Designing Strategic AI

WarBirds allows some very interesting complexity to be built into the game via the new AI files. While initially designed only to provide some training content, with some ingenuity, some very complex strategic model concepts can be introduced into the arenas. Designers may use the following models as modules for arena strat as per need and scenario.

- **AI files** are the files generated by the Mission Editor within the WarBirds application itself.
- **DTF files** are text files generated by any text editor and contain dot commands for binding together AI files, change settings in an arena, sending radio traffic messages, etc, or containing script elements for performing conditional testsof items in an arena: field control, AI formation status, base object status, etc. DTF file can call other DTF files via the .script <path> command.
- **SCRIPT files** are DTF files which specifically contain script conditional elements.

AI (MBL) Files

AI files define the actual units and paths those units will take within an arena. They can be defined by hand, but it is vastly easier to define them using the built-in Mission Editor, included with the WarBirds application.

AI files contain 3 sections:

- **Basic Information about the AI -** Primary vehicle type (see table at end of article), controlling country, base of origin, number of vehicles, start and restart times (see below), and flags which affect the AI.
- **Waypoint Definitions** - The location of each waypoint, type, formation used, spacing between units, waypoint specific data (e.g. wait time and target for strike waypoints), and any DTF files called.
- **Vehicle Definitions** - Vehicle number (i.e. type of vehicle), Skill settings and Attack range for each vehicle being used in the AI file.

STARTTIME - this defines the number of seconds the AI will wait to become active after being initially loaded. For example: A start time of 300 seconds will cause the AI to wait for 5 minutes before appearing and starting to run through its designed track.

RESTARTTIME - this defines the number of seconds the AI will wait after being destroyed or after it finishes its defined track before it will again resume activity starting with its first waypoint. For example: a restart time of 900 seconds will cause an AI to wait for 15 minutes before it reappears and starts running again.

Designers may place comments throughout the AI file by placing a # at the beginning of a comment line. The entire line after the # will be ignored by the AI. A good use for this is to comment the beginning of each AI file for what it's supposed to do (including references to actions taken by DTF files), so that the purpose of the AI file may be easily determined. Another useful location for comments is immediately before a major waypoint definition line (i.e. one where a base would be captured, another formation launched, etc) to explain the purpose of that waypoint.

Waypoints

Each AI file contains 'waypoints' which can be thought of as segments of activity within the AI's defined track. Each segment exists from just after the immediately previous waypoint or the beginning of the AI track if the first waypoint to the defined waypoint itself. Each segment contains a START position (at the beginning of the segment), END (at the end of the segment - i.e. the actual location of the waypoint), and DEAD (if all the elements in the AI are killed within the segment). These activity points within the waypoint segment may be accessed by clicking on the 'Dot Files' button in the Mission Editor. They may also be accessed by editing the AI mobile (MBL) file with any text editor that will save in a simple text format. (i.e. without any format settings)

The 3 waypoint segment locations are called:

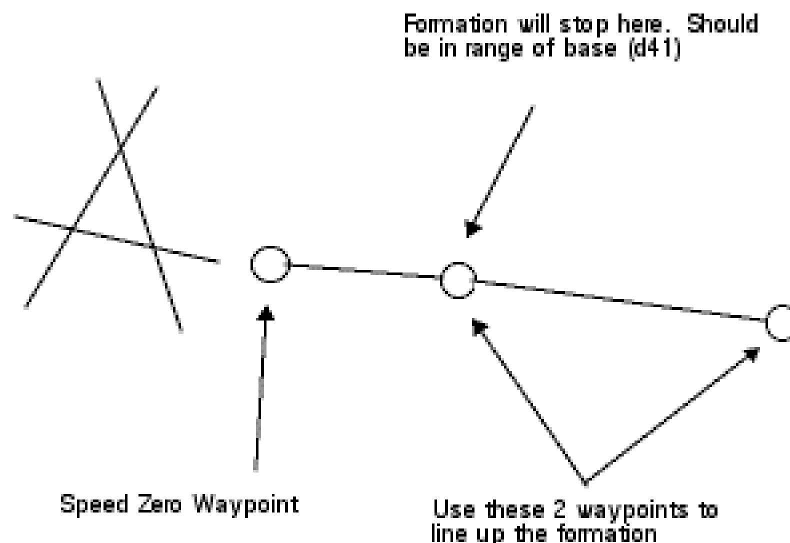
- **DOTFILE_START <path>** - the DTF file which will be run at the beginning of the segment
- **DOTFILE_END <path>** - the DTF file which will be run at the end of the segment
- **DOTFILE_DEAD <path>** - the DTF file which is run if every element of the AI file (i.e. all GVs or AC are destroyed within the segment)

There are basically 3 types of waypoints (how they appear in the AI file itself are in parentheses):

- **Nav (CONTINUE)** - used to simply move from one point to another
- **Fld Ops (TAXI, TKOFF, LAND)** - used to make an AI Taxi around an airfield, takeoff, or land. When landing, all planes will attempt to land in alignment with runway 1.
- **Halt (PAUSE)** - used to cause Ground AIs to stop and wait for a given number of seconds before continuing with the next waypoint segment. It is also necessary to define the number of seconds the AI formaition will **WAIT** at the waypoint before proceding. This definition is entered in the textbox labeled wait on the ME screen.
- **Strike (STRIKE)** - used typically by bombers to attack specific targets at a given base. If Ju52s or M5s are given a strike order, they will launch their troops at the tower/church of the base in question. This has no effect (other than destroying the tower or church offline), but is interesting otherwise. Sometimes it can be used to simulate an assault infantry attack on a base with the actual capture resulting from the AI formation reaching a waypoint in the base when a DTF is run affecting the capture.
- **CAP** - used by fighter AIs. When using this type of waypoint, fighters will patrol briefly near the waypoint, attacking any and all AC that come within targeting range.

To this we will add the following 'Artificial' waypoint types:

- **Stop** - defined as 3 waypoints. A waypoint where the AI formation (ground or naval) will stop and wait indefinitely. The waypoint immediately proceeding the 'stop' point which determines the line up of the formation by the line between it and the stop point. And a 'last' waypoint which has defined for it a speed of zero across the waypoint. AIs will stop at the stop point, lined up based on the line up point, and move no further because the actual last point has no speed, and the AI can never reach it.



- **Random** - This is the last waypoint in an AI, but instead of calling a DTF to unload the file or call another DTF, a DTF file is loaded which contains a random switch statement in it, allowing for the current AI to end, and another randomly chosen AI to pick up where the current leaves off.

Each waypoint has the following items that are adjustable:

- **Turn Distance -** defines how far an AI unit must be from the waypoint before concluding it has reached it. A common mistake is to try making this radius very small. If the AIs cannot reach this radius (perhaps because they were trying to get in formation and missed it, then can't turn tightly enough to get inside the limit), they will turn endless circles around the waypoint. Each vehicle will attempt to reach this limit, so it is possible that some AC/GV may make the limit and move on and others will endlessly circle the waypoint, trying to get in close enough to reach it. It is recommended that land based waypoints have a Turn Distance of 50-100 feet, air operations use waypoints set for 500-2000 feet (fighters are shorter, bombers are usually longer), and 500-1000 feet for Naval operations.
- **Wait Time** - defines how many seconds the AI formation should wait before continuing if the waypoint is a **Halt** type waypoint. If not, entering a value here has no effect on the AI behavior.
- **Speed** - the speed of the lead unit of a group of the waypoint segment. this is the speed the whole group will try to maintain. If you set it to the maximum possible for the vehicle type in your AIs, some of the AI units may have a difficult time maintaining proper formation, especially if changing the formation.

(continued next page)

- **Frag** - (button) click this button to determine what the AIs will consider appropriate targets over the current waypoint segment.
    1. **Aircraft / Vehicles** Formation will attack all enemy AC or GVs it encounters. How and whether is attacks is dependent on the type of AC/GV being used. e.g. Divebombers will not attack other AC or bombers, but will attack any GVs if encountered.
    2. **Fixed Ack** Formation will attempt to attack and destroy any AA (anti-aircraft or ack) guns encountered at bases.
    3. **Ground Structures** Formation will attempt to attack any building in addition to acks at bases.
    4. **Strike Fixed Asset** Formation will attempt to attack a specific structure (use pull down menu to define). The string designation of the structure is also used by DTF scripts when testing for control or destruction, so it's useful to note these. When selecting this, the developer will be presented with a pulldown menu for selecting the target to strike, and a textbox for entering the CEP value. This is simply the distance before the target the bomber will initially aim for when dropping dropping bombs. It is most useful when designing bomb runs that will maximize damage through salvos.
- **Formation** - (button) click this button to determine what formation the AIs will use over the current waypoint segment.
    1. **Trail (TRAIL)** Each vehicle or plane follows the one in front of it and attempts to maintain a straight line throughout a turn. (i.e. the end vehicle will attempt to move the fastest to keep a lineup with the first vehicle)
    2. **Vic (VIC)** All AC/GV will line up in a V formation up to 8 before it appears they are overlapping each other. AC/GV will be added to the left of the leader first, then the right.
    3. **Wedge (WEDGE)** Essentially a Vic formation that is spread out a bit more
    4. **Echelon_right (ECHELON_RIGHT)** All AC/GV will line back and to the right of the lead unit.
    5. **Echelon_left (ECHELON_LEFT)** All AC/GV will line back and to the left of the lead unit.
    6. **Diamond (DIAMOND)** AC/GV will form into a diamond shaped 'box' formation.
    7. **Ladder (LADDER)** Acts similarly to **Trail** except AC will attempt to stack up behind the lead. GVs will treat this formation like **Trail**.
    8. **Fighting Wing (FIGHINGWING)** - in AC, this formation will produce a 'Finger Four' type formation. AC/GV will break up into pairs of vehicles, fighting together.
    9. **Inverse Ladder (INVLADDER)** This formation works just like a **Ladder** except the AC stack each lower than the previous unit. GVs treat this as **Trail**.
    10. **Line Abreast (LINEABREAST)** Usually a naval formation, all the units will travel in a row. In GVs this is a useful formation for concentrating firepower on a base being attacked.
    11. **Road (ROAD)** Each AC/GV will exactly follow the unit in front of it, attempting to turn in the exact same places. This is a good formation for traversing narrow defiles with ground vehicles.
- **Dot Files** - (button) click this button to bring up a dialog box where you will have to type in the path to the DTF file yourself. The path to the file is relative to the WarBirds application.

AI Unit Attributes

Both offline AI and the online AI Generals are restricted in the number of both AI files and the number of active AIs running. No more than 64 AI files (MBL) can be loaded into either offline play or the online AI General. This is an absolute limit. If an attempt is made to load a file which would exceed 64, the load request is ignored. Similarly, only 128 active AI units (i.e. all GV and AC combined) may be loaded. While more than 128 units may be loaded, only 128 of them will be active, and it's not a good idea in the first place. Better is to unload what is not needed first then load the new files. Also, unload a file (with a suitable DTF) when not needed, in order to make space for other files.

Each AI unit itself has 6 attributes:

1. **Vehicle Number** - defined as Model 0 if using the major type defined for the entire AI file, or using the vehicle number (see table at end of article) for any if desired.
2. **Loadout** - this can be set in the 'Detail' dialog for each AI unit after the AI file has been initially saved, but each number corresponds to the loadout in the player's pulldown menu. (i.e. the first location in the menu is loadout 0, the 2nd is loadout 1, etc.)
3. **Tactical (skill1)** - defined as the tactical intelligence of an AI. This will define how well the AI moves after coming into contact with enemy units. (e.g. does the AI try to keep the range open an use a better gun, or does it charge blindly in at an enemy gun line?) It also defines how well the AI will track its next waypoint. If the number is very low, the AI can sometimes end up driving in circles.
4. **Weapons (skill2)** - defined as how well the AI shoots (think otto accuracy). High values are more devestating to enemies, since a higher percentage of shots will hit. Lower values are used to simulate lesser quality troops, training, or under calibre armament.
5. **Aquisition Range** - defines the range (in feet) that the AI will start tracking enemy units (both AI and player controlled). If those tracked units come within range of the AI's weapon and the aquisition range, it will begin firing. Adjusting this up to a range just inside the maximum range of the AI's weapon type (but is still likely well outside the Icon and Clutter Distance range for the players) gives the AIs a way of simulating forward observers when using M3s as infantry artillery.

DTF (Dot) Files

Dot files provide the mechanism for designing ‘behavior’ into the AIs. As such they make heavy use of the various dot commands used (both online and offline). If building AIs for use offline, each player has full control over the offline environment. For online consumption, it’s better simply to place comment lines (proceeded by a #) to allow a CM or other person with arena authority to place the appropriate dot commands therein. Dot files can also contain Script elements in them, allowing for test conditions to modify the behavior of AIs based on current conditions rather than a ‘canned’ response. Parameters which are set in <> are required; those in [] are optional.

Dot Commands (Common)

Command	**Action** (offline or online)
.offcmtool	(offline) Used to tell the WB application AI that the following commands will be used in an offline scenario. (this command need only be used by the .off file which initially loads the AI)
.loadmo <path> .ldmo <path>	(offline / online) Used to load an AI file. The path element is the location of the file relative to the WB application folder.
.unloadmo <path>	(offline / online) Unloads an AI file. It will stop executing immediately and all AI elements in the file will disappear.
.restore_field <base>	(offline / online) Rebuilds a base to a fully repaired (i.e. no structures damaged) status.
.restore_fields	(offline / online) Rebuilds all fields
.resall	(offline / online) Resets all settings of an arena or offline environment
.setfield <base> <country>	(offline / online) Used to set control of a base to a given country
.clearmo	(offline) Clears all player and drone waypoints
.terrain_load <terrain name>	(offline) Loads the appropriate terrain map. Use the terrain name from its file minus the .vfc suffix (e.g. small ETO terrain is called terr001)
.date <month> <day> <year>	(offline / online) Sets date
.offtime <24hour> <minute>	(offline) Sets time
.time <24 hour> <minute> <speed>	(online) Sets time online. Speed setting is a divider for defining how fast time will progress. e.g. a speed multiplier of 2 will result in a 24 hour WB ‘day’ lasting 12 real hours.
.clouds <0,1>	(offline / online) Sets whether clouds are 0 - Off, or 1 - On
.weather <num>	(online) Sets the prevailing weather in the entire arena. 0 - Clouds, 1 - Clear. These are the only types of weather currently defined
.offarenaicons ENEMY_PLANES <range> <ID enemy> <ID friendly name>	(offline) Sets visibility of icons offline. I like offarenaicons ENEMY_PLANES 1 2401 6001 The range numbers are in feet.
.country <1, 2, 3, 4>	(offline) defines which country the player will start in

Dot Commands (Continued)

.field <base>
: (offline) defines which base the player will start at

.plane <number>
: (offline) defines which plane (1 - 79) the player will start with

.selectableon
: (offline / online) Turns on the ability to define availability of GV/AC at a base

.enable <0-disable, 1-enable> <planeNo> <countryNo + baseNo>
: Enables a GV or AC at a base (See Table below)
If a baseNo of 00 is used, all bases of the defined country will be affected.
A countryNo + baseNo of -1 will affect ALL bases in the arena.
A planeNo of -1 will cause all plane types to be affected in the same way at the given base.
Examples:

.enable 1 16 314	(enables the Bf110G2 at Gold base 14)
.enable 1 79 -1	(enable the truck at all bases)
.enable 0 31 400	(disables P51D at all Purple bases)

.radio <button set> <channel>
: Used to set a radio channel. Right now, the AIs can only communicate over button set 1, so all AI communication should use the following procedure:

```
.radio 1 103 (talk over Gold channel)
.echo This is what I'm going to say
```

.echo Text to Broadcast
: Broadcasts the following text over whatever channel set 1 is currently set to. There is no need for quotes containing the text

.prompt "1st line" "2nd line" "3rd line", etc <seconds on screen> <red value> <green value> <blue value>
: (offline) Places multiple lines of text in the center of the screen, using the included RGB values. Text must be in quotes.

.script <path>
: (offline / online) Runs a script file. File must be a pure script (not have %%) (cannot be inside imbedded script or any script command)

.dotfile <path>
: (offline / online) Runs a dotfile. This file may be a normal file or imbedded script (cannot be inside imbedded script or any script command)

.return [path]
: (offline / online) Hands off control of script to the listed script in the path. This functions as if the named script were actually a part of the existing script. One of the best uses for this is for nested conditionals. i.e. a series of test statements inside a test statement. (WB Scripting does not explicitly allow this. Using .return is the only way to manage it)

.dtf_timed <path> <repeat> [reps]
: (offline / online) Loads a script at **<path>** and runs it every **<repeat>** seconds for a number of times equal to **<reps>**. If **rep** is missing, then 1 time is assumed. If **reps** is -1, then the script will run until unloaded.

.dtf_timedclear [path]
: (offline / online) Clears a timed script from the list. If **path** is missing,then all scripts will be unloaded.

.dtf_timedlist
: (offline / online) Produces a human readable list of the timed scripts currently loaded.

.listmo
: (offline / online) Produces a human readable list of the MBL files currently loaded and their status (**RUNNING** or **OFFLINE** - i.e. not running)

Adding Test Script to WarBirds DTF Files

WarBirds has the ability to write conditional scripts which will allow for a greater degree of testing, flexibility, and unpredictability in the behaviors of the AIs. These tests are placed inside DTF files and are called in 2 ways:

- Calling a DTF file with only test code in it by using .script <path>
- Imbedding the test code in another DTF file by using %% before and after the code segment (most common)

This addition adds tremendous flexibility and unpredictability to modeled AI files and strategic models. We can check for base affiliation, status of a specific base structure (e.g. is Factory 1 destroyed?), even select a random future path for an AI (more later).

Syntax: (an example with comments should suffice)

```
# Lets say we want to build a DTF file which will allow us to trigger
# a base Sentry for base 30 for example
#
# We want...
#   1) That base 30 be controlled by country 3
#   2) That a randomly selected starting location
#           for that sentry be loaded
#

# We use %% in the left-most column to denote that test script
# is starting and ending

# Start Test Code
%%
if (COUNTRY(FIELD(30)) = 3)
{
# Since the base is ours, we switch to another file to do the loading
.return  s30loader.dtf
}
else
{
# Lets possibly load a counter attack force
.loadmo  counterAttack30.mbl

# Radio
.radio 1 103
.echo Sending out force to retake Base 30
}
%%
# End Test Code

# Some more dotcommands might go here

# End Example
```

(Loader Example continued next page)

Inside the **s30loader.dtf** file, we find...

```
# So the base is ours, lets get a random starting life
# We've built 4 versions of the start life for this sentry
#  sentry30a1
#  sentry30b1
#  sentry30c1
#  sentry30d1
SWITCH (RAND(1, 4))
{
# Use one case statement for each possible result in the SWITCH
case 1
# load the first random life
.loadmo  sentry30a1.mbl

# Use break at the end of a case segment to end it
break

case 2
# this is the 2nd possibility
.loadmo  sentry30b1.mbl
break


case 3
# this is the 3rd result
.loadmo  sentry30c.mbl
break

case 4
# this is the last one.  Break is not necessary here since it's last
# but I like putting in a break anyway because I might add more cases
.loadmo  sentry30d.mbl
break
}
# ends switch

# End Loader Example
```

It is important to remember that all code in the test segment **MUST** go in the leftmost column of the file on each line (unless part of a function argument). There can be no indenting, or the script will not run correctly (if at all). Similarly, there can be no spaces between a function name i.e. **COUNTRY** and its list of arguments. i.e. **COUNTRY(<arg>)** is correct, **COUNTRY (<arg>)** is not.

Script Commands

There are many commands that are usable within the Warbirds AI scripts. They are all involved with making tests and branching to a specific case for multiple responses to the tests made.

```
if (<test code resulting in TRUE or FALSE>)
{}
else
{}
```

IF blocks are used to make simple tests and allow for execution of different sets of dot commands depending on the results. Only one test may be made within each <test code> statement. **IF** blocks may not be nested. (i.e. one **IF** block cannot be placed inside another) Similarly, **SWITCH** statements (see below) cannot be placed inside of **IF** blocks. (nor **IF** within **SWITCH**) If it is necessary to perform muliple tests inside one section of an **IF** block, route execution of the script to another script by using the .script <path> command or using the various bit commands. (see below)

```
switch (<a number>)
{
case 0
break

case 1
break

# etc.
}
```

Switch blocks are used if one wishes to have a variety of responses for a single test. Switch blocks work by comparing the number passed in the argument to each **case** statement, executing the commands following a match until either the end of the switch statement is found (with a }) or a break is encountered. Similarly to **IF** blocks, we cannot place a **SWITCH** inside another **SWITCH** or **IF** statement. Nevertheless, we have a powerful tool. One possibility is to use a **SWITCH** with the **RAND** statement (see below), allowing us to randomly load a new AI. What this can mean is that we can make an AI formation branch randomly to continue its path. Switch blocks are also used when considering multiple conditionals in a complex test by switching to a numeric result based on the added results of the multiple tests.

One interesting ability in the switch statement is that multiple case statements can be strung together to provide a window of responses for a given condition. Say, for example, we have a condition that results in a calculated number. We want 1, 2, and 3 to respond in one way and 4, 5, 6 to respond in another.

We achieve this with the following code:

```
# Start Example
switch (aNum)
{
case 1
case 2
case 3
# This will be our first response
# more dotcommands here
break

case 4
case 5
case 6
# This will be our 2nd response
# Note that each series will be responded to based on a group
# of cases, not individual cases
break
}
# End Example
```

Scripting Functions

COUNTRY(<object>)
is used to determine the controlling country of a particular base, AI, or even ground object. In short, it's used to get the controlling country for **any** object within an arena.
returns: Country Number of terrain object

FIELD(<base number>)
is used to return the database object that represents the base passed in the argument. This is the only way to access the base for testing purposes.
returns: Terrain Object representing base

GROUNDOBJECT(<object name>)
is used to access the database name for the object passed. The name of the object may be determined by typing .grndlabels in an arena and looking around. This is what is passed (in quotes) to the function. (see supply script example below)
returns: Terrain Object represented by the Object Name

FIELDSFRIENDLY(<baseNo1>,<baseNo2>)
is used to determine if 2 given bases are both controlled by the same side. This is most useful for when a particular event should only occur if 2 bases are both controlled or both not controlled.
returns: FALSE - Bases are not controlled by the same power, TRUE - Both bases are controlled.

MBLLOADED(<path>)
is used to determine if a given MBL file (the physical manifestation of AIs in an arena) is loaded. It does not have to be running, just loaded. The path passed is relative to the WB application folder. It is not possible to access paths outside of the WB folder.
returns: FALSE - Not Loaded, TRUE - Loaded

RUNNING(<path>)
is used to determine if a given MBL file (the physical manifestation of AIs in an arena) is actually running.
returns: FALSE - Not Running, TRUE - Currently Running

DESTROYED(<path>)
is used to determine if a given MBL file's AI units have been destroyed or not or a passed ground object.
returns: FALSE - Not Destroyed, TRUE - Destroyed

SCRLOADED(<path>)
is used to check if a particular script has been loaded into the timed list. This will allow for other timed scripts or DTF scripts running from waypoints to determine if a script has been started or needs starting. This will allow for better use of indices and determination if various offmap events are taking / have taken place.
returns: FALSE - Not Loaded, TRUE - Loaded

RAND(<starting number>,<ending number>)
is used to generate a random number starting with the 1st number passed to the function, and ending with the last. e.g. RAND(1,20) will return a randomly determined number between 1 and 20. This function uses a flat distribution (i.e. each result is equally probable of being returned)
returns: Random number between Starting and Ending numbers, inclusive.

RCL()
is used to retrieve the value stored within the internal number accumulator.
returns: Number stored in bitwise Accumulator.

RCL_AND(<integer>)
recalls the number stored in the accumulator and logically **ANDs** it with the passed number, returning the result.
returns: Result of the recalled value in the accumulator and the logical AND of the passed value.

INTRCL("variableName")
is used to retrieve a stored value in one of the defined global integer variables for use elsewhere.
returns: The integer stored within a previously defined global variable.

Bitwise Commands

One of the the limitations of the Warbirds AI is that we cannot make serveral tests within an **IF** block, and we can't have an **IF** or **SWITCH** inside another **IF** or **SWITCH**. There is a partial solution for this. We have an internal accumulator (think storage box for a number) in the Warbirds AI. We can only store a single number at a time (i.e. if you store a number again, you will lose what you had in the accumulator previously. However, this allows us to make several tests, store the results (using binary switching) and respond based on the final result.

.bitsto [n] accumulator = n - Store a number into an internal register for use later
.bitor [n] accumulator |= n - Performs an inclusive OR between the accumulator and the passed number
.bitand [n] accumulator &= n - Performs an AND between the accumulator and the passed number
.bitxor [n] accumulator ^= n - Performs an exclusive OR
.bitnand [n] accumulator &= ~n - Performs a NAND
.bitadd [n] accumulator += n - Adds the passed number to the accumulator
.bitsub [n] accumulator -= n - Subtracts the passed number to the accumulator

.bitsl [n] accumulator << n - Slides all the bits in the accumulater **n** places to the left, inserting 0 at the right for each place
.bitsr [n] accumulator >> n - Slides all the bits in the accumulator **n** places to the right, inserting 0 at the left for each place
.bitres - print out accumulator in decimal notation (for testing purposes)
.bitreshex - print out accumulator in hexidecimal notation (for testing purposes)

Global Variables

Variables which are readable by any script running on the same AI General can be used. These are called global variables. Uses would include tracking numbers of victories by a side, supply or manpower levels, production levels, intensity levels for submarine or offmap strategic bombing effects scripts, and more. Slightly limited in use in that they are only really usable within **SWITCH** statements, global variables are nonetheless a very powerful addition to the WB scripting tools.

.intsto "varName" [n] - Stores an integer value into a global variable named varName
.intadd "varName" [n] - Adds an integer to the global variable named varName
.intsub "varName" [n] - Subtracts an integer from the global variable named varName
.intmul "varName" [n] - Multiplies a variable named varName by a passed number.
.intdiv "varName" [n] - Does an integer division (i.e. drops any decimals after the division) of varName by a number.

@varName@ - (used within the **.echo** command) Echos (prints) the value of the variable varName to the text box
.macrochar [ch] - Changes the character used in the **@varName@** command

These work with Strings also.

.strsto "varName" [n] - Stores a string value in a variable
@varName@ - will recall the string value similarly to the integers.

Conditional Operators

When using the if statement, we have the ability to test for certain conditions. For example, we can test if the country controlling a base **equals** a specific number. To do this, we can use the following test operations:

=	**Equals**
<	**Less than**
>	**Greater than**
<=	**Less than or Equal to**
>=	**Greater than or Equal to**

Bit Operators 101

Taking advantage of bits in storing results is not difficult, but does require a slight stretch of the imagination. A bit is a simple number, either 0 or 1. If we string several bits together (e.g. 10010110), we have a potential result. In the example, we had 8 bits. This could have represented 8 different tests (each which answered TRUE = 1 or FALSE = 0) or even a simple number from 0 to whatever.

This can be explained in the following example

```
%%
# slide the bits left to prepare for the new test result
.bitsl 1
if (TEST1)
{
# if the TEST1 were successful
.bitadd 1
}
# if the TEST1 failed, nothing need be done

# and repeat for each of the 8 tests
%%
```

The result of all 8 tests is now stored. How this translates into a number is defined by the binary number system. The rightmost bit represents 1. The 2nd rightmost bit represents 2. The 3rd bit is 4, and each bit further left is just 2x the value of the bit to its right. If a 1 is in a bit location, add its value, otherwise ignore.

Thus: **10010110** represents the places **(128: 1)(64: 0)(32: 0)(16: 1)(8: 0)(4: 1)(2: 1)(1: 0)** or **128+16+4+2=150**. To respond to this result, we could then use a **SWITCH** statement with a **case 150** in it to get the appropriate response.

Sliding Left and Right

Think of each bit as a box with either a marble in it (a 1) or nothing in it (a 0). When we shift left, we take each marble and shift it to the next box to the left. If we shift left 3 places, we move each marble 3 boxes, shift 4 places, 4 boxes, etc. This means that the box(es) on the right end of our group are now empty (i.e. have 0 in them). Numerically, shifting left has the effect of multiplying by 2, so shifting 3 places has the effect of muliplying by 8 (i.e. 2x2x2).

So...

If we have 23 (binary 10111) in the accumulator, then shift left 3 places with .bitsl 3, the result is 10111000. (note the 3 0s on the right now) This number is now 8 x 23 = 184.

Sliding Right is similar, except any marble in the rightmost box is dropped away - lost forever. Our example from before (23 - binary 10111) shifted right 3 places with the command .bitsr 3 becomes 10 (the 3 1s are dropped off the right end), thus equals 2 now. This is the equivalent of subtracting 1 from the number then dividing by 2 and dropping fractions.

Thus...

1 time) (23 - 1) / 2 = 11
2 times) (11 - 1) / 2 = 5
3 times) (5 - 1) / 2 = 2

Binary Operations

AND, OR, XOR, NAND - sound like some rock band initials at times, but are really just binary operations. Once understanding how they work, it's much easier to understand what they're used for.

AND is used for masks and other times when it's necessary to convert an accumulated number into a result dependent on another result.

How it works: whenever 2 corresponding bits in 2 numbers are both 1s, the resulting bit in the result of the and is a 1.

Example: **11100010101 AND 11000000111** Lets pair up bits...

11100010101 **1813**
11000000111 **1543**
11000000101 **1541** And only keep the 1s where a 1 appears in both numbers

OR is used making certain that certain conditions persist, like no matter what you do to a number, you what the condition represented by that (maybe it's a count of factories or something) to be retained for all friendly controlled bases. This represents an "Either Or or Both" type of argument.

How it works: whenever either bit in 2 corresponding bits is 1, the resulting bit is 1.

Example: **11100010101 OR 11000000111** Pair up bits...

11100010101 **1813**
11000000111 **1543**
11100010110 **1814** Pass through any 1 in either number

XOR is used to make certain that only 1 of several conditions is retained. This represents a "Or but not both" type of argument.

How it works: whenever either bit in 2 corresponding bits is 1, the resulting bit is 1 unless both are 1, then the resulting bit is 0.

Example: **11100010101 XOR 11000000111** Pair up bits...

11100010101 **1813**
11000000111 **1543**
00100010010 **274** Pass through a 1 if it appears in one number only

NAND is used as the opposite of **AND**. i.e. you want to turn off a condition if 2 other conditions are true.

How it works: whenever both corresponding bits of 2 numbers are 1, the resulting bit is 0, otherwise it's a 1.

Example: **11100010101 NAND 11000000111** Pair up bits...

11100010101 **1813**
11000000111 **1543**
00111111010 **506** Pass through a 1 unless there's a 1 in both numbers

.bitadd <num> does exactly that. It adds the passed number to the stored value in the accumulator and stores the result.

.bitsub <num> subtracts the passed number from the stored value in the accumulator and stores the result.

This leaves just **RCL()** and **RCL_AND(<num>)** to define. RCL simply recalls the value stored in the accumulator exactly as it is stored. RCL_AND will peform an **AND** function on the stored number with the passed number and return the result.

Putting It All Together

Now that you've had some experience putting together scripts, it's time for a serious one. This is an example script that could be used to determine the supply status level (based on the number of intact friendly factories and warehouses) for Bizerta (T5) in the Tunisia terrain. This level could then be used to determine the degree (and even open status) of all surrounding and supported bases near Bizerta.

```
# This file will check supply levels by running through all the strategic objects of a town
# and adjoining bases (defining the confines of a port area) to determine the supply level
# in that town (and possibly be used to determine the level of disruption the port should
#  experience)
#
# Example: Lets say we want to check the supply level of Bizerta in Tunisia (based on the
# the number of factories and warehouses surviving).
# We will be checking both Bizerta (T5) and its adjoining airbase (F4).
# There are 6 total ground objects we'll be checking.
#
# To keep track of strategic objects, we use the rightmost bits of the accumulator to store
# the number of undestroyed strategic objects at a base.  In this case Bizerta and its airbase
# are both controlled by the same side, so it will not be necessary to differentiate if a
# particular ground object is in a friendly controlled base vs. not friendly but still intact
#  otherwise.
#
# in this example: there are 8 objects we have decided to count towards strategic supply
# purposes in Bizerta itself and 5 more in the adjoining airbase.  This is a total of
# 13 possible objects that should be counted.

%%
# Prime the accumulator to clear any old values and be ready to count the intact structures
# We start with the number of possible structures and subtract the destroyed ones
.bitsto 13

# Test the first strategic object - Warehouse 02
if (DESTROYED(GROUNDOBJECT("F05WH02")))
{
# remove the object from the list of undestroyed objects
.bitsub 1
}

# Repeat for each of the other factories and warehouses in Bizerta we're checking
# Factory 03
if (DESTROYED(GROUNDOBJECT("F05FM03")))
{
.bitsub 1
}

# Factory 04
if (DESTROYED(GROUNDOBJECT("F05FM04")))
{
.bitsub 1
}

# Factory 05
if (DESTROYED(GROUNDOBJECT("F05FM05")))
{
.bitsub 1
}
```

(continued next page)

```
# Warehouse 06
if (DESTROYED(GROUNDOBJECT("F05WH06")))
{
.bitsub 1
}

# Warehouse 36
if (DESTROYED(GROUNDOBJECT("F05WH36")))
{
.bitsub 1
}

# Warehouse 44
if (DESTROYED(GROUNDOBJECT("F05WH44")))
{
.bitsub 1
}

# Warehouse 58
if (DESTROYED(GROUNDOBJECT("F05WH58")))
{
.bitsub 1
}

# Now check the objects for supply at the airbase
# Warehouse 11
if (DESTROYED(GROUNDOBJECT("F04WH11")))
{
.bitsub 1
}

# Warehouse 12
if (DESTROYED(GROUNDOBJECT("F04WH12")))
{
.bitsub 1
}

# Warehouse 13
if (DESTROYED(GROUNDOBJECT("F04WH13")))
{
.bitsub 1
}
# Factory 25
if (DESTROYED(GROUNDOBJECT("F04FM25")))
{
.bitsub 1
}

# Warehouse 26
if (DESTROYED(GROUNDOBJECT("F04FM26")))
{
.bitsub 1
}
```

(continued next page)

```
# Check for control of the base
# We AND to preserve the object count if friendly, make it zero if enemy
if (COUNTRY(FIELD(5)) = 3)
{
.bitand 15
}
else
{
.bitand 0
}

# Tell the world what the supply level is
# We've decided the following supply states:
# 0 = no supply
# 1-3   = defensive supply only
# 4-6   = basic GVs
# 7-10  = GVs and fighters only
# 11-13 = full supply with all available AC/GV
switch (RCL())
{
case 0
.echo Bizerta and its airbase are closed
break

case 1
case 2
case 3
# dot commands for establish the level of availability we want go here
.echo Defensive Supply only
break

case 4
case 5
case 6
# dot commands for establish the level of availability we want go here
.echo Basic GVs
break

case 7
case 8
case 9
case 10
# dot commands for establish the level of availability we want go here
.echo GVs and Fighters
break

case 11
case 12
case 13
# dot commands for establish the level of availability we want go here
.echo Full Supply at Bizerta
}
%%
```

Timing is Everything...

We also have the ability to use timed scripts in WarBirds. These will allow us to manage events that don't require a physical AI in the terrain in order to function. A possible future use (although not fully defined as yet) involves using the timed scripts as a sort of offmap event simulator, allowing for an offmap simulation to occur which affects events in the arena. An example would be if the terrain represented the Western Front. The timed scripts might be used to simulate events on the Eastern Front or Italy, thus causing some type of effect within the arena. This could be expanded to include that events in the arena could affect the offmap simulation also, possibly leading to one arena being able to affect the war in another.

The Syntax is:

.dtf_timed <script path> <repeat interval> <iterations>

if the iterations setting is left out, 1 is assumed. If iterations is set to -1, the script will run undefinitely. Presently, there is no limit to the number of timed scripts that can be loaded, and it is currently unknown what the maximum repeat interval can be. Still, this is a VERY powerful tool even as is.

.dtf_timedclear <script path>

Removes a given script from the timed list. If the path argument is left off, all scripts are cleared.

.dtf_timedlist

Returns a list (for human use) of the scripts currently loaded and their settings.

(Note) Need some stuff here on text triggered scripts and 1-shots

Types of AI Models

Each model is built from several AI files (each defining a different part of the model's behavior), and multiple (sometimes quite a number) of DTF (dotfile) files that glue the AI files together and provide the overall behavior of the model. Each type has a **Responsibilities** section which addresses the uses of the model, a **Contents** section which defines what the number and type of files needed to build that model, and a **Procedures** section which defines how the model functions to perform its responsibilities.

Sea Supply Convoy

Sea supply convoys provide an offmap source of supply (both to spawn offensive formations and rebuild friendly controlled bases). They typically are given 5 lives (sometimes more if simulating a very large or determined merchant marine) to simulate a merchant marine consisting of more than just the actual ships in the convoy model. At present, Kagas with the skill2 setting (i.e. Gunner ability) set very low (below 20) are used to simulate freighters. Destroyer escorts can be either the Fubuki or Fletcher destroyers. Light Cruisers can be simulated by setting the gunner value (again skill2) very high (80+). If we ever get actual freighters or CLs, we can use the real ones.

Sea Supply Convoys also provide the model for land based offmap supply sources. In this case, use Trucks instead of Freighters, and M16s instead of Destroyers.

Responsibilities

- Spawn Port AIs at receiver location (this may be a base, but doesn't have to be - e.g. Mulberry harbors)
- Spawn Invasion AIs (via the Offensive Formation model)
- Spawn Naval Operations
- Check control of the receiver port for possible switching to an alternative location

Contents

- 5 AI files defining each life of the convoy (including escorts)
- 1 Script file defining transitions for each life
- 1 Script file to test for control of the receiver port (for possible switching to a backup port). This will also define transitioning between lives of primary and secondary ports. (i.e. life1 of primary to life1 of secondary, etc)
- A miscellaneous number of DTF files for making ETA announcements, arrival on map, etc. (these may be mixed in with other test scripts)

Total: 5 AI, 2 (+?) DTF files

Procedures

1. At first waypoint, run dotfile with script to test for control of receiver port (DOTFILE_START). If uncontrolled, may either unload convoy (thus it never really arrives) or switch to a backup port
2. Follows path until at unload waypoint, then runs dotfile to spawn port AI (DOTFILE_END)
3. Sea Supply should always be paired with a land based 'port'
4. if killed runs dotfile to transition next life (DOTFILE_DEAD)

Port

Ports are the first land leg of the supply net. They rebuild bases within range (typically 20 miles) of the receiver port as well as disabling then reenabling AC/GV availability at the supported bases. They are responsible for spawning depots. They thus represent supply which must be renewed in order to exist. Ports are spawned via 2 methods. Either an offmap supply column (sea or land based) arrives and spawns the port, or the last life of the port respawns the first.

Responsibilities

- Rebuild bases (using the .restore_field command) within the affectation range of the receiver port (usually 20 miles)
- Enable/Disable vehicle availability
- Spawn Depots
- Spawn Offensive Formations
- Spawn Naval Operations

Contents

- 5 AI files defining the locations that the port will affect (one file for each base that the port will directly supply)
- 1 Script file for transitioning between lives
- 1 Script file for handling the death of a port incarnation
- 1 Script file to test for control of the port's base and spawn the port.
- 1 DTF file to unload the port if the port is killed on its first waypoint
- 1 DTF file to halt ops at each base the Port is to supply
- 1 DTF file to reenable basic GV/AC availability at each supported base (see: **Supply**)
- 1 DTF file to reenable all GV/AC availability up the limitations defined for that base (i.e. for all double pass supplied bases) for each supported base
- 1+ DTF file to possible spawn **Offensive Formations**, Depots, or **Naval Ops**.

Total: 9 AI, 4 (2+)(+?) DTF files

Procedures

1. Each life of a Port should be in a different location within the support area of that port. This is to keep attackers from easily disrupting port operations
2. Over the first waypoint, a port should do nothing (possible even just PAUSE - Halt type waypoint). This gives the base acks a chance to destroy the port in the event that it is enemy controlled
3. If the Port is destroyed on its first waypoint, run the DTF file to unload the Port. It will have no other effect on supply until the next cycle
4. Over the track of the Port, each base it is to directly supply should have its operations capability halted (i.e. all AC/GV availability is turned off) by running a haltops DTF (file: usually haltops#.dtf)
5. After a reasonable wait period (usually 5 minutes - during which the Port can be running over other waypoints performing other tasks), a resupply DTF is run (file: usually rs#.dtf) which reenables vehicle availability
6. If the support base is a double pass supplied, a 2nd DTF is run on another waypoint.
7. Ports can check on base control and start up response/offensive forces by referring to the trigger defined to start the offensive force

Depot (see Port)

Depots are the 2nd land leg of any supply network. They perform exactly like ports, but usually support a larger area.

Sentry

Sentries are the defensive formations in WB AI. They represent forward pickets and recce units operating out of command HQs, thus they simulate the dynamism of a front line. Sentry models are probably the most complex to build, since they are not simply point-to-point formations, but move out from a base and halt at their location of initial defense and must site good defensive positions. Sentries typically have 5 lives, but can have more or less depending on circumstances and the expected strength of the defended position. (e.g. the sentry of a major supply hub might be given 7 or 9 lives to reflect the generally much more difficult defenses of those bases)

Responsibilities

- Cover approaches to a base from likely attacking directions
- Announce that it is under attack when 1st life dies (and possibly subsequent lives)
- Lives are placed in good defensive positions to allow for maximum firing visibility
- Usually based on 2 GVs depending on terrain and the style of defense
- Spawns defensive replacement column to reenable player GV/AC availability if one of the Sentry's lives dies

Contents

- 5+ AI files defining each life of the sentry
- 1 AI file defining the replacement column for defensive supply
- 1 Script file for performing a control check on the base for the sentry (conditional loader)
- 1 Script file for unloading all components of the Sentry if the sentry is no longer needed because of enemy capture, friendly capture of a mor forward base, or regional surrender
- 1 Script file for handling the transitions between lives
- 1 Timed Script for checking supply conditions in order to spawn replacement convoy in the event that the 1st check failed
- 1 Script for unloading the Timed Script checking replacement conditions.
- 1 Script file for resupplying base (includes .restore_field and reenabling available infantry vehicles. Tanks must be restored via port, depot or offensive supply)

  Total: 6+ AI, 6 DTF files

Procedures

1. Each sentry life should be placed in a good defensive position with good visibility. Depending on the situation, the sentry may want to set up in a good ambush position
2. The last waypoint of any sentry life should likely be a **stop** waypoint, unless the sentry is patrolling an area.
3. The initial sentry position should be no farther than 4-5 miles from the base it's defending, unless there is a major crossroads or some other strategic terrain feature that needs covering
4. If a subsequent base is captured (i.e. is closer to a 'front') the immediate sentry to the rear should be unloaded to preserve AI file slots
5. As each sentry life dies, the next is spawned, if a selected life dies, the base the sentry is defending should be considered 'cut off' (i.e. closed)
6. When the defensive resupply column arrives, the 1st sentry life is loaded and all others unloaded

Notes: when placing sentries in their defensive positions, it is important to remember that AIs have a much lower 'eyeball' than we do. (1.5ft as opposed to about 5ft) Thus, the developer may have to place the AI much farther forward (hence forward of a good hull down position). It is also important to remember that the 1st GV in a group will rotate until pointing straight downhill because of differential traction. Each GV after the first is not affected, so sentry 'artillery' (i.e. M3s) are best placed in the 2nd (or later) position. To get a sentry to halt, place a **stop** type waypoint (see above) at the end of each life's track.

Offensive Formation

These are what do the assault work for the AIs. Offensive formations are responsible for the actual capture of territory. Players should not be using M5s or Ju52s themselves to capture. (in other words, player capture is disabled, except in those cases that the players may 'capture' an enemy base to disrupt supplies but cannot use that base) Offensive formations may also take the form of a response force (either air, land, or sea based) for a defensive strike or sudden counter offensive. Offensive formations should only be spawned by Ports and Depots typically (on occasion by Sea Supply Convoys) as they represent a serious use of supply that must be renewed.

Responsibilities

- Should move forward along strategically defined approach routes (both for sentries and to appear realistic - in other words, no driving over cliffs to get into position)
- Should pause for a typical 5 min (300 seconds) at the capture waypoint before the DTF running the capture is called (objectives simply didn't roll over by a formation just driving past)
- Should announce that they are leaving and that they will use an appropriate tac channel to keep players on its side updated to its progress
- Should call up an **Offensive Supply** column upon reaching an appropriate advanced position within its track (may be before reaching the objective)

Contents

- 2 AI files to represent the lives of an offensive AI formation (may be more depending on circumstances)
- 1 Script file for determining control of the formation's source and objective bases. (loads Off Form if both pass test) This file will spawn the Offensive Formation (may also announce that formation will be spawning)
- 1 Script file to transition between lives (including killing off the formation)
- 1 DTF file for each base passed through (to act as a capture mechanism). The objective base should unload the Offensive AI

Total: 2 AI, 2 (+?) DTF files

Procedures

1. The Port or Depot spawning the Offensive Formation runs the Loader script. If it passes (i.e. the source base is friendly and the objective base is enemy), the Offensive Formation will be loaded.
2. Capture each base the formation passes through
3. Pause at each base passed through (typically 5 minutes)

Notes:

One possible method for making this model feel more realistic is by using a basic AI as a forward screen of a larger formation. If that screen is destroyed, we would know which waypoint it was destroyed on. The **REAL** offensive formation (which could be much larger) is then loaded and has movement and formations in anticipation of fighting along the waypoint where the screen was destroyed. At some time afterwards, a new screen is spawned and resumes movement against the objective. This will vastly complicate the AI, since it will be necessary to have screen and combat AI formations for virtually every waypoint along an offensive's route, plus all the DTF files needed to glue it all together, but will make the formation more realistic and believable in its operations, plus being less predicatable. With the new RAND statement in the script files, we can end an AI file with a random choice of which next file to load and run, thus making the formation's route considerably less predictable.

Offensive Supply

Offensive supply formations are used to supply bases beyond the original front lines of a designed scenario. As such, they represent the fragility of maintaining supplies in frontal areas. It is possible that a Depot could be designed which also supplies these bases, but by using offensive supply only, you ensure that frontal areas are only supplied whenever new supply arrives from off map, and that supply is of a more vulnerable nature.

Responsibilities

- Always move from a base that would be supplied from a Port or Depot
- Have no limit to lives
- Provide supply to all bases the formation passes through
- Should use a WAIT type waypoint at every supply point.
- Unloads itself when done
- Provides supply for all types of vehicles, up the supply status of the base

Contents

- 1 AI file defining the path of the supply column
- 1 Script file to test for control of the source base and load the supply column. This would also check if the supply column is already loaded and active
- 1 DTF file to unload the supply column
- 1 DTF file per passed base to provide supply

Total: 1 AI, 2 (+?) DTF files

Procedures

1. Run the control test script to determine if the source base is under friendly control
2. Always are spawned from DTF files run by Offensive Formations from a Depot or Port location
3. Usually consist of only Trucks or a limited number of M16s for defense (keep the number of AIs down)
4. As the column passes through a base, it should pause (5 minutes is sufficient) before continuing
5. Calls a DTF to unload the Supply Column on the last waypoint (DOTFILE_END)

Trigger

Triggers are used to test for control conditions - either enemy or friendly. It may be assumed if a trigger survives that the base is friendly controlled and if killed is enemy controlled. (while this is not completely true, it is sufficient for a test condition, and the exceptions will add to the uncertainty in an arena)

Responsibilities

- Will spawn or not spawn a formation based on whether the trigger lives or dies
- Any DTF file results are the responsibility of the model incorporating the trigger into its design

Contents

- 1 AI file (with only 1 waypoint) defining the location of the trigger
- 1 DTF for starting trigger if in a sequence
- 1 DTF for starting another trigger in a possible sequence

Total: 1 AI, (+2) DTF files

Procedures

1. Should have only 1 waypoint. A halt (PAUSE) waypoint.
2. Should wait at least 1 minute at the first waypoint before continuing
3. Should be spawned in the middle of the test base's ack defenses (to increase the chances of being destroyed if the base is enemy controlled)

Notes:

Triggers are most often used in Garrison models. (the Garrison actually being the trigger) They can be parked at a base to trigger a quick response to the capture of a strategically important base (and that response wouldn't wait for a slower process like supply or timer scripts)

Naval Operations

Naval operations are perhaps the most interesting models to design. They can be developed to come in from offmap and bombard an enemy controlled base (via test control triggers). They can anchor near a port in the terrain until the port is bombed or captured, when the naval formation pulls out and move offmap or to another port. With the new RAND function, they can even choose a random entry position, or even a random strategy (via the AIs that are built).

Note: This model is still under complete development. I haven't yet fathomed all the nuances of them. I'll provide an example.

Allied Destroyer group at Bone (T3) in the Tunisia terrain.

Responsibilities

- Defend or Bombard Bone as per need
- Withdraw to offmap if Bone is axis occupied or bombed

Contents

- 1 AI file defining the approach route from offmap to Bone harbor (ends with a **STOP** waypoint)
- 1 AI file defining the withdrawal route from Bone harbor to offmap (which starts on top of the last movement waypoint in the 1st segment)
- 1 AI file acting as a trigger to determine control of Bone (2 waypoints, a location waypoint and a zero speed waypoint to force the trigger to wait indefinitely)
- 1 DTF file to unload 1st destroyer segment
- 1 DTF file to unload 2nd destroyer segment
- 1 DTF file to transition from anchored state (end of 1st DD segment) to withdrawal state (2nd DD segment)

Total: 3 AI, 3 DTF files

Procedures

1. DD 1st segment moves from offmap to Bone harbor. This segment ends with a **STOP** waypoint
2. Spawns the trigger in the center of Bone
3. If the trigger is destroyed, unload 1st segment and load 2nd
4. 2nd segment starts on top of the end of the 1st waypoint, so it appears indistinguishable from the 1st when starting

Notes:

Use your imagination

Garrison

Garrison models are used to ‘occupy’ a base for an unknown period of time or more likely until another event causes them to be unloaded or perform a given task. (e.g. a DD formation leaving its anchorage port, moving up to and bombarding an enemy port, then moving back to its anchorage) Basically, any time you have a formation which is needed to move, then sit and wait for an indeterminate period of time, then perform some action would use a Garrison AI model. This waiting period can be built in 2 ways: either use a trigger placed in a location where its destruction would trigger the subsequent behavior of the garrison, or use a timed dotfile which randomly loads itself or runs the subsequent behavior (see example)

Responsibilities

- Should use a Trigger to determine when to continue or leave
- Should always be placed in a location that will provide support to surrounding forces. e.g. a DD group in a harbor that has clear fields of fire in most directions.

Contents

- 1 AI file for each leg of the garrison’s life (2 minimum - 1 inbound, 1 outbound)
- 1 AI file to act as a trigger for determining additional legs of the garrison’s life
- 1 DTF file to act as a loader for the trigger
- 1+ DTF files that may be used to provide additional conditions for loading/unloading legs of the garrison.

  Total: 2+ AI, 1 (1+) DTF files

Procedures

1. The Trigger should be placed in a location that **will** be affected by the condition that causes the garrison to get under way. (DOTFILE_DEAD in Trigger)
2. Each leg of a Garrison should be placed so it will seamlessly load on top of the last position of the previous leg.

Notes:

Garrisons are most useful for naval operations, but are effective for positioning armored reserve formations that will respond to attacks or incursions, but it isn’t know ahead of time where the formation will be needed. A more complicated version of the Garrison would involve several triggers in multiple locations, each of which will load (and unload all the others once activated) the appropriate additional leg of the Garrison depending on which Trigger was destroyed.

Research

Research models are used to determine the order that certain vehicles or events will become available or occur within a larger environment.

Responsibilities

- Each incarnation of the research model should take an appropriate period of time to reach its destination (thus triggering the event the model is simulating), in order for other players to have a chance to help or hinder its progress.
- Research models should announce their progress when nearing completion. It is not necessary to inform anyone where the testing is taking place, but the friendly player base should be made aware that it is nearly ready.
- The scripting RAND function can be used to randomly determine locations of testing, so that enemy forces have a more difficult time raiding and slowing the development of hardware.

Contents

- 1 AI file for each stage of research (or delivery) required to bring a given aircraft or ground vehicle into an arena. This may also involve a 'testing' AI (i.e. the AI is tested at a base as prototype).
- 1 Script file defining the transitions between the stages.
- 1 Script file for effects resulting from adding the research technology or hardware.

Total: 1+ AI, 2 DTF files

Procedures

1. This can define almost any hardware which must be researched or developed before using.
2. All AC/GV should go through a series of develpment stages. e.g. design, prototype, initial testing, field testing, etc. before being 'released' to be used by the general player base.
3. If a prototype is destroyed, the research line may be forced to start all over again. Other stages of development may only be forced to restart at their beginning.

Notes:

These are extremely effective for driving RPS concerns in an arena. For example - sea convoys could be used to simulate the materials needed to research a particularly advanced type of aircraft or tank. If those convoys are destroyed, the AC/GV will never be built. They are very good for any situation where a location or base would be built up over time. A given base in an arena could be disabled at the start, but build up and add functionality over time (e.g. Peenemünde in Germany)

Under Development...(or "Cooked up in my mind but not built yet")

UBoat Pen / Strategic Bomber War

Basically will be a method for simulating the UBoat war in various parts of the world. Essentially, an AI will run a track. Each time it completes a lap, a script is run which randomly picks an effect. Among these effects are adding additional AI Runners to simulate more UBoats and destroying various convoys or AC/GV availability on the opposing side (to simulate loss of supply and replacement parts). If a Runner dies, a script is run and an effect is randomly chosen, among which are the elimination of the Runner. To eliminate the UBoat threat, the ports must be bombed over time until the Runners fail to respawn. Another method would be to capture designated 'supply' points, thus eliminating the threat immediately. This model could be used to simulate any offmap strategic war effect.

Antisubmarine activity could be built up from another AI based on ships. Every time the ASW AI runs **its** track, a random effect is chosen which determines whether or not a submarine is destroyed.

The same kind of thing can be done for strategic bombing. In this case the track AIs would be the strategic bombers and the strategic interceptors.

Offmap War

This would involve using timed scripts and strategic object checks. One that I was dreaming up was to have checks for the progress of an Operation Dragoon. If the axis is able to retain enough intact structures during each check, the progress of Dragoon is assumed to be retarded, since the axis might have had additional forces to oppose them in Southern France. If Dragoon procedes well enough, an armored force could be loaded that 'invades' the ETO map from the south. As the timed script ability is further developed by iEN, this model can be further determined.

(note) I need to include a writeup for the Malta Submarine simulation in the Tunisia scenario

Supply

When designing the AI for an entire arena or even a segment of one, it is necessary to define the final supply status of a base. This is useful from the point of view of replenishing supply and because 4 strategic models address supply (Ports, Depots, Defensive Supply, and Offensive Supply). Supply is provided via DTF files (although AI files provide delivery through their waypoints).

Typical file names for supply files look like:

Offensive Supply: sup<baseNo>.dtf
Defensive Supply: rsup<baseNo>.dtf
Port or Depot Supply: rs<baseNo>.dtf

A typical supply DTF looks like:

```
# Rebuild Base

%%
if (COUNTRY(FIELD(<baseNo>) = <countryNo>)
{
.restore_field <baseNo>

# Set Supply Status to Truck, M5, M16, and M3
.selectableon
.enable <baseNo> 79
.enable <baseNo> 64
.enable <baseNo> 66
.enable <baseNo> 67

# To halt all avalibility of AC/GV at a base, simply use 0 to turn them off.
# All AC/GV are represented by -1 in the availability command
# Example: turning off all AC/GV at say Gold base 14
# .enable 0 -1 314

# Halt Operations at Base No.
.selectableon
.enable 0 -1 <countryNo><baseNo>
}
else
{
# We might want to launch a counter offensive here
# but only on a 50/50 chance

.dtf_timed CounterAttack.dtf 1 1
}
%%

# End Checker
```

(Attack script on next page)

```
# Start Counterattack

%%
switch (RAND(1,2))
{
case 1
.loadmo myCounterOffensive.mbl
break

case 2
# we're not doing anything here, but you want to announce it
.radio 1 103
.echo Hey, baseNo has been captured by the enemy, get off your duff!
}
%%

# end example
```

Supply Procedure

Use this procedure whenever a supply event is run by a Port or Depot

1. Halt ops at the base to be supplied. This is done by simply disabling access to all AC or GVs except the M5 and Truck. An announcement may or may not be made. It is suggested that the Port or Depot run through an entire waypoint (preferably a PAUSE) before disabling ops in the event that the base in question happens to be enemy controlled. Control of that base should be tested in a script.
2. At a later waypoint, ops are reenabled. Either a base is singly supplied, in which case a single reenable event is reached (via the end of a waypoint), or it is doubly enabled which requires 2 such events.
3. On a first supply pass, all GVs (except tanks) and fighters are enabled (except perhaps the more sophisticated ones). On the second pass, any remaining GVs (up to the availability type of the base), and bombers (plus special fighters) are enabled.

Notes for Supply Model Version 3

Now that we have global variables to track values in scripts from one iteration to the next, we can now define a much more realistic supply model. The model basically works like this:

1. Ports run a script on their last waypoint which delivers AI offensives, Sentry Replacements (manpower), and Base Rebuilds. If a port iteration is destroyed, the points to be added by that iteration are lost.
2. Depots will run continuously (previously they were spawned by Port activity), and should have a fixed wait period (RESTARTTIME) inbetween 1st lives. As each Depot runs, it will run scripts that will check for offensive conditions and rebuild conditions.
3. Offensives will run if the base control condtions are met and if there are enough offensives in the the Depot pool.
4. Base Rebuild Scripts check for major damage to various bases supported by the Depot. As each requires rebuilding, a convoy is spawned if there is enough damage and if there are enough rebuilds left in the pool.
5. When the last life of a Sentry is destroyed, a Replacement Supply Convoy is spawned only if there are enough Manpower points left in the Depot pool.

Supply Status Codes

Ground
G0 = Truck and M5 only
G1 = G0 + M16
G2 = G1 + M3
G3 = G2 + PzIVD (or support tank for allies should we get one)
G4 = G3 + any Battle Tanks

Air
A0 = G0
A1 = A0 + basic fighters
A2 = A1 + bombers and advanced fighters

It is useful to go through all the bases of an arena terrain for a side and determine what the supply status will be for each one before designing any AI. This will give the designer an idea of what he wishes to consider "Important Real Estate" and as such should design sentries, supply formations, and offensive formations (or counter offensives) with this in mind.

Plane/Ground Vehicle/Ship Types

1	F6F5	31	P51D	61	Internal Use
2	F4F4	32	D3A Val	62	Internal Use
3	FM2	33	B5N Kate	63	Internal Use
4	F4U1	34	SBD Dauntless	64	M3
5	A6M5 Model 21	35	Ju88A4	65	M4A1 Sherman
6	A6M5	36	B25H	66	M16
7	A6M5 Model 52	37	B25J	67	M5
8	Ki43	38	B17G	68	PzIVH
9	Ki84	39	TBF1 Avenger	69	PzIVD
10	Bf109E4	40	B17F	70	Panther
11	BF109F4	41	B25C	71	F86
12	Bf109G6	42	P40B	72	A36
13	Bf109G6/R6	43	P47C	73	Kaga CV
14	Bf109K	44	P51B	74	Fletcher DD
15	Bf110C4	45	Seafire II	75	Mc202
16	Bf110G2	46	Spitfire XIV	76	Mc205
17	Fw190A4	47	F4U4	77	Fubuki DD
18	Fw190A8	48	Me262	78	Enterprise CV
19	Fw190D8	49	Yak3	79	Truck2
20	Hurricane Ia	50	Yak9D	80	P39Q
21	Hurricane IIC	51	Ju87D2	81	P400
22	Spitfire 1a	52	Mosquito FB Mk VI	82	Mosquito NF Mk II
23	Spitfire Va	53	Mosquito B Mk IV	83	Mc202-II
24	Spitfire IXe	54	Ju52	84	T-34
25	P38F	55	G4M2a Betty	85	M4A3 Jumbo
26	P38J	56	Ki61	86	Troop Transport (future)
27	P38L	57	Ju87G	87	Bf109F1
28	P39D	58	B24D	88	G4M1 Betty
29	P40E	59	B24J	89	G4M3 Betty
30	P47D	60	F4F3		

Ground Vehicle Formation Equivalents

Although each army in WWII had a different model for organizing their units, some basic organizations can be determined that will provide some standardization in design.

Name of Formation Type	**Size in Next Smallest Unit**	**Number of Vehicles**
Section	n/a	2
Platoon	2 Sections	4
Company	4 Platoons	16
Battalion	3 Companies	48
Regiment	2-3 Battalions	96-144
Division	2-3 Regiments	192-432

e.g. A Sentry with 5 - 9 lives would represent about 10-18 vehicles or about 1 company in strength.

It is sometimes helpful to define what level of defense or attack the designer wants before granting lives to various formations.

e.g. An Offensive Formation is planned to be aproximately a Division in strength. The designer had wished to make all defensive formations (i.e. Sentries) about a Regiment in strength, but decided to simulate this with Company formations only. Since the defenses are downgraded by 2 levels, it is advisable to downgrade the offenses as well (i.e. use Battalion strength attackers). The designer then decides to use an Offensive Formation of 12 vehicles with 3 lives, whose total of 36 vehicles provides the desired strength. If using 8 vehicles, the designer might want to use 5 lives (i.e. 40 vehicles) to represent the Battalion.

version 3.5
Jeff Stahl “Keller”

Contributions by:
Dan Hammer “Spindz”
Matt Davis “Target”
Guy Skaggs “Elvene”
Frank Baldwin “Randon”