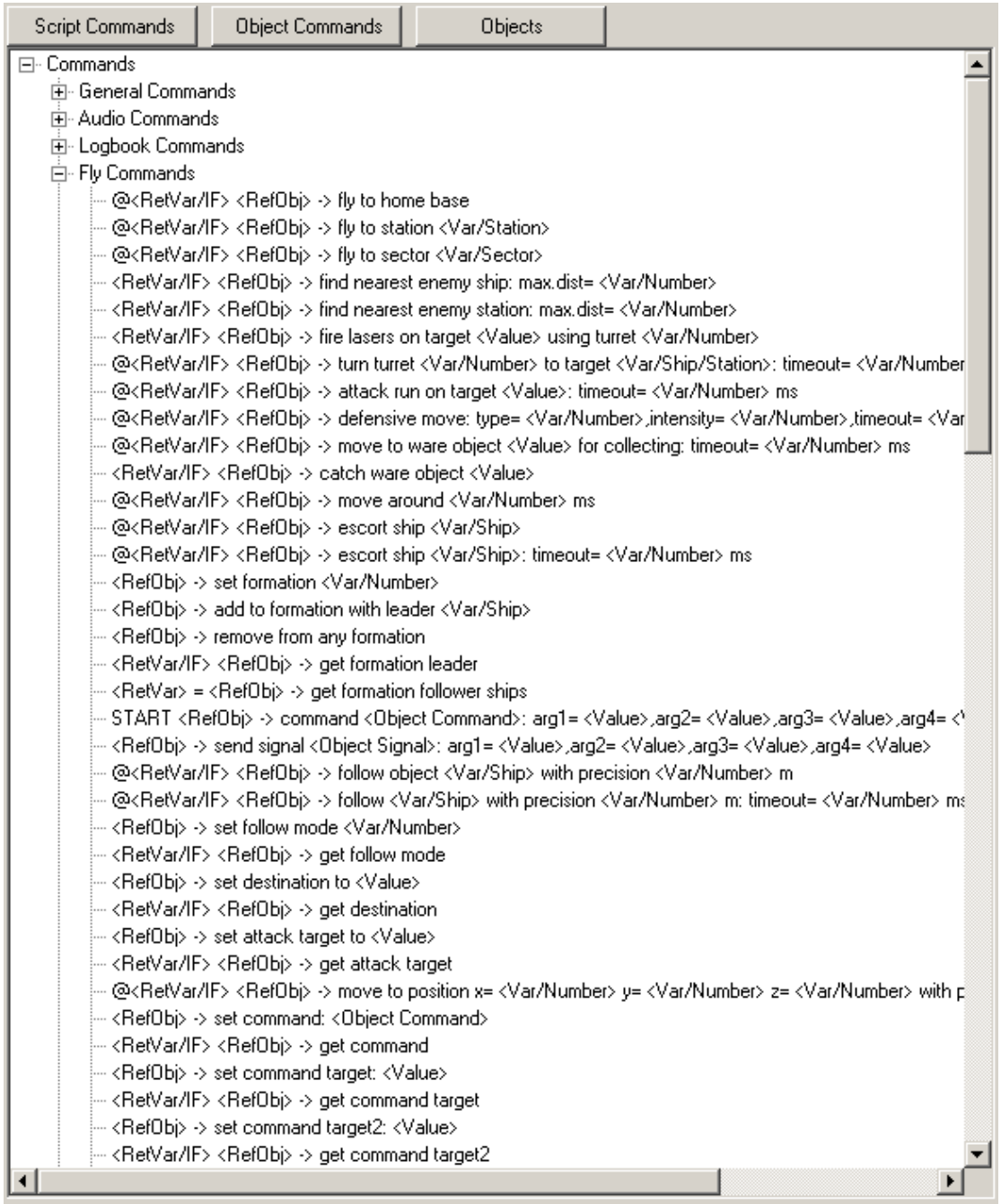
```
0001 restart:
0002
0003 skip if $victim->existserror
0004   return null
0005 if not $victim->is of class {Ship}
0006    * ES Code
0007    * skip if $victim->is of class {Station}
0008    * return null
0009    * XTM Code to allow LI class as viable targets
0010    $OK = [FALSE]
0011    skip if not $victim->is of class {Station}
0012       $OK = [TRUE]
0013    skip if $victim->is of type {Rock Beetle 15711 7_326}
0014       $OK = [TRUE]
0015    skip if $victim->is of type {Rock Beetle 15711 7_330}
0016       $OK = [TRUE]
0017    skip if $victim->is of type {Shivan Dragon 15701 7_327}
0018       $OK = [TRUE]
0019    skip if $victim->is of type {Shivan Dragon 15701 7_328}
0020       $OK = [TRUE]
0021    skip if $victim->is of type {Unknown Drone 15721}
0022       $OK = [TRUE]
0023    skip if $OK
0024       return null
0025 end
0026
0027 if not [THIS]->is of class {Kha'ak Cluster M3}
0028   if not [THIS]->get maximum laser strength
0029     = wait randomly from 100 to error ms
0030       return null
0031    end
0032 end
0033
0034 $followdist1 = 4000
0035 $sl = [THIS]->get size of object
0036 $followdist = $victim->get size of object
0037 $followdist1 = $followdist + $sl + $followdist1 + 100
0038
0039 set script command target: error
0040 $fighter.ai = [THIS]->is of class {Fighter}
0041 skip if $fighter.ai
0042    $fighter.ai = [THIS]->is of class {Fight drone}
0043 skip if $fighter.ai
0044    $fighter.ai = [THIS]->is of class {M6}
0045 skip if $fighter.ai
0046    $fighter.ai = [THIS]->is of class {M7}
0047 if $fighter.ai
0048    if not [THIS]->is of class {Kha'ak Cluster M3}
0049       = [THIS]->call script '!plugin.acp.fight.attack.object' : arg0=$victim  arg1=$
0050       return null
0051    end
0052 end
0053 $IsPlayer = [FALSE]
0054
```

```
Compile errors detected:
- Error on line 3 - Not a valid expression
- Error on line 29 - Not a valid expression
- Error on line 39 - Not a valid expression
Compile failed
```

Notice that if the compiler does not recognise a command, it will tell you that it is "not a valid expression" and tell you which line has the problem. These are errors and mean that the script has problems that prevent it from being compiled; you cannot save a script to .XML format in this state. Occasionally you might also get compiler warnings, which are less critical errors; these do not prevent a script from being saved, but may indicate a problem that still needs to be solved. For example, if you have a script call inside your script (i.e. a "call script" command) and the script you are calling does not exist,

then the compiler will generate a warning to tell you that it could not check the parameters for that script.

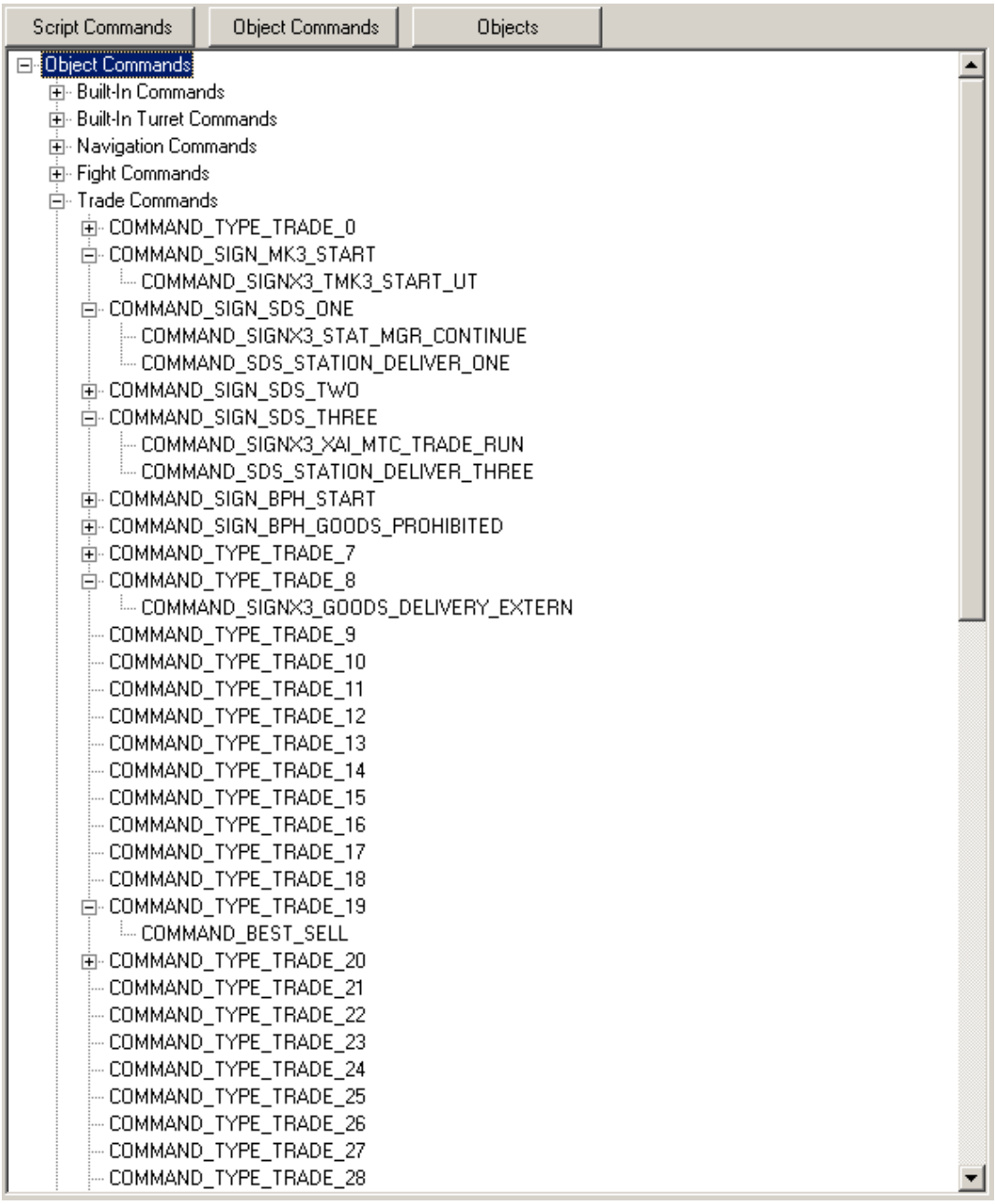
2.3 Object List



The Object List on the right hand side of the interface is there to make it possible to write scripts without memorising all 800+ script commands, not to mention the hundreds or even thousands of wares, ships, stations, ship commands and so forth. It has three modes, selectable using the buttons above it. In the first mode, the List will show all the script commands available, sorted into the appropriate categories. Double clicking on any of these commands will copy it into the Script Editor at the current cursor position; you

will then need to replace any parameters (these are the parts surrounded by <angle brackets>) with the appropriate data - variables, constants, literals etc. Ideally you would put the correct type of parameter in; putting a string where the command expects a <Var/Number>, for example, is not advised. Unlike the ingame script editor, where it is virtually impossible to put the wrong type of parameter in, the external editor does very little checking of parameters: it is up to you not to do something stupid.

In the second mode, the Object List displays all the possible Object Commands - these are ship or station commands.



These are also divided up into categories, just like the script commands. Notice how some of the commands have more commands underneath them;

the normal name of a command is the built-in X3 name, and if that command has been given a new name in a mod or in a script's language file, then that name is shown beneath it. If there are multiple such names, then it means there is a conflict - you have more than one script trying to use this command slot. I have no idea which will end up as the name used in game, but as far as the external script editor is concerned, it makes little difference which you choose; if you are concerned, use the built-in name instead. Just as with script commands, double clicking on a command name will copy it into the Script Editor, except surrounded by {curly braces}, since object commands are Literals.

The third and final mode is the Object mode:

This shows all the various objects and entities you can use in the script:

- Wares - these are things you can put in your ship, such as weapons, cargo, and command software.
- Ships - these are ship types, e.g. the Argon Buster.
- Stations - station types, e.g. Teladi Trading Station.
- Races – alien races etc.
- Object Classes - types of objects, e.g. Big Ship, M5, Equipment Dock, UFO, Space Fly. Note that the SQUASH Mine in this list has "(object class)" after it; this is to distinguish it from the Ware of the same name.
- Flight Returns - these are constants used in a lot of fight commands.
- Formations – these are constants used especially in formation flight commands.
- Data Types - the built in names for different data types. Use these if you're using "is datatyp" script commands.
- Relations - Friend, Foe, Neutral
- Serials - these are the extra bits on the names of stations, e.g. Solar Power Plant Alpha. Basically a list of Greek letter names; for some reason, Omega is included twice.
- Transport Classes - the cargo size indicators: S, M, L, XL, ST.
- Sector Names - just as it says on the tin, these are the names of the sectors in the game. You should ideally avoid using these wherever possible, for two reasons: firstly, if someone is using a custom galaxy, those sectors might not exist; secondly, using a sector name directly in a script causes X3 to load it in a different way, and sometimes this can break with externally edited scripts.

Once again, double clicking copies it into the Script Editor as a {Literal}.

2.3.1 Duplicates

You may notice that a lot of entries in the Object list - especially ships and wares - have numbers after their names. This is a workaround for a major problem any external script editor faces. In X3, there are often lots of objects with the same name; several Argon Busters, for example, or two ores named Nividium. There is no way for Exscriptor to be able to distinguish "Nividium" from "Nividium". In the ingame editor, this isn't a problem (for the game, at least - it's still puzzling for the user) since you're selecting items from a list that the game can convert internally to the correct entity. Externally, this is not possible.

So, to solve the problem, any duplicates are given unique names by adding numbers after the original name, e.g. "Argon Buster 7_6" or "Nividium 15_12". Usually the numbers afterwards represent the maintype and subtype of the object; e.g. the Argon Buster is a ship (maintype 7) with several possible subtypes, including 6; the two Nividiums include an ore (maintype 15) and an object in the "random junk" category (maintype 12) which includes special objects like "water", "cyborgs", or the "black crystal" you have to pick up as part of the plot. In that particular case you will almost always want to use the

maintype 15 one, but for the most part you'll have to figure out the correct object to choose yourself. If it's a custom object you've added yourself this shouldn't be a problem, but in the case of the Argon Buster, the only way to know for sure is to create a test script ingame creating one of each so you can actually see what they all are.

2.4 Preprocessor macros

Version 1.012 of the Exscriptor introduced the concept of *preprocessor macros*. These are "fake" script commands – commands that you can use in the Exscriptor but which do not exist in the ingame editor. The idea is that you can use these commands as shortcuts, automating common scripting tasks like looping through or declaring an array.

The preprocessor commands can be found at the bottom of the script command list in the Object List. Currently, the four commands available are as follows:

- `foreach <RetVar> in <Array>`
- `for <RetVar> = <Var/Number> to <Var/Number> inc <Var/Number>`
- `for <RetVar> = <Var/Number> to <Var/Number> dec <Var/Number>`
- `dim <RetVar> = { <Value>, <Value>, <Value> }`

The **foreach** command allows you to loop through all the items in a variable, and it is equivalent to the following script code:

```
$i = size of array $array
while $i
      dec $i =
      $value = $array[$i]
```

The **for** command cycles through a set of numbers. For example:

```
for $n = 0 to 10 inc 1
```

is equivalent to:

```
$n = 0 - 1
while $n < 10
   $n = $n + 1
```

You can use the "inc" version if you want to count up and the "dec" version if you want to count down. Notice that the for loops will stop *before* they reach the final value – so a for loop from 0 to 10 starts on 0 but finishes on 9. In other words, the upper boundary is not inclusive, but the lower boundary is.

Finally, the **dim** command declares and initialises an array with a set of values. You can have as many values as you like, not just three (as long as it's more than one). For example:

```
dim $array = { $x, $y, [TRUE], 'jelly bean', 123 }
```

is expanded to

```
$array = array alloc: size=5
$array[0] = $x
$array[1] = $y
$array[2] = [TRUE]
$array[3] = 'jelly bean'
$array[4] = 123
```

When you use a preprocessor command, the Exscriptor will automatically expand it before compiling. However, there is also the option (in the Options) for "Enable macro post-processing", which will compact the commands back down if compilation is successful. So for example, the following code:

```
for $n = 10 to 0 dec 2
      $x = $x + $n
end
```

gets expanded to:

```
*- for $n = 10 to 0 dec 2
$n = 10 + 2
while $n > 0
   $n = $n – 2
   $x = $x + $n
end
```

prior to compilation. The code is then checked and compiled, and if post-processing is enabled and the compiler detects no errors, the Exscriptor will reverse the macro expansion back down to the original form:

```
for $n = 10 to 0 dec 2
      $x = $x + $n
end
```

making it look as if the macro expansion never took place. You can of course turn this feature off, in which case compiling will always expand the macros and they will remain expanded. This happens automatically if any errors are detected by the compiler, allowing you to check *all* of the code.

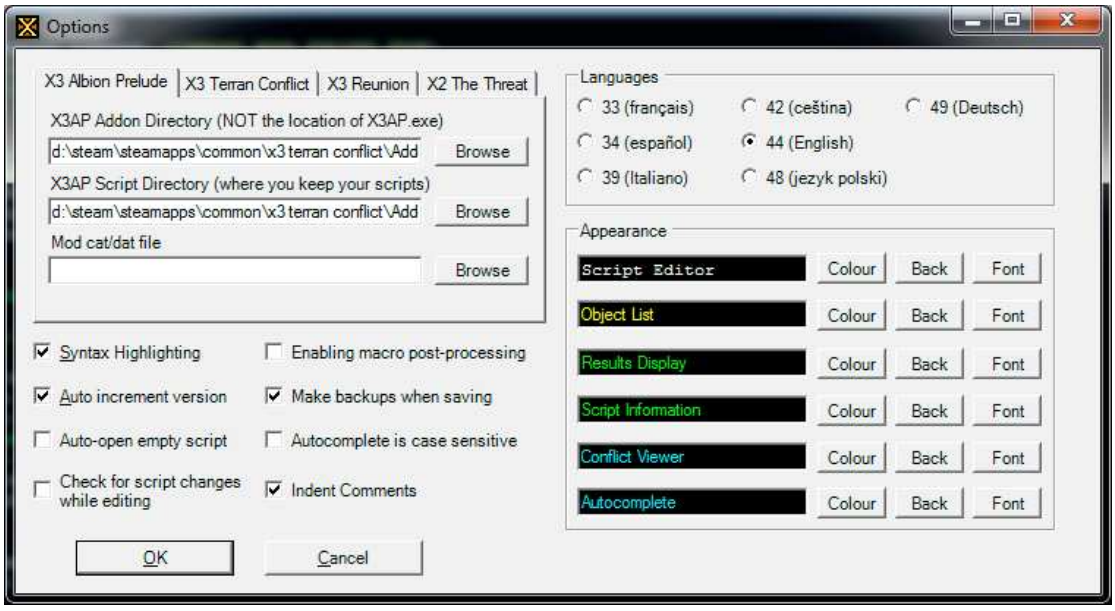
2.5 Options Menu

The Options Menu allows you to set the options which determine the Exscriptor's behaviour. The menu itself is shown on the next page. The first tab box allows you to locate the X2/X3/TC/AP directories and the directories where you keep your scripts. The first box (game dir) especially is very important as without this path, the Exscriptor will be unable to find the game data files and thus will not be able to load the data. Usually, the script directory will be inside the X2 / X3 / TC / AP directory, so typing in the game directory box will automatically be copied into the script directory box; however, you can always override this yourself by typing or selecting a new script directory. You can click the Browse buttons if you want to locate the directories yourself rather than just typing them in.

This section also includes a field where you can locate a mod cat/dat file to use (e.g. one in the mods directory). This allows Exscriptor to load in mod data without having to unpack it first. However, you will need to restart the Exscriptor for this to take effect.

Next are some check boxes that determine user interface behaviour:

- *Syntax Highlighting* – this turns syntax highlighting (colouring) on/off.
- *Auto-Increment version* – this increments the script version every time you save.
- *Auto-open empty script* – when enabled, if you close all the scripts, a new empty one will be created.
- *Enabling macro post-processing* – when on, macros will be recondensed after compilation (assuming it was successful).
- *Make backups when saving* – copy the script file to a BAK file of the same name before saving to make a backup. Note that it is up to you to then restore the file (by deleting the new, incorrect one and renaming the old one).
- *Autocomplete is case sensitive* – if checked, the autocomplete box will be case sensitive.
- *Check for Script Changes* - if checked, Exscriptor will prompt you to update when it detects that a script file you are currently editing has been changed outside the editor. Note: this can be a bit buggy and sometimes will continue to prompt even when there are no changes. If this happens, just turn this option off.
- *Indent Comments* - you can also choose whether or not comments get indented with the rest of the code.



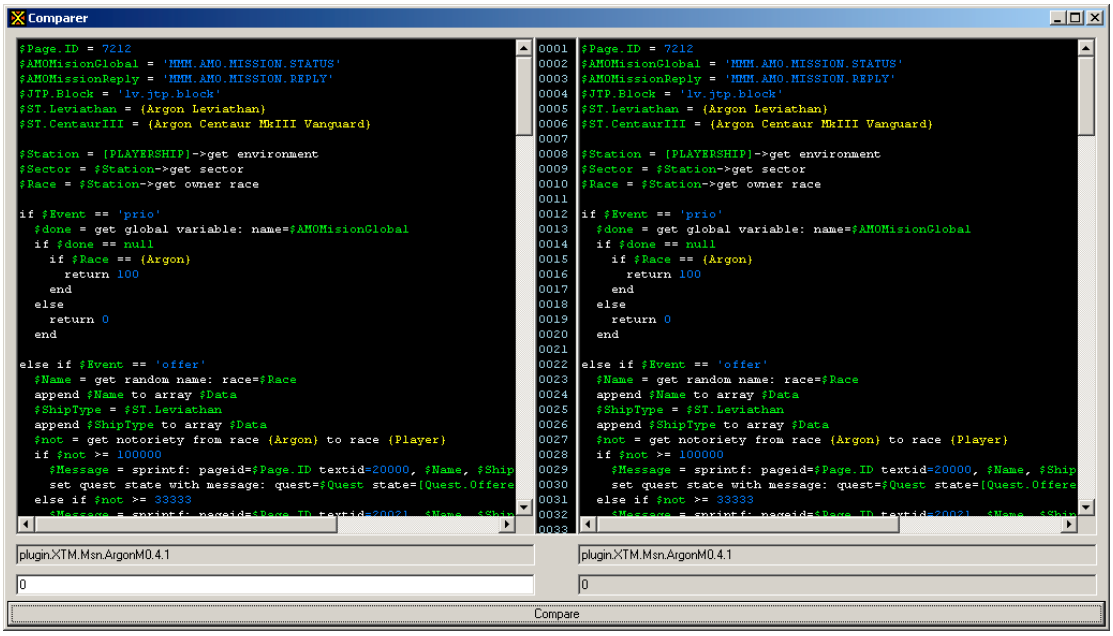
Next, there is also the language file selection box, allowing you to select which set of data files to use. This is the same as selecting the language directly from the menu.

Finally, on the right hand side is a set of boxes and buttons to allow you to change the appearance of the Exscriptor. The three buttons for each one are the foreground colour, background colour, and the font; you can change these for the Script Editor (though the foreground – i.e. text – colour will only show if you are not using syntax highlighting), the Object List, the Results/Status Box, the Script information beneath the Object List, Autocomplete and the Conflict Viewer.

Your options will be saved when you close the Exscriptor, so you should only need to set your preferences once; the Options menu will automatically appear when starting the Exscriptor if it cannot find these saved settings.

2.6 Script Comparator

As of V1.014, there is now a script compare function in the Exscriptor. You can use this to look for (text) differences between two scripts. To do so, first open a script as normal (this will be the script you want to change) and then choose "Compare with" from the View menu. You will be asked to select another script file, and then a new window will open comparing the two, as below:

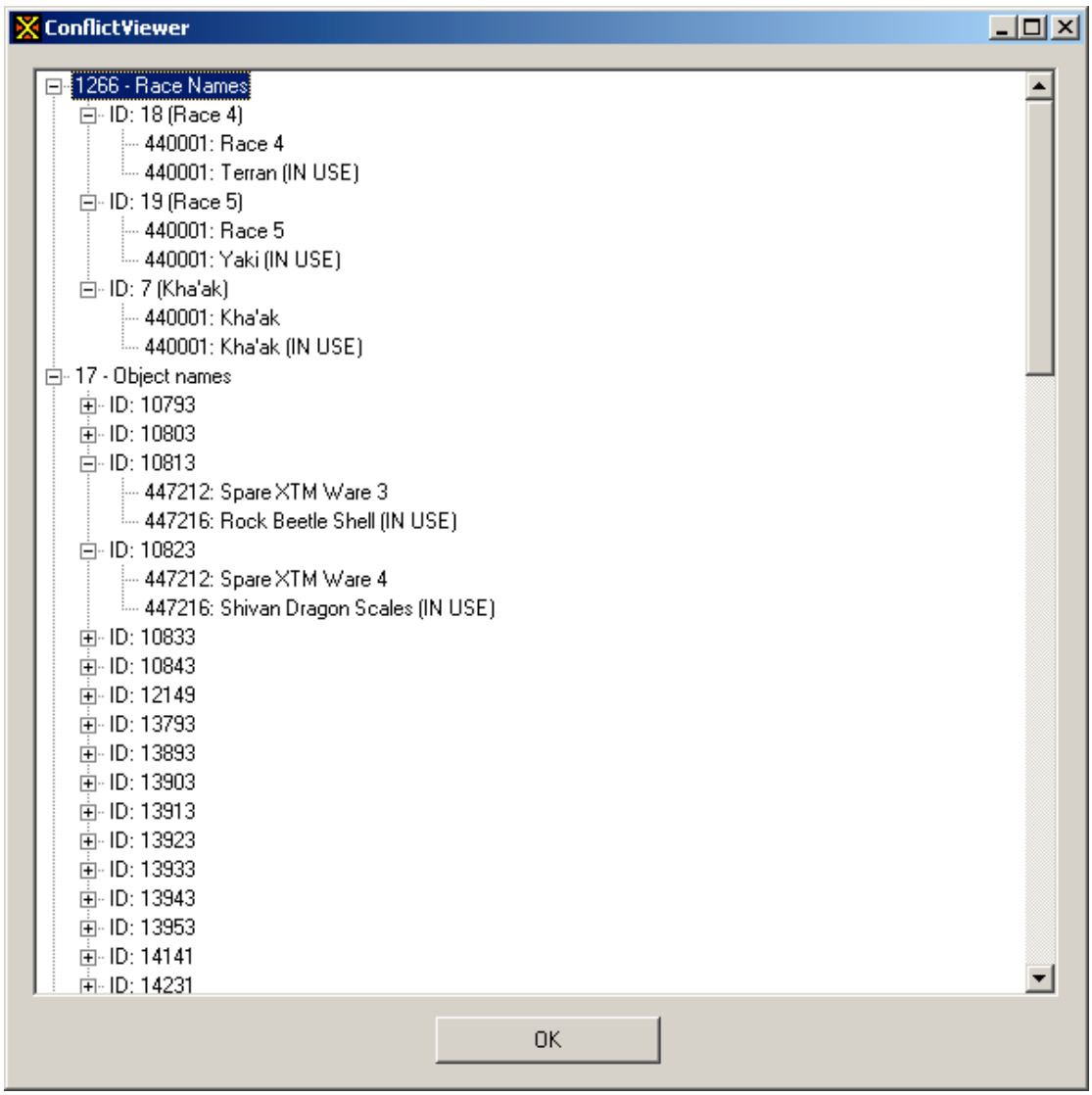


Differences will be highlighted in red. As you scroll down using either script window, the other window should scroll down at the same time, allowing for easier comparison. You can change the text in the left-hand window but whenever you change text it will need recomparing – simply press the big "Compare" button at the bottom. You cannot edit text in the right-hand window, however. When you close the window (which you must to return to the main screen), your changes will be reflected in the original script. At the bottom, the script names and command numbers are also displayed; you can only change the command number of the left-hand script.

2.7 Conflict Viewer

From V1.025 onwards, in the Tools menu there is a new utility called the Conflict Viewer. This opens up a simple window, shown below, that indicates all of the conflicts in the language files read in by the Exscriptor. Race names, Object names (ships, wares etc), Commands, and Sector Names are all included.

The structure of the tree is as follows: firstly is the Page ID. For example, 1266 contains Race Names and 17 contains Object names. Beneath each of these are all the IDs in that page that have conflicts, i.e. more than one file using the same ID. If the ID is present in the main language file (e.g. 440001.xml) then the 440001 name will be used to give you some more information on what the ID means. Underneath each ID is a list of every file that uses that ID and what name they use. For example, in the screenshot below, you can see that two files – 447212 and 447216 – use the ID 10813 in the Object Name page. The value used by the Exscriptor is marked IN USE (and is from the last file read in).



2.8 Support for multiple games (X2/X3/TC/AP)

Version 1.031 of the Exscriptor introduced full support for X2. This means you can now use the program with X2 as well as with X3, including loading the X2 game data files and loading/creating X2 scripts.

V1.2.0 added X3TC support and V1.2.40 also added X3AP support. However, because there are several differences between scripts from different games (a lot less available commands for instance – X2 has less than half as many as X3AP), and because the game data is obviously very different, you cannot edit both types at the same time. Instead, the Exscriptor has multiple modes: an X2 mode, an X3 mode, an X3TC mode, and an X3AP mode, accessed through the Tools menu. It can function in only one at a time. Attempting to load an X3 file if you are in X2 mode will cause an error, and similarly loading an X3TC file in X3 mode etc; loading a script from an earlier game in a higher game mode is allowed, but be aware that there are still some rare differences that may cause problems in this case.

Changing modes is done via the Tools menu – simply choose "X2 Mode" or "X3 Mode" etc. Obviously this requires that you have set the appropriate directories in the Options menu, otherwise the Exscriptor will not be able to find the game data.

IMPORTANT
To use X3AP, you also need to make sure you have the path to an X3TC install set up. The program will prompt you if you forget to do this.

2.9 Script file updating

Version 1.2.32 adds a new awareness to Exscriptor so that it will now tell you when a file you are working on has been updated. If you have not edited the file yet, it will ask you if you want to refresh the file (you can tell it to ignore it if you want). If you have edited it yourself, it will not tell you until you try to save the file, at which point it will warn you that it has been updated since it had been loaded originally.

This functionality sometimes gets a bit... over-enthusiastic, in which case you can turn it off in the Options menu.

3 Using the Command Line version

As of V1.2.32, Exscriptor also has a command line version available called `ExscriptCmd.exe`. This is a simpler version of the tool capable of compiling and decompiling script files from the command line, though it does not have the full range of features (e.g. the preprocessor) that the main GUI has.

Command line usage is as follows:

```
exscriptcmd <game dir> <input file> [output file] [options]
```

- <game dir> is the directory of the game. The tool needs this to be able to find the game's data files. It should be a path, e.g. C:\X3TC.
- <input file> is the name of the input file. You can use wildcards here, e.g. *.txt or *.xml, or even script?.dosomething.*.xml etc. Valid input files are .txt files, .xml files, or .pck files; the tool will figure out what to do with those files based on the extension: .txt files are compiled to .xml, .xml or .pck files are decompiled to .txt.
- [output file] is an optional output file. If you have a single input file, you can provide an output file to produce output to a different filename. By default, the tool will use the same file name but with a different extension.
- [options] are one or more options, as described below:
  - `mod=<filename>` Allows a mod .cat/.dat to be used
  - `game=<X2|X3|X3TC|X3AP>` Allows compiling to a different game format (X3AP is assumed to be the default).

The tool should provide messages and warnings etc as appropriate when it runs. Note that it still takes a few seconds to load the game data, but this is only done once with each run, e.g. if you wish to compile many files, use wildcards rather than run the tool multiple times.

Note that the command line tool still requires all the same data files as the main GUI version, including both the parameter .XML files and the various .DLL files, so do not remove these!

Remember that the text files used must be in the text file format used by Exscriptor. The main difference is that it includes a header to add the extra information needed. For completeness, the header for the text file is as follows:

```
test.script.1                ← name
0                            ← command
Last mission for M0          ← description
194                          ← version
1                            ← number of arguments
name|value|9|name of ship    ← argument
---END-OF-METADATA---
```

4 Known bugs and other pernicious problems

> **Debugging**, *v.* a programming activity analogous to removing all of the needles from a haystack while looking through a microscope.

As every programmer will tell you, there are no bug-free programs. This one is unfortunately no exception; quite the opposite, in fact. Despite my efforts, there are still plenty of problems – some important, some not. Hopefully any errors will be handled without Exscriptor crashing; in addition, in the case of something going wrong, an `errorlog.txt` file will be created in the Exscriptor directory. The information within will hopefully help diagnose the error.

The main (known) bugs are listed below.

4.1 Bugs in the user interface

4.1.1 Syntax Highlighting

This doesn't always work quite right. For the most part it will colour everything correctly, but occasionally it gets a bit confused, either refusing to colour something or colouring something when it shouldn't. For example:

```
$message = sprintf: pageid=$page.ID textid=2000, $arg1, $arg2,
null, null, null
```

It also does this when using numbers as an array index, e.g.

```
$array[5]
```

There are other, less cosmetic problems. For example, occasionally you may also get strings of numbers appearing for no reason when you delete or add text, a bug I have never been quite able to eliminate completely.

4.1.2 Flickering

Sometimes the text in the main Script Editor window will flicker; this often happens when you undo/redo or insert something from the Object List, for example. This is just the code being updated and rehighlighted and is (currently) unavoidable.

4.1.3 Undo / Redo

Again, partly due to the syntax highlighting, the Undo/Redo behaviour is sometimes a little strange; often it takes a couple of undos or redos. If the highlighting is turned off this behaves more correctly. Also, you cannot undo/redo more than the last 50 changes (in each script).

4.1.4 Copy/Paste

Copying and pasting using the usual shortcuts (Ctrl-C, Ctrl-X, Ctrl-P) only works in the Script Editor box. Copying or pasting from the Status Box should work too, but nowhere else. If you wish to copy/paste you should also be able to use either the alternative shortcuts (Ctrl-Insert, Shift-Delete, Shift-Insert respectively) or the mouse right-click menu – usually at least one of the two methods work.

4.1.5 Red error highlighting

When the compiler detects an error, it does not always succeed in highlighting the line in red. Sometimes this is because the compiler does not know which line the error occurred on; sometimes because it reaches an error earlier on that means it cannot continue.

4.1.6 Interrupt signs (@)

Interrupt signs are completely ignored by the compiler and the decompiler, mainly because they're a pain. Commands can still have them, and you can still type them, but you don't need to – they won't be displayed when you load a script and they won't be included by the compiler, either.

4.1.7 Find/Replace

Despite my best efforts, the Find/Replace dialogue boxes *continue* to frustrate me by manifesting a seemingly infinite number of small but annoying bugs. No matter how many times I fix it, something else goes wrong. So do not be surprised if you find it acting strangely from time to time; having to click "find" more than once before it realises another item exists is not uncommon, for example.

4.1.8 Non-standard system fonts/sizes

Another long-running issue has been with non-standard font sizes or types. Hopefully this should be fixed in V1.2.40 but you may still experience some issues, e.g. text boxes being slightly covered by other controls etc.

4.1.9 Using CTRL + mouse wheel to resize the font

This doesn't work, and never has, because it's never been clear to me how it works or how to get control over the process (it seems to be built in functionality hidden from the programmer). Unfortunately you have to use the Options Menu to set the fonts instead.

4.2 Bugs in the compiler/decompiler

These are potentially more serious than the interface bugs. However, I tested the compiler on over 1200 scripts, including the XTM scripts, comparing the resulting code against the original; a few couldn't be opened (and thus couldn't be compiled), and of the rest, only about 1% showed any differences, mostly only minor difference – using ` instead of ', for example, or having variables in a different order. Still, there are some outstanding problems:

4.2.1 Random value from 0 to …

This command comes in two forms: from 0 to a max value, and from a min value to a max value. However, when the minimum value is set as 0, it is impossible to distinguish between the two commands. The compiler does not always choose the same one as the ISE in these cases, leading to a minor difference in the code. Fortunately, this makes no difference to the functioning of the script, but if you wish to change the lower value in the ISE you may need to change the entire command.

4.2.2 Apostrophes

For various reasons, the compiler replaces normal ' apostrophes with ` ones (one reason being that you cannot have ' in a string in Exscriptor). This can sometimes lead to differences with the original.

4.2.3 The Infamous and Enigmatic 1185

There is one command in the script editor, with ID 1185, that plagued me incessantly. According to the language files, the command is this:

```
START %0->command %1 : arg1=%2, arg2=%3, arg3=%4, arg4=%5
```

The problem is that this is exactly the same as command 514, also:

```
START %0 command %1 : arg1=%2, arg2=%3, arg3=%4, arg4=%5
```

Or so it looks, from the language file. It turns out that internally – and in some scripts – the 1185 version can take *five* arguments, i.e.:

```
START %0->command %1 : arg1=%2, arg2=%3, arg3=%4, arg4=%5, arg5=%6
```

Except it doesn't always, and when it doesn't, that missing last parameter is apparently ignored. I've tried to add in special code to handle this command, but to avoid any problems, it's best to use the 4 argument version (514) if possible.

4.2.4 Bad XML

Some scripts evidently don't quite conform to standard XML as well as they should, and Exscriptor is much more sensitive about this than X3 apparently is. This happens especially when you use special characters like "<", ">", or "&" somewhere in your script. If you wrote the script in the external script editor, this shouldn't be a problem, but if it comes from the ISE or somewhere else, it can sometimes cause load errors.

Note this can also apply to the language files too - if you get an error like this:

- Error while reading xml file H:\X3 Reunion\t\441234.xml: An error occurred while parsing EntityName. Line 12, position 34.

It means you are the proud owner of an invalid XML file. In these cases, the Exscriptor *will not be able to load the file* – and so anything important in it will also be absent. You will have to fix the problem (usually an & or something) manually so that the file can be loaded. To do this you just need to convert the symbols, e.g.:

& → &
< → <
> → >

You can also check to make sure that an XML file is valid by loading it in a web browser; most will tell you if the file is invalid in some way. X3 itself seems to be very relaxed when it comes to XML and accepts a lot of errors.

4.2.5 XML Output differences

If you open up an XML script file in a web browser, it will show you the code (as long as the `x2script.xsl` file is present). Files produced by Exscriptor exhibit a few minor differences to those produced by the ISE. Literals are often a little different, e.g. whereas the ISE might say "Buster", the externally produced script will say "Argon Buster"; similarly, duplicate literals will include their identifying numbers in externally produced scripts. The ISE also tends to produce scripts with two equals signs, e.g.

```
$Name = = get random name: race=$Race
```

Externally produced scripts reduce this to a single equals but occasionally (rarely) it seems to omit any equals sign. Finally, spacing and the green highlighting is often subtly different. However, all of these discrepancies are purely cosmetic and have no effect on the running of the script.

4.2.6 Unresolved literals

You may get an error that tells you that there is an "Unresolved literal". This means that you have either typed in a literal (the things in { curly braces }) that is not recognised, e.g. a ship type that doesn't exist, or that you have loaded a script that contains an unrecognised literal; in which case, it will

appear as {?}. This is almost always because of a missing or unreadable language file (or, occasionally, something missing in one of the Type files). If you're loading someone else's script, there's not much you can do about this.

4.2.7 No return value

If you're using a command that expects a return value but you want to ignore it, you still need to put the '='. For example:

```
$retval = [THIS]->call script 'any.script':
```

is fine, but

```
 [THIS]->call script 'any.script':
```

is not; you need to add the =

```
= [THIS]->call script 'any.script':
```

otherwise it won't be recognised.

4.2.9 Commented commands

As of v1.023, the Exscriptor once again handles commented commands (commands prefixed with a *) during compilation, including conditional commands (if, while etc). This means you can "uncomment" them again in the ISE. I restored this feature to deal with the problem of XML symbols (especially ">") being corrupted by X3. However, this feature is not guaranteed to be 100% perfect – there's a lot that can and probably will go wrong. One known problem is that conditional commands do not work if they have more than one space after the * (e.g. "* if $config"); in these cases they get converted to normal text comments. My advice is to avoid using commenting out commands wherever possible.

4.2.10 Using double quotes in comments

In short, **please do not do this**. This applies to X3 as well. Comments are usually stored in script files as simple text within double quotes, e.g.

```
<sval type="string" val="Your comment here" />
```

If you put double quotes " " around your comment, these get doubled:

```
<sval type="string" val=""Your comment here"" />
```

which breaks everything because the file is no longer valid XML.

4.2.11 Variable checker

Exscriptor will attempt to check when a variable is being used before being defined or alternatively not being used at all. However, it frequently gets this wrong and is easily fooled. Apart from instances where it simply gets it wrong (expression assignments on the first line or two often suffer from mistaken "variable is being used before being assigned" bugs, for example), there are also situations where it is prohibitively difficult to check for sure, e.g.:

```
gosub subroutine
$a = $b
return $a

subroutine:
$b = 10
endsub
```

This situation will cause the Exscriptor to believe that $b is being used before being defined, when in fact it's defined later, in the subroutine. Short of analysing the program flow, this is pretty much unavoidable unfortunately.

In summary, the variable checker warnings should be taken as just that: *warnings*, advice to check to make sure a variable is being used correctly. It does not necessarily indicate an error on your part.

4.2.12 Ambiguous names

There are *many* ambiguous names in X3 (especially TC). For example, the words "Navigation Relay Satellite" can refer to a ship, a ware, and an object class. Though Exscriptor can handle duplicates of the same type (e.g. the many Argon Busters present in X3) it struggles to handles duplicates of different types. There is a limit to how far this can be solved in V1.2 but hopefully V2 will handle this sort of thing better.

In the mean time, I have tried to sort out the worst ambiguities (laser towers, satellites, fighter drones etc) so that they are all separate. Wares normally have their language ID numbers after them, ships will have maintype/subtype and always a race name (e.g. "Argon Navigation Relay Satellite"), and confusing object classes will sometimes be appended with "(object class)". Hopefully that will sort out the main problems, but there may be others too.

4.2.13 Escape characters in strings

Strings in the scripting engine are delimited by apostrophes, i.e. the ' character. If you wish to include an apostrophe within a string, then you should use the backslash as an escape character like so:

```
$x = 'This is a \' valid string'
```

Note, however, that the backslash only functions as an escape character in this specific instance. No other escape character sequences are recognised and the string:

```
$x = 'This is a \\ string with two slashes in it'
```

is exactly as it appears and will contain two backslashes, not one.

4.2.14 Loading mods in cat/dat files

This has historically been somewhat problematic in Exscriptor, though the functionality should work. However, if the editor fails to pick up your modded TFiles correctly, there are other options you can try, e.g. setting up your mod as a false patch or simply unpacking the mod to the normal directories. In both cases, Exscriptor should (in theory) pick up the extra files correctly.

5 Behind the scenes

5.1 Introduction

This section covers the sort of technical, behind the scenes information that most people are unlikely to be interested in. However, since this project has taken a significant amount of time and effort, and since along the way I have acquired a lot of information (most of it learnt the hard way), I feel entitled to write it all down somewhere. Besides, it might help someone else produce a better editor in the future.

5.2 The X3 Files

There are several types of file that are necessary to understand the scripts in X3. First, obviously, are the scripts themselves (described next). However, the part of the script that does all the work, the codearray, contains virtually no text – it's mostly numbers and XML. These numbers refer to things in other files, and so to understand the scripts, you need to understand the other files too.

5.2.1 Language Files

Most important are the language files. These are the files found in the "\t" directory of your X3 folder (or in the 01.dat, 02.dat, 03.dat … etc files, which can be unpacked). The language files have names like 440001.xml or 440001.pck. The first two digits are the language (44 being English), the rest the file ID which is used by the "load text" scripting command to load them. These are simple XML files containing the text definitions for virtually everything in the game – wares, sectors, ships, script commands, everything. Almost everything will have a language ID which is an index into some part of one of these files; for example, as mentioned earlier, this command:

```
START %0 command %1 : arg1=%2, arg2=%3, arg3=%4, arg4=%5
```

has the language ID 514, and you can find it in the script command section of 440001.xml, page 2003.

The important parts of the language files – i.e. the parts you need to know to understand all the scripts – are as follows:

PAGE	CONTENTS
7	This contains the names of all sectors in the game.
17	All wares, objects, ships, stations etc. Odd numbers are usually the name and even numbers are the accompanying description. Note that this is not usually the number used inside a script to refer to these items, however (see maintype and subtype, below).
1266	Race names – Argon, Boron etc

2000	Script types, e.g. "Value", "Var/Number", "String" etc. These are used in the descriptions of script commands, e.g. "RetVar" is the return value of a command.
2001	Script operators – logical and mathematical operators, like +, *, AND, OR, and so forth.
2002	Script constants – things like [THIS], [TRUE] and the various [Find] constants.
2003	Script commands – the commands themselves, like "START %0 command %1: arg1=%2, arg2=%3, arg3=%4, arg4=%5" etc. The percentage signs represent parameters.
2004	"Hidden" script commands – nothing exciting, just things like "end", "else," and the comment symbols. These are not present in the main part of the codearray, hence I call them the "hidden" commands.
2005	Script command categories – General, Flight, Trade etc
2006	Script object classes – Ship, Big Ship, Station, M4, that sort of thing.
2007	Script jump commands – called return variables or RetVars. These include the humble "=" but also conditionals like "if", "while", and "skip". These aren't true commands but actually more like modifiers for expressions.
2008	Script object commands – ship / station command names, e.g. COMMAND_RETURN_HOME.
2009	FLRETs – Flight command return values, constants used in flight commands (especially combat commands).
2010	Long version of script object commands, e.g. COMMAND_RETURN_HOME has long name "Return home". This is what you see in your ship's menu in the game.
2011	Short version of script object commands – "FlyHome", for instance.
2012	Formation constants – delta, line etc
2013	Data types – the names of the actual data types used in scripts, e.g. DATATYP_INT is a number.

Some of these pages have "extensions" that have been added in later patches; these generally have the same page ID, but add 300000, e.g. 300007 contains the new sector names. You need to check for extended versions of all the above, e.g. 302003 contains new script commands.

In Exscriptor, all of this data is stored centrally when the program first loads, so it can decipher the scripts. It checks all language files (of the same language), attempting to load the contents of them all; this means it can include files added via mods or new scripts.

Some entries in the language files are actually references to other entries – these look like this: {17, 1234}, which is a reference to page 17, id 1234. Exscriptor can handle these to an extent, but if you have a reference to a reference then it starts to get confused!

5.2.2 Type Files

The other set of files you need to understand are the Type files, found in the "\Type" directory (at least when unpacked). Various editors, like the X3 Editor, allow you to change these files, which contain information about ships, wares, and stations. Whereas the language files only contained text, these contain data about the various entities – speed of ships, shields on a station, that sort of thing. The important ones are:

TYPE FILE	MAINTYPE	CONTENTS
TWareT	16	Contains technology objects – duplex/triplex scanners, for example.
TWareM	15	Contains minerals/ores – nividium, ore, and silicon.
TWareF	14	Processed foods – Cahoonas, BoFu, Space weed etc
TWareB	13	Basic foodstuffs and other biological stuff – Delexian wheat, for instance, or Argnu Beef
TWareN	12	Random junk, typically plot or mission related – water, Black Crystal, Vacuum cleaners, Hand Weapons, Wimbli's Trident
TWareE	11	Just energy cells
TMissiles	10	Missiles, obviously
TShields	9	Shields
TLaser	8	Lasers
TShips	7	All ships
TFactories	6	Normal factories, shipyards
TDocks	5	Trading docks, equipment docks etc

Each of the items in these files will have a specific *maintype*, which is essentially the category they fall into. In addition to the maintype, each item has a *subtype* – effectively its index in its Type file, so the first item has subtype 0, the next subtype 1, and so on. However, this count does not include comments (beginning with //) or lines that have less than six entries. The only other thing of importance (for scripting purposes) in most of these files is invariably the 7th entry in each line, and this is the index into the language file (page id 17 – wares & ships etc).

In some cases, extra info is needed; for ships and stations, we also need the race (the 46th entry in TShips, 14th in TDocks/TFactories). For ships we need the variant (Vanguard, Sentinel etc), which is the 51st entry (and needs 10000 adding to it to get the entry into the Language file); for stations, we also need the cargo size (M, L, XL etc), which is the 18th entry and is 2 (M), 5 (L), or 10 (XL).

The maintype and subtype are how scripts refer to all ships, wares, and stations. The two numbers are combined into a 4 byte int: the lower two bytes contain the subtype and the upper two bytes contain the maintype. Thus the Argon Buster (or one of them, at least), which has maintype 7 (ships) and subtype 6, will have the following number as its identifier:

```
                    7 | 6
0000 0000 0000 0111 | 0000 0000 0000 0110
                  = 458758
```

Rather than use this as the unique identifier for duplicates, however, I used the maintype and subtype since it's easier to understand. Even so, it's a complex process converting 458758 into "Argon Buster":

1. Break down the number into maintype / subtype
2. Obtain the appropriate object list from the maintype (ships, in this case)
3. Find the language ID from the subtype (3141)
4. Look up that entry in the language files to get the name

Doing it in reverse, when compiling a script, isn't much easier.

Incidentally, the common error "Unresolved literal" occurs when this process breaks down; if the maintype/subtype combo is not present, or the language ID is missing, then the decompiler or compiler won't be able to identify the literal and will complain; when loading a script, the literal appears as just {?}.

5.2.3 Sectors

There is one other type of index: sectors. Sectors are exceptions to virtually every rule in the script file. Firstly, if you include a "raw sector" – a reference to an actual sector, like "Argon Prime", then a special flag is set and the game has to load it differently. Secondly, whenever a script command mentions a data type, it will use the ID from page 2013 – *except* for sectors, whose data type is 65544 – 2^{16} + 8, the normal datatype for a sector. I assume this is to make it easier for the game to detect raw sectors. And finally, the sectors themselves use neither a normal language ID nor the maintype/subtype combo; instead, their ID is based on the coordinates of the sector. For example, Home of Light is at coordinates X = 2, Y = 5 (if you start with Kingdom End being 1,1). The Y coordinate makes up the rightmost two bytes, the X coordinate the leftmost, and we need to subtract 1 so it starts from 0:

```
                    1 | 4
0000 0000 0000 0001 | 0000 0000 0000 0100
                  = 65540
```

Fortunately, these are relatively easily related back to the language file ID (and therefore the sector name), since they're stored by coordinate in the language file. All sectors in the language file have IDs starting 102, then the Y coordinate, and finally the X coordinate (two digits each). So Kingdom End is 1020101, and Home of Light is 1020502.

5.2.4 Exscriptor-specific files

There are also couple of types of data files specific to Exscriptor, which can be found in the same directory as the editor. The first are the {X2/X3/X3TC/X3AP}CommandList.txt files, which contain all the script commands listed in their correct categories and with their correct parameters shown (thus differing from the lists in the language file). This is used to display the commands in the Object List.

The other set of files are the {X2/X3/X3TC/X3AP} CommandParameters.xml files. These is much more important as they get used by the decompiler and compiler to help determine which parameter goes where – which parameter is a RefObj, which is a RetVar, and so forth.

It is actually possible to add new commands to Exscriptor by editing these two files, though if the commands were at all unusual (e.g. a new type of call command) this would not necessarily work.

5.3 Structure of the Script XML files

5.3.1 Basics

Once again, I would like to give thanks to all the people who posted in this thread:

http://forum.egosoft.com/viewtopic.php?t=89990&postdays=0&postorder=asc&start=0

without whom this whole endeavour would probably not have been possible. Much of the information below was based on that thread; the rest was gleaned via experimentation and just generally poking about. There are still some things that are a bit uncertain, and definitely many things that are inconsistent, but for the most part the following information should be accurate.

So, to begin. Inside a script XML file, you have several main parts. At the beginning is the script's metadata, namely:

- the script's name;
- its version number;
- the script engine version (more on this in a moment);
- the description of the script.

The only tricky one here is the engine version; the rest is accessible from the ISE (ingame script editor). This value presumably tells the game what version of the engine the script is written for. Versions beginning with 2 are for *X2: The Threat*, e.g. 25 is (I believe) X2 version 1.4. Some of the old X2 scripts were moved across to X3 and you can still see these older values in their natural habitat; for the most part, they're still compatible with X3 (though I suspect the reverse is not usually true). Versions beginning with 3 are for *X3: Reunion*; 32 appears to be X3 version 2.0, and 33 is version 2.5. In any case, as long as you don't try to use a newer script in an older version, I don't think this number has much of an effect on anything.

The next section of the file is for the script arguments. Each argument has an index, a name, a type (from the "script types" – page 2000 of the language file), and a description. These serve as variables in the rest of the script, their IDs taking precedence; if there are four arguments, the first variable will have ID 4 (it starts from 0).

Next is the SourceText. This is the bit you see when you open up the script file in a web browser. Each line of code is represented separately: first the line number, then the indentation, then the code itself, suitably highlighted (<var> is green, <text> is white, <comment> is the grey comment colour). Note that symbols are usually changed, e.g. ">" becomes `>` and "&" becomes

`&` etc. Also, all spaces – including those in the indentation – are replaced by ` `.

Of course, all of this information is cosmetic. You can take it all out of a script and the game will still accept it as long as you remember to include the most important part: the codearray.

5.3.2 The Codearray

The codearray is essentially ten sets of numbers, and it duplicates all of the information already contained in the file in text format in a form more easily read in by X3. The ten entries are as follows:

1. Script name. This is the important one, and determines how it shows up in game.
2. Engine version (e.g. 33 for X3 2.5, 25 for X2 1.4)
3. Script description.
4. Script version.
5. Loading flag (see below).
6. List of variables in the script – just their names, their IDs are implicit based on the ordering.
7. The codearray within the codearray: this contains the script commands themselves.
8. Script arguments – an int and a description. The int is an index into the variable list (which gives you the name).
9. "Hidden" commands – things like "end", "else", empty lines, and comments.
10. The command name – usually a number (which in turn is an index into the language file, page 2008), but occasionally a string.

The complex ones are #5, #7, and #9. The fifth entry determines how (or indeed whether) X3 loads this script. Usually, in 99% of cases, this is 0 and has no effect; if the script contains a reference to a raw sector (like referring directly to Home of Light), the this value is 2, and presumably X3 waits until it has loaded all the sectors in before reading in this script. The general theory is that if this value is 1, then the script is not loaded (technically, it's not even saved) because it contains a reference to an active object – a spaceship flying around somewhere, for example. This is impossible to check unless the game is already running so the script cannot refer to it.

5.4 A Detailed Look at the Codearray

The codearray-in-the-codearray is the most important part, as this contains all the script commands themselves. Each command is represented as an array with, in general, the following structure:

- First is the command ID, which is a language file index
- Next is usually the RefObj – the object to which the command is applied. For example, in `[THIS]->get sector`, [THIS] is the RefObj. It's easily recognised because it has the arrow next to it (in programming terms, this is like calling a member function using an object reference or pointer). This comes in two parts – first is the type identifier, almost always either 131074 (for variables) or 131075 (for constants, like [THIS]). I've never been quite sure what these are supposed to refer to. The second part is either the variable ID or the language file ID of the constant.
- The third element is usually the return variable, this time only a single entry. It can never be 131075 since you cannot assign to a constant, so it has to be a variable, specified by the variable ID. There is an exception to this rule, though (see Conditional Statements below).
- Finally, all the parameters are given, all in two parts – type followed by ID. These can be variables (131074), constants (131075), or other data types (see the language IDs in page 2013), followed by a language ID or variable ID.

Note that you can see the order of the parameters (indeed, the order of everything in the array) by checking the script command itself in the language file. For example, the despised 1185:

```
START %0->command %1 : arg1=%2, arg2=%3, arg3=%4, arg4=%5, arg5=%6
```

Where present, the RefObj is always %0 (with one exception) – the first entry. RetVars are then usually %1 (the second entry), unless there is no RefObj, in which case they're usually %0 instead. Then the other parameters come in the order they're given in the language file.

This general structure holds for the vast majority of commands, but there are some that don't quite fit this structure: expressions (ID 104) and calls (ID 102) in particular.

Calls aren't too complex; their language entry looks like this:

```
%1 %2->call script %0 :
```

The %0 parameter is actually the script name you're calling, and %1 is then the RetVar and %2 the RefObj. This is, I believe, the only time that the RetVar comes before the RefObj. Then there are up to 5 (?) extra parameters possible on the end, depending on the script you're calling. Exscriptor attempts to load up the other script to check, but the ISE has no such problems. These are then displayed in the ISE according to the script

arguments of the called script, but in Exscriptor you can give the parameters any name you like (the names aren't checked, only the number). This means in practice that call commands can be of varying length in the codearray.

5.4.1 Expressions

Expressions are, by far, the most complex and difficult script commands. They're just so *bizarre*. They're also the most common type of script command, which just makes them all the more troublesome. The language entry of the expression command is simply:

```
%0%1
```

Helpful, isn't it? Basically, %0 is the return and %1 is the "expression", which in practice can be almost anything. The general structure of an expression is therefore:

1. 104 – Expression ID
2. Return value or conditional command; this is not in two parts, as there is no need for a datatype. If the value is positive, then it's a variable ID; if it's negative, it's a jump (explained below).
3. 1st expression section, in Reverse Polish (postfix) form. First entry is the number of operands and operators. Each following item has two parts, a datatype and then an ID. Adding two variables would be something like: 3, 131074, 1, 131074, 2, 15, 11 (where 15 is an operator and 11 is +).
4. 2nd expression section, in infix form. First entry is again the number of operands and operators, followed by the single entries, one for each operator and operand, in infix form. Operands are negative, operators are positive (or 0, for ==). Using the same example, we might have something like 3, -1, 11, -2. The operands are, I think, like indexes into the postfix part:–1 is the first entry in the postfix array, the –2 is the second, and so on.

As a full example, let's try the simple expression `$z = $x + $y`.

Variable IDs:
1 $x
2 $y
3 $z

```
<sval type="array" size="13">
   <sval type="int" val="104" />     // Expression
   <sval type="int" val="3" />       // $z (return var)
   <sval type="int" val="3" />       // Number of parts
   <sval type="int" val="131074" />  // Variable
   <sval type="int" val="1" />       // Var $x
   <sval type="int" val="131074" />  // Variable
   <sval type="int" val="2" />       // Var $y
   <sval type="int" val="15" />      // Operator
   <sval type="int" val="11" />      // +
```

```
   <sval type="int" val="3" />       // Number of parts
   <sval type="int" val="-1" />      // First operand
   <sval type="int" val="11" />      // +
   <sval type="int" val="-2" />      // Second operand
</sval>
```

I should also mention unary operators. Operators in expressions are listed in the language file, page 2001. However, three of the operators – the bit negation (~), negative (-), and logical negation signs (!) – are unary operators, meaning they apply only to one operand. Apparently, their IDs are altered slightly to make this clear, since the ISE adds 65536 to them. So the IDs of these three signs, at least as they appear internally, are:

65554	~
65555	-
65556	!

Parsing expressions is tricky enough; emitting the correct code is even harder.

But there is one thing worse than an expression – a conditional expression.

5.4.2 Conditional Commands

As stated earlier, the return variable is always a single entry in the array – the variable ID. However, sometimes this will be a negative number, and in that case it is not a variable – it's a conditional jump. This can apply to any command that returns something, but it's seen most frequently in expressions.

The negative number is a composite 8 byte number. The first four bytes are always 255 (which is why the number is negative). The fifth byte is a value indicating the type of jump – either jump-if-true or jump-if-false. So for example:

```
if $x == [TRUE]
```

is a "jump-if-false" command, since if $x is not [TRUE], we jump to the next else or end. An `if not` is a "jump-if-true" command. The values are 160 for "jump-if-false" and 224 for "jump-if-true" (not commands). The *eighth* byte – I'm leaving 6 and 7 for a moment – represents the actual command. This is an index into the language file, page 2007; for example, an "if" is 3, a "while" is 9. There is also the START command but this is a little different as it doesn't jump anywhere.

The sixth and seventh bytes are the difficult ones. These determine the line we're supposed to jump to. Note that these line numbers are a bit misleading: they're like indexes into the codearray, *not* line numbers in the original script. This is because the codearray line numbers do not include "hidden" commands like "end" and "else" etc. So in effect we jump to the first "real" command after the end, like so:

```
1     $sum = 0
2     $x = 10
3     while $x
4           $sum = $sum + $x
5           dec $x =
6     end
7     write to player logbook $sum
```

Line 6 is missing in the codearray, so the while actually jumps (if false) to line 7 (though it will say it's jumping to line 6).

Except it's a little more complex than that, because there are also *hidden jumps*, command ID 112. These are just as the name implies: invisible goto commands hidden in the codearray but not visible to the scripter. If we looked at the above while loop in the code array, we'd really see this:

```
0     $sum = 0
1     $x = 10
2     while $x
3           $sum = $sum + $x
4           dec $x =
5     hidden goto 2
6     write to player logbook $sum
```

So when the code says it's jumping to line 6, it actually does mean line 6 in this case. The situation is similar in an if-else statement:

```
0     if $x
1           * do something
2           hidden goto 5
3     else
4           * do something else
5     end                                  Not present in codearray
5     write to player logbook $x
```

Hidden jumps need to be inserted automatically by the compiler, but only once all of the commands have been parsed (otherwise, there's no way of knowing where to jump to). In other words, it takes two passes to generate the jump information.

5.4.3 Datatypes

A note on using datatype commands (e.g. "`= is dataype [$value] = {DATATYP_SHIP}`"): these should have IDs as listed in the language XML files from 0 to 26. However, *instance* datatypes – those that represent instanced objects in the game universe like ships, sectors, stations, wings etc – have 65536 added to them. Furthermore, VAR and CONST datatypes use the values 131074 and 131075 as mentioned above.

5.5 "Hidden" commands

These make up the 9th element of the codearray. As already mentioned, "hidden" commands are commands present to make it easier for users to read the script but which are not present in the main part of the codearray (the 7th element). These include things like comments and empty lines but also things like "else", "break", "continue" and "end". The language file shows them all in page 2004:

ID	Command	Explanation
1	`* %0`	A normal text comment
2	`-`	An empty line
3	`*`	A commented command
4	`end`	The end of a while or if statement
5	`else`	Part of an if-else
6	`continue`	Continue command (returns to the previous while)
7	`break`	Break command (breaks out of the current while)
101	`%0:`	Label

and they are also present in the main script command section (2003) with the same IDs.

The 9th section of the codearray lists all of these commands. Each entry consists of a line number and then the ID from the table above; often that's it, but for comments (both types) and labels, there's also a set of parameters. So, while an "end" on line 124 would look like this:

```
<sval type="array" size="2">
<sval type="int" val="124" />
<sval type="int" val="4" />
</sval>
```

a comment on line 99 would look like this:

```
<sval type="array" size="3">
<sval type="int" val="99" />
<sval type="int" val="1" />
<sval type="string" val="This is a comment on line 99" />
</sval>
```

Note that, once again, these line numbers are not the numbers you see in the final script; these are indexes into the proper script codearray. The hidden commands are inserted there. It's actually more complex than it sounds, since as you insert commands the line numbers of subsequent commands change, so you have to make sure you insert them in the right order. This is why you will sometimes see consecutive hidden commands with the same line number – they all get inserted at the same point, one after the other.

Commented commands are the most complex. These are, essentially, entire script commands (just as in the main part of the codearray) that have been moved to the hidden command list. The first two elements are still the line number to insert to and the ID (3, for commented commands), but then the rest is essentially just the script command as it would normally appear. There are some exceptions, though; if a variable is used, it will appear as its name, not using its ID (which may change when you uncomment the line). So rather than

```
<sval type="id" val="4" />
```

you might get

```
<sval type="string" val="aJellyBean" />
```

instead. Another curious thing is that jump commands are still fully formed but apparently their destination is unused. Finally, expressions are also considerably different: the second part of the codearray for an expression (which shows the infix order) is absent, and instead the first part of the codearray (the postfix part) is given, but reordered to make it infix. If this sounds confusing, that's because it is. For example, take this expression:

```
$x = $y * 2
```

In a normal expression, the code array would be:

```
<sval type="array" size="13">
   <sval type="int" val="104" />     // Expression
   <sval type="int" val="0" />       // $x (return var)
   <sval type="int" val="3" />       // Number of parts
   <sval type="int" val="131074" />  // Variable
   <sval type="int" val="1" />       // Var $y
   <sval type="int" val="4" />       // Literal (4 = int)
   <sval type="int" val="2" />       // 2
   <sval type="int" val="15" />      // Operator
   <sval type="int" val="13" />      // *
   <sval type="int" val="3" />       // Number of parts
   <sval type="int" val="-1" />      // First operand
   <sval type="int" val="13" />      // *
   <sval type="int" val="-2" />      // Second operand
</sval>
```

However, in the commented command, it would be:

```
<sval type="array" size="11">
   <sval type="int" val="xxx" />     // Line to insert to
   <sval type="int" val="3" />       // This is a command
   <sval type="int" val="104" />     // Expression
   <sval type="string" val="x" />    // Return
   <sval type="int" val="3" />       // Number of parts
```

```
   <sval type="int" val="131074" />  // Variable
   <sval type="string" val="1" />    // Var $y
   <sval type="int" val="15" />      // Operator
   <sval type="int" val="13" />      // *
   <sval type="int" val="4" />       // Literal (4 = int)
   <sval type="int" val="2" />       // 2
</sval>
```

Notice how it's shorter? And how the expression is in infix order, not postfix? And how there's only one section now, not two? Weird, isn't it?

5.6 Differences in X2

There are actually very few differences between X2 and X3 scripts, aside from the extra commands in the newer game. The only real difference is that some constants in X2 have 65536 added to them – in particular, those constants referring to in-game objects:

1	THIS
3	PLAYERSHIP
4	HOMEBASE
5	ENVIRONMENT
6	SECTOR
11	DOCKEDAT

Apart from this, and the different script engine version, scripts should be compatible.

5.7 Differences in X3TC

X3TC introduces many new changes to the script engine, ranging from new data types (e.g. wings, passengers), many new commands (including the ability to call scripts by their names), and changes to the data files and structure (including the .PCK compression). Some of these are more easy to adapt to than others.

Another major difference (in terms of script compatibility) is that X3TC will only load X3TC scripts – to open old X3/X2 scripts in game, you will first need to load and save them in Exscriptor's X3TC mode (or alternatively, manually change the two engine IDs to 40 or 41 inside the script file).

It is important to note that some pre-existing X3 (or even X2) commands have been changed; if your script uses these, then you may need to check them in more detail (but then again, it would be a good idea anyway, as X3TC may offer better ways of doing things).

Once again, I would like to point out that V1.2.x of the Exscriptor has essentially been "stretched" to cover X3TC, and there may be gaps in its support for the new features. Caution is strongly advised.

5.7.1 Data file changes

Some of the most important changes in X3TC are in its data files. Firstly, the .PCK format has changed; in X2/X3, it was a zipped file that was then "encrypted" by XORing the file with a certain value. In X3TC, Egosoft inexplicably changed this so that .PCK files are now essentially just zip files (and can therefore be unzipped with normal tools like WinZip or 7zip etc). This necessitated changes in the file structure libraries underlying the Exscriptor.

Secondly, the language files have changed from their familiar XXYYYY format, where XX is the language and YYYY the ID, to YYYY-L0XX. Again, this meant updating the functions that read in the language files. More importantly, the contents of the main language files have been extended with many new entries. Later X3 patches introduced new data by giving it a 30 prefix, so that e.g. script commands at 2003 were extended with new commands at 302003. X3TC introduces a third section with the prefix 35, meaning script commands are spread across page IDs 2003, 302003, and 352003.

New pages were introduced too. Ship commands can be found in pages 2008, 2010, and 2011 (and often 30xxxx versions too). As well as new 35xxxx versions, an entirely new section – Wing Commands – has been added at pages 2028, 2030, and 2031 (i.e. the ship command pages + 20). To support these, new data types (e.g. DATATYP_WINGCMD) and script parameter types (e.g. `Var/Wing Command`) have been added.

Similarly, new data/parameter types have been added to support passengers (e.g. DATATYP_PASSENGER, `Var/Passenger`) and the new user menu system (such as the new `Script Reference Type` parameter type). All of these involve additional type checking in the compiler.

5.7.2 Command/codearray changes

Aside from the changes already mentioned, there are many new commands[2] and in some cases, these new commands override earlier ones (many of the "find" commands, for example, now have an "exclude array" parameter). For the most part, these required no extra handling in the compiler, but there are some baffling changes. The infamous and irritating 1185 has had its name changed so it is less ambiguous (it is now starts a "delayed" command, whatever that means), but its number of parameters remains totally inconsistent; in some official X3TC scripts it still has 5 parameters, in others only 4.

Similarly, the DATATYPes are handled inconsistently in command 125, the `is datatyp [` *`value`* `]` command. In some cases they have higher bits set (and so have values of 65536 + ID) and in other cases they do not. I have yet to figure out whatever logic (if any) lies behind this.

[2] One particular curiosity: command 1442, "`add marine to attack group on ship`", does not appear in the in-game script editor, but it does work. It's very handy for "beaming" marines directly aboard enemy ships…

Command 1384 is an extreme case. It is a new command, "`set wing command`", and according to the data files, it has 4 parameters; in the code array, several official scripts give it *six* parameters. 1496 is another example, except this is totally incomprehensible as it has just one extra codearray entry; this is not enough for an extra parameter and it seems to have a value (10) that does not seem to correspond to anything.

Finally, there are a number of built-in/official scripts that call non-existent scripts or scripts with the wrong number of parameters (the "find station" commands with exclude arrays are particularly prone to this).

Since the logic behind many of these "features" is unclear (and these are just the ones I've found), Exscriptor may get it wrong while compiling them. In such cases, loading the script into the game and saving it there may either fix the problem or at least highlight it.

5.8 Differences in X3AP

Whereas X3TC made a lot of changes over X3R, X3AP made less drastic changes, mostly restricted to adding lots of new script editor commands and a few constants etc. X3AP however also has a different directory structure, since it requires X3TC's data files; this necessitated some fiddling with the data file loading. X3AP's new entries are typically prefixed by 38 in the language files, thus:

- xxxx (e.g. 2003) are the X2 entries
- 30xxxx (e.g 302003) are the X3 entries
- 35xxxx (e.g. 352003) are the X3TC entries
- 38xxxx (e.g. 382003) are the new X3AP entries

The game also adds new "Fleet Commands" in the 1300 range, together with three invisible commands (1619-1621) used in the Graph.* scripts which are not in the language files:

<RetVar> = get player ship usage time: <Var/Ship Type>
<RetVar> = get player object killed count: <Var/Ship Type/Station Type>
<RetVar/IF> = <RefObj>->get complex hangar

The script engine also seems to have been updated to version 50.

Big thanks to Shimrod for beating me to it and updating the X3AP command parameters files!

6 Glossary

Codearray	The important part of a script file. This is where X3 loads its data from. It has ten parts, the most important of which is the seventh part which contains all the script commands themselves
Conditional command	A command that performs some kind of comparison or makes some kind of decision, e.g. an "if" command. Many commands can be made conditional by changing the return variable to "if", "while", or "skip if" etc.
Expression	An expression is a special type of script command that performs some calculation and then either assigns or compares the result. For example, "`$x = $y + 10`" is an expression, as is "`if $x > 10`"
Hidden commands	These are commands not present in the main codearray of a script and include comments, empty lines, and flow commands like "end", "break", and "else".
ISE	Ingame Script Editor, i.e. the script editor in X3.
Language Files	These are the .xml or .pck files in the "\t" directory of X3. They all have numeric 6 digit names, the first two digits of which represent the language (e.g. 44 = English). They contain virtually all the text used in the game.
Maintype/ Subtype	Every ship, ware and station has a maintype and a subtype. The maintype tells you what kind of object it is (e.g. maintype 7 is for ships) and the subtype identifies that ware within the maintype (it is also the index into the Type file).
Object List	The box in the right side of the interface where various objects and commands are listed.
RefObj	The object you're performing a script command on, e.g. in "`[THIS]->get sector`", [THIS] is the RefObj. You can spot a RefObj as it is always on the left of the -> arrow.
RetVar	A return variable (usually), i.e. the variable you store the result of a script command in. Can also be a conditional command like "if", "while" etc.
Script	An X3 script file, usually in .xml format.
Script Editor	Both any editor capable of editing X3 scripts (including the ingame editor) and the black box on the left of the interface where the code of the script is edited.
Script Metadata	Data about the script – its name, its version, the engine version it was designed for, what command it represents, its description.
Status Box	The box in the lower left corner of the Exscriptor interface where messages and results are shown.
Syntax	Syntax is the structure or grammar of a language (scripts, in this case).
Type files	These are the .txt files that (when unpacked) can be found in the "\type" directory of X3. They usually have names beginning with T, e.g. "TShips.txt", and contain data about objects, ships, wares, and stations etc.