BCG/UC GBS-III SCRIPTING README

TABLE OF CONTENTS

1.1 SCRIPTING CONCEPTS

Writing a mission script using the Game Builder System is a very simple process once you know what you are doing. Things to bear in mind are:

(a) know the world layout. The only way you are going to get a clear understanding is to print out the maps of the galactic regions. You will also need to print out the navigation links because you will need these in
order to reference objects starting points in the region

(b) know the alien nation relationships. For this, you would need to read
the manual. A lack of understanding in this area can and will lead to scripts that are illogical and which will fall through. For instance, the
Terrans are hostile toward the Gammulans especially the 'military' castes.
So if you script two ships of this race/caste in the same region, regardless of what your script does, a combat situation will ensue (c) you
also need to print out the list of objects (stations, ships etc), the trading database, the medals distribution etc.

Using the GBS, you can create single missions or a collection of missions
making up a scenario. You can also create scenarios with branching or linear storylines. There is no limit to the number of missions that you

can
create nor the number of missions that a scenario can contain. The longer
the script, the longer it will take to parse and interpret.

You also need to think logically. This is not your everyday run of the mill
system. The GBS allows you to do very powerful things within the game
world. You can alter the world itself (which is not covered or allowed in
this 'lite' version of the GBS), the objects in it etc. There is no GUI
here. You will be using a text editor to script the mission and then use a
tool to parse and play it.

All objects in the game are either DYNAMIC or STATIC. The former are
objects that can move, make decisions, respond to external AI instructions
etc. Examples would be a starstation or ship. These are called 'actors'.
The latter are objects that cannot move nor respond to external AI
instructions. Examples would be planets or jump anomalies.

1.2 RACE AND CASTE DISCUSSION

There are twelve 'known' races in the game world and 22 castes (see
the game appendix file).

The AI engines directly affect the race and caste relations. Though the
world is initialized with a default alliance matrix (see OBJECTS.TXT), you
can override this in your script. However, you only have a high level
access and *not* a low level one. For instance, though the Gammulans are
hostile to the Terrans by default, using a script command (MOD_ALLIANCE),
you can make them friendly. They can organize cookouts or go to powerball
games together. However, a Terran raider is going to attack a Gammulan
diplomat regardless of what the race relation is, leaving it up to the
decision makers to sort it out. The castes themselves interact with each
other based on their internal AI goals. A TERRAN MILITARY ship is going to
ignore a GAMMULAN DIPLOMAT but will attack a GAMMULAN MILITARY ship. A
TERRAN EARTHCOM ship is going to attack a TERRAN INSURGENT ship. A TERRAN
POLICE unit is going to attack a GAMMULAN RAIDER regardless of the race
relationship. It gets more complex at lower levels but we don't need to
bother with that right now. These relationships are what make it possible
for the world to continue operating regardless of what you are doing. A
station that is scheduled to launch a patrol (or invasion) into a hostile
region only needs to send the units (actors) to the region and their
internal AI will take over. No additional work is involved. Even when
scripting, to create a skirmish, all you have to do is introduce a unit to
the region and that's it. You can script a GAMMULAN MILITARY ship to appear
in the Earth region and GALCOM HQ will immediately launch a fighter wing to
take it out. If the ship is within range, it will also launch missiles or
activate it's turret firing system. You don't have to send any script

instructions to GALCOM HQ or the ship in question. If the ship is a carrier, then it too will launch it's own fighters to engage the fighters launched at it. If a scripted ship lays mines in Earth region and GALCOM HQ detects the mines, it will launch a wing of fighters with MINESWEEP order to remove them.

An understanding of the race/caste relation is vital to the integrity of the scripts. Using the powerful option of actually altering these relations, you can alter and tip the balance of power at a galactic scale.
You may for instance, via a campaign consisting of several scripted missions and scenarios, create a diplomatic situation between one race and another, leading all the way up to outright hostility based on the outcome and resolution of the scripted missions. For instance you could script an ship to deliver an object to another ship somewhere. If the object is not delivered, you can then alter the relationship slightly between the two races etc.

1.3 THE GBS PARSER

The scripting system is a high level interpreted language that is executed at runtime. The PREPARE parser takes the script, parses it (like a language compiler such as C++), checks for errors and produces a binary file that the interpreter built into the game engine, uses at runtime. The parser has no knowledge of how illogical or inconclusive your scripts are. It doesn't care either. So, just because the script parses OK, does not mean that all is well and good. You should be able to read your script and tell if it's logical or not. Then, once its in the game, you have to pray that some AG actor is not going to render your script pointless (as previously discussed). During the parsing of a script, the parser will halt at any errors it finds and provide a line number that the error occured on.

All scripts must be created from the folder in which the parser resides.

The last line of EVERY script file MUST have the $ character. This is used to signal and end of file condition.

1.4 LAYOUT AND SYNTAX

When writing a script, use indentations as well as a combination of CAPS and lowercase so that the logic jumps out at you. When nesting IF blocks, indent them so that, again, they are readable. Here is a script segment that illustrates this. Notice the indentation and the use of CAPS and lowercase syntax. As you become an expert (yeah, right), you will be able to look at a script block and immediately notice the logic and/or

errors.

The parser is not case sensitive so 'START' is the same as 'start', 'IF' is
the same as 'if' and so on. If you create a script that is hard to read, you won't be able to make sense of it, nor will the person you are sending
it off to.

Though the maximum line length is 128 characters, if you have a line of script that exceeds the screen display, i.e., line 79, it will be hard to
read and understand. All script commands MUST be on their own line. There
is no line continuation.

SAMPLE SCRIPT EXCERPT:

```
[dynamic]
vesperon diplomat,,f3,mrt15.3dc,DIPLOMAT3
[ai]
.f3,100,100,100,100,100,100,0
start 120000 near canaanz^jmp-33 2000 jmp-32
stop
events 101
!startup
   jump majorisz^wrm-23
   broadcast "DIPLOMAT3 LAUNCHED"
!under_attack this
   broadcast "We are under attack!"
   reset_under_attack this
!destroyed this
   say TO,"Vesperon diplomat3 has been destroyed"
!arrived majorisz
   IF EXIST majoris_s THEN
    broadcast "Executing docking approach to station Majoris"
    dock majoris_s
   ELSE
    broadcast "Station Majoris not operational, returning to Eridani"
    IF EXIST eridani THEN
     dock eridani
    ELSE
     flee
    ENDIF
   ENDIF
!docked majoris_s
    say CMO,"Vesperon diplomat3 has docked at station Majoris"
endevents
```

1.5 OBJECT IDENTIFIERS

Use descriptive names for all object identifiers. When you are writing a script, avoid id abbreviations such as 'f3' used above unless you are an expert and can keep track of this stuff without scrolling the page. If you
are scripting multiple ships, then names like enemy_ship1, friendly_ship2
are more descriptive than e1 and f1. The max id length is 128 characters.
If you are creating wings, you can then use friendly_wing1 etc.

NOTE: You *CANNOT* use duplicate identifier names in the same script,

even
if you are breaking them up. Therefore, 'enemy1' can only be used once. If
you have, for instance, two files that are joined together using #include,
this rule also applies because all the files are parsed as one file.

1.6 LOGICAL LAYOUT & ORGANIZATION

Use comment lines # to separate scripted entities. This also improves readability. When possible, include comments in areas where there is no clear indication as to what is happening.

Augmented blocks such as [sets] and [event] should always be at the end of
the file. This also improves readability.

If you are writing a campaign, break it up into smaller script files and then use the #include directive to connect them. For instance, you may have
a 20 mission scenario and you can break it into 4 files each with 5 missions and name them sensibly. In this example, we assume that the 'master' script is called BCIA0201.SCR and that each of the other files each contain their own mission blocks.

NOTE: The #include directive MUST be on column 1 of the line!

You would have BCIA0100.SCR, TOD201.SCR, TOD202.SCR, TOD203.SCR and from within BCIA0100.SCR (usually after the comment) you would include the smaller files. You can use any filename convention you like.
The parser will first parse all the included files and then move into the
current file and parse them. This also means that you can also #include files at the END of the master file. This would be the case if the first file, BCIA0100.SCR contains 5 missions and the subsequent files contain 5
each in sequence. Of course the master file doesn't have to contain any scripts at all. It can simply contain the description of the scenario script and lines which include the missions themselves.

```
#
# The BCIA0100.SCR master file will include 3 other smaller script files
#
#include tod201.scr
#include tod202.scr
#include tod203.scr
#
```

As your scripts get larger, they will become more and more difficult to read. By grouping object entities and blocks together, you will be able to
read and understand them more clearly. In example1, all the objects are identified at the beginning of the script. In example2, they are identified
when and where needed. This probably won't make sense until you start scripting four or more entities in a script.

e.g. #1

```
[dynamic]
vesperon diplomat,,f3,mrt15.3dc,DIPLOMAT3
vesperon diplomat,,f4,mrt15.3dc,DIPLOMAT4
```

```
[ai]
.f3,100,100,100,100,100,100,0
start near canaanz^jmp-33 2000 jmp-32
stop
events
!startup
   broadcast "DIPLOMAT3 LAUNCHED"
endevents
#
.f4,100,100,100,100,100,100,0
start near canaanz^jmp-33 5000 jmp-32
stop
events
!startup
   broadcast "DIPLOMAT4 LAUNCHED"
endevents
```

e.g. #2

```
#
# Vesperon diplomat is travelling to Majoris/Alpha Majora
# where it will meet up with another Vesperon diplomat
#
[dynamic]
vesperon diplomat,,f3,mrt15.3dc,DIPLOMAT3
[ai]
.f3,100,100,100,100,100,100,0
start near canaanz^jmp-33 2000 jmp-32
stop
events
!startup
   jump majorisz^wrm-23
   broadcast "DIPLOMAT3 LAUNCHED"
!detect f4
   broadcast "DIPLOMAT4 detected on radar!"
endevents
#
# The ship will patrol Majoris until diplomat3 arrives
#
[dynamic]
vesperon diplomat,,f4,mrt15.3dc,DIPLOMAT4
[ai]
.f4,100,100,100,100,100,100,0
start near canaanz^jmp-33 5000 jmp-32
stop
events
!startup
   jump majorisz^wrm-23
   broadcast "DIPLOMAT4 LAUNCHED"
!arrived majorisz
   patrol majorisz
   broadcast "DIPLOMAT4 starting patrol pattern"
!detect f3
   broadcast "Hey guys, about time!"
endevents
#
```

1.7 GAME SYSTEM FILES

All your master scripts MUST contain the system files that the world needs
to run. You should ensure that the following block is in your master

file
after the description. You do NOT need to put these in files that you have
broken up into smaller parts. ONLY the master file needs this. See the included script examples. In the example above, this block of text would
only be in the main script file and not the smaller #included files. It is preceeded by your scenario description and followed by your #include directives if you are including other smaller script files (see above)

e.g.

```
[dynamic]
#include glob_dyn.scr
#
[ai]
#include glob_ai.scr
#include glob_ag.scr
```

1.8 CREATING A MISCON DESCRIPTION FILE

MISCON description files are also ascii text files that have a .DES extension. These are used to describe the basic premise of the scenario created. This is the file that is displayed when the user selects a scenario in MISCON.

Each line should be no more than 80 characters wide and each page should not exceed 30 lines or the display will overflow into the graphics area.

The last line of the file should be a carriage return or parsing will fail.

Like .MIS files, the .DES description file must be copied over to the SCRIPTS folder where it can be accessed by MISCON at runtime.

1.9 FILENAME CONVENTIONS

The GBS adheres to strict scenario naming conventions. Whatever you do, do NOT use the same file naming convention used by the game.

When naming your scripts, do NOT use the naming convention used by game scripts!! e.g. BCIA0101 to BCIA0125 are a game scripts, so you should start
your numbering from BCIA0201 to BCIA0299, BCIA0301 to BCIA0399 etc. This applies to all script types (ROAM, ACM, IA, TA) but each have their own naming
conventions.

2.0 GBS FILE TYPES EXPLAINED

You will be working with the following file types, so I may as well explain
them briefly.

1. SCR / MIS

   When you parse a source .SCR script file, the resulting file is the binary
   version with a .MIS extension. This is the file that the game runs.

2. DES

When you create a .DES description file, it is displayed, as is, when MISCON starts. The system checks for scenarios in the resource files and the SCRIPTS sub-folder and will display those that it finds. It will
use the most recent file if a file exists in that folder and in the resource files. If you create a mission with no description file, it will not show up in MISCON and therefore cannot be played.

3. TXT / DAT

The game has multi-language support built in but was never used. The parser
was also updated to support this system. Therefore, when you write a script, all strings scripted using the SAY and BROADCAST commands as well
as those within the [ACM] block, are stored in a .TXT text file. Each time
you build a script, you should run the parser with the 'phrases' parameter.
This creates a .TXT file and a .DAT file. The former contains the actual
strings and the latter contains the parsed version that the interpreter uses.

4. GLOB_AG.SCR, GLOB_AI.SCR, GLOB_DYN.SCR, IA_AG.SCR, IA_AI.SCR, IA_DYN.SCR
OBJDEFS.SCR, OBJCLASS.SCR, OBJCLASS.LST, STD_ID.HPP, TEMPLATE.HPP, WORLD.SCR

There are several system files that you MUST not be altered in any shape
or form. These files are used by the parser to set up internal variables. If these files are altered, not only will your play world be
invalidated, the parser won't even parse your script. You have been warned.

ONLY modify the OBJDEFS.SCR and OBJCLASS.SCR files if you know what you are doing. And even then, you will NOT be able to run the scenarios
which shipped with the game. You will only be able to run your own scenarios
which make use of these modified files.

2.1 PLANNING AND CREATING SCRIPTS

Unless you are just experimenting or modifying the sample script to your heart's content, there will come a time when you are going to lose it and
attempt to create the best mission script you have ever seen. I sure hope
you have a lot of hair left on your head. Start small, think small and progress. The GBS is a very powerful system and you are going to get carried away long before you even get your first script to run. Be patient.

To design, write and run a good script (no script is perfect), you have to

1. Think about what you want to achieve in the script.

2. Look at the maps and figure out where it is going to take place.

3. Make a note of any hostile forces operating in the regions that the script takes place in. Such as the Insurgent station in Sygan.

4. Make a note of the current race/caste relations. Make a note of what kind of objects you are going to be using in the script.

5. Think up a descriptive short scenario briefing for display in MISCON where you get to choose the script you are about to run.

6. Think up short and descriptive mission orders that are going to be displayed in the COMMLINK computer when the particular scenario starts. Remember that the if you are creating an ACM scenario, whatever
you have the .DES file, should also be in the [ACM] block because when
the gamer does to COMMLINK, it is the [ACM] block description that is
displayed, NOT the one in the .DES file (that one is only displayed in MISCON).

2.2 EASY STEPS FOR CREATING/PLAYING A MISSION SCRIPT

1. Create the mission script in text editor and give it a name based on the convention already discussed.

2. Create the MISCON description file and give it a name based on the convention already discussed.

3. Parse the script using PREPARE in the form, 'PREPARE <filename.scr>'

4. Generate the strings file i.e 'PREPARE PHRASES <filename.scr>'

5. Copy the parsed script file, the description file and the language data file, to the SCRIPTS sub-folder in your game runtime environment.

6. Run the game, select the scenario from the list in MISCON and play it!

   Once you start the game, the description file, will be displayed in MISCON.
   Select it and launch to run your script. You should then proceed to the
   area where your script takes place and watch it in action.

2.3 DISTRIBUTING SCRIPTS

To share scripts with your friends, send them the entire script set. A script set consists of the .SCR, .DAT, .TXT and .DES files. All they have
to do is put the files in their SCRIPTS folder and run the game. Of course,
since they have the source (.SCR file), they can make any modifications they wish, to the script and build their own variance.

WARNING: There WILL be set name conflicts for instance if you send someone
a BCIA0201 set and they already have their own or one from another source.
The user will simply have to rename all the files to a vacant 'slot' on his

system. So for instance if you send them the BCIA0201 set and they already
have one, but have the BCIA0501 'slot' free, all they would have to do is
rename all the files. For example 'ren BCIA0200*.* BCIA0501*.*' will do the
trick. Of course they can overwrite any script set that they no longer wish
to keep.

Foreign language speakers can edit the .DES and .TXT files to match their
language. All text displayed in MISCON and COMMLINK (generated by the
script) will be in the foreign language. Once the .TXT file is translated
from English to, say, German, the script .SCR source must then be parsed in
order to generate a German version of the .DAT file.

Finally, when writing scripts, you MUST include a comment block (you can cut and paste the excerpt below) at the top of the main file. This block will contain a brief description about the script. For more information about what its about, they can always read the .DES file.

```
#
# SCRIPT        Script file name, ie BCIA0201
# NAME          Author's name
#               list multiple authors on separate lines
# EMAIL         email address
# WEB           Website URL
# DATE          Date of creation
# NEW SOUNDS    Y | N (this will always be N unless you have new
soundfx)
# VERSION       The version of the PREPARE parser used. This is
displayed
#               when the parser starts.
#
# SCENARIO      Brief description of the scenario.
#               Use multiple lines if necessary.
#
```

2.4 SCRIPT DEVELOPMENT TIPS

Since you will be working from a console MS-DOS box, you can make use of multi-tasking and reduce stress. You cannot write five lines of a two thousand line scenario and expect to run it right away. Unless of course you are testing. You have to finish the script, parse it, run the game and then note any inconsistencies. If the script parses, then it will run,
all you will be looking for are flaws in the logic (as previously) discussed.

You should open your text editor as normal and create the script from there. Open up another windowed DOS session (with a minimum of 8MB of memory available in the shortcut properties) and log to the folder containing the parser and your script. Once you have finished writing the
script, even if it is five lines, you simply minimize the editor and run the parser from the DOS window. You will then be able to catch any errors,
hit the enter key to return to the DOS prompt (allowing PREPARE to close the file), make a note of the line and error type, then switch to the

editor and correct the error.

2.5 THE SCRIPT AND OBJECT PARSER

The PREPARE parser is used to parse an ascii script file into a format that
can be read by the interpreter. It can also parse .3D objects and jam relevant AI instructions into them for use by the game engine.

When a script is parsed, each block is access and parsed one at a time. If
an #include directive is encountered, the file is loaded and parsed. Processing then continues in the original file. Prepare does multi-passes
in order to resolve forward references. This means that an object can be scripted to reference an object that is probably not even in scope at the
time of interpretation.

Blocks can be arranged in any order but for clarity, you must follow the convention and be consistent. If you script the [ai] block for an actor and then it is followed by instructions that are only valid in a [dynamic]
block, an error will occur. You must ensure that your script line is in context with the block being parsed. This is why, in the previous section,
you were advised to let each actor have it's own [dynamic] and [ai] block.
The [sets] and [event] blocks should be last in the file. If you put event
identifiers in a [sets] block and vice versa, an error will occur.

PREPARE expects all files to be terminated with the $ character. This tells
it that an end of file has been reached.

All blocks MUST be on column 1 of the line. This also applies to the files
included with the #include directive.

When an error is encountered, PREPARE will terminate with an explanation and possibly a line number containing the error. If the line is correct then you must check that the command is in context with the relevant block.
A typical mistake would be to be putting [ai] block commands in [dynamic]
blocks etc.

2.6 UNIVERSAL TIME FORMAT

There is a universal time format that can be used in [acm] and [event] blocks.

<minutes> eg 10 = 10 minutes, or
<hours>:<minutes> eg 1:30 = 1 hour and 30 mins = 90 minutes, or
<days>:<hours>:<minutes> eg 1:4:10 = 1 day 4 hours 10 mins = 1690 minutes

The parser will not allow certain dubious times, for example

180 minutes (3 hours) can be specified as follows:

```
180      <--- ok
0:0:180  <--- ok
3:0      <--- ok
0:3:0    <--- ok

2:60     <--- invalid
1:120    <--- invalid
```

Similar restrictions apply to hours (must be < 24) if days are specified.

For convenience, days can be specified as:

```
4:: <-- 4 days
```

Times of zero are not allowed.

3.0 SCRIPTING BLOCKS - OVERVIEW

Each game script can contain definitions for objects, regions and even AI processing information for those objects and regions. A script is divided into separate blocks which determine how each actor/entity is handled.

There are system and mission scripts with the only difference being that mission scripts actually contain orders and goals for the entities created.

Mission scripts can include system scripts which contain definitions for the game universe itself.

Each script command MUST be on its own line. There is NO continuation character for script lines. So if you have word wrap turned on in your editor and it causes a lengthy command, ie "broadcast", to wrap to the second line, the PREPARE parser will choke. In general, you should always
have wordwrap turned OFF in your editor but remember to adhere to the maximum line limit for script commands and the [ACM] scenario description
block (see below).

Script blocks in GBS are :

```
[acm]         - ACM orders and definitions block
[dynamic]     - definitions for dynamic entities
[ai]          - goal related instructions for defined entities
[event]       - scripted events identifiers
[sets]        - defines sets of entities
[macro]       - macro definitions for entities and commands
```

SYNTAX LEGEND

```
< >       indicates a mandatory field
[ ]       indicates an optional field
< >,< >,  indicates a collection of mandatory fields
[< >,< >] indicates a collection of optional fields
```

3.1 SCRIPTING BLOCKS - [ACM]

This is usually the first block in a script. It defines the scenario region

and also displays the mission instruction to the player. You can have as many [ACM] blocks as you like with varying duration. Each block will define
a new mission.

The maximum character length for the ACM block is 65. So, you should turn
on wordwrap at line 64 and end all lines with hard breaks. There is no word
wrapping when this block is displayed in COMMLINK at runtime. If you fail
to do this, then the mission description displayed in COMMLINK will not be
formatted properly.

SYNTAX

[acm]
<orders description>

:<acm_id>,<theatre>,<setup_event>,<begin_event>,<resolve_event>,<acm_tick>
 <time1>,<time2>,<time3>

<orders description>

Text describing the ACM scenario. Must not exceed 65 characters per line.
There is no restriction on the number of lines because this is displayed in
COMMLINK. You are advised to keep it short as possible but enough to clearly explain the goals of the mission. You CANNOT have a blank line within this field. If you want to insert a blank line in order to make the
description readable, put a period (.) in column 1 of the line.

The next line, preceed by a colon character, is the ACM definition line which consists of 9 parameters. ALL parameters MUST be provided and the entire definition MUST be on one line. So, if you have word-wrap turned on,
turn it off or this line will wrap at the predefined column setting for your editor.

<acm_id>

This is an identifier for the current mission within the scenario. It can
be any positive non-zero value. Example, mission 1 in TOD 1 containing several missions can be identified as 101, 102, 103...etc

This id is also used in the events/endevents block in order to restrict processing for the particular ACM scenario only. If you fail to use this id, then ALL events will be fired within ALL [acm] blocks instead of just
the current one.

The ACM id can be used in an IF/IFNOT statement as well in order to test if
the condition is true/false for the current ACM scenario.

  IF ACM 101 <command>

meaning: if the current ACM scenario is 101 do <command>

<theatre>

is where the mission scenario starts, it must be a valid region name. This
is where the player has to go for the ACM events (see below) to be fired.

<events>

setup_event : signalled just before the time1 timer is started
begin_event : signalled just before the time2 timer is started
resolve_event : signalled after time2 has expired

These are 'one off' events equivalent to user defined events specified in
an [event] block. They can be signalled using the signal keyword just like
other user defined events if necessary. An error is issued if the event names are already defined or are keywords. These can be any string name you
choose but try to use short and descriptive event names that are easy to remember.

You can use these events anywhere in your script within an events/endevents
block.

The 'setup_event' is rarely used.

In principle, you ALWAYS want to use the 'begin_event' in your scripts so
that all actors pertaining to this scenario are activated when the CURRENT
scenario starts.

The 'resolve_event' is very important because it signals the end of the scenario. You can use it to determine what happens when the scenario is finished. You can then do normal cleanup using this event. An example would
be to send an RTB order to all surviving ships etc.

<acm_tick>

is an event called every 5 seconds & used to terminate/control scenario using above conditional statements. A 'tick event' may be placed in any object, but the usual use would be to place the event in a persistent object which lasts for the duration of the ACM scenario. You can also use
this tick in an event/endevents block for checking events every 5 seconds.

EVENT TIMERS

<time1>:The max arrival time, the time allowed for the player to reach the theatre before the scenario starts, if the player doesn't get there in time the scenario will begin without him after
this time has elapsed. If the player reaches the theatre in this time, the scenario will begin as soon as he arrives.

<time2>:Scenario time, the time the player is allowed to resolve the scenario before it is resolved for him. The scenario will always take this amount of time even if the player has suceeded in destroying the bad guys etc. The player cannot request new orders
until this time has elapsed.

<time3>:Scenario interval time, this is the max time the player can lick his wounds between scenarios before he will get new orders from Galcomhq. During this time, the player can "request new orders" This terminates this interval delay and starts the next scenario.

The 'time2' variable is vital because, in conjunction with the 'resolve_event', it determines when the scenario ends. Sometimes, you won't
know how long a scenario will take. You should estimate how long you think
it would take and then add 15 minutes to this time. Even if there is excess
time left over and the player is idle, you can always use an 'event' to trigger the next mission using the 'resolve_event' event timer. For example, if you estimated that the player would take 15 minutes to destroy
a target and he finishes in 5, you can use an event trigger (perhaps in the
object he is sent to destroy) and a combination of the 'acm_next' command
to start the next mission without having him wait around for 10 minutes plus whatever 'time3' is set to.

NOTE: * Normally, you would use minutes in an [acm] block for simplicity.
* Refer to SECTION 2.6 for notes on the universal time format

USING COMMENTS

Comment lines (lines starting with #) are skipped. Orders text lines cannot
be blank since these are eaten by the parser, to specify a blank line (for
spacing the orders neatly), place a . (period) at the start of the line. The entire line will be replaced by a blank line ("").

Multiple acm blocks are allowed, but for safety, each must be a complete mission scenario (orders text and :params) otherwise an "acm definition is
incomplete" error is issued. This message will appear if there is spurious
text after a :params line.

ACM TRIGGERED ALERT CONDITIONS

Condition is YELLOW until player arrives at theatre or timeout
Condition is RED while scenario is in progress

NOTE: The internal low level alert condition, i.e, the CAS and SAS alerts,
takes precedence over the ACM alert condition.

Fresh orders can be requested by the player using ALT-CTRL-C if this enabled with the 'acm_script_enable' command.

In debug mode, current ACM number,state and date of next event is displayed above the SCRIPT line. ACM state is

```
   0 = No ACM,    (only when no ACM script)
   1 = ARRIVAL    (until player arrives)
   2 = INPROGRESS (duration of scenario)
   3 = RESOLVED   (until new orders requested/received)
```

The first mission is issued one minute after the game starts for the first
time.

EXAMPLE

```
[acm]
CAMPAIGN.................TEST TOD1
THEATRE OF OPS...........EARTH/SOL
MISSION CLASS............LONG RANGE TOW/DELIVER TEST
.
A diplomat ship is starting in the Canaan region.
.
You have 5 mins to get to the area before the scenario starts.
This scenario will run for 30 mins before it is auto-resolved.
You have 15 mins downtime to the next mission.
.
:101,earthz,setup_tod1,begin_tod1,resolve_tod1,tod1tick,5,30,15
#
```

3.2 SCRIPTING BLOCKS - [DYNAMIC]

SYNTAX:

```
[dynamic]
<race> <caste>,[<skill>],<id>,<model or class>,[<vdd name>],[<alias>]
```

<race>

The race is mandatory and can be any valid race, ie

TERRAN, SYRION, KANDORIAN, EMPIRIAN, VESPERON, DROIDAN, FALKERIE, MANDORIAN, GAMMULAN, ZELON, VALKERIE, CREDIAN

The race can also be:

MIXED meaning it is not "owned" exclusively by a single government, or that it has no ownership (eg a comet) It is neutral.

ALLY meaning that the object is any race that is allied to the terrans picked at random when the object is created

ENEMY meaning that the object is any race that is an enemy of the terrans
picked at random when the object is created

NEUTRAL meaning that the object is any race that is neutral to the terrans picked at random when the object is created

ANYRACE meaning that the object can be any race picked at random when the
object is created.

<caste>

The caste is mandatory and can be any of the following :

Aggressive Castes:

EARTHCOM, POLICE, RAIDER, CRIMINAL, INSURGENT ASSASSIN, MILITARY, MERCENARY, MARINE (Note: these are all aggressive castes)

Harmless Castes:

WORKER, SENGINEER, FENGINEER, PARAMEDIC, SCIENTIST, HACKER, DIPLOMAT, TOURIST, EXPLORER, JOURNALIST, COLONIST, TRADER, COMMERCIAL

HARMLESS : Any harmless caste picked at random

AGGRESSIVE: Any aggressive caste picked at random

ANYCASTE : Any caste picked at random from HARMLESS and AGGRESSIVE types

[<skill>] : The skill is optional and used to define the skill level of
the object. If not specified, a random skill level is generated which is made up of several skill related attributes. Though this field is optional, use a comma place
holder instead of a 1 to 5 value to indicate that you want the interpreter to pick a random skill.

Valid skill levels are from 1 (low) to 5 (high)

SUPERACE : Highest
ACE :
VET :
CADET :
NOVICE : Lowest

<id> : This is mandatory. Each object in a script MUST have a unique identifier which is used to reference it. This identifier MUST be unique in the script it is defined in
as well as in any other scripts that are linked to the current script using the #include directive. For instance,
you can't have 'ship1' in the current script as well as in
a script that is linked to the current script. You can however use this identifier in any other script set that
is not linked to the current one.

<model|class> : This is mandatory and can be the .3D filename of any valid object in the game or a class of object (advanced users). ie BCRUZMK2.3D (object) or BATTLECRUISER_MK2 (class) can be used. See OBJCLASS.SCR file for a list of
valid classes, and OBJDEFS.SCR for a list of models.

[<vdd name>] : This is optional and overrides the actual 'filename' of the object being used. An object with a filename of BCRUZMK2.3D will be listed as BCRUZMK2 in the VDD. To

change it to something else, ie 'primary target' use this
parameter.

[<alias>] : Optional name of unit. This overrides the VDD name. There
should be NO spaces in this parameter. To include a space
use the underscore character

EXAMPLE

```
[dynamic]
terran,military,,friend1,solnar.3d,primary_target
```

3.3 SCRIPTING BLOCKS - [AI]

The [ai] block contains AI processing commands for all entities within the
script. Each entry within an [ai] block must reference an identifier in the
[dynamic] block. All parameters are mandatory! The object's definition
determines its attributes when created.

SYNTAX

```
[ai]
.<id>,<shield>,<armor>,<engine>,<weapon>,<launch
bay>,<reactor>,<cloak_rating>
[FACE <id>]
START <parameters>
<nav action>
<nav action>
STOP
EVENTS [<acm_id>]
!<event id>
  <command>
!<event id>
  <command>
ENDEVENTS
```

The actions/orders which the object can perform must be nested within and
EVENT and ENDEVENTS block. The EVENT syntax must also contain an identifier
which references the scenario in which the event is triggered. You can have
as many event ids and commands within the 'event block' as you like, as
long as they are logical and make sense.

Every event MUST be on a separate line and preceeded by the ! (exclamation
mark) character which MUST be on column 1 of the line.

You can place an event id anywhere within the EVENTS/ENDEVENTS block. The
location within the block does not matter.

An <event id> can be any of the three defined in the [acm] block or a user
defined event that you create.

The <acm_id> is a numeric value which is defined in the ACM syntax line.

The events between EVENTS and ENDEVENTS are only processed if the current ACM scenerio is equal to <acm_id>, otherwise they are all fired when the scenario starts. If you don't provide an <acm_id>, then the default value of 0 is used. This means that these events are processed for
all acm scenarios.

When the block is processed by the interpreter, ALL the <commands> defined
for an <!event> are fired at the same time and in order. There is no pause.
Therefore if you the first command you give the 'jump' order and the second
command is 'halt', then the object will jump to the region and then halt when it gets there. Because all <commands> are processed, you can use the
very powerful IF conditions within <!event> blocks to determine when a command is fired.

WARNING: You MUST use an <acm_id> parameter when creating scripts that span
several ACM missions in a single scenario. Failure to do this will result
in script commands for the ENTIRE script to be fired at once. So for instance if you have two scenarios with the 101 and 102 ACM identifier (as
explained in the [acm] block above) and you fail to include these identifiers for the events, once the script is started ALL the events for
both missions will be fired at once. This is a sure fine way to create unpredictable scripts. If you only have a single ACM mission, then you can
safely omit this parameter.

BLOCK CONTINUATION

AI block continuations are also allowed. To continue scripting an object (possibly in a different file) use the following syntax in any
[ai] block.

Example, I need to add new directives for 'objectA', which is defined somewhere in the current script file or in another file that is INCLUDED by
the current one using the #include directive. So I would simply create new
events for this actor within another actor's block or by itself.

Lets assume that this block is the original definition for the object (where it was first created) in the current or other script file.

```
[dynamic]
terran military,,objectA,lrt10,primary_target
#
[ai]
.objectA,100,100,100,100,100,100,0
FACE karl.bmp
start inactive near earthz^jmp-12 2000 jmp-11
cruise earthz^jmp-11
stop
events 101
```

```
!begin_tod1
  activate this
!reach earthz^jmp-11
  broadcast "Arrived at Jump Point 11 in Earth region"
endevents
```

Now, lets assume that further down in the current script file or in another
file that is linked to this one using the #include directive, that I want
to create additional orders that this object can perform either during
the processing of the current ACM (id is 101) or when another is active, I
would simply reference it by including its id in a new [ai] block (eg1) or
in a block (eg2) already defined for another object. In which case, I would
do the following.

e.g. #1

In this example (somewhere further along in the current or different script
file) the continuation is in its OWN [ai] block. This is valid. You can
even insert it AFTER the [dynamic] block for objectA but that would be BAD
scripting practice because it makes it harder to read and is confusing.
TIP: Always try to pair the [dynamic] and [ai] blocks for the entities they
are created for.

```
[dynamic]
terran military,,objectB,generis.3d,secondary_target
[ai]
.objectB,100,100,100,100,100,100,0
start inactive near earthz^galcomhq 5000 jmp-11
stop
events 102
!begin_tod2
  activate this
  broadcast "Standing by"
endevents
#
[ai]
.objectA
events 102
!begin_tod2
 rtb
 broadcast "Returning to base"
endevents
```

e.g. #2

In this example, the continuation is within ANOTHER object's [ai] block.
This is perfectly valid because the [ai] block for objectB is 'in scope'.
If there was another block, ie a [dynamic] block in scope (eg3), then I
would be forced to use the method in eg1 other the parser will flag it as
an error.

You could even insert an [ai] block definition before the .objectA

identifier on line 13 of the script (after the comment separator) and it would be valid. This would be only to make it readable but it is valid. Why? Because even though you have an [ai] block for objectB and the one for
objectA comes after, 'an' [ai] block is in scope, making the operation perfectly valid.

Also, as you can see from the examples that you do not need to provide the
'AI definition flags' for objectA (when continuing) because you had already
provided them when the object was first created in eg1. All you need is the
id for objectA (the actor in question).

```
[dynamic]
terran military,,objectB,generis.3dc
[ai]
.objectB,100,100,100,100,100,100,0
start inactive near earthz^galcomhq 5000 jmp-11
stop
events 102
!begin_tod2
  activate this
  broadcast "Standing by"
endevents
#
.objectA
events 102
!begin_tod2
 rtb
 broadcast "Returning to base"
endevents
```

e.g. #3

This is an example of what NOT to do. If you use this method, the parser will flag it as an error because the [dynamic] block for objectB is in scope and therefore you MUST use the method in eg1 to continue the orders
for objectA. To make this valid (but a BAD scripting practice), you would
insert an [ai] block definition on line 3 of the script. This would put the [ai] block for objectA and objectB in scope with no problems.

```
[dynamic]
terran military,,objectB,generis.3dc
.objectA
events 102
!begin_tod2
 rtb
 broadcast "Returning to base"
endevents
#
[ai]
.objectB,100,100,100,100,100,100,0
start inactive near earthz^galcomhq 5000 jmp-11
stop
events 102
!begin_tod2
  activate this
  broadcast "Standing by"
```

endevents
#

If you are going to be defining a continuation for another ACM mission id
(102) (as I did in eg1) then I would need to provide the <acm_id>. For a secure script, even if the continuation is for the current ACM mission that
is in scope (currently running and has not yet reached the 'resolution phase'), you MUST always provide the <acm_id>, this way, you are certain that the object is going to perform as scripted (subject to its own internal AI overrides of course)

If the object is out of scope, i.e destroyed, docked etc, the entire process will simply be ignored because it won't find the actor. This can happen if you scripted a ship in, say in an earlier ACM mission and it got
destroyed by the time it was called again in another future ACM mission.

TIP: This permits me to mention here that if you are going to script entities that are to be preserved for an extended period, since you don't
know if they will be around by the time they are needed again, it is always
wise to either make them indestructible or use a combination of IF flags to
see if its in scope or not. This way, your script won't fall through. An example would be if you scripted a transport to go from Earth to Mars in an
ACM mission. Then several missions later, you want it to return home or perhaps you want the player or another AI actor, to interact with it. If your script relies on this sort of action, it will faill if the ship was destroyed in the interim. See what I mean?

AI DEFINITION FLAGS

The object's ai definition lists its attributes that it is created with. Each entry MUST be preceeded with a period indicating the starting of the
object's definition. Though all parameters are mandatory, some are ignored
in objects for which they are not applicable, ie asteroids do not have weapons, fighters do not have launch bays etc.

The integrity level displayed for the object in it's TTD is the mean of it's ENGINE, WEAPON, LAUNCH BAY and REACTOR levels. When hit, damage is applied to one of these systems at random and based on the proximity of the
hit. If the mean of these systems is less than 15, the ship will be disabled. For a station, if it reaches 25, it will emit an SOS signal allowing it to be captured.

<id> : Identifier created in [dynamic] block, used to reference object Each object in a script MUST have a unique id.

<shield> : The object's shield level. This offers a level of protection
to the object. When hit, the shield absorbs the damage. If breached, then the armor and subsequently the object itself will take damage.

0 (no shields) - 100 (full shields)

<armor> : The object's armor level. This offers a level of protection to the object. When hit, the armor absorbs the damage if the shields are down or have been breached. If breached, then the object itself will take damage.

0 (no armor) - 100 (full armor)

<engine> : This parameter affects the object's turn performance and the recharge rate of it's jump engines. The lower the engine's integrity, the worse the performance of the ship.

0 (no engines) - 100 (fully operational)

<weapon> : This parameter affects the integrity of the ship's weapon systems. If this value is less than 25, the guns won't fire and missiles cannot be launched.

0 (no weapons) - 100 (fully operational)

<launch bay>: This parameter affects the integrity of the ship's launch bays. If this value is less than 25, support ships such as fighters and shuttles cannot be launched.

0 (no launches) - 100 (fully operational)

<reactor> : This value affects the integrity of the ship's reactor system. Though reactor performance is not modelled, if this value drops to 0, regardless of the state of the other systems, the ship/station will explode and be destroyed.

0 (reactor breach) - 100 (fully operational)

<cloak rating> : This allows the ship to be cloaked when created if it has cloaking capabilities. If the ship's integrity falls below 65% it will decloak and recloak once it's repaired. Ships that have cloaking capabilities will use it based on their skill level. However, once a ship launches a weapon or another ship, it will briefly decloak and eventually recloak. A ship's cloaking is activated using the 'CLOAK' script command. This flag does NOT cause the object to be cloaked when created. It just says that the object has a cloaking device and that it can use it as its AI permits. You can cloak any object even if the parameter is set to 1 because the 'CLOAK' script command does not check to see if the object has this flag set to 1.

0 (no cloak) - 1 (cloak capable)

3.4 SCRIPTING BLOCKS - [EVENT]

You already know that you can use the [acm] block event identifiers within the EVENT/ENDEVENT section of the [ai] block in order to make entities do interesting things. So far, in the examples you have seen in the previous sections, only the [acm] identifiers were used.

Also, the event identifiers in the [acm] block are checked at a predetermined time and the <acm_tick> event is fired every 5 seconds. What if you wanted to create your own events that you can check at whatever interval you choose?

Enter the [event] definition block. Don't confuse this with any other block. This block is used to register user defined timer events which occur once or in a user defined cyclic (repeated based on a timed pattern) sequence. In this block, you can create events that you can use in ANY object's [ai] block.

The [event] definition block may appear anywhere in the script, and forward references are allowed. There may be more than one [event] definition block and subsequent blocks augment the last [event] block. Though you can put this block anywhere in the file, you must make it common practice to put it at the END of the file. This way, if you want to add/remove and event id, you know where to look.

SYNTAX

```
[event]
!<event_identifier>            // user defined identifier
[<date>[,<repeat interval>]]   // date, repeat etc
```

The leading ! character indicates that the string is an event identifier. Internal event identifiers (See SECTION 7.0) are already registered internally and need not be registered here. This block is only for registering external user defined events which trigger specific actions determined by the script.

The event identifier is equivalent to predefined event names and can be any string. The <event identifier> may NOT be a keyword nor the name of a previously used actor identifier.

Optional lines following the event identifier are dates at which the event is signaled. For each date, an optional repeat interval (in minutes) can be specified. The event will be signaled on the date/time specified, and then again repeatedly every <repeat interval> minutes. If the <repeat interval> is not specified the event is signalled only once. Several lines may be used to list more dates/intervals when the event is signaled, until an !

is
found which starts the definition of a new event. This is called a
CYCLIC
event. Once this type of event is fired, it will continue to be
signalled
indefinitely or until the pre-defined date expires.

The cyclic timers are optional, in which case the event is not signaled
by
time, but by a signal action.

EXAMPLE

User defined events can be signaled at any time within the [ai] block of
any object by using the 'signal <event_id>,[<time>]' script command
within
another event. You can have as many commands within an event as you
like.
You can even create an event loop if you'd like. Just make sure you get
the
timing right.

e.g. #1

In this example:

- The !saythis1 event is fired immediately (no time given) when the
!begin_tod2 event is fired. Once fired, the 'broadcast' command defined
for
the event, is fired.

- The !saythis2 event is fired 5 minutes after the !begin_tod2 event is
fired.

- The !saythis3 event is fired 2 minutes after the !saythis2 event is
fired.

- The !saythis4 event is fired every 6 minutes once its called for the
first time. Giving 0 as the date, signals it at the current date/time.
To
schedule another date, refer to SECTION 4.4 on how to use the unversal
date/time format. The commands may seem strange to you because you don't
know about them yet. But, I have used an IF condition in this cyclic
event
so that certain conditions are tested every 6 minutes. Every 6 minutes,
objectB checks to see if the player's BC is in its current region. If
this
is true, then it powers up its engines, arms its weapon systems and
attacks
the player. If not, it broadcasts a message until it is fired again, 6
minutes later. You will also notice that I use the !resolve_tod2 ACM
event to allow this object to go home. I also remembered to tell it to
activate its engines and systems. This is a safeguard because if the
player
never shows up this region, then objectB will never be able to fly nor
fire
weapons. This means that when the ACM scenario resolves, it won't be
able
to leave.

- Notice that even though objectB was the one that fired the !saythis3
event, objectA can also do something because that event is also defined

within its event definition. In this case, objectA will perform two actions
once the event is fired. WARNING: If I had defined the cyclic !saythis4 event for objectA, the script would still work but the logic would be different! Why? Because the FLEE command tells objectA to leave the area if
the player's BC shows up. But wait! Since the FLEE command causes it to leave the current region and the event is cyclic, every 6 minutes, objectA
will check its current region for the BC and if its there, it will again perform a FLEE to another region. Cool huh?

Notice also that in the [event] block definition, the cyclic event timer only pertains to !saythis4 because it was defined immediately AFTER the event it was defined for. Had I placed it BEFORE !saythis4, then the !saythis3 event would be cyclic and !saythis4 would be a normal event that
is fired once only.

```
[dynamic]
terran military,,objectB,generis,friendly_unit
[ai]
.objectB,100,100,100,100,100,100,0
start inactive near earthz^galcomhq 5000 jmp-11
stop
events 102
!saythis1
  broadcast "standing by"
  !saythis
!saythis2
  broadcast "still standing by"
  signal saythis3,2
!saythis3
  broadcast "stopping all engines"
  broadcast "weapon systems now off"
  halt on this
  weapons off this
  signal saythis4
!saythis4
  IF HERE bc THEN
   halt off this
   weapons on this
   attack it
  ELSE
   broadcast "Player not detected in current region"
  ENDIF
!begin_tod2
  activate this
  broadcast "Standing by"
  signal saythis1
  signal saythis2,5
!resolve_tod2
  halt off this
  weapons on this
  rtb
endevents
#
.objectA
events 102
!saythis3
  IF HERE bc THEN
   broadcast "I'm outta here!"
```

```
   flee
  ENDIF
endevents
#
[event]
!saythis1
!saythis2
!saythis3
!saythis4
0,6
#
$
```

Using events, you can create some pretty advanced and complex event conditions. In the next example, I use this trick to simulate a conversation among the BC crew during a scenario. Notice that I use the ACM
!todtick (called every 5 secs) to check for the ship. If found, thats what
triggers the dialogue. Notice how I use timing sequences to simulate conversation where someone says something and some time passes before someone else responds. Because the commands are fired all at once but are
processed in order, everyone will speak one after the other in some instances.

This is a script fragment so some definitions are missing. Notice the use of the FLAG1 in the 'reingard' actor. Since the !tod1tick is processed every 5 secs, I don't want the dialogue repeated. So, I use this
flag to get out of the event loop once its been set, i.e. the dialogue has
already been played once.

e.g. #2

```
events 102
!todtick
   IF FLAG1 reingard RETURN
   IF HERE bc THEN
    say CO,"We seem to have located the Reingard commander"
    say CO,"Scanning for vital signs now..."
    signal found_reingard,4
    signal talk1,1
    signal talk2,3
    signal talk3,4
    FLAG1 ON reingard
   ENDIF
!talk1
   say RE,"No vital signs detected commander"
   say RE,"....looks like they're all dead"
   say CO,"Searching computer logs now"
   say RE,"I'll do a system wide analysis check..."
   say RE,"Kara, set me up with a class 5 diametric scan range"
!talk2
   say TO,"...am getting an interference from the antimatter reactor"
   say TO,"looks like the gravimetric stabilizer has be polarized"
   say CE,"I can probably bypass it by rigging a distortion field"
   say CE,"in the systems relay matrix"
   say CE,"...lets tractor it"
   say FO,"Roger that Kendrick"
!talk3
```

```
  say CO,"Commander, I have accessed the computer logs..."
  say CO,"The probe is broadcasting from Polaris!"
  say RE,"All we have to do now is figure out what happened to the
  crew"
  say CE,"....and the Hyperion Subspace Device"
endevents
#
[event]
!talk1
!talk2
!talk3
found_reingard
$
```

Note the use of the $ character in these examples. This is to show that the [event] block is at the bottom of file. Remember what I said earlier that the $ character must be the last line in the script source file?

WARNING: If you are going to be signalling an event from within the [ai] blocks of several actors, be sure to know *exactly* what each actor is going to be doing. This is one area where I've seen many scripts behave differently because the logic is flawed.

3.5 SCRIPTING BLOCKS - [SETS]

Though you can script individual actors, there are times when you will want to work with a group of actors. This is where sets come in. Using sets, you can work with actors, entities as well as cargo items using a pre-defined selection set within the [set] block. Like the [event] block, you should try to keep the set definition block at the end of the file so that its easy to locate and modify.

SYNTAX

```
[sets]
:<set identifier>              // user defined identifier
[<set member>,...]             // set members
```

The leading : character indicates that the string is a set identifier.

The set identifier is equivalent to predefined event names and can be any string. The <set identifier> may NOT be a keyword nor the name of a previously used actor identifier. All set identifiers must be unique.

<set member> may be a collection of races, castes, actors, objects, concrete classes, previously defined sets, and numeric store items. The set acts as a collection of one each of these entities. The <set member> may be given in any order and may span several lines,

In the case of subsets, the set contains all the elements of the named subset as well as it's own elements. A set may not include iteself in its definition.

A new set definition, new block, or end of file "closes" the current set.

The set may subsequently be used in several internl or user defined events
and commands including !detect, !exist, create etc. Forward referencing of
the set is allowed.

There can be any number of user defined sets. The only penalty of a set is incurred when it is used in !detect or create because each one is processed individually.

EXAMPLES

e.g. #1

In this example, the items placed in the cargo pod are not in a set. When
the pod is created there will be 100x67, 1x189, 50x277 items. The items use
the syntax of the ADD_STORE command (quantity x item_number). Once the 'cargo_pod' actor is created, it will contain the items defined using the
ADD_STORE command.

```
[dynamic]
terran scientist,,cargo_pod,cargo2,pod_with_stuff
[ai]
.cargo_pod,100,100,100,100,100,100,0
start inactive near earthz^jmp-10 2000 jmp-04
stop
events 102
!begin_tod2
  activate this
  add_store this,100,67,1,189,50,277
endevents
```

e.g. #2

In this example, the items placed in the cargo pod are in a set. Also notice that there are two lines in the set. You can have as many lines as
you want. The shorter the line, the easier it is to read and understand. The items in the set definition use the syntax for the ADD_STORE command (quantity x item_number). Once the 'cargo_pod' actor is created, it will contain the items defined in the 'cargo_pod1' set.

Also in this example, there are two other sets which create generic cargopods (with no items in them) when the !begin_tod2 event for the cargo_pod actor is fired. (1) The 'cargo_pod2' set uses the internal 'cargopod' class name. This allows the system to pick a random type of cargopod. There are only two types of cargo objects in this internal class.
It consists of cargo1.3dc and cargo2.3dc objects. See SECTION 8.0 for a list of internal classes and the OBJDEFS.SCR file for a list of objects.
(2) The cargo_pod3 set uses the object name directly.

Note that unlike the cargo_pod actor which is created by the script, the other two pods are created using the CREATE command which uses the set to
determine what type of entity to create. It creates one of each set type.

The [event] block is just there to demonstrate how these two sets can

co-exist at the end of the file. If you remembered the discussion on events, then you will see that the cargo_pod actor will signal an event 2
minutes after it is created. It is that event that causes the two other pods to be created.

Another difference with these 3 pods is that only the first one has an id.
Using this id, you can fire an event that uses a command to add items to it. Since the other two don't have an id, you cannot script any events for
them and this is why they are created with no items in them. If you target
the first pod in the VDD, you will be able to see its contents and have a
shuttle collect them.

```
[dynamic]
terran scientist,,cargo_pod,cargo2,pod_with_stuff
[ai]
.cargo_pod,100,100,100,100,100,100,0
start inactive near earthz^jmp-10 2000 jmp-04
stop
events 102
!createit
  create 1,cargo_pod2,cargo_pod3
!begin_tod2
  activate this
  signal createit,2
endevents
#
[sets]
:cargo_pod1
100,67,1,189
50,277
:cargo_pod2
cargopod
:cargo_pod3
cargo2.3dc
[event]
!createit
$
```

e.g. #3

In this example, a set of hostile targets is defined in a set and accessed
collectively. The friend1 actor will broadcast that message if it detects
any member of the 'hostiles' set, in this case, hostile1 or hostile2.

You will also notice that it also has provision for detecting just one ship. In which case, it will broadcast both messages if it detects hostile1
but the second message will not be broadcast for hostile2. You could add a
'!detect hostile2' event for a message specific to that actor.

You may be wondering about the 'attack it' command. Well, its perfectly valid. The actor will act the first member of the set that it detects. If
it gets destroyed, then it will move on to the next member until all

members of the set are out of scope (left the region, destroyed, docked etc).

The 'hostiles_class' set lists races and a caste that the player has determined to be hostile. In this case, if any Gammulan or Vesperon type actor is detected, the message will be sent. It will also be sent if an insurgent caste is detected. The '!detect gammulan,vesperon,insurgent' event could also have been used, but I used a set to illustrate the benefits of using a set. This set can be accessed by any object without have to remember all the set entities as you would if you were writing each
one individually.

Though you can mix and match entities in a set, be aware of how the actor
will act based on each member of that set. In the example you will see that
the friend1 actor will repeat the broadcast under several circumstances as
explained above.

Also, create unique sets when possible to avoid confusion. This means that
while you can mix races and castes in a set, try not to put cargo items in
that set too. You will get confused because of how the parameters are passed.

```
[dynamic]
terran military,,friend1,interceptor_mk1,friendly_unit
gammulan military,,hostile1,vandal,leader
gammulan military,,hostile2,starlance.3d,escort
[ai]
.friendl,100,100,100,100,100,100,0
start inactive near earthz^jmp-10 2000 jmp-04
cruise earthz^jmp-04
stop
events 102
!begin_tod2
  activate this
!detect hostiles
  broadcast "hostile ships detected!"
  attack it
!detect hostile_class
  broadcast "hostiles detected!"
!detect hostile1
  broadcast "leader detected!"
endevents
#
.hostile1
start inactive near earthz^jmp-10 5000 jmp-04
stop
events 102
!begin_tod2
  activate this
  patrol earthz
endevents
#
.hostile2
start escort hostile1 250
stop
events 102
```

```
!begin_tod2
  activate this
  broadcast "escorting leader"
!detect friend1
  broadcast "enemy target detected"
endevents
#
[sets]
:hostiles
hostile1,hostile2
:hostile_class
gammulan,vesperon,insurgent
$
```

In conclusion, sets can be very powerful and make a script more dynamic and readable. Also, you can use sets to generate a random set of entities. The script for Xtreme Carnage uses sets to generate a random type of hostile ships in some levels.

3.6 SCRIPTING BLOCKS - [MACRO]

Sometimes scripts can be repetitious and long especially if you are designing a lengthy campaign. In these instances, you may want to use macros to simplify some of these commands that you use frequently.

Macros are defined in a [macro] block. The [macro] block should end with a line containing [endmacro]. The macro body may contain most types of commands including other script blocks.

More than one macro can be defined in a macro block. The [macro] block is also subject to the guidelines for creating other blocks such as [ai] and [dynamic]. The block must be in scope. You are advised to keep macro blocks either at the beginning or at the end of the script file. This way, it is easy to locate and change.

SYNTAX

```
[macro]
:<macro name>[,formal parameter[=default value]]...
<macro body>
:MACRO_END
[endmacro]
```

The leading : character indicates that the string is a macro identifier.

In the macro, text substitution of formal parameters is effected by placing the parameter within % metacharacters, i.e

```
%<formal parameter>%
```

When a macro is encountered in a script, it is expanded to the general form using the following syntax:

```
<macro name>([actual parameter][,...])
```

If an actual parameter is not specified the default value (if given in the macro definition) is substituted for it.

The macro name MUST begin on the first non-whitespace character in the line, and a macro with no arguments must be specifed as follows:

<macro name>()

If <macro name> is not previously defined, no expansion occurs.

Macro definitions may call other macros, which need not be previously defined.

Macro definition continuation lines for macros with many arguments is allowed. If the last charactor on the line is a comma, then the following
line is considered a continuation and is appended onto the previous line.
continuations may span several lines, eg

:macro(param1,param2,param3,param4,param5,param6,param7,param8)
<macro body>

may be written as:

:macro(param1,param2,
param3,
param4,param5,
param6,param7,param8)
<macro body>

EXAMPLE

This macro in eg1 generates ODS platforms using a default set of parameters
or replacement parameters. It contains a macro for actually creating the ODS object as well as an event block macro!

The lines that invoke the macro, do not need to be placed in any particular
block but for safety's sake, put them where they are most logical. Why? Because the macro defines all the valid blocks required for an object. This
includes the [dynamic] and [ai] blocks.

In this example, two types of ODS platforms are created around Earth and Sygan. When the friend actor is created, it can also reference a macro object because the macro also provides a valid object id.

Notice how in the first macro call, the default orbital speed of the Earth
ODS is over-ridden with a new value.

Observing the macro definition, you can see how easy it is to call the 'make_ods' macro without having to write a new definition each time.

WARNING: When calling the macro, you MUST observe the order in which the variables were ordered in the macro defnition. From eg1, you will see that
the first call to the macro overrides the object type and the orbital speed. These values are in the proper order. If the '1000' was before

the
'xtender.3dc' override variable, the parser will flag this as error because
that variable expects an object and not a numerical value. It is IMPORTANT
for you to remember this. You can bypass, i.e, use the default macro
variable, by simply using a , metacharacter where the value should be if
you are going to be skipping one value in order to override the next.

e.g. #1

```
[macro]
:ods_events()
events
!begin_tod1
  broadcast "I'm operational"
endevents
:MACRO_END
#
:make_ods(race caste,skill,planet,
distance=170000,
speed=500,roll=90,pitch=0,
obj_type=trancor.3dc,)
[dynamic]
%race% %caste%,%skill%,%planet%_ods,%obj_type%
[ai]
.%planet%_ods,100,100,100,100,100,100,0
start orbit %planet%,%distance%,%speed%,%roll%,%pitch%
stop
ods_events()
:MACRO_END
[endmacro]
#
make_ods(terran military,,earth,,1000,,,xtender.3dc)
make_ods(terran insurgent,,sygan)
#
[dynamic]
terran military,,friend,ic2.3dc
[ai]
.friend,100,100,100,100,100,100,0
start inactive near earthz^jmp-10 2000 jmp-04
stop
events 102
!begin_tod2
  activate this
!detect earth_ods
  broadcast "Orbital Defense System detected"
endevents
```

e.g. #2

In this example, a macro is used to generate a variety of ships. Note that
I have created a macro for almost every parameter allowed.

I have also created a macro expansion for a simple form of START/STOP type.
You can do this if you want but just bear in mind that the START/STOP
section in the [ai] block, takes several different parameters!
See SECTION 6.1 for more on this section.

Notice how I augment the default events for the second ship (insurgent)

created in the macro, with another event. Since you can have several [event] blocks for an object, this is perfectly legal. Just remember what
I've said about the scope of blocks and where to place them. If I had inserted the new events section when the [dynamic] section for another actor was in scope, an error would occur. As your scripting skills improve,
you will see how to use the object id anywhere in a script to override the
default events for an actor. In this example, you can assume that I have used this technique to augment another object's (friend1) macro events.

```
[macro]
:ship_events()
events 102
!begin_tod1
  activate this
endevents
:MACRO_END
#
:make_ship(st_mode,st_region,
race caste,skill,obj_id,obj_type,vddname,
shield=100,armor=100,engine=100,weapon=100,bay=100,
reactor=100,cloak=0,)
[dynamic]
%race% %caste%,%skill%,%obj_id%,%obj_type%,%vddname%
[ai]
.%obj_id%,%shield%,%armor%,%engine%,%weapon%,%bay%,%reactor%,%cloak%
start %st_mode% IN %st_region%
stop
ship_events()
:MACRO_END
[endmacro]
#
make_ship(inactive,marsz,terran military,,friend1,ic2.3dc,friendly_ship)
make_ship(inactive,syganz,terran
insurgent,,enemy1,vandal.3dc,enemy_ship)
events 102
!detect bc
  attack it
!resolve_tod1
  rtb
endevents
#
[ai]
.friend1
events 102
!damaged this,50
  broadcast "Damaged by 50%"
endevents
#
```

Macros can be as powerful as you want them to be. They can also be confusing and complex. Use wisely. When in doubt, do it the hard way.

3.7 SCRIPTING BLOCKS - [REGION]

You have no business messing with the [region] block. So, this section is for information purposes only.

There is absolutely NO reason for you to use this block because the internal

defaults are in use by the internal master script. Defining a [region] block
in your own script will not only alter the game universe, it will most likely wreak havoc with the 'world order'.

This block is used to define initial region settings & autogeneration.

SYNTAX

The following syntax is used for defining region startup parameters. A region may only be ONCE in the script and can apply to space or planet regions. See NAVCHART.PDF for more on region settings.

SET <race>,<rp>,<dp>,<con>,<tlvl>,<dlvl>,<tclass>,<inf>,<sec>,<pop>

<race> - Any valid race
<rp> - Resource Points
<dp> - Defense Points
<condition> - Normal,Unstable,Critical,Barren,Invasion
<tlevel> - Tech Level (1-10)
<dlevel> - Defense Level (1-10)
<tclass> - Tech Class (AD,MN,AG,RO,HT)
<inf> - Inflation Level (0-5)
<sec> - Security Status (LEGAL,ILLEGAL)
<pop> - Surface Population (0-100%)

DEFINING REGION AUTO-GENERATION ENTITIES

The following syntax is used for auto-generated entities. If a <race> appears as the first parameter instead of the SET keyword, this indicates to
the parser that it is an auto-generated entity. Autogeneration is region and
event based. These all appear within the region block consisting of region
names in the following format:

Note: You can turn auto-generation on/off using the AUTOGEN script command.

:<region>,<active_max>,<inactive_max>

<region> - Any space or planet region, eg EARTHZ, EARTH etc

<active_max> - the max number of objects that can be generated in a region
that is ACTIVE or one jump away from the player or a player controlled probe/ship.

<inactive_max> - the limit of the number of objects generated for a region
that is INACTIVE.

The region specification is followed by one or more event probabilities and
class instance which may be augmented. The format of each event is:

<race>,<caste>,<probability>,<event_interval>,<loiter_interval>,
<start_flag>,<search_flag>,<exit_flag>,<class>,<int_ag_max>,<int_ag_prob>

<race> - Any valid race.
Can also use E,F,N,* for enemy, friendly, neutral

(default), or random

<caste> - Any valid caste

<probability> - The probability in percent that the unit will actually be generated. Floating point value between 0.002 and 100.
The smallest allowable value is 0.002 which might be used
to generate a galaxian ship. 100 means the unit is always
generated if the region limits are not exceeded.

<event interval> - Time in minutes between each generation of a unit of this
class.

<loiter interval> - The max time in minutes that the unit will wait before leaving if it can't find anything interesting to do.

<start flag> - S,P,M,U,J,*, // station/planet/moon/uncharted/jump/all. Acts as location mask, if not specified, won't be visited.

e.g PM means the unit comes from a moon or planet only the actual planet or jump point etc is chosen at random

<search flag> - S,P,M,U,J,*, // station/planet/moon/uncharted/jump/all. Acts as search mask, if not specified, won't be visited. The unit will visit each flag type before leaving the region. Used for search and patrol actions.

<exit flag> - S,P,M,U,J,*, // station/planet/moon/uncharted/jump/all. Acts as exit mask and determines how entity leaves region.
If not specified, will not be used.

<class> - Class of object being generated.

<int_ag_max> - max number of intruders to create on players ship when this object is created. The probability of this occuring is given by int_ag_prob. Use 0,0 to disable the generation
The BC must be in the player region or the intruder generation is ignored. Intruders can't beam to player's ship if it is cloaked.

<int_ag_prob> - Intruder autogeneration probability (floating point)

EXAMPLE

e.g. #1

```
[macro]
:enemy_aggressive()                                    // AG macro
ENEMY,RAIDER,10,5,5,J,J,J,cruiser,2,5
ENEMY,CRIMINAL,10,5,5,J,J,J,cruiser,2,5
ENEMY,ASSASSIN,10,5,5,J,J,J,cruiser,2,5
ENEMY,MERCENARY,10,5,5,J,J,J,cruiser,2,5
ENEMY,RAIDER,10,5,5,J,J,J,fighter,0,0
ENEMY,CRIMINAL,10,5,5,J,J,J,fighter,0,0
ENEMY,ASSASSIN,10,5,5,J,J,J,fighter,0,0
```

```
ENEMY,MERCENARY,10,5,5,J,J,J,fighter,0,0
:MACRO_END
#
:random_aggressive()
ANYRACE,MILITARY,10,5,5,J,J,J,cruiser,4,5
ANYRACE,RAIDER,10,5,5,J,J,J,cruiser,4,5
ANYRACE,RAIDER,10,5,5,J,J,J,fighter,0,0
ANYRACE,CRIMINAL,10,5,5,J,J,J,cruiser,2,5
ANYRACE,MILITARY,10,5,5,J,J,J,fighter,0,0
ANYRACE,CRIMINAL,10,5,5,J,J,J,fighter,0,0
ANYRACE,ASSASSIN,10,5,5,J,J,J,cruiser,2,5
ANYRACE,ASSASSIN,10,5,5,J,J,J,fighter,0,0
ANYRACE,MERCENARY,10,5,5,J,J,J,cruiser,2,10
ANYRACE,MERCENARY,10,5,5,J,J,J,fighter,0,0
:MACRO_END
#
[region]
:earthz,10,5                                          // Region
SET TERRAN,0,0,NORMAL,10,10,AD,0,LEGAL                // Region definition
TERRAN,MILITARY,70.5,10,10,J,J,S,fighter,0,0          // AG entity
TERRAN,HARMLESS,80.5,9,10,J,P,J,carrier,0,0           // "
enemy_aggressive()                                    // macro expansion
friendly_military()                                   // "
#
```

NOTES:

(1) A region can be made truly dynamic using very powerful macros with eclectic entities. However, be certain that you know what you are doing. Since AG entities use their own internal AI logic, they will use the default AI rules for their 'type' based on their race/caste. So for instance if you generate 'terran military' and 'gammulan military' actors in the same region, a fight will ensue. Period. Similarly, if diplomats are generated in a region that has a probability of raiders showing up, they will get attacked. This sort of dynamic nature of the AG process is what keeps the universe dynamic. You can define protected regions, trade routes etc.

It is not an easy task to create a truly dynamic world and most especially one which will allow directed (actor using defined scripted actions) or 'free-form' (AG actor using internal AI rules and logic) actors. It has taken me over 5 years or more to get it just right and there are still a lot of variations that I still haven't investigated and permutations that I haven't considered.

This is all due to the advanced AI that actors possess and which determines what they do whether they are directed or free-form. In fact, creating a directed actor (as you know by now) only gives it direct orders. Once it has performed those, or if there is a conflict with a user defined order and its internal AI, an actor will always consult its internal AI rules in order to exist as it was intended. An Insurgent is an Insurgent, even if you told it to halt and park next to a Terran station. The minute it is created, it will

consult its internal AI logic in order to figure out what to do once the station starts launching attack ships at it. A directed actor created in a region, will find something to do even if you don't give it an order. In this instance, it acts just like a free-form actor.

(2) The active/inactive amounts can be overridden by other objects within the region. This means that if a region is scripted to only have 10 AG entities at any one time, once those entities start launching their own ships (carriers etc), the number of objects in the region will increase.

(3) An ACTIVE region is one that the player or one of his ships or probes has visited. It is also one that has an AI actor that requires background processing. This includes stations, ODS systems, comets etc. Therefore, out of the 91 space regions, 75 planets and 145 moons in the game, since each one probably has an entity, this means that about 75% of them are active at the start of the game. The only exception would be a region that only has a region with no station, ODS or comet. These actors that require constant AI updating are called 'dynamic' entities. Others, called 'static' entities, such as the planet object, jump gates etc, do not require background processing because they have no AI.

Once you, one of your ships/probes, or perhaps a scripted ship pass through this type of region (one without dynamic actors), it will switch it from INACTIVE to ACTIVE. The planetary region itself (if the space region contains a planet/moon) however will remain INACTIVE until the said entities actually enter it. So, yes, the space and planetary regions are processed independently of each other.

Once a region is made ACTIVE, it cannot be switched back. This is why, if you travel enough around the galaxy, the more skirmishes that are going on, the slower the game gets. In the next generation of this engine which will debut in the sequel, BC:3020AD, this processing has been streamlined remarkably. Future versions may probably have SMP support.

3.8 CREATING A WING OF UNITS

You can collect a group of units into a single wing and then be able to manipulate the wing as a single entity or access each unit in the wing directly.

[dynamic]
:<wing_id> (single colon to start the wing block)

<object list>

:: (two colons to end the wing block)

<wing_id> must not contain a '.' character.

<object list> is a list of one or more object definitions using the format of the [dynamic] block.

This will create a group of local wing objects that can be

simultaneously ordered in ai blocks.

<id> cannot be accessed in [ai] blocks unless the current scope is the wing it is defined in, but it can be globally accessed using the object syntax

<wing_id>.<id>

e.g attack wing1.f1

The syntax of <object_id> allows multiple <object_ids> to be used. The following script then is applied to all listed objects. The syntax is:

.<object_id>[,<object_id>...] [params]

or

..

The second form is used to cancel scripting for the current wing (if any) and return to the global script scope

e.g. This is a simplified script with multiple objects

```
[dynamic]
vesperon diplomat,,enemy1,lrt10,leader
vesperon military,,enemy2,vandal,escort1
vesperon military,,enemy3,corsair,escort2
[ai]
.enemy2,enemy3,100,100,100,100,100,100,0 #1
.enemy2
start inactive escort enemy1 150 3       #2
stop
.enemy3
start inactive escort enemy1 150 9       #2
stop
.enemy2,enemy3                           #3
events 101
!beginL1
   activate this
!detect player
   attack it
!resolveL1
   flee
endevents
```

Notes:

#1 applies parameters to all objects listed.

#2 the objects have unique start directives so that they don't appear at the same location.

#3 this defines the following script commands to apply to both objects, since these orders are identical.

A script that uses .<wing_id> in place of an .<object_id> can be used to encapsulate the objects in the wing. The example above can be rewritten as shown below:

```
[dynamic]
```

```
:enemy_wing1
vesperon diplomat,,enemy1,lrt10,leader        #1
vesperon military,,enemy2,vandal,escort1      #1
vesperon military,,enemy3,corsair,escort2     #1
::
[ai]
.enemy_wing1,100,100,100,100,100,100,0        #2
.enemy1,50,50,15,100,100,100,0                #2
start inactive near earthz^earth 5000 jmp-10
stop
.enemy2
start inactive escort enemy1 150 3
stop
.enemy3
start inactive escort enemy1 150 9
stop
.enemy2,enemy3
events 101
!beginL1
   activate this
!detect player
   attack it
!resolveL1
   flee
endevents
```

Notes:

#1 The object ids enemy1,enemy2,enemy3 are local to the enemy_wing1 group and may be used for other objects. You can then define wings in a [macro] block and then call-up multiple instances of them without the difficulties of unique naming.

#2 Specifying object params for the wing sets these parameters for all objects in a wing, they are redefined here for T1. previosly it was invalid to specify object paramters more than once. It is now possible to re-define them. The last definition is the one that is applied to the object. These paremeters are fixed once the script is running in the game (i.e you cannot change them "on-the-fly")

An example use of a macro that defines a wing

```
[macro]
:WINGTYPE1(NAME,M_NUM,M_NAME)
#
# Note that the [dynamic] line is part of the macro
# definition
#
[dynamic]
:%NAME%
vesperon diplomat,,enemy1,lrt10,leader
vesperon military,,enemy2,vandal,escort1
vesperon military,,enemy3,corsair,escort2
::
#
# Again [ai] is part of the macro definition
#
[ai]
.%NAME%,100,100,100,100,100,100,0
.enemy2
start inactive escort enemy1 150 3
```

```
    stop
    .enemy3
    start inactive escort enemy1 150 9
    stop
    .enemy1,enemy2,enemy3
    events %M_NUM%
    !begin%M_NAME%
       activate this
    endevents
    .enemy2,enemy3
    events %M_NUM%
    !detect player
       attack it
    !resolve%M_NAME%
       flee
    endevents
    :MACRO_END
    #
    # If you suspect problems with macros try adding the following
    # between after any [macro] blocks that end immediately before
    # [dynamic] or [ai] blocks
    # (as in this case)
    #
    [endmacro]
    #
    # Orders specific to the lead ship
    #
    [ai]
    .ew1.enemy1,50,50,15,100,100,100,0
    start inactive near earthz^earth 5000 jmp-10
    stop
    #
    # call up a wing for mission 101 (day 1 phase 1)
    # Note this must be done after defining the lead
    # ship START position (ew1.enemy1 above) since the fighters need to know
    # where the transport is located. Otherwise you will
    # get a "defined outside region" error
    # The transport ship definition in the [dynamic] section in the
    # macro definition is processed before the preceding
    # lead ship start/stop placement (in a prior pass) so it is
    # valid to reference it before it is defined.
    #
    WINGTYPE1(ew1,101,L1)
    #
    # The macro can be used again to create another similar
    # wing of ships for example:
    #
    .ew2.enemy1
    start inactive near earthz^earth 25000 jmp-02
    #
    # This time the lead will dock with a station
    #
    dock galcom
    stop
    #
    WINGTYPE1(ew2,102,L1)
```

4.0 FREQUENTLY ASKED QUESTIONS

Q. HOW DO I DISTRIBUTE A SET OF SCENARIO SCRIPTS?

A. You pack up the MIS, DES and DAT files associated with the script and send to the person. They would then copy it to their SCRIPTS folder and run the game. The files can either be in a zip file (neater) or not. e.g. you could have the files in a MYSCENARIO.ZIP file or just have the three files in the folder unzipped.

If there are naming conflicts e.g. you create scripts which have the same name as the game script, the game will use the scripts with the most recent date/time stamp. So, e.g. if you created a Roam scenario script for the commander and call it CMDR_BCR0000.SCR, the new version will be used by the game.

NOTE: When naming your scripts, do NOT use the naming convention used by game scripts!! e.g. BCIA0101 to BCIA0125 are a game scripts, so you should start your numbering from BCIA0201 to BCIA0299, BCIA0301 to BCIA0399 etc. This applies to all script types (ROAM, ACM, IA, TA) but each have their own naming conventions.

Q. CAN I MODIFY THE SPACE OR PLANETARY WORLD?
A. No. You can only create new scenario scripts.

Q. CAN I CREATE NEW MODELS SUCH AS SHIPS ETC?
A. No. You can only use existing 3D assets already in the game. For a list of game 3D assets, refer to the OBJDEFS.SCR file as well as the game appendix files.

Q. CAN I MESS WITH THE GAME INI FILES (DATA and PTE2 FOLDERS)?
A. No, some are MD5 encrypted and the game will refuse to start if you mess with them.

Q. HOW DO I MODIFY THE PARAMETERS OF AN EXISTING UNIT IN THE GAME?
A. If you are interested in changing settings such as flight dynamics, weapon loadouts etc, look in the OBJDEFS.SCR file.

If you are interested in changing the class definitions such as support crafts, types, complement etc, look in the OBJCLASS.SCR file.

e.g.

To change the firing rate of the Battlecruiser MK3 carrier, from one shot every 350ms to one shot every 100ms, go to OBJDEFS.SCR line 161 and change the %350 parameter to %100

To change its HJ transit time from 120 secs to 10 secs, change the J120000 to J10000

To change its HJ recharge time from 90 secs to 5 secs, change the j90000 to j5000

To change the damage factor of the shots fired by the Battlecruiser MK3 carrier (and all crafts that use that shot type), go to OBJCLASS.SCR line 502 and make a note of which shots it uses. They are type SHOTl4r. Now go to OBJDEFS.SCR line 589 and change the E100 to E1000 to make each shot give 1000 units of damage instead of the default 100.

The description of all the parameters are located at the top of the

system script file.

NOTE: that editing any of the .SCR system script files requires preparation
of ALL script files associated with them! If you do not do this (see the PREPSCRIPTS.BAT file comments), then you will have problems. Since you don't have the sources for the scenarios which ship with the game, the new parameters will NOT work with those scenarios, they will only work with your scripts. As such, if you make drastic mods to these system files, NONE of the scenarios which ship with the game, will work.

Q. HOW CAN I MODIFY THE ATTRIBUTES OF STATIONS ALREADY IN THE GAME WORLD?
A. Because stations, starbases, orbital defense systems, supply stations etc are also scripted, you can access them WITHOUT having to modify the system scripts in which they are defined. This example modifies the GALCOMHQ station from within your script and does not affect the station when other scenarios are run.

```
 [ai]
.galcomhq
events 1
!setupl1
  damage this 15 25 -1 -1 -1 -1
  weapons off this
  launches off this
  signal ghq_back_online,15
!detect player
  broadcast "OUR LAUNCH AND WEAPON SYSTEMS ARE OFFLINE"
  broadcast "SHOULD BE BACK ONLINE IN ABOUT 15 MINUTES"
!ghq_back_online
  launches on this
  weapons on this
  signal launch_wing,5
!launch_wing
  launch fighter,4,defend galcomhq
  broadcast "PROTECTIVE WING LAUNCHED"
endevents
#
[event]
!ghq_back_online
!launch_wing
```

Note that you don't need to include the [ai] block if you are adding this definition within a pre-existing [ai] block. Because this script fragment uses the station's identifier (see GLOB_DYN.SCR), you have direct access to it.

In the example above, the station's shield and armor integrity are reduced to 15% and 25% respectively. Its weapons and launch systems are also disabled. They are however turned back on 15 mins after this script starts via an event trigger. And five minutes after that, it launches a wing of four fighters (picked at random) to defend itself from further attack.

Q. HOW DO I OBTAIN THE SPACE REGION NAME AND ENTITIES FOR USE WITH SCRIPTS?
A. You can start objects in a space or planetary region. In the case of a space
region, you can use the "START NEAR" or "START IN" directive. But you need to know the following "

(1) The name of the region

(2) The name of an object in the region.

To obtain the name of the region, look in the WORLD.SCR file. Any region name ending with the Z character, is a space region. Without it, the region
is a planet or moon.

e.g. in world.scr line 110, is the name of the space region where the planet (line 111) and moon (line 112) are located. Between lines 113-120 are the jump anomalies in the region.

If you start the game and go to the Earth space region, the go to Tacops, you will see these two planets and those jump anomalies in the space region.
Using Tacops (you can take a screen shot of the region from the top-down view) is a good way to figure out where the anomalies are located, their distances and relation to each other etc.

Now that you know the name of the space region, the planet and the jump anomalies, you can use them to define the start locations of your units.

The following examples use variations and script directives

e.g #1

```
[dynamic]
terran military,,f1,bcruzmk2.3d
[ai]
.f1,100,100,100,100,100,100,0
start inactive near earthz^jmp-10 5000 jmp-02
stop
events 1
!beginl1
   activate this
!startup
   broadcast "I activated in Earth space region"
   broadcast "I am 5km from jump point #10 facing jump point #2"
endevents
```

e.g #2

```
[dynamic]
terran military,,f1,bcruzmk2.3d
[ai]
.f1,100,100,100,100,100,100,0
start in earthz
stop
events 1
!beginl1
   activate this
!startup
   broadcast "I started in Earth space region at random location"
endevents
```

e.g #3

```
[dynamic]
terran military,,f1,bcruzmk2.3d
[ai]
.f1,100,100,100,100,100,100,0
start near marz^mars 150000 moon-phobos
stop
```

```
events 1
!startup
   broadcast "I started in Mars space region"
   broadcast "I am 150km from the planet Mars and facing the Phobos moon"
endevents
```

Note also that in addition to using region names and anomalies within them,
you can also use other objects that exist in the region. As long as they exist, they can be used - and they can be any object (e.g. another ship, cargo pod etc). This example uses a pre-existing station in orbit around the planet Earth.

e.g #4

```
[dynamic]
terran military,,f1,bcruzmk2.3d
[ai]
.f1,100,100,100,100,100,100,0
start near earthz^galcomhq 50000 earth
stop
events 1
!startup
   broadcast "I started in Earth space region"
   broadcast "I am 50km from GALCOMHQ station and facing the planet Earth"
endevents
```

NOTE: You can also start the player in a space region by using the SET_REGION
command. It takes the same parameters as START IN or START NEAR.

In this example, the player is starting at a random location in the Earth space region.

e.g. set_region earthz

Q. HOW DO I OBTAIN THE PLANET REGION NAME AND ENTITIES FOR USE WITH SCRIPTS?
A. You can start objects in a space or planetary region. In the case of a planet region, you must use the "START ON_PLANET" directive. But you need to know the following "

(1) The name of the planet region

(2) The name of the mzone within the mzone

The PTEDEFS.INI file located in the PTE2 folder contains the list of all mzones and scenes in the game world. You would need the PTEstudio tool in order to view them without having to run the game. See below.

e.g.

```
.f1,100,100,100,100,100,100,0
start on_planet on_planet earth earth00[sbase] 175.86 51.14 0 180
stop
events 1
!startup
   broadcast "I started on the Earth planet in the earth00[sbase] mzone"
   broadcast "My X/Y locations within the mzone are 22.847,4.666"
   broadcast "I am at zero altitude on ground heading South at 180
   degrees"
endevents
```

NOTE: You can also start the player on a planet by using the SET_REGION command. It takes the same parameters as START ON_PLANET, except that it takes the scene parameter as well.

In this example, the player is starting in the base_earth01 scene located in the earth00[sbase] mzone on the Earth planet. This command can also take relative X/Y/Z locations within the mzone.

e.g. set_region earth,earth00[sbase],base_earth01

4.1 TOOLS - PTESTUDIO SCENE VISUALIZER

This tool is used to view the mzones and scenes on planets in order to obtain start location for scripts which take place on planets.

SETUP

1. Copy ptestudio3.exe and ptestudio.ini files to your game install folder. It will not work from ANYWHERE else!!

2. Create a shortcut with the desired parameters. See the ptestudio.txt for more info on valid parameters

USAGE

3. Start the program and go to FILE/SELECT PLANET/SCENE then scroll down to near the end of the list and left-click on the Earth (planet) in the first window, then EARTH00[SBASE] (mzone) in the second window, BASE_EARTH01 (scene) and click on the GO TO SCENE button. Wait for it to load.

Once you are taken to the scene, you will be in the middle of the scene at ground zero.

NOTE: That there usually is no model at the center of the scene location because the primary unit (e.g. a starbase) is scripted in order to have access to it at script level. This is because entities within a .3DG group file cannot be accessed from within a script.

1. Go to OPTIONS/LOCAL MAP to display the map of the area you are in. You should see a Yellow rectangle. This defines the mzone limits. The White box inside are the scenes within the mzone. The Yellow line indicates your location and direction of view/travel.

2. Go to OPTIONS/STATS and you will see a bunch of info, including the following important ones

The entries you are interested in are:

- planet : Earth
- region : terran/military
- mission zone : earth00[sbase],26,170
- mzone position : 176528.562500m,51721.531250m

Grab hold of paper and pen to write down the info you need from all this.

Unlike space based scripts which use an actor as a starting point, planet based
scripts primarily require a planetary location based starting point.

So, assuming you wanted to create an IA scenario which takes place in an mzone/scene above, you would need to know where to start the player and any scripted actors you wish to create within your script. Here's what you need to do:

1. Use the UP/DOWN/LEFT/RIGHT arrow keys to move to the desired location on the map. It can be inside or outside the scene but it must be within an mzone.
You can increase your traversal range using 0-9. When you reach the desired spot,
stop.

2. Now note your current position and write it down e.g.

mzone position : 176256.812500m, 51397.625000m

3. Repeat steps 1-2 as often times as needed. You would need to do this if you were starting actors at various locations within the mzone.

At this point you can exit the program and go back to scripting. Unless of course you want to see exactly where this mzone is located. In which case, go to OPTIONS/ZONE SELECTOR. The Red crosshair on the Zone Selector map is your current location on the planet. This is also the same map yo see in Tacops, though Tacops uses a more advanced version with a lot more options.

If you have the local map displayed, you can right-click anywhere on the Zone Selector, including inside the Yellow mission zone boxes and instantly be warped there. You will then see where you are on the local map. Then you can toggle off the Zone Selector and get a full view of the local area. You can also right-click inside the Local Map in order to warp to a location on the planet.

If you want to calculate the distance between two points e.g. you want some hostiles to intercept a target from a certain range and you don't want them to have to travel too far over boring landscape, then you would use the marker.
To do this, press M and a flashing diamond marker will appear. Now move to the location you want to measure the range to (you can either travel using the keyboard as normal, or right-click to warp there). As you move (or when warped to the location), the Marker Distance info will appear in the stats display. So if it says 21347.207321m, then thats how far you are from the marker. To cancel the marker, press SHIFT+M.

NOTE: That the program will always override the local time by default. As such,
it will always be daytime on the map you are on. To change it and use the actual
Time Of Day for the area you are on, go to ENVIRONMENT/CONTANT TOD
and toggle it.

WARNING: If you move outside of the local scene, the position will display NULL.
Thats normal and the mzone position co-ordinates will still be valid. But you must
NOT move outside the mission zone. If you do and it goes NULL, you're in deep trouble.

ALSO:

DO NOT!! mess around with the map editors in the Editor menu. If you do, you will
most likely screw up the INI files and invalidate the game environment!!

THE SCRIPTING PART
==========

NOW that you have the info, you can now use them in your IA script.

Below is a script excerpt which uses this info to start the player. The command
which uses these parameters is in bold letters. You can take a look at one of sample scripts which uses these similar commands.

```
    !init_player
      set_race terran
      set_caste insurgent
      set_career elite_force_marine
      set_region earth,earth00[sbase],base_earth01
```

If you go back to the info above, you will see that EARTH00[SBASE] mzone is under Terran/Military control. As such, the Terran/Insurgent EFM player
is being started in that mzone inside the BASE_EARTH01 scene on the planet Earth.
In other words, smack in the middle of a hostile base.
>
Because the starting location for the player (in sp and mp) is determined by the /wp tags within various objects, we don't need an actual starting location.
The system knows to pick a waypoint within the scene and start our hero off there.
But if you wanted to add an exact starting location, you could use this format which
puts you at that exact location inside the mzone.

```
      set_region earth,earth00[sbase],base_earth01,144.962,194.030,0,0
```

Below is an excerpt which uses more info to start an actor. Actors use the on_planet
parameter of the start command and NOT the set_region command. However, both use the
same values derived from the mzone.

```
    .badguy1,100,100,100,100,100,100,0
    start on_planet <b>earth00[sbase] base_earth01 176.256 51.397 0 0</b>
    stop
    events 1
    !startup
      broadcast "Ready to kickass and chew gum. But I'm all outta gum!"
    endevents
```

You are probably wondering why the values for the X and Y start locations for

this
actor are not the same as those you wrote down. This is because the co-ordinate system
ranges are calculated differently in the game itself. All you need to do in order to
arrive at valid numbers is to discard all numbers to the right of the decimal point
and divide the numbers preceeding the decimal point by 1000 to obtain the values
required by the game engine. In this case the recorded position of

   176256.812500m, 51397.625000m

becomes

   176.256,51.397

and the start location for your actor will be where you were on the map.

HOWEVER there are times when you need to use more precise values. After all, if you
are creating a team of marines, you probably want them being meters instead of
kilometers apart. In this regard, after you have actually converted the values,
you would further adjust the numbers precisely. Below is an actor who are merely
2 meters (along Y) apart from the first one above.

```
    .badguy2,100,100,100,100,100,100,0
    start on_planet <b>earth00[sbase] base_earth01 176.254 51.397 0 0</b>
    stop
    events 1
    !startup
      broadcast "Here, I have an extra pack"
    endevents
```

Though this FAQ entry is not to teach you how to use a scripting command, let me
explain the last two parameters used by this on_planet script command.

The Z component is only used for actors (e.g. crafts) who need to be started
in the air (though they can still be started on the groun). And if you're
thinking of doing this, be sure to be aware of the craft's capabilities because
if you script a fast mover at something like 1000 meters above the ground,
e.g. 166.723 11.912 100 0, it will drop like a rock when activated and hit
the ground before it has time to start its engines. In this case, you want
its start altitude to either be zero (it will take off just fine) or at
least 1000m above the ground.

The heading component (specified in degrees) e.g. 166.723 11.912 0 90
is rarely used because actors will turn toward their target. This parameter
is included for convenience only. Who knows, you might want to create an
actor who is looking in one direction, while someone is about to cap his
ass from the rear.

4.2 TOOLS - BCSTUDIO MODEL VIEWER

This tool is used to view the 3D models and characters in the game

SETUP
=====

1. Copy bcstudio.exe and bcstudio.ini files to your game install folder.
   It will not work from ANYWHERE else!!

2. Create a shortcut with the desired parameters. See the bcstudio.txt
   for more info on valid parameters

USAGE
=====

Start the program and use File/Open to load a .3D file from the MODELS
folder.

=== THE END ===