WWW
HTTP://WWW.AURAN.COM

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

The Official
Tactics Engine
Artificial Intelligence Personalities (AIP)
and
Scenario End Conditions
Guide

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Conditions of use

Permission to use the manual contained in this site ("Manual") is conditional upon you agreeing to the terms set out below. Do not proceed to download the manual until you have read and accepted all the conditions. If you do not wish to accept the conditions, do not download the Manual.

The user is granted a non-exclusive licence to download and use the Manual contained in this site on the following conditions;

1. The Manual download from this site remains the property of Auran;

2. The Manual downloaded from this site, in whole or in part may be used:

(a) to configure the Tactics software engine; or

(b) in conjunction with other software packages or programs,

provided that:

(c) such use is not for a commercial purpose or any financial gain; and

(d) these conditions are included without alteration wherever the Manual is reproduced;

3. Auran cannot warrant the performance or the results obtained from using the Manual contained in this site;

4. Auran makes no warranties, express or implied with respect to the Manual as to merchantability or fitness for purpose or nonconfringement of third parties rights but in the event that any legislation implies terms which cannot be lawfully excluded, such terms will apply except that the liability of Auran for breach of any such implied term will be limited to replacement of the Manual to which the breach relates or the supply of an equivalent manual;

5. Auran will not be liable for any damages whatsoever including;

(a) direct, indirect, incidental, consequential damages; or

(b) loss of business profits; or

(c) special damages,

arising from the use of the Manual contained in this site, even if Auran has been notified of the potential for such damages to arise;

6. The user acknowledges:

(a) they have not made known to Auran any particular purpose for which the Manual contained in this site is required; and

(b) they have not relied on Auran to provide the Manual as being suitable for any such purpose.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

CONTENTS

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Introduction

The Artificial Intelligence (AI) in Dark Reign is extremely configurable. The mission designers have control over the strategic behaviour of the computer-controlled teams and the mission designers and players have control over the behaviour of the individual units (computer controlled and human controlled, respectively). This document is intended to explain the full scope of the AI behaviours and to serve as a user's guide for designing appropriate AI behaviours for missions.

The AI is separated into two major components: *tactical* and *strategic*. The tactical AI is responsible for the behaviour of individual units. It determines whether a unit fights or flees, and whom a unit targets when a fight begins. The strategic AI decides where a computer-controlled team sends it troops, as well as what troops and buildings it builds. The tactical AI information is relevant for both the human and the computer controlled teams, and the strategic AI information is relevant only to the computer controlled teams.

An integral part of the AI is the FSM – the Finite State Machine. This is the mechanism used to set conditions which may either end the game or trigger some definable action.

Tactical AI

Units in both the computer teams' armies and the human teams' armies use the tactical AI system. The two major components of this system are the *targeting system* and the *unit behaviour system*. The targeting system decides at whom a unit should shoot. The unit behaviour system is used to determine when a unit should enter combat, seek repair or healing, or ignore an immediate threat in order to satisfy a higher level strategic purpose.

Targeting System, Unit Behaviour System and Projectile Response

Who is informed?

At present, all units within a specified radius of a projectile's impact will have an opportunity to react; of course, the unit that is hit directly by the projectile is also informed and has an opportunity to respond. The radius chosen is dependent upon several factors, but it will always encompass ALL units that are damaged by the projectile (within the projectile's area of effect). In general the radius is the maximum of the following: the projectile's area of effect and a radius specified by the designers for the type of target that the projectile hit. Currently there are two different defined radii, one for units/ground and one for buildings.

Who responds?

Of the units that are informed of the explosion, not all of them will respond to the projectile. The various factors are described below. In general, a unit will not respond. In order for a unit to respond, it must meet at least one criterion that says "will respond" but no criteria that say "won't respond". In other words, if a unit qualifies for response for one reason but not another, it won't respond.

Damage

Units that take damage from the projectile will respond.

Relationships to shooter and target

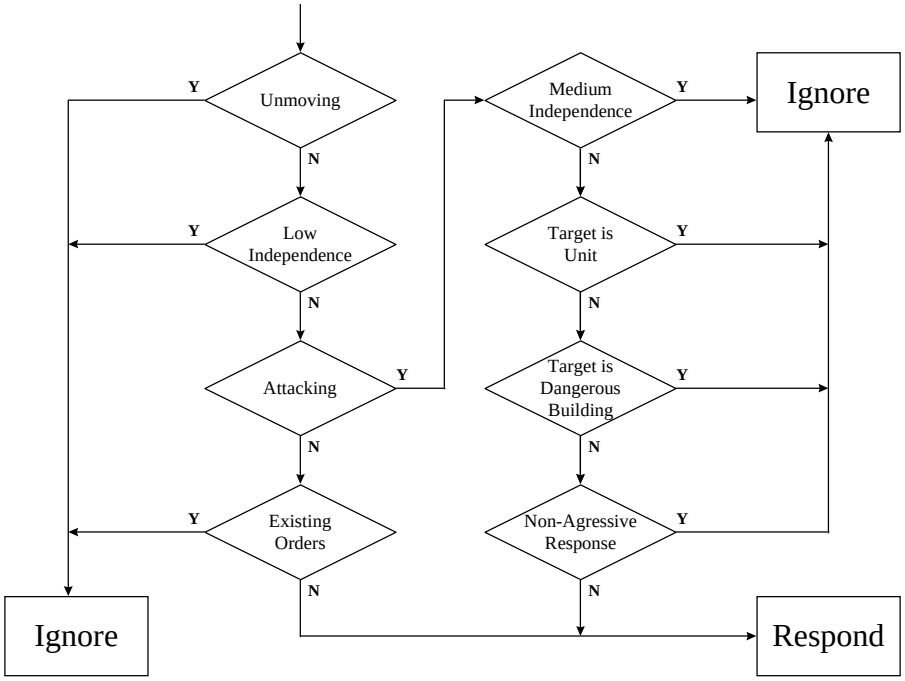
Units allied to the target unit's team (notice that this includes the target team itself) will respond if they have a weapon that can damage the shooter, which must be an enemy to that team.

To restate:If the unit is an ally of the target and an enemy of the shooter and can hurt the shooter then it will respond.

Tactical AI settings

There is a separate flowchart for the tactical AI. "Ignore" means that the unit won't respond and "Respond" means that the unit will respond.

Tactical AI Projectile Response Flowchart



What is the response?
The actual response of the unit is dependent upon several factors as well. A responding unit that can attack the shooter will run towards the shooter a distance equal to the radius of response that notified that unit. In other words, if the projectile hit a unit, the responding units within the defined unit response radius from the target will run a distance equal to the unit response radius. If the projectile hit a building, the responding units within the defined building response radius from the target will run a distance equal to the building response radius.

A responding unit that cannot attack the shooter will run a distance as far as it can see directly away from the shooter.

In both of these cases, if the unit (for any reason) cannot reach the target destination, it will move to a random location within the radius of its seeing range.

What happens then?
First of all, if a unit was doing anything in terms of an order (moving, attacking, etc.) and decided to respond to the projectile, it will return to what it was doing after responding. If the unit was doing nothing, and its pursuit range is less than high, it will return to its initial location after responding.

Strategic AI

Dark Reign's strategic AI can be divided into three major components: a *troop allocation system (TAS)*, a *building and unit construction system (BUCS)*, and a *finite state machine (FSM)*. The principle concepts behind the Strategic AI system are that in a game as configurable as Dark Reign, the computer had better be able to play a *decent* game in any situation and that a serious mission designer should be able to customize the AI for a particular mission so that the gameplay is more exciting.

The troop allocation system and the building and unit construction system primarily address the first principle, that the computer should play a decent game in any situation. There are two goals: 1) a designer should be able to generate a map and populate it with units and be able to play a game right out of the blocks in order to streamline the initial design process, and 2) the computer should

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

be able to react to the current game situation and not depend on pre-scripted events so that the gameplay is more dynamic. The TAS periodically evaluates the state of the game, defines and ranks certain strategic goals, and allocates the appropriate troops to those tasks. In tandem, the BUCS can be used to have the computer build a base for the computer team and to supply the TAS with troops for allocation.

The Finite State Machine is a system that allows the designers to set a variety of triggers in the game that will allow in-game modification of the TAS and BUCS. Additionally, the FSM lets the designers set specific troops aside for special events in the game. With the FSM, the designers can have the strategic behaviour of a computer team change in the appropriate circumstances.

The Troop Allocation System

The Troop Allocation System matches a team's forces with that team's strategic goals. This can be a fairly CPU intensive activity, so it is broken down over a number of game cycles. Furthermore, it is only initiated periodically (roughly every 5-30 seconds depending on how the designer sets it up).

Overview of what the TAS does

The first step in Troop Allocation is to evaluate the playing field. This is done by dividing the entire map with a coarse grid, and evaluating each grid cell as a strategic goal. For a medium sized map, this could be a 10 by 10 grid or more. Some cells have no strategic value, and are dropped from consideration in this strategic planning cycle.

The next step is to group our forces into squads. Pre-existing squads are usually left alone, and units that have not yet been assigned a squad are grouped with other units in the same gridcell.
At this point, the matching phase begins. Each usable squad of units is compared against any likely strategic goal. Every squad-goal pair is called a "matching". Each matching is assigned a numeric value according to a designer-defined matching function. For goals in regions that the AI team knows about (can see or has seen), the matching function is just a linear combination of the estimated enemy threat in that region, the distance from the squad to the goal, the number of enemy buildings in that goal region, the number of our buildings in that goal region, a scripted value assigned to that goal by the designers, and whether the squad was previously assigned to that goal (persistence priority):

Formula 1:

matching_value = team's threat priority X cell's threat value

- team's distance priority X distance from squad to cell
- team's defend buildings_priority X number of our buildings in cell
- team's attack enemy base priority X num of enemy buildings in cell
- team's scripted priority X scripted_value
- team's persistence priority (if this is the same goal as last time!)

If the goal region is unexplored, the matching value is a linear combination of the distance from the squad to the goal region, a priority value for searching unexplored regions, the scripted value for the region, and the persistence priority:

Formula 2:

matching_value = team's distance priority X distance from squad to cell

- team's scripted priority X scripted_value
- team's exploration priority;
- team's persistence priority (if this is the same goal as last time!)

Once a matching is created for each possible squad-goal pair, the computer finds the best matchings (highest matching values). It then allocates troops from the squads to the goals of the best matches until the most important current goals have enough troops committed to them to fulfill their requirements. If the system cannot allocate enough troops to fulfill a goal's minimum troop requirements, it will not allocate any troops to that goal.

At the end of a matching cycle, the TAS should have a large number of goals fulfilled. There may be units from multiple squads attached to a particular goal, so the old squads are eliminated and new squads are formed out of all the units allocated to a single goal. The TAS then sends each revised squad on its way to the goal to which it's matched.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

What can the designer control?

Through the AIP files (files with extensions **.aiP**), the designer has control over enough parameters of the TAS to create exquisitely balanced AI play or truly psychotic behaviour. Each computer-controlled team has a current AIP that defines how the TAS and BUCS work. The FSM is responsible for switching AIPs at the appropriate times.

The beginning of a sample AIP file might look like this:

```
//*****************************************************************************
//Default.aiP
//  *****************************************************************************

#include "aiPdef.h"
// How often do we recompute the strategy?
int recompute_strategy_period = 100;

// PRIORITIES
int threat_priority = 150;
int distance_priority = -3;
int defend_buildings_priority = 30;
int attack_enemy_base_priority = 75;
int persistence_priority = 30;
int exploration_priority = 50;
int scripted_priority = 50;

// TROOP COMMITMENT STUFF
double max_matching_force_ratio = 3.0;
double min_matching_force_ratio = 1.0;
double max_building_defense_force_ratio = 2.0;
double min_building_defense_force_ratio = 1.0;

// RELAXATION STUFF
int relaxation_cycles = 1;
float relaxation_coefficient = 1.0;
```

The first few lines are just comments. In fact, the AIP files use the same commenting conventions as the programming languages C & C++. Any text following the characters "//" on a line is ignored. Additionally, any text between the characters "/*" and the characters "*/" is also ignored. Thus you can put anything you want in a comment, and it will not affect the behaviour of the AIP, or its ability to be loaded properly by the Dark Reign game.

Any variable which is preceded by "int" is an integer variable (no decimal points). Any variable preceded by "float" or "double" is a rational number (decimal points are allowed). The difference between float and double is something related to how the computer stores information, and all a designer has to do is use whichever one is specified.

Line-by-line

Here are what each line controls:

```
// How often do we recompute the strategy?
int recompute_strategy_period = 100;
```

A planning phase can take several hundred game cycles to finish. Once it's done we wait some number of game cycles before we start again. The value assigned to "recompute_strategy_period" is the number of cycles that team will wait until starting the next cycle.

```
// PRIORITIES
int threat_priority = 150;
int distance_priority = -3;
int defend_buildings_priority = 30;
int attack_enemy_base_priority = 75;
int persistence_priority = 30;
int exploration_priority = 50;
int scripted_priority = 50;
```

The "PRIORITIES" section of the AIP file contains the coefficients used in computing the matching values in Formulas 1 & 2.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

```
// TROOP COMMITMENT STUFF
double max_matching_force_ratio = 3.0;
double min_matching_force_ratio = 1.0;
```

For every viable strategic goal, the computer must compute how much troop strength is needed to accomplish that goal. In the simplest case, the goal is simply an attack on a concentration of enemy strength. The computer has a formula for computing the strength of any unit on the board (a combination of many things including firepower, firerate, & hitpoints). The min_matching_force_ratio and max_matching_force_ratio say how much strength we must commit to a goal in order to continue to pursue it (min) and how much strength we'll consider more than enough (max). The point of this is to make sure we commit enough troops to a goal in order to stand a reasonable chance of accomplishing the goal, while not overcommitting troops to the neglect of other goals.

As an example, assume that all units had a strength value of 1. If a given goal has 10 enemy units in it, our min and max matching force ratios would mean that we wouldn't send any troops to that goal if we couldn't muster up at least 10. Also, we wouldn't send more than 30 because that would be wasteful overkill.

```
double max_building_defense_force_ratio = 2.0;
double min_building_defense_force_ratio = 1.0;
```

Currently, these two values don't mean exactly what they say. What they are used for is to allocate extra troops to goals where there are buildings (ours or the enemy's). If there's one building in a goal's region, we'll add at least one unit of average strength to that goal and no more than two. If there were two buildings, it would be two and four, respectively.

```
// RELAXATION STUFF
int relaxation_cycles = 1;
float relaxation_coefficient = 1.0;
```

The current scheme for calculating the threat value of a region is to add up the strength of enemy units in that region. If that were all, however, a region with one or no units would look to be pretty weak even though the next region over might have a hundred enemy units in it. Thus, we perform a "relaxation" of the borders of regions to let the threat value of a region bleed over into its neighboring regions. "relaxation_cycles" says how far from any region to allow the bleed to go (1 means just to the neighbor). The relaxation coefficient says how much of the threat to let bleed over.

Details: A coefficient value of 1.0 means that we should add the whole threat value of a region to its neighbours. A value of 0.5 for the coefficient and two cycles would mean that we perform two cycles in which we add half of each region's values to each of its neighbours. If there was one region with a threat of 100 (an entirely arbitrary number), after the first cycle, the regions immediately next to it would have a threat value of 50, and the one region would still have a threat value of 100. After the second relaxation cycle, regions two away from the original region would have threat values of 25, while the one step away regions would have threat values of 100 (we add another 50!), and the main region would have a threat value of 200 (if it had four neighbours, each of which had a value of 50 after the first cycle). The threat values are then scaled down so that the original value is 100 (representing real troop strength again), the 1-neighbours have 50, and the 2-neighbours have 12.5.

The Building and Unit Construction System

The TAS and the first part of the AIP files let the designers direct the personality of the AI team in how it commits its available forces to battle. The next part of the AIP files, though, lets the designers specify what forces are available in the game. The starting forces, of course, are set up in the map editor, but the Building and Unit Construction System (BUCS) enables the designer to tell a team what buildings and forces to make in the course of a game.

Accounts

The principle conceit in the BUCS is the "Account". Each account represents a linear construction program: "Build some of these, then when that's done some of these, then when that's done some of these others, etc.". Most importantly, though, there can be multiple accounts. Each account has a linear construction program, each has some money in the bank, and each gets some portion of the incoming money at any given time. Multiple accounts with their own money allow the designers to guarantee that the money gets spread appropriately between a number of different needs.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Account Elements
The most basic component of an account is the Account Element, or element. Each element consists of an item name of a building or unit type to build, a priority level for building it, a method for building it, and an amount to build.

Item Name
The item name is the name of the item as it appears in **units.txt** (formerly **parts.txt**). The item can be a building type or a unit type.

Priority Level
The priority level of an Account Element is just an integer. The higher the priority the sooner that will get built by an account. The numbers don't correspond to anything in the game in particular, but the priority levels still mean something between two accounts for the same team.

To explain: Each account will only be building things from one priority level at a time. It will not move on to anything at a lower priority level unless everything at a higher level has been finished. However, different accounts can be at different priority levels at the same time. This only becomes important when two accounts both need to use the same facility to produce an item (a building to make a unit, or a construction crew to make a building). When there is such a conflict, the priority levels are used to decide which account gets the facility. It's handled as follows: if one account is building at priority level n, and another is at level m, then n times out of (n+m) conflicts, the first account will get the facility in question, and the other m times the other account will get the facility. Example: account "Defense" is at priority 3, and account "Offense" is at priority 6. Both need to build things out of a unit training ground. "Defense" get to build something 3/9s, or 1/3, of the time, and "Offense" will get to start its construction the other 2/3s of the time.

Build Method and Build Amount
There are four available build methods: NUMBER_TO_HAVE, NUMBER_TO_BUILD, RATIO_TO_BUILD, and RATIO_TO_HAVE. The "NUMBER" methods can be used at any priority level for an account, and terminate after a certain finite number of units have been built. The "RATIO" methods keep turning out units forever (or until the account runs out of money!); this precludes them from being used anywhere other than at the lowest priority level for an account. ***NOTE: All elements at the same priority level for an account must use the same Build Method!!!!***

NUMBER_TO_HAVE means we keep building this element until there are "Build Amount" number of that item on the map for our team. Thus if we have NUMBER_TO_HAVE of seven tanks, and we already have three tanks on the map, we'll build four more.

NUMBER_TO_BUILD means that we just build "Build Amount" number of that item. If we specified NUMBER_TO_BUILD seven tanks, and we had three tanks on the map already, we'd end up with 10 (assuming the enemy destroyed none in the intervening construction time).

RATIO_TO_BUILD means that we just keep cranking out all of the units at this priority level in the specified ratios until the account money is exhausted (which doesn't have to happen if the AI team gathers resources quickly enough!).

RATIO_TO_HAVE is slightly more complicated in that it looks at the map and sees how many of each unit of the specified units (at that priority level) the AI team already has. It then makes enough of each element to make the ratios correct. Then it defaults to RATIO_TO_BUILD to keep turning out units in the right ratio. For example, if we specify one medic per five infantry using RATIO_TO_HAVE, and we already have 10 infantry, but no medics, we'll end up building two medics immediately, and then we'll crank out medics and infantry at the rate of one medic per five infantry.

Construction Order
The way the BUCS works is this: at any point in time, each account tries to fulfill the highest priority fulfilled element in its construction program. If there are several elements with the same priority, the account tries to build all of the elements at the same time. Only when each element of the same priority has been fulfilled within an account will elements of lower priorities be considered. Furthermore, since there are multiple accounts, each account is trying to build its most pressing elements.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Replacing Destroyed Units/Buildings

If a building or a unit is destroyed in the course of a game (or used up in some other fashion, such as construction crews), the current priority level of an account can be set back. That is, if we have specified at a high priority level that we have a "NUMBER_TO_HAVE" of some unit or building, and we ever don't have that many, then we return to the that unsatisfied priority level for that account. For example, if we specified that we must have two construction crews with a priority of 9 in one account, and we build them and go on to a lower priority level to crank out infantry, but one of the construction crews is destroyed, we'll stop making infantry and replace that construction crew immediately. Once the replacement is made, we return to the level we were at before. This can be used effectively for making sure our base is maintained properly (always keep a headquarters, etc.).

What if I can't build that?

If the prerequisites have not been satisfied for some unit which an account is saying to build, guess what? It won't get built. Furthermore, we will not progress past that priority level, although we will still make other things at the same priority level. It is up to the designers to make sure they don't specify things that cannot be built. Note that one account can specify the prerequisites for something that is built by another account. This is not necessarily recommended, but it can be a way for one account to save up money for some advanced unit while another account blows all of its money on making the facility for that unit.

The BUCS and AIPS

The syntax for the BUCS part of the AIP files is even more important than for the TAS part. If anything is incorrect (leaving out a semi-colon, adding an extra comma or other punctuation, mis-spelling #END_DATA, et cetera), the whole program can fail to load. Remember that white space and comments are irrelevant (fortunately).

The first part of the BUCS section of the AIP says how many accounts there are and names them:

```
////////////////////////////////////////////////////////////////////////////
// The UCP Data
// ————————————
// This specifies what type of units to build.
UNIT_CONSTRUCTION_PROGRAM unit_construction_program[MAX_ACCOUNT_COUNT];
#DATA
//  Which Account       BUDGET
//————————————————————————-
   "Slush_fund",        UNLIMITED;
   "Base_building",     50;
   "Offensive",         50;

#END_DATA
```

The line that starts with "UNIT_CONSTRUCTION_PROGRAM" ***must always be the same***. Don't mess with it. Ditto with the "#DATA" and "#END_DATA".

What you can change is between the #DATA and #END_DATA. Each line consists of an account name in quotes (no white space within the quotes, please!). The name must consist of all letters and numbers and underscores, and must start with a letter. Valid names might be "Slush_fund", "Offense1", "My_bologna_has_a_1st_name", or "D232_1". Invalid names would be "My bologna has a first name", "123_go", or "my.account".

The budget is some portion of the team's money that the account gets. The budget numbers don't have to add up to 100 or anything like that (remember, computers are good at figuring out stuff like that). One special budget amount is "UNLIMITED". This should only be used for very special accounts with only a few things in them, because these are accounts that are considered to have first dibs on any money they need. In fact, you should only have one such account, and it should only contain a few things that the team absolutely needs to have on the board, such as a headquarters, some extra construction crews, some power plants, or something else like that. Don't ever put "RATIO" stuff in an "UNLIMITED" account, because then no other account will ever get any money.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Syntactically: There must always be a comma after the account name (which is enclosed in quotes), and a semi-colon after the budget amount.

```
// Build type can be NUMBER_TO_HAVE, NUMBER_TO_BUILD, RATIO_TO_BUILD, or RATIO_TO_HAVE

////////////////////////////////////////////////////////////////////////////
// Slush_fund
// ————————————
// This specifies the baseline super-critical account
ACCOUNT_ELEMENT Slush_fund[MAX_ACCOUNT_ELEMENTS];
#DATA
// priority   item name                build method            build amount
//————————————————————————————————————————————————————
      9,     "fh",                    NUMBER_TO_HAVE,       1;
      9,     "FGConstructionCrew",    NUMBER_TO_HAVE,       3;
      8,     fgpp",                   NUMBER_TO_HAVE,       1;

#END_DATA



////////////////////////////////////////////////////////////////////////////
// BASE_BUILDING
// ————————————
// How do we want to go about building our base?
int Base_building_count = 3;                  // How many items are in the Base account

ACCOUNT_ELEMENT Base_building[Base_building_count];
#DATA
// priority   item_name                build_type              build_amount
//————————————————————————————————————————————————————

      8,     "fglp",                  NUMBER_TO_HAVE,       1;
      8,     "fgmn",                  NUMBER_TO_HAVE,       1;
      8,     "fgww",                  NUMBER_TO_HAVE,       5;


#END_DATA


////////////////////////////////////////////////////////////////////////////
// OFFENSIVE
// ————————————
// What sort of offensive units and support structures do we want
int Offensive_count = 6;                      // How many items are in the Offensive account
ACCOUNT_ELEMENT Offensive[Offensive_count];
#DATA
// priority    item_name               build_type              build_amount
//————————————————————————————————————————————————————
      6,     "fu",                    NUMBER_TO_HAVE,       1;
      6,     "fc",                    NUMBER_TO_HAVE,       1;

      5,     "SpiderBike",            RATIO_TO_BUILD,       1;
      5,     "HWFreedomFighter",      RATIO_TO_BUILD,       1;
      5,     "FreedomFighter",        RATIO_TO_BUILD,       2;
      5,     "TankHunter",            RATIO_TO_BUILD,       1;


#END_DATA
```

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

The Finite State Machine (FSM)

The FSM is the highest level of AI for a computer-controlled team. To simplify, the Tactical AI determines the behaviour of individual units, the AIPs determine the current personality and priorities of an AI team, and the FSM switches AIPs to match the current strategic situation. Each computer-controlled team has an FSM for determining its behaviour. Additionally, each team in a game (even each human team) has an Endcondition Tree, which is just an FSM that can cause that team to win the game if a certain state is reached.

The FSM actually does more than just switch AIPs. The FSM consists of any number of "States". Each State contains actions that occur when that state is entered and criteria for switching to other states. One of the actions is switching AIPs; others may involve sending particular units to certain places or changing some priorities on the map. Criteria for switching state are often called triggers. These are checks that are made by the AI to see if particular events have occurred or if particular conditions have arisen (time passing, enemy troops reaching large numbers, etc.).

Thus, a simple FSM could have three states: a construction-oriented state, and offensive state, and a defensive state. When the game starts, the AI team might start in the construction-oriented state that would load an FSM with a rich BUCS. Once a specified amount of time has passed, it could switch to an offensive AIP. If it's getting trounced on the field, it could switch to a defensive AIP and so on...

More complex FSMs can have some knowledge of the map in question. They could know that if units from a particular team reach a certain region they have to send troops there right away. Or a designer could give team very few credits to start with, but if they accomplish some task (destroying a critical building on the enemy team), they could get extra cash as a reward.

There are no practical differences between Endcondition Trees and FSMs, except that when an Endcondition Tree changes state to state 0, the team that has that Endcondition Tree wins the game. Also, some FSM actions are not relevant for human teams, and cannot be used in Endcondition Trees.

General notes

The first thing to know is that designing an FSM is computer programming. As such it is inherently NOT EASY. There are many ways to create bad FSMs, and getting the proper behaviour out of a complicated FSM is both an art and a science.

FSMs consist mostly of ***states***, and states consist of ***actions*** and ***criteria***. Actions are things that happen when a state is entered, and criteria are conditions that have to be met to cause a change of state. As with any computer programming conceit, actions and criteria have names and can accept parameters. The names tell the computer what kind of thing to do, and the parameters specify the exact details. For an everyday example, the name of an action could be "WashACar" and a parameter could be "MyPorsche". Additionally, as with any computer programming, there is an appropriate ***syntax*** for expressing each action or criteria. The syntax is how you say (or write out) an action or criteria; it includes the grammar and punctuation. So, an action for car washing might need its parameter (which car) expressed in parentheses as follows:

WashACar(MyPorsche)

In the Dark Reign FSMs, there are two parameter types, *number* and *name*.

- Numbers are always integers (whole number). Parameters are not allowed to be negative at the moment.
- Names are names of things. For example, the SetAipFile function takes the name of a file.

In the syntax descriptions, we use the following additional notation

- … means you can specify this parameter as many times as required. For example, DefineSpecialForces unit_id … means many unit ids can be specified in the list of unit ids. Note that each parameter is separated by a space.
- … also appears after 'actions' in a conditional state. This means this action can be performed multiple times by specifying the command again with different parameters (e.g. TriggerSpecialForces, multiple Special Forces groups can be triggered on entry to a conditional state.).
- Note that when a building_id or unit_id is required, this means that the building or unit must

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

be on the map at the beginning of the game (ie. created by the scenario editor). The ids can be found in the scenario file (.scn) for a scenario.

- The AI Conditional Tree is specified in a separate .FSM file. Three separate FSMs can be specified for each team; in future versions of the engine these will specify AIs tuned to difficulty levels in order from left to right: easy, medium and hard. These FSM files in turn can specify different AIP files, criteria, and actions.
- The SetFSM, SetAIPFile and #include commands (with an AIP file only) all search the current scenario directory first for a matching file, and then they search the \AIP directory. This allowscommon files to be shared.
- Comments: Any text on a line after a semi-colon is ignored (treated as just a comment for the designer to use to remember what's going on).

Scenario File Definitions

File Structure
Basic Structure of the .scn file

```
'scenario stuff'          // includes terrain set, scenario version 'team stuff'
'team stuff'              // settings for each team – credits, end condition / FSM.
'scenario stuff'          // includes buildings, units and credits at start of game, and team
'regions'                 // rectangular regions used by FSM
'special forces groups'   // groups of sf soldiers that are commanded by FSM
```

The following are actions that can occur in the scenario files (and one can occur in the FSMs, too).

SetAlliance
SetAlliance (*number* t0 *number* t1 *number* t2 *number* t3 *number* t4 *number* t5 *number* t6 *number* t7)
This must appear in a SetTeam construct. There is also an 'action' version of this command that can be used as an action in an FSM. It specifies how this team sees the other seven teams (as an ally, neutral or enemy). The parameter list specifies a number for each team (including this team), which represents this teams view of the particular team. Note that the other team may have a different view of this team. The alliance state in the parameter list for this team should always be allied (it does not make sense for a team to be an enemy of itself.) Note that if this statement is not specified for a team, then it is assumed that this team thinks of all other teams as enemies t0..t7 are 0 – enemy, 1 – neutral, 2 – ally

Example:

```
SetAlliance(2 0 0 0 1 2 0 0)
```

This says that the team who calls this considers teams 0 and 5 to be allies, team 4 to be neutral, and all the rest it sees as enemies. As you can see, when the definition of an action or criteria specifies "*number* t1", the "number" just says what sort of parameter to use, and the "t1" is just a convenient name so that you can remember what is supposed to go where. When you use the action, just put in a number.

SetEnd
SetEnd (*name* endcondition_filename)
This must appear in a SetTeam construct. This specifies the endcondition file to use for the particular team. An endcondition file specifies the actions a team must perform to win the game. If this is not specified, then the teams endcondition defaults to allied victory. It applies to the team which issues the command in the set team command.

Example:

```
SetTeam(0)              ; team number 0
{
SetTeamSide(1)          ; is imperium (0=FreedomGuard, 1=Imperium, 2=civilian ?)
SetCredit(20000)        ; and starts with this many credits
SetEnd (defend.end)     ; and reads the 'defend.end' file to get its win conditions
}
```

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

SetFSM
SetFSM (*name* FSM_1 *name* FSM_2 *name* FSM_3)
This must appear in a SetTeam construct. This specifies three FSM files which can be used by this team - in order - easy, medium and hard. It applies to the team which issues the command in the set team command.

Example:

```
SetTeam(0)                                  ; team number 0
{
SetTeamSide(1)                ; is imperium (0=FreedomGuard, 1=Imperium, 2=civilian ?)
SetCredit(20000)                            ; and starts with this many credits
SetFSM (easy.FSM medium.FSM hard.FSM )      ; and has this choice of FSM's
}
```

If a team does not set an FSM, it will attempt to load a default FSM. The default FSM name is constructed as follows - 'def_tt_v.FSM' – where tt is a two digit team type and v is the FSM variation number (difficulty). First the scenario directory is searched, then the \AIP directory (where global default FSMs are to be stored.) For example – Def_02_1.FSM is a civilian – medium difficulty FSM. eg. Default0.FSM is the name for a default FSM for Imperium teams and default1.FSM is the name of default FSM's for freedom guard teams. Note that there must be a default FSM for every team type (including Civilian), that is used in a scenario. To be on the safe side, specify a default FSM for team types 0..4 and have a single state which does nothing in those types that you don't wish to develop further.

DefineRegion
DefineRegion (*number* region_id r *number* x1 *number* y1 *number* x2 *number* y2)
DefineRegion (*number* region_id c *number* x *number* y *number* radius)
This defines a rectangular or circular region of ground. The units of this command are pixels, 0 0 is the top left-hand corner of the map. Each map tile is 24 pixels wide and high, so to convert a tile x or y position to a pixel x or y position, multiply it by 24. Regions are used to trigger special forces, specify areas of the map to harass, hold etc.

Example:
DefineRegion(1001 r 0 0 24 72)
This defines a rectangular region in the upper-left hand corner of the map which is one tile wide by three tiles tall and assigns it the ID number of 1001.
NOTE THAT CIRCULAR AREAS CANNOT BE USED FOR AI OR SPECIAL FORCES PURPOSES.

DefineSpecialForces
DefineSpecialForces (*number* special_forces_id *number* team)
{
number unit_id ...
}

This construct defines a group of units into a crack squad of special forces. This prevents the AI system from taking control of these units. Instead, their behaviour is defined by the FSM commands TriggerSpecialForces and ReleaseSpecialForces.

The command takes a unique id and a team number. This id is the same one to be used in the other special forces commands that act on this group of special forces. Note also that the id must be unique in the file - no building, overlay (tree etc.), unit or anything may have this id.

Example:
DefineSpecialForces(2001 44 46 47 51)

In this example, a group of special forces containing units 44, 46, 47, and 51 is created and assigned the group ID 2001.
See also: TriggerSpecialForces, ReleaseSpecialForces.

In .FSM and .end files
This section contans everything that can be put in and FSM or endcondition file.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

DefineEndCondTree
DefineEndCondTree (*number* time_limit)
{
DefineCondState ...
}

This must be the first keyword in an '.end' file. The End Condition files must all reside in the scenario directory. The team that this tree applies to is the team that loaded this end condition tree from the scenario file.

There are two FSM's which can be specified for each team, the End Condition Tree and the AI Condition Tree. There can be multiple conditional states in each tree.

Define an end game (win) Condition Tree. If an End Condition is not specified for a team, it defaults to kill all units and buildings of non allied teams.

The conditional states are numbered in the order they are listed. The first one is state 1, the second one is state 2 and so on. Note that conditional state 0 is the win state, a transition to this state causes the team to win (and therefore all other teams not winning at this time to lose).

There is no lose condition. To lose the game, another team, or teams, must win the game. This means that if you have had all units and buildings destroyed, and another team donates you a construction crew or fleet of tanks, you can continue in the game.

Team is the team number that this end Conditional Tree applies to. Only one End Conditional tree may exist per team.

The time_limit is the total time (in game cycles) allotted to complete this Conditional Tree. The bonus times specified in some conditions / actions can add to this allotted time. If time limit is 0, it has no effect. If this time limit runs out, then this team cannot EVER win the game)
See also: DefineCondState, DefineAICondTree

DefineAICondTree
DefineAICondTree ()
{
DefineCondState ...
}

There are two FSM's which can be specified for each team, the end condition tree and the AI Condition Tree. There can be multiple conditional states in each tree. The conditional states are numbered in the order they are listed. The first one is state 1, the second one is state 2 and so on.

This must be the first keyword in an FSM file. An FSM file is the only place this keyword can appear. The team this FSM applies to is the team which loaded the FSM using the SetFSM keyword. An FSM can be used by multiple teams simultaneously. (Separate instances of the same file are loaded.)

Define an AI behaviour tree.

See also: DefineCondState, DefineEndCondTree

DefineCondState
DefineCondState ( )
{
ReleaseSpecialForces ...
TriggerSpecialForces ...
GiveSpecialForces ...
AdjustRegionPri ...
SetaiPFile
SetMessageFile

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

```
DefineCondition ...
TriggerEvent ...
SetAlliance
TriggerMessage...
}
```

This defines a conditional state. A Conditional Tree may have many conditional states. The conditional states are numbered in the order they are listed. The first one is state 1, the second one is state 2 and so on. Note that in the End Condition Tree, conditional state 0 is the win state, a transition to this state causes that team to win.

The states are linked to each other, in that DefineCondition specifies the condition that must be met in order to go to the next state. There may be many conditions in each condition state each connecting to a different state.

Note that if the DefineCondition command is not specified, then the state will never exit. This will prevent this end / AI Condition Tree from ever winning the game, or executing any more statements. This is useful only when there is nothing more to achieve in this tree, and this tree doesn't want to cause the game to be won.

DefineCondition

```
DefineCondition
(
number next_state number time_limit number bonus_time
number score name debug_message
)
{
criteria single_criteria
}
```

Define a transition condition - this specifies what requirements need to be met to get to another state.

- next_state is the state to jump to when the criteria is satisfied.
- time_limit is the time the team has to complete this condition.
- bonus_time adds to the total time the team has to complete an end conditional tree (to win the game) This ensures that if a complicated state is reached, more time can be given to the player to complete it. For an AI conditional tree, this must always be 0.
- score is currently not implemented, but if it were it would be the number to add to the teams score on completion of the state.
- Debug_message is a message that is printed on the game screen when this state transition occurs. It also prints the team number, current state, next state and if AI or end condition. NOTE THERE CANNOT BE ANY SPACES IN THIS MESSAGE, and it MUST start and end with a quote. For example: "Yes_Killed_Them"

There can only be one criteria listed, however it may be the AND or OR ciriteria, which can have two criteria as its input. This structure may recurse any number of levels deep.

The DefineCondition construct can be specified more than once in a conditional state if there are two different states that can be reached when different conditions are met. For example, state 2 may be the 'AI team has approx the same number of units as the enemy' state and may jump to state 1 when the condition 'team has less than 50 % of units of enemy' is met, or jump to state 3 when the condition 'team has more than 150 % of units of enemy' is met.

Actions Within a State

These are actions that are carried out when the conditional state is entered. If the state is entered more than once, then the action will be issued every time the state is entered.

TriggerSpecialForces

TriggerSpecialForces (*number* special_forces_id *number* region_id)

Triggers a special forces group to go to the region specified. The region must be specified as a rectangular region, not a circular one. The units in the special forces group will be distributed randomly in the rectangular area. If all units are wanted to be in a single tile, then the area should be specified as one tile wide in the define region construct. Note however that only one unit can ever occupy a tile at a time.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

This command will override any previous orders issued to this group of special forces - if two states were changed quickly, and the first one ordered the special forces to a particular region, and the next state orders the group to another, it is conceivable that the special forces will never arrive at the first specified region.

ReleaseSpecialForces
ReleaseSpecialForces (*number* special_forces_id)
Control of the special forces is handed from the FSM to the AI system. When the special forces have finished their assignment, they can be handed to the AI system to use, instead of continuing to guard or harass the same region.

GiveSpecialForces
GiveSpecialForces (*number* special_forces_id *number* team)
Give units to the team specified team. The units must pre-exist on the map. If any of the units are destroyed before the give units command is executed, those destroyed units will not be given. The units do not have to be from the team that is executing the give command, they can be from any team.

AdjustRegionPri
AdjustRegionPri (*number* region_id *number* priority *number* min_forces *number* max_forces)
This adjusts the scripted value in the AI scheduler of all grid cells within a particular region to be the new priority value. Additionally, it must specify the minimum and maximum unit strength points needed to satisfy the goal. This currently only works with rectangular regions, circular regions cannot be used.

BonusCredits
BonuesCredits (*number* team_id *number* free_money)
This can be used to give the specified team extra money for free. It could be a reward for achieving a goal, or whatever you want. The credits given do not come from the team that possesses the FSM; they come from nowhere.

SetAIPFile
SetaiPFile (*name* filename)
Define an AIP file to switch into, for this state. The AIP file currently must be in the scenario directory, however this will probably change to first search the scenario directory and then search the \AIP directory. AIP's are documented separately.

SetMessageFile
SetMessageFile (*name* filename)
Define a message file to display on the screen of this player upon entering this state. It currently displays just the filename.

This command is only valid in End Conditional Trees, it is meaningless in an AI Conditional Tree.

TriggerMessage
TriggerMessage (*name* multi_language_key)
Play a sound (.wav file) when a state is entered. The messages will be queued and played in the order they are listed in the condition state. This can provide a taunt, or some training information. The multi language key is used to index the multi language support text configuration file. This in turn plays the correct wave file.

SetAlliance
SetAlliance (number t0 number t1 number t2 number t3 number t4 number t5 number t6 number t7)
There can also appear in a SetTeam construct in the scenario file (this is where the default alliances should be set). It specifies how this team sees the other 7 teams (as an ally, neutral or enemy). The parameter list specifies a number for each team (including this team), which represents this team's view of the particular team. Note that the other team may have a different view of this team. The alliance state in the parameter list for this team should always be allied (it does not make sense for a team to be an enemy of itself.) This set of alliances can only apply to the team that issued this command from its FSM / end condition list. (It can never affect a different team.) t0..t7 are 0 – enemy, 1 – neutral, 2 – ally.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Criteria

This section describes the various criteria (conditions) which can be used to trigger changes between states.

CritOR

CritOR()
{
criteria criteria_1
criteria criteria_2
}

The criteria is complete if either criteria_1 or criteria_2 is complete.

CritAND

CritAND()
{
criteria criteria_1
criteria criteria_2
}
The criteria is not complete until both criteria_1 and criteria_2 are complete.

CritNOT

CritNOT()
{
criteria criteria
}
This criterion is completed if the nested criteria is incomplete or cannot be completed. If the nested criteria are complete, this criterion will be marked incomplete.

CritMoreUnitsThanEnemy

CritMoreUnitsThanEnemy (*number* percentage)
The number of enemy units is defined as being the number of units of the enemy team with the largest number of units, NOT the total number of enemy units. This is always true if there are no enemy teams. Neutral and allied teams are not considered in this calculation. This is true when we have ((team_strength * 100) > (max_enemy_strength * percentage))

CritLessUnitsThanEnemy

CritLessUnitsThanEnemy (*number* percentage)
The number of enemy units is defined as being the number of units of the enemy team with the largest number of units, NOT the total number of enemy units. This is always false (incomplete) if there are no enemy teams. Neutral and allied teams are not considered in this calculation. This is true when we have ((team_strength * 100) < (max_enemy_strength * percentage))

CritTimer

CritTimer (*number* game_cycles)
This criterion is complete when the number of game cycles has passed whilst in this condition state. If the state is exited and re-entered, then the count begins again from 0.

CritTimerGame

CritTimerGame (*number* game_cycles)
This criterion is complete when the number of game cycles has passed since the beginning of the game. Note that once this criterion is complete, it can never become incomplete, as the game cycle number will have always passed.

CritStealPlan

CritStealPlan (*number* team *name* item_name)
Completed when the team 'team' steals the plan for the unit or building 'item_name'. To steal the plan, they must send a spy to the facility, spy on it, and walk back into their own headquarters. The plan is not considered stolen until that time. This means that 'team' must be able to build the stolen item before this condition is true. (They do not actually have to build the item however.) Note that the team 'team' can steal the plan from any team they choose, not necessarily the one executing this criterion.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

CritHoldRegion
CritHoldRegion (*number* region_id *number* limiter *number* flags)
The criterion is complete if a region is held by the team that is testing this criterion for a certain amount of time. A region is held during a time period if at the end of the time period there are no enemy units within it and there are allied units present within it. Note that neutral units have no effect on the calculation. Note enemy buildings ARE allowed to remain in the region.

'limiter' is the number of time periods that the team is required to hold the region for. Currently, one time period is 32 game cycles. (=> 10 is 320 game cycles. There are usually between 15 and 40 game cycles per second, depending upon the game speed selection and the machine speed.)
Note that the region does not have to be held continuously, just so long as it is held for 'limiter' number of time periods.

'Flags' is not used and should be 0 for future compatibility.

CritHarassRegion
CritHarassRegion (*number* region_id *number* limiter *number* flags)
The criteria is complete if this team does damage to a region for a certain amount of time.
Limiter is either the total hitpoints damage to be done to the region or the number of time periods that harassment must be performed over. The damage / time limit is reset whenever harassment ceases, unless the CF_ACCUMULATIVE flag is set.

Flags = 0 – the limiter is a number of time periods which the region must be harassed for. See CritHoldRegion for a definition of how this time period is calculated. The harassment must be continuous. In other words, there may be no time period where damage was not done within the region, otherwise the time counter or damage counter is reset to the original value of limiter.

Flags = 2 (CF_DAMAGELIMITER) – the limiter is an amount of damage done to a region, not an amount of time. The damage done is increased by the hitpoint value of a projectile when it explodes in a region (on a building or unit within the region, or on the ground).

Flags = 1 (CF_ACCUMULATIVE) – the harassment need not be continuous. In other words, there may be time periods where damage is not done within the region. Damage is accumulated.

Flags = 3 (CF_DAMAGELIMITER | CF_ACCUMULATIVE) – continuous harassment is not required, the limiter value supplied is an amount of damage required, not a time value.

CritBuildBuilding
CritBuildBuilding (*name* building_type *number* region_id *number* amount)
The criteria is complete when the team has completed construction of the required number of buildings of type 'building_type' within the region specified by region_id. Amount is the number of buildings of the type that have to exist within the region at this point in time. It is not the number of buildings this team has commenced building. (Buildings may have been destroyed.)

Note that only rectangular regions are supported, circular regions are not supported.

If the region_id is 0, then the region is considered to be the whole map.

Building_type is the symbol id of the building from the build.txt file (the same one as used in the CritStealPlan criteria).

CritBeginBuildBuilding
CritBeginBuildBuilding (*name* building_type *number* region_id *number* amount)
The criteria is complete when the team has commenced or completed construction of the required number of buildings of type 'building_type' within the region specified by region_id. Amount is the number of buildings of the type that have to exist within the region at this point in time. It is not the number of buildings this team has commenced building. (Buildings may have been destroyed.)

Note that only rectangular regions are supported, circular regions are not supported.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

If the region_id is 0, then the region is considered to be the whole map.

Building_type is the symbol id of the building from the build.txt file (the same one as used in the CritStealPlan criteria).

CritBuildUnit
CritBuildUnit (*name* unit_type *number* amount)
The criteria is complete when the team builds a unit of type 'unit_type'. Amount is the number of units of the type that have to be built. Unit_type is the type symbol id of the unit from the unit.txt file.

CritMoveUnitsToRegion
CritMoveUnitsToRegion (*number* region_id *number* survivors)
{
number unit_id ...
}

This condition becomes true when at least 'survivors' number of units have been into the region specified. They do not all have to be in the region at the same time. If enough units die such that there can never be 'survivors' number of them in the region, then the condition can never be completed.

In effect, this criteria means 'At least survivors number of units from the list must have at some point in the game been in the region'. The units may be from any team, or a combination of teams. They do not necessarily have to be on this team.

Currently the units must be in the region and visible. They cannot be inside a transport, or inside a building.

CritDestroyBuilding
CritDestroyBuilding ()
{
number building_id ...
}

This criterion is completed when all buildings in the list of buildings are no longer present. This usually means the building is destroyed, but it could also mean the building is sold or otherwise removed. Note that these buildings could be from any team (including the team that is checking the condition). All buildings in this list must exist at the start of the game, when the map is loaded (i.e. created with the map editor). There is no provision here to check for the destruction of buildings that are built after the game begins.

CritDestroyThing
CritDestroyThing ()
{
number thing_id ...
}

Criteria completed when marked all 'things' in the list are destroyed. It is not defined what a thing is. This criteria is currently not implemented, and always returns incomplete / false. It is intended to detect destruction of an overlay (tree / rock etc.).

CritDestroyUnit
CritDestroyUnit ()
{
number unit_id ...
}

This is criteria is completed when all units in the list of units are destroyed. Note that these units could be from any team (including the team that is checking the condition). All units in this list must exist at the start of the game, when the map is loaded (i.e. created with the map editor). There is no provision here to check for the destruction of units that are built after the game begins.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

CritCollectMineral
CritCollectMineral (*number* amount)
Completed when an amount of minerals are mined by the team since the beginning of the game. The units are the same as given in the statistics screen at the end of the game. Note that this is the amount actually mined by buildings from this team, not the amount used in this team's power stations.

CritCollectWater
CritCollectWater (*number* amount)
Completed when an amount of water is mined by the team since the beginning of the game. The units are the same as given in the statistics screen at the end of the game. Note that this is the amount actually mined by buildings from this team, not the amount sold by this team's water launcher.

CritKillEnemyUnits
CritKillEnemyUnits (*number* percentage)
This criteria is completed when the total number of units all enemy teams have becomes equal to or falls below (starting_number_of_enemy_units * percentage / 100). For example, if the enemy teams started with 30 units, and the criteria was CritKillEnemyUnits (20) then the criteria will become true when the enemy team has six or fewer units left. Even if the enemy teams have the original starting units destroyed this criteria will remain false if enough new units were produced to keep the unit count above the required threshold. Note that enemy teams are teams that this team is listed as an enemy of on the comms menu. The units of allied and neutral teams are not counted in this calculation at all.

CritKillTeamUnits
CritKillTeamUnits (*number* team *number* percentage)
This criteria is completed when the number of units the specified team has becomes equal to or falls below (starting_number_of_teams_units * percentage / 100). For example, if team 2 started with 30 units, and the criteria was CritKillTeamUnits (2 20) then the criteria will become true when team 2 has six or fewer units left. Even if the specified team has all of its original starting units destroyed, this criterion will remain false if enough new units were produced to keep the unit count above the required threshold.

CritDestroyEnemyBuildings
CritDestroyEnemyBuildings (*number* percentage)
Completed when a percentage of enemy buildings are destroyed. This relates to the number of buildings that the (current) enemy teams had when the scenario was loaded, even if they were not enemies at that time. The percentage is the percentage to destroy, not the percentage allowed to remain. Note that new buildings enemy teams build count as if to replace a destroyed building.

CritDestroyTeamBuildings
CritDestroyTeamBuildings (*number* percentage)
Completed when a set percentage of a team's buildings are destroyed. This criteria is completed when the number of buildings the specified team has becomes equal to or falls below (starting_number_of_teams_buildings * percentage / 100). For example, if the team 2 started with eight buildings, and the criteria was CritDestroyTeamBuildings (2 25) then the criteria will become true when team 2 has two or fewer buildings left. Even if the specified team has all of its original starting buildings destroyed, this criterion will remain false if enough new buildings were produced to keep the number of buildings above the required threshold.

CritKillAll
CritKillAll ()
Completed when ALL NON-ALLY units/buildings are destroyed. This does not include resource-producing facilities. Combat units do not automatically target enemy water wells and mineral mines, as any team can mine resources from them.

CritKillAllAndAllies
CritKillAllAndAllies ()
Completed when ALL units/buildings are destroyed, even those belonging to your allies. This does not include resource-producing facilities.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

CritDestroyBuildingType
CritDestroyBuildingType (*name* building_type *number* team)
Completed when a team has no buildings of a particular type remaining on the specified team. The building type is the symbol id of the building (from the build.txt file).

CritKillUnitType
CritKillUnitType (*name* unit_type *number* team)
Completed when a team has no units of a particular type remaining on the specified team. The building type is the symbol id of the unit (from the units.txt file).

CritEnemyInRegion
CritEnemyInRegion (*number* region_id)
This criterion is true if any enemy unit is currently in the region. It is false even if enemies were in the region, and no longer are.

CritInRegion
CritInRegion (*number* region_id)
This criterion is true if any units from this team are currently in the region. It is false even if enemies were in the region, and no longer are.

CritTeamInRegion
CritTeamInRegion (*number* region_id *number* team_id)
This criterion is true if any units from the specified team are currently in the region. It is false even if enemies were in the region, and no longer are.

CritHaveCredits
CritHaveCredits (*number* credits)
This criterion is true when the team's credits get to be greater or equal to the specified number of credits.