STUB Engine Documentation

Overview

The Slitherine TUrnBased Engine (STUB) is an engine designed to be flexible enough to manage almost any turn-based game using a tile approach. It uses scripting in a C-style format to manage almost all of the game and UI tasks, and allows for almost every file in the game to be over-ridden on a per-campaign basis. The main components of the system are listed below.

[Note that this documentation is subject to constant update and addition].

Campaign

A campaign is the high level package for individual missions. It contains a list of the missions, as well as text files holding campaign specific text.

A campaign can also have a script which can control or provide a basis for higher level logic (e.g. You could have enemy behaviour differ based on the outcomes of previous missions, or hook custom UI into flags denoting abilities for your units).

Scenarios

A campaign consists of one or more scenarios. Each scenario consists of a map file (generated by the built-in mission editor) and an optional script file which deals with all the mission specific scripting of win conditions, enemy behaviour, etc.

Units

The 3D game objects are exported to the game in a custom format. The layout of this format is designed to be as simple as possible to create in whichever 3D package is being used.

These objects are then referenced from the squads.csv file (which can be set up per campaign if desired). The squads file defines various values (known in the system as Attribs). Some are required (such as AP or Action Points) while most will be custom Attribs used by the various scripts you use to define and control your game model. You can also arrange Attribs in arrays which can indexed from inside the script files.

All unit behaviour is controlled by script files. A correctly arranged script file creates the onscreen UI options, controls the unit when the order is given, and gives units information on how to react to other units moving. Each object can have its own script file, or you can share scripts amongst them.

UI

All UI objects are defined using a simple text layout. Every UI object can have an associated script which responds to activation, deactivation, and button controls.

Various script commands allow you to affect UI objects and change their state and contents.

Maps

Maps are built from 2 main components. Tiles (used for the ground) and Objects (used for trees, buildings, or anything which is a real 3D object).

Tiles are made up of 2 types, base tiles (which are solid) and overlays (which are alpha-blended). There are multiple layers of tiles available, and tiles can be rotated. It is also possible to build meta-tiles which are more than one game tile in size.

Tiles can also be set at different heights to allow the the construction of simple terrain.

Objects are 3D objects which are placed on the ground. There are various flags which denote how objects affect the tile they are on, such as blocking line of sight, affecting the terrain type of the tile (e.g. adding trees might change the terrain from open to cover), etc.

Bonuses

Bonuses are script files which have special UI hooks which allow them to be allocated to missions in the mission editor. Their scripts control how they look and behave. Examples of a Bonus script might be an airstrike with a cooldown between activations.

Quickstart

Want to jump right in? Want to quickly start trying things out? Just follow these instructions:

1 – Find a campaign directory (e.g. `3Bulge`) from the Campaigns directory in the game install, and copy it into your local `My Documents\My Games\BBCBA\Campaigns` directory.

2 – Copy the `DataSquads.CSV` file into the root of your copy of the campaign directory (e.g. `My Documents\My Games\BBCBA\Campaigns\3Bulge`).

3 – Copy the `Data\Battle\Scripts` directory into e.g. `My Documents\My Games\BBCBA\Campaigns\3Bulge\Data\Battle\Scripts`.

4 – Open the `Text1.txt` file in your campaign. Change the `IDS_CAMPAIGN_NAME` string so you can easily pick it out on the campaign list.

Start the game and you should see your new campaign, ready to go! You can now tweak and change these files without breaking or affecting your main game.

You should NEVER change files in the main install directories, this can stop the game running correctly, and also prevent multiplayer games from playing.

Campaigns

Simple Campaigns

To create a simple campaign, which will act as a container for custom scenarios you might make, you can use the in-game editing tools. When you first select the Editor option from the main menu, you will be prompted to create a new campaign. You enter in the folder name (no spaces allowed) which is the sub-folder where all the files for the campaign will be stored. You also enter a name for the campaign, which can be whatever you like – this will be shown on the campaign selection screen.

You will then be taken to the editor screen, and can create and save scenarios as you desire. Go ahead and play with the editor (there are tips shown onscreen the first time you use each control, plus a helper section later in this document). Once you have your first scenario built, save it and you will be ready to test it out through the campaign selection screen.

In a Simple Campaign, the order in which scenarios appear on the scenario selection screen is not defined.

Advanced Campaigns

If you want to control the order in which scenarios are listed, set up custom text for scenarios, or customise any other aspects of the game when playing your scenarios, you will need to edit additional files. Editing these files is easy, and their contents are explained below.

CAMPAIGN.TXT

This file denotes the order of any campaigns that are included on the scenario selection screen. Note that if this file does not exist, then the game will attempt to use all the scenarios in the SCENARIOS directory.

The form of the file is simply:

```
[Scenario1]

[Scenario2]
```

Where the scenario names are the filenames of your user scenarios. So in this case you would have Scenario1 and Scenario2 BSF files from the editor.

Custom Text

You can have multiple text files inside the campaign directory. They must all be named TEXT<N>.TXT where N can be any number from 0 to 63. For example `text0.txt` or `text12.txt`.

These text files must be UNICODE files, and so support non-ASCII characters [Note: you may need to use applicable fonts to allow the display of non-ASCII characters].

The built in Notepad application in Windows allows you to save UNICODE text files.

The format of these (and all) strings files is:

```
<stringTag>, "<string>",
```

The commas are important! Note that to insert a newline into a string, you need to use the ~ character.

Strings used by the game are:

```
IDS_CAMPAIGN_NAME, "The campaign name shown on the campaign selection
screen",

IDS_SCENARIO_DESC_<scenarioName>, "one string per scenario, shown
over the scenario selection display if there is no custom UI for the
scenario (see below)",

IDS_CAMPAIGN_TEXT, "Text describing the campaign, shown on the
campaign selection screen if there is no custom UI",
```

Some of these strings are set up for you when you create the campaign, others you can add if you want to have more customised text shown when people play your campaign.

Campaign Graphics

There are two files you can create to customise the look of your campaign. These files can be in either TGA or DDS (recommended) format. If either of these files are not defined, then the game uses some default images. 1024X1024 or 512x512 are usual sizes for these two images.

Note: All image files and textures you create for the game should be square and with sizes which are powers of 2 (32,64,128,256, etc), and ideally no larger than 1024x1024.

`BACKDROP.DDS` (or TGA)
This file is used as the backdrop behind your scenarios after you select your campaign to play.

`ICON.DDS` (or TGA)
Shown on the campaign selection screen. The image should be alpha blended (that is, transparent) and should only use the right hand 55% of the image (as it is extended across the whole screen, and should avoid going under the campaign list).

Campaign Scripts

If you create a `CAMPAIGN.BSF` script file, then it will enable you to have campaign-level logic which you can then deal with within your battles.

Campaign scripts make use of campaign variables, which are special global variables defined in the `CAMPAIGN.TXT` file. To define a campaign variable, you need to add a line of the form:

```
VAR <name>
```

for example

```
VAR TotalKills
```

You can then use the `SetCampaignVar` and `GetCampaignVar` commands any scripts that you use, including any campaign script. You are limited to 64 campaign variables, although each one can be accessed as an array of 8 entries (see the documentation for `Get/SetCampaignVar` for more details).

Campaign Script Callbacks

Callbacks

Many times we want to be able to respond to events from within the game. The game uses callback function to do this. These are specifically named functions which are passed information by the game.

You can define certain campaign functions which will be called when specific actions happen in the game. They are:

```
// Called whenever a unit is destroyed.
FUNCTION KILL_CALLBACK(side, id)

// called every time a campaign is loaded up.
// Note:  Not just when a new campaign is started.
FUNCTION INIT_CAMPAIGN()
```

Summary

You can create your own campaigns entirely from within the game, but you can also create custom text and images to change the look of your campaign.

You can also create custom UI, animated briefings, custom units, and even give units entirely different behaviours! How to do this is described in upcoming sections.

Scenarios

Without a special script, a scenario will end once all of the enemy's units are destroyed. But a scenario script allows you to have much finer control of the course of a battle.

The scenario script interacts with the mission through a number of callback functions.

Tick

```
// Called every tick
FUNCTION TICK( side )
```

This callback is called every tick of the model (30 times a second). The side is the side currently having its turn. Use this function to track ongoing events, like the death of units, or movement of the enemy.

StartTurn

```
// Called at the beginning of each turn
FUNCTION STARTTURN( side, mode )
```

At the start of each new turn, this function is called. The mode parameter describes the type of game in progress (e.g. single player, PBEM, etc), and is generally not needed. Use this function to trigger events based on the turn.

Note that this function is called once per turn, and the sequence of turns is as follows:

-1	: This is a pre-turn called when a scenario is first begun
0	: First turn for side 0
1	: First turn for side 1
2	: Second turn for side 0
3	: Second turn for side 1
4	: etc

StartTurnPost

```
// Called after all the default start turn logic has been done
FUNCTION STARTTURNPOST( side, mode )
```

After the StartTurn function is called, there is other internal logic which occurs (for example, updating the APs for each unit, and calling any unit script functions). This function can be used optionally to make changes once all the normal logic and value resets are done. And example is altering the default sight ranges, or number of APs a unit has.

Win_Scenario_Callback

```
// Called each tick.   Can determine whether there is a winner for
the scenario.
FUNCTION WIN_SCENARIO_CALLBACK( winner )
```

This callback if called every tick provided no winner has been selected. The code passes in the side it thinks is the winner (-1 for no winner yet). It returns a new winner value. 0 or 1 for a winning side, and -1 for no winner. Thus it can prevent the default game logic from ending a battle should it need to.

Note that this function is no longer called once a winner is set in any way (e.g. through the `EndBattle` command).

DrawScenarioUI

// Called every frame to render out any scenario specific UI
FUNCTION DRAWSCENARIOUI()

You can use this function to render any custom UI which your scenario might desire. Examples would be tracking achievements (see below), showing a turn countdown if there is a limit in the mission. But it could be anything you like using the UI script commands (see the list of script commands).

Note that this display can be disabled by the user using the toggle which appears just below the Exit Battle button whenever this function is defined in the scenario.

Achievements

Achievements are special events that you can tell the game to look for and congratulate (or berate!) the player. Examples would be for numbers of kills in a timeframe, or before some other event has happened. They are very simple to set up. Here is an example (it is the code used in the tutorial to congratulate you for killing your first enemy):

```
FUNCTION ACHIEVEMENT_FIRSTBLOOD()
{
        // did we kill the only unit on the enemy side?
        if( GetUnitDead(1000) == 1)
        {
                // pop up some UI to tell the player
                AddVizFunctionCall("ShowUIScreen", "BattleHead0",
                "Anim1", "IDS_T1_A1","BHead0Image:achievement_star");

                // return 1, tell the game we got the achievement
                return 1 ;
        }

        // otherwise return 0 to tell the game to keep checking
        return 0 ;
}
```

You create an achievement by adding a function to your scenario script which has a name which begins with ACHIEVEMENT_ and then adding some logic to return 0 when it has not been achieved, and 1 once it has (at which point the game will stop calling to check).

The game looks for a string called IDS_ACH_<name> to use as the tooltip on the stars which appear on the Force Selection and Results screens.

Unit Scripting

The ability to alter the behaviour of the units is one of the most powerful areas of the game scripting. Each unit type can have its own set of scripts and behaviour (although you don't have to do this). As you will see, you can get units to do almost anything with a well written script.

A unit will look for a script which is called <name>.BSF in the DATA\BATTLE\SCRIPTS directory in your My Documents campaign directory, where <name> is the unit name as denoted by the Name entry in the squads.csv file. If the code cannot find a unique BSF file, it will load up a script called $DEFAULT.BSF to handle all the unit's behaviours.

Unit Scripting

The approach that the game itself uses is to have all units use a single set of scripts, but to use the unit attributes (as defined in the squads file) to determine what a unit can or cannot do. This has the advantage that abilities can be added or removed from a unit simply by altering data.

On the other hand, you might have units with very disparate abilities, in which case you might be best using individual scripts for each unit, bearing in mind you can always INCLUDE any shared functionality.

Unit Callback Functions

There are a variety of functions which a unit can have which are called at times by the code.

StartTurn

```
FUNCTION STARTTURN( me )
```

This function is called at the start of a unit's side's turn. This would be used to reset any per-turn counters or limits. Examples would be a unit's shots, or tick up its morale. STARTTURN is a *required* function for every unit.

CustomTrigger

```
FUNCTION CUSTOM_TRIGGER( me, a, b, c, d )
```

The custom trigger is a low level function used to trigger scripting from the unit animation files. It passes through any of the 4 parameters allowed in the animation file for a unit.

Init Functions

```
FUNCTION INIT_*( me )
```

Any function which is named with INIT_ as the first 5 characters will be called on unit creation. You can use this to set up attributes, initialised starting values, etc. Note that if there is more than one INIT_ function then the order they will be called in cannot be guaranteed.

Unit Ability Functions

To actually allow a unit to do anything, it needs to have scripted behaviours. An example can be seen in the Case Study at the end of this document, but the key functionality is described below. Once again we also recommend reading through and editing (in your own custom campaign, *never* in the main game data!) the existing unit scripts to see how it all hangs together.

There are 3 types of ability, depending upon the contents of the tile you are attempting to action. These are TILE_, ALL_, and UNIT_. TILE_ can operate only on empty tiles, ALL_ will attempt to work on all tiles, and UNIT_ will only operate on a tile with a unit present on it.

The key functions which will allow you to begin creating unit behaviours are described below. Note we will show all examples using the ALL_ tag, but this can be replaced with the TILE_ or UNIT_ tag to simplify the scripting as desired.

Note that for all the functions described here, you can omit any of the parameters you don't use, and that the code will correctly assign them no matter where in the param list they are. Indeed, you can have extra parameters in a function header and they will be set to zero if they are called by the code rather than by you.

Checking

```
FUNCTION ALL_CHECK_<name>( me, unit, tilex, tiley )
```

This function is used to determine whether a unit can perform a task. The return value can be either the cost of the action in AP, or -2 if you with the task icon not to be shown, or -1 if you wish to show the icon in a disabled state.

Icon Setup

```
FUNCTION UISETUP_ALL_<name>( x, y, width, height, me, tilex, tiley )
```

This function can be used to setup and draw the order icon. The x,y,width,height parameters are the onscreen positions of the button.

Executing

```
FUNCTION ALL_<name>(me, unit, tilex, tiley)
```

This is the function which is called when the user selects the action for the unit. Note this cannot happen if the check function returns -1 or -2 as the icon will not be selectable.

Scripting Reference

Script Syntax

CScript is a simple scripting language, based on the basic C syntax. It provides a simple framework to hook it up to an application, effectively the function call handling.

Even if you have not had experience with scripting languages before, you can begin to add or alter your custom scripts using just a simple text editor (see the Tools section for details on how to help make editing better with syntax highlighting and other useful features).

The team recommend taking a look at the existing scripts to begin to see the way the scripting works. Beginning by tweaking existing scripts is a great way to see something onscreen quickly.

All scriping must be part of a function. A function has the form:

```
FUNCTION <name>( [<param>], [<param>], ...)
{
        <function body>
}
```

for example

```
FUNCTION Tick( side )
{

}
```

there is a limit of 8 parameters that can be passed to a given function.

All functions return a value, although you do not have to use it, nor use the return should you decide not to. To return a value use:

```
        return <value> ;
```

Note that a call to return will skip any further processing in the function and exit with the value immediately.

Control

You have 2 ways to control the flow of a function. The IF statement, and the FOR statement. All code executed by an IF or FOR statement must be enclosed in a block (within a { } pair).

An IF statement has the form:

```
        if( <condition> )
        {
                <execute if condition true>
        }
        else
        {
                <execute if condition false>
        }
```

Note that the else keyword and following code is optional. Where the condition can be made up of various logical operators such as && (and) and || (or). So an example statement would be:

```
        if( ( a == 10) || ( b == 5 ) )
        {

        }
```

Note that CScript does not have robust bracket handling, and so if you have complex conditionals it is generally best to construct them in a sequence of IF statements.

A FOR statement has for form:

```
        for(<start>; <condition>; <delta>)
        {
        }
```

<start> is a statement initialising the loop variable, <condition> is a check where the loop will continue so long as it is true, and the <delta> statement is a simple expression incrementing (or decrementing) the loop variable. So the statement:

```
        for( i=0; i<10; i++ )
        {
        }
```

would repeat the code inside the block with i having values of 0 to 9 inclusive, before moving onto any following code.

Key Differences with C

- There is no operator priority. Expressions are parsed left to right. Brackets are supported and their use recommended.
- IF statements can can use only a single && or || statement pair when expressions are not contained in brackets. That is:

```
if( a == 0 && b == 0 && c == 0 )                  is invalid

if( (a == 0) && (b == 0) && (c == 0) )            is valid
```

- There is no concept of ELSE IF(...) in the language. You can only pair a single ELSE with any IF statement.
- FOR statements do not allow bracketed terms in the increment (3rd) expression. That is:

```
for( a = 0 ; a < 10 ; a += (10 - c ) )            is invalid
```

- All variables are signed integers.
- String expressions are only allowed as input to function calls. Currently there is no string manipulation functionality.
- Single line IF and FOR result expressions are not allowed:

```
if(a == 0)                                is invalid
        a = 10 ;

if(a == 0)                                is valid
{
        a = 10 ;
}
```

Case Study

Here is a quick example to show how the system works. We are going to add a new command to our units. We assume that this new script file has been included in a larger script which applies it to our units. (e.g. the $DEFAULT.BSF file which is used by any units which cannot find a custom script file).

```
//
// Turn and face a tile
//

// This is the function which actually does the work when we want to activate the
// ability.  The ALL_ prefix tells us we can do this to any tile we like.
FUNCTION ALL_TURN(me, tilex, tiley)
{

          AddVizUnitTurnToFace(me, tilex, tiley) ;
          if(GetUnitTurret(me) == 1)
          {
                    // turn the turret to face the target as well.
                    AddVizUnitTurnToFire(me, tilex, tiley) ;
          }
}


// This CHECK_ function tells the UI how and if to display the option to the user
// when we are over this tile.  The unit param is either the index of any unit on
// the tile, or -1 if there isn't one.
// CHECK_ functions return the cost of the action (0 or more), or -1 to show the
action greyed out (disabled), and -2 to not show the option at all.
FUNCTION CHECK_ALL_TURN(me, unit, tilex, tiley)
{
int ret ;

          ret = 0 ;

          // only allow it to work for tiles <=3 from us
          if( GetDistanceFromUnit(me, tilex, tiley) > 3)
          {
                    ret = -2 ;
          }

          return ret ;
}


// This is how we control the UI for the onscreen button.
FUNCTION UISETUP_ALL_TURN()
{
          // Begin to construct a string
          StartString() ;
          PrintStringLiteral("Turn") ;

          // Now set that string as a tooltip
          SetUITooltip() ;

          // finally define the button texture and colour
          SetUITexture("action_turn", "ffffffff") ;
}


// This function would allow us to do any custom drawing etc on the button as it is
shown
FUNCTION UIBUTTON_ALL_TURN(me, x,y,width, height, tilex, tiley)
{
          // we don't want to show anything fancy this time!
}
```

Creating 3D Models

To create 3D models, you will need access to the MAX modelling software, as well as a paint tool to create the model textures. Before you begin, you will need to download the 3D starter pack from

http://www.slitherine.com/files/bbc_ba/modding/BA_3D_Pack.zip

this contains the exporter script, as well as a couple of example units to give an idea of how to construct them.

[*Note: we are currently in the process of creating exporters and example assets for other 3D modelling packages. If you have a favourite one which isn't supported, let us know on the forums!*]

Installing the Exporter

1 : place the file in the Scripts\Startup folder inside the MAX install folder.
2 : start MAX and go to Customize -> Customize User Interface.
3 : select the Toolbars tab.
4 : in the Catagory dropdown select the SlitherineTools category.
5 : drag the Export S3F entry onto your toolbar.

Using the Exporter

The exporter will export only the selected objects in the scene. When dealing with skinned objects (such as infantry) you should not select the bones, only the meshes. This is because the exporter uses a per-vertex export method, meaning you can animated the meshes any way you want.

You should then use the dialog to set which meshes are static (that is, do not deform, such as tank turrets, etc) and those which do (such as an infantry skin).

Finally export to an S3F file. This file can be read in by the game.

[*Note: when you run the game it will recode any S3F files into S4F files, the binary equivalent, which are much faster to load. It will then load this file from then on. So if you are tweaking the S3F file, you will need to ensure you delete the S4F to see changes in the game. A batch file can be helpful here.*]

Adding Assets to the Game

There are 2 main 3D components to the game, objects and units. We will explain how to use both.

Objects

To create a custom object set, you must:

1 : create an OBJECTS folder inside your custom campaign.

2 : create a folder for each object set you wish to create. Usually one will be enough for a given environment, and can contain many objects.

3 : Build you objects and export them into the folder you have created. Object names should not have spaces in them. The object texture should go in this directory too. Each object can only have a single texture applied to it, although objects in a folder can of course share a single texture (this is recommended for performance).

4 : Create the 3 state textures which are needed. These are all named based on the name of the texture applied to the object (which should be its 'normal' state texture). The 3 additional textures required are:

FOW	name_M.DDS

Required to show the object when it is not revealed

Damaged	name_D.DDS

Only needed for objects which can take damage (see below)

Damaged FOW	name_D_M.DDS

Again, only needed for objects which can take damage

5 : Set up any object data. Objects without any data are fine, and will just sit on the map. But you can define various behaviours for them by creating an entry in the OBJECTS.TXT file. The entry should have the following form. All the entries are optional:

[objectName]	
BLOCKING	object blocks LOS
HIDEDAMAGE	don't show damage on the tile when this object is applied
EDGE	object goes on tile edges (e.g. hedgerows)
HIDES	object is not shown until tile is revealed
FADE	should object fade when a unit is on the tile
RANDROT	place the object with a random rotation
PLACEMENT	limit object placement around the tile center. Values should be between 0 and 16. Smaller values force toward the center of the tile.
HEIGHT <float>	object height (e.g. used when placing the ?)
UIHEIGHT <float>	does the object lift the UI tile display (e.g. bridges)
CLUSTER <float>	do units cluster near the center of the tile. Must be 0 to 1

	inclusive. Smaller numbers cluster towards the center of the tile more.
RANDSCALE <float>	randomly scale up the object when placing. E.g. A value of 1 would scale the object between its original size and 2x its size.
DAMAGE <byte>	amount of tile damage which will cause an object to show as damaged
TERRAIN <string>	placing this object will set the terrain type of the tile to this

If the filename of your object is building.s3f, then the chunk header would be

[building]

followed by any data lines you wanted to use. Check out the OBJECTS.TXT files in the main game folders for examples.

Units

Building a Unit

When building a unit, there are a few special object names and characters to know.

_1turret
You should name the main turret object of any unit with a turnable turret with this name. The game will then rotate it horizontally as needed.

#
Placing a # as the first character in an object name will make it invisible

!
If the code finds a ! in an object name, it will attempt to read the following characters as a number. This number is assumed to be an emitter for various particle effects used in the game. E.g. !0 would define the object as being emitter 0. You will see many objects in the example models with names like #!0 which denotes them as hidden emitter objects.

You should use the example units as templates for which emitters are used. The particle effect editor is currently not ready for use outside the team, but will be coming soon!

Animations

All unit animations need to be included in a single file. The animations will be defined in a text file which will also trigger fx, sounds, etc. There is no fixed order required for animations. All animation will be played back at 30 fps in the game.

Once your unit is built and animated, you should export it to an S3F file. This file need to be placed into a DATA\BATTLE\UNITS folder in your custom campaign. Its texture should go into a DATA\BATTLE\UNITEXTURES folder in your custom campaign directory. A unit can only have a single texture applied to it. If you wish to have a dead texture for the unit, it should be named as per its main texture, but with _D appended, e.g. A unit with MYUNIT.DDS would use a death texture of MYUNIT_D.DDS.

To set up the animation file for a unit, you need to create a text file inside the DATA\BATTLE\UNITS folder which has the same name as your S3F file, e.g. If your unit is called MYUNIT.S3F, then you create MYUNIT.TXT.

[*Note: This is slightly simplified. Specifically, for technical reasons, you would need to name the S3F file **MYUNIT_0.S3F** (that's a zero). But the TXT file and the entry in the assetfilename slot of the SQUADS.CSV file would still remain MYUNIT.*]

Inside the TXT file, you need to set up an entry for each animation you want the unit to use. Most animations will be triggered from the scripts, but two are assumed to exist. These are WAIT and MOVE. You can have multiple animations for a give action, and the code will pick randomly from them when it wants to use an animation. The basic layout for an animation entry is:

[<anim><index>]	the index must be a 2 digit number, in sequence, starting at 00, e.g. WAIT00, WAIT01, etc.
FRM <start> <end>	**required;** the start and end frames of this animation
FX <frame> <effect>	trigger an effect at this frame
DIETEX <frame>	switch the unit's texture to its dead version
CALL <frame> [<a> <b> <c>]	call the unit's CUSTOM_TRIGGER callback, passing in the supplied parameters, all of which are optional and dependent
upon	the callback functionality.
AMBIENT <a> [<b> <c> <d>]	play a sound effect constantly while this animation is active. The entries are the ID values of the sound effects (as defined in the sfx*.txt files). Picks randomly if more than one defined.
SFX <frame> <a> [<b> <c>]	play a sound effect at this frame, pick randomly if more than one is defined.

Examine some of the existing unit .txt files for examples of how these files are laid out.

Once you have your unit data set up, all you need to do is set the assetfilename entry in the SQUADS.CSV file inside your custom campaign, and it will make use of the new asset.

Tools

Grab these free tools to make editing files simple and easy.

Notepad++

You can grab this awesome free editor from

http://notepad-plus-plus.org/

Download and install the app, then set Notepad++ as the default application to open *.BSF files. Then run Notepad++, and go to Settings->Style Configurator...

Click on the C entry in the Language: list, and then enter BSF in the User Ext.: box.

Now simply specify Notepad++ as the default application to open .BSF files and you have an editor which will highlight syntax and also tell you when brackets are missing, along with being awesome.

Open Office

Open Office allows you to open and edit the CSV file used for the unit data. You can download it from

http://www.openoffice.org/

Appendix B: Custom Campaign Folder Layout

Here we describe the folder and file layout of all possible entries you might want to have in a custom campaign.

Not all the files listed here are required, in fact most are not needed to build a simple campaign, but this is included to provide an aid to correct placement of the files needed to add a given feature to your campaign.

See next page for the layout.

```
MYCAMPAIGN
│   backdrop.dds
│   Campaign.txt
│   icon.dds
│   sides.txt
│   Squads.csv
│   Terrain.txt
│   text1.txt
│   uniticons.dds
│
├───AI
│       AI.bsf
│
├───ANIM
│       MyAnim.s3f
│
├───BONUS
│       MyBonus.BSF
│
├───CARDS
│       cards.txt
│
├───DATA
│   │   music.txt
│   │   sfx0.txt
│   │
│   ├───BATTLE
│   │   ├───SCRIPTS
│   │   ├───UNITS
│   │   │       MyUnit.txt
│   │   │       MyUnit_0.s3f
│   │   │
│   │   └───UNITTEXTURES
│   ├───MUSIC
│   ├───SOUNDS
│   └───UI
│       │   Campaign_Overlay.txt
│       │   EndCamp.txt
│       │   SCENUI_MyScenario.txt
│       │
│       └───TEXTURES
│           │   CampaignListIcon.dds
│           │
│           └───ICONS
│                   MyUnit.dds
│                   MyUnit_SLOT.dds
│
├───OBJECTS
├───TILES
├───SCENARIOS
│       MyScenario.bam
│       MyScenario.bsf
│       MyScenario.dds
```

Appendix B: Scripting Commands

This is a list of all the commands available in scripts. This list is generated by the game into the AUTODOCS directory in My Documents\My Games\BBCBA, and so that file will always be up to date, but it is repeated here for ease of access with the other documentation.

```
//returns the turn count.
GetTurn()

//returns the tick in the current turn (in 30ths of a second)
GetTurnTick()

//end the battle setting the denoted side as the winner.  Will be
ignored if the battle already has a winner...
EndBattle(winner)

//return a random number [min, max] i.e. with a min value of min and
a max value of max inclusive.  MUST be multiplayer safe.
Rand(min, max)

//return a random number [min, max] i.e. with a min value of min and
a max value of max inclusive.  Can be used for non-multiplayer/visual
only tasks which are not multiplayer safe
FXRand(min, max)

//take a value [0,100] and scale using Sine.  Use to make random
values clump around 100 more and be less likely to be 0
SinScale(value)

//Output a string to the debug stream, for debugging.  Can have up to
4 values which will also be logged.
Log(string, [value, value, value, value])

//returns the min of the 2 values
Min(a, b)

//returns the max of the 2 values
Max(a, b)

//get a rand between and including the values of attrib[index] and
attrib[index+1]
GetAttribRand(id, string, index)

//return the value of the give attrib for the give unit
GetAttrib(id, string)

//return the value of the give attrib at the index for the give unit
(attrib[index])
GetAttribArray(id, string, index)
```

//return the base value of the give attrib for the give unit
GetBaseAttrib(id, string)

//return the base value of the give attrib at the index for the give unit (attrib[index])
GetBaseAttribArray(id, string, index)

//set the value of the give attrib for the give unit
SetAttrib(id, string, value)

//set the value of the give attrib at the index for the give unit (attrib[index])
SetAttribArray(id, string, index, value)

//create a new attrib (which will be available across all units) does nothing is attrib already exists.
AddAttrib(string)

//create a new attrib array (available across all units) does nothing is attrib already exists.
AddAttribArray(string)

//return the value of the campaign var, if index is not provided 0 is used. index must be [0, 8]
GetCampaignVar(tag, [index])

//set the value of the campaign var, if index is not provided 0 is used. index must be [0, 8]
SetCampaignVar(tag, value, [index])

//returns 1 if in a deployment phase, 0 otherwise
IsDeploying()

//tells the game to skip the first player turn if the mission is set to have a deploy phase. Obviously only useful to call on turn -1 or 0
SkipFirstPlayerTurn()

//returns a map ordinate reworked correctly for the current system (border sizes etc). Should be used anywhere hardcoded coords are used. Deprecated.
MapX(x)

//returns a map ordinate reworked correctly for the current system (border sizes etc). Should be used anywhere hardcoded coords are used. Deprecated.
MapY(y)

//return a value denoting whether the tile is valid. 1 means it is on the playable area, 0 means it is in the border, -1 means it is off the map entirely
IsValidTile(x,y)

//returns 1 if we are currently executing an AI turn in battle mode, 0 otherwise
IsAITurn()

//return the side whose turn it currently is
GetCurrentSide()

//returns the side who is currently viewing the map (used for tile LOS etc, to show what should be seen)
GetShowSide()

//return the id of the currently selected unit. Returns -1 if none
GetCurrentSelectedUnit()

//add a unit to the map. The string is the unit name. If the tile is not clear the unit is not added. Returns the id of the new unit, or -1 if there was a problem
AddUnit(x, y, side, typeString)

//Set a squad to route to the designated tile. react denotes whether the move triggers reaction fire.
SetRoute(id, x, y, [react])

//set a squad route, but factor in the threat map denoted by the threatIndex, currently [0,2]
SetRouteUsingThreat(id, x, y, threatIndex)

//Find the route length for the given squad. -1 if cannot find a route. showroute show it on the map. ignoreunits 0(can't route), 1(try to avoid), 2(route as if not there)
GetRouteCost(id, x, y, [showRoute], [ignoreUnits])

//return 1 if the unit can step to the tile x,y (which must be adjacent). Basically is a shortest route 2 tiles in length.
CheckTileStep(id, x, y)

//return the cost of this tile for this unit. Useful to find whether a unit can sit on a tile, for example.
GetTileCost(id, x, y)

//Route cost for the squad. -1 if cannot find route. showroute show it on the map,use threat map denoted by the threatIndex, currently [0,2]. ignoreUnit 1=target, 2=all units
GetRouteCostUsingThreat(id, x, y, showRoute, ignoreUnit, threatIndex)

//get the X ordinate of the [index] entry into the route just generated by a call to GetRouteCost*
GetCheckRouteX(index)

//get the Y ordinate of the [index] entry into the route just generated by a call to GetRouteCost*
GetCheckRouteY(index)

//return the number of tiles in the route generated by a call to GetRouteCost*
GetCheckRouteLength()

//return the unit side
GetUnitSide(id)

//returns 1 if the id unit is of type string (e.g. Sherman) as per the Name field in the squads file. 0 otherwise
IsUnitType(id, string)

//returns 1 if the unit is of the squad type string (e.g. INFANTRY) as defined in the terrain file
IsUnitSquadType(id, string)

//returns 1 if the unit is moving, 0 otherwise
IsUnitMoving(id)

//get the face that the enemy is presenting to my position. front,side,back = 0,1,2
GetFacingFrom(me_id, enemy_id)

//get the angle the enemy is presenting to my position, 0=looking straight at me, 180=I am directly behind them
GetAngleFrom(me, enemy)

//get the angle the enemy is presenting to the denoted tile, 0=looking straight at me, 180=I am directly behind them
GetAngleFromTile(x, y, enemy)

//get the face that the enemy is presenting to my position. front,side,back = 0,1,2
GetTurretFacingFrom(me_id, enemy_id)

//get the angle the enemy is presenting to my position, 0=looking straight at me, 180=I am directly behind them
GetTurretAngleFrom(me, enemy)

//get the angle the enemy is presenting to the denoted tile, 0=looking straight at me, 180=I am directly behind them
GetTurretAngleFromTile(x, y, enemy)

```
//return the unit's X coord
GetUnitX(id)

//return the unit's Y coord
GetUnitY(id)

//return the X coord of the specified man, in 100ths
GetManX(id, man)

//return the Y coord of the specified man, in 100ths
GetManY(id, man)

//place this unit on the tile.  Fails if the tile is occupied.
SetUnitTile(id, x, y)

//return how many men are in the unit
GetUnitMen(id)

//is this man dead?  0 = no, 1 = yes
GetUnitManDead(id, man)

//is the whole unit dead? 0 = no, 1 = yes
GetUnitDead(id)

//is the unit active (e.g. not dead, loaded, or removed from the map)
0 = no, 1 = yes
GetUnitActive(id)

//does this unit have LOS to the tile?  0 means no
GetUnitLOS(id, x, y)

//does this unit have LOS to the target?  0 means no
GetUnitLOSToUnit(id, target)

//return an id for the unit on a tile, -1 for none
GetUnitOnTile(x, y)

//tell our unit to turn and face the tile
SetUnitTurnToFace(id, x, y)

//flag this unit that enemies can react to what we just did
SetUnitForReaction(id)

//returns 1 if the unit has a turret, 0 otherwise
GetUnitTurret(id)

//returns the status of the unit, [0,3]
GetUnitStatus(id)
```

//sets the status of a unit. Valid values are 0,1,2,3. Changes the icon set used, and can be used to give extra abilities
SetUnitStatus(id, status)

//do an immediate switch for a unit to a given facing [0,359]
SetUnitFacing(id, angle)

//call a unit script each tick until fn returns 0. Can only be checking one fn at any time. Must be defined in the unit script. Can ONLY be used to call functions with up to 6 fixed params, one used forname.
SetUnitCheckFunction(id, function, [value, ...])

//show tracer fire, x,y are the end coords (in 100s), point is index of emitter 3d object, time is the duration in ticks, type is the visual type of the tracer
SetUnitTracer(me, man, x, y, point, time, type)

//set the weather. 0 = none, 1 = rain. 2 = snow.
SetWeather(mode)

//return 0 if the unit could sit on the tile, 1 if not
IsTileBlocked(id, x, y)

//move a unit instantly to this tile. If there is a unit already there, it will be swapped with the current unit
UnitDeploy(id, x, y)

//if the unit script contains the function return the number of params, -1 otherwise
GetUnitFunction(id, functionName)

//call a function on a unit with params in the expected order, limited to functions with 6 params or less, returns the function result
CallUnitFunctionDirect(id, functionName, [value, ...])

//call a function on unit. Assumed to have some/all of the params X, Y, UNIT, ME as per standard menu functions.
CallUnitFunction(id, functionName, x, y, me, unit)

//flag the men in the unit (as defined by the mask) as being dead
DestroyUnit(id, mask)

//remove this unit from play. Flags = 0 just remove from tile, =1 immediate removal, =2 sink out of sight and remove
RemoveUnit(id, flags)

//resurrect the denoted men in a unit. This will remove any dead bodies, will appear in place on the unit tile. WILL NOT work on dead units
ReviveUnit(id, mask)

//return the width of the map
GetMapWidth()

//return the height of the map
GetMapHeight()

//clear threat map index with the provided value [0,255]
ClearThreatMap(index, value)

//get the threat map value from map index at x,y
GetThreatMapValue(x, y, index)

//set the threat map value from map index at x,y. Value must be [0,255]
SetThreatMapValue(x, y, index, value)

//load the toBeLoadedID unit onto the id unit. NOTE: Units can have more than 1 loaded unit
LoadUnit(id, toBeLoadedID)

//does the unit have another unit loaded onto it. Returns the first unit found if more than one loaded, -1 if none found.
GetLoadedUnit(id)

//unload the unit that we are carrying onto the given tile, returns 0 on success, -1 otherwise
UnloadUnit(id, x, y)

//get the terrain cover value for the given x,y
GetTerrainCoverValue(x, y, index)

//get the terrain cover value for the given x,y, based on side's LOS to the tile
GetTerrainCoverValueFromLOS(x, y, index, side)

//return 1 if the tile is a water tile, 0 otherwise
IsTileWater(x, y)

//set the los of a tile. Values are: 0=hide, 1=show
SetTileLOS(x, y, side, los)

//return the LOS value of a tile. 0=hidden, 1=shown, 2 = cover
GetTileLOS(x,y,side)

//return the cover flag for a tile. 1 set means can be hidden in
GetTileCoverFlag(x,y)

//damage a tile by the specified amount. Tile damages ranges [0,255]
DamageTile(x, y, amount)

//play an effect on these men in a unit, using the mask as per AddVizAnim
PlayEffect(id, mask, effect)

//return the distance between the two units
GetDistanceBetween(id, id)

//return the distance between two tiles
GetDistanceBetweenTiles(x1, y1, x2, y2)

//how far is the unit from the tile
GetDistanceFromUnit(id, x, y)

//play one of the defined sfxTypes for the the unit. can optionally specify a bank to play from
PlayUnitUISFX(id, sfxType, [bank])

//play a specific UI sample. can optionally specify a bank to play from
PlayUISFX(sfx. [bank])

//play a world positioned sfx on the specified unit. can optionally specify a bank to play from
PlayUnitSFX(id, sfxType, [bank])

//play a sample at a specified position in the world (x,y are in 100ths). can optionally specify a bank to play from
PlaySFX(x, y, sfx, [bank])

//play a screen shake for N ticks, the scale determines the amount of movement, in 100ths, so 100 is the default x1 value
Shake(ticks, scale)

//tell the game whether or not to show the cover ? markers. 1 = yes, 0 = no
ShowCoverMarkers(show)

//show a text marker at the specified tile center. flags is a mask. 1=show even if tile hidden. id is user defined. must not be zero. colour is a hex string
ShowTextMarker(id, x, y, string, fontid, colour, flags)

//hide the specified text marker
HideTextMarker(id)

//show anim at center of tile xy. id is a user handle to allow for turning off etc-must NOT be zero. Use a negative id to loop the effect. sidemask=optional [0,1] only that side can see. Set MinimapColourString to non-zero to show
ShowEffectMarker(id, x, y, animFile, effect, [showside], [MinimapColourString])

//remove an effect marker
HideEffectMarker(id)

//set the flag for turning on or off the force visible flag. 0 for off, anything else for on
SetForceVisible(id, value)

//set the flag for player control of a unit. 0 for off, anything else for on
SetCannotControl(id, value)

//define which viz queue to use. They both execute symultaneously. Remember to set back to 0 when you are done.
UseVizQueue(queue)

//add a queued anim (string) to the unit, the mask is the men affected (1 == man 0, 2 == man 1, etc) neg mask means play until you tell it play something else
AddVizAnim(id, anim, mask)

//add a queued anim without a wait until the end (string) to the unit, the mask is the men affected (1 == man 0, 2 == man 1, etc. neg mask means play until you play something else)
AddVizAnimNoWait(id, anim, mask)

//move the camera (if needed) to show this tile
AddVizCam(x, y)

//move the camera (if needed) to center the screen on this tile
AddVizCamCenter(x, y)

//move the camera (if needed) to point to this unit
AddVizCamUnit(id)

//insert a pause in the viz queue (and thus the game) of delay ticks
AddVizDelay(delay)

//add a display of text to the tile center, in the colour (colour is a string hex value). alwaysShow means show even if tile is not visible to viewing side
AddVizText(x, y, text, colour, [alwaysShow])

//add a display of text to the unit, in the colour (colour is a string hex value). alwaysShow means show even if tile is not visible to viewing side
AddVizUnitText(id, text, colour, [alwaysShow])

//on unit id, using the mask for men, play the file with the effect (-1 for none) and define the end time (1000ths of a sec, -1 for end of the anim). animFile must be in the campaign or core ANIM dir
AddVizUnitSpotAnim(id, mask, animFile, effect, endTime)

//at x,y (in 100ths) play the file with the effect (-1 for none) and define the end time (1000ths of a sec, -1 for end of the anim). animFile must be in the campaign or core ANIM dir
AddVizSpotAnim(x, y, animFile, effect, endTime)

//turn the unit to face the specified tile. if nowait!=0 then it doesn't pause the main vizq, only the unit.
AddVizUnitTurnToFace(id, x, y, [noWait])

//turn the unit or the unit's turret (if it has one) to face the tile. if nowait!=0 then it doesn't pause the main vizq, only the unit.
AddVizUnitTurnToFire(id, x, y, [noWait])

//call a function from the vizQ. Can ONLY be used to call functions with up to 7 fixed params, as one is used for the name.
AddVizFunctionCall(function, [value, ...])

//Set the value of a global bonus base value
SetBonusBaseValue(name, side, value)

//Get the value of a global bonus base value
GetBonusBaseValue(name, side)

//Get the current value (taking into account player bonuses) of the global value. If base is provided, we actually look for the string base+name as the global
GetBonusValue(name, side, [base])

//set the icon mask. This is used when working out how to tag units. Each of 8 bits add a different icon to the unit. Sets the value, not OR. Only used to pass the value back to the code.
SetIconMask(mask)

//set the icon mask. Used by friendly units to show all the time, rather than to show actionable units for the current unit. Uses same icon set. ORs this value with the current value.
SetUnitIconMask(id, mask)

//turn off the bits in icon mask. Used by friendly units to show all the time, rather than to show actionable units for the current unit. Uses same icon set. Turns off only the bits requested.
ResetUnitIconMask(id, mask)

//begin a new string construction
StartString()

//add the string defined by the string id to the current string construct
PrintString(stringID)

//add the string defined by the string id to the current string construct (e.g. IDS_STRING0 would be accesed by (IDS_STRING, 0)
PrintStringIndexed(stringID, index)

//print the contents of string into the string construct (mainly for spacing, newlines etc)
PrintStringLiteral(string)

//print the value into the construct string
PrintInt(value)

//returns the string height in the requested font
GetStringHeight(font)

//begin a new work string construction. This is for building filenames etc. Up to 16 strings, uses 0 if no index
StartWorkString([index])

//print a literal to the work string
PrintWorkStringLiteral(string, [index])

//print an integer to the work string
PrintWorkStringInt(value, [index])

//return the work string to any function which takes a string argument
GetWorkString([index])

//render the string to the screen. colourString is a hex value (as is borderColourString if used, 0 for not used). Returns the printed text height. Everything done in 1024x768 space
RenderString(x, y, font, colourString, [borderColourString], [width], [height])

//flag an area of the screen as UI. Means the UI won't treat the mouse as pointing to the map and will not show icons or options etc. If tooltip is 1 then it takes the current string and uses it as a tooltip for the area
BlockUIArea(x, y, width, height, [tooltip])

//render an image based on its filename (UI/TEXTURES base dir unless path is given. Resets the image area. Neg width/height flips)
RenderImage(x, y, width, height, colourString, imageName, [path])

//Sets a restricted area to render from on the next call to RenderImage. All values are in 1000ths, so to only draw the bottom half of the image you would use 0, 500, 1000, 500
SetImageArea(x, y, width, height)

//Set the current UI object which is being rendered to the texture and colour (colour is a hex string). Do not use the file extension
SetUITexture(filename, colour)

//set the current string as the tooltip which should be used by the UI for this control (if applicable) use in CHECK and/or UISETUP
SetUITooltip()

//add some info to the tile tooltip which shows as the mouse is over the map. Only really useful in the CHECK fn
AddTileTooltip()

//flags that this option can act as a single click if it is the only option, use in the CHECK fn
AllowUISingleClick()

//allows us to sort the order in which the onscreen order icons are shown. Default priority for all icons is 0. Use in the CHECK fn.
SetIconPriority(priority)

//return the string which is the filename of the icon for the unit whose id is given
GetUnitIcon(id)

//return the string ID for the unit whose id is given
GetUnitStringID(id)

//returns 1 if the key is down, 0 if not. Valid values are 0 (shift), 1 (control)
IsModKeyDown(keyID)

//returns 1 if the UI object with the objectName has the provided id. Use in the handler function to see which object has been actioned
IsUIObject(id, objectName)

//sets the UI object to have the system string (from StartString etc) as its string property (if relevant)
SetUIObjectString(objectName)

//enable or disable the UI object, disable if state = 0, otherwise enable
SetUIObjectEnabled(objectName, state)

//hide or reveal the object. Hidden if state = 0, shown otherwise
SetUIObjectVisible(objectName, state)

//set the texture of the provided object, if possible. If no path, comes from UI\TEXTURES (campaign or core)
SetUIObjectTexture(objectName, textureName, [path])

//turn on tile display - this will show up tiles as per the movement when they have their tile display flag set. Anything which causes a reset of the model, like movement or selecting a new unit, turns it off
ShowTileDisplay()

//turn off the tile display
HideTileDisplay()

//set a texture for a tile display flag value, max of 4 atm
SetTileDisplayTexture(index, texture)

//returns 0 if tile display mode is off, 1 if it is on
GetTileDisplay()

//clear all the tile display flags
ClearTileDisplayFlags()

//set the tile display flag for this tile. -1 is off, any other value is the index of the texture you want to show [0,3] atm
SetTileDisplayFlag(x, y, value)

//return the X ordinate of the Nth AI point
GetAIPointX(id)

//return the Y ordinate of the Nth AI point
GetAIPointY(id)

//are all the units in the team at or as close as they can be to the destination. 0 for no, 1 for yes
CheckTeamDestination(side, team, x, y)

//set the done value of the team
SetTeamDone(side, team, value)

//return the done value for the team
GetTeamDone(side, team)

//set the team destination
SetTeamDestination(side, team, x, y)

//return the X ordinate of the team destination. Usually use -1 for none set
GetTeamDestinationX(side, team)

//return the Y ordinate of the team destination. Usually use -1 for none set
GetTeamDestinationY(side, team)

//return the aggression value
GetTeamAggression(side, team)

//set the aggression value
SetTeamAggression(side, team, aggr)

//set the target of the team to be the unit index (use -1 unit index to flag no target)
SetTeamTarget(side, team, unit)

//returns the team targeted unit, or -1 for none
GetTeamTarget(side, team)

//return an average team position X
GetTeamX(side, team)

//return an average team position Y
GetTeamY(side, team)

//return the data from the team. The index can be [0, MAX_TEAM_DATA]. 0 is generally used to denote any AI point that the team will head for, but this is dealt with in the script
GetTeamData(side, team, index)

//set the team data
SetTeamData(side, team, index, value)

//make a list of the units closest to the any member of the team - there is only one list. Returns the number of units on the list
MakeClosestUnitsToTeamList(side, team, [includeHidden])

//make a list of the units closest to the unit - there is only one list. Returns the number of units on the list
MakeClosestUnitsToUnitList(id, [includeHidden])

//returns the smallest distance from the tile x,y to any unit in the defined team, or -1 if the team does not exist (or is all dead)
FindDistanceFromTeam(side, team, x, y)

//get the closest unit on the current closest unit list. -1 if invalid in some way
GetClosestUnit(index)

//Set the destination of this unit. Must be a valid tile
SetUnitDestination(id, x, y)

//clear out the unit destination (set to -1, -1)
ClearUnitDestination(id)

//return the X unit destination. -1 for none set
GetUnitDestinationX(id)

//return the Y unit destination. -1 for none set
GetUnitDestinationY(id)

//set a target for the unit. -1 for none
SetUnitTarget(id, target_id)

//return the id of the target unit. -1 for none
GetUnitTarget(id)

//return the team ID of the specificed unit
GetUnitTeam(id)

//set the team value of the specified unit. team should be [0,7]
SetUnitTeam(id, team)

//returns the number of sides in the game (usually 2)
GetSideCount()

//returns the unit count for the side. Includes ALL units including those that are dead or inactive.
GetUnitCount(side)

//returns 0 if a unit is dead/hidden/inactive (loaded, etc). 1 otherwise
IsUnitValid(id)

//return the unit id of the Nth unit in a side
GetUnitID(side, index)

//set a route to a destination, but the actual route set is the route which costs up to the cost value in AP
SetRouteWithCostWithThreat(id, x, y, cost, threatMapIndex)

//set a route to a destination, but the actual route set is the route which costs up to the cost value in AP
SetRouteWithCost(id, x, y, cost)

// Call bonus function. The function is called from a bonus script. Returns function return value. If the function is not in the mission, returns -999. If the fn does not take a param it will be ignored
CallBonusFunction(functionName, side, x, y, unit)

//same as ShowUIScreen below, but sets this config as being the briefing screen so it can be reviewed later again by the player. Does not show the screen.
SetBriefingScreen(screenName, animName, textTag, [imageTags])

//show the UI object screen, setting up with text of the form textTag_objName for each object. Image tags are 1 or more obj:file pairs seperated by a $
ShowUIScreen(screenName, animName, textTag, [imageTags])

//show the briefing screen (if set up). Just a helper to avoid having to maintain 2 data sets
ShowBriefing()

//hide a UI screen which is showing
HideUIScreen(screenName)

//Hides the custom UI if state is 0, otherwise shows it
ShowCustomUI(state)