



# The X<sup>2</sup> MSCI Programmer's Handbook

by  
Kurt Fitzner

Revision 1.1

Copyright ©2004, Kurt Fitzner

License to make personal copies and printouts is granted.

All other rights reserved.

You may not distribute this work to others in any form without the author's written consent.



# THE FURTHER

## Table of Contents

1. About This Manual.....	5
1.1 Acknowledgements.....	5
1.2 Example Code.....	5
1.3 Terminology.....	5
1.4 Parameters and Syntax.....	6
1.5 Companion Files.....	6
2. Introduction.....	7
2.1 Overview.....	7
2.2 Enable Scripting.....	7
2.3 Scripting Menu.....	7
Script Editor.....	7
Reinit Script Caches.....	8
Script Debugging.....	8
Clear Debug Messages.....	8
Script Debugger Menu.....	8
Global Script Tasks.....	8
3. Viewing and Editing Scripts.....	10
3.1 Viewing Scripts.....	10
Viewing Scripts You Write.....	10
Viewing the Included Scripts.....	10
3.2 Script Structure.....	10
Name.....	11
Version.....	11
Description.....	11
Arguments.....	12
Source Text.....	12
3.3 Script Line Structure.....	12
3.4 First Script.....	13
4. Scripting Fundamentals.....	17
4.1 Variables.....	17
4.2 Arrays.....	19
4.3 Conditional Instructions.....	20
IF Blocks.....	20
Skip IF.....	20
Building a Conditional Instruction.....	21
Null – the Special Condition.....	21
4.4 Loops.....	21
4.5 Flow Control.....	22
Continue & Break.....	22
Goto.....	23
4.6 Look to Examples.....	23
5. Script Interface.....	25
5.1 Debugging Scripts.....	25
Built-In Debugger.....	25
Logging.....	26
5.2 Scripts as Commands.....	26
Command Scripts.....	26
XML Language File.....	27
Setup Script.....	29



# T H I S I S B O O K I S A T

## The X<sup>2</sup> MSCI Programmer's Handbook

6.Reference.....	31
6.1 General (Flow Control).....	31
6.2 General (Script Calls).....	31
6.3 General (Arrays).....	32
6.4 General.....	33
6.5 Audio Commands.....	42
6.6 Logbook Commands.....	43
6.7 Fly Commands.....	44
6.8 Trade Commands (for Ships).....	56
6.9 Trade Commands (for Stations).....	59
6.10 Trade Commands (for Ships and Stations).....	60
6.11 Trade Commands.....	61
6.12 General Object Commands.....	63
6.13 Universe and Sector Commands.....	76
7.Advanced Topics.....	78
7.1 Processes and Tasks.....	78
7.2 Concurrency, Interrupts, and Atomic Operations.....	80
Concurrency.....	80
Interrupts.....	81
Signals.....	81
Atomic Operations.....	82
7.3 Artificial Life (AL) Engine Plugins.....	84
Registration Script.....	84
Event Handler Script.....	84
Timer Handler Script.....	87
7.4 Automatic Command Restarting.....	87
Detecting Script Changes.....	88
Performing the Restart.....	89
Annex A. Data Charts.....	90
A.1 Plot states.....	90
A.2 Audio Samples Catalogue.....	91
A.3 Speech Samples Catalogue.....	96
Page 7 – Sector names.....	96
Pages 9 & 12 – Latin and Greek Letters.....	97
Page 13 – Miscellaneous Phrases.....	98
Page 17 – Object, Ship, and Factory Names and Descriptions.....	103
Stations.....	103
Wares & Upgrades.....	104
Ships.....	105
A.4 Default Start Actions.....	106
A.5 Object Hierarchy.....	107
A.6 Search Flags.....	108
A.7 Asteroid Types.....	109
A.8 Nebula Types.....	110
A.9 Sun Subtypes.....	111
A.10 Planet Subtypes.....	113
A.11 "Special" Object types.....	115
Special Object Descriptions.....	115
Special Object Images.....	117
Index.....	121



# 1. ABOUT THIS MANUAL

## 1.1 ACKNOWLEDGEMENTS

The author wishes to thank the following people who have contributed to the making of this manual:

- My wife, Linda, for being a computer widow for a month over this book<sup>1</sup>, for suggestions, encouragement, support, and proof-reading.
- [ticaki](#), for many helpful suggestions.
- [esd](#), for kindly allowing me to base the [Audio Samples Catalogue](#) in Annex A.2 on his excellent [wave table guide](#). It was heavily edited, but his work gave a great head start.
- [BurnIt!](#), for pointing out the source of the data used to create the table in [Annex A.1](#).
- [IvanT](#) for a kick in the pants to get back into the X community, and for the help in becoming a beta tester. The only thing else I could ask for more is an introduction to that girl in his avatar.
- The entire Egosoft team for making an incredible game. A game with a scripting component such as this one has is unprecedented.
- The X community at large, for encouragement, advice, and for being the best gaming community in the world. Here's to you.

## 1.2 EXAMPLE CODE

Any time example code is given, it will be shown in a Courier type face, with colouring similar to what is displayed in X<sup>2</sup>'s script editor. For example:

```
TestLockupFix:
```

```
001 while [TRUE]
002 @ wait 10 ms
003 end
```

## 1.3 TERMINOLOGY

The word “**command**” can have many connotations in X<sup>2</sup> – especially in scripting. In this manual, command will be used to indicate a script or game function that a player causes to run through a menu. Such as to “command” a ship to jump to the next sector. The actual script file that is attached to a command will be referred to as a “**command script**”.

The word “**instruction**” will be used to indicate a scripting statement in a script. For example, two important scripting instructions are the [wait](#), and [call script](#) instructions.

The word “**statement**” will also sometimes be used. This will generally indicate a scripting instruction as it appears in an actual script. For example: the statement at line 2 in the above script is very important to keep the script from causing the game to freeze.

The word “**script**” can also have different connotations. Some use it to refer to a single script file. Some use it to refer to all the scripts that make up a given command – that is, the actual command script itself plus all the support scripts. This manual will use the word “script” to mean a single script file. When referring to a combination of scripts that are bundled together to perform a common task, the word “**plugin**” will be used.

<sup>1</sup> To be honest, this should read “for being *more* of a computer widow for a month over this book”



T

H

M

T

H

R

M

A

T

## ***1.4 PARAMETERS AND SYNTAX***

Text in square brackets indicate syntax or a parameter that is optional (a bold typeface is used to denote when a square brackets actually occur in an instruction). For example:

**[skip] if [not]|while [not]|<retvar> = <arrayvar>[<element>]**

In the above syntax description of one of the array instructions, **[skip]** indicates that 'skip' is optional syntax. The bold-face brackets around <element> indicate that those square brackets are actual characters that occur in the instruction.

Angle brackets indicate descriptive words describing a parameter. The text between the brackets is a description of what to put, not verbatim text that you would enter. It indicates text that you would replace with something of your own creation. In the above example, <arrayvar> indicates a parameter where you would specify an array variable.

The vertical bar character, “|”, is used to indicate a list of syntax possibilities that you would choose from. Continuing from the above example, the section “**[skip] if [not]|while [not]|<retvar>**” indicates that you would choose [skip] if [not], or while [not], or <retvar>. Think of it as an “or” character in a syntax description.

## ***1.5 COMPANION FILES***

There should be a set of companion files available from the same location you obtained this book. These companion files contain a set of high-resolution images to go along with the thumbnail pictures in Annex A.7 through A.11.



T  
H  
E  
T  
H  
I  
S  
I  
S  
B  
O  
O  
K

## 2. INTRODUCTION

### 2.1 OVERVIEW

MSCI – Manual Ship Computer Interface. That little four-letter acronym changed the X gaming experience forever. X² is perhaps the most extensible space simulation ever written. Available to every player is a method of writing programs that can affect almost every aspect of the game. These programs, called scripts, can perform trading runs, fire your turrets, transfer money to or from your stations, protect your transports... the sky isn't even your limit. Much of the behaviour of ships in the game is governed by scripts that were written by the developers. When you order a transport to “Buy ware at best price”, it's a script that is controlling it. In fact, all the commands that are in the Flight, Trade, Combat, etc menus run scripts.

The best thing about scripts is that they can be created, edited, and debugged right from within the game. The game itself is a whole development environment for creating more of the game.

### 2.2 ENABLE SCRIPTING

To enable scripting, exit out of any menu and type “Thereshallbewings”. Type it fairly slowly and make sure the 'T' is capitalized. After this is done you should hear a beep.

It should be noted that once scripting is enabled, the current game you are playing is tagged as a modified game. A small label will appear in the top left corner of the game screen marking it as modified. Because a script can do almost anything, a player with scripting enabled could simply script up a billion credits at the very beginning of the game. This label is intended to mark screen shots taken on the game so that it is clear to other players that the player is working on a non-standard game.

If you intend to do a lot of scripting, it might be wise to create a new game especially for development and not use your main game saves.. This is a good idea not only because of the modified tag, but also because it is quite possible to damage a game in ways that aren't readily apparent. Don't mix script development and game playing.

### 2.3 SCRIPTING MENU

To bring up the scripting menu once scripting has been enabled, just press **SHIFT+C** to call up your ship's command menu, then hit **S**. The first time this is done you will see a small help file with information on the script editor. Once you hit **ENTER** to leave that screen, you will see six items in the scripting menu that will appear:

#### Script Editor

This menu entry takes you to the actual script editor itself. The very top entry allows you to create a new script. All the currently editable scripts are listed and can be opened. Just use the arrow keys to highlight the script you want and hit **ENTER**. To copy an existing script, highlight it and press **C**, and to delete a script, highlight it and press **DEL**.

When inside a script, editing it, you can use the **↑** and **↓** arrow keys to move to different lines of code (the **PG↑** and **PG↓** keys can also be used to quickly move up and down through a script), the **←** and **→** arrow keys move left and right within a line of code, and the **INS** and **DEL** keys to insert or delete lines.



## Reinit Script Caches

When a game loads or starts, every script that is attached to a command menu item is stored in a script cache. This allows for faster command execution. However, it also means that a command will continue to use an “old” version of a script even after the script has been changed. The “Reinit Script Caches” menu entry allows you to force X<sup>2</sup> to reload all the scripts it has in its cache.


This does not affect any ship which is currently running a command at the time you reinitialize the cache. It only affects ships you issue that command to after the cache is reinitialized.

For example: transports *Dromedary* and *Bactrian* are both running the imaginary command “Buy ware at worst price”. This command is attached to the script “trade.buywareworst”. You discover that the script has a bug – *Bactrian* bought a ware at a trade dock when it could have paid more for it at the nearby Frosted Sugar Cahoona Cereal factory. You open up the trade.buywareworst script and fix the bug. You now issue the “Buy ware at worst price” command to *Bactrian* again. You soon notice, however, that the change you made to the script hasn't affected *Bactrian*, even though you gave it the command again. This is because when *Bactrian* was issued the command, it read the script from the script cache.

You now select “Reinit script caches” from the script menu so that you can test out your changes. At this point, both *Dromedary* and *Bactrian* are still running the old version of the script. The “Reinit script caches” command doesn't force all the ships that are running a script to reload that script. It only reinitializes the cache where scripts are read from when a new command is issued.

Now you issue the “Buy ware at worst price” command to *Bactrian* yet again. Finally, *Bactrian* will be running the changed version of the script. Poor *Dromedary*, on the other hand, will still be hampered by that old version of the script until you reissue the command to it as well.

## Script Debugging

There are two debugging modes, “logging” and “trace”. Hitting  once turns on logging, hitting it twice turns on script tracing.

## Clear Debug Messages

Removes all log or trace entries from the debugging log for that ship.



## Script Debugger Menu

Displays the script log or trace once enabled.

Script debugging is covered fully in section 5.1 on page 25.

## Global Script Tasks

A running script can be associated with an object (a ship or a base) – in which case it is said that the object is running the script. A script can also have no association – it can be unattached to any ship or base. In this case it is called a “global” script.

It is fairly easy to see what scripts are running on a ship. Just target the ship and hit  to pull up the ship's information screen. Once scripting is enabled, pressing  on the info screen will reveal all the running scripts for that ship. Terminating a script that is running on a ship is usually as easy as pulling up the ship's command menu and ordering it to do “None”.

This can't be done for global scripts since, by definition, they aren't attached to a ship. The





T

H

M

T

H

R

M

A

T

## The X<sup>2</sup> MSCI Programmer's Handbook

“Global Script Tasks” entry in the scripting menu allows you to see all the running scripts that aren't attached to any object. If you have a global script that you want to terminate for any reason, highlight it with the arrow keys and press .

If you are terminating a script, do be careful that you are terminating the correct one. There are often many new global script tasks being starting and stopping. It is quite possible to hit the key only to find that the entry you had highlighted changed a fraction of a second before you did it – so fast that you didn't notice. Make a note of the last few digits of the PID (Process ID) number before you press . That way you will be able to double-check that you are terminating the correct process.

See section 7.1 on page 78 for a detailed discussion of processes.



## 3. VIEWING AND EDITING SCRIPTS

### 3.1 VIEWING SCRIPTS

Before going into the hows of scripting, it's useful to talk about how a script can be viewed outside of X². Scripts are all located in the <X²>/scripts directory (where <X²> is the directory where you installed X²). What you need to do in order to view a script depends on whether or not you wrote it:

#### Viewing Scripts You Write

The script editor saves all scripts as XML files. Egosoft has included a nice XML style sheet in that directory, so all you need to do in order to view a script is double-click it<sup>2</sup>.

#### Viewing the Included Scripts

The scripts that Egosoft wrote and included with the game are distributed in a compressed form and have the extension .pck (meaning packed). To decompress them you need to download either the official [X² modder kit](#), or the tool [X² Modder](#) by OlisJ. The latter is a graphical tool that can compress or decompress a whole directory at once.

The official kit has a program, x2tool.exe, that (among other things) can compress and decompress the built-in scripts. To decompress a script, use the syntax:

```
x2tool -unpack ascript.pck ascript.xml
```

This will unpack the script and turn it into a standard XML file that can be viewed with almost any browser.

### 3.2 SCRIPT STRUCTURE

Every script follows the same general pattern. Viewing a script in a browser shows a script's structure very clearly. The following is one of the built-in scripts as it might look in a browser:

#### Script !ship.command.follow.pl

Version: 1  
for Script Engine Version: 20

#### Description:

Ship Player COMMAND FOLLOW

#### Arguments:

- 1: target, Var/Ship, 'the target'

#### Source Text

```
001 • = [THIS] -> call script '!ship.cmd.follow.std' : the target=$target  
follow distance=null  
002 return null
```

signed

<sup>2</sup> A minor bug in the stylesheet x2script.xsl in versions of X² prior to 1.3 causes an error if you try and view a script with a Mozilla-based browser. It is highly recommended that you upgrade to the latest version of X² anyways.



The above example is a script that is attached to the navigation command “Follow ship” for player-owned ships. It is very simple – it just passes control off to a more generalized script to handle the command. You will find that many of the built-in command scripts are like this. Very short, simple scripts that take the specific command and pass it off to a larger, more generalized script to handle.

Though small, the above example illustrates the structure that is common to all scripts:

### Name

The first part of writing a script is choosing its name. This actually isn't as trivial a decision as it may sound – there are important considerations to keep in mind when choosing a name.

Any script which has a name beginning with an exclamation point is considered to be non-editable. It won't be listed in the script editor.

Script names beginning with “init.”, “setup.”, “al.plugin.”, and “galaxy.” all have special meanings as well. They are all run automatically by the game at certain times.

- **Init Scripts:** Scripts with names beginning with “init.” or “!init.” are run automatically every time a new game is started or loaded. They *can* be used to configure menu commands, and at one time were used extensively to do just that. However, they are run very early in the process of starting or loading a game, at a time when many aspects of the galaxy aren't initialized yet. Before the galaxy's stations and ships have been created. Because of this, special precautions have to be taken in an init script if it is to do something like, for example, adding a new upgrade to an equipment dock. Because of these limitations, the use of init scripts has largely been discontinued in favour of:
- **Setup Scripts:** Scripts with names beginning with “setup.” or “!setup” are also run automatically every time a new game is started or loaded. These scripts, however, are run later on in the game initialization sequence. All the galaxy's stations and ships will have been created before the setup scripts are run. This makes them much safer and easier to use as a general configuration system than init scripts.
- **AL Plugin Scripts:** An AL (Artificial Life) plugin is a script designed to be a background process that enhances the realism of the game universe. Passenger ships that fly from station to station (even player-owned stations) and warships that fly patrol routes protecting several sectors. Any script with a name beginning with “al.plugin.” or “!al.plugin.” is a script that is intended to configure one of these AL plugins. The concept of AL plugins is covered in detail in section 7.3 beginning on page 84.
- **Galaxy Ship Init Scripts:** When a custom galaxy map is used, a special script is needed to create the player's initial ship. The game engine will look for a script with a name beginning with “galaxy.<mapname>.” or “!galaxy.<mapname>” (where <mapname> is, of course, the name of the galaxy map).

### Version

Every script has a version number associated with it. This can help both you, the script developer, and the end user keep track of when a script is updated. It can also help you to give your scripts the ability to restart themselves (see section 7.4 beginning on page 87 for a detailed discussion of command restarting).

### Description

Some text describing the purpose of this script. Don't leave this blank – even a very short description can be quite helpful.



## Arguments

Your scripts aren't getting into fights – an argument is simply a parameter. Information that a script is given when it is run or called from another script. If your script is meant to fly a ship to a sector, then an argument for the script would probably be which sector.


There are three pieces of information which comprise each argument:

1. Name – the name of an argument becomes a variable in your script.
2. Type – Every variable and argument has a type. This describes what kind of information is being stored or passed.
3. Description – akin to the name, this is a short bit of text that describes the purpose of the argument. This is for other people who will be using your script. In the example shown at the [beginning of the section](#), the *call script* instruction is running another script. The text “the target” and “follow distance” are the descriptions of the arguments used by the script that is being called on that line.

## Source Text

This is the actual body of the script – the lines of code that make it up. Generally, this will be by far the largest part of any script.

### 3.3 SCRIPT LINE STRUCTURE

The script editor allows a line of script to be put together like blocks of lego. Every single script instruction is available through a menu – there is no need to type in the commands. Simply hit enter on the “<new line>” label at the bottom of the script, or on any existing line. This will bring up the menu that allows you to select script instructions<sup>3</sup>. Navigate to the section containing the script instruction you want, and hit  to “paste” that instruction into your script.

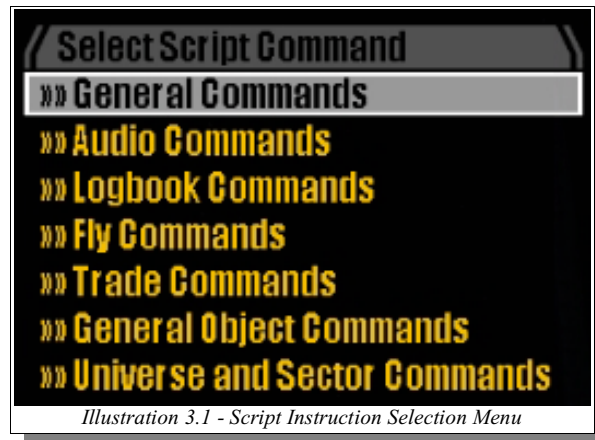






Illustration 3.1 - Script Instruction Selection Menu

The [reference section](#) contains a list of all the available script instructions in the order they appear in the menu, along with an explanation of each one.

Once you hit , the instruction (command) you have selected will appear in your script. Usually the script editor will then automatically switch to a menu to allow you to select the first parameter to that instruction. What the first parameter is will depend on the instruction. Most of the time it is to select the object that the instruction will be working on. Sometimes it's not entirely clear what is being asked. Simply hitting  at this point will drop you out of this second menu back into the script editor. At this point, the script instruction you picked will be the currently selected line. The line will be highlighted in yellow. If the instruction takes any parameters (most do), the one currently will be highlighted in red. The  and  arrow keys will highlight the previous or next parameter on the current line.

<sup>3</sup> The script editor refers to them as “Script Commands”. This manual will refer to them as script “instructions”. See the section on terminology in the introduction for an explanation of why.



### 3.4 FIRST SCRIPT

We will now step through creating a very simple new script from scratch.

1. Enable scripting as explained in section 2.2 on page 7.
2. Bring up the scripting menu as described in section 2.3 on page 7.
3. Press on the menu entry, "Script Editor"
4. The very top entry on the screen that follows will be <New Script>. Press with that selected. You will be asked for a script name. For this example, use "a.sample.first". At this point, you should be looking at a screen like this:

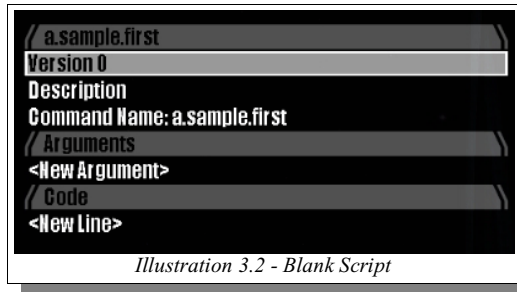


Illustration 3.2 - Blank Script

5. The top line (Version 0) is highlighted – meaning if you press , you will be able to edit it. You can use the and arrow keys to change which line is highlighted. Change the version to "1", and the Description to "Hello World" (yes, we are creating the archtypical first program). You will have to backspace over the word "Description" before you can do this. We won't be changing the command name.
6. We will add one argument to this script. Press over <New Argument>. You will first be asked for a name. Call it "Message". You will next be asked for a type. Since this argument is going to be a message, select "Var/String" from the list of types (this will be close to the bottom of the list) as shown in illustration 3.3. Lastly you will be asked for a description. The name of the argument is pretty descriptive by itself, so just put in "Message" again.
7. In case you haven't figured it out yet, our sample script will be displaying a message to the player's message log. The first line of our script will create the message. Use the arrow key to highlight <New Line>. This will be highlighted in yellow – a little different from the Version, Description, and Command Name lines. Press now and you will get the menu that allows you to select script instructions as shown in illustration 3.1. The instruction we want is the first instruction under "General Commands" – it is labelled "<RetVar/IF><Expression>". This will more than likely be the most common scripting instruction you will use. It is this instruction that allows you to set variables, make expressions, and create general purpose loops. Highlight that instruction and press . This "plugs" that instruction into your script. It will now display a menu where you can choose a variable, create a new variable, or select a type of *if*, *skip*, or *while*. This is where you select what

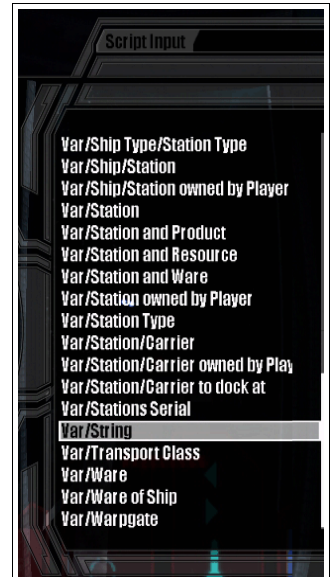


Illustration 3.3 - Step 6, "Select Var/String from the list of types..."



# TUTORIAL

type of instruction this expression will be. In our case, we want to create a new variable. Move down the list until **<Variable>** is highlighted and press **ENTER**. You will now be prompted for a variable name. Use “DisplayText” as the name and press **ENTER** again. If you have done this correctly, the screen displaying your script will now look like this:

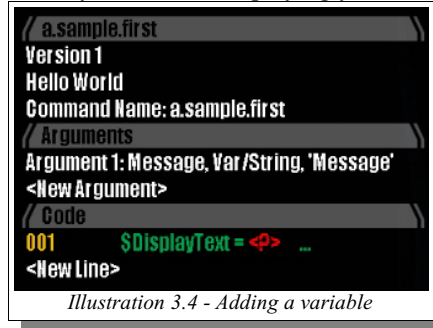


Illustration 3.4 - Adding a variable

8. We now have the beginnings of an expression that will assign a value to the variable \$DisplayText (all variables begin with a '\$'). \$DisplayText is now being set to *something*. We just need to tell it what. The **<?>** is highlighted red. Pressing **ENTER** now will allow you to select the what we will set it to. Do this, and a menu will appear holding all the different constants and variables that you can assign to \$DisplayText. Move down the list until **<string>** is highlighted. This means you will be entering a string (text) to store in \$DisplayText. Hit **ENTER** again and you will be prompted for the the string. Type in: “Hello World! Message from player - “ and hit **ENTER** to accept it.

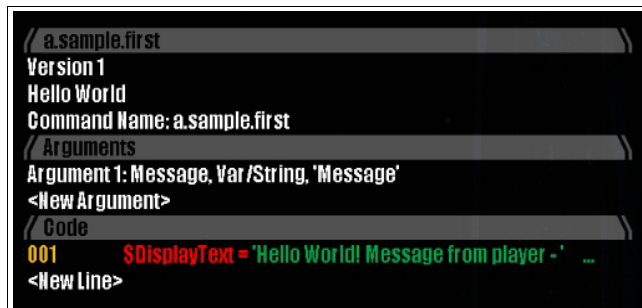


Illustration 3.5 - Adding text to the variable

9. Use the **→** arrow key until the “...” is highlighted red and press **ENTER** to continue to add on to our expression. Now since this is not the first element of the expression, the menu that appears will contain operations (often called 'operators' in programming jargon). Select the “+” operator from the list and press **ENTER** again.
10. Make sure the “...” is again highlighted red and press **ENTER** to add our last element to the expression. Now select “Message” from the list of “**Available Variables**”. Our expression is now complete.
11. Use the **↓** arrow key to once again highlight **<New Line>** and press **ENTER** to begin selecting a new instruction to add. The instruction we want this time is located in the “**Logbook Commands**” section. Pick the instruction “**write to player logbook <value>**”. As the only parameter to this instruction is **<value>**, you know that when the script editor prompts you for the first parameter for the instruction that it is this one. We will be using our newly created variable for the parameter, so select “**DisplayText**” from “**Available Variables**” and press **ENTER**. Our simple script is now almost complete. The last instruction that we need will actually be filled in automatically by the script editor. Every script must have a *return* instruction in it. It is this instruction that tells the script engine that the script has finished. If you don't explicitly enter this instruction into your script, when you save your script the script editor will place it as the last instruction in the script for you. Before we save, check to make sure the script looks like it should:





Illustration 3.6 - The completed script

12. Now we will save our script. Press **[ESC]** – you will be asked if you wish to save. Your answer to this question should probably be yes. You should now end up in the script chooser menu again. This time, your new script should be at the top of the list (the name “a.sample.first” will probably come alphabetically before any other script you have installed).

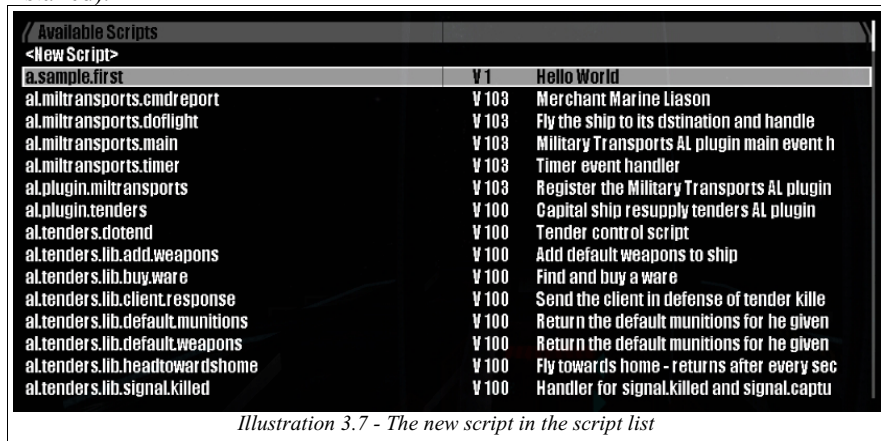


Illustration 3.7 - The new script in the script list

13. Now you will of course want to run the script. To run any script, simply highlight it and press **[R]**. The first thing that rill happen is you will be given a menu with two entries: **null**, and **Select Ship or Station**. This is where you choose what object will be running the script. We don't need any object to run it, so select **null** which will make the script engine run it as a global script (a script not attached to any object).
14. Once you select **null**, you will be presented with a screen giving you the name of the script and all its arguments. This is just a confirmation screen letting you know what you're about to do. Press **[ENTER]** here to move on (alternatively, if you've picked the wrong script, you can hit **[ESC]** to cancel).
15. If the script hadn't of had any arguments, it would now have been run. Because it has an argument, you are now presented a screen where you can enter in a value for it. You can either select **null**, which would send a null value to the script, or select **<string>**. We want to feed a value to the script, so select **<string>**. You can now enter in the string to send to the script. How about, “Hope this works!”? Once you enter the text and hit **[ENTER]**, you'll be unceremoniously dumped back in the script list screen. Our script should have written to the message log, so to see if it has worked, hit **[ESC]** until you are back to the ship's cockpit and pull up the player message log.



# T H I S I S I S T O R Y A T

## The X<sup>2</sup> MSCI Programmer's Handbook

Yours should look something not exactly unlike this:

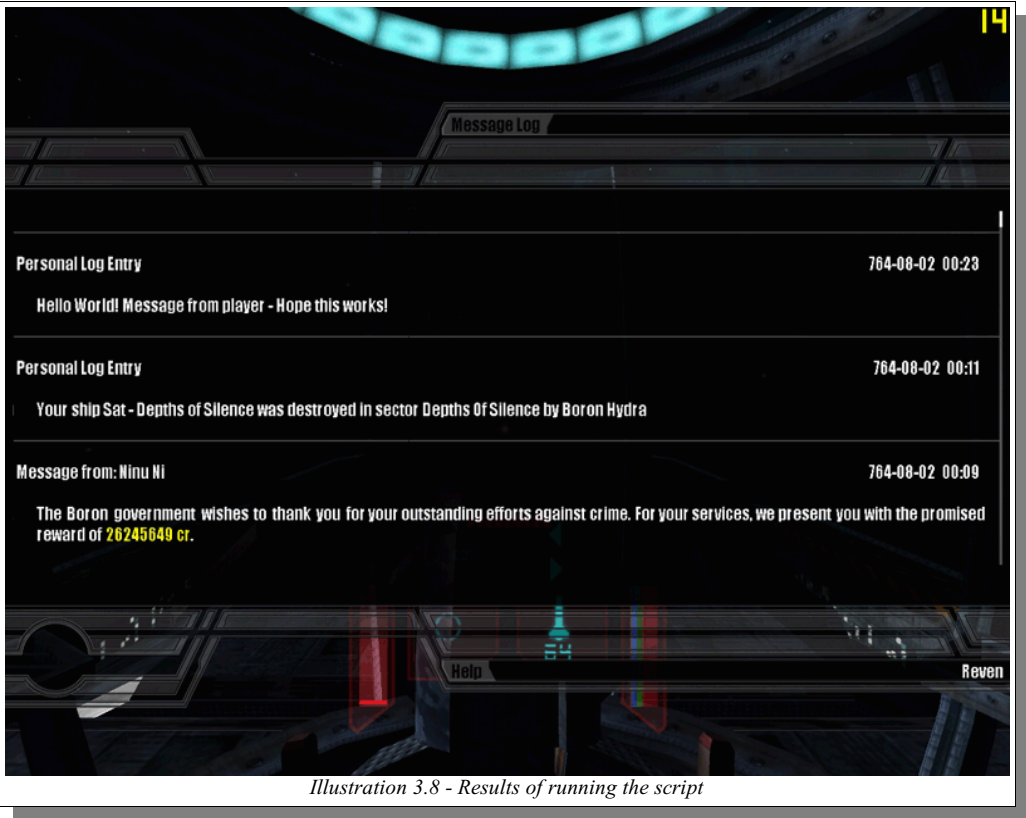


Illustration 3.8 - Results of running the script

Congratulations! You've just created your first script.





## 4. SCRIPTING FUNDAMENTALS

An X<sup>2</sup> script is, in its simplest form, a series of instructions that makes something happen within the game environment. It is beneficial to think of the different types of instructions as belonging to two groups:

1. Instructions that perform a task. This is the “meat and potatoes” of a script. Instructions that make a ship fly somewhere, shoot a gun, send a message. These instructions usually have a very direct cause and effect relationship. The instruction executes and something happens in the game universe. Line 2 in the [sample script](#) created created in section 3.4 is an example of this. It simply writes a message to player's message log. The instruction executed, the message is written – cause and effect.
2. Organizational instructions. These instructions that really do anything in the *game* universe, they do something purely inside your script. These instructions make sure that the task instructions get the right data; they make the task instructions execute at the right time and in the right order. Line 1 of the same [sample script](#) is an example of this. Nothing happens in the X<sup>2</sup> universe when it is executed. It is simply organizing the message that will be sent in line 2.

This is an arbitrary classification, but a useful one. And one that holds true for many (if not all) programming languages. Generally, all programming languages will have some instructions that “do” things – that interact with the “real world” in some way, and others that “organize” things.

There is no easy way to learn the first type of instruction. This is purely rote memorization. And the worse news is that knowing what sort of tasks can be performed in one programming or scripting language is often of very limited use in another. In this case it's made a little easier by the fact the script editor organizes all its instructions into categories. But you still won't know what sort of tasks are possible until you go over each category, and each instruction. The good news is that simply learning how to play the game gives you a good idea ahead of time of what sorts of tasks a script is capable of. You know that ships can attack other ships, that they can fly from sector to sector and land on stations. You know they can buy and sell wares, or move them from place to place. You already know the *types* of tasks can be performed. The rest is just learning what instructions perform which task. This is covered in section 6, the reference section.

The second type of instruction – the so-called “organizational” instructions – these are arguable the more important of the two types. Usually when you are writing a script, you already know the tasks you want to do. You just need to find out the task instructions that do it. Organizing your script, though, is paramount to ensuring that everything runs at it should. It is the framework of any script. This section will focus on the organizational type of instructions, and give an overview of how to frame your script. Organizational instructions tend to be very similar across different programming languages, so if you are new to programming in general, pay particular attention to this section. Once you know how to build the frame, putting in the rest will be much easier.

### 4.1 VARIABLES

Variables are simply a way of storing some piece of data temporarily inside your script. No script would get very far without this ability.

Every variable has two properties, its type and value. The type is just that – a description of the



type of data the variable is storing. You don't actually have to tell X<sup>2</sup> what type a variable is. Whenever data is stored in a variable, the scripting engine sets the type of the variable for you based on what sort of data is being stored. There are 22 different data types:

<i>Data Type Name</i>	<i>Description</i>
DATATYP_NULL	Variable has a null value. The null value has its own data type, since it (by definition) isn't of any other data type.
DATATYP_UNKNOWN	You shouldn't ever see this data type – this means the scripting engine can't establish the data type of a variable.
DATATYP_VAR	Variable – used internally in the scripting engine. Used internally in the scripting engine. This type won't be seen associated with a script variable.
DATATYP_CONST	Constant – used internally in the scripting engine. Used internally in the scripting engine. This type won't be seen associated with a script variable.
DATATYP_INT	Integer – a number from -2,147,483,648 to 2,147,483,647 (32-bit signed integer)
DATATYP_STRING	String – text.
DATATYP_SHIP	A “pointer” to a ship. Variable references a ship object.
DATATYP_STATION	A “pointer” to a station. Variable references a station object.
DATATYP_SECTOR	A “pointer” to a sector. Variable references a sector object.
DATATYP_WARE	A ware – anything that can be bought or sold
DATATYP_RACE	A race (Argon, Boron, Player, etc).
DATATYP_STATIONSERIAL	A station serial “letter” - one of the letters of the Greek alphabet.
DATATYP_OBJCLASS	Variable contains a class of an object (see section <a href="#">Annex A.5</a> on page 107)
DATATYP_TRANSPORTCLASS	Variable contains the transport class of a container. ST, XL, L, etc.
DATATYP_RELATION	Friendly, Neutral, Enemy
DATATYP_OP	Operator, as found in an expression. Used internally in the scripting engine. This type won't be seen associated with a script variable.
DATATYP_EXPR	Expression – used internally in the scripting engine. This type won't be seen associated with a script variable.
DATATYP_OBJECT	A “pointer” to an object. Variable references a space object that is not a ship, station, or sector. For example a sun, nebula, or asteroid.
DATATYP_OBJCOMMAND	Variable contains a command mnemonic.
DATATYP_FLRET	Variable contains a “Fly Command” return code. See section 6.7 on page 44.
DATATYP_DATATYP	Variable contains a data type. For example, when determining the data type of another variable using the <a href="#">get datatype</a> instruction
DATATYP_ARRAY	Variable is an array.

Table 4.1 - Data Types

Every scripting instruction that returns a result can have that result stored in a variable. You can then manipulate the result in whatever way you like, test it, or pass it along as the input to some other instruction.

The [general expression](#)<sup>4</sup> instruction is often used to manipulate variables. With this instruction, you can generate arbitrary (and quite complex) expressions. You can perform arithmetic, logical, and binary operations on numbers. You can concatenate strings. You can perform tests and even loops. It will probably be the single-most used instruction in all your scripts. It was this instruction that was used to concatenate the strings together for the “first” example script in section 3.4 on page 13.

<sup>4</sup> What is an expression? Simply put, an expression is just a series of symbols and operators that return a value. “1 + 2” is an expression, returning the value “3”. In that expression, “1” and “2” are symbols, and “+” is the operator.



This instruction is available through the script editor under “General Commands” as “<RetVar/IF><Expression>”.

Any time you wish to create a new variable from the output of any instruction, in the script editor choose <variable> when asked what to do with the output (this is usually the first “question” asked by the script editor after selecting an instruction). You will then be asked for the name of the new variable. See section 3.4 for a walk-through of this procedure.

## 4.2 ARRAYS

An array is simply a variable that contains more than one value. Generally they are related values. A group of ships owned by the player, a set of coordinates, etc. Think of an array like a column in a spreadsheet or a database. The title of the column is the name of the array. All the different values in that column would be the array's contents. Each individual value (a “cell” in our mythical spreadsheet) is referred to as an element of the array.

An array is created with the *array alloc* instruction:

```
100 $MyArray = array alloc: size=10
```

This creates an array with ten elements and stores it in the variable MyArray. In X<sup>2</sup> scripting (and most programming languages), the elements are numbered from zero to size minus one. In this case, zero to nine. An array **must** be created with the *array alloc* instruction before it can be used. The only exception to this are instructions that return arrays. Those instructions will allocate the arrays internally themselves.

To place a value in an array, use the <array>[<element>] = <value> instruction. This will let you set an element of an array to any value or variable. You can retrieve an element from an array using the <RetVar/IF><array>[<element>]. The latter will allow you to copy out an element of the array into a normal variable, or to create a conditional statement (if/skip if) or *while* loop based on it.

When a variable is holding an array, it is actually not storing the contents of the array, but a reference to those contents. For example, consider the following script:

```
100 $Array1 = array alloc: size=1
101 $Array2 = $Array1
102 $Array1[0] = 'Array1'
103 $Array2[0] = 'Array2'
104 write to player logbook $Array1
```

You might expect the result that is displayed in the player logbook to be: “**Array1**”, but this isn't the case. What will be displayed is: “**Array2**” (actually, it will be: “**array {Array2}**”), which means the variable is an array that contains one string element that is “Array2”). This is because when the above script sets Array2 to be equal to Array1 in line 101, it is not copying the array, it is only copying a *reference* to the array. It is saying, point \$Array2 to the same array that \$Array1 is pointing to.

There are specific script instructions that allow you to copy the contents of an array. See the reference section containing array instructions for information on these instructions (section 6.3 starting on page 32).

In most programming languages, all the elements of an array must be of the same type. In X<sup>2</sup>, however, each element can have its own data type. In fact, you can even use an element of one array to store a reference to another array. Remember the above, though – arrays are stored by reference, not by value.



## 4.3 *CONDITIONAL INSTRUCTIONS*

You are going to want your scripts to be able to make decisions. This ability is given to a script with conditional statements - organizational instructions that choose what parts of your script to run based on decisions you program.

There are two main types of conditional instructions in X² scripting. The “if/else/end” set of instructions, and its younger (and simpler) brother, “skip”.

### IF Blocks

The if/else/end set of instructions is usually referred to as an “if block” or an “if/else block”. It is called a block because the instructions “bracket” a group of other statements, creating a block of instructions. For example:

```
100 if $MyVariable > 500
101   ... do something
102   ... do many somethings if you want
103   ... in fact, do as many somethings as you like
104 else if $MyVariable > 50
105   ... do something else
106 else
107   ... or here do something else entirely
108 end
109 ... the script keeps going ...
```

In the above example, if the variable \$MyVariable contains a number greater than 500, then the first group of instructions (lines 101 through 103) will be executed and then the execution will continue at line 109. If \$MyVariable isn't greater than 500 but is greater than 50, then the second group of instructions (in this case just one line – 105) will be executed and then the execution will continue at line 109. Otherwise the third group of instructions (line 107) will be executed, again followed by line 109. Each group can contain as many statements as you like.

There can be many *else if* statements in an *if* block – as many as you need. A particular *else if* will only be checked if all the conditions above it fail. In the above example, let's say that \$MyVariable was equal to 750. That is larger than 500, so the condition at line 100 “passes” and the instructions in that group (lines 101-103) are executed. The number 750 is also larger than 50, but because the condition at line 100 passed, the condition at line 104 will never be checked. That's what *else if* means – if the first condition doesn't pass, then check the second one, and so on.

There can be many *else if* instructions, but only ever one *else* instruction in an *if* block. That is because an *else* by itself is the “do this if all the other conditions fail” contingency instruction.

There doesn't have to be any *else if* statements in an *if* block, or any *else* instructions. There does have to be an *if* (of course) in an *if* block, and an *end*.

### Skip IF

A *skip if* instruction is sort of like a mini *if* block. It creates a conditional where there is only one instruction, and no *else*. For example:

```
100 skip if $Target -> exists
101   return null
102 ... the script keeps going ...
```

The above example is actually a pretty common test performed in X² scripts. The statement at line 100 says skip the next line if the object pointed to by \$Target exists. This means that line 101 would only be executed if the object didn't exist. So, if the object doesn't exist, terminate



the script, otherwise, keep going.

### Building a Conditional Instruction

The **general-purpose expression** instruction is the instruction most used to make conditional statements with. This is because it is the **only** instruction that lets you create an expression. This means that it is the only instruction you can use to test a variable against a value or another variable. In the first example above, both the condition at line 100 and at line 104 were build out of a general purpose expression.

A conditional instruction doesn't have to be built out of an expression, though. It can be built out of almost any instruction that returns a value. Almost anywhere that the script editor will let you pick a variable to store the result of an instruction, it will also allow you to pick one of several conditional “prefixes”:

<i>Prefix</i>	<i>Description</i>
if	passes (the statements between this instruction and the following <i>else</i> or <i>end</i> are executed) if what follows the <i>if</i> is true.
if not	passes if what follows the <i>if not</i> is <b>not</b> true.
else if	passes if what follows the <i>else if</i> is true <b>and</b> all previous conditions in the same set failed.
else if not	passes if what follows the <i>else if not</i> is <b>not</b> true <b>and</b> all previous conditions in the same set failed.
else	passes if all previous conditions in the same set failed.
skip if	the next instruction is skipped if what follows the <i>skip if</i> is true.
skip if not	the next instruction is skipped if what follows the <i>skip if not</i> is not true.

Table 4.2 - Conditional Prefixes

### Null – the Special Condition

There is one (and only one) condition that you never need an expression to test. That condition being the test to see whether or not something is (or is not) null/zero.

For example the following conditional statements are all identical:

```
if $MyVariable != null
if not $MyVariable == null
if $MyVariable
```

All of the above tests pass if \$MyVariable is not null. In actuality, this is not so much as a special condition as it is simply the way conditionals work. They work by testing whether or not something is true. Something is considered by a conditional to be true if it isn't null/zero.

## 4.4 LOOPS

A loop is a group of instructions that is executed repeatedly until some condition occurs. Loops are built out of the **while** and **while not** prefixes available on any instruction that returns a value. In X², loops are simply a special kind of conditional instruction. The loop prefixes are in the same list as the if/skip/else prefixes. So, the above table should have two more entries:

<i>Prefix</i>	<i>Description</i>
while	Loops (the statements between this instruction and the following <i>end</i> are executed) as long as what follows the <i>while</i> is true
while not	Loops as long as what follows the <i>while not</i> is <b>not</b> true.

Table 4.3 - Looping Prefixes



It is fairly easy to make a loop that will execute a specific number of times:

```
100 $LoopCount = 100
101 while $LoopCount
102   dec $LoopCount
103   ... do whatever you want to do in the loop
104 end
```

In the above example, everything between the *while* and the *end* will execute exactly one hundred times; line 101 causes it to repeat as long as LoopCount isn't zero.

It is also fairly easy to make a loop that will execute until some external condition occurs:

```
101 while $Target -> exists
102   ... do whatever you want to do in the loop
103 end
```

In that example, everything between the *while* and the *end* will execute as long as the object pointed to by \$Target exists.

The test for whether the instructions inside a *while* loop should be executed is performed at the very beginning of the loop. If that test fails the first time, the contents of the while loop are never executed at all. In the above example, if \$Target didn't exist by the time the script reached line 101, then line 102 would never have been executed at all.

Some programming languages have a type of loop (usually called a 'for' loop) where the test for whether the loop should repeat is performed at the end. This would mean that the instructions inside a loop of this type would always execute at least once. X<sup>2</sup> has no equivalent to this type of loop.

## 4.5 FLOW CONTROL

Flow control instructions are just as the name suggests – instructions that control the flow of script execution. A conditional and a loop could be considered to be forms of flow control. They control what parts of a script can execute, or how many times a part of a script should execute.

There are also explicit flow control instructions – instructions that change the flow of script execution unconditionally. These are the *continue*, *break*, and *goto* instructions.

### Continue & Break

The *continue* and *break* instructions both work on a loop. They are opposites of a sort. A *continue* forces the current loop to skip all the rest of the instructions between it and the loop's *end*. It forces the loop to continue from the beginning. The contents of the loop aren't necessarily executed – that still depends on whether the while loop's test passes or not.

For example, consider this scenario. You are creating a script that simulates a special weapon. This weapon works on all ships within 5km of your ship. Paranid ships are, however, immune to this weapon's effects. The code might look something like this:

```
001 $Targets = find ship: sector=[SECTOR] class or type=Ship race=null flags=
    [Find.Multiple] refobj=[PLAYERSHIP] maxdist=5000 maxnum=1000
    refpos=null
002 $Target.Index = size of array Targets
003 while $Target.Index
004   dec $Target.Index
005   $Target.Owner = $Target -> get owner race
006   skip if $Target.Owner != Paranid
007   continue
009   ... do whatever your weapon does here
050 end
```





The above example iterates through all ships within 5km of the current player ship. Any time the loop comes across a Paranid ship, the *continue* at line 007 is executed. This causes the rest of the instructions inside the loop to be skipped over. The loop continues on with the next ship in the list.

A *break* instruction is the opposite. It doesn't force a loop to continue on, it is an “early exit” from a loop. It causes the current loop to stop looping, and for execution to resume after its *end*.

What is meant by “current loop” for both *continue* and *break* is the inner most loop. For example, it's quite possible to have nested loops – a loop inside a loop:

```
100  while [TRUE]
101    ... pick a target
102    while $Target -> exists
103      ... shoot at the target
104      skip if $Target -> is enemy
105      break;
106    end
107 @ wait 100 ms
107  end
```

In the above example, there are two loops. One loop, the “outer” one, is a permanent loop – it will execute forever. The inner loop continues as long as \$Target exists. But let's say you don't want to shoot at the target if the pilot ejects and you capture it. So, in line 104 there is a check to see if the target is still an enemy. If it's not an enemy, then the *break* at line 105 executes.

That *break* causes the current, or innermost loop to terminate. If that *break* executes, then execution passed beyond the inner loop's end at line 106 to the *wait* at line 107. The outer loop isn't “broken” too. If you want to break out of two nested loops, you need two breaks.

## Goto

*Goto* is a very simple instruction that does just what it says – it goes to another location. You define where the *goto* will jump to using the *define label* instruction. You first create a label somewhere in your script, and the *goto* instruction causes the script engine to jump to where that label is.

## 4.6 LOOK TO EXAMPLES

The above sections have given a good overview on the different types of instructions that are available to help you organize your scripts.

None of this, though, can really teach you how to structure your scripts. The tools have been explained, the methods haven't. Those methods aren't something that can be taught in any one book. People go to school for years to learn those methods. On the other hand, there are also people (who are sometimes lynched for being excessive smarty-

### Technical Tidbit

Once upon a time in the early days of writing programs, *goto*-like instructions were used a lot. In fact, some programmers used them so much, many programs became almost impossible to read and a nightmare to debug.

Computer science teachers and professors began to teach students that it was very bad to use a *goto* in a program. Many anal professors took it a step further, though, and taught that *goto* should never, ever, **ever**, **EVER** be used in a program. Gleeful teaching assistants would stand hawkish watch over hapless students. The first *goto* to pop up earned a failing grade on the program. Even asking why programming languages had such “evil” instructions like *goto* if it was so bad was a terrible faux pas. Entire lecture halls full of students would gasp at the hapless “newby” who had the temerity to ask such a question. In shame the poor student would be run out of the school.

Ok, it wasn't quite that bad, but *goto* got a really bad wrap. Nowadays, it's not considered to be so fundamentally evil. There are cases where a *goto* or two will significantly simplify a program (much to the dismay of the anal professors of the 80's and 90's). The moral of the story is, if it will make things easier, use it. But don't fall into the *goto* trap – it's not a substitute for a good design.



T

H

M

T

H

R

M

A

T

## The X<sup>2</sup> MSCI Programmer's Handbook

pants) that look simply look at other people's work and learn how to do design and write scripts that way. This is a very good idea, especially when there are so many scripts available to look through and learn from.

Decompress the built-in scripts and look through them. Better yet, download all the signed "bonus plugin" scripts you can from the X<sup>2</sup> web site and look at those. Many of them have comments inside that explain how they work. Some of them might even have comments in a language you understand. Those will be valuable in learning how to organize your scripts.

Learn by example, then learn by doing.





## 5. SCRIPT INTERFACE

It would be pretty useless if you couldn't run your scripts. It also would be pretty useless if you were the only one who every got to run your scripts. The whole point of developing scripts is to release them so others can benefit from them too (or so that you can get fame and the adulation of the X<sup>2</sup> community – that's good too).



This section will explain how your scripts can interface with the real world, or at least the fictional X<sup>2</sup> universe. First, though, some information on how to use the interface to debug you scripts.

### 5.1 DEBUGGING SCRIPTS

The unfortunate reality of script programming is that bugs will be a regular occurrence. Sometimes the reason for a particular script not working isn't immediately obvious. Since you're not going to help anyone or get famous (you might get notorious though) if your scripts don't work, this is where the built-in debugger comes in very handy.

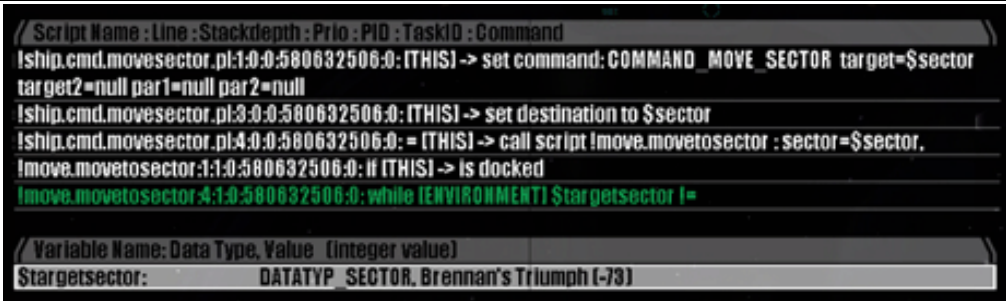
#### Built-In Debugger

The debugger can be enabled for scripts running on any ship – but only for scripts running on a ship. It is enabled on a ship-by-ship basis through the “Script Debugging” menu entry of the scripting menu. Whichever ship's command menu you entered the script menu through is the ship that will have debugging turned on or off through this menu entry. To make this clearer to the user, the game displays the name of the ship “Script Debugging” menu entry.

To activate debugging for that ship, just hit  on “Script Debugging” in the [scripting menu](#). There are two debugging modes, “logging” and “trace”. Hitting  once turns on logging, hitting it twice turns on script tracing.

With script logging, every statement that is executed on that ship is recorded. You can access that log and see what has been run by selecting the “Script Debugger Menu” entry two listings below “Script Debugging”.

With script tracing, the scripting engine enters what is often referred to as a single-step mode. In this mode each line of a script is executed one at a time, only going to the next line when you tell it to. This is also done through the “Script Debugger Menu” command. Selecting this after tracing has been turned on will produce what looks like this:



```
Script Name : Line : Stackdepth : Prio : PID : TaskID : Command
!ship.cmd.movesector.pl:1:0:0:580632506:0: [THIS] -> set command: COMMAND_MOVE_SECTOR target=$sector
target2=null par1=null par2=null
!ship.cmd.movesector.pl:3:0:0:580632506:0: [THIS] -> set destination to $sector
!ship.cmd.movesector.pl:4:0:0:580632506:0: = [THIS] -> call script !move.movetosector : sector=$sector,
!move.movetosector:1:1:0:580632506:0: if [THIS] -> is docked
!move.movetosector:4:1:0:580632506:0: while !ENVIRONMENTI $targetsector !=

Variable Name: Data Type, Value (Integer value)
$targetsector: DATATYP_SECTOR, Brennan's Triumph (-73)
```

Illustration 5.1 - Script Debugger Menu with tracing enabled

The top section lists the script statements. Those in white have already been executed. The one in green is the current one. The bottom window lists any variables that are used in the currently executing statement. Each variable is listed with its name, its data type, and its value.



# T H I S I S I S T O R Y

You will notice that the current statement in green looks a little odd. It is a while instruction built out of an expression. The debugger lists all expressions in [Reverse Polish Notation](#)<sup>5</sup> (also called postfix notation because the operators come after the operands). While this makes it a little less readable for those not familiar with this notation, it has the benefit of showing exactly what the expression interpreter thinks of any expressions you write. If you learn to read this notation, you will be better able to debug problems that arise from expressions you put in your scripts.

In trace mode the current instruction will pause for as long as you want. It will only go to the next instruction when you hit . This allows you to see exactly what occurs in the script. If a conditional isn't executing the way you think it should, you can see exactly why.

## Logging

Another way to debug scripts is through the use of the player log and log files. The player log is great as a quick-and-dirty method of testing something out. If you aren't clear on what result an instruction will give, dump the result to a variable and then write it to the player logbook.

For more complex testing, an external log file is the best way to go. There are three versions of the [write to logfile](#) instruction (see the reference section under [Logbook Commands](#)). They take allow you to write arbitrary data (including the contents of any script variables) to an external file. Even a script that behaves properly will have people that claim it doesn't (the notorious PEBCAK problems, meaning Problem Exists Between Chair and Keyboard, are often the hardest ones to solve). Well written scripts, ones that aren't of trivial length or complexity, should have some sort of debug output. If for no other reason than it can be used to prove the script is working properly.

Do **not** use the logging output capability as a way to get in-character game information out of the game for external analysis. While it might be interesting in some ways to allow players who use your scripts to create nice XML charts of what sorts of things your script is doing, it can also ruin the in-game experience. If a player has to use analytical tools outside of the game in order to get the most from your script, then your script is violating a gaming principal. In fact it's a general showmanship principal. That is, never expose what happens backstage to the audience. It destroys the fundamental contract between player and game, which is the willing suspension of disbelief. A player can't be living in the game universe if you force that player back to reality in order to analyze log files.

## 5.2 SCRIPTS AS COMMANDS

Attaching a script to a menu command is probably the single most common reason for writing scripts in the first place. Doing this is a three step process of writing the command script, preparing an XML language file, and making a setup script.

### Command Scripts

The arguments that a command script takes will depend on what sort of command it is going to be attached to. The only type of command that provides an argument by default are turret commands. When the script attached to a turret command is run, it is provided a number representing which turret it is being run on. Any other arguments in any command script will cause the scripting engine to prompt the player for input corresponding to their type.

There are also a few more things to consider when writing a script that will be attached to a command as opposed to one that you run yourself through the script editor. Normally if one is

<sup>5</sup> See <http://www.calculator.org/rpn.html> for information



# T H I S I S I S U E A T

writing a command script, it is because it is going to get released to the X<sup>2</sup> community. Perhaps even submitted for consideration as an official bonus plugin<sup>6</sup> (signed script) that will be made available on Egosoft's web site. A few rules of thumb for writing a script for “release” are as follows:

1. Make sure the script can handle **all** contingencies. If the script is a trade script, then can it handle its home base being destroyed? If it is a turret script, does it stop shooting if the target stops being an enemy (if you capture the ship)? Does it make assumptions about a sector being at a certain X/Y location on the map? This won't work on custom maps. Spend time thinking about all the possible results at each step in your script. Never make any assumptions that an object still exists later in your script even though it existed earlier. The rule of thumb is, if your script depends on a certain object existing, check to make sure it still exists after every instruction that is an interrupt point (see section 7.2 on page 80).
2. Any text that is displayed to the user must be read from an XML language file. If not, then the script can't be used by players who speak other languages. This isn't necessarily limited to text displayed in the player's message log. It could also be text used to name a ship. Any text that a player will see should be in a language file so it can be translated easily. More on XML language files a little later in this section.
3. The script should not cause any errors – not even ones that the code expects. For example, you might write code like this:

```
100 Target.Exists = $Target -> exists
101 Target.IsEnemy = $Target -> is enemy
102 if $Target.Exists AND $Target.IsEnemy
103 ...
104 else
105 ...
106 end
```

This is incorrect code even though the test at line 102 makes tests for both conditions. It is incorrect because if \$Target doesn't exist, the statement on line 101 will cause an error. Even errors the code expects can cause delays in the scripting engine. If you need to have an “else” based on both conditions, then the code should instead be written like this:

```
100 Target.Exists = $Target -> exists
101 skip if $Target.Exists
102 Target.IsEnemy = $Target -> is enemy
103 if $Target.Exists AND $Target.IsEnemy
104 ...
105 else
106 ...
107 end
```

There are essentially two tests to see if the target exists, but this code won't cause any errors and will be able to be included in a signed “bonus plugin” script.

## XML Language File

Most of the game-text for anything from wares to ship descriptions to menu commands are stored in external XML language files. These language files allow in-game text to be easily translated to different languages.

These files reside in one of two places:

<sup>6</sup> The official contact people for signed “bonus plugin” scripts are **ticaki** and **Burianek**. Contact them on the X<sup>2</sup> forums to discuss getting your scripts signed. You can also attach your scripts to an email and send it to [scripts@egosoft.com](mailto:scripts@egosoft.com)



T

H

M

T

H

R

M

A

T

1. Inside the <X<sup>2</sup>>\t directory, where <X<sup>2</sup>> is the directory that you installed X<sup>2</sup> into.
2. They can be inside container file pairs that act in a similar way to .zip files. These containers each consist of two files, a catalogue (\*.CAT) file and the data (\*.DAT) file. As of version 1.4 of X<sup>2</sup>, there are four of these containers included with the game, named from 01.CAT/01.DAT through 04.CAT/04.DAT. Both the official [X<sup>2</sup> modder kit](#), and the tool [X<sup>2</sup> Modder](#) by OlisJ are capable of extracting the files from these containers. Internally these containers have the same directory structure as the the game directory. XML language files, for example, are stored in the \t directory inside these containers. Any file that is inside one of these containers is treated just as if it were in one of the normal subdirectories under where you installed X<sup>2</sup>.

There is a pattern to the way these XML language files are named. The first two digits is the language code; 44 is English, 49 is German. After the language code, the rest of the filename contains a four-digit number that is used to identify an individual language file to any script that wants to use it. The [load text](#) instruction uses this four-digit code to load the contents of a language file. Which prefix it uses depends on the language of the game the player has. Thus, if you want your added scripts to even work on a version of X<sup>2</sup> that is in a different language from yours, then you will need to provide XML language files for each language.

The four-digit “ID” code you use for your language file should be unique – it should not be used by any other script. Pick a number greater than 1000 that isn't used by anyone else<sup>7</sup> for your XML language file.

Here is a sample XML language file for the fictitious trade command “Maintain Product Quantity”:

```
<?xml version="1.0" encoding="UTF-8" ?>
<language id="44">

  <page id="2008" title="Script Object Commands" descr=" ">
    <t id="430">COMMAND_MAINTAIN_QUANTITY</t>
  </page>

  <page id="2010" title="Commands" descr=" ">
    <t id="430">Maintain product quantity...</t>
  </page>

  <page id="2011" title="Commands" descr=" ">
    <t id="430">MaintainQuant</t>
  </page>
</language>
```

Example 5.1 - Sample XML Language File

Notice that the language ID of the file is stored inside the file, as well as being part of the file's name.

Language files are divided into pages and entries within a page. There are three pages that you will be interested in for creating command scripts:

- 2008 This page contains all the mnemonic names for every command and signal. These mnemonics are used in instructions that take a command or signal as a parameter. For example, the [global script map](#) and the [set script command upgrade](#) instructions. When you place one of these instructions in your script and then select the command/signal parameter, the script editor gives you a list of all current commands. Each of the names of those commands is a mnemonic set in page 2008 of an XML

<sup>7</sup> It is a good idea to check with the X<sup>2</sup> scripting community on what in-game resources you are planning to use in a script, including language file ID codes. Post to the X<sup>2</sup> Scripts and Modding forum at <http://www.egosoft.com/x2/forum/>



T

H

M

T

H

R

M

A

T

language file.

2010 This page contains the long form of the command name. This is the text that is actually displayed on the command menu.

2011 This page contains the short form of the command name. This is also displayed on the command menu (to the right of the long name), and is also used in other information screens as a short-form of the command text.

In the [example](#), the entries within each of the pages has the same id number, 430. The number that you will need to use for your commands depends on the type of command – depends on what menu it will be added to. Each menu type command type has 32 available slots that script-writers can use for extra commands:

Type	Id Numbers	Menu	Mnemonic
Navigation	200-231	Main command menu, Navigation	COMMAND_TYPE_NAV_00 to 31
Combat	300-331	Main command menu, Combat	COMMAND_TYPE_FIGHT_00 to 31
Trade	400-431	Main command menu, Trade	COMMAND_TYPE_TRADE_00 to 31
Special	500-531	Main command menu, Special	COMMAND_TYPE_SPECIAL_00 to 31
Piracy	600-631	Main command menu, piracy	COMMAND_TYPE_PIRACY_00 to 31
Custom	700-731	Main command menu, Custom	COMMAND_TYPE_CUSTOM_00 to 31
General	800-831	Main command menu, General	COMMAND_TYPE_GENERAL_00 to 31
Turret	900-931	Turret menu (under the main)	COMMAND_TYPE_TURRET_00 to 31
Station	1100-1131	Station “Command Console”	COMMAND_TYPE_STATION_00 to 31
Ship	1200-1231	“Additional Ship Commands” menu under the turrets.	COMMAND_TYPE_SHIP_00 to 31

Table 5.1 - Additional command slots

As shown, each of the “extra” commands already has a mnemonic attached. This is why making your own isn't strictly necessary. However, the provided mnemonics are necessarily pretty general. Your script will be a lot more readable if you provide your own. You will, however, absolutely need to provide the long and short command names.

When choosing which “slot” to use for your command, you need to make sure that it hasn't been used in other commands. The first sixteen of each type are reserved for Egosoft use. Generally these are used for signed “bonus plugin” scripts. It may be inevitable that your command script will need to use a slot that someone else's uses. There are a lot of scripts out there that add new commands, and there are only 16 slots available for each menu. Some menus are not used as much. The “Custom”, and “General” menus are less used than, say, the “Navigation” and “Fight”. If your plugin is generally a combat plugin, but also has commands for configuring it, consider using different menus for the configuration commands.

Once you have decided what type of command your script will be, create an XML language file for it as shown in the example, and move to the next step:

## Setup Script

At this point you should the script you want to use for your new command and you should have an XML language file that will be used for the command's menu text. Now all you need is a way to link them together. This is precisely what a setup script is for.





# T H I S I S I S T E R A T E

Section 3.2 on starting on page 10 talked about setup scripts. These are scripts with a name that begins with “setup.”. They are run automatically every time a game is started or loaded.

Your setup script will need to perform three tasks:

1. Load the XML language file. Language files are loaded by the game on demand when a script executes the *load text* instruction. The id number used in this instruction is the four-digit id number you chose as part of the filename for the language file.
2. Link the chosen command with a ware using the *set script command upgrade* instruction. This will make it so the script is not available to a player unless the player has a certain ware. For example, the COMMAND\_GET\_WARE\_BEST command (the menu item Trade->Buy ware for best price) is attached to the ware “Trade Commands Software mk.1”. If you want the command to be available regardless of any ware, use [TRUE] for the ware. See the reference section for details.
3. Link the command with your script. Now all that needs to be done is to tell the scripting engine which script is to be executed when the player selects the command. This is done with the *global script map* instruction.

For our fictional command, a setup script might look like this:

```
001 * First the XML language file - this will load it
002 * from nn2498.xml, where nn is the language number
003 load text: id=2498
004
005 * Now attach the command to the correct ware
006 set ship command upgrade: command=COMMAND_MAINTAIN_QUANTITY
    upgrade=Trade Command Software MK1
007
008 * Now we attach the command to the correct script
009 global ship map: set: key=COMMAND_MAINTAIN_QUANTITY , class=Ship,
    race=Player, script='ship.cmd.maintprod', prio=0
010 global ship map: ignore: key=COMMAND_MAINTAIN_QUANTITY , class=Big
    Ship, race=Player
011
012 return null
```



T

H

M

T

H

R

M

A

T

## 6. REFERENCE

An itemization of all X<sup>2</sup> MSCI instructions, their function, what they return (if anything), and their parameters. They are listed here in the order they appear in the code editor's instruction selection menus.

### 6.1 GENERAL (FLOW CONTROL)

#### **end** (conditional)

Ends an *if* or *while* statement's code block.

#### **else**

Ends the primary code block for an *if* statement and begins the alternative execution code block (the block that is executed in the case that the *if* statement's test is negative)

#### **continue**

Immediately bypasses any remaining code inside a *while* loop's code block. The *while*'s test is then re-run and the loop begins again or terminates if the test fails.

#### **break**

Breaks out of the current *while* loop's code block and begins executing code after that loop's *end* statement.

#### **goto** <label>

Causes the script execution to jump to the point defined by the label you specify.

#### **define label** <label>

Allows you to define the name and location of a label that is used as the jump destination for a *goto* statement.

### 6.2 GENERAL (SCRIPT CALLS)

```
@8 [START|[skip|else] if [not]|while [not]|<retvar> = ] <object  
pointer> -> call script <scriptname> [<parameter>=<value>]  
[...]
```

Runs the script specified by <scriptname> on the object specified by <object pointer>. If <object pointer> is null, then it is run as a global script. If the prefix 'START' is used and <object pointer> is either null, or points to an object that is different from the current object (i.e. not [THIS]), then the script will be “forked” off and executed in parallel with the current script. A new process will be started for the script. In this case, the call will return immediately without waiting for the called script to *return*. If START is not used, your script will wait after issuing this instruction until after the called script ends. This is called 'blocking', as the calling script is blocked until the called script returns.

If this statement is used to *START* a script on another object, then any script already running on that object under task zero will be terminated in favour of the new script.

<sup>8</sup> An '@' character in a scripting statement marks an “interrupt point”. See section 6.1 for details.



A return variable can be assigned to the value *returned* by the called script. In this case, whatever value is passed to the *return* statement in the called script is assigned to the specified variable in the calling script. The result of the call can also be used as a condition in an *if*, *skip*, or *while*. If this is done, then the result can only be tested for whether or not it is null. If it isn't null, then the conditional test passes. If it is null, then it fails.

**return null|<value>**

Ends the currently running script and either returns control to the script that called it, or terminates the process in the case there is no calling script.

**6.3 GENERAL (ARRAYS)****<variable> = array alloc: size=<size>**

Allocates an array with <size> elements and assigns the result to <variable>. The size can be zero, which will give you an array with no elements – this array can be later *appended* to.

Arrays are always zero based (the first element is element zero).

**[skip] if [not]|while [not]|<retvar> = <arrayvar>[<element>]**

Allows you to assign the value of an array element to a variable, or to create a condition or while loop based on whether the element is set to a value or is null.

**<arrayvar>[<element>] = <value>**

Allows you to set the value of the element of an array. The array must have previously been allocated the correct number of elements. You cannot expand the number of elements in an array with this instruction.

**[[skip|else] if [not]|while [not]|<retvar> = ] size of array  
<array>**

Determines the size of an array and either assigns it to a variable, or uses it as a condition in an *if*, *skip if*, or *while* statement.

**<newarray> = clone array <sourcearray> :index <startelement> ...  
<endelement>**

Creates a new array, assigning it to <newarray> with contents of <sourcearray> starting at <startelement> and ending with <endelement>. This new array is allocated when this instruction executes, so a previous call to *array alloc* array is unnecessary.

**copy array <sourcearray> index <startelement> ... <endelement>  
into array <destarray> at index <destelement>**

Copies the contents of <sourcearray> from elements <startelement> to <endelement> into the already existing destination array <destarray>. The destination array must have been previously allocated large enough to hold all the elements.

**insert <value> into array <array> at index <element>**

Inserts the specified value into the specified array. The value becomes the array's new element <element>, and the old element that was at that index moves up one. The target array is one element larger after this instruction executes.



**append <value> to array <array>**

Adds a new value to the end of the specified array. The target array is one element larger after this instruction executes.

**remove element from array <array> at index <element>**

Deletes the specified element from an array. The array is one element smaller after this instruction executes.

**resize array <array> to <size>**

Resizes the target array so that it has exactly <size> elements. This instruction can either make an array smaller or larger.

## 6.4 GENERAL

```
[[skip|else] if [not]|while [not]|<retvar>= ] [<variable>|  
  <constant>|<operator>|<value>] [...]
```

The general purpose expression instruction. This instruction is capable of setting variables, and creating general *if*, *skip if*, and *while* statements. This is the instruction that must be used to create any conditional statement that tests for a condition other than null. An entire loop can be created using nothing but variations of this statement and *end*:

```
Loop = 0  
while Loop < 10  
  Loop = Loop + 1  
end
```

The following lists gives all the supported operators in order of their evaluation precedence:

1. *Primary*: ( and ) precedence override – to explicitly give the order of operator evaluation
2. *Unary*: ~ arithmetic negation (bitwise complement), ! logical negation
3. *Multiplicative*: \* arithmetic multiplication, / arithmetic division, mod arithmetic modulo
4. *Additive*: + arithmetic addition, - arithmetic subtraction
5. *Relational*: < less than, > greater than, <= less than or equal to, >= greater than or equal to
6. *Equality*: == equal to, != not equal to
7. **&** bitwise AND
8. ^ bitwise exclusive OR
9. | bitwise inclusive OR
10. **AND** logical AND
11. **OR** logical OR

**inc <variable>**

Increments the specified variable by one. Functionally identical to:

```
<variable> = <variable> + 1
```

**dec <variable>**

Decrements the specified variable by one. Functionally identical to:

```
<variable> = <variable> - 1
```

**@ [[skip|else] if [not]|while [not]|<retvar>= ] wait  
<milliseconds> ms**

Causes the script to pause for the length of time specified by <milliseconds>. The delay is approximate. Generally it can be counted on that the delay will be at least for the specified length of time. No delay can be less than the length of time it takes for one screen refresh<sup>9</sup>, so a one millisecond wait will delay until the next refresh.

**@ [[skip|else] if [not]|while [not]|<retvar>= ] wait randomly  
from <leasttime> to <mosttime> ms**

Causes the script to pause for a random length of time between <leasttime> and <mosttime> milliseconds. As in the case of *wait*, this is approximate.

Every script that runs for an indefinite period of time should include have at least one of these statements in its main loop. This makes it very unlikely that large numbers of scripts will all become active at the same time.

**<random> = random value from 0 to <maximum> - 1**

Returns a random number in the range of zero to <maximum> minus one. The fact that it automatically subtracts one from the maximum makes it ideal to use for selecting random elements from an array, where <maximum> is the size of the array.

**<random> = random value from <minimum> to <maximum> - 1**

Returns a random number in the range of <minimum> to <maximum> minus one.

**\* <comment>**

A null statement – does nothing but add a comment to your code.

**<version> = script engine version**

Returns an integer with the version of the current MSCI script engine. As of version 1.4 of X<sup>2</sup> the number is 25.

This can be used in the event you need to write a script that needs to know whether certain instructions are available – perhaps if it needs to support more than one version of the game.

**<priority> = get script priority**

The priority of a running script determines whether or not another script or signal can interrupt it. This instruction returns the priority that the current script is running at.

<sup>9</sup> Take the current refresh rate in frames per second and divide that into 1000 to get the number of milliseconds per frame. This will be the minimum wait time. A utility called 'fraps' (<http://www.fraps.com>) can tell you what your current frame rate is. Of course, the frame rate in any given situation will depend on the processing power of the user's computer who is running your script. But it's a good thing to remember that any wait placed in your script will delay for at least one frame. For example, any wait placed inside a loop will cause the loop to iterate at **most** once per frame.



**set script priority to <priority>**

Changes the priority that the current script is running at to <priority>.

**[skip|else] if [not]|while [not]|<retvar>= is script with prio  
<priority> on stack**

The call stack is the list of scripts running in a given process. Every time one script calls another, that script is added to the stack. Scripts are also added to the stack during an interrupt or signal. Determines if a script of a given priority is on the call stack. This instruction always runs on the current task – it cannot be used to determine if a script of a given priority is running under a different task ID.

Different script priorities are used for different functions. Most notably, the different signals operate at different priorities. This instruction is often used to determine if a particular signal has been received by a ship. By checking to see if a script of priority 99 is on the stack, for example, a script can tell if the ship it is running on has been attacked and was in the process of fighting back against its attacker.

Prio	Used by
0	All scripts by default
50	Flee response to SIGNAL_ATTACKED
99	Attack response to SIGNAL_ATTACKED
100	SIGNAL_ATTACKED
150	SIGNAL_LEADERNEEDSHELP SIGNAL_FOLLOWERNEEDSHELP
200	SIGNAL_FORMATIONLEADERCHANGED
300	SIGNAL_CAPTURED
10000	SIGNAL_KILLED

Table 6.1 - Script Priorities

**<object> -> start task <taskID> with script <scriptname> and  
prio <priority>: arg1=<value1> arg2=<value2> arg3=<value3>  
arg4=<value4> arg5=<value5>**

Starts a new task on the target <object> with the given <taskID> and runs the specified script on it, setting it at the given <priority> and passing it up to five arguments. Arguments that are not used should be set to null.

This instruction can be used to start turret scripts (tasks with a task ID of from 1 to 6), so-called “additional ship command” scripts (tasks with a task ID of 10 or 11), and station tasks (tasks with an ID of 10-19 on stations).

See section 7.1 on page 78 for an in-depth explanation of tasks.

**<object> -> interrupt task <taskID> with script <scriptname> and  
prio <priority>: arg1=<value1> arg2=<value2> arg3=<value3>  
arg4=<value4> arg5=<value5>**

If a task is running on task ID <taskID> on the given <object>, and if the currently running script on that task's stack is running with a priority less than <priority>, then this instruction will cause that script to pause at it's next interruptible statement (statements marked with a '@' prefix are interruptible) and cause the script <scriptname> to begin running. When the new script terminates, the interrupted script will resume at the interrupted statement.

Do note, that the task must already be running to be able to be interrupted.

See section 7.2 on page 80 for an in-depth explanation of interrupts.



# T H I S I S I S S U E A T

**<object> -> interrupt with script <scriptname> and prio  
<priority>: arg1=<value1> arg2=<value2> arg3=<value3>  
arg4=<value4>**

The same as the *interrupt task* instruction, except this one operates exclusively on task zero (the “main” task of an object). For some reason known only to the developers, this version of the instruction only accepts up to four arguments, unlike the general form which supports five.

See section 7.2 on page 80 for an in-depth explanation of interrupts.

**[skip|else] if [not]|while [not]|<retvar>= get task ID**

Returns the current task's ID number. Task zero is always the “main” task of an object. When you give a normal navigation, combat, or trade command to a ship, for example, the script that carries out that order runs on task zero of the target object. Tasks one through six run scripts for turrets one through six. Tasks ten and eleven are used for the scripts that run from a ship's “additional ship commands” menu.

**[skip|else] if [not]|while [not]|<retvar>= get pid**

Every time a script runs, it is given a unique identifier called 'pid'. The term *pid* traces its routes to Unix systems where this meant “process ID”. The distinction between a process and a task in X<sup>2</sup> is a thin one. A task ID is a short form to reference a process that is attached to a given object to perform a given set of functions. A process ID is the internal reference number given each time a new instance of a script is instantiated. Different ships will have script that run under identical task ID numbers. A process ID number for a running process will never be the same as the process ID for any other currently running process. See section 7.1 for an in-depth look at processes and tasks.

**<object> -> interrupt with script <scriptname> and prio:  
<priority>**

Use this instruction to interrupt task zero on an object if you don't need to supply the interrupting script any arguments.

See section 7.2 on page 80 for an in-depth explanation of interrupts.

**<object> -> connect ship command/signal <commandorsignal> to  
script <scriptname> with prio <priority>**

Changes the mapping between a command/signal and a script for a single ship or base so for that object, the specified script is run whenever the command/signal occurs.

**<object> -> set ship command/signal <commandorsignal> to global  
default behaviour**

Undoes a *connect ship command/signal* instruction and returns the handling of <commandorsignal> to the default behaviour on that object.

**<object> -> ignore ship command/signal <commandorsignal>**

Causes the given command or signal to be ignored on the specified ship.

**enable signal/interrupt handling: [TRUE]|[FALSE]**

Turns signal and interrupt handling on or off. This instruction works for the process in which the script is running – not globally.

**[skip|else] if [not]|while [not]|<retvar>= is signal/interrupt handling on**

Returns true if signal/interrupts are enabled for the current process, false if not.

**global script map: set key <commandorsignal>, class=<class>| null, race=<race>|null, script=<scriptname>, prio=<priority>**

Globally maps a script to a command or signal. The command can also be used selectively to map only for a particular race, or for a particular class of object, or a combination of the two. If race is null, all races can use the command. If class is null, all object classes can use it.

**global script map: remove key <commandorsignal>, class=<class>| null, race=<race>|null**

Removes the link between a command/signal and a script either globally, or selectively for a class of objects or a race's objects or a combination of the two.

This is often used to create exceptions to the setting of a link. For example, if for some reason you want a command to be available to all ships except for M1 class ships, you could do the following:

```
100 global script map: set key COMMAND_MY_COMMAND, class=ship, race=null,  
    script='plugin.myplugin.myscript', prio=0  
101 global script map: remove key COMMAND_MY_COMMAND, class=M1 Battleship,  
    race=null
```

**global script map: ignore key <commandorsignal>, class=<class>| null, race=<race>|null**

Causes a particular command or signal to be ignored either globally, or for a particular race's objects or for a class of object or for a combination of the two.

**set script command upgrade: command=<commandorsignal>, upgrade=<ware>**

Links a command with a ware. Once the link has been made, the command will only be available to a player's ship if that ship has the ware (upgrade) on board.

**<ware> = get script command upgrade: command=<commandorsignal>**

Returns the ware that is required (that has been set with a previous *set script command upgrade* instruction) for a command to be enabled on a player's ship.

**set script command: <commandorsignal>**

Tells the script engine what your script is currently doing. This is most often used as a way to display the currently worked on task to the player in whatever menu the player used to run the command. It is also used sometimes for an interrupt or signal handler script to determine what sort of activity it was interrupting.

**[skip|else] if [not]|while [not]|<retvar>= get script command**

Returns whatever the last *set script command* instruction has set the current process' command to. This is most frequently used by an interrupt or signal handler to determine what type of activity it is interrupting.



T

H

M

T

H

R

M

A

T

**set script command target: <target>**

The script command target can be set individually for each task running on an object. This instruction sets the target for the task it is running on to <target>. This is different from the *set command target* instruction which sets the target for the whole ship. This instruction is used to set the target, for example, on a single turret of a ship. The target can then be retrieved by an interrupting script which can then determine what the interrupted script was doing when it was interrupted.

**[skip|else] if [not]|while [not]|<retvar>= get script command target**

Returns the command target for the current process.

**<retvar> = get datatype[<value>]**

Returns the data type of the specified variable.

**[skip|else] if [not]|while [not]|<retvar>= is datatype[<value>]  
==<datatype>**

Returns a [TRUE] or [FALSE] depending on whether or not the data type of <value> is equal to <datatype>.

**<retvar> = read text: page=<pageid> id=<textid>**

Returns the text stored in an external XML language file. The file read from must have been previously loaded with a *load text* instruction. Both <pageid> and <textid> are numbers which specify the XML page and id tags where the text is stored in the language file.

**<retvar> = sprintf: fmt=<format>, <value1>, <value2>, <value3>,  
<value4>, <value5>**

Formats a string according to the format specifier in <format>. This derives from the 'C' language standard library function by the same name. The format string can contain from one to five “%s” symbols. Each time a %s is encountered, it is replaced with the corresponding value. For example:

```
Message = sprintf: fmt='You flying the ship %s in sector %s.',  
[playership], [sector], null, null, null
```

The above will store the string 'You are flying the ship <shipname> in sector <sector>' in the variable Message.

**<retvar> = sprintf: pageid=<pageid> textid=<textid>, <value1>,  
<value2>, <value3>, <value4>, <value5>**

Identical to the first form of sprintf above, except that it uses text stored in an XML language file as the format specifier as per the *read text* instruction.





T

H

M

T

H

R

M

A

T

### **load text: id=<languagefileid>**

Loads in an XML language file. The filename is made from a combination of the language code of the language the player's copy of X² is combined with a four digit number <languagefileid>. The language code is actually the international telephone prefix for the language's country of origin.

For example, a *load text* instruction with an id of 21 on an English version of X² would load in the file 440021.xml.

All language files are stored in the <X²>/t directory, where <X²> is the directory where X² is installed.

Code	Language
7	Russian
33	French
39	Italian
44	English
48	Polish
49	German

Table 6.2 - Language Codes

### **[skip|else] if [not]|while [not]|<retvar>= state of news article: page=<pageid> id=<textid>**

Returns whether or not a BBS news article is "active" (whether or not it was currently being shown on the BBS network or not). Scripts are used in conjunction with the game's internally hard-coded BBS engine to make news articles come and go.

### **set state of news article: page=<pageid> id=<textid> to [TRUE] | [FALSE]**

Enables or disables a BBS news article – sets whether or not it is currently being shown on the BBS network. Scripts are used in conjunction with the game's internally hard-coded BBS engine to make news articles come and go.

### **[skip|else] if [not]|while [not]|<retvar>= system date is month=<month>, day=<day>**

Returns [TRUE] if the game date is on or past the given date. Otherwise, returns [FALSE].

### **<retvar> = playing time**

Returns the amount of game time that has elapsed in seconds.

### **infinite loop detection enabled=[TRUE] | [FALSE]**

Turns on or off the scripting engine's infinite loop detection for the running process. Intended to shut down a runaway script. In practise, it rarely works. The best protection against runaway scripts is liberal use of the *wait* instruction and good programming.

### **set script command upgrade: command=<commandorsignal>, upgrade=<ware> script=<scriptname>**

Similar to the previously described *set script command upgrade*, except this version allows the setting of a check script. The check script is a script that is run prior to a menu being displayed that includes the given command. The return value of the check script determines if the given command is allowed to run or not. If it is not allowed to run, then the option for it is greyed out in the command menu. The possible return codes for check scripts are in the constants list available in the script editor when you pick 'Select Constant' and are listed in the following table:



<i>Constant</i>	<i>Description</i>
<code>CmdConCheck.OneTime</code>	The check script is run once for each time the menu is displayed. The status of the command is thus not checked again unless the menu is terminated and redisplayed. Bitwise OR (the <code> </code> operator) this with one of the bottom four results.
<code>CmdConCheck.Infinite</code>	The check script is run continuously while a menu is being displayed. If the status changes, then the menu is updated while it is being displayed. Bitwise OR (the <code> </code> operator) this with one of the bottom four results.
<code>CmdConCheck.NeedHomeBase</code>	The command is available only if the ship has any home base set.
<code>CmdConCheck.NeedHomeStation</code>	The command is available if the ship has a station (not another ship) set as its home base.
<code>CmdConCheck.Available</code>	The command is available.
<code>CmdConCheck.Disabled</code>	The command is disabled.

Table 6.3 - Check Script Return Constants

The following is an example of a check script. It is the script used to determine if the Navigation command, “Jump to sector” is available or not.

```
001 $flags = [CmdConCheck.OneTime]
002 if $ship -> get amount of ware Jumpdrive in cargo bay
003 * jumpdrive installed, allow command
004   $flags = $flags | [CmdConCheck.Available]
005 else
006 * no jumpdrive but nav software mki is installed, disable command
007   $flags = $flags | [CmdConCheck.Disabled]
008 end
009 return $flags
```

Because the *CmdConCheck.OneTime* constant is ORed in to the returned result, this check script is only run once (just before each time the Navigation commands menu is displayed). The script checks for the Jumpdrive ware – if it is present, the “Jump to sector” command is made available. If it is not present, the command is disabled.

#### **set local variable: name=<varname> value=<value>**

This instruction takes any script variable that is running, and stores its value on the object (ship or station) on which the script is running. The term “local variable” is probably unfortunate – the concept is more akin to storing a file on a hard drive. Think of <varname> as the file name, and <value> as the data in the file. Any variable type can be stored in this way, including an entire array.

The difference between a global variable and a local variable, as the *set local* and *set global* instructions use the terms, is whether or not the value is stored locally on a particular ship or base, or stored globally. To continue the file analogy, the *set local variable* instruction is like storing a file in a directory, where each and every ship and station has its own directory. The *set global variable* instruction is like storing the file in the root directory – any script running on any ship or base can access or change the value when it is stored as a global variable.

#### **[skip|else] if [not]|while [not]|<retvar>= get local variable: name=<varname>**

Returns the value of a previously set local variable (see: *set local variable*). If the local variable was not previously set, then this instruction returns null.



T

H

M

T

H

R

M

A

T

**set global variable: name=<varname> value=<value>**

This instruction takes any script variable that is running, and stores its value. The term “global variable” is probably unfortunate – the concept is more akin to storing a file on a hard drive. Think of <varname> as the file name, and <value> as the data in the file. Any variable type can be stored in this way, including an entire array.

The difference between a global variable and a local variable, as the *set local* and *set global* instructions use the terms, is whether or not the value is stored locally on a particular ship or base, or stored globally. To continue the file analogy, the *set local variable* instruction is like storing a file in a directory, where each and every ship and station has its own directory. The *set global variable* instruction is like storing the file in the root directory – any script running on any ship or base can access or change the value when it is stored as a global variable.

**[skip|else] if [not]|while [not]|<retvar>= get global variable: name=<varname>**

Returns the value of a previously set global variable (see: *set global variable*). If the global variable was not previously set, then this instruction returns null.

**al engine: register script=<scriptname>**

Registers an Artificial Life Engine plugin script. The script registered with this instruction becomes the main event handler for the AL plugin.

See section 7.3 on page 84 for more information.

**al engine: unregister script=<scriptname>**

Unregisters an Artificial Life Engine plugin script. Essentially turns off that AL plugin.

**al engine: set plugin <pluginname> description to <description>**

Sets the description (the text that is shown on the Artificial Life Settings menu) for an AL plugin.

See section 7.3 on page 84 for more information.

**al engine: set plugin <pluginname> timer interval to <interval>s**

Sets the interval that the AL plugin timer event occurs at to the specified number of seconds.

See section 7.3 on page 84 for more information.

**<retvar> = get script version**

Returns the version number for the currently running script.

**<retvar> = get script name**

Returns the name of the currently running script.

**[skip|else] if [not]|while [not]|<retvar>= is plot <plotnum> state flag <plotstate>**

The instruction returns [TRUE] if the specified plot is in the specified state. In *The Threat* there are three plots, each with many states. A list of each of the states for all the plots is in [Annex A.1](#).



**<retval> = get random name: race=<race>**

Creates a random name that is appropriate for the given race.

**<retval> = get khaak aggression level**

Returns the current level of aggression for the Khaak. This is a global setting from zero to one hundred. Fifteen is the standard level. Higher than that will increase the number and size of Khaak clusters that spawn randomly.

**set khaak aggression level to <level>**

Sets the current level of aggression for the Khaak. This is a global setting from zero to one hundred. Fifteen is the standard level. Higher than that will increase the number and size of Khaak clusters that spawn randomly.

## 6.5 AUDIO COMMANDS

**play sample <samplenum>**

Plays a sound effect sample. A chart of the possible effects is in [Annex A.2](#).

**play sample: incoming transmission <transmissiontype>, from  
object <source>**

Sends a voice message to the player with news of a hail, incoming transmission, scan, or SOS depending on <transmissiontype>. The obvious intention was to also support informing the player of which ship originated the message, but as of version 1.4 of X<sup>2</sup> – *The Threat*, this does not function.

There are four constants available in the script editor's "Select Constant" list that are specifically for this instruction:

<i>Constant Name</i>	<i>Value</i>	<i>Message Spoken</i>
[IncomingTransmission.Greeting]	1362	"We are being hailed"
[IncomingTransmission.Message]	1361	"Incoming message"
[IncomingTransmission.Scanned]	1363	"We are scanned"
[IncomingTransmission.SOS]	1360	"Emergency message from"

*Table 6.4 - Incoming Message Constants*

The values for the constants are taken from the text list found in the general XML language file for the game (xx0001.XML where xx is your language code) under page 13. See the *speak text* instruction for more information.

**<object> -> send audio message <messagetype> to player**

Sends an audio/video message to the player with the ship specified in <object> being the "source" of the message. The message is different depending on the race that owns <object>, or according to the race of the pilot (in the case of pirate-owned ships). There are four classes of messages, each with its own symbolic constant in the script editor's "Select Constant" list:



<i>Constant Name</i>	<i>Value</i>	<i>Message Type</i>
Comm.DLG_POL_ILLEGAL_GOODS	75	Illegal goods detected on the player's ship – drop the goods
Comm.DLG_POL_LAST_WARNING	76	Last warning to drop the illegal goods on the player's ship
Comm.DLG_POL_LEGAL_GOODS	77	Scanned ship, no illegal goods
Comm.C_START_FIGHTING	4	Message send as the ship is about to attack you

Table 6.5 - Audio Messages

**send incoming message <message> to player: display it=[TRUE] | [FALSE]**

The player will hear the “incoming message” signal and voice, and <message> will be displayed in the player's message log. If display it=[TRUE], then the message will pop up in front of the user immediately. This is generally a very bad thing to do unless the message is of critical importance<sup>10</sup>.

**[START|[skip|else] if [not]|while [not]|<retvar>=] speak text: page=<pagenum> id=<idnum> priority=<priority>**

This instruction can play any game voice message. It can be used to do the same task as [play sample: incoming transmission](#), and with some good scripting, can give the player the source of the message. It can also play the same voice clips as the [send audio message](#) instruction, although it will not display the video.

The common denominator to any message played with this instruction is that the text for the message is stored in an XML language file. If the player has not turned off the displaying of subtitles, this instruction can display as a subtitle the text from any message in any language file – even user-created language files supplied with scripts. If the message has a corresponding audio component, it is played at the same time.

The file nn0001.XML (where nn is your [language code](#) 44 for English) contains the text for all the phrases where there is a corresponding audio or audio/video recording. See [Annex A.3 Speech Samples Catalogue](#) for a breakdown of most of the playable voice clips.

## 6.6 LOGBOOK COMMANDS

**write to player logbook <value>**

Writes any value as a message in the player's logbook. References to objects will return the name of that object. For example, writing [PlayerShip] will write the name of the current ship the player is flying.

**write to player logbook: printf: fmt=<format>, <value1>, <value2>, <value3>, <value4>, <value5>**

Formats and writes a message to the player's logbook. Formatting is performed the same as for the [sprintf](#) instruction.

<sup>10</sup> Causing the message to display immediately is a good way to ensure that your script will never be signed as an official “bonus plugin”



**write to player logbook: printf: pageid=<pageid>  
textid=<textid>, <value1>, <value2>, <value3>, <value4>,  
<value5>**

Formats and writes a message to the player's logbook. The format string is read from an XML language file as per the *sprintf: pageid* instruction.

**<object> -> write to player logbook <value>**

It's not known what the intention of this instruction was – perhaps to put a prefix on a message indicating its source. In any case, it doesn't do anything.

**write to logfile #<lognumber> append=[TRUE]|[FALSE]  
value=<value>**

Writes <value> to an external file named logNNNNN.txt where NNNNN is the number specified in <lognumber>. If append is true, then the value is added to the end of the log file. If append is false, then the contents of the log file are replaced with <value>.

**write to logfile #<lognumber> append=[TRUE]|[FALSE] printf:  
fmt=<format>, <value1>, <value2>, <value3>, <value4>,  
<value5>**

Formats and writes text to an external file. See *write to logfile* and *sprintf* for more information on the log file and the formatting respectively.

**write to logfile #<lognumber> append=[TRUE]|[FALSE] printf:  
pageid=<pageid> textid=<textid>, <value1>, <value2>,  
<value3>, <value4>, <value5>**

Formats and writes text to an external file. See *write to logfile* and *sprintf: pageid* for more information on the log file and the formatting respectively.

## 6.7 FLY COMMANDS

Many of the return values for flight and combat instructions are from a group of constants reserved for this purpose. They start with FLRET\_ and are available to select (such as to put in an *if* statement) from the “Select Constant” menu in the script editor.

Please make special note of the return values for each instruction. There is little consistency between instructions (even related instructions) on which conditions return which result.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> fly to home  
base**

Causes the specified ship to undock, fly to the sector where its home base is located, and dock.

This instruction will return FLRET\_LANDED on success, FLRET\_BREAK if another “fly command” instruction is executed on the ship before it completes, FLRET\_INTERRUPTED if the script executing this instruction is interrupted, or FLRET\_ERROR on error. Not having a home base set on the target ship is an error.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> fly to  
station <station>**

Causes the specified ship to undock, fly to the sector where <station> is located, and dock.





This instruction will return FLRET\_LANDED on success, FLRET\_BREAK if another “fly command” instruction is executed on the ship before it completes, FLRET\_INTERRUPTED if the script executing this instruction is interrupted, or FLRET\_ERROR on error.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> fly to sector <sector>**

Causes the specified ship to undock and fly to <sector>.

This instruction will return FLRET\_NOCOMMANDS either when it arrives at the destination sector or if the destination sector is unreachable, FLRET\_BREAK if another “fly command” instruction is executed on the ship before it completes, or FLRET\_INTERRUPTED if the script executing this instruction is interrupted, or FLRET\_INVALIDPARMS if <ship> is not valid.

**WARNING:**  
An invalid argument for <sector> in this instruction will cause the ship to fly to Kingdom End

**[skip|else] if [not]|while [not]|<retvar>= <object> -> find nearest enemy ship: max.dist=<distance>**

Returns the nearest enemy ship within the specified range, or null if there are none.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> find nearest enemy station: max.dist=<distance>**

Returns the nearest enemy station within the specified range, or null if there are none.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> fire lasers on target <target> using turret <turretid>**

Fires the ship's main (if <turretid> is zero) turret lasers at <target>. It should be noted that there is actually no requirement to have the ship exactly face the target in the case of the main lasers, or to turn a turret to face the target in the case of turret lasers. There is actually no enforced correlation between the direction a turret “faces” (ie: the direction a turret's camera is facing) and the direction the lasers fire. The target does, however, need to be within the firing arc of the guns. The refire delay for the weapons in the turret is enforced – the instruction will fail if an attempt is made to fire the weapons faster than they are capable.

Returns [TRUE] (1) on success, [FALSE] (null) on failure.

**@ [skip|else] if [not]|while [not]|<retvar>= <object> -> turn turret <turretid> to target <target>: timeout=<timeout>ms**

Turns the specified turret's camera to face <target>. This instruction is used to enforce a relationship between a turret's camera and its guns. The general structure of a script that will control a turret, is:

1. Obtain a target
2. Turn the turret to face the target
3. If FLRET\_FIREFREE is returned when you turn the turret, then *fire lasers on target*.
4. Repeat from step 2 until one side cries “Hold, enough!”

It should be noted that the timeout time only indicates how much time the camera has to turn to face the target – not how long the instruction will actually take to execute. In practise, the delay introduced by this instruction ranges in the 150-250ms range regardless of the timeout specified. This makes it very difficult (read impossible) to fire some weapons, such as mass drivers, at anywhere near their rated firing rate unless the turn-turret step is ignored.<sup>11</sup> Other

<sup>11</sup> Having a script signed that ignores the “turn turret to target” step will be near to impossible. Regardless of the firing rate



# T H I S I S I S U E A T

weapons, such as HEPT/PPC with 270ms re-fire delays can, with very careful timing, be fired at close to their maximum re-fire capacity.

Returns FLRET\_FIRFREE on success, FLRET\_TIMEOUT if the timeout period elapses before the camera is able to turn to face the target, FLRET\_INTERRUPTED if the script executing this instruction is interrupted, FLRET\_NOCOMMANDS if an invalid turret is specified, or null if any of the other arguments (including <object>) are invalid.

**@ [skip|else] if [not]|while [not]|<retvar>= <ship> -> attack  
run on target <target>: timeout=<timeout>ms**

The fighter equivalent to *turn turret to target*, this instruction is used to point a ship at a target in preparation for firing using the ship's main lasers. This instruction turns the ship to the target and accelerates to maximum. If the ship is pointed at and in firing range of the target within <timeout> milliseconds, then FLRET\_FIRFREE is returned signalling that it is clear to fire weapons. Further calls to this instruction do not initiate a “new” attack run, they will merely return FLRET\_FIRFREE to signal that the ship is still on its attack run. This will continue to happen until at some point the target ship manoeuvres out of range, or the firing ship gets so close that the collision avoidance kicks in. This, then, is the general structure of a fighter combat script:

1. Obtain a target.
2. Approach the target approximately to weapons range.
3. Perform attack run
4. If FLRET\_FIRFREE is returned, *fire lasers on target*..
5. Repeat from step 3, perhaps performing a defensive manoeuvre from time to time.

Of course, this is just a simple outline, but with allowances for following a target to other sectors is pretty much exactly what the built-in fight.attack.object script does.

Returns FLRET\_FIRFREE on success, FLRET\_TIMEOUT if the timeout period elapses before entering firing range, FLRET\_BREAK if an invalid target is specified or if the instruction is “cancelled” before completing (such as if the ship gets too close to the target and automatically performs collision avoidance), FLRET\_INTERRUPTED if the script executing this instruction is interrupted, or FLRET\_INVALIDPARMS if an invalid <ship> is specified.

**@ [skip|else] if [not]|while [not]|<retvar>= <ship> -> defensive  
move: type=<type>, intensity=<intensity>,  
timeout=<timeout>ms, avoid object=<object>|null**

Causes the ship to begin a defensive manoeuvre – breaking off, spirals, and whatnot. Currently setting the type of manoeuvre does nothing – this is completely controlled by the hard-code. Set this to null in all scripts. Intensity is in percent – from 1 to 100, and indicates how sharp the manoeuvres will be. An object to avoid can be specified, in which case the ship will attempt to move away from that object. This can be left null to perform general evasive manoeuvres.

Returns FLRET\_NOCOMMANDS after the timeout expires (assume success), FLRET\_INTERRUPTED if the script executing this instruction is interrupted, or null if <ship> is invalid.

**@ [skip|else] if [not]|while [not]|<retvar>= <ship> -> move to  
ware object <ware> for collecting: timeout=<timeout>ms**

Manoeuvres a ship to prepare it to collect a flying ware (a ware that is in space and collectible). When finished the manoeuvre (if successful) the ship will be directly pointed at the ware and

---

issue, this is considered “cheating” on the part of a script.



within range to either collect it or shoot it.

Returns FLRET\_FIRFREE on success, FLRET\_BREAK if another “fly command” instruction is executed on the ship before it completes or if <ware> is an invalid object<sup>12</sup>, FLRET\_INTERRUPTED if the script executing this instruction is interrupted, or FLRET\_INVALIDPARMS if <ship> is invalid.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> catch ware object <ware>**

Brings <ware> into the cargo bay of a ship once the ship has been put into position by use of the *move to ware object for collecting* instruction. The ware must be of a size class that the ship is capable of carrying, and the collecting ship must have enough room in its cargo bay for it. After executing this instruction, <ware> will be removed from the sector, whether or not the collecting ship has room for it.

Returns [TRUE] (1) if the ware is added to the collecting ship, [FALSE] (0) if the ware is not added to the collecting ship, or if any of the arguments are invalid.

**@ [skip|else] if [not]|while [not]|<retvar>= <ship> -> move around <timeout> ms**

While there are many times in the X universe where a ship isn't doing anything, there seems to be a cardinal rule that no ship can **look** like it's not doing anything. This instruction does exactly what it says – causes a ship to just move around aimlessly for the specified length of time. Most of the universe's capital ships are doing this most of the time – which is why you generally don't want to get too close to one with a capital ship of your own, because this instruction emulates a pilot with severe myopia who has consumed large amounts of space fuel.

Returns FLRET\_NOCOMMANDS when the <timeout> period elapses, or FLRET\_INVALIDPARMS if <ship> is invalid.

**@ [skip|else] if [not]|while [not]|<retvar>= <ship> -> escort ship <target>**

Causes a ship to escort (follow) another, launching the ship first if it is docked/landed. This instruction has no timeout period – it will continue *ad infinitum*. This instruction is also very difficult to cause to break. Most “fly command” instructions, when they are executed in one process will stop and return FLRET\_BREAK if another process executes a different “fly command” instruction on the same ship. Not so with this instruction. It is possible to get control of the ship back from this instruction with a script in a different process by using a *set follow mode [FALSE]* instruction, but that will not cause the original *follow object* instruction to return. Once one script has executed this instruction on a ship, only the destruction or removal (to another sector) of the escorted ship, or the termination or interruption of the script will cause it to return.

Returns FLRET\_BREAK if <target> is invalid or if the escorted ship is destroyed or moves out of sector, FLRET\_INTERRUPTED if the script executing this instruction is interrupted, or FLRET\_INVALIDPARMS if <ship> is invalid.

<sup>12</sup> In this case, invalid object means if it is passed a value that isn't an object or an object that doesn't exist. If <ware> is, for example, actually a ship then the instruction will still work. In this sense, there is little difference between this instruction and *attack run on target*.



**@ [skip|else] if [not]|while [not]|<retvar>= <ship> -> escort ship <target>: timeout=<timeout> ms**

Similar to *escort ship*, except this instruction will time out after a <timeout> milliseconds.

Returns FLRET\_BREAK if <target> is invalid or if the escorted ship is destroyed or moves out of sector, FLRET\_TIMEOUT if the timeout period elapses, FLRET\_INTERRUPTED if the script executing this instruction is interrupted, or FLRET\_INVALIDPARMS if <ship> is invalid.

**<ship> -> set formation <formation>**

Sets the formation for all ships that are following the specified ship. There are seven symbolic constants (available through the script editor's "select constant" list) available for <formation>

Constant	Value	Example
[Formation.Delta]	0	
[Formation.Line]	1	
[Formation.X]	2	
[Formation.XDelta]	3	
[Formation.BigShipEscort]	3	
[Formation.Pyramid]	4	
[Formation.Random]	5	

**<ship> -> add to formation with leader <leadership>**

Adds a ship to the flight formation of <leadership>. The target ship will take up a slot in the leader's formation – when the target ship is ordered to *escort* the leader, it will do so in formation. The target ship also becomes eligible to receive formation-associated *signals*.

**<ship> -> remove from any formation**

Removes the specified ship from any formation it may be in.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get formation leader**

When <ship> is any ship in a formation (including the lead ship), this instruction will return the lead ship.

Returns the lead ship of a formation (if any), or null if <ship> is invalid or is not in any formation.

**<retvar>= <ship> -> get formation follower ships**

Returns an array of ships that are the followers of a formation in which <ship> is a member, or null if <ship> is invalid or if there are none.

**START <object>-> command <command>: arg1=<value1>, arg2=<value2>, arg3=<value3>, arg4=<value4>**

Starts a command on an object as though a player had selected the command through the menuing system, except that with this method, the command's arguments are preselected.

**<object> -> send signal <signal>: arg1=<value1>, arg2=<value2>, arg3=<value3>, arg4=<value4>**

Sends a *signal* to the target object. A signal sent in this manner cannot be distinguished from a signal that occurs “naturally”.

See section 7.2 on page 80 for more information on signals.

**@ [skip|else] if [not]|while [not]|<retvar>= <ship> -> follow object <target> with precision <precision> m**

Causes the ship to follow the target object, launching first if it is landed/docked. Whereas *escort ship* is intended for use in formation flying and making one ship protect another, this instruction is mostly used to cause a ship to get up close to something as a prelude to something else – often an attack.

The autopilot will attempt to get the ship within <precision> metres of the target object. Like the *escort ship* instruction, this one is almost impossible to terminate. Executing a countervailing “fly command” instruction in another process will not cause this instruction to break. It is possible to get control of the ship back from this instruction with a script in a different process by using a *set follow mode [FALSE]* instruction, but that will not cause the original *follow object* instruction to return. Only the destruction of the target object, its removal to another sector, or the termination or interruption of the script running this instruction will stop it. Be very careful about using this instruction to make a ship follow a faster ship. In most cases, it is highly recommended to use the *follow object ... timeout=<timeout>* instruction instead.



# T I M E R U M A T

Returns FLRED\_TARGETREACHED upon success, FLRET\_INTERRUPTED if the script executing this instruction is interrupted, FLRET\_BREAK if the target is invalid or if the target is destroyed or moves to another sector, or FLRET\_INVALIDPARMS if <ship> is invalid.

**@ [skip|else] if [not]|while [not]|<retvar>= <ship> -> follow  
object <target> with precision <precision> m :  
timeout=<timeout> ms**

Similar to *follow object*, except this one will time out after <timeout> milliseconds.

Returns FLRED\_TARGETREACHED upon success, FLRET\_TIMEOUT if the timeout period expires, FLRET\_INTERRUPTED if the script executing this instruction is interrupted, FLRET\_BREAK if the target is invalid or if the target is destroyed or moves to another sector, or FLRET\_INVALIDPARMS if <ship> is invalid.

**<object> -> set follow mode [TRUE]|[FALSE]**

Turns follow mode on or off for <object>. Turning follow mode off will allow a different process to take control of a ship that is executing an *escort ship* or *follow object* instruction. Doing this, however, will not cause the original *escort* or *follow* instruction to return.

**[skip|else] if [not]|while [not]|  
<retvar>= <ship> -> get follow  
mode**

Returns [TRUE] if <ship> is currently following another object, [FALSE] if not.

**<ship> -> set destination to  
<destination>**

Sets the destination for <ship>. Other than displaying this for the player on a ship's info screen, this has no effect. The destination can be set to an object in space, or to a sector. The ship doesn't actually have to be doing anything to have its destination set.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get  
destination**

Returns the destination of a ship as set by the *set destination* instruction.

**<ship> -> set attack target to <target>**

Sets the attack target for a ship. It has no function other than to change the text displayed on the ship's information screen beside the "Action" heading to "Attacking <target>".

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get attack  
target**

Returns the attack target of a ship as set by the *set attack target* instruction.



Illustration 6.1 – Ship's info screen showing its destination





**@ [skip|else] if [not]|while [not]|<retvar>= <ship> -> move to position x=<xcoordinate> y=<ycoordinate> z=<zcoordinate> with precision <precision> m**

Causes a ship to move to within <precision> metres of the specified position. The position coordinates are in metres from the centre of a sector's trade grid. It is advisable to be more liberal with the precision as the size of the ship increases, especially if the position might be around other objects.

Unlike the *escort ship* and *follow object* instructions, this one can be overridden by a “fly command” instruction in another process. However, like the *escort ship* and *follow object* instructions, if this is done then the *move to position* instruction will never return – it does not return a FLRET\_BREAK in that circumstance.

Returns FLRET\_NOCOMMAND when <ship> reaches its destination coordinates, FLRET\_INTERRUPTED if the script executing this instruction is interrupted, or FLRET\_INVALIDPARMS if <ship> is invalid.

**<object> -> set command <command>**

Set the apparent command that <object> is executing. This is for information purposes only – it does not actually cause <command> to execute on <object>. It causes the text associated with the specified command to appear on a ship's info screen beside the “Cmd” heading.

While the instruction ostensibly allows you to set the command on any object, it only actually works when <object> = [THIS] – that is, it only works when the instruction is executed by the object that it is trying to set the command on. Additionally, the ship's command setting resets once the script that executes this command terminates.

This instruction is useful if you are writing a script that has multiple functions. You can use *set command* to allow the user to see what the ship's current goal is.

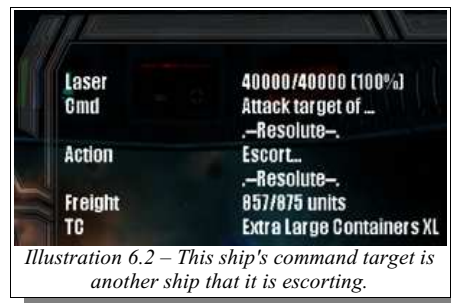
**[skip|else] if [not]|while [not]|<retvar>= <object> -> get command**

Returns the command set for an object. This can be either a command previously given an object through the command menus, or the command set on an object through the use of the *set command* instruction.

**<object> -> set command target <target>**

Many commands have a target they operate on. The obvious example is ordering a ship to attack something. This instruction sets the target of the current command. This is displayed on the a ship's info screen under the command.

This instruction has the same limitations as the *set command* instruction in that it must be executed by a script running on <object>.



**[skip|else] if [not]|while [not]|<retvar>= <object> -> get command target**

Returns the command target set for an object. This can be either a command target previously given an object through the command menus, or the command target set on an object through



T

H

M

T

H

R

M

A

T

the use of the *set command target* instruction.

**<object> -> set command target2 <target2>**

While many commands have a target they operate on, some commands also have a secondary target. For example, ordering a ship to buy a ware – you must tell the ship what ware, and where to buy it. In that case, the ware is the target, and the station to buy it from is the target2. This instruction allows you to set target2 on an object. This is displayed on a ship's info screen under the command and target.

This instruction has the same limitations as the *set command* instruction in that it must be run on <object>.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> get command target2**

Returns the command target2 set for an object. This can be either a command target2 previously given an object through the command menus, or the command target2 set on an object through the use of the *set command target2* instruction.

**<retvar>= <ship> -> select new formation leader by: ship class=[TRUE]| [FALSE] strength=[TRUE]| [FALSE] min.speed=[TRUE]| [FALSE]**

When the lead ship in a formation is destroyed, this instruction can be used to select a new leader – for example, in a handler for the *SIGNAL\_KILLED* signal. You can individually select whether ship class, strength, or speed are used as selection criteria for the new leader. This instruction does **not** actually give leadership to the new ship – use the *give formation leadership* instruction for that.

Returns the ship chosen as the new leader on success, or null on failure.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> has formation ships**

Returns [TRUE] if <ship> has ships following it, [FALSE] if not.

**<retvar>= <ship> -> give formation leadership to <newleader>**

This instruction allows you to explicitly set a new ship to take over leadership of a formation.

Returns [TRUE] on success, [FALSE] on failure.

**<ship> -> set tactical to <number>**

**<retvar>= <ship> -> get tactical**

Not used. Currently these instructions have no effect.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> is <target> in firing range of turret <turretnum>**

Checks to see if a given ship's turret is capable of firing on <target>. This can also be used to check to see if a ship's main laser can fire on a target by using a <turretnum> of zero. This instruction checks the range and arcs on the specified turret using the weapons it currently has loaded.

Returns [TRUE] if <target> is in the turret's arcs and range, [FALSE] if not.



# T H I S I S I S T R I B U T E

**<retvar>= <ship> -> find enemy in firing range of turret  
<turretnum>**

Searches for an enemy ship that is in the given turret's firing range and arcs. There is no guarantee that this enemy is the closest, strongest, etc – there is no prioritization of targets whatsoever.

Returns an enemy ship on success, null on failure.

**<object> -> set command: <command> target=<target>  
target2=<target2> par1=<par1> par2=<par2>**

A single instruction that performs the functionality of the *set command*, *set command target*, and *set command target2* instructions. The par1 and par2 settings are for extra parameters – this information does not seem to be given to the user on any info screen.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> fire  
missile <missile> on <target>**

Fires the specified missile (this should be the missile's ware) on <target>.

Returns [TRUE] if a missile was fired, [FALSE] if not.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get current  
missile**

Returns the ware of the currently selected missile for <ship>, or null if there is none or on error.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> find best  
missile for target <target>**

Returns what the game engine believes is the most appropriate missile to fire on <target> out of the missiles that <ship> currently has in stock.

**<retvar>= best missile type for target <target>**

Returns what the game engine believes is the most appropriate missile to fire on <target>.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> launch  
<dronecount> fight drones: protect me or attack  
target=<target>**

Launches the given number of fighter drones with orders to protect <ship>, or to attack the given target.

Returns [TRUE] if drones were able to launch, [FALSE] if not.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> is missile  
<missile> ready to fire**

Returns [TRUE] if <ship> is capable of firing <missile> and if it has them in stock, [FALSE] if not.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> should a  
missile be fired**

Returns true if, according to a ship's set missile fire probability, it should currently fire a missile, or [FALSE] if not. This instruction is largely ignored in favour of the *check*, *select* and



T

H

M

T

H

R

M

A

T

*fire missile* instruction.

**<retvar>= <ship> -> get missile fire probability**

Returns the probability of a missile launch in percent. This is the same probability that is set from a ship's command screen.

**<ship> -> set missile fire probability to <probability>**

Sets a ship's missile fire probability to <probability> percent.

**<retvar>= <ship> -> get missile fire time difference in seconds**

**<ship> -> set missile fire time difference in seconds to <time>**

Not used. Currently these instructions have no effect.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> fits laser  
<laser> into turret <turret>**

Returns [TRUE] if the ware <laser> is a laser that turret number <turret> on <ship> is capable of mounting, [FALSE] if not. Set <turret> to zero to check the main ship's gun.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get max  
number of lasers in turret <turret>**

Returns the number of guns that <turret> can hold. Set <turret> to zero to check the ship's main gun.

**<retvar>= <ship> -> get laser type in turret <turret> at slot  
<slot>**

Returns the ware of the gun currently fitted in the specified slot and turret, or null if there is no gun fitted.

**<retvar>= <ship> -> get number of turrets**

Returns the number of turrets on <ship>. The main gun is not counted. The only way to tell if a ship has a main gun is to use *get max number of lasers in turret 0*.

**<retvar>= get range of missile type <missile>**

Returns the range in metres for the specified missile.

**<retvar>= <object> -> find nearest missile aiming at me**

Returns the closest missile currently targeted on <ship>, or null if there is none. This is the **only** instruction capable of searching for a missile.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> decouple  
ships**

Causes a Khaak cluster to break apart. Returns [TRUE] if successful, [FALSE] if not.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> is  
decoupled ships leader**

Returns true if <ship> is the formation leader for a decoupled Khaak cluster (sometimes called



T

H

M

T

H

R

M

A

T

the “Khaak thinker”).

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get current galaxy flight timestep in ms**

The flight timestep is the number of milliseconds between updates in sectors that are not rendered – that is, all the other sectors except for where the player's ship is located (often referred to as “out-of-sector”). In sectors that are not being observed this is 30000. In sectors that are being observed, it is 5000. This instruction returns the current flight timestep for the sector that <ship> is in.

Note that while the value is only meaningful for out-of-sector ships, this instruction will return a value of 30000 for in-sector ships. To determine if a ship is currently an in-sector or out-of-sector ship, use the *get sector object ID* instruction.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> is landing**

Returns [TRUE] if <ship> is currently landing, [FALSE] if not. Landing means that it is on “final approach” to a station, carrier, or gate.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> is starting**

Returns [TRUE] if <ship> is launching from a station or carrier, [FALSE] if not.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> use jumpdrive: target=<target>**

Cause <ship> to use its jumpdrive (if available) to jump to <target>. The target must be a sector or a gate. If it is a sector, then <ship> will jump into the very centre of the sector.

Returns [TRUE] on success, [FALSE] if a jump was not possible.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> needed jumpdrive energy for jump to sector <sector>**

Returns the number of energy cells needed for a jump from the target ship's current sector to <sector>. By setting <sector> equal to the ship's current sector, you can determine the amount of energy needed per sector.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> check, select and fire missile on <target>**

Checks to see if a missile should be fired at the target, based on the ship's missile fire probability, and fires the missile if it should be. Essentially a combination of the *should a missile be fired* and *fire missile* instructions.

Returns [TRUE] if a missile was fired, [FALSE] if not.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> add big ship <bigship>**

Places the specified big ship (M6, M2, M1, or TL) in dock with <object>. Returns [TRUE] on success, [FALSE] on failure.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> has a free big ship dock slot**

Returns [TRUE] if the object is capable of docking a large ship and has a free slot, [FALSE] otherwise.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> can be controlled by race logic**

When a ship is created that doesn't belong to the player, the built-in game race logic for its owner will take it over and control it after a few moments. This instruction determines whether that internal race logic can do that for the given object, and whether or not it is enabled.

Returns [TRUE] if the game's race logic can take over the object, [FALSE] if not.

**<object> -> set race logic control enabled to [TRUE]|[FALSE]**

Allows a script to explicitly tell the game's race logic whether or not it is allowed to take control of <object>. This is useful for AL plugins that create ships owned by other races, but where the AL plugin keeps control of the ship.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> can execute StartAction**

An object's start action is the default command it has based only on its class. This is different from the race control logic which is based on the object's class and race. This instruction determines if the start action is enabled for the given object.

Returns [TRUE] if the start action is enabled, [FALSE] if not.

**<object> -> set StartAction enabled to [TRUE]|[FALSE]**

An object's start action is the default command it has based only on its class. This is different from the race control logic which is based on the object's class and race. This instruction allows a script to tell the game engine whether or not it is allowed to execute a ship's start action. See [Annex A.4](#) for a list of default start actions.

**<ship> -> set as player wingman: [TRUE]|[FALSE]**

Sets whether or not <ship> is a wingman for the player's ship.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> is player wingman**

Returns [TRUE] if <ship> is a wingman for the player's ship, [FALSE] if not.

## ***6.8 TRADE COMMANDS (FOR SHIPS)***

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> buy <count> units of <ware>**

Purchases up <count> units of <ware> from the station <ship> is docked at. If there is not enough room in the cargo hold for the entire purchase or if there aren't enough on the station, then as many as possible will be purchased. If <ship> is owned by the player, money is automatically deducted from the ship's home station if it has a home set, or from the player's account if there is no home set.





T

H

M

T

H

R

M

A

T

Returns the number of units actually purchased.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> buy <count> units of <ware> to a max price of <price> Cr**

Purchases purchase up to <count> units of <ware> from the station <ship> is docked at, but only if the ware's price is less than or equal to <price>. If there is not enough room in the cargo hold for the entire purchase or if there aren't enough on the station, then as many as possible will be purchased. If <ship> is owned by the player, money is automatically deducted from the ship's home station if it has a home set, or from the player's account if there is no home set.

Returns the number of units actually purchased.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> sell <count> units of <ware>**

Sells up to <count> units of <ware> to the station <ship> is docked at. If <count> is larger than the amount on hand, then the entire amount on hand will be sold. If there isn't enough room on the station, then as many as possible will be sold. If <ship> is owned by the player, money is automatically added to the ship's home station if it has a home set, or to the player's account if there is no home set.

Returns the number of units actually sold.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> load <count> units of <ware>**  
**[skip|else] if [not]|while [not]|<retvar>= <ship> -> unload <count> units of <ware>**

Attempts to loads or unload <count> units of <ware> from or to the station that <ship> is docked at. No money is added to or deducted for this transfer. The actual amount transferred can be less than requested, depending on the amount of room on the ship/station, and the actual number of units on hand to transfer.

Returns the number of units actually transferred.

**<retvar>= <ship> -> get max. ware transport class**

Returns the largest ware class that can be transported by <ship>.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get cargo bay size**

Returns the total cargo bay size for <ship>.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get free volume of cargo bay**

Returns the amount of free space in a ship's cargo bay.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get volume of ware <ware> in cargo bay**

Returns the total volume taken by <ware> in a ship's cargo bay.



T

H

M

T

H

R

M

A

T

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get true volume of ware <ware> in cargo bay**

Returns the total volume taken by <ware> in a ship's cargo bay, less any volume taken up by any of <ware> that are currently installed as weapons or shields.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get free volume of ware <ware> in cargo bay**

Returns a ship's free capacity for more <ware> in its cargo bay. If the ware takes up more than one unit of volume, then the returned number will be a multiple of that.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get total volume in cargo bay**

Returns the total used volume in a ship's cargo bay.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> can transport ware <ware>**

Returns [TRUE] if a ship can transport <ware>, [FALSE] if not.

**<ship> -> add default items to ship**

Adds to a ship what the game has pre-set as its default equipment. This includes shields, weapons, and munitions.

**<ship> -> switch laser in turret <turretid> gun <gunid> to <gun>**

Changes the weapon in the given slot of <turretid> to <gun>, assuming that the ship has one on hand. Use zero for <turretid> to change a ship's main weapons.

**<ship> -> set wanted ware count to <count>**

This instruction will set the quantity of a ware that is displayed on the info screen when that ship's command is set to COMMAND\_GET\_WARE or COMMAND\_GET\_WARE\_BEST.

**<ship> -> set wanted ware to <count>**

This instruction will set the ware that is displayed on the info screen as being the ware a ship is currently buying. The ship's command must first be set to either COMMAND\_GET\_WARE or COMMAND\_GET\_WARE\_BEST.

**<retvar>= <ship> -> get wanted ware count**

Returns the number set with a previous call to *set wanted ware count*.

**<retvar>= <ship> -> get wanted ware**

Returns the ware set with a previous call to *set wanted ware*.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> has illegal ware onboard: race=<race>**

Returns [TRUE] if <ship> has a ware onboard that <race> considers illegal, [FALSE] otherwise.



T

H

M

T

H

R

M

A

T

## ***6.9 TRADE COMMANDS (FOR STATIONS)***

```
[skip|else] if [not]|while [not]|<retvar>= <station> -> get best  
store amount of ware <ware>  
[skip|else] if [not]|while [not]|<retvar>= <station> -> get max  
store amount of ware <ware>
```

Returns the maximum amount of a ware that a dock (get best) or factory (get max) can hold.

```
[skip|else] if [not]|while [not]|<retvar>= <station> -> can buy  
ware <ware>  
[skip|else] if [not]|while [not]|<retvar>= <station> -> can sell  
ware <ware>
```

Returns [TRUE] if the station can buy or sell <ware>, [FALSE] if not.

```
[skip|else] if [not]|while [not]|<retvar>= <station> -> get  
price of ware <ware>
```

Returns the currently set price of <ware> on a station.

```
[skip|else] if [not]|while [not]|<retvar>= <station> -> get  
average price of ware <ware>
```

Returns the current game setting for a ware's average price.

```
<station> -> set price of ware <ware> to <price> Cr
```

Sets the price that a station trades a ware at to <price> credits.

```
[skip|else] if [not]|while [not]|<retvar>= <station> -> uses  
ware <ware> as primary resource  
[skip|else] if [not]|while [not]|<retvar>= <station> -> uses  
ware <ware> as secondary resource
```

Returns [TRUE] if <ware> is a primary or secondary resource on <station>, [FALSE] if not.

```
[skip|else] if [not]|while [not]|<retvar>= <station> -> trades  
with ware <ware>
```

Returns [TRUE] if <ware> is traded at a station, [FALSE] if not.

```
[skip|else] if [not]|while [not]|<retvar>= <station> -> get  
product ware
```

Returns the ware that a station produces as a product, or null if there is no product (in the case of trade and equipment docks and shipyards).

```
[skip|else] if [not]|while [not]|<retvar>= <station> -> get  
number of resources  
[skip|else] if [not]|while [not]|<retvar>= <station> -> get  
number of primary resources  
[skip|else] if [not]|while [not]|<retvar>= <station> -> get  
number of secondary resources
```

Returns the total number, the number of primary, or secondary resource wares on a station.



This returns just the count, not a list of the wares themselves.

```
[skip|else] if [not]|while [not]|<retvar>= <station> -> get max  
trade jumps
```

Returns the maximum number of jumps that a station's transport can travel to buy/sell wares.

```
[skip|else] if [not]|while [not]|<retvar>= <station> -> only  
player owned ships can trade with
```

Returns [TRUE] if only player ships can trade with the given station, [FALSE] if not.

```
[skip|else] if [not]|while [not]|<retvar>= <station> -> get  
tradeable ware array from station
```

Returns an array containing all wares that a station can trade. If the station has a product, it is always first, followed by primary, then secondary resources.

## ***6.10 TRADE COMMANDS (FOR SHIPS AND STATIONS)***

```
[skip|else] if [not]|while [not]|<retvar>= <object> -> get money
```

Returns the amount of money that a station or a ship's owning station (depending what <object> is) has on hand.

```
[skip|else] if [not]|while [not]|<retvar>= <object> -> add  
money: <amount>
```

Adds <amount> to a station or a ship's owning station cash.

```
[skip|else] if [not]|while [not]|<retvar>= <object> -> get  
amount of ware <ware> in cargo bay
```

Returns the amount (the actual quantity, not the volume) of a ware in an object's cargo bay.

```
[skip|else] if [not]|while [not]|<retvar>= <object> -> get true  
amount of ware <ware> in cargo bay
```

Returns the amount (the actual quantity, not the volume) of a ware in an object's cargo bay less any that are currently installed as shields or weapons.

```
[skip|else] if [not]|while [not]|<retvar>= <object> -> get free  
amount of ware <ware> in cargo bay
```

Returns the amount ( the actual quantity, not the volume) of a ware that can be added to an object's cargo bay given the amount of cargo already in the bay.

```
[skip|else] if [not]|while [not]|<retvar>= <object> -> get max  
amount of ware <ware> that can be stored in cargo bay
```

Returns the total amount of a ware that could be stored in the cargo bay. This is essentially adding the results of *get amount of ware in cargo bay* and *get free amount of ware in cargo bay*.



T

H

M

T

H

R

M

A

T

## 6.11 TRADE COMMANDS

**[skip|else] if [not]|while [not]|<retvar>= get player money**

Returns the amount of money the player has.

**add money to player: <amount>**

Adds <amount> to the player's money. Use a negative amount to subtract.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> add  
<amount> units of <ware>**

Adds the specified number of units of <ware> to the cargo hold on <object> as long as there is room enough to hold it.

Returns [TRUE] on success, [FALSE] on failure.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> install  
<amount> units of <ware>**

Installs the specified number of units of <ware> on <object>. For example, if <ware> is a shield, the shield will be installed in the object's shield bay. The ware is not added to the object's cargo bay. This instruction can be used to add engine tunings, cargo bay extensions, and rudder optimizations among other things.

Returns [TRUE] on success, [FALSE] on failure.

**<retvar>= get volume of ware <ware>**

Returns the amount of cargo space that one unit of <ware> occupies.

**<retvar>= get transport class of ware <ware>**

Returns the transport class of <ware>.

Tiny Containers
Small Containers S
Medium Containers M
Large Containers L
Extra Large Containers XL
Station Containers ST

Table 6.6 - Container  
Transport Classes

**[skip|else] if [not]|while [not]|<retvar>= find station: product  
<product> with best price: max.price=<price>,  
amount=<amount>, max.jumps=<jumps>, startsector=<sector>,  
trader=<trader>**

**... find station: product <product> with min. jumps ...  
... find station: resource <product> with best price ...  
... find station: resource <product> with min. jumps ...  
... find station sells: resource <product> with best price ...  
... find station sells: resource <product> with min. Jumps ...**

These instructions are used to find stations that buy or sell wares. The “best price” instructions prefer price over distance, the “min. jumps” instructions are the opposite. Finding a product looks for stations selling a ware it produces – best price in this case means lowest. Finding a resource looks for stations buying a ware – best price means highest here. Finally, “find stations sells” is used for trading docks, which are a station that sells its resources. In this case, “best price” also means lowest.

Returns the station on success, null on failure.



T

H

M

T

H

R

M

A

T

```
<retvar>= get average price of ware <ware>
<retvar>= get max. price of ware <ware>
<retvar>= get min. price of ware <ware>
<retvar>= get max. price of ware <ware> as secondary resource
<retvar>= get min. price of ware <ware> as secondary resource
```

Returns the average, minimum, or maximum price for <ware>. When a ware is a station's secondary resource, the maximum and minimum prices are different, so there are separate instructions for this case.

```
[skip|else] if [not]|while [not]|<retvar>= is ware <ware>
illegal in <race> sectors
```

Returns [TRUE] if the specified ware is illegal to transport in sectors owned by <race>, [FALSE] if not.

```
<retvar>= get maintype of ware <ware>
```

Each ware belongs to a main and sub-category called maintype and subtype. This instruction returns the specified ware's numerical maintype.

```
<retvar>= get subtype of ware <ware>
```

Each ware belongs to a main and sub-category called maintype and subtype. This instruction returns the specified ware's numerical subtype.

```
<retvar>= get ware from maintype <maintype> and subtype
<subtype>
```

Each ware belongs to a main and sub-category called maintype and subtype. This instruction takes a numerical maintype and subtype and returns the associated ware.

```
<retvar>= get number of subtypes of maintype <maintype>
```

Returns the number of subtypes in the given maintype ware category.

```
<retvar>= <object> -> get maintype
```

Every space object has an associated ware that could be considered to be the “type” of space object it is. Also, every ware has a numerical main and sub category number called maintype and subtype. This instruction looks at an object, and returns the numerical maintype of the ware associated with it.

```
<retvar>= <object> -> get subtype
```

Every space object has an associated ware that could be considered to be the “type” of space object it is. Also, every ware has a numerical main and sub category number called maintype and subtype. This instruction looks at an object, and returns the numerical subtype of the ware associated with it.

```
[skip|else] if [not]|while [not]|<retvar>= <object> -> find
station: product <product> with best price:
max.price=<price>, amount=<amount>, max.jumps=<jumps>,
startsector=<sector>, trader=<trader>
```

This series of commands is identical to the above *find station: product with best price/min.*





T

H

M

T

H

R

M

A

T

*jumps* except for this series allows you to specify an object to calculate max. jumps from. The previous series calculates max. jumps from the current location of <trader>.

## 6.12 GENERAL OBJECT COMMANDS

**[skip|else] if [not]|while [not]|<retvar>= <object> -> get  
object class**

Returns the class of an object. See [Annex A.5](#) for a breakdown of all the classes.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> get owner  
race**

Returns the race that owns <object>.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> get  
environment**

If <object> is docked then this instruction returns the station or carrier it is docked at, otherwise it returns the sector the object is in.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> get  
sector**

Returns the sector that <object> is located in. If <object> is docked, it returns the sector that its host is located in.

**[skip|else] if [not]|while [not]|<retvar>= <ship> -> get  
homebase**

Returns the home base of <ship>, or null if there is none set.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> is of  
class <class>**

Returns [TRUE] if object belongs to the <class> or any of its children. If the player's current ship is a Teladi Albatross, then both Test1 and Test2 will be true in the following example:

```
001 $Test1 = [PLAYERSHIP] -> is of class Ship
002 $Test2 = [PLAYERSHIP] -> is of class Big Ship
```

See [Annex A.5](#) for a breakdown of all the classes.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> exists**

Returns [TRUE] if <object> is a valid, existing in-game object. [FALSE] otherwise. This should be used often inside scripts. Remember, the universe is dynamic – things get destroyed all the time. Never assume that because at the beginning of your script a station existed that it still exists in the middle or at the end.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> get ware  
type code of object**

Every space object has an associated ware that can be said to reflect the type of object it is. This instruction returns the ware type associated with <object>.



**[skip|else] if [not]|while [not]|<retvar>= <station> -> get serial name of station**

The serial name of a station is the Greek letter associated with its name. Only non-player stations have them. This instruction returns the serial name of <station>.

**<station> -> set serial name of station <serial>**

Sets the serial name of <station>. This will have no effect on any player-owned station.

**[skip|else] if [not]|while [not]|<retvar>= find station in galaxy: startsector=<sector> class or type=<type> race=<race> flags=<flags> refobj=<object> serial=<serial> maxjumps=<maxjumps>**

Outside of trading scripts (where the *find station: product*, series of instructions are used most), this is one of the most commonly used search instruction. The parameters in general are used to restrict the scope of the search. Any of them, except for startsector and maxjumps can be null – which leaves that part of the search unrestricted.

The *class or type* parameter can be any object class that is a child of “station” (see [Annex A.5](#)), or it can be an explicit station type. The race parameter restricts the search to stations belonging to a particular race or to the player. Flags is a special setting which allows you to control the scope of the search in several ways. There are ten different flags that can be used individually, or ORed together to form multiple parameters (see [Annex A.6](#)). The refobj parameter is used to specify an object for use with the Find.Nearest flag. The serial parameter restricts the search to alpha, beta, etc. stations. The maximum number of jumps that a station can be from startsector is specified with the maxjumps parameter.

The requirement for the *find station in galaxy* to be given a valid start sector isn't as trivial as it might sound. Remember, there is no guarantee that a player is using the built-in map supplied with the game. This rules out using the *get sector from universe* instruction as that requires using a stated X and Y, and that sector might not exist for some player maps. A common method used is:

```
[PLAYERSHIP] -> get sector
```

This will succeed in getting the player's sector, but what if the player's ship doesn't exist yet (such as in the beginning of the game before Ban Danna gives Julian a ship)? Also, the player's ship might be in a Khaak sector. There are no jump gates out there, so using a Khaak sector as the source might not work – the *find station in galaxy* instruction will fail if you are trying to search for a random station that way. Keep these things in mind when you code so that your code works in every situation.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> is of type <type>**

Returns [TRUE] if <object> is of the given type (either station or ship type), [FALSE] otherwise.

**[skip|else] if [not]|while [not]|<retvar>= get jumps from sector <startsector> to sector <endsector>**

Returns the number of jumps to get from one sector to another.



# T H I S I S I S U E I S T

**[skip|else] if [not]|while [not]|<retvar>= get next sector on route from sector <currentsector> to <endsector>**

Returns the sector that is the next sector a ship needs to travel to in order to get to <endsector>. There is no capacity for alternate routes, and the instruction does not respect undiscovered sectors.

**<ship> -> set homebase to <newhome>**

Sets a ship's home base to a carrier or base.

**[skip|else] if [not]|while [not]|<retvar>= <object> get current shield strength**  
**[skip|else] if [not]|while [not]|<retvar>= <object> get maximum shield strength**

Returns the current/maximum shield strength of <object>. This is an integer expressed in kilowatts. The maximum is the megawatt value of all installed shields added together and multiplied by 1000.

**[skip|else] if [not]|while [not]|<retvar>= <ship> get current laser strength**  
**[skip|else] if [not]|while [not]|<retvar>= <ship> get maximum laser strength**

Each laser has a number representing the amount of energy it stores. Firing a shot from any turret reduces the energy which is then restored at a certain rate. This instruction returns the current/maximum laser strength of <object>. The maximum is the laser strength of all installed lasers added together.

**[skip|else] if [not]|while [not]|<retvar>= <ship> get max. laser strength in turret <turret>**

Returns the maximum laser strength for all the guns in one turret.

**[skip|else] if [not]|while [not]|<retvar>= <object> get max. shield type that can be installed**

Returns the best shield that can be installed on <object>.

**[skip|else] if [not]|while [not]|<retvar>= <object> get number of shield bays**

Returns the total number of shields that can be installed on <object>.

**[skip|else] if [not]|while [not]|<retvar>= <ship> get number of laser bays**

Returns the total number of lasers that can be installed on <ship>.

**[skip|else] if [not]|while [not]|<retvar>= <ship> get max. missile type that can be installed**

Returns the most powerful missile that <ship> can fire.



T

H

M

T

H

R

M

A

T

**[skip|else] if [not]|while [not]|<retvar>= <object> get relation to object <subject>**

Returns whether <subject> is friendly (owned by the same race), neutral (owned by a different race but not an enemy), or enemy of <object>.

**[skip|else] if [not]|while [not]|<retvar>= <object> get relation to race <race>**

Returns whether <race> is generally friendly, neutral, or enemies with the owner of <object>.

**[skip|else] if [not]|while [not]|<retvar>= <object> get notoriety to race <race>**

Intended to return the integer value that represents how friendly (positive numbers) or unfriendly (negative numbers) <race> is toward the owner of <object>. This instruction does not work as intended – currently it always returns zero.

**[skip|else] if [not]|while [not]|<retvar>= get notoriety from race <onrace> to race <anotherace>**

Intended to return the integer value that represents how friendly (positive numbers) or unfriendly (negative numbers) one race is toward another. This instruction does not work as intended – currently it always returns zero.

**[skip|else] if [not]|while [not]|<retvar>= <object> is <subject> a enemy**

**[skip|else] if [not]|while [not]|<retvar>= <object> is <subject> a friend**

**[skip|else] if [not]|while [not]|<retvar>= <object> is <subject> neutral to me**

Returns [TRUE] if the subject object is friendly/enemies/neutral toward <object>, [FALSE] otherwise.

**[skip|else] if [not]|while [not]|<retvar>= <object> get shield type in bay <bay>**

Returns the type of shield currently installed in an object's specified shield bay.

**[skip|else] if [not]|while [not]|<retvar>= <ship> get laser type in bay <bay>**

Returns the laser that is currently installed in the specified bay of an object's main weapons (turret zero).

**[skip|else] if [not]|while [not]|<retvar>= <object> has same environment as <subject>**

Returns [TRUE] if <subject> is in the same environment as <object>. This means both are either undocked in the same sector, or both are docked in the same station. Returns [FALSE] otherwise.



**[skip|else] if [not]|while [not]|<retvar>= <object> is in same sector as <subject>**

Returns [TRUE] if <subject> is in the same sector as <object>, [FALSE] otherwise.

**[skip|else] if [not]|while [not]|<retvar>= <ship> is landed**

Returns [TRUE] if <object> is a small ship that is landed inside a carrier or station, [FALSE] otherwise.

**[skip|else] if [not]|while [not]|<retvar>= <ship> is docked**

Returns [TRUE] if <object> is docked – this means either “landed inside” or “docked outside”.

**[skip|else] if [not]|while [not]|<retvar>= <object> is docking possible of <ship>**

Returns [TRUE] if <ship> is physically able to dock with <object>, [FALSE] if not.

**[skip|else] if [not]|while [not]|<retvar>= <ship> is docking allowed at <target>**

Returns [TRUE] if <target> will allow <ship> to dock with it. This doesn't mean that <target> is physically capable of holding the ship, or that it has enough room to hold it.

**[skip|else] if [not]|while [not]|<retvar>= <object> is in a sector**

Returns [TRUE] if <object> is not docked, [FALSE] if it is.

**[skip|else] if [not]|while [not]|<retvar>= <object> get attacker**

If <object> is being attacked, this instruction will return one of the attackers. Returns null otherwise.

**<object> -> set attacker to <attacker>**

Informs the game engine that <attacker> is to be considered as attacking <object>. This also occurs automatically when shots from an attacker hit an object.

**<retvar>= get distance between <oneobject> and <anotherobject>**

Returns the distance in metres between two objects, assuming they are in the same sector.

**<retvar>= <object> -> get distance to: x=<x> y=<y> z=<z>**

Returns the distance in metres between <object> and the specified coordinates.

**<retvar>= <object> -> get distance to: position array=<position>**

Returns the distance in metres between <object> and the coordinates stored in the array <position>. The array should have three integer members in the order X, Y, Z.

**<retvar>= <object> -> get distance: position array1=<position1> position array2=<position2>**

Returns the distance in metres between positions represented by two arrays. Each array should have three integer members in the order X, Y, Z.



# THE MATHS

```
<retvar>= create ship: type=<type> owner=<owner> addto=<target>  
          x=<x> y=<y> z=<z>
```

Creates a new ship in <target>. The target can be a sector, station, or carrier. If adding to a station or carrier, the coordinates are ignored. Never add a big ship to a station this way – it will end up landed internally inside the station.

Returns a “pointer” (a value that can be used in other <object> -> style instructions) to the newly created ship on success, null on failure.

```
<retvar>= <object> -> get x position  
<retvar>= <object> -> get y position  
<retvar>= <object> -> get z position
```

Returns the X, Y, or Z coordinate of <object>.

```
<retvar>= <object> -> get position as array
```

Returns the position of <object> as an array of three integers in the order X, Y, Z.

```
<retvar>= get player ship
```

Returns the current player ship, or null if there isn't one (as there isn't at the beginning of a standard game). This isn't used very often as it is generally more convenient to use the symbol [PLAYERSHIP].

```
<object> -> set relation against <target> to <relation>
```

Changes the relation of <object> toward <target>. This may have to be done in both directions.

```
<retvar>= create station: type=<type> owner=<owner>  
          addto=<sector> x=<x> y=<y> z=<z>
```

Creates a new station and adds it to <sector> at the given coordinates. This does not add any products or resources to the station – that has to be performed separately.

Returns a “pointer” (a value that can be used in other <object> -> style instructions) to the newly created station on success, null on failure.

```
<retvar>= create gate: type=<type> addto=<sector> gateid=<id>  
          dstsecx=<sectorx> dstsecy=<sectory> dstgateid=<linkto> x=<x>  
          y=<y> z=<z>
```

Creates a gate and links it to a gate in another sector. The type represents whether it leads North (0), South (1), West (2), or East (3). The gateid is an integer that simply uniquely identifies the gate within a sector – each gate in a sector must have a different one. It is this id number that links gates together. To determine the target sector, specify its X and Y coordinates on the map in dstsecx and dstsecy. Which gate within that sector the new gate will deposit a ship at is specified with dstgateid. Generally for all built-in gates, the type and gateid are the same. Finally, the location coordinates of where the gate is within its sector are specified in X, Y, Z format.

Returns a “pointer” (a value that can be used in other <object> -> style instructions) to the newly created gate on success, null on failure.





**<retvar>= create asteroid: type=<type> addto=<sector>  
resource=<resource> yield=<yield> x=<x> y=<y> z=<z>**

Creates a new asteroid at a given set of coordinates within a sector. Type specifies the style of asteroid (see [Annex A.7](#)). The resource specifies whether the asteroid contains ore (0), silicon (1), or nividium (2).

**<station> -> add product to factory or dock: <ware>  
<station> -> add primary resource factory or dock: <ware>  
<station> -> add secondary resource factory or dock: <ware>  
<station> -> remove product from factory or dock: <ware>  
<station> -> remove primary resource from factory or dock:  
    <ware>  
<station> -> remove secondary resource from factory or dock:  
    <ware>**

Adds or removes <ware> as a product/primary/secondary resource of <station>. No built-in stations have multiple products, however the game supports having them. No player-owned factory uses secondary resources but, again, the game supports it.

**<retvar>= create nebula: type=<type> addto=<sector> x=<x> y=<y>  
z=<z>**

Creates a nebula and adds it to <sector>. There are 13 different types of nebula that can be produced with this instruction. See [Annex A.8](#) for a list.

**<retvar>= create sun: subtype=<subtype>  
r=<red> g=<green> b=<blue> addto=<sector> x=<x>  
y=<y> z=<z>**

**WARNING:**  
Creating suns of subtype 23 has been known to cause crashes-to-desktop. Avoid this subtype.

Creates a sun and adds it to <sector>. Suns are unique in that they do not appear to get smaller or larger depending on how far your ship is from them. Thus, the placement in the sector can be anywhere to give the best lighting effect without having to worry about how large it appears.

The r, g, and b parameters specify the amount of red, green and blue light the sun gives off. This does not control the colour of the sun itself, just the light it adds to the sector.

There are 25 subtypes in all – see [Annex A.9](#) for a list.

**<retvar>= create planet: subtype=<subtype> addto=<sector> x=<x>  
y=<y> z=<z>**

Creates a planet and adds it to <sector>. In this case, planet means “a solar body that has a spherical 3d model”, since two planets are actually used as suns.

There are 17 subtypes – see [Annex A.10](#) for a list.

**<retvar>= create special: type=<type>  
addto=<sector> x=<x> y=<y> z=<z>**

**WARNING:**  
Using a type > 73 has been known to cause crashes-to-desktop.

This instruction can produce a whole melting pot of different items. From a working gravidar hanging in space, to the wreckage of stations and ships.

There are 74 types that can be created. See [Annex A.11](#) for a list.



# T H I S I S I S T E R Y

```
[skip|else] if [not]|while [not]|<retvar>= find ship:  
    sector=<sector> class or type=<classtype> race=<race>  
    flags=<flags> refobj=<refobj> maxdist=<maxdist>  
    maxnum=<maxnum> refpos=<position>
```

One of the five general purpose search instructions, this one is used to find ships in the given sector.

Set <classtype> equal to any class that is a child of Ship (see [Annex A.5](#)), a specific ship type, or null to have no restriction based on the class or type of ship. Set <race> to restrict the search to ships owned by that race, or to null for no restriction. The flags parameter is a special setting which allows you to control the scope of the search in several ways. There are ten different flags that can be used individually, or ORed together to form multiple parameters (see [Annex A.6](#)). Set <refobj> to be a space object if you wish to search for ships based on their proximity to something. Use the maxdist parameter to specify how far from <refobj> or <refpos> the searched-for ships can be. If you use the [Find.Multiple] flag, then <maxnum> is the maximum number of ships that the instruction will return in its return array. The refpos parameter is used in conjunction with maxdist to search for ships that are within a certain distance of a point in the sector. If it is set, it needs to be a position array – an array of three integers in the order X, Y, Z. This instruction should not set both <refobj> and <refpos>.

Returns either a single ship or an array of ships on success, null on failure.

```
[skip|else] if [not]|while [not]|<retvar>= find asteroid:  
    sector=<sector> resourcetype=<resource> min. yield=<yield>  
    flags=<flags> refobj=<refobj> maxdist=<maxdist>  
    maxnum=<maxnum> refpos=<position>
```

One of the five general purpose search instruction, this instruction finds asteroids within a sector.

Set <resource> to 0 to search for asteroids with ore, 1 for silicon, 2 for nividium, or null for no restriction on the resource. Set <yield> to the minimum yield of an asteroid that the search can return, or null for no restriction. The flags parameter is a special setting which allows you to control the scope of the search in several ways. There are ten different flags that can be used individually, or ORed together to form multiple parameters (see [Annex A.6](#)). Set <refobj> to be a space object if you wish to search for asteroids based on their proximity to something. Use the maxdist parameter to specify how far from <refobj> or <refpos> the searched-for asteroids can be. If you use the [Find.Multiple] flag, then <maxnum> is the maximum number of asteroids that the instruction will return in its return array. The refpos parameter is used in conjunction with maxdist to search for asteroids that are within a certain distance of a point in the sector. If it is set, it needs to be a position array – an array of three integers in the order X, Y, Z. This instruction should not set both <refobj> and <refpos>.

Returns either a single asteroid or an array of them on success, null on failure.

```
[skip|else] if [not]|while [not]|<retvar>= find flying ware:  
    sector=<sector> maintype=<maintype> subtype=<subtype>  
    flags=<flags> refobj=<refobj> maxdist=<maxdist>  
    maxnum=<maxnum> refpos=<position>
```

One of the five general purpose search instruction, this instruction finds flying wares within a sector. A flying ware is a ware that has been ejected into space, or left behind when a ship is destroyed.

All the game wares are organized into several groups called maintypes and their position within



T

H

M

T

H

R

M

A

T

that group is their subtype. Set either or both the `<maintype>` and `<subtype>` parameters to restrict the search accordingly<sup>13</sup>. The `flags` parameter is a special setting which allows you to control the scope of the search in several ways. There are ten different flags that can be used individually, or ORed together to form multiple parameters (see [Annex A.6](#)). Set `<refobj>` to be a space object if you wish to search for flying wares based on their proximity to something. Use the `maxdist` parameter to specify how far from `<refobj>` or `<refpos>` the searched-for flying wares can be. If you use the `[Find.Multiple]` flag, then `<maxnum>` is the maximum number of flying wares that the instruction will return in its return array. The `refpos` parameter is used in conjunction with `maxdist` to search for flying wares that are within a certain distance of a point in the sector. If it is set, it needs to be a position array – an array of three integers in the order X, Y, Z. This instruction should not set both `<refobj>` and `<refpos>`.

```
[skip|else] if [not]|while [not]|<retvar>= find station:  
  sector=<sector> class or type=<classtype> race=<race>  
  flags=<flags> refobj=<refobj> maxdist=<maxdist>  
  maxnum=<maxnum> refpos=<position>
```

One of the five general purpose search instructions, this one is used to find stations in the given sector. This is different from the *find station in galaxy* instruction, which is able to find a station across multiple sectors. Also, the *find station in galaxy* instruction cannot use the `[Find.Multiple]` flag, whereas this instruction can.

Set `<classtype>` equal to any class that is a child of Station (see [Annex A.5](#)), a specific station type, or null to have no restriction based on the class or type of station. Set `<race>` to restrict the search to stations owned by that race, or to null for no restriction. The `flags` parameter is a special setting which allows you to control the scope of the search in several ways. There are ten different flags that can be used individually, or ORed together to form multiple parameters (see [Annex A.6](#)). Set `<refobj>` to be a space object if you wish to search for ships based on their proximity to something. Use the `maxdist` parameter to specify how far from `<refobj>` or `<refpos>` the searched-for ships can be. If you use the `[Find.Multiple]` flag, then `<maxnum>` is the maximum number of stations that the instruction will return in its return array. The `refpos` parameter is used in conjunction with `maxdist` to search for stations that are within a certain distance of a point in the sector. If it is set, it needs to be a position array – an array of three integers in the order X, Y, Z. This instruction should not set both `<refobj>` and `<refpos>`.

Returns either a single station or an array of stations on success, null on failure.

```
[skip|else] if [not]|while [not]|<retvar>= <object> -> is  
  disabled
```

At one point there was a plan to incorporate missiles that disabled a ship. Currently this instruction is not used.

```
<object> -> station trade and production tasks: on=[TRUE]|  
  [FALSE]
```

Stations that are created using the *create station* instruction are created “turned off”. This is to give the script time to set the station's product and resources. This instruction can then be used to turn on the station.

<sup>13</sup> A comprehensive list of all wares along with their main and sub types is beyond the scope of this manual. However, it is not difficult to get this information as needed with a little script experimenting.



**[skip|else] if [not]|while [not]|<retvar>= <object> -> get SectorObject ID**

Returns the Sector Object ID of the given space object. This instruction can be used in a script to determine if the sector a ship is in is currently “rendered” (the sector where the player ship is located), since only “in-sector” space objects have a SectorObject ID.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> get ware type of SectorObject <sectorobject>**

Returns the ware type of the object linked to the given SectorObject.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> exists SectorObject <sectorobject>**

Returns [TRUE] if the given SectorObject exists, [FALSE] otherwise.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> get object from SectorObject <sectorobject>**

Returns the normal “pointer” to an object based on its SectorObjectID on success, null on failure.

**<object> -> destruct: show no explosion=[TRUE]| [FALSE]**

Destroys <object>, either with an explosion or silently.

**<object> -> set position: x=<x> y=<y> z=<z>**

Moves <object> to the position specified by the given coordinates. No checking is done on whether or not something is already in that location.

**<object> -> set rotation: alpha=<alpha> beta=<beta> gamma=<gamma>**

Sets the direction an object is pointing. Alpha is the heading, beta is the elevation, and gamma is its rotation around its Z axis. All angles are expressed as sixteen bit integers with 65536 “degrees” in a circle.

**set position of sector object <sectorobject>: x=<x> y=<y> z=<z>**

Identical to *set position*, except that this instruction operations on a SectorObject.

**set safe position of sector object <sectorobject>: x=<x> y=<y> z=<z>**

Whereas *set position* does no checking on whether something is already sitting at the target location, this instruction ensures that the object is moved safely. If the target location has something there already, then the game engine will shift the actual location where the object is

#### Technical Tidbit

As most everyone knows, only the sector that a player is currently in is actually “rendered”. Every object in a sector that is rendered, including the sector itself, has what is called a “SectorObject”. This is, simply, the visual part of an object – it’s model.

When a sector is rendered, each object in the sector is given a “SectorObject” in turn, starting with the sector itself which is usually followed by the sector’s sun, nebulae, then stations, ships, and asteroids. The “Sector Object ID” is an integer that starts at one for the first object rendered in the first sector entered after a new game is started or loaded, and increases by one for every new one created. When a ship enters the rendered sector, a new SectorObject with a unique ID is created. When it leaves, it is destroyed. Objects which normally can’t be searched for, like suns and nebulae, with some work can be found by going through all the SectorObject IDs in a sector. It’s tricky, but possible.



moved to.

**set rotation of sector object** <sectorobject>: **alpha**=<alpha>  
**beta**=<beta> **gamma**=<gamma>

Identical to *set rotation*, except that this instruction operations on a SectorObject.

Using Sector Objects is dangerous, and has almost no benefit. It is a very good way to never get your scripts signed, and also to damage your save game.

**<retvar>= create sector object:**  
**maintype**=<maintype> **subtype**=<subtype>

Allocates a new SectorObject. The model is loaded, but it is not displayed in a sector until *start sector object* is called. Be advised that creating and freeing sector objects is playing around with the game engine at a very low level. It is much more advisable to *create ship* and *destruct* directly.

Returns the ID of the new SectorObject on success, null on failure.

**free sector object** <sectorobject>

Deallocates <sectorobject>. Do **not** do this unless your script has allocated the sector object itself in the first place, or you will remove the model from the sector, but the game engine will think the actual space object is still in the sector. Be advised that creating and freeing sector objects is playing around with the game engine at a very low level. It is much more advisable to *create ship* and *destruct* directly.

**kill sector object** <victim>: **reason** <reason>,  
**killer sector object** <killer>

Informs the game engine that one sector object has killed another for the specified reason. Be advised that dealing directly with sector objects is playing around with the game engine at a very low level. It is much more advisable to use *destruct* directly.

Shot by laser	2
Collision	3
Hit by missile	5

Table 6.7 - Kill Reasons

**start sector object** <sectorobject> **in space** <sectorid>

Places a previously allocated sector object in space. In this instruction, <sector> is the SectorObject ID of the sector. Be advised that creating and freeing sector objects is playing around with the game engine at a very low level. It is much more advisable to *create ship* and *destruct* directly.

**<retvar>= create flying ware:** **maintype**=<maintype>  
**subtype**=<subtype> **count**=<count> **sector**=<sector> **x**=<x> **y**=<y>  
**z**=<z>

Creates a new flying ware (see *find flying ware* for more information) of the given main and sub type and adds it to <sector> at the given coordinates.

Returns a "pointer" (a value that can be used in other <object> -> style instructions) to the newly created flying ware on success, null on failure.

**<retvar>= <object> -> get rot alpha**  
**<retvar>= <object> -> get rot beta**  
**<retvar>= <object> -> get rot gamma**

Returns an object's current orientation. Alpha is the heading, beta the elevating, and gamma the rotation around the Z axis. All angles are expressed as sixteen bit integers with 65536



T

H

M

T

H

R

M

A

T

“degrees” in a circle.

**<retvar>= <object> -> get size of object**

Returns the size of an object. This is the radius of the object's bounding sphere (a sphere that would encompass the object) multiplied by 222.

**<retvar>= <object> -> get max upgrades for upgrade <ware>**

Returns the maximum number of upgrades of the given ware that can be installed on <object>.

**<retvar>= <object> -> get max speed**

Returns the maximum speed of an object in metres per second.

**<retvar>= <object> -> get max hull**

Returns the maximum hull strength for <object>.

**<retvar>= <object> -> get hull**

Returns the current hull strength for <object> taking into account any damage it has suffered.

**<retvar>= <object> -> get hull percent**

Returns the current strength of an object's hull expressed as a percentage of its maximum strength.

**<retvar>= <object> -> get shield percent**

Returns the current strength of an object's shields expressed as a percentage of its maximum strength.

**<retvar>= <object> -> get shield and hull percent**

Returns as a two-entry array the current strength of an object's shields and hull expressed as a percentage of its maximum strength.

**<retvar>= <ship> -> get max upgraded speed**

Returns the maximum speed that a ship can travel at if it has full engine tunings installed.

**<retvar>= <object> -> get dock bay size**

Returns the maximum number of ships that can land on <object>. For a station this is one million.

**<retvar>= <object> -> get number of landed ships**

Returns the number of ships landed on <object>.

**player loses police license for race <race>**

Causes the player to lose the police license for the given race.

**<race> -> add notoriety: race=<subject> value=<value>**

Adds the given amount of notoriety toward <subject> race to <race>.





# T H I S I S I S T R I B U T E

**<object> -> set ship disabled to [TRUE]|[FALSE]**

Not used.

**<object> -> put into environment <environment>**

Moves <object> into the specified environment. This can be a sector or a base. This is useful for moving a ship to a base without the fuss of having it dock. Don't try this with a big ship.

**<station> -> station send defend squad against ship <target>**

Causes <station> to launch fighters with orders to attack <target>.

**<retvar>= <object> -> get name**

Returns the name of <object>.

**<retvar>= <object> -> set name to <name>**

Changes the name of <object> to <name>.

**<retvar>= <object> -> set owner race to <race>**

Changes the ownership of object so that it belongs to <race>.

**[skip|else] if [not]|while [not]|<retvar>= <object> -> find ship: sector=<sector> class or type=<classtype> race=<race> flags=<flags> refobj=<refobj> maxdist=<maxdist> maxnum=<maxnum> with homebase=<homebase>**

A second variant of *find ship* that allows a ship's home base to be used as a search criteria.

**[skip|else] if [not]|while [not]|<retvar>= find station in galaxy: startsector=<sector> class or type=<type> race=<race> flags=<flags> refobj=<object> serial=<serial> maxjumps=<maxjumps> num=<count>**

A second variant of *find station in galaxy* that will return multiple stations.

**<retvar>= <object> -> get id code**

Returns the ID code of the given object (YM2XL-21, for example, is a possible ID code for a player-owned M2).

**<retvar>= <ship> -> get pilot name**

Returns the name of the given ship's pilot. For player ships, this will most frequently be the player's currently set name.

**<retvar>= <ship> -> set pilot name to <name>**

Changes the name of the given ship's pilot. This will not change the player's name if used on a player-owned ship.

**<ship> -> set pilot speaker: voice=<voice>, face=<face>, Pirate subrace=[TRUE]|[FALSE], Argon female=[TRUE]|[FALSE]**

There are five different voices and faces for each race. This instruction allows a script to set



T

H

M

T

H

R

M

A

T

the specific voice and face associated with a ship's pilot. Both <voice> and <face> are integers.

## ***6.13 UNIVERSE AND SECTOR COMMANDS***

**[skip|else] if [not]|while [not]|<retvar>= get sector from  
universe index: x=<x>, y=<y>**

Returns the sector in the universe found at the grid location specified. Kingdom End is 0,0. Portable scripts shouldn't depend on a particular sector being at the location it is on the built-in game map. Different maps can be used.

**<retvar>= get max sectors in x direction  
<retvar>= get max sectors in y direction**

Gets the X or Y size of the map, often as a prelude to iterating through all the sectors.

**[skip|else] if [not]|while [not]|<retvar>= <sector> -> is sector  
known by the player**

Returns [TRUE] if the player has discovered <sector>, [FALSE] otherwise.

**<retvar>= <sector> -> get universe x index  
<retvar>= <sector> -> get universe y index**

Returns the X or Y index of the given sector.

**<retvar>= <sector> -> get warp gate: gate id=<gate>**

Finds and returns the warp gate of a given ID in a sector. For all warp gates in the default map, the ID corresponds with the direction the gate "travels". North is 0, South is 1, East is 2, West is 3.

**<retvar>= <sector> -> get north warp gate  
<retvar>= <sector> -> get south warp gate  
<retvar>= <sector> -> get east warp gate  
<retvar>= <sector> -> get west warp gate**

Returns the warp gate from <sector> that goes in the given direction, or null if there isn't one.

**<retvar>= find random sector: startsector=<sector>,  
jumps=<maxjumps>, owner=<owner>**

Returns a sector at random from the map that is within <maxjumps> of <sector>. Owner can be null, but the startsector parameter must be a valid sector.

**<retvar>= <object> -> get ship array from sector/ship/station**

Returns an array of all ships that are in the target ship/station/sector.

**<retvar>= <sector> -> get station array from sector  
<retvar>= <sector> -> get factory array from sector  
<retvar>= <sector> -> get dock array from sector**

Returns all stations/factories/docks in the target sector. A station can be any base, factory is any factory or shipyard, dock is any dock.



T

H

M

T

H

R

M

A

T

## The X<sup>2</sup> MSCI Programmer's Handbook

`<retvar>= <sector> -> get player owned ship array from sector`  
`<retvar>= <sector> -> get player owned station array from sector`

Returns all player-owned ships or stations in <sector>

`<retvar>= <sector> -> get asteroid array from sector`

Returns all asteroids located in <sector>.



T

H

M

T

H

M

M

A



T

## 7. ADVANCED TOPICS

### 7.1 PROCESSES AND TASKS

The purpose of writing scripts is to add functionality to the game. This would be pretty difficult if only one script could run at a time. After all, the whole idea is to have scripts doing all sorts of interesting things all at the same time – fighting, monitoring, trading, performing missions. With many hundreds of ships running thousands of scripts, this can't be a haphazard affair. There needs to be organization. The way X<sup>2</sup> organizes them is into separate processes and tasks.

Every script that runs, runs inside its own “virtual” computer called a process. Every process that runs is assigned “process ID” – a number that is different from any other process running. The process ID starts at zero for the very first process that ever ran in your current X<sup>2</sup> game, and increments by one for every new process. The maximum is 4294967296, after which it wraps back to zero (though unless you play one game for several years, this is unlikely to happen).

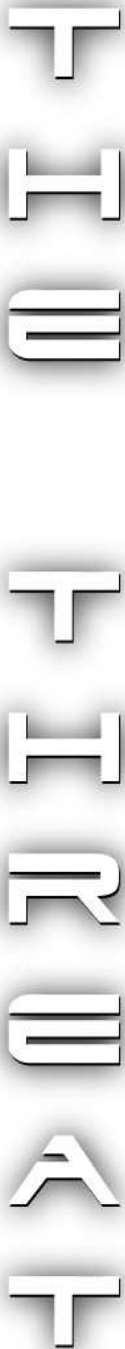
If you go into the scripting command screen ( then ), you will see an entry at the bottom of the menu labelled “Global Script Tasks” (this probably should have been labelled “Global Script Processes”). If you select this, you will see a list of all the currently running global processes – processes that aren't attached to any particular ship, station, or object – along with their associated process IDs.

Process IDs are great for the game to use for organizing all the thousands of scripts that get run, but they are ungainly for humans to use in scripts. In order to simplify things for people writing scripts, there is also the concept of a “task”.

A task is simply a process that is attached to an object. Different tasks are used to make an object do more than one thing at once. For example, the *Argon One* has a task that runs the main patrol script, and another task that runs scripts that control each of its turrets.

To further simplify matters, tasks were organized into the role they serve. The following chart lists all the tasks with predefined roles:

Task #	Task Role (ships)	Task Role (stations)
0	Main task – the ship's overall task	Main task – usually “idle”
1	Task for turret 1	
2	Task for turret 2	
3	Task for turret 3	
4	Task for turret 4	
5	Task for turret 5	
6	Task for turret 6	
10	First “Additional Ship Commands” task	Station Commands slot 1 task
11	Second “Additional Ship Commands” task	Station Commands slot 2 task
12		Station Commands slot 3 task
13		Station Commands slot 4 task
14		Station Commands slot 5 task
15		Station Commands slot 6 task

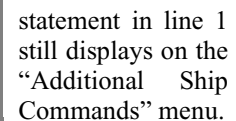


Any time you issue an order to a ship through the “Navigation”, “Combat”, “Trade”, or “Special” menus, that command starts a process and attaches that process to task zero on the ship you give the order for. Any time you give an order to turret 3, for example, the process that runs that script is attached to task 3 on the target ship.

TestCommand:

TestTask:

TestCommand, when run, starts up task number 10 on the current player ship. When TestTask is started, even though it isn't started through the command menus, the result of the *set script* command



pg. 79



## 7.2 CONCURRENCY, INTERRUPTS, AND ATOMIC OPERATIONS

### Concurrency

As many script writers find out early in their career, one of the easiest ways to lock up X<sup>2</sup> is by making an infinite loop. The following is a very simple script that, if run, will cause your game to hang:

```
TestLockup:  
001 while [TRUE]  
002 end
```

A very simple script – not just as deadly as it is simple.

The way that thousands of scripts all run at the same time is called “concurrency” - multiple scripts running concurrently with each other. To a player it might seem (and hopefully it *does* seem) that they are truly all running at the same time. This isn't the case – and the answer to why the above script locks up X<sup>2</sup> is the answer to how X<sup>2</sup> achieves concurrency.

Because all the ships in the game are simulated, and the shipboard computers are simply pretend computers, all the thousands of scripts that run in the game are actually run on one computer – yours. There is only one computer for all those scripts, so how does X<sup>2</sup> run them all at the same time? The answer is that it doesn't. It runs a little piece of script from one process, then switches to another process, and another, and so on in a round robin scheme. This is very similar to the way that your operating system can seem to run more than one program at once. This is important to remember – in X<sup>2</sup> there is only **ever** one script actually running at a time.

X<sup>2</sup> uses a type of process switching called “cooperative multitasking”<sup>14</sup>. Under this scheme, a script that is running stays running until it voluntarily cedes control to another script. This is done at special instructions that act as interrupt points. Script instructions which are interrupt points are marked by the “@” symbol in front of them. The most common ones are *wait* and *call script*. The scheme is called cooperative multitasking because all the scripts must cooperate in order to run effectively. If any one script refuses to give up control, then no other script can run.

You will of course notice that there are no interrupt points in the TestLockup script. This is why the script will cause the game to freeze – no other script can possibly run because our little script simply isn't cooperating. The script engine itself is so busy running that tiny little loop, that nothing else in the game can occur. Hence the freeze. Just one simple change to the script would keep it from locking up the game:

```
TestLockupFix:  
001 while [TRUE]  
003 @ wait 10 ms  
002 end
```

With the inclusion of an interrupt point, the little script will now no longer cause the game to freeze, though it will still run forever (so don't save your game if you actually try it).

This is very important to remember – once your script is running, it will stay running at least until it reaches an interrupt point. Without an interrupt point, X<sup>2</sup>'s multitasking system cannot switch processes to give run time to any other script.

14 See: [http://en.wikipedia.org/wiki/Co-operative\\_multitasking](http://en.wikipedia.org/wiki/Co-operative_multitasking)





## Interrupts

Something else that can happen at an interrupt point besides multitasking is an actual interrupt. An interrupt occurs either when some script uses an *interrupt* instruction, or when a signal (more on this in a bit) is received by the object your script is running on.

An interrupt is the way something else – either the game engine or another script – gives the object your script is running on a new command without replacing the old command. A common example of this is when you have several ships following you, and you use the command menu to tell your ship to jump to a new sector. When that happens, the ships that are following you jump with you (if they have jump drives and enough energy cells). Those ships could have been following you for one of several reasons. For example, they could have been given a “follow” command, or a “protect me” command. When you execute the command “jump to sector” and that command tells the ships following you to jump too, you don't want those ships to forget what they were doing when they arrive in the new sector. You want them to keep following you or protecting you. Here is the code from “!move.jump.xml” (the script that actually handles the “jump to sector” command) that tells your followers to jump too:

```
017 if $withfollowers
018   $afollowers = [THIS] -> get formation follower ships
019   if $afollowers
020     $s = size of array $afollowers
021     while $s > 0
022       dec $s=
023       $ship = $afollowers[$s]
024       $ship -> interrupt with script '!ship.signal.leaderjumps' and
         prio 0: arg1=[THIS] arg2=$target arg3=$withfollowers arg4=null
025     end
026   end
027 end
```

Line 24 is what we are interested in. This is how the ships that are following you jump with you but don't forget what they were doing before. Whatever commands they are running are interrupted with a new script.

When the following ships' scripts reach an interrupt point, it is like it is forced to execute a *call script* instruction. The new script runs as if it were called by the old script. The new script performs its function, and when it executes a *return* instruction, the old script resumes running where it left off. It doesn't even know it was interrupted.

## Signals

As mentioned earlier, another way to interrupt a ship is with a signal. A signal is a predefined interrupt that occurs when certain game conditions are met, the most common one being when a ship is attacked. There are actually seven different types of signals, and each of them has a defined script that is associated with them:



Signal Name	Occurs when...	Associated Script <sup>15</sup>
SIGNAL_ATTACKED	A ship is attacked	!ship.signal.attacked
SIGNAL_COLLISIONWARN	A ship is in danger of colliding with something.	No default handler script
SIGNAL_LEADERNEEDSHELP	A follower is in a formation and the leader is attacked	!ship.signal.leaderneedshelp
SIGNAL_FOLLOWERNEEDSHELP	The leader is in a formation and a follower is attacked	!ship.signal.followerneedshelp
SIGNAL_FORMATIONLEADERCHANGED	The leader of a formation is changed (because the old leader was killed, for example) this signal is sent to the rest of the ships in that formation.	!ship.signal.formationleaderchg
SIGNAL_CAPTURED	A ship is captured	!ship.signal.captures
SIGNAL_KILLED	A ship is killed	!ship.signal.killed

Table 7.2 – Signals and their handler scripts

So, what exactly must occur to trigger a signal script to run? First, the signal's handler script is associated with a signal. For example (from the init script !init.ship.globalscriptmap.std.xml):

```
072 global script map: set: key=SIGNAL_ATTACKED, class=Ship,  
    race=$race, script='!ship.signal.attacked', prio=100
```

Once the signal handler script is associated with a signal, then all that needs to happen is the signal's condition needs to be met. In the case of the above example, when a ship is attacked. As an example, imagine flying around in an Argon Buster and chancing upon a Pirate Ship. That pirate ship is most likely running a “buy ware for best price” command for the pirate base that owns it. Assuming you're not a pirate sympathizer, you fire on it.

The moment that your first shot hits the pirate ship, the game recognizes that a signal condition has been met. Whatever script the pirate ship was running is now automatically sent an interrupt. When that script reached an interrupt point, as before it is as if it is forced to execute a *call script* instruction and the signal's handler (the !ship.signal.attacked) script now begins to run.

Now, suppose the pirate launches a Hornet at you and you, deciding that discretion is the better part of valour, bug out. Eventually the pirate will go back to what it was doing before. Just like with the interrupt, when the combat script terminates, it returns control back to whatever was running before the signal was sent.

As described above, every time your script reaches an instruction that is an interrupt point, the scripting engine checks for two conditions. Is there another script that is trying to interrupt this one (through an interrupt or signal), and is there another script that is waiting to run.

Remember, there are thousands of scripts and only one is ever actually “running” at any one time. Even if there isn't another script that is trying to interrupt yours, you can be sure that there is probably another script that is waiting to run.

There are times, however, when you need to make sure another script doesn't run. At least not yet.

## Atomic Operations

Consider the following scenario. You are writing an advanced script to run your turrets and the

<sup>15</sup> The most common associated script is listed. Different classes of ships can (and do) have their own different scripts attached as the signal handler for a signal.



# T H I S I S T H I R D A T

design for this script calls for charging the player money for the turrets to be able to run. You want the turrets to shut down if the player runs out of money, and to send a message to the player. The catch is, of course, that there are up to six turrets on any one ship, and you don't want the player to get sent six messages. So you want to write some code that ensures that no matter how many turrets are running, only one turret sends the message, but they all stop running. Some code that can accomplish this is as follows:

```
100 if <player doesn't have enough money>
101   if not [THIS] -> get local variable: name='turret.mutex'
102     [THIS] -> set local variable: name='turret.mutex' value=[TRUE]
103     write to player logbook 'Bad player - you ran out of money. All of
      your turrets are shutting down'
104   @ wait 10000 ms
105     [THIS] -> set local variable: name='turret.mutex' value=[FALSE]
106   end
107   return null
108 end
```

The concept above is pretty simple. A local variable is used as a lock. The first turret that happens to run this code will check to see if the local variable `turret.mutex` is set. It won't be set, so it goes on to line 102 which sets it. After that, you can be sure that none of the other turrets that are running the same script can get past line 101. They will check to see if the local variable is set, and when they see that it is set, they will continue on to line 107 where they will exit.

The reason why this works is because lines 101 and 102 are an atomic operation. There is no interrupt point in between the line that checks to see if the local variable is set and the line that sets it. To any other script that is running, both these operations appear to happen at the same time. This is what is meant by atomic.

The wait instruction at line 104 is also very important, because what we now need to do is make sure all the other turret scripts get a chance to run and check the `turret.mutex` local variable. If the entire section were atomic, then there would be no point to setting the local variable. No other script could possibly run to even check to see if the local variable is set. At this point, you need to stop the atomic operation, and give the script engine an opportunity to make sure the other turrets get an opportunity to run. The wait instruction makes the script wait long enough to ensure that all the other turrets get a chance to exit before the local variable is reset.

If the code were written like this instead:

```
101 if not [THIS] -> get local variable: name='turret.mutex'
102 @ call script 'some.other.script'
103   [THIS] -> set local variable: name='turret.mutex' value=[TRUE]
```

...then the operation to test the local variable and the operation to set the local variable would not be guaranteed to occur atomically due to the presence of an instruction that is an interrupt point at line 102. Thus, in the second example, the following is now possible:

1. Turret 1's script reaches line 101 and checks to see if the local variable `'turret.mutex'` has been set. It has not been set, so it passes to line 102.
2. Turret 1's script is an interrupt point (the scripting engine is now allowed to decide if another script should be allowed to run). The scripting engine suspends turret 1's script and runs turret 2's script.
3. Turret 2's script reaches line 101 and checks to see if the local variable `'turret.mutex'` has been set. Since turret 1's script was suspended before reaching line 103, it has not been set and turret 2's script passes to line 102 as well.



Now both turret 1's and turret 2's script will be inside the check and will both send messages to the player – our code that was supposed to prevent this failed.

In the above example, putting a *call script* statement in at line 102 wasn't really a very logical thing to do anyways. The natural way to write the code was the way shown in the first example, which would have made the code atomic. However, it is good coding practise to always remember to check where your code is and isn't atomic. Getting multiple scripts that are running concurrently to interact and operate with each other can be troublesome. Just because the code reaches an interrupt point doesn't mean the scripting engine has to suspend the script and restart another. This all depends on many factors that are beyond a script's control. This means that a script that depends on certain actions being atomic that inadvertently have interrupt points at the wrong spot might run just fine one time, and fail at another. At what interrupt points the scripting engine decides to suspend one script and restart another. Making sure you are always aware of when and where your interrupt points occur and what can happen at them will help ensure your code performs the way you intend it to perform.

### 7.3 ARTIFICIAL LIFE (AL) ENGINE PLUGINS

An Artificial Life (or AL) plugin is simply a script – or rather a series of scripts – that follow a certain structuring convention. The system is designed to make it easier to write scripts that perform environmental functions. That is, scripts that run in the background that add to the general ambiance.

In general an AL plugin usually consists of at least three scripts:

1. Registration script – this script informs the system of the presence of an AL plugin.
2. Event handler – this script handles the several AL system events.
3. Timer handler – the real meat of an AL plugin, this script is run periodically.

#### Registration Script

The simplest of all the AL plugin scripts. The name of this script must be “al.plugin.<pluginname>”. All scripts that are named thus are run automatically by the AL engine every time a new game starts, or a game is loaded.

All this script needs to do is register an AL plugin's event handler script:

```
001 al engine: register script=<scriptname>
```

Whatever script is specified in the *al engine: register script* instruction becomes that plugin's main event handler script.

#### Event Handler Script

The event handler script must be written to accept two string arguments. The first is a string that acts as a unique identifier for that AL plugin (the AL engine actually uses the name of the event handler script for this). The second is a string that identifies the type of event that is occurring. There are several different AL events – all of them need to be dealt with by an AL plugin's event handler.



Event	Description
'init'	Occurs when a new game starts, or when a new AL plugin is detected for the first time.
'reinit'	Occurs each time a game is loaded.
'start'	Occurs when the player turns on an AL plugin in the Artificial Life Settings menu.
'stop'	Occurs when the player turns off an AL plugin in the Artificial Life Settings menu.
'isenabled'	Occurs when the Artificial Life Settings menu is about to be displayed. Informs the AL engine whether or not this plugin is enabled.
'timer'	Occurs at a regular interval as set with the <i>al engine: set plugin timer interval</i> instruction. This event occurs whether or not the plugin is actually enabled or not.

Table 7.3 - AL Engine Event Types

The “isenabled” event is required because the AL system does not store the state of a plugin. Even though you can select whether a plugin should be run from the Artificial Life Settings menu, it is each individual plugin that actually stores this information. Your plugin could thus decide that it didn't want to ever turn off, if it wanted to, by ignoring “stop” events and always returning [TRUE] when it is asked if it is enabled.

Once a timer interval has been set with an *al engine: set plugin timer interval* instruction, the “timer” event will occur regularly whether or not the plugin is enabled (remember, the system doesn't even know if the plugin is enabled or not).

The AL engine doesn't store any information about a plugin. This means it is up to each individual plugin to store its own internal state. Because of the way that arrays are stored when stored as a global variable, they are uniquely suited to storing an AL plugin's state. An AL plugin should use a single array as the method of storing all its data. This has the added benefit of using only a single global variable for any given AL plugin. The plugin ID as passed to the plugin's event handler should be the name used to store this global variable under. Typically, the first two elements of the array store the version of the array, and the state of the plugin (whether or not it is enabled). It's important to have a version number in case later versions of the plugin change the structure. Once the structure definition changes, without a version number in the array, the script would have no way to tell what version of the plugin created the array – and thus what the actual structure of the array was. With a version number in the array as the very first element, the script can check to see what the data structure's version is and take appropriate measures if it is an old version.

Of course, there is no actual **requirement** to store your data in this way. However, as all built-in AL plugins do, and as any future plugin that gets signed will store its data this way, it's a very good idea to conform to this standard. It will make the plugin easier to read for others if it is written in the expected way.

What follows is an example of an AL plugin's event handler. This is from the Hired Gunnery

#### Technical Tidbit

When an array is stored as a global variable, it is moved into an area of memory that X<sup>2</sup> preserves when you save your game. When you subsequently retrieve the array using the *get global variable* instruction, what is actually retrieved isn't the whole array, but a the address to where the array is. A pointer to it. It is done this way because arrays are typically a lot larger than other types of data. Giving a script a pointer to the array is a lot more efficient than moving the entire array itself.

Since the entire array isn't copied out, changes you make to that array are actually stored back in the global variable memory area. This means that when a value in that array is changed, you don't need to execute a *set global variable* instruction to store the changed array.

Only arrays work this way. The actual value of other data types are copied out when a *get global variable* instruction is executed.



# THE TUTORIAL

Crews plugin:

Arguments

1: al.PluginID, Var/String, 'Plugin ID'  
2: al.Event, Var/String, 'AL Event'

```
001 * -----
002 * Hired Gunnery Crews version 3.00
003 * Military Transports AL Plugin
004 * Written by Kurt Fitzner - a.k.a. Reven
005 * -----
006 * Main AL event handler
007 * -----
008
009 * First some DEFINES - to make the state array easier to read
010 $AL.STATE.VERSION = 0
011 $AL.STATE.ENABLED = 1
012
013 $al.State = get global variable: name=$al.PluginID
014 if not $al.State
015     $al.State = array alloc: size=7
016     $Version = get script version
017     $al.State[$AL.STATE.VERSION] = $Version
018     $al.State[$AL.STATE.ENABLED] = [TRUE]
019     set global variable: name=$al.PluginID value=$al.State
020 end
021
022 $al.Retval = null
023 if $al.Event == 'init' OR $al.Event == 'reinit'
024     $al.PluginDesc = sprintf: pageid=2498 textid=1100, null, null, null,
        null, null
025     al engine: set plugin $al.PluginID description to $al.PluginDesc
026     al engine: set plugin $al.PluginID timer interval to 900 s
027 else if $al.Event == 'start'
028     $al.State[$AL.STATE.ENABLED] = [TRUE]
029 else if $al.Event == 'stop'
030     $al.State[$AL.STATE.ENABLED] = [FALSE]
031 else if $al.Event == 'isenabled'
032     $al.Retval = $al.State[$AL.STATE.ENABLED]
033 else if $al.Event == 'timer'
034     @ = [THIS] -> call script 'al.miltransports.timer' : Plugin
        ID=$al.PluginID AL state data=$al.State
035 end
036 return $al.Retval
```

- Lines 1-9: Just a script identification and a little commenting. Comments are your friend.
- Lines 10-11: Arrays are hard to read when used as a packed structure. Making some variables for use like a 'C' language *#define* is a good idea to help readability.
- Lines 13-20: If the global variable holding the plugin's state array doesn't exist yet, then create the array. Use the current version number of the script as the array's data version number. This lets you know exactly what version of the script created the array. The array is created large enough to store all the persistent data needed by the plugin.
- Line 22: An AL plugin's event handler only needs to return a value when called with the “isenabled” event. This line just initializes the variable used to return a value to null.
- Lines 23-26: Code to handle the “init” or “reinit” events. For this particular plugin (actually, for almost all plugins) there isn't a need to perform different activities for “init” and “reinit” events. The plugin's timer interval is set here – in this case to 30 minutes (900 seconds). The plugin's description is also set here. Reading the description from a language





file makes the plugin easier to customize for different languages. This description is what is displayed in the Artificial Life Settings menu.

- Lines 27-30: The “start” and “stop” events. All that is done here is to set the internal state variable used to store whether the plugin is enabled or not to [TRUE] or [FALSE]. Anything else that needs to be done when your plugin starts or stops can be performed here.
- Lines 31-32: Handle the “isenabled” event. Simply set the return value for the script to be equal to the state variable that holds the enabled state for the plugin.
- Lines 33-34: Handle the “timer” event. Since most of an AL plugin's functionality is in the timer event, normally this is an external script called from the event handler script. This makes the event script smaller and easier to read.

### Timer Handler Script

The timer handler script is where most of the actual meat of an AL plugin is performed. It is executed periodically at an interval set by the *al engine: set timer interval* instruction. As mentioned above, this event is performed whether or not the plugin is actually enabled. Therefore, your plugin's timer handler should test to see whether or not the plugin is enabled before it does anything:

#### Arguments

1: `al.PluginID`, Var/String, 'Plugin ID'  
2: `al.State`, Value, 'AL state data'

```
001 $AL.STATE.VERSION = 0
002 $AL.STATE.ENABLED = 1
003 if $al.State[$AL.STATE.ENABLED]
...   all the work done here
100 end
```

After the test for whether or not the AL plugin is enabled, place all your code that performs your AL plugin's duties. Of course, you can also distribute duties to other scripts which are called from here.

## 7.4 AUTOMATIC COMMAND RESTARTING

The [Reinit Script Caches](#) section in the introduction touched on the concept of script command caching. This caching has a side effect that may not be readily apparent. This section will discuss the consequences of script caching and talk about how you the developer can work around it.

First of all, a review of script caching. There are actually two types of caching that occur within the scripting engine:

1. The global script cache. Every script that is attached to a command, such as with the [global script map](#) instruction, is stored in a cache. Whenever that command is executed on any ship, the script is not read from its file, it is read from this cache. Using the [Reinit Script Caches](#) command in the scripting menu will cause this cache to be refreshed. It also causes all setup scripts be rerun.
2. Running script cache. Every script that is currently running is stored separately. If there are one hundred ships all running the same script command, then there are one hundred copies of that script in memory and saved in save games.

The first type of caching doesn't have any real consequences to script developers. It is easy to use the Reinit Script Caches command to cause this cache to be recreated. The second type of



T

H

I

T

H

R

I

A

T

caching can have a significant effect.

Imagine that you wish to make changes to a script that is used extensively – perhaps one that is attached to a trade command that has tens or even hundreds of ships running it. Each one of those ships has its own copy of that script, and there is no external way to cause them all to reload the script. The only solution is for every one of those scripts to terminate and reload. Do you as a developer want to force a player to go through hundreds of ships and reissue a command to every one when your script is updated? Your script won't be very popular if a player has to do this more than once.

The answer is scripts that are intelligent enough to know when they are changed, and that can restart themselves.

There are two main considerations when giving your scripts the ability to self-restart: detecting when a change has occurred, and performing the restart.

### Detecting Script Changes

There is a convenient versioning system built right into the script editor. Every script has a number which represents its version. The *get script version* instruction will return that version number. It cannot, however, be used directly in a script to determine when the script has changed, because the instruction will return the version of the script that was loaded when it was first run – not the version of the script that currently exists on disk. Your script is going to need a little outside help to determine when it has been changed. A common source of this help is in a setup script.

Setup scripts aren't attached to a menu command, so they aren't stored in the global script cache. They also aren't persistent – they run, perform their duty, and terminate. This means that every time they are run, they are loaded fresh from disk. The scripting engine runs them automatically every time a new game is started or a saved game is loaded. A setup script is required for any script which is attached to a command anyways, so they are ideal for our purposes. Adding two lines to a setup script will give our restarting code all the help it needs:

```
100 * This code goes in the setup script
101 $Version = get script version
102 set global variable: name='myplugin.version' value=$Version
```

All the code in our setup script has to do is get its version number, then store that as a global variable.

Our actual script – the one that will be detecting when it changes, it now just needs to check a global variable against its own version number:

```
120 * This code goes in the command script
121 $CurrentVersion = get script version
122 $GlobalVersion = get global variable: name='myplugin.version'
123 if not $CurrentVersion == $GlobalVersion
124 * Restart
125 end
```

Quite simple – get the global version, get the script version, if they don't match then restart. The one “gotcha” is this mechanism assumes the version numbering between the setup script and the command script is the same. Some script writers like to have a different version for each separate script file in a plugin that tracks changes to that individual file. The script change detection method above, however, assumes that you have a common version number across all scripts for a given plugin<sup>16</sup>. If you use different version numbering for each script file, then

<sup>16</sup> The author recommends using a common version number across all scripts in a given plugin for several reasons. For one, it allows the use of the above-mentioned restarting system. Hard-coding in the version numbers for restart detection purposes is a less visible method. Having a common version number also makes all files associated with your plugin



change the above code – eliminate the *get script version* instructions and replace them with a hard coded number.

## Performing the Restart

The most important thing to remember when coding your restart system is that a command cannot directly restart itself. This is a limitation in the scripting engine. In order to restart your command properly, it must be done in two steps. A global process is started and given the information on which script to restart on which object. It is that global process which then restarts the command. The above example with the restart code filled in is as follows:

```
120 * This code goes in the command script
121 $CurrentVersion = get script version
122 $GlobalVersion = get global variable: name='myplugin.version'
123 if not $CurrentVersion == $GlobalVersion
124     $NULL = null
125     START $NULL -> call script 'plugin.myplugin.lib.restart' : Command to
        restart='MyCommand' Task ID=0 Target Ship=[THIS]
126     return null
125 end
```

And here is the restart “helper” script:

### Arguments:

```
1: WhichScript , Var/String , 'Script to restart'
2: TaskID, Var/Number, 'Task ID'
2: Ship , Var/Ship owned by Player , 'Target Ship'
```

```
001 @ = wait randomly from 400 to 600 ms
002 if $WhichScript == 'MyCommand'
003     $Ship -> start task $TaskID with script 'plugin.myplugin.mycommand' and
        prio 0: arg1=$TaskID arg2=null arg3=null arg4=null arg5=null
004 else if $WhichScript == 'MyOtherCommand'
005     $Ship -> start task $TaskID with script 'plugin.myplugin.myothercommand'
        and prio 0: arg1=$TaskID arg2=null arg3=null arg4=null arg5=null
006 end
007 return null
```

- Line 1: A wait to give the original script enough time to terminate.
- Line 2: There may be several commands distributed with a plugin – no sense in having a separate restart script for each of them. Just give your restart script an argument telling it which command to restart. Then, depending on what the argument is, a different script can be restarted.
- Line 3: You'll notice that the *call script* instruction was not used. With the START prefix it could have been. However, it is possible that you may want to restart scripts used as “additional ship commands” or turret commands. In those cases, the scripts won't be running as task zero. You will have to use “start task” in order to start a script under a different task ID. If you aren't making any scripts that run on a different task, then using *call script* here is fine. It is also fine to continue to use *start task* with a task ID of zero.

---

easier to pick out. It also makes the “release version” (the version number you release your plugin under) and all the individual script file versions the same. Simply number your scripts “100” to mean version 1.00, “210” to mean version 2.10, etc.

In all, it is the author's opinion that this is a more visible and elegant solution.



## Annex A. DATA CHARTS

### A.1 PLOT STATES

<i>Plot</i>	<i>State Flag</i>	<i>Description</i>	<i>Plot</i>	<i>State Flag</i>	<i>Description</i>
1	3	Main introduction ended	2	18	AP Gunner mission
1	4	Message from Terracorp received	2	19	AP Gunner mission
1	5	Contacted Terracorp	2	20	Arrived at Black Hole Sun
1	6	Accepted the delivery mission	2	21	Failure of AP Gunner mission
1	7	Received message regarding the mission	2	22	Message from Ban Danna received
1	8	Successfully completed delivery	3	3	Have contacted Ban Danna
1	9	Received transport mission briefing	3	4	Have contacted Mik Balser (M1 C.O.)
1	10	Accepted transport mission	3	5	Found the correct asteroid
1	11	Received message regarding the mission	3	6	Given the order to destroy the installation
1	12	Passenger on board transport	3	7	Installation successfully destroyed
1	13	Attack of the pirates	3	8	Given the order to collect the Khaak parts
1	14	Rescued by “secret” ship	3	9	Successfully collected the Khaak parts
1	15	Transport mission failed	3	10	Reported back to Mik Balser
1	16	Passenger successfully transported	3	11	Returned and reported to Ban Danna
1	17	Khaak activation	3	12	Contacted Saya Kho
1	18	One or more failed plot 1 missions	3	13	Contacted Mi'ton
1	19	Mission to meet Brennan	3	14	Payed Mi'ton (the little weasel)
2	3	Introduction of the Khaak	3	15	Received the Khaak coordinates
2	4	Arrived at Brennan's Triumph gate	3	16	Jumped into Khaak space
2	5	Looking for the cover plate	3	17	Returned to Saya Kho
2	6	Have found the cover plate	3	18	Found the “black box”
2	7	Met Bret in Antigone Memorial	3	19	Returned with the “black box”
2	8	Visited the Antigone Memorial museum	3	20	Instructed to fly to Omicron Lyrae
2	9	Jumpdrive installed in ship	3	21	First Khaak wave arrived
2	10	Arrived at Paranid Prime	3	22	Received message from Kyle Brennan
2	11	Purchased the LFL device	3	23	Khaak ships have arranged themselves
2	12	Escorting Bret	3	24	Close to the Khaak battleship
2	13	Escorting Saya	3	25	The Khaak battleship has fired
2	14	AP Gunner mission	3	26	Ordered to destroy the M0 power generators
2	15	AP Gunner mission	3	27	Power generators destroyed
2	16	AP Gunner mission	3	28	Battleship is destroyed
2	17	AP Gunner mission	3	29	Invited to award ceremony

Table 8.1 Plot states for the is plot instruction



## A.2 AUDIO SAMPLES CATALOGUE

A catalogue of all audio sound-effect samples in X<sup>2</sup>.

#	Dur.	Description	PS <sup>17</sup>
102	0.67	Engine sound (Capital ships)	No
103	1.02	Engine sound (Argon/Boron/Xenon M3)	No
104	1.45	Engine sound (All M4 except Khaak)	No
105	1.46	Engine sound (Boron/Split TP, All M5 except Khaak)	No
106	2.00	Engine sound (Argon/Paranid/Teladi TP, All TS, Goner Ship)	No
107	2.37	Engine sound (Paranid/Split/Teladi/Pirate M3)	No
108	1.95	Ambient station sound	No
109	0.90	Engine sound (UFO/Spacefly)	No
110	0.74	Engine sound (Unknown)/Dynamo	No
111	0.16	Weapons fire (Khaak laser)	No
112	1.71	Engine sound (Missile)	No
113	0.92	Weapons fire (Ion Disruptor)	No
114	1.45	Engine sound (Khaak M3, M4, M5)	No
903	1.02	Hull impact (no shields)	Yes
904	1.42	Hull impact (shields)	Yes
906	1.02	Hull impact (no shields) - identical to 903	Yes
907	1.12	Weapon impact (IRE, Alpha PSG)	Yes
908	1.30	Weapon impact (PAC, Beta PSG)	Yes
909	1.59	Weapon impact (HEPT/PPC, Gamma PSG)	Yes
912	0.44	Weapons fire (Alpha IRE)	Yes
913	0.73	Weapons fire (Beta IRE)	Yes
914	0.65	Weapons fire (Gamma IRE)	Yes
915	1.33	Small (Mosquito/Wasp) missile launch	Yes
916	2.11	Medium (Dragonfly/Silkworm) missile launch	Yes
917	3.80	Large (Hornet) missile launch	Yes
918	1.70	Missile/small ship explosion	Yes
919	1.41	M6 ship Explosion	Yes
920	6.42	Station/Capital ship explosion	Yes
922	2.97	Fade in and out low white noise	Yes
923	0.76	Alert sound (hostile contact)	Yes
924	0.39	Electric buzzer	Yes
925	0.75	Alert sound (incoming missile)	Yes
928	0.25	High-pitched electric buzzer	Yes
929	1.13	Unconventional weapons fire or effect	Yes

<sup>17</sup> This column indicated the samples that are available to be played through the *play sample* instruction. Unfortunately, most of the engine effects are not available to be played this way.



# THE X2 SOUND EFFECTS

## The X<sup>2</sup> MSCI Programmer's Handbook

#	Dur.	Description	PS
930	0.29	Radio keying off	Yes
931	0.42	Radio beep	Yes
932	0.36	Radio keying on	Yes
934	5.00	Engine cycling up (cut scene?)	Yes
935	0.90	Engine sound (unknown - cut scene?)	Yes
936	2.55	Engine cycling down (cut scene?)	Yes
937	3.80	Engine background ambiance	Yes
938	3.42	Engine or systems background ambiance	Yes
940	0.55	Engine or systems background ambiance	Yes
943	1.61	Background clunking (station FX?)	Yes
944	0.76	Electric motor ambiance	Yes
945	3.76	Door sealing or industrial press with echo	Yes
946	1.57	Jet engine effect with tarmac-like reverb	Yes
949	3.59	Quiet cockpit ambiance	Yes
950	5.38	Ambient engine sound	Yes
951	3.64	Ambient reactor sound	Yes
952	2.48	Ambient engine sound	Yes
953	0.54	Menu sound (menu on)	Yes
954	0.54	Menu sound (menu off)	Yes
955	0.28	Menu sound (select)	Yes
956	0.27	Menu sound (error)	Yes
957	0.11	Menu sound (cursoring)	Yes
958	0.67	Weapons fire (Alpha PAC)	Yes
959	0.99	Weapons fire (Beta/Gamma PAC)	Yes
960	0.67	Weapons fire (PPC/HEPT)	Yes
961	26.52	Moon landing "Eagle Has Landed"	Yes
962	1.05	Electric vibration	Yes
963	1.39	Docking arms extending (cut scene)	Yes
964	2.50	Cargo bay opening	Yes
965	3.42	Chemical rocket effect (planet ferry launch)	Yes
966	3.22	Scanner effect	Yes
967	4.04	Industrial electric motor whine, clunking to a stop	Yes
968	0.23	Menu "open window" effect	Yes
969	1.64	Cargo bay closing effect	Yes
970	1.71	Mechanical clunking, start of industrial electric motor	Yes
971	0.76	Fizzle effect - electric arc	Yes
972	0.29	Short beep	Yes
973	17.10	Chemical rocket effect (longer version of 965)	Yes





# THUNDERBOLT

## The X² MSCI Programmer's Handbook

#	Dur.	Description	PS
974	15.97	Energy field/motor/lift effect, then winding down	Yes
975	10.80	Engine starts, runs, clunk, engine stops	Yes
976	16.04	Fade in jet with Doppler effect then long explosion or sonic boom	Yes
977	5.61	Engine shutting down	Yes
978	21.26	Long engine wind up	Yes
979	0.48	Camera shutter/wind (Video Enhancement Goggles)	Yes
980	0.61	Alarm sound/Sudden energy power up	Yes
981	1.22	Alarm sound/Sudden energy power down	Yes
982	0.11	Camera shutter	Yes
983	8.71	Quiet engine windup	Yes
984	12.84	Large reverberating explosion/Thunder	Yes
985	0.27	Computer bleep (used in Engine Booster script)	Yes
986	0.02	Short, sharp computer beep	Yes
987	0.35	Sector Map zoom in noise	Yes
988	0.27	Computer bleep - loud (same as 985 but louder)	Yes
989	0.40	Sector Map zoom out noise	Yes
990	3.40	Pneumatic seal	Yes
991	3.86	Transporter effect	Yes
992	0.32	Bug splat	Yes
993	21.33	Heavy breathing/Spacesuit panic (cut scene)	Yes
994	16.00	Heavy breathing/Spacesuit panic - shorter (cut scene)	Yes
995	0.62	Video camera zoom in effect (cut scene)	Yes
996	1.03	Video camera zoom in effect - longer (cut scene)	Yes
997	10.19	Pneumatic seal/motor whine - Extend docking arms (cut scene)	Yes
998	5.16	Rolling cymbal-like effect fade in & out	Yes
1000	9.34	Spacesuit visor cracking (cut scene)	Yes
1001	1.41	Mechanical press-like effect - distant	Yes
1002	1.27	Rotary grinder/loud modem-like effect	Yes
1003	13.84	Alien Planet ambiance	Yes
1004	14.30	Underwater bubbles	Yes
1005	9.43	Electronic background buzz/hum	Yes
1006	1.37	Station announcement paging effect	Yes
1007	2.25	News Station Audio Logo	Yes
1008	3.22	Incoming Message alert effect	Yes
1100	3.95	Promotion alert effect	Yes
1101	2.85	Electric growl, distant, pitch bends down	Yes
1102	1.21	Electric whoop, distant, pitch bends up slightly	Yes
1103	0.19	Electric/computer system bleep, distant	Yes



# THE X2

## The X<sup>2</sup> MSCI Programmer's Handbook

#	Dur.	Description	PS
1104	8.19	Mechanical clunking, intermittent, ambient	Yes
1105	2.40	Reverberating electrical white-noise with sweeping effect	Yes
1106	4.72	Ring modulated sweeping electrical effect with approach and fade out	Yes
1107	6.83	Heavy breathing	Yes
1108	8.19	Person trying to fake wind effect into a microphone	Yes
1109	4.51	Dull explosion	Yes
1110	2.75	Small, low, dull explosion or shock wave	Yes
1111	55.72	Long electrical modulated ambiance with rhythmic clack	Yes
1112	31.50	Very long engine build up, sustain, quick release	Yes
1113	7.49	Electrical/pneumatic sounds interspersed with a whirring buildup (used in Engine Booster script)	Yes
1114	0.32	Bloop-bloop-bleep (used in Gunnery Crews scripts)	Yes
1115	2.77	Mechanical sound with pneumatic recycling (used in Gunnery Crews scripts)	Yes
1116	0.12	Slightly metallic click/switching effect	Yes
1117	5.89	Loopable electrical modulated ambiance with rhythmic clack	Yes
1118	1.59	Alternating DTMF warning beeps (used in Engine Booster script)	Yes
1119	0.13	Click	Yes
1120	0.24	Solenoid engage/release (quick)	Yes
1121	60.55	Ship stealing effect (cut scene)	Yes
1122	100.46	Lift/electric-powered wheeled vehicle moving about	Yes
1123	18.09	Quiet electronic ambiance with communication sounds	Yes
1124	0.61	tri-tone beep alert	Yes
1125	3.50	Electric hatch opening or closing	Yes
1126	0.40	Servo shifting	Yes
1127	1.44	Metallic impact/mechanical clank	Yes
1128	2.53	Electric motor/drill	Yes
1129	1.75	Obnoxious alarm/hailing effect (used in Sector/Universe Trader script)	Yes
1130	1.02	Mechanical clack with lead out	Yes
1131	52.69	Typist/Slow telex machine	Yes
1132	4.99	Computer drive (old-style stepper motor) access effect	Yes
1133	2.42	Chime, long and soft	Yes
1134	0.75	Electric buzzer	Yes
1135	0.43	Key press	Yes
1136	0.43	Key press	Yes
1137	2.56	Industrial ambiance, distant	Yes
1138	4.47	Industrial ambiance/bottles clanking, distant	Yes
1139	6.35	Crashing with meowing cat	Yes
1140	8.18	Electronic radio communications/RTTY	Yes
1141	35.96	underwater bubbles (like 1004 but longer)	Yes



T

H

U

T

H

R

U

A

T

## The X<sup>2</sup> MSCI Programmer's Handbook

#	Dur.	Description	PS
1142	4.28	Three metallic clanks/hammer strikes on metal	Yes
1143	18.38	Submarine hull popping/creaking	Yes
1144	4.07	Weapon firing effect or deadly automated defence mechanism	Yes
1145	8.07	Energy field/electronic ambiance - loopable	Yes
1146	8.04	Energy field/electronic ambiance	Yes
1147	7.03	Heavily ring modulated effect/communications/weapon/force field	Yes
1148	13.39	Large explosion with several secondaries	Yes
1149	4.08	Mechanical/pneumatic recycling effect	Yes
1150	3.77	Distant explosion	Yes
1151	1.00	Dull impact	Yes
1152	2.33	Explosion/artillery firing	Yes
1153	2.00	Explosion/artillery firing (distant)	Yes
1154	4.40	Distant explosion	Yes
1155	4.00	Distant explosion with trailing reverb	Yes
1156	0.34	Fast pass by (train sweeping past effect from cut scenes)	Yes
1157	0.44	Mechanical/electronic system effect	Yes
1158	1.44	Person in pain	Yes
1159	6.05	Clanks and crashes (from opening cut scene explosion during escape attempt)	Yes

*Table 8.2 - Audio Samples*



### A.3 SPEECH SAMPLES CATALOGUE

The following charts list messages from the built-in X<sup>2</sup> language files for which there are corresponding speech or video. The messages are divided up into pages, where each page generally contains all the phrases for one particular theme or purpose.

#### Page 7 – Sector names

Page 7 contains all the sector names. The format for the ID numbers is 102XXYY, where XX and YY are the X and Y coordinates for the sector on the sector map, with X=01, Y=01 being the top-left most sector (Kingdom End). Spoken by the on-board computer voice.

ID #	Sector	ID #	Sector	ID #	Sector	ID #	Sector
1020101	Kingdom End	1020402	Argon Prime	1020614	Circle Of Labour	1021003	Emperor's Wisdom
1020102	Rolk's Drift	1020403	The Wall	1020616	Xenon Sector 472	1021004	Trinity Sanctum
1020103	Queen's Space	1020404	Farnham's Legend	1020617	Thyn's Abyss	1021005	Preacher's Refuge
1020104	Menelaus' Frontier	1020405	Bala Gi's Joy	1020701	Emperor Mines	1021006	Shore of Infinity
1020105	Ceo's Buckzoid	1020406	Blue Profit	1020702	Paranid Prime	1021007	Lucky Planets
1020106	Teladi Gain	1020407	Rhonkar's Fire	1020703	Priest Rings	1021008	Rolk's Legacy
1020107	Family Whi	1020408	Rhonkar's Clouds	1020704	Priest's Pity	1021009	Great Trench
1020114	The Vault	1020409	Tharka's Sun	1020705	Danna's Chance	1021010	Ceo's Doubt
1020117	Unknown Sector	1020410	Cho's Defeat	1020706	Nopileos' Memorial	1021104	Bad Debt
1020118	Xenon Sector 534	1020415	Family Tkr	1020707	Hatikvah's Faith	1021110	LooManckStrat's Legacy
1020120	Xenon Sector 596	1020416	Tkr's Deprivation	1020708	Aladna Hill	1021202	Unknown Sector
1020201	Three Worlds	1020417	Ghinn's Escape	1020709	Akeela's Beacon	1021203	Rhy's Desire
1020202	Power Circle	1020418	Hila's Joy	1020712	Scale Plate Green	1021204	Ministry Of Finance
1020203	Antigone Memorial	1020419	Ocean of Fantasy	1020713	Nyana's Hideout	1021210	Mi Ton's Refuge
1020204	Rolk's Fate	1020501	Red Light	1020714	Omicron Lyrae	1021215	Unknown Enemy Sector
1020205	Profit Share	1020502	Home of Light	1020715	Treasure Chest	1021217	Unknown Enemy Sector
1020206	Seizewell	1020503	President's End	1020716	Black Hole Sun	1021303	Family Rhy
1020207	Family Zein	1020504	Elena's Fortune	1020802	Empire's Edge	1021310	Moo-Kye's Revenge
1020214	Shareholder's Fortune	1020505	Olmancketslat's Treaty	1020803	Duke's Domain	1021316	Unknown Enemy Sector
1020215	Mines Of Fortune	1020506	Ceo's Sprite	1020804	Emperor's Ridge	1021401	Depths Of Silence
1020218	Getsu Fune	1020507	Family Rhonkar	1020808	Light of Heart	1021402	Dark Waters
1020219	Menelaus' Paradise	1020510	Patriarch's Keep	1020811	Eighteen Billion	1021403	Reservoir Of Tranquillity
1020220	Xenon Sector 597	1020511	Two Grand	1020812	Xenon Sector 347	1021404	Barren Shores
1020301	Cloudbase North West	1020517	Family Njy	1020816	Nathan's Voyage	1021409	Priest Refuge
1020302	Herron's Nebula	1020518	Njy's Deception	1020817	Wastelands	1021410	Cardinal's Domain
1020303	The Hole	1020519	Family Ryk	1020818	Unknown Sector	1021411	Sacred Relic
1020304	Atreus' Clouds	1020601	Cloudbase South West	1020819	Unknown Sector	1021415	Unknown Enemy Sector
1020305	Spaceweed Drift	1020602	Ore Belt	1020902	Preacher's Void	1021417	Unknown Enemy Sector
1020306	Greater Profit	1020603	Cloudbase South East	1020904	Pontifex' Realm	1021503	Great Reef
1020307	Thuruk's Pride	1020604	Split Fire	1020906	Light Water	1021509	Spring Of Belief
1020308	Family Pride	1020605	Brennan's Triumph	1020908	Montalaar	1021510	Friar's Retreat
1020310	Patriarch's Retreat	1020606	Company Pride	1020910	New Income	1021511	Pontifex' Seclusion
1020315	Home Of Opportunity	1020607	Thuruk's Beard	1020911	Ianamus Zura		
1020319	Bluish Snout	1020611	Profit Center Alpha	1020917	Interworlds		
1020401	Ringo Moon	1020612	PTNI Headquarters	1021002	Duke's Vision		

Table 8.3 - Speech Samples, Page 7 - Sector Names



T

H

U

T

H

R

U

A

T

**Pages 9 & 12 – Latin and Greek Letters**

Pages 9 and 12 consist of the letters of the Latin and Greek alphabets respectively, spoken by the on-board computer voice.

<i>Page 9 – Latin Alphabet</i>		<i>Page 12 – Greek Alphabet</i>	
<i>ID #</i>	<i>Letter</i>	<i>ID #</i>	<i>Letter</i>
500	A	100	alpha
501	B	101	beta
502	C	102	gamma
503	D	103	delta
504	E	104	epsilon
505	F	105	zeta
506	G	106	eta
507	H	107	theta
508	I	108	iota
509	J	109	kappa
510	K	110	lambda
511	L	111	mu
512	M	112	nu
513	N	113	xi
514	O	114	omicron
515	P	115	pi
516	Q	116	rho
517	R	117	sigma
518	S	118	tau
519	T	119	upsilon
520	U	120	phi
521	V	121	chi
522	W	122	psi
523	X	123	omega
524	Y	124	omega
525	Z		

Table 8.4 - Speech Samples, pages 9 and 12 - Latin and Greek Alphabets



THE  
MISC  
TIT  
L

## Page 13 – Miscellaneous Phrases

This page consists of all the various phrases, warnings, alerts, and other tidbits that are pieced together to make up most of what you hear when playing the game. Spoken by the on-board computer voice.

<i>ID #</i>	<i>Phrase</i>
1	Entering system
2	No Autopilot installed!
3	Autopilot damaged!
4	No Aim!
5	ejected
6	Targeting
9	No station for docking found!
10	Alert: Missile closing
11	Alert: Mine closing
13	No missiles
15	landed
17	Unidentified object
18	self-destructed
20	Message received
22	collision of
23	with
24	not installed
25	not available
27	available
29	destroyed
30	damaged
31	This ship is too large to dock with an M6 class corvette ship. Only M5 class scout ships can dock on a corvette
32	This ware type does not fit into your ship.
33	Container type too large for cargo hatch!
34	New formation initiated
35	Formation established
36	Target outside communication range
37	Remote connection lost
38	Game successfully saved
39	Saving failed
40	Loading savegame
41	Your are being promoted
42	This is a pirated copy of X2
43	Savegame only possible inside stations or with salvage insurance.





# THE X² FACTORY

## The X² MSCI Programmer's Handbook

ID #	Phrase
44	New mission received
45	This is the maximum distance in sectors your transporter ships will fly to buy the resources needed for this factory.
46	You can move funds to and from your factory. A factory needs money to be able to automatically send transporters to buy resources using advanced trading commands.
47	You can set if ships of other races are allowed to buy products from this factory
48	You can program the ship computer to use missiles during fights more or less frequently with this setting
49	You can set a flight formation if this ship is part of a group
50	You can set the homebase for a ship to allow advanced trade commands
51	You can program the ship computer to custom friend foe settings with this command
52	You can program the ship computer to inherit your global friend foe settings with this command
53	This advanced feature allows you to write custom made programs for the ship computer.
130	Please set remote command mode
131	Command accepted
132	New command set
133	New remote command cannot be executed
134	This missile is not compatible with your ship
135	Command rejected
136	This weapon is not compatible with your ship
137	This shield is not compatible with your ship
200	Selected
1200	You cannot buy this factory at the moment, because you do not have the necessary storage space to transport all the equipment included in this package. You need to find and hire a TL Class transportation cruiser!
1245	Factory construction initiated in sector:
1246	Factory construction finished in sector:
1247	Your factory in sector:
1248	needs more resources for production!
1250	No space for additional lasers
1251	No space for additional shields
1252	No space for additional missiles
1253	Not enough space in cargo hold
1254	Insufficient credits!
1255	Warning! Cargo bay open, shields are down!
1256	Cargo bay closed.
1257	Cargo bay now contains (continues at 1293)
1258	Target is now in firing range!
1259	Target left firing range!
1261	Autopilot now locked on
1262	Ship is under attack by



# F I R S T A T

## The X<sup>2</sup> MSCI Programmer's Handbook

ID #	Phrase
1263	Autopilot off
1264	Autopilot activated
1265	Attention. Energy low!
1266	Danger! Entering atmosphere
1267	No information available
1268	Ejecting
1269	Turbo engaged
1270	Shields critical
1271	Warning
1272	sold
1273	bought
1274	installed
1275	removed
1276	Emergency Signal from
1277	Assistance required at
1278	We are scanned
1279	by
1280	The ship is scanned
1282	Target lost
1283	Docking granted
1284	Docking denied
1285	Docking aborted
1286	transferred
1287	New price selected
1288	Funds transferred
1289	Resources are not sold
1290	Products are not bought
1291	Spherical Radar mode
1292	Planar Radar mode
1293	Cargo bay emptied
1294	ready
1295	The credits that this factory made by selling its products can be transferred to your account. Also credits from your account can be transferred to the factory, so freighters working for you can buy resources.
1296	You can specify at which price the freighters working for this factory should buy resources
1297	Changing this setting is only possible after you hire a transporter to work for this factory
1298	You can specify at which price other factories and their transporters can buy your product. If your price is competitive it is more likely other factories and stations will choose your factory
1299	You can order transport ships to help you transport the necessary resources to this factory. After you order one or more transporters from the shipyard that manufactured this factory, these transport ships will fly to this factory and work for you.



# T H I S I S I S T E R

## The X² MSCI Programmer's Handbook

ID #	Phrase
1300	Transporter ordered at Shipyard
1301	You can not sell transporters
1302	Transporter removed
1303	You can order fighter ships that will help to protect your factory. After you order one or more fighters from the shipyard that manufactured this factory, these fighters will fly to this factory and protect it.
1304	Fighter ordered at Shipyard
1305	You can not sell fighters
1306	Fighter removed
1307	Target locked on
1308	Target on
1310	Not functional in playable demo
1311	Not available in playable demo
1312	This demo will time out in t minus one minute
1313	Cheat Mode enabled
1319	Jump device charging at 10%...20...30...40...50%...60...70...80...90...Jumping...
1320	Shields
1321	Laser
1322	Ship Hull
1323	critical
1332	Receiving upgrade information for database
1333	Incoming message
1334	from
1339	Credits received
1340	We are being hailed
1341	scanning
1342	This object is owned by you
1350	System Check
1351	Alert. Ship systems malfunctioning. System check initiated.
1352	working
1353	broken
1354	not working
1355	out of order
1356	not responding
1357	malfunction
1358	Conventional Engine
1359	Weapon Systems
1501	Connection to local trading network established.
1502	Successfully docked
5001	Recording started



# THE M I S C E L L A N E O U S

## The X<sup>2</sup> MSCI Programmer's Handbook

<i>ID #</i>	<i>Phrase</i>
5002	Recording stopped
901299	You can specify how many of your own transporter ships should work for this factory.
901302	Transporter without destination will follow you and wait for new instructions.
901303	You can specify how many of your own fighter ships should work for this factory.
901306	Fighter without destination will follow you and wait for new instructions.
1100003	Warning, Proximity Alert
1100014	Boardcomputer ready to receive new command
1100015	New command accepted
1100016	Affirmative
1100017	New homebase accepted
1100018	Attention: One of your factories is under attack
1100019	Attention: One of your ships is under attack
1100020	Attention: One of your factories stopped producing. Check resource and money supply.
1100025	Attention: X2 could not verify the original game CD on start-up. Please make sure you use the correct CD and restart the game.
1100026	Self destruct sequence initiated
1100027	Attention: X2 executable file has been manipulated
1100039	Station outside autodocking range
1100040	Target outside transporter range. Teleportation impossible.
1100041	Target not your property. Teleportation impossible.
1100042	No wingman available
1100043	No other ships owned by you are currently in this sector
1100044	New Command:
1100045	No fighting droids available
1100046	No mine available
1100047	Attention: Mine detected
1100048	The factory does not have enough credits to buy resources
1100049	The requested ware can not be bought for less than the current price limit.
1100056	Jump device aborted - Not enough energy available
1100065	Welcome aboard

*Table 8.5- Speech Samples, page 13 - Miscellaneous Phrases*



T  
H  
I  
S  
I  
S  
I  
S  
U  
E  
I  
S  
T  
A  
T

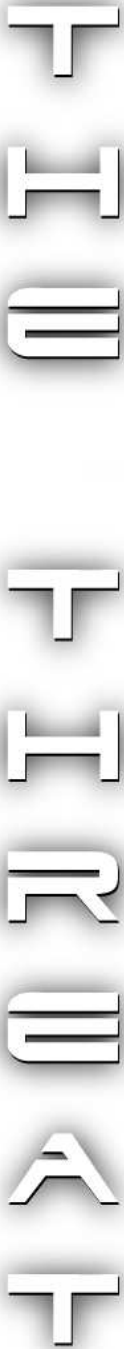
## Page 17 – Object, Ship, and Factory Names and Descriptions

All of the different station, ship, object, and ware names along with their descriptions are in page 17. The description text is not give here, as there is little use for it in a script. All the names are given in the following tables which are organized by type:

### Stations

<i>ID #</i>	<i>Station</i>	<i>ID #</i>	<i>Station</i>	<i>ID #</i>	<i>Station</i>
2011	Solar Power Plant	2681	Beta HEPT Forge	4281	Royal Boron Dry-dock
2021	Ore Mine	2691	Gamma HEPT Forge	4381	Split Dockyard
2031	Silicon Mine	2711	Shield Prod. Facility 1MW	4481	Paranid Pier
2041	Weapon Component Factory	2721	Shield Prod. Facility 5MW	4581	Teladi Showroom
2051	Crystal Fab	2731	Shield Prod. Facility 25MW	4621	Ion Disruptor Forge
2061	Quantum Tube Fab	2741	Shield Prod. Facility 125MW	4631	Mobile Drilling System Factory
2071	Chip plant	2811	Mosquito Missile Factory	4671	Alpha PSG Forge
2081	Computer Plant	2821	Wasp Missile Factory	4681	Beta PSG Forge
2141	Wheat Farm	2831	Dragonfly Missile Factory	4691	Gamma PSG Forge
2151	Cattle Ranch	2841	Silkworm Missile Factory	4811	Advanced Satellite Factory
2171	Rimes Fact	2851	Hornet Missile Factory	4871	Alpha PPC Forge
2181	Cahoona Bakery	3101	Federal Argon Shipyard	4881	Beta PPC Forge
2191	Space Fuel Distillery	3181	Free Argon Trading Station	4891	Gamma PPC Forge
2241	Plankton Farm	3191	Argon Equipment Dock	4901	Khaak missile forge - Sting
2251	Bio Gas Factory	3201	Royal Boron Shipyard	4911	Khaak missile forge - Needle
2271	Stott Mixery	3281	Royal Boron Trading Station	4921	Khaak missile forge Thorn
2281	BoFu Chemical Lab	3291	Boron Equipment Dock	4941	Lasertower Weapon Forge
2341	Scruffin Farm	3301	Split Shipyard	4951	Mass Driver Forge
2351	Chelt Space Aquarium	3381	Split Trading Port	4961	Ammunition Factory
2371	Massom Mill	3391	Split Equipment Dock	4971	TerraCorp Headquarters
2381	Rastar Refinery	3401	Paranid Shipyard	4981	TerraCorp Crystal Fab
2441	Soyfarm	3481	Paranid Trading Dock	4991	TerraCorp Computer Plant
2451	Snail Ranch	3491	Paranid Equipment Dock	5001	TerraCorp Solar Power Plant
2471	Space Jewellery	3501	Teladi Shipyard	5011	TerraCorp Wheat Farm
2481	Sovery	3581	Teladi Trading Station	5021	TerraCorp Cattle Ranch
2541	Flower Farm	3591	Teladi Space Equipment Dock	5031	TerraCorp 125MW Shield Prod. Facility
2551	Teladianium foundry	3681	Xenon Shipyard	5041	TerraCorp Quantum Tube Fab
2561	Dream farm	3691	Xenon Station	5061	Federal Argon Installation
2571	Sun oil refinery	3741	Goner Temple	5071	Royal Boron Research Station
2581	Bliss Place	3761	Pirate Anarchy Port	5081	Paranid Communications Facility
2611	Alpha IRE Forge	3781	Pirate Base	5111	Argon Merchant Scrapyard
2621	Beta IRE Forge	3891	Satellite Factory	5121	Boron Ship Reclamation Yard
2631	Gamma IRE Forge	3911	SQUASH Mine Factory	5131	Split Scrap Merchants
2641	Alpha PAC Forge	3921	Lasertower Factory	5141	Paranid Salvage Merchants
2651	Beta PAC Forge	3931	Drone Factory	5151	Teladi Company Scrap Dealers
2661	Gamma PAC Forge	4181	Federal Argon Wharf	9021	Unknown Enemy Station
2671	Alpha HEPT Forge				

Table 8.6 - Speech Samples, page 17 - Station Names

pg. 104





## The X<sup>2</sup> MSCI Programmer's Handbook

ID#	Ware	ID#	Ware	ID#	Ware
2563	Swamp Plant	4823	Ore Collector	5693	Trade Command Software MK2
2573	Nostrop Oil	4833	Camouflage Device	5713	Fight Command Software MK1
2583	Space weed	4863	Spacefly Collector	5723	Fight Command Software MK2
2613	Alpha Impulse Ray Emitter	4873	Alpha Photon Pulse Cannon	5733	Special Command Software MK1
2623	Beta Impulse Ray Emitter	4883	Beta Photon Pulse Cannon	5873	Trade Command Software MK3

Table 8.7 - Speech Samples, Page 17 – Wares & Upgrades

## Ships

ID#	Ship	ID#	Ship	ID#	Ship
3111	Argon Colossus	3571	Teladi Vulture	4231	Boron Barracuda
3121	Argon Titan	3611	Xenon J	4241	Boron Mako
3141	Argon Buster	3621	Xenon K	4261	Boron Manta
3151	Argon Discoverer	3631	Xenon L	4311	Split Dragon
3161	Argon Mammoth	3641	Xenon M	4351	Split Jaguar
3211	Boron Shark	3651	Xenon N	4361	Split Iguana
3221	Boron Ray	3661	Xenon I	4371	Split Caiman
3251	Boron Octopus	3671	Xenon H	4411	Paranid Nemesis
3261	Boron Orca	3771	Pirate Ship	4441	Paranid Pericles
3271	Boron Dolphin	3811	Pirate Orinoco	4461	Paranid Hermes
3281	Royal Boron Trading Station	3821	Pirate Bayamon	4471	Paranid Demeter
3291	Boron Equipment Dock	3831	Pirate Mandalay	4511	Teladi Osprey
3311	Split Raptor	3841	Navigation Relay Satellite	4541	Teladi Buzzard
3321	Split Python	3851	SQUASH Mine	4551	Teladi Harrier
3331	Split Mamba	3861	Lasertower	4561	Teladi Toucan
3341	Split Scorpion	3871	Fighter drone	4731	Xperimetal Shuttle
3361	Split Elephant	4011	Khaak Carrier	4741	Space Suit
3381	Split Trading Port	4021	Khaak Destroyer	4751	Goner Ship
3391	Split Equipment Dock	4031	Khaak Fighter	4771	Camera Drone
3401	Paranid Shipyard	4041	Khaak Interceptor	4931	Paranid Perseus
3411	Paranid Zeus	4051	Khaak Scout	5051	Navigational Beacon
3421	Paranid Odysseus	4061	Khaak Cluster	5091	Argon One
3451	Paranid Pegasus	4081	Khaak Station	5501	Saiien II
3461	Paranid Hercules	4111	Argon Centaur	5511	Nikkonofune
3471	Paranid Ganymede	4131	Argon Nova	5521	Blue Arrow
3511	Teladi Condor	4161	Argon Express	5543	TRACKER Mine
3521	Teladi Phoenix	4171	Argon Mercury	5581	Pyramid Income
3531	Teladi Falcon	4211	Boron Hydra	5591	Saiien III
3561	Teladi Albatross				

Table 8.8 - Speech Samples, Page 17 - Ship Types



T

H

M

T

H

R

M

A

T

## A.4 DEFAULT START ACTIONS

All ships have a default command that is executed on them when they are newly created. What this command is depends on the ship's class. The following table lists all the default start actions for each class:

<i>Class</i>	<i>Default Start Action</i>
M0	COMMAND_KILL_ENEMIES
M1	COMMAND_KILL_ENEMIES
M2	COMMAND_KILL_ENEMIES
M6	COMMAND_KILL_ENEMIES
M3	If player-owned, COMMAND_NONE. If ship has a leader, COMMAND_PROTECT protect it. If ship is being attacked, COMMAND_ATTACK on its attacker. If ship is a Xenon, Pirate, or Khaak, COMMAND_KILL_ENEMIES. Otherwise, COMMAND_RETURN_HOME
M4	If player-owned, COMMAND_NONE. If ship has a leader, COMMAND_PROTECT protect it. If ship is being attacked, COMMAND_ATTACK on its attacker. If ship is a Xenon, Pirate, or Khaak, COMMAND_KILL_ENEMIES. Otherwise, COMMAND_RETURN_HOME
M5	If player-owned, COMMAND_NONE. If ship has a leader, COMMAND_PROTECT protect it. If ship is being attacked, COMMAND_ATTACK on its attacker. If ship is a Xenon, Pirate, or Khaak, COMMAND_KILL_ENEMIES. Otherwise, COMMAND_RETURN_HOME
TS	If player-owned, COMMAND_NONE. If a home is set, COMMAND_RETURN_HOME. Otherwise, COMMAND_DOCKAT the nearest shipyard.
GO	If player-owned, COMMAND_NONE. Otherwise, COMMAND_PREACH.
UFO	COMMAND_PREACH
SPACEFLY	COMMAND_FOLLOW
All others	COMMAND_NONE

Table 8.9 - Default start actions

## A.5 OBJECT HIERARCHY

All objects that can be manipulated in scripting belong to a class. Each class can also itself be a member of another class, and so on. All classes “derive” from the most basic one called “Object”. The following chart illustrates the hierarchy of classes that all objects fall under:

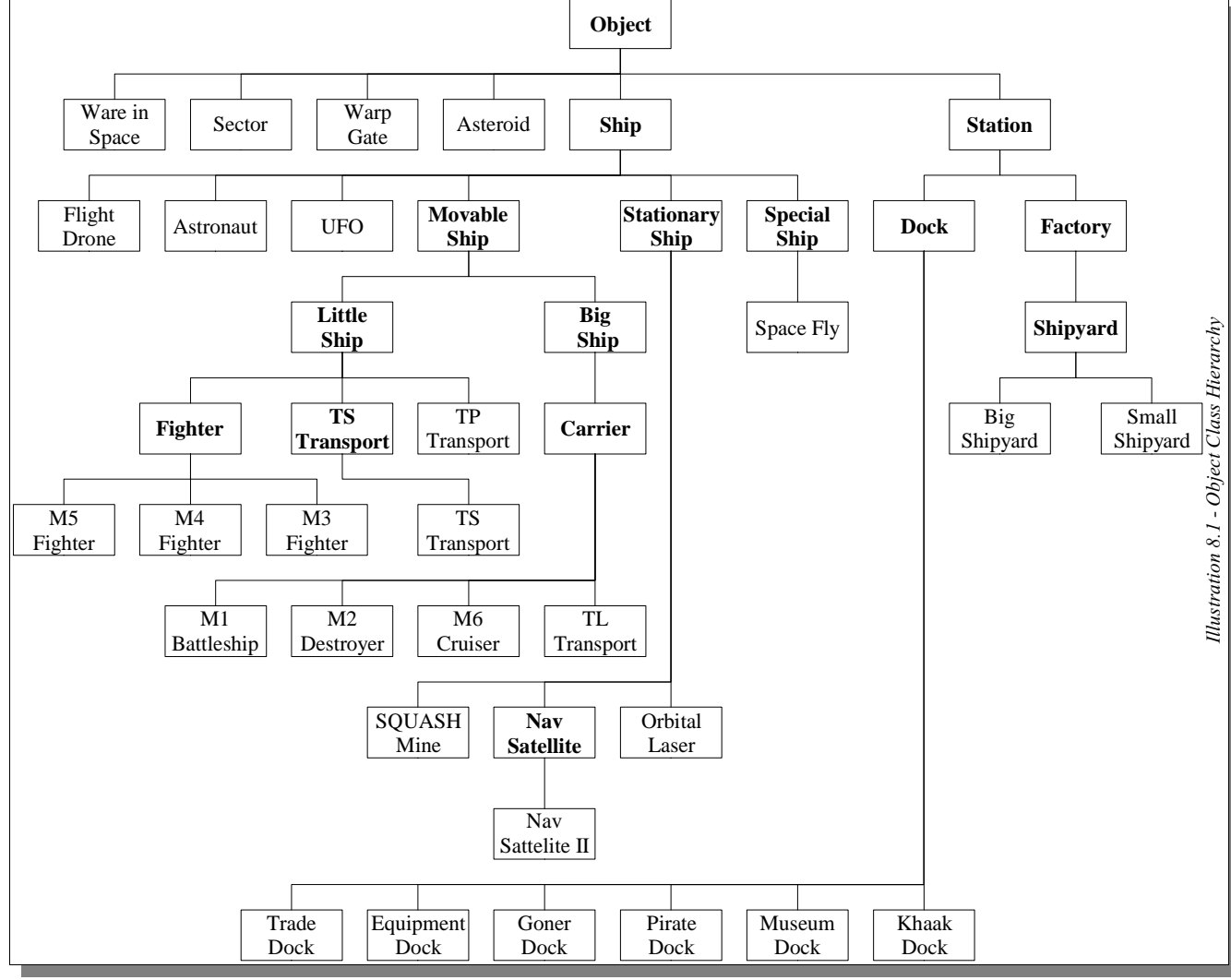
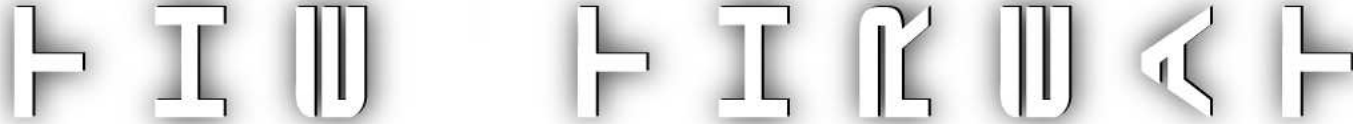


Illustration 8.1 - Object Class Hierarchy





## A.6 SEARCH FLAGS

There are five general-purpose searching instructions (some with more than one variant) that have a parameter called 'flags'. This parameter allows you to control the scope of the search. The flags are available in the “Select Constant” menu in the script editor and are listed below:

<i>Flag</i>	<i>Value</i>	<i>Description</i>	<i>Applies to:</i>
Find.IllegalWare	0x00800000	Find illegal wares	find flying ware
Find.Random	0x01000000	Choose a single item from all that match the search parameters	All
Find.Nearest	0x02000000	Choose the item closest to <i>refobj</i> single item from all that match the search parameters	All
Find.ExactJumps	0x04000000	Choose a result that is exactly <maxjumps> number of jumps away – no more, no less	<i>find station in galaxy</i>
Find.Enemy	0x08000000	Filter out all except enemies from the list unless Find.Neutral and/or Find.Friend are also specified	All
Find.Neutral	0x10000000	Filter out all except neutral (all that are not enemies and are not owned by the player) unless Find.Enemy and/or Find.Friend are also specified	All
Find.Friend	0x20000000	Filter out all except friends (player-owned) unless Find.Enemy and/or Find.Neutral are also specified	All
Find.Multiple	0x40000000	Allows multiple results – the instruction returns an array	All except <i>find station in galaxy</i>
Find.TypeAsWareCategorie	0x80000000	Not used	Not used

Table 8.10 - Search Flags

Not all flags can be used with all the instructions. See the “Applies to” column to determine what the restrictions are for any particular flag.

To use more than one flag in a search, OR them together as shown:

```
100 $SearchFlags = [Find.Enemy] | [Find.Multiple]
```



**A.7 ASTEROID TYPES**

There are ten different styles of asteroid that the *create asteroid* instruction can produce. All are shown below pictured with a Teladi Albatross for size comparison. Full-size images are available in the companion files to this book.











 Asteroid00	 Asteroid01	 Asteroid03
 Asteroid03	 Asteroid04	 Asteroid05
 Asteroid06	 Asteroid07	 Asteroid08
 Asteroid09		

Table 8.11 - Asteroid Types



**A.8 NEBULA TYPES**

There are 13 types of nebulae that the *create nebula* instruction can produce. All are shown below pictured with a Teladi Albatross for size comparison. Full-size images are available in the companion files to this book.








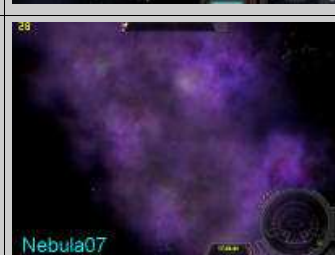





		
		
		
		
	<div>0. White gas, long, <b>with storms</b> 1. Small red, no storms 2. White gas, long, no storms 3. Red, multiple parts, <b>with storms</b> 4. Large white, no storms 5. Blue &amp; white, multiple areas, no storms 6. White swirls, red area <b>contains storms</b> 7. Huge purple, <b>with storms</b> 8. Large white swirls with red, <b>with storms</b> 9. Long bright blue, no storms 10. Large teal, <b>with storms</b> 11. Large teal, <b>with storms</b> 12. Small red, <b>with storm</b></div>	

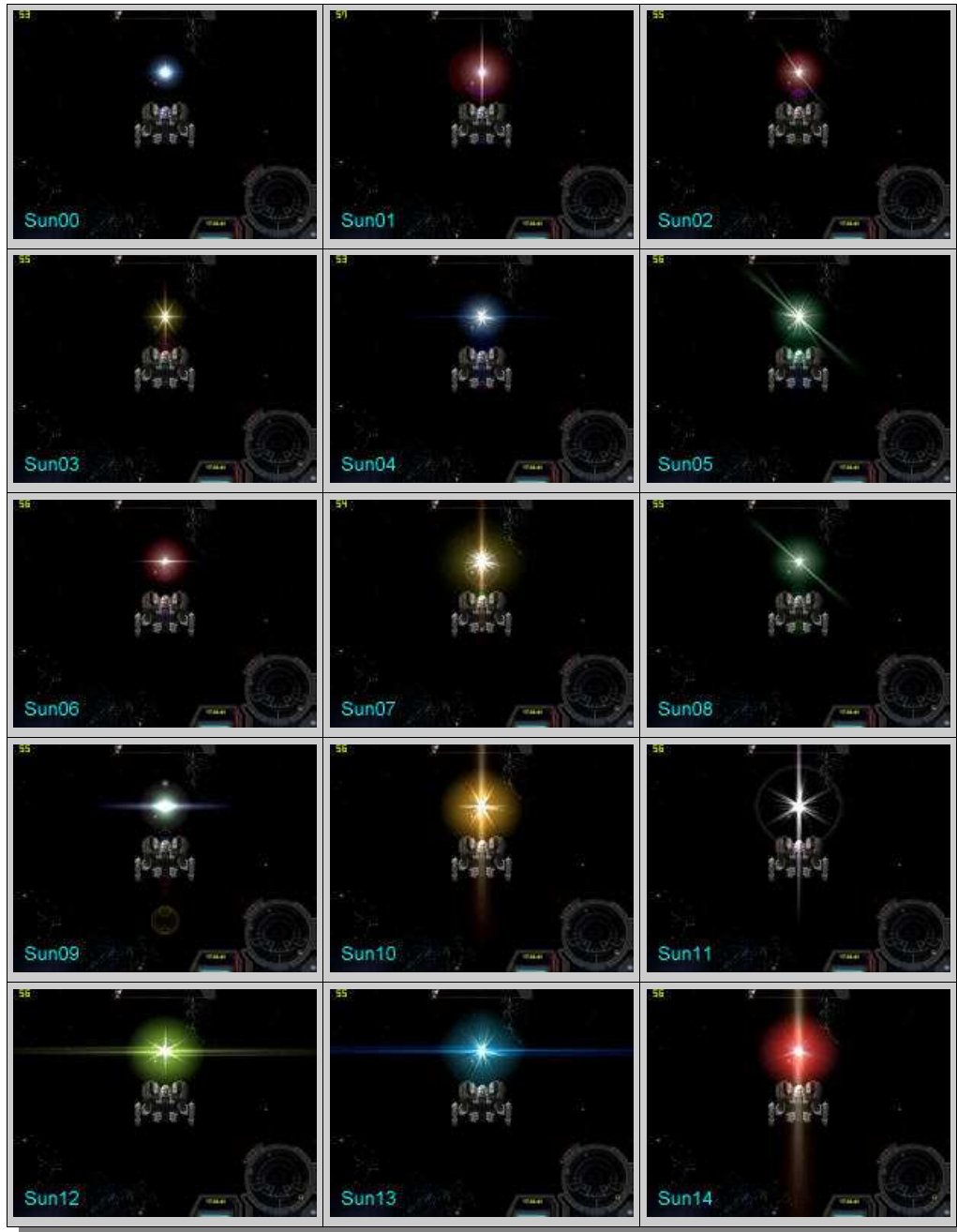
Table 8.12 - Nebula Types





## A.9 SUN SUBTYPES

There are 25 types of suns that *create sun* the instruction can produce. All are shown below. Full-size images are available in the companion files to this book.





# T H E X 2 M S C I P R O G R A M M E R S H A N D B O O K

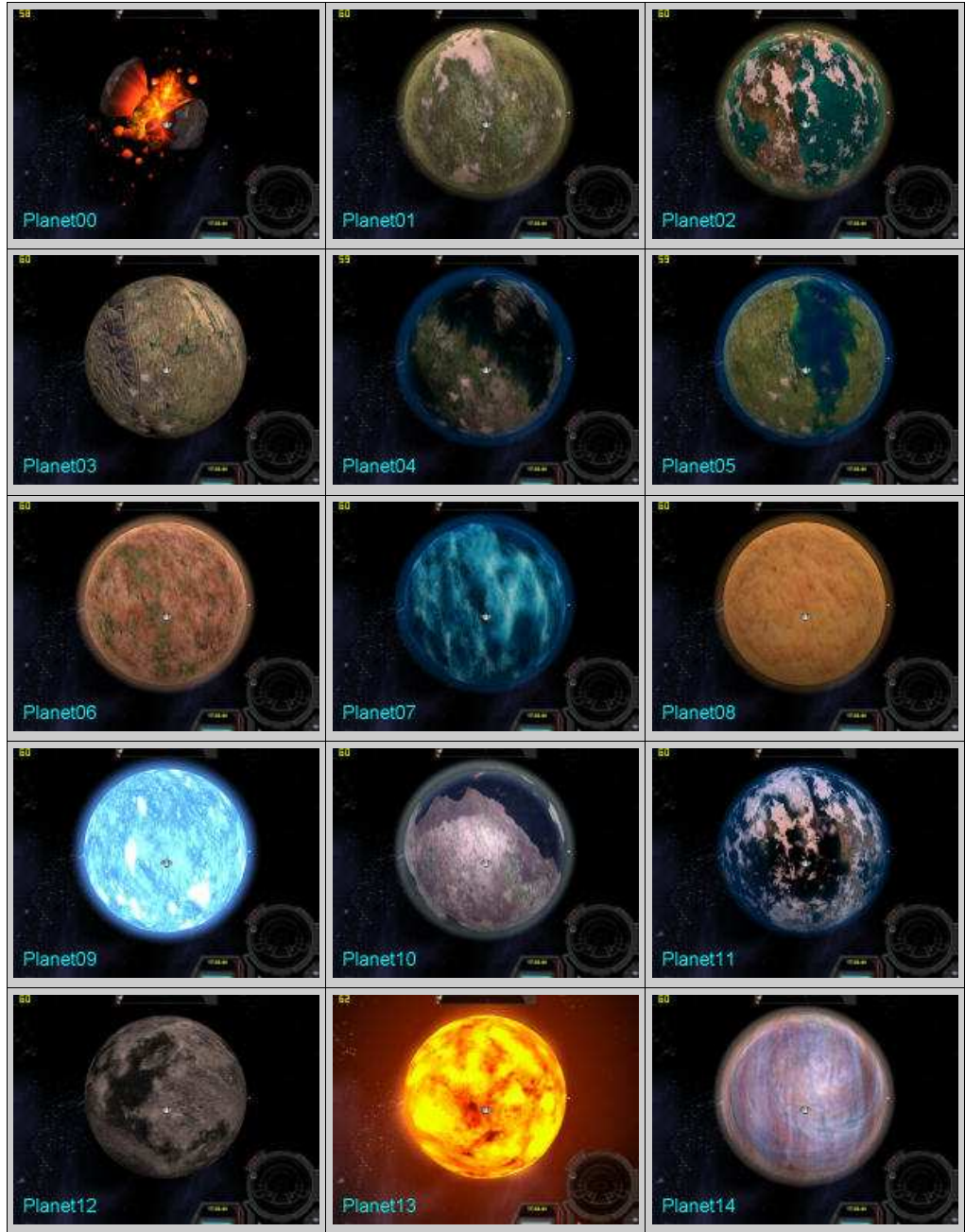


Table 8.13 - Sun Subtypes



## A.10 PLANET SUBTYPES

There are 17 types of planets that the *create planet* instruction can produce. All are shown below pictured with a Teladi Albatross for size comparison. Full-size images are available in the companion files to this book.





T

H

M

T

H

R

M

A

T

 A screenshot from a game showing a planet with purple and blue horizontal bands. The text "Planet15" is in the bottom left corner. A small "60" is in the top left corner. A circular UI element is in the bottom right corner.	 A screenshot from a game showing a planet with blue and white horizontal bands. The text "Planet16" is in the bottom left corner. A small "60" is in the top left corner. A circular UI element is in the bottom right corner.	
---	---	--

Table 8.14 - Planet Subtypes



## A.11 "SPECIAL" OBJECT TYPES

There are 74 types of special objects that can be produced with the *create special* instruction. Some objects have a spoken "name" that is announced by the on-board computer when they are targeted. Some do not.

The following chart lists all the objects, whether they or not they can be targeted, and what is spoken by the on-board computer when they are targeted.

### Special Object Descriptions

Type	Target	Board Computer	Description
0	Yes	Kessler Ring	Kessler Ring
1	Yes	Kessler Ring	Kessler ring
2	Yes	Unknown object	Ship part – front of Xenon K
3	Yes	Asteroid	Asteroid
4	Yes	Asteroid	Asteroid
5	Yes	Unknown object	Wall piece from X-Tension station construction animation
6	Yes	Unknown object	Framing X-Tension station construction animation
7	Yes	Unknown object	Wall piece from X-Tension station construction animation
8	Yes	Unknown object	Piece from X-Tension station construction animation
9	Yes	Unknown object	Shipyard clamshell
10	Yes	Unknown object	Plating
11	Yes	Unknown object	Sample sector map
12	Yes	Unknown object	Post-like object
13	Yes	Unknown object	Engine housing?
14	Yes	Unknown object	Translucent debris
15	Yes	Unknown object	Asteroid with a constructed tunnel
16	Yes	Unknown object	Gravidar (an actual working copy of your ship's gravidar)
17	Yes	Unknown object	Kessler ring
18	Yes	Unknown object	Kessler ring
19	Yes	Unknown object	Finish line?
20	No		Asteroid field
21	No		Asteroid field
22	No		Asteroid field
23	No		Asteroid field
24	Yes	Nacelle	Gate nacelle
25	Yes	Nacelle	Gate nacelle
26	Yes	Nacelle	Gate nacelle
27	Yes	Gate	Gate wreckage
28	Yes		Small asteroid
29	No		Unfinished or corrupted model - vertices improperly placed.
30	Yes	Ship debris	Debris of a Split Dragon (Kyle Brennan's destroyed ship)
31	No		Debris
32	No		Debris



# THE M I N I M A T

<i>Type</i>	<i>Target</i>	<i>Board Computer</i>	<i>Description</i>
33	No		Debris
34	Yes	Navigational beacon	Navigational beacon
35	No		Khaak debris
36	No		Debris
37	Yes	Unknown object	Khaak debris
38	Yes	Station debris	Destroyed station (President's End)
39	Yes	Station debris	Destroyed station (President's End)
40	Yes	Station debris	Destroyed station (President's End)
41	Yes	Station debris	Destroyed station (President's End)
42	Yes	Station debris	Destroyed station (President's End)
43	Yes	Ship debris	Debris of Argon Titan (model takes a very long time to load)
44	Yes		Cargo hatch with different details on both sides (two images below)
45	Yes		Window
46	Yes		Nothing visible
47	Yes		Internal tunnel
48	Yes		Cargo hatch (same as #44)
49	Yes		Ring
50	Yes		Tunnel
51	Yes		3d targeting reticle
52	Yes		Tunnel
53	Yes		Cargo hatch (same as #44)
54	Yes		Nothing visible
55	Yes		Tunnel
56	Yes		Nothing visible
57	Yes		Nothing visible
58	Yes		Nothing visible
59	Yes		Nothing visible
60	Yes		Cargo hatch (same on both sides)
61	Yes		Translucent purple sphere
62	Yes		Nothing visible
63	Yes		White cloudy spot - planar
64	Yes		Nothing visible
65	Yes		Red nav beacon
66	Yes		Cargo container
67	Yes		Nothing visible
68	Yes		Nothing visible
69	Yes		Cargo hatch (same as #44)
70	Yes		Nothing visible
71	Yes		Nothing visible
72	Yes		3d targeting reticle (same as #51)
73	Yes		Cargo hatch (same as #44)

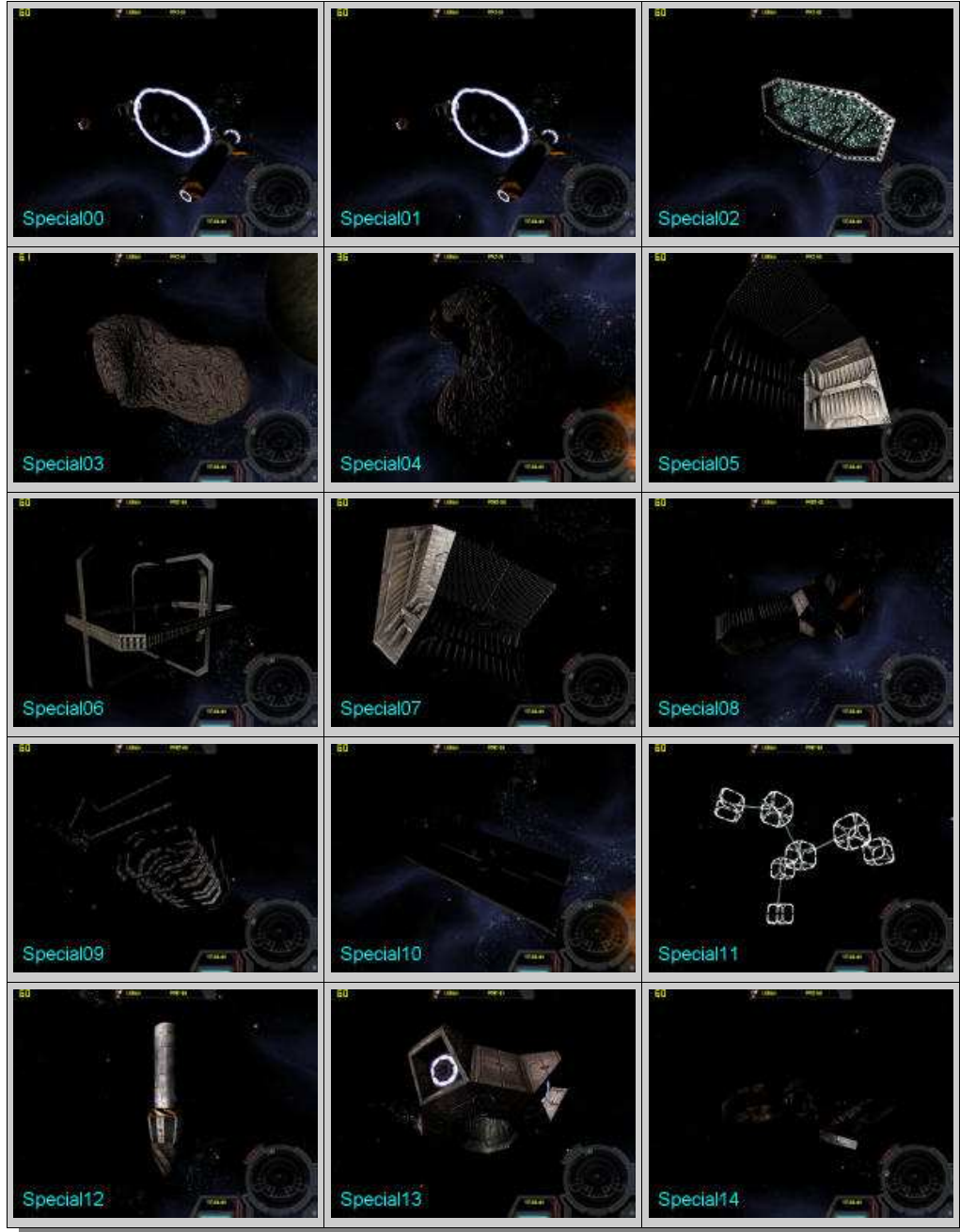
Table 8.15 - Description of Special Object Types





## Special Object Images

Full-size images are available in the companion files to this book.





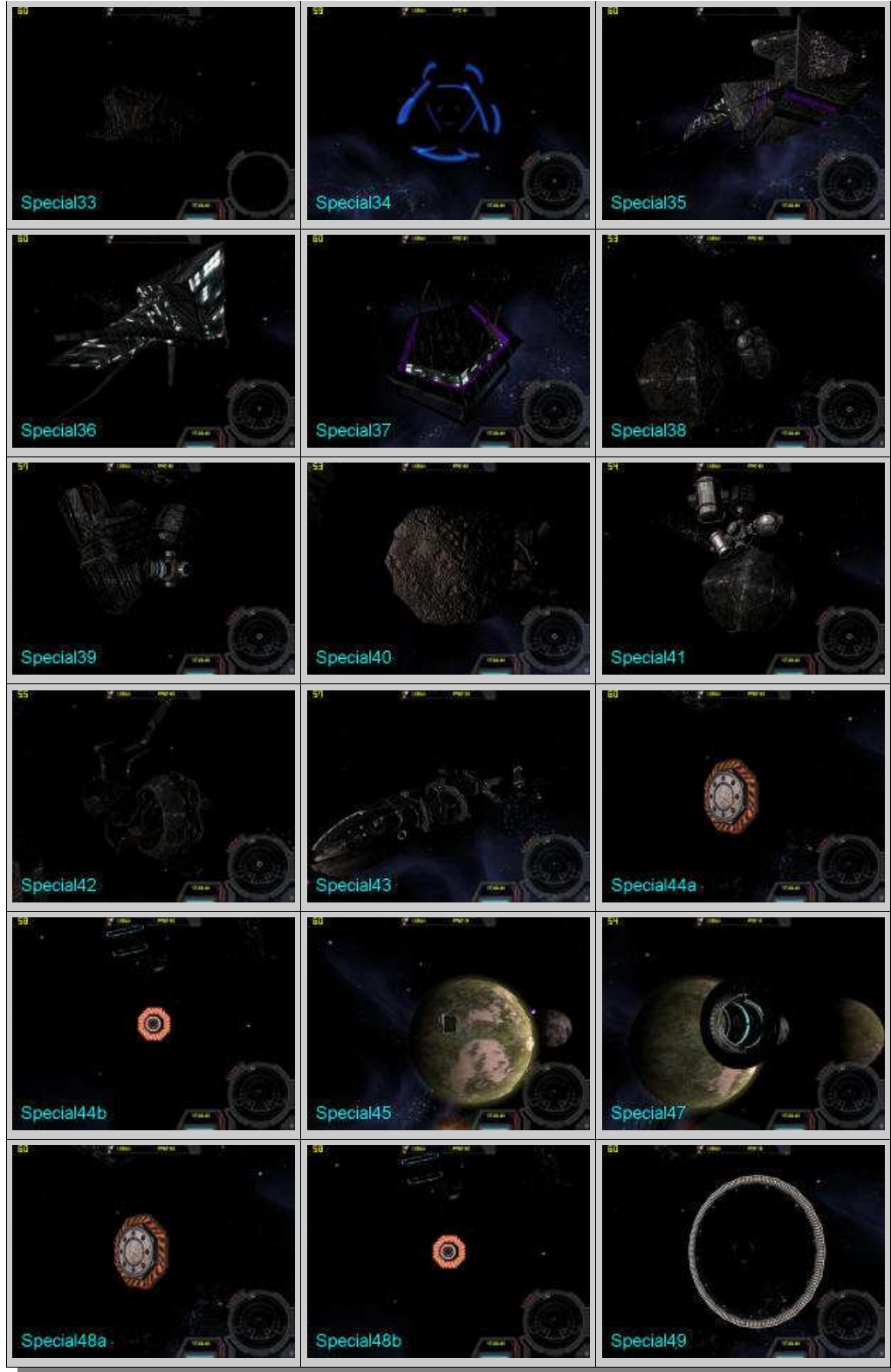
# THE ULTIMATE TUTORIAL





# THE ULTIMATE X² PROGRAMMER'S HANDBOOK

## The X² MSCI Programmer's Handbook





# THE ULTIMATE GUIDE















 Special50	 Special51	 Special52
 Special53a	 Special53b	 Special55
 Special60	 Special61	 Special63
 Special65	 Special66	 Special69a
 Special69b	 Special72	 Special73a
 Special73b		

Table 8.16 - Special Object Type Images



# Index

## A

add big ship, instruction.....	55
add default items to ship, instruction.....	58
add money to player, instruction.....	61
add money, instruction.....	60
add notoriety, instruction.....	74
add primary resource factory or dock, instruction.....	69
add product to factory or dock, instruction.....	69
add secondary resource factory or dock, instruction.....	69
add to formation with leader, instruction.....	49
add ware, instruction.....	61
al engine: register, instruction.....	41
al engine: set plugin description, instruction.....	41, 86
al engine: set plugin timer interval, instruction.....	41, 86
al engine: unregister, instruction.....	41
AL plugin.....	84
append to array, instruction.....	33
array alloc, instruction.....	32, 86
array assignment, instruction.....	32
arrays.....	19
array instructions.....	32
atomic operation.....	82
attack run on target, instruction.....	46

## B

best missile type for target, instruction.....	53
break, instruction.....	22, 31
buy ware to max price, instruction.....	57
buy ware, instruction.....	56

## C

call script, instruction.....	31, 83, 86
can be controlled by race logic, instruction.....	56
can execute StartAction, instruction.....	56
can sell ware, instruction.....	59
can transport ware, instruction.....	58
catch ware object, instruction.....	47
check, select and fire missile, instruction.....	55
Clear Debug Messages, menu item.....	8
clone array, instruction.....	32
command.....	5
command script.....	5
concurrency.....	80
connect ship command/signal, instruction.....	36
continue, instruction.....	22, 31
cooperative multitasking.....	80
copy array, instruction.....	32
create asteroid, instruction.....	69
create flying ware, instruction.....	73
create gate, instruction.....	68
create nebula, instruction.....	69
create planet, instruction.....	69
create sector object, instruction.....	73
create ship, instruction.....	68
create special, instruction.....	69
create station, instruction.....	68
create sun, instruction.....	69

## D

data types.....	18
debugging.....	25
debugging.....	
Clear Debug Messages, menu entry.....	8
Script Debugger Menu, menu entry.....	8
Script Debugging, menu entry.....	8
dec, instruction.....	34, 81
decouple ships, instruction.....	54
defensive move, instruction.....	46
define label, instruction.....	31
destruct, instruction.....	72

## E

else, instruction.....	31, 40
enable signal/interrupt handling, instruction.....	36
end, instruction.....	5, 31, 33, 40
escort ship, instruction.....	47
exists SectorObject, instruction.....	72
exists, instruction.....	63
expressions.....	33

## F

find asteroid, instruction.....	70
find best missile for target, instruction.....	53
find enemy in firing range of turret, instruction.....	53
find flying ware, instruction.....	70
find nearest enemy ship, instruction.....	45
find nearest enemy station, instruction.....	45
find nearest missile aiming at me, instruction.....	54
find random sector, instruction.....	76
find ship, instruction.....	70
with homebase.....	75
find station (product resource sells resource), instruction.....	61
find station in galaxy, instruction.....	64
multiple stations.....	75
find station, instruction.....	71
fire lasers on target, instruction.....	45
fire missile, instruction.....	53
fits laser into turret, instruction.....	54
fly to home base, instruction.....	44
fly to sector, instruction.....	45
fly to station, instruction.....	44
follow object, instruction.....	49
free sector object, instruction.....	73

## G

get amount of ware in cargo bay, instruction.....	40, 60
get asteroid array, instruction.....	77
get attack target, instruction.....	50
get average price of ware, instruction.....	59
get best store amount of ware, instruction.....	59
get cargo bay size, instruction.....	57
get command target, instruction.....	51
get command target2, instruction.....	52
get command, instruction.....	51
get current galaxy flight timestep, instruction.....	55
get current laser strength, instruction.....	65
get current missile, instruction.....	53
get current shield strength, instruction.....	65
get datatype, instruction.....	38
get destination, instruction.....	50
get distance between, instruction.....	67
get distance to, instruction.....	67
get dock array, instruction.....	76
get dock bay size, instruction.....	74
get environment, instruction.....	63
get factory array, instruction.....	76
get follow mode, instruction.....	50
get formation follower ships, instruction.....	49
get formation leader, instruction.....	49
get free amount of ware in cargo bay, instruction.....	60
get free volume of cargo bay, instruction.....	57
get free volume of ware in cargo bay, instruction.....	58
get global variable, instruction.....	41, 86
get homebase, instruction.....	63
get hull percent, instruction.....	74
get hull, instruction.....	74
get id code, instruction.....	75
get jumps from sector to sector, instruction.....	64
get khaak aggression level, instruction.....	42
get laser type in bay, instruction.....	66
get laser type in turret, instruction.....	54
get local variable, instruction.....	40, 83
get maintype of ware, instruction.....	62

get maintype, instruction.....	62
get max amount of ware that can be stored in cargo bay, instruction.....	60
get max hull, instruction.....	74
get max laser strength in turret, instruction.....	65
get max missile type that can be installed, instruction.....	65
get max number of lasers in turret, instruction.....	54
get max sectors in x/y direction, instruction.....	76
get max shield type that can be installed, instruction.....	65
get max speed, instruction.....	74
get max store amount of ware, instruction.....	59
get max trade jumps, instruction.....	60
get max upgraded speed, instruction.....	74
get max upgrades for upgrade, instruction.....	74
get max ware transport class, instruction.....	57
get maximum laser strength, instruction.....	65
get maximum shield strength, instruction.....	65
get min max average price of ware, instruction.....	62
get missile fire probability, instruction.....	54
get missile fire time difference, instruction.....	54
get money, instruction.....	60
get name, instruction.....	75
get next sector on route, instruction.....	65
get north south east west warp gate, instruction.....	76
get notoriety from race to race, instruction.....	66
get notoriety to race, instruction.....	66
get number of landed ships, instruction.....	74
get number of laser bays, instruction.....	65
get number of primary resources, instruction.....	59
get number of resources, instruction.....	59
get number of secondary resources, instruction.....	59
get number of shield bays, instruction.....	65
get number of subtypes of maintype, instruction.....	62
get number of turrets, instruction.....	54
get object class, instruction.....	63
get object from SectorObject, instruction.....	72
get owner race, instruction.....	63
get pid, instruction.....	36
get pilot name, instruction.....	75
get player money, instruction.....	61
get player owned ship array, instruction.....	77
get player owned station array, instruction.....	77
get player ship, instruction.....	68
get position as array, instruction.....	68
get price of ware, instruction.....	59
get product ware, instruction.....	59
get random name, instruction.....	42
get range of missile, instruction.....	54
get relation to object, instruction.....	66
get relation to race, instruction.....	66
get rot alpha beta gamma, instruction.....	73
get script command target, instruction.....	38
get script command upgrade, , instruction.....	37
get script command, instruction.....	37
get script name, instruction.....	41
get script priority, instruction.....	34
get script version, instruction.....	41, 86
get sector from universe index, instruction.....	76
get sector, instruction.....	63, 64
get SectorObject ID, instruction.....	72
get serial name of station, instruction.....	64
get shield and hull percent, instruction.....	74
get shield percent, instruction.....	74
get shield type in bay, instruction.....	66
get ship array, instruction.....	76
get size of object, instruction.....	74
get station array, instruction.....	76
get subtype of ware, instruction.....	62
get subtype, instruction.....	62
get tactical, instruction.....	52
get task ID, instruction.....	36
get total volume in cargo bay, instruction.....	58
get tradeable ware array from station, instruction.....	60

get transport class of ware, instruction.....	61
get true amount of ware in cargo bay, instruction.....	60
get true volume of ware in cargo bay, instruction.....	58
get universe x/y index, instruction.....	76
get volume of ware in cargo bay, instruction.....	57
get volume of ware, instruction.....	61
get wanted ware count, instruction.....	58
get wanted ware, instruction.....	58
get ware from maintype and subtype, instruction.....	62
get ware type code of object, instruction.....	63
get ware type of SectorObject, instruction.....	72
get warp gate, instruction.....	76
get x y z position, instruction.....	68
give formation leadership, instruction.....	52
global script map, instruction.....	82
global script map: remove key, instruction.....	37
global script map: set key, instruction.....	37
Global Script Tasks, menu item.....	8
goto, instruction.....	23, 31
H	
has a free big ship dock slot, instruction.....	56
has formation ships, instruction.....	52
has illegal ware onboard, instruction.....	58
has same environment as, instruction.....	66
I	
if, instruction.....	20, 33, 83, 86
ignore ship command/signal, instruction.....	36
inc, instruction.....	33
infinite loop detection enabled, instruction.....	39
init scripts.....	11
insert into array, instruction.....	32
install ware, instruction.....	61
instruction.....	5
interrupt task, instruction.....	35
interrupt with script (no arguments), instruction.....	36
interrupt with script, instruction.....	36, 81
interrupts.....	81
interrupts.....	
interrupt task.....	35
interrupt with script.....	36
is signal/interrupt handling on.....	37
is a enemy, instruction.....	66
is a friend, instruction.....	66
is datatype, instruction.....	38
is decoupled ships leader, instruction.....	54
is disabled, instruction.....	71
is docked, instruction.....	67
is docking allowed, instruction.....	67
is docking possible, instruction.....	67
is in a sector, instruction.....	67
is in firing range of turret, instruction.....	52
is in same sector as, instruction.....	67
is landed, instruction.....	67
is landing, instruction.....	55
is missile ready to fire, instruction.....	53
is neutral to me, instruction.....	66
is of class, instruction.....	63
is of type, instruction.....	64
is player wingman, instruction.....	56
is plot state flag, instruction.....	41
is script with prio on stack, instruction.....	35
is sector known by the player, instruction.....	76
is signal/interrupt handling on, instruction.....	37
is starting, instruction.....	55
is ware illegal, instruction.....	62
K	
kill sector object, instruction.....	73
L	
launch fight drones, instruction.....	53
load text, instruction.....	39
load ware, instruction.....	57



<b>M</b>	
move around, instruction.....	47
move to position, instruction.....	51
move to ware object, instruction.....	46
<b>N</b>	
needed jumpdrive energy for jump, instruction.....	55
<b>O</b>	
only player owned ships can trade with, instruction.....	60
<b>P</b>	
parameter.....	6
play sample, instruction.....	42
play sample: incoming transmission, instruction.....	42
player loses police license, instruction.....	74
playing time, instruction.....	39
plugin.....	5
processes.....	78
put into environment, instruction.....	75
<b>R</b>	
random value, instruction.....	34
read text, instruction.....	38
Reinit Script Caches, menu item.....	8, 87
remove element from array, instruction.....	33
remove from any formation, instruction.....	49
remove primary resource from factory or dock, instruction.....	69
remove product from factory or dock, instruction.....	69
remove secondary resource from factory or dock, instruction.....	69
resize array, instruction.....	33
return, instruction.....	32, 40
<b>S</b>	
script.....	5
Script Debugger Menu, menu item.....	8
Script Debugging, menu item.....	8
Script Editor, menu item.....	7
script engine version, instruction.....	34
select new formation leader, instruction.....	52
sell ware, instruction.....	57
send audio message, instruction.....	42
send incoming message to player, instruction.....	43
send signal, instruction.....	49
set as player wingman, instruction.....	56
set attack target, instruction.....	50
set attacker to, instruction.....	67
set command target, instruction.....	51
set command target2, instruction.....	52
set command, instruction.....	51
with target, target2, par1, and par2.....	53
set destination, instruction.....	50
set follow mode, instruction.....	50
set formation, instruction.....	48
set global variable, instruction.....	41, 86
set homebase, instruction.....	65
set khaak aggression level, instruction.....	42
set local variable, instruction.....	40, 83
set missile fire probability, instruction.....	54
set missile fire time difference, instruction.....	54
set name, instruction.....	75
set owner race, instruction.....	75
set pilot name, instruction.....	75
set pilot speaker, instruction.....	75
set position of sector object, instruction.....	72
set position, instruction.....	72
set price of ware, instruction.....	59
set race logic control enabled, instruction.....	56
set relation, instruction.....	68
set rotation of sector object, instruction.....	73
set rotation, instruction.....	72
set safe position of sector object, instruction.....	72
set script command target, instruction.....	38
set script command upgrade, instruction.....	37
with check script.....	39
set script command, instruction.....	37, 79
set script priority, instruction.....	35
set serial name of station, instruction.....	64
set ship command/signal to global default behaviour, instruction.....	36
set ship disabled, instruction.....	75
set StartAction enabled, instruction.....	56
set state of news article, instruction.....	39
set tactical, instruction.....	52
set wanted ware count, instruction.....	58
set wanted ware, instruction.....	58
setup scripts.....	11, 29, 87
should a missile be fired, instruction.....	53
signals.....	81
connect ship command/signal.....	36
default handler scripts.....	82
enable signal/interrupt handling.....	36
global script map.....	37
ignore ship command/signal.....	36
is script with prio on stack.....	35
is signal/interrupt handling on.....	37
send signal.....	49
set ship command/signal to global default behaviour.....	36
size of array, instruction.....	32, 81
skip if, instruction.....	20
speak text, instruction.....	43
sprintf, instruction.....	38, 86
START command, instruction.....	49
start sector object, instruction.....	73
start task, instruction.....	35, 79, 89
state of news article, instruction.....	39
statement.....	5
station send defend squad, instruction.....	75
station trade and production tasks, instruction.....	71
switch laser in turret, instruction.....	58
syntax.....	6
system date is, instruction.....	39
<b>T</b>	
tasks.....	78
trades with ware, instruction.....	59
turn turret, instruction.....	45
<b>U</b>	
unload ware, instruction.....	57
use jumpdrive, instruction.....	55
uses ware as primary resource, instruction.....	59
uses ware as secondary resource, instruction.....	59
<b>V</b>	
variables.....	17
assignment.....	33
<b>W</b>	
wait randomly, instruction.....	34, 89
wait, instruction.....	5, 34, 79, 80, 83
while, instruction.....	5, 21, 33, 80, 81
write to logfile, instruction.....	44
write to player logbook, instruction.....	43
<b>X</b>	
x2tool.....	10