Heist scripting document version 5.

CONTENTS.

Click on a link below to jump straight to that section.

Introduction.

- Getting started.
- Filenames.
- Where to save script files.

The Max file ('messaging').

Max entities.

- Darwin.
- Script / incoming / outgoing / recipient Parameters.
- Example of messaging.

Specific scripting objects.

- List of Max object types / properties.

Exporting.

- TBS maker.
- Debugging.

The script files ('scripting').

Codegenie.

Speech.

- Setting up and triggering speech.

Mission script.

- Script symbols.

Writing scripts.

- Notes.

ASL natives.

- List of natives functions.

ASL commands.

- List of commands.

Scripting examples.

- Spawn enemies example.
- Triggering and cancelling delayed speech.

Troubleshooting.

List of problems.

- Problems with the ASB file.

INTRODUCTION.

This document is an attempt to detail the Heist scripting system. Note that at the time of writing the scripting system isn't complete so it's possible that things mentioned in this document will be out of date by the time you get to it. In which case, tough.

The Heist scripting system is made up of objects inside a Max file (this side is henceforth called 'messaging') and several text files (called 'scripting'). The Max objects are strung together, with each one sending messages to the next when a given condition is met. The text files watch over pieces of the level that messaging can't deal with automatically and deal with larger game events (such as the time limit). Theoretically it's possible to script an entire level using just text files and ignore the Max objects altogether (and vice versa). But the reason the Max objects exist is to avoid levels ending up with scripting files made up of thousands of lines of code, and the whole thing ending up a nightmare to debug (which is what happened with Alias).

Getting started.

Each level requires the following:

- The 'design' Max file containing the level's collision mesh and various entities.
- At least one script file called levelname.ASL. The script files are currently being written in Code Genie, which can be found in H:\Programmers\Tools\CodeGenie. There are instructions on setting up the program in the same folder. Make sure you have up-to-date versions of the ASL commands, ASL examples and ASL natives files in the right places too (these files are in the CodeGenie folder).

Script file filenames.

You can call the level's ASL file whatever you like, but it can't have any spaces in its name, or the names of any of the folders in its path. So 'Bank_2/Lobby' is fine, but 'Bank 2/Lobby' is not. Spaces in either the script's name or Window's path will mean the level fails to

IMPORTANT!

For Heist the script files should be saved in:

H:\Resource\All_level_scripts\LevelX_scripts\PhaseXX_scripts.

There is also a folder here called 'misc_scripts' which should be used to place any generic scripts you come up with that the other designers can also use. Please comment any generic scripts quite heavily so that everyone else knows what's going on in them.

MAX ENTITIES.

The various messaging objects can all be accessed from the Max 'create' tab under helpers / asc helpers.

- Link nodes are used for setting up strings of entities that are form a single 'thing'. For instance patrol paths, tremblers along a wall, drop edges, etc.

- Markers are used whenever you need a visible object to appear in the world. By pointing the external AFF file at an object that object will be visible in game (subject to scripting commands – see later for details on this).

- Double box and double sphere are used to set up proximities that will react when a character steps inside them. They are made up of an inner and outer volume but at the moment the game only uses the outer volume – so ignore the inner one.

- Game nodes are used for pretty much everything else. You would use these for passing messages back and forth, pointing to external script files, spawning NPCs, etc. etc. You set them up by using the Darwin properties window (see below).

Darwin.

Darwin is the game independent Max plugin that deals with object properties and links. Everything Darwin related is controlled from its toolbar, which contains the following buttons:

1. As it's game independent Darwin has to be pointed at an XML file containing the specific entities for a given game. Clicking this will bring up a dialogue box where you can browse for the relevant file (the Heist XML file is generally stored in H:\Resource\Common\GameTypes\GameTypes.xml).

2. This will export your level to a specified folder.

3. This brings up the Darwin properties window that you'll be using to set up all the messaging in the level. Opening this with no object selected will allow you to browse or search for any object.

4. This will refresh the Max views, showing any changes you may have made to messaging objects.

5. This allows you to search for a specific messaging object by its properties. Note that this button is mostly redundant as there is a more powerful search option in the Darwin property window (button 3).

6. This will toggle showing the links from messaging object to object. Normally the links are only visible for a specific object when it’s selected.

7. This button is not used messaging / scripting. Clicking it causes your PC to burst into flames and your monitor to explode.

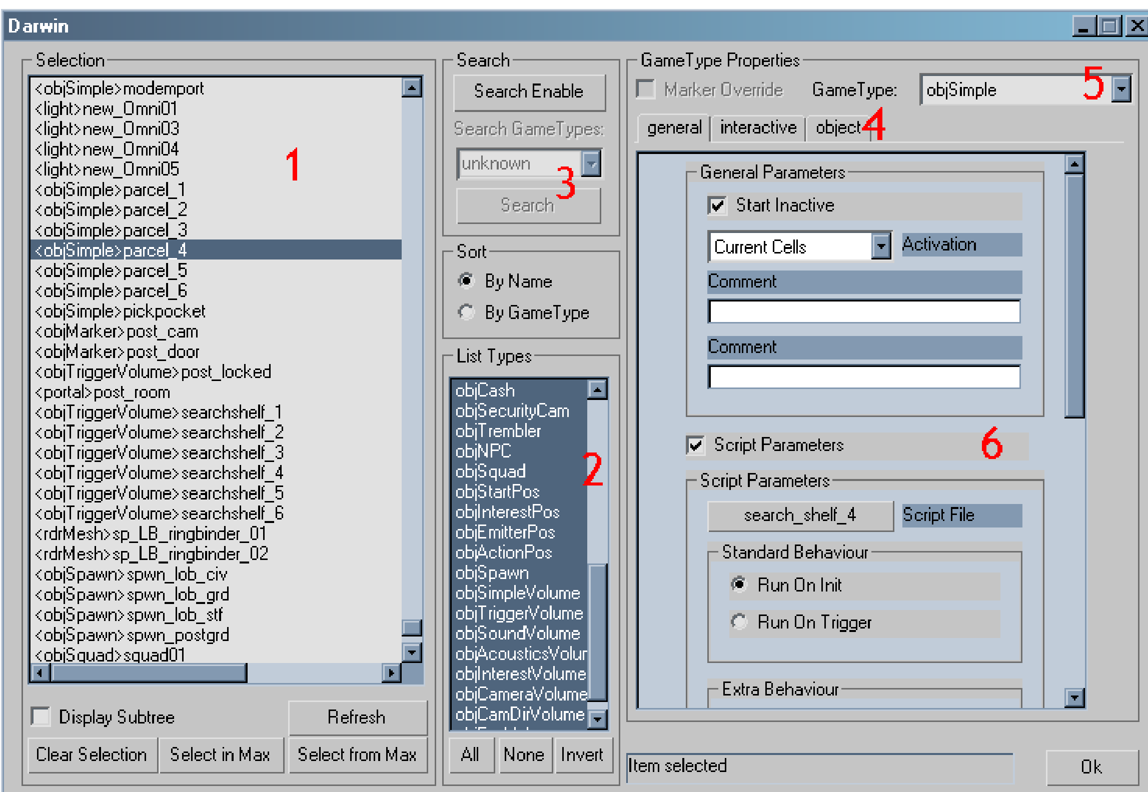
Darwin works by assigning entities a type based on the word they have in <> brackets, followed by a unique name. So <objSimple>blahblah.

Some notes:

- The word in the brackets IS case sensitive.
- No two objects can have the same name (or the exporter will fail). Note that the word in brackets is ignored as far as naming objects is concerned. So <objMarker>Moose and <objSquad>Moose are considered to have the same name.
- The name (after the brackets) can be up to 16 characters long. Objects should not include spaces in their names. NEVER start a name with a number (i.e. 2ndbadguy).

Darwin properties window.

This is where most of the messaging work will take place. From here you can assign entities their Darwin types and edit exactly what they do. Here’s what the various bits do:



1. This is a list of every object in the level, with various options to sort the list by name, type, etc. The properties of anything selected here will appear on the right side.

2. This is a list of ever possible entity type available. By using this box you can choose which entity types are displayed in the list to the left.

3. This enables you to search through the entities in the level by type and property. Click on search enable to start and then choose the type of entity you're looking for. Now you can either click on search to have it highlight every one of that type in the level, or use the parameters on the right to narrow down your search. Clicking search will then highlight every relevant object in the list on the left.

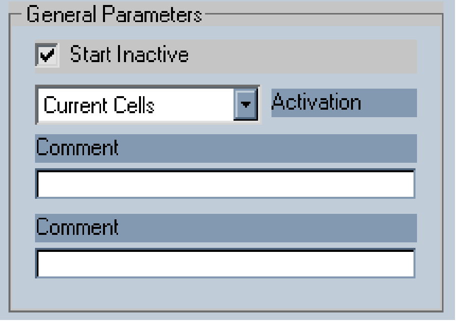
4. These tabs will change depending on the entity type selected.

5. This pull down list is where you assign the Darwin type to an object. Simply choose a type from the list and the <type> name will be assigned for you. Note that it won't let you assign inappropriate types to objects (a trigger volume to a single point entity for instance).

6. This is where the bulk of the messaging work takes place. The specifics of what appears here depends on the type of object selected. This window is divided up into general, script, incoming, outgoing and recipient parameters. By ticking or unticking the boxes to the left of the name you can show or hide each type of parameters (note that while it's possible to change a parameter and then untick and hide it, this will make debugging your level very difficult for anyone else).

PARAMETER SPECIFICS.

General parameters.



- Start inactive.

Generally objects will start a level already active but you can specify that they start inactive (so a door wouldn't open, an NPC spawner wouldn't create anyone, etc.). Note that simply activating an object won't necessarily make it do what you want, the object might need triggering too. There's more information on the differences between triggering / activating in the scripting section below.

- Activation.

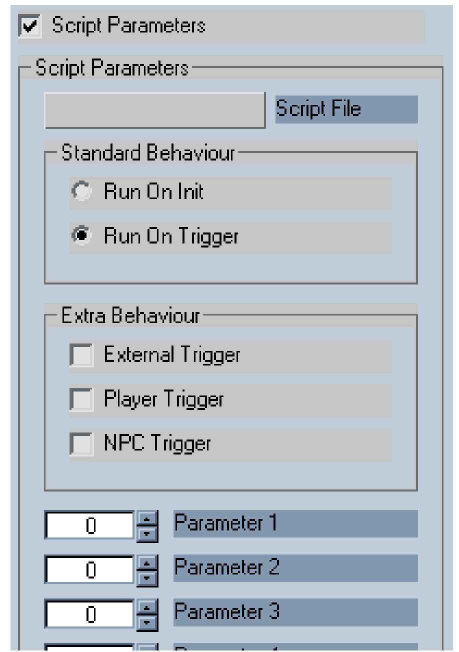
This specifies the rules for this object regarding cells. Normally an object is only active if it's in a loaded cell (the one you're in and any adjacent ones as a default). Changing this will allow the object to stay active even if it's not in the currently loaded cell. Note that

you should use this for the objects ‘mission start’ and ‘mission speech’ (see below for details of these) as they need to be used during the entire level.

- Comment.

These two boxes are for entering comments to make debugging easier.

Script parameters.



- Script file.

Most levels will use several script files to deal with very specific bits of scripting that would be difficult to set up using Max objects. Each of those scripts needs to be referenced by one of the objects in the Max file and clicking on this button will allow you to specify where the script (ASL) file is located. More details on scripts later.

- Standard behaviour.

If an object is set to run on init then it will come online and perform its job as soon as it’s activated. If however you set it to run on trigger then the object won’t do anything until something (another Max object or a script file) specifically sets it off. You could use this instead of ‘start inactive’ to keep an object asleep until you need it.

- Extra behaviour.

This determines what will trigger this object. You won’t need it very often but you might want to set a proximity so that only the Player or an NPC can trigger it by walking through it. I think external trigger means that it will only go off if called by another Max object and will ignore things like the Player, NPCs etc.

- Parameters.

These 4 parameters are used by script files. There are more details in the scripting section but basically you can change the numbers here and then have the script watch those numbers.

Incoming / outgoing parameters.

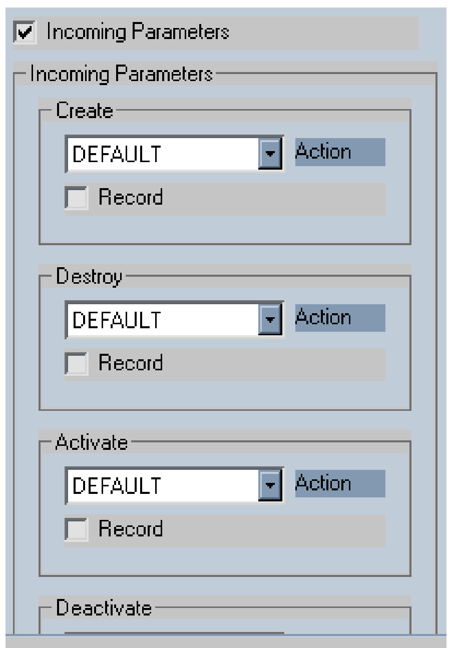
This bit of the messaging system is pretty complicated so bear with me. Although it’s a pain you are going to be using this stuff all the time so it’s worth sticking with it.

All Max entities have incoming and outgoing parameters which deal with messages coming in (from other Max objects or script files) and going out to other objects. The messages moving around are divided up into several types (listed below) including create, destroy, etc. The basic idea is that if you sent an object a create message then it

would create whatever is in its recipient list and then forward that message on to other entities. Whereas if you sent it a destroy message then it would destroy something, etc. This whole process sounds (a lot) more complicated than it actually is, so I've gone into more detail on it below.

- **Incoming messages.**

This deals with messages sent from other Max objects or a command from a script file. Once an object receives an incoming message it performs its 'default' action and then forwards that message onwards. More details on this below.



You can also use incoming messages to end the level, either in success or failure. You do this by changing the 'action' tab from default to something else. You'll note that there are several action tabs, one for each type of incoming message. This means that you could for instance end the level when the object receives a create message (by changing its default action under the create tab) or when it receives a destroy message (by changing the destroy default). Either one works so just pick one and use that.

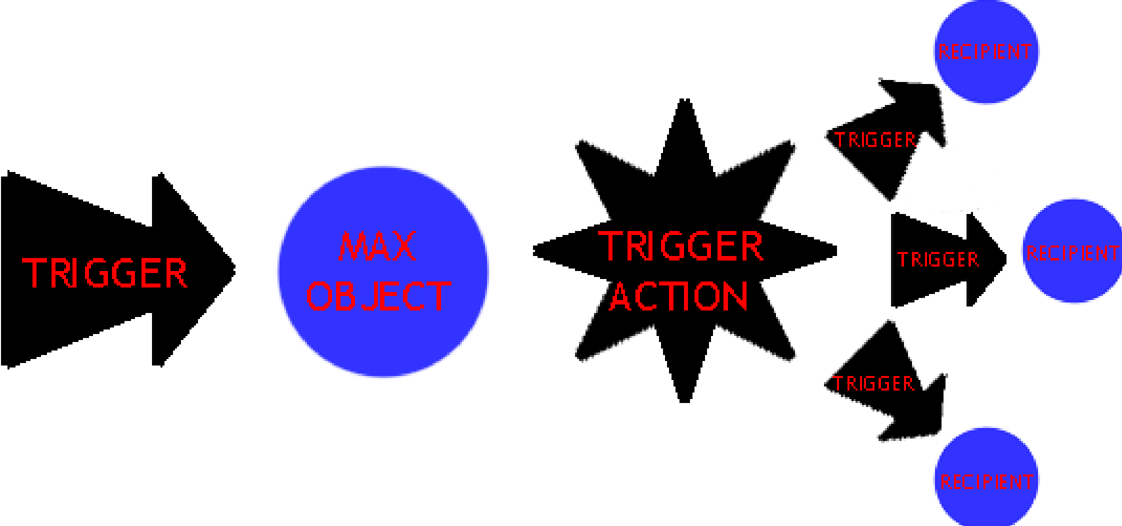
So as an example you could end the level by changing the default action of the create parameter to 'mission failed' and when you send that object a create message it will instead fail the level.

The incoming parameters also have a 'record' tick box. Clicking record on any of the above parameters will cause the game to note the time that this happened for use in the next phase. This works by sending the same message to the same object in the next phase. This is how we will set up concurrent gameplay in Heist, with messages being sent by the code at the same time the Player triggered it last phase. So to use this feature ensure that you have an object with a matching name in the next phase – that object will be triggered / activated / destroyed / whatever automatically at the concurrent time.

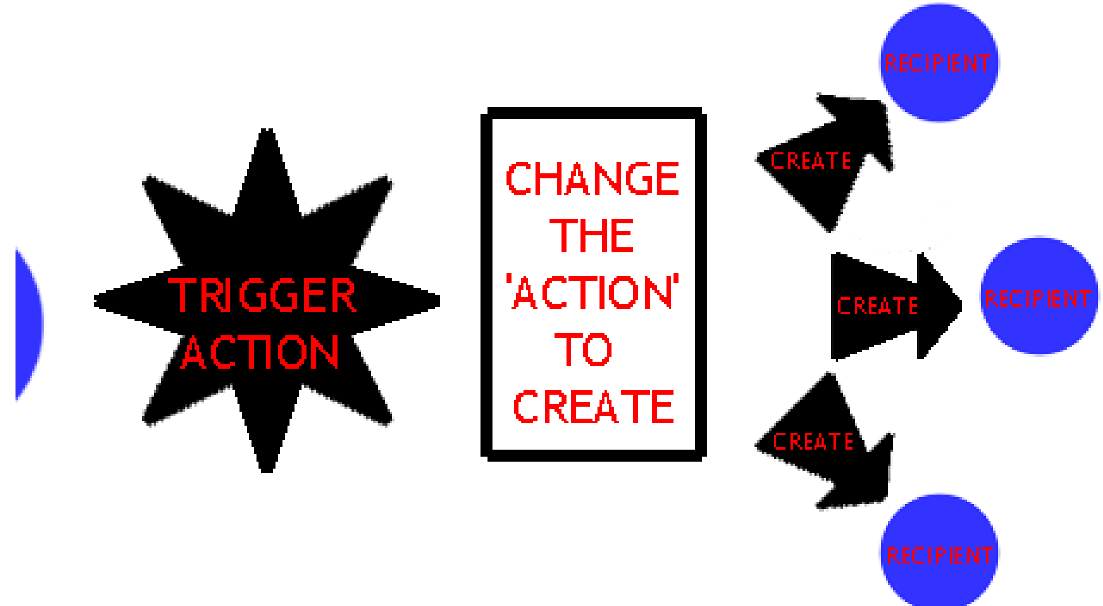
- **Outgoing messages.**

Outgoing messages are what the object will send out when it has received an incoming message and then carried out its normal role. After it's done both of those things the object will then forward on the same message type that it just received to all the other objects in its recipient list. So if you send an object a destroy message for instance then it will do its job (destroy something) then send a destroy message to every object in its recipient list (and those objects will continue the chain in exactly the same way).

The diagram below shows how the objects receive a message, do the job and then forward that message on.



So, first you send a trigger message to a Max object (the trigger message comes from a script or another object). The Max object then performs its 'normal' action based on what a trigger message means to it (so a door opens, a spawner spawns, etc.). Having performed its action the object then forwards on the same trigger message that it received to all the objects you have selected in its 'recipients' box. Those objects have now received a trigger message so they will continue the chain on by performing their actions, forwarding on the message, etc. etc.



The reason that the Max object above forwarded on a trigger message is that in its 'outgoing action' tab we have left the 'same' option selected. This means that when it receives message type A it forwards on message type A. It's possible to change the 'same' action of an object to something else. This will mean that it receives message type A but forwards on message type B (or whatever). This would result in the result above.

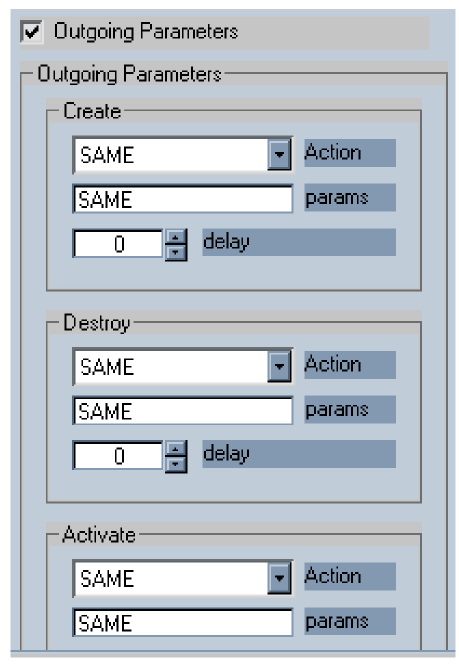
Note that the object still received a trigger message so it still did its standard trigger actions, but because you changed 'same' to 'create' it then forwarded on a create message to its recipients.

This approach makes the messaging system very powerful as you could quickly set up a string of objects that forward messages from one to the other, changing the type of message each time. The downside is that if you send an object a message then as a default it will forward that message on, possibly setting off other objects (and those objects will forward on messages again!). If not kept in check you could find that sending a message to an object results in half your level being accidentally triggered.

To stop an object forwarding on its messages you have two choices. First you can just not put anything in its recipient list (so the messages have no-one to be forwarded on to). Or you can change the action tab from 'same' to 'none.' This will mean the object receives the message and does its thing but then doesn't forward that message on.

Outgoing parameter actions.

There are 3 settings to each outgoing parameter type.



The first is the type of action that will be forwarded on. See above for details on this.

The second setting is for params (short for parameters). This is generally ignored so set it to 1, but some objects will use it. If you sent a message to the level's speech manager then the parameter entered here is the speech bank that will be played (more on speech later). The number entered here WILL be forwarded on to the recipients when the message is passed on.

The final setting is delay. If you use this then the object won't send the outgoing message until that many milliseconds have passed. This is exactly like putting a delay in the script file, so use whichever you like.

What the parameters actually do.

- Create.

This will 'create' (i.e. make an object appear in game) whatever's in the recipients tab. You would create something like a moving vehicle or a power-up.

- Destroy.

This will remove a specified object from the game instantly. Use it on NPCs to 'kill' them.

- Activate.

This will make an object active and aware. Objects that aren't active will ignore all scripting messages coming their way (apart from activate obviously).

- Deactivate.

Shuts down an object so that it ignores scripting commands. Use this on NPC spawners to stop them.

- Pickup.

This will delete the specified object and add it to the Player's inventory.

- Trigger.

This will make the specified object do its thing. So a speech object will speak, an NPC spawner will spawn NPCs, a door will open (or shut), etc. etc. You should read the scripting section later for the rules on using trigger versus activate.

- Damage.

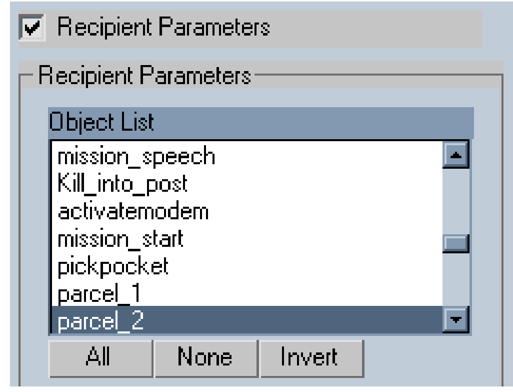
This will apply an amount of damage to the specified object. This is only really relevant to NPCs as most objects ignore damage messages.

- Update.

This is basically for coders to use. It over-writes an objects standard functionality with something else.

Recipient parameters.

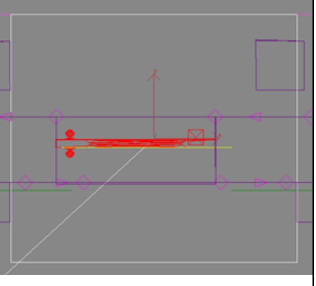
The object(s) selected in here are the ones that will receive the message type that you set in the outgoing section. So if you wanted to turn off a proximity box after a certain point you would select the deactivate action and then highlight the name of the proximity in the recipients.



Note that there's no way to distinguish which objects will perform which action. So if you chose create and trigger above and two objects here then both would receive a create and a trigger message. If you need one to create and the other to trigger you'll have to use two separate objects (or just use a script file – which can distinguish between objects).

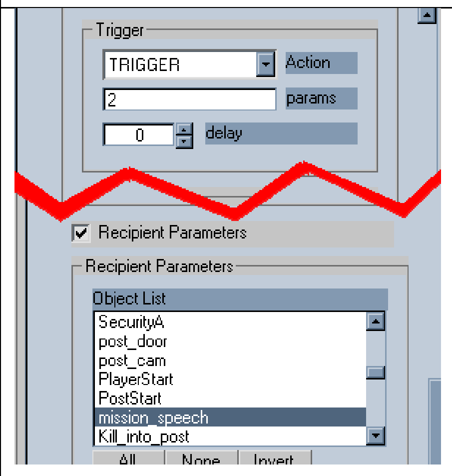
Incoming / outgoing example.

Seeing as the entire section above might be a little confusing below is an example of how messaging objects are used to run a section of a level 3, phase 1A.



A proximity box sits around the locked door. The box will trigger when the Player steps into it as its script parameters have been set up as:

Run on init (the prox-box is awake as soon as the level starts).
Plus Player trigger (so it goes off when the Player steps into it, not NPCs).



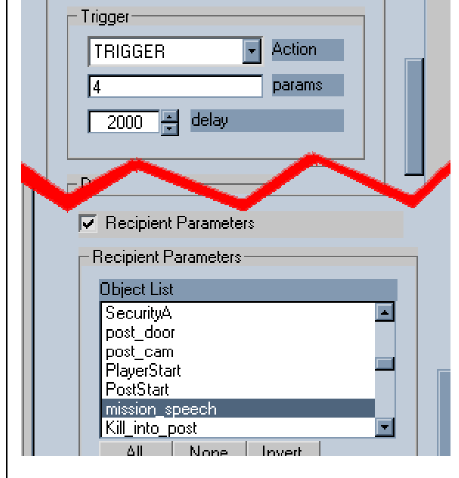
The Darwin properties of the proximity box have two other areas of interest.

In the outgoing section the ‘trigger’ parameter has been set to ‘trigger’ (as you would expect). The parameter for this has been set to 2.

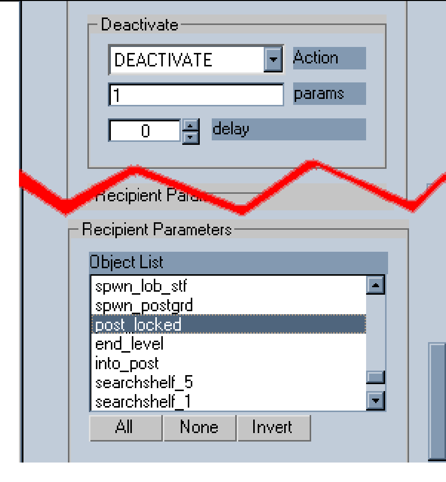
The recipient has been pointed at an object in the Max file called mission_speech. This means that mission_speech will be triggered when the Player activates the prox-box (by stepping into it).

Mission_speech will automatically play some speech when triggered, and the parameter 2 tells it to play speech from bank number 2 (in this case telling the Player to get lost). More on speech later in the doc.

At this point the Player goes off and does some stuff to unlock the door. It would take too long to list all that stuff here. Suffice to say the door is now unlocked.

<table>
<tr>
<td>
</td>
<td>With the door unlocked we need to do two things.

First play some speech telling the Player it’s open, second shut down the prox-box around the door (otherwise it will continue telling the Player the door is locked even though it isn’t).

One of the objects involved in the door has the set up on the left. It’s sending another trigger message to mission_speech, this time calling speech bank 4 with a delay of 2 seconds.</td>
</tr>
<tr>
<td>
</td>
<td>A second object involved in the door a different set up (on the left). This will immediately deactivate the prox-box called ‘post_locked’.

Note that we have to use two separate objects to get these two things going off. This is because if we used one object with both a deactivate and a trigger message then it would play some speech and deactivate the object mission_speech (which isn’t very helpful as we’re going to need that object a lot). The alternative to this approach is to set this all up in the script file. It’s entirely a matter of personal preference as the end result is the same.</td>
</tr>
</table>

SPECIFIC SCRIPTING OBJECTS.

The following is a list of all the current entities that can be placed in a Max file. This list may become out of date as new objects are added to the game by the programmers. The list is in alphabetical order.

- **Cell (N/A).**

These will generally be placed by artists. A given level is divided up into a number of cells, with only the cell you're in and the adjacent ones loaded into memory. AI characters that were active in a cell that has just been unloaded (due to the Player leaving the area) will freeze at their last location and start moving again if the cell is reloaded.

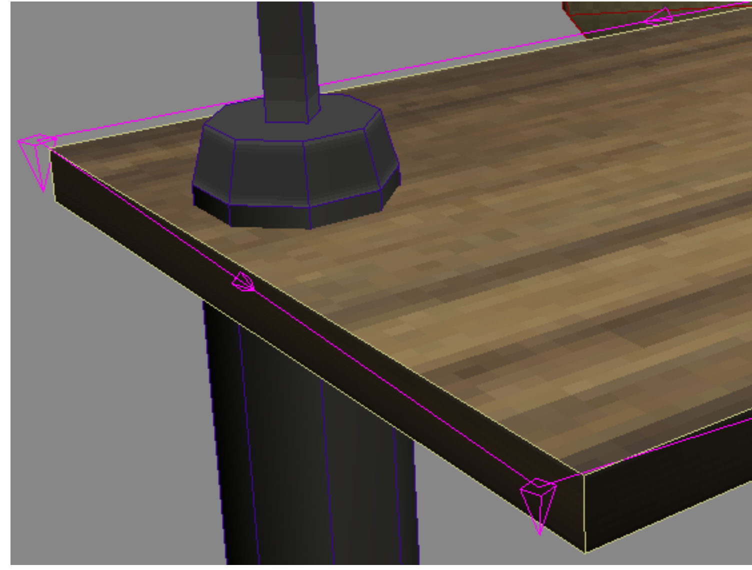
- **Colmesh (change geometry to this type).**

The Colmesh is what the characters collide with and generally it will be set up by the artists as a simplified version of the rendermesh. The only parameter you might need to deal with is 'floor include' which, when ticked, will punch a hole through the floormesh (below) of the level in the shape of that object. These holes in the floor are used by the AI to help it navigate. You would tick floor include for things like pillars, crates, desks, etc. You wouldn't tick it for walls around the edge of the level as the floormesh should deal with that.

- **Corona (N/A).**

Only used by artists. A type of fizzy pop which will give you wind. Manners!

- **Dropedge (linknode).**



These are set up in a line or loop around an object that you A) want the Player to be able to take cover behind, B) want NPCs to hide behind, and C) want the Player to be able to climb on top of (last one is optional).

Simply place a dropedge at every key point of an object (usually the corners) and then link them up using the 'add links' button. You can get it to link them for you as they're created by using the 'autolink from previous' tick box. Note; it's important that you link a loop of dropedges in an ANTI-CLOCKWISE direction so that it knows the outside of the loop from the inside. If you link them in a clockwise order it will think you can hide inside the object instead of outside.

Dropedges do not have to be a loop, you can just place them along walls (by corners for instance) or wherever. If you need to change height from dropedge to dropedge do so using right angles, not slopes. So you would move horizontally from DE1 to DE2 and then shoot vertically up to DE3 (exactly above DE2) before continuing horizontally to DE4 and so on. Sloping dropedges cause characters using them to slide.

If you want the Player to be able to climb up onto some dropedges then change their ‘drop type’ from ledge to floor. This will prompt the Player to press X when nearby to climb up (assuming the dropedges aren’t too high or low from the ground).

- **Dummy (N/A).**

These are from Alias so don’t use them. “A gottle of geer!” Eh? Eh? Eh?

- **Floormesh (change geometry to this type).**

This is a 2D plane created from the colmesh of the level (just clone the floor of the colmesh and change its type). The floormesh is used by the AI to navigate the environment so if you leave holes in it then the AI won’t be able to cope (although you could do this deliberately to constrain the AI). You will need to use ‘floor include’ in your colmesh (above) to set up the level’s floormesh properly.

The floormesh can have slopes and steps in it but it currently can’t cope with rooms being above other rooms (it’s counted as a 2D top-down map so it gets confused).

> Note; your levels MUST have a floor space, even if they don’t have any NPCs to use it, otherwise the level will fail to load.

- **Phase ().**

X chris

- **Light (N/A).**

Only used by artists. Will make your Max file float into the sky and poke God in the eye.

- **Lookatpoint (N/A).**

These are from Alias and currently don’t work. If you want to use them then speak to a programmer about getting them resurrected. Lookatpoints where placed around patrolpoints and would cause the NPC arriving at that point to look at each one in turn before continuing on their route.

- **Objacousticsvolume ().**

X andy

- **Objactionpos (gamenode).**

These are very similar to objsimples in that they can be used to mark a position in space, link messaging objects together and have scripts hung off them. They work fine but you might as well use an objsimple instead (as objsimples do everything this can plus more).

- **Objcamdirvolume (N/A).**

These are from Alias, so don't use. Have you seen 'Belly of the best'? You should.

- **Objcameravolume (N/A).**

These are from Alias, so don't use. It's a fantastic film. Easily Seagal's best work.

- **Objcash ().**

X chris

- **Objdoor (N/A).**

These aren't used. Instead place an objmarker in the level and point it at one of the door.AFF files.

Doors.

Doors have several properties than can be accessed by clicking 'marker override' in Darwin and heading for the 'door' tab. The main parameters are 'use keep open volume' which specifies whether a door will stay open when the Player or an NPC is stood nearby, and 'start locked'.

All doors are automatically active and can't be deactivated by scripting (because the door needs to be active to see if anyone is trying to open it – even if it's locked). Ticking 'keep locked' will prevent the Player (but not NPCs) from getting through the door. To unlock a door you need to send it a trigger command with one of several parameters:
Parameter 0 will close the door.
Parameter 1 will open the door.
Parameter 2 will toggle (i.e. open or close it depending) the door.

- Note that if you open a door using this method you need to change its 'close type' setting to 'manual' otherwise it will immediately shut itself again.

Parameter 3 will unlock the door without opening or closing it.
Parameter 4 will lock the door without opening or closing it.
Parameter 5 will toggle unlocking the door (i.e. lock or unlock it depending).

The final door tick box makes it an 'NPC door' which, as the name suggests, can only be used by NPCs (so they're placed by spawn rooms and so on). To prevent the Player getting through one of these they have an invisible collision sphere that expands outwards as the door opens. This will push the Player away as the door swings open.

- **Objemitterpos (N/A).**

These will emit particles into the level (flames and suchlike). While they can technically be used by designers they will generally be added to the levels by the artists.

- **Objfogvolume (N/A).**

Only used by artists. A very bad horror film. See also: The Blob and Mary Poppins.

- **Objgun (N/A).**

This is generally only used by artists. Each NPCs AFF file will contain one of these which tells that NPC which type of gun to use as a default.

- **Objinterestpos (N/A).**

Not used any more. How interesting. Ah ha ha ha. Sigh…

- **Objinterestvolume (N/A).**

Nobody knows what this is so don't put any in your levels. There are some things man is just not meant to dabble in. The horror!

- **Objmarker (marker).**

These are used to place actual physical objects in the Max file. Things like doors, pickups, physics objects and anything else the Player will be able to see in-game. To use an objmarker you need to specify the in-game object's model by using the 'external aff file' button. You can tick the 'display aff mesh' box to have the object drawn in Max (although this is not saved so will have to be re-ticked each time you load Max).

Markers are also used to create NPCs (not the Player – that's an objstartpos), but you'll need to see the patrolgroup entry below for details on setting up NPCs.

Objmarkers are used for doors, but see the objdoor entry above for details on them.

- **ObjNPC (N/A).**

These are not inserted into the levels directly, as they're taken care of automatically by pointing an objmarker at an NPC model.

- **Objprojectile (N/A).**

In general this will only be used by artists. It tells each type of gun which sort of bullet to fire. Also; a type of vomiting to be avoided at all costs.

- **Objsecuritycam (marker).**

This object covers security cameras and IR sensors (which are just cameras that don't turn and can be shot). Clicking 'marker override' in Darwin will allow you to change a camera's properties (on the securitycam tab). These allow you to change the speed, view range, starting angle, field of view, pitch, etc. Ticking 'track player on sight' will cause the camera to follow you around if it spots you (this makes the camera very dangerous). Ticking 'IR' will swap the camera to an IR sensor.

- **Objsimple (gamenode).**

Objsimples are the generic messaging object. They consist of a position in space, an orientation and a couple of optional things; a collision mesh (can touch the object) and a render mesh (can see the object). Objsimples are used for marking a point in space (i.e. is the Player close enough) and hanging scripts off (they're links in the messaging chain).

Objsimples have an ‘object’ tab in Darwin where you can set their health, mass and rendermesh(s). If you give an objsimple a rendermesh then it will be visible in the world. If you give it two rendermeshes then it will swap to the second one when it takes damage.

- **Objsimplepos (gamenode).**

Objsimplepos’ are used by spawn managers (below). These are simply a point in space and an orientation (the exact centre of the gamenode is used). See the section on objspawns below for details on why these are useful.

- **Objsimplevolume (dblbox or dblsphere).**

These are used to restrict certain NPCs to a specified area. If you tie an NPC to a simplevolume then that NPC will stay inside that volume as much as possible. The only time an NPC will leave a volume is if the Player has entered the volume and there’s no more usable cover for that NPC (in which case the NPC will leave the volume in its search for new cover).

Another interesting use for a simplevolume is that if an NPC is assigned to one but doesn’t start in it then that NPC will head for the volume and then stay there. This could be used to make NPCs in combat keep moving to a specified point somewhere in the level (while the Player tries to keep them away or something).

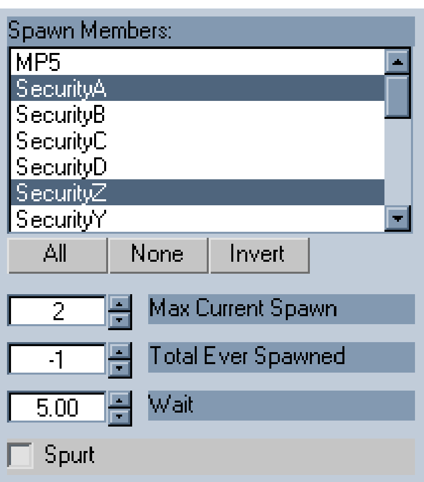
To use objsimplevolumes just place them over the area the NPC will be restricted to then open the Darwin properties *of the NPC*. Click on ‘marker override’ then go to the ‘NPC’ tab. Down at the bottom is the ‘bounding volume’ box where you can choose from all the objsimplevolumes in that level. To place a dblbox or sphere you just drag their XY size, then their Z height. Note that if you haven’t ticked the ‘single’ box you will now be expected to set another XYZ volume for the dblbox / sphere. This inner volume is not used at all in Heist so just set it to the same size as the outer one.

- **Objsoundvolume (N/A).**

These are placed by the sound guys. If you have a sound related joke insert it here.

- **Objspawn (gamenode).**

These are used to control the creation of NPCs into the world. Looking at the spawn tab of the object will give you several parameters to set up.



First is the names of every NPC in the level that this particular objspawn is in charge of creating. Each objspawn can be pointed at a maximum of 8 NPCs. Note that the NPCs will be created in the order that they appear in this list (which allows you to customise which ones appear where in a group, etc.).

Next is a list of spawn points. Displayed in here is a list of all the objsimplepos' in the level. If you select one or more objsimplepos' in this list then the NPCs from this particular manager will not spawn at their location in the world. Instead they'll spawn wherever the objsimplepos' are located. This would allow you to place some NPCs in the world as a sort of 'hit-squad' and then spawn them elsewhere (wherever you've placed the objsimplepos'). You could also point several spawn managers at the same objsimplepos' and so reuse them later in the level (ideal for sending in waves of Police or something).

Max current spawn is the number of NPCs that will be active at any one time (it will keep creating NPCs from its list until this number is reached).
Next is the total number of NPCs ever spawned by this objspawn. This tells it whether to keep re-spawning NPCs as old ones die. If you set this to the same number as the max current spawn then the NPCs will not be recreated at all. If you set this number to –1 then it will continue to spawn an infinite number of NPCs until deactivated by something else.
Next up is wait, which is the delay (in seconds) between each NPC being spawned.
Finally you have spurt. This will set the spawner up to create 'waves' of NPCs instead of a constant dribble. The objspawn will create its first NPCs (number set by the max and total spawn amounts) and then won't respawn any of them until every single one is dead. Then it will spawn the whole lot again (assuming it hasn't reached its totals).

Note:
Objspawns behave a little differently to most other objects as far as scripting is concerned. Normally objects have to be 'activated' and then 'triggered' (see the scripting section for details on this). However, objspawns do different things depending on whether they just been activated or triggered.

If activated an Objspawn will do exactly what is described above (creating a max and total number of NPCs, possibly respawning them, etc).

If triggered, an Objspawn will only create a number of NPCs equal to the parameter fed into it along with the trigger command. So if you said trigger (4) then it would create a total of 4 NPCs irrespective of the number entered into its total box. The difference between the two could be used to set off a few NPCs at one point during the level and then a whole load of them later on for instance.

Objspawns also have a spawn messaging box which allows you to customise when that manager sends out a scripting message. Entering a number in the box will cause the manager to send out a 'destroy' message when that many of its NPCs have died. That destroy message is sent to the recipients as normal (and can be hijacked and converted to a different message type if needed). Setting the number to –1 means the manager will never send a destroy message. Finally, note that the manager will only ever send one destroy message, not multiple ones (so if you set it to 3 deaths then it will send one then, but not at 6, 9, 12, etc).

- **Objstartpos (gamenode).**

You need one or more of these in your level to act as start positions for the Player (NPCs use objmarkers to start from). The first one in a level needs to be called PlayerStart, but subsequent start positions can be called anything you like. You would use the in-game debug menu to teleport between the start positions in a level.

- **Objsquad (gamenode).**

This is a relic of the old Alias system. All NPCs have to be a member of a squad, so just put several in the level and divide the NPCs up between them. Other than that objsquads don't do anything.

- **Objtrembler (linknode).**

These are set up as a chain of linknodes which then have their type changed to objtrembler. They will detect noise in an area around the chain (so a sausage shape with the linknodes running through the core). Note that to edit the properties of a chain of tremblers you always edit the first one in that chain. The others will ignore you.

The properties that can be edited are range (how far out from the spine of the trembler it will detect noise), sensitivity (how quickly the trembler will sound the alarm when it detects noise) and reset period (how quickly it forgets about noise).

- **Objtriggervolume (dblbox or dblsphere).**

This sets up an area that will register when a specified object moves into it. You can either use its various actions to trigger other things or have a script file watch it to see when a specified object has stepped inside. Trigger volumes can be activated and deactivated to make them pay attention or not, and can be set to notice the Player or other NPCs. Note that volumes need to be taller than the Player and sunk slightly into the floor to guarantee that they will detect him.

To place a dblbox or sphere you just drag their XY size, then their Z height. Note that if you haven't ticked the 'single' box you will now be expected to set another XYZ volume for the dblbox / sphere. This inner volume is not used at all in Heist so just set it to the same size as the outer one.

- **Objuse (gamenode).**

These are an expanded version of an objsimple. The extra bit involved is that an objuse has a position on it that the Player can press X to 'use'. This will cause the Player to play a specific animation and start whatever other behaviour is coded into the object (so a copier copies and lowers the heat bar, etc).

- **Patrolgroup (gamenode).**

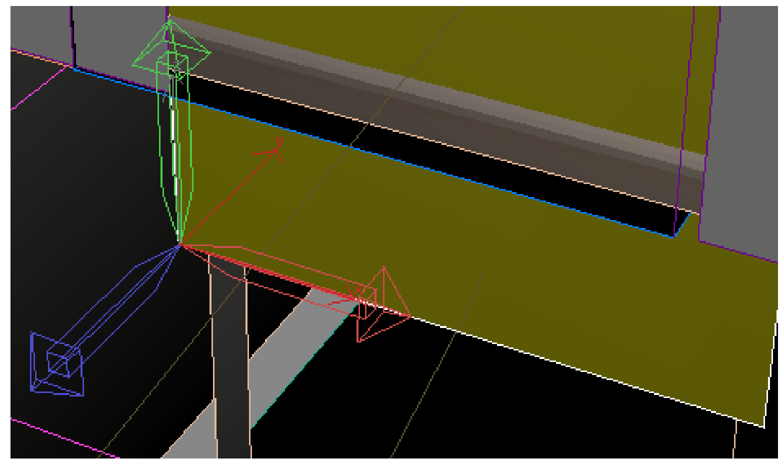
Patrolgroups are another legacy of Alias. They are used to tell an NPC which of the patrolpoints in a level are to be used by that NPC. The only parameter to set is selecting which of the patrolpoints will be used by the NPC in the 'patrol points' list.

To tie an NPC and its patrol group together you place them at the same XYZ position in the Max file (or at least very, very close). The actual name of the patrolgroup is irrelevant. So to make an NPC you need an objmarker (where you choose the NPCs AFF model), the first patrolpoint in that NPCs patrol route (see below), and a patrolgroup listing every patrolpoint used by the NPC. All 3 of these are at the exact same XYZ point.

Patrolpoint (linknode).
Patrolpoints are strung together to tell NPCs where to walk when they're not in combat (in combat their AI takes over and they use the floorspace to navigate). Basically you place a string of patrolpoints down and link them together (in the same way you would link dropedges). By placing the NPC at the start of the route (together with their patrolgroup – see above) they will begin patrolling as soon as they're created.

Patrolpoints shouldn't be placed too close to geometry as they AI may get caught on it. They should also be placed roughly 0.25 to 1 metre above the floor so that they don't get counted as being under the floor and ignored. Multiple NPCs can share a patrol route but they won't move if they can't see a clear patrolpoint in front of them (because another NPC is stood on it). You can place splits in the patrol route and each NPC will pick a different path when they reach them. Finally, patrol routes do not have to be loops, the NPC will simply stop when they reach the last patrolpoint.

- **Portal (plane).**



Portals are used as a gateway between two cells (above). They are created from a 2D plane which must be made up of only 2 polygons (set the length and width segments to 1 each).

The portal should be placed across the area where you will swap from one cell to another. Make a note of which cell the visible side of the portal (see diagram) is be facing, this is the 'source' cell. The other cell is the 'destination' cell. If you edit the Darwin properties of the portal you will find a tab to set the names of the source and destination cells from a list of the ones present in the level.

The final thing you'll need to set up with a portal is the placement of its pivot point. This needs to be snapped to the bottom left corner of the visible side (see diagram). This will means that the green and red arrows point at the edges of the portal and the blue arrow points at the source cell. To ser the pivot point up go to the 'hierarchy' tab in Max then choose 'affect pivot only'. You need to snap the pivot point to the corner precisely (using 2.5D snap set to vertex helps).

- **Rendermesh (change geometry to this type).**

This is the visible, on-screen geometry which is normally set up by the artists. You can't collide with rendermesh so you'll need colmesh in approximately the same areas.

- **Unknown (N/A).**

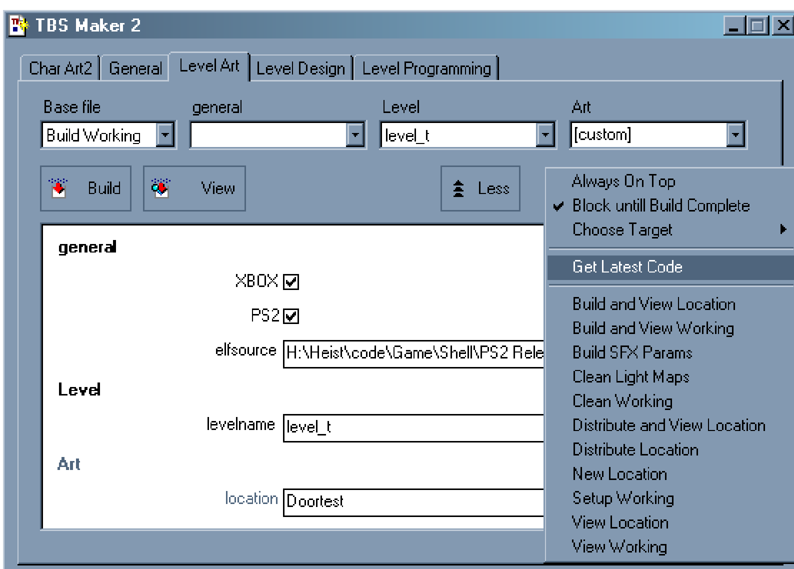
This is the default entity type in Darwin. You can't do anything to an unknown except change it to another entity type. If you try to change an object to an entity type it can't cope with (a gamenode to a proximity box for example) then it will become an unknown.

EXPORTING.

Obviously once you've set up your level you need to export and then build it (using the TBS Maker). Here are a few export notes:

- If you export a level somewhere and then save it the export path will be saved and the file will default to exporting to that same path next time.
- Hidden or frozen objects are not exported.
- If you select some objects (either normally or by using a selection set) when you export then those objects will be 'watched' when the game is running. This will provide you with extra debug information on what those particular objects are doing.

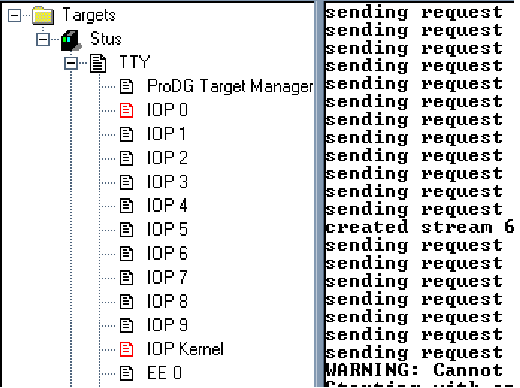
TBS Maker.



Note that you need a COMPILE_SCRIPTS line in the level's TBS file to make it build your script files. See the troubleshooting section at the end of this document for details on how to add this line if necessary.

Note. If you minimise the TBS Maker it will move to the right-hand corner of your taskbar from where its main commands can still be accessed by right

Debugging.



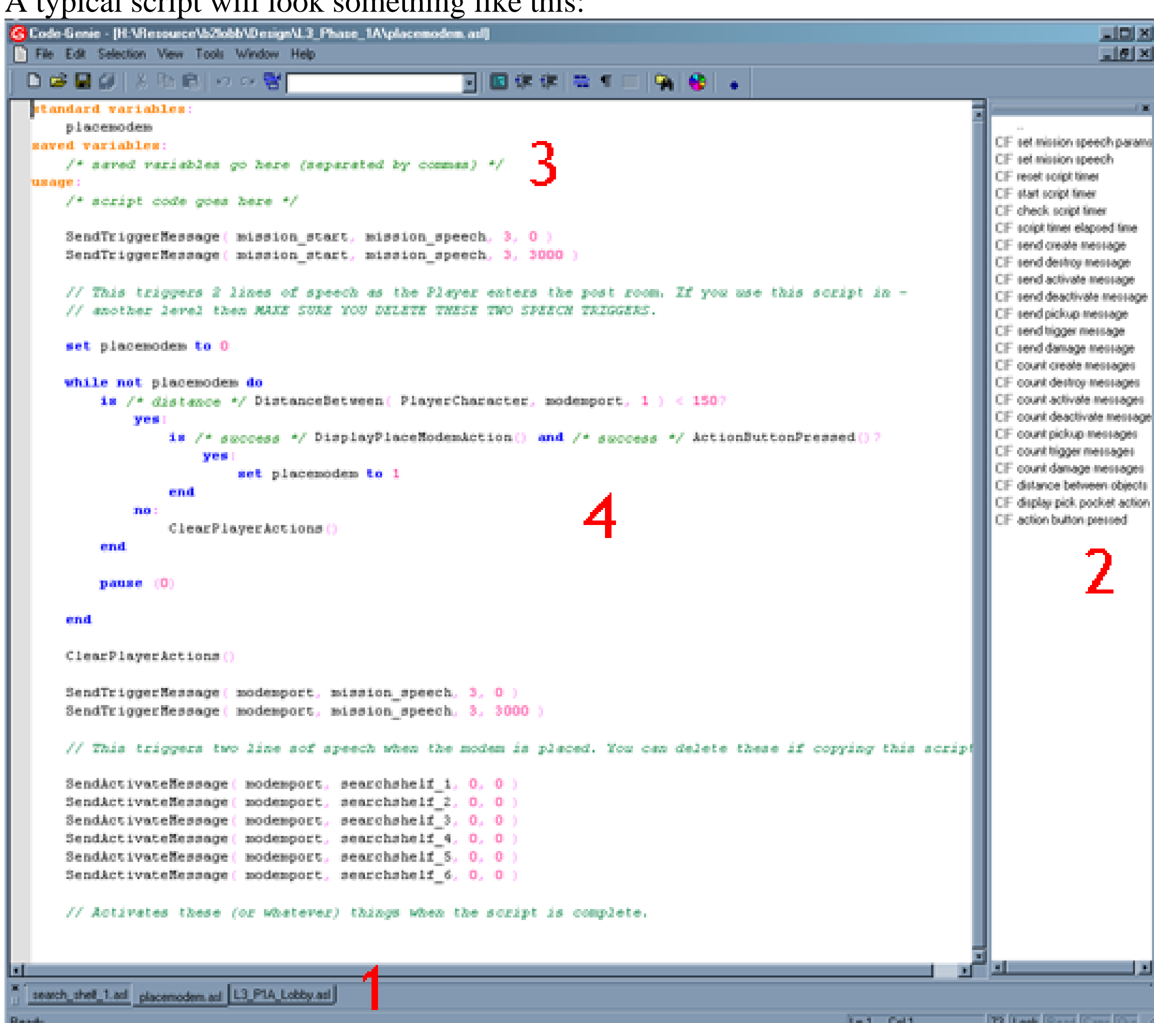
While the level is running you can find out exactly which bits of your script are being executed using the PS2's debugger (I assume it's similar for the XBox too but I haven't been able to try it).

Using ProDG target manager you can open the name of your PS2 in the targets folder, then open the TTY menu and click on EE 1 (emotion engine 1). The information on the right will display which bits of script are being called and which messages are flying about. Note that the window will not clear between PS2 restarts (instead it says TTY Channel 1 (Designer Output) opened), if you want to clear the window right-click and choose 'clear'.

THE SCRIPT FILES.

Every level needs at least one script file to deal with things like overall game events (the time limit) and to store a list of objectives, speech, etc. It's likely that each level (actually it's phases we're dealing with here but it's the same thing) will have several script files. In general each script will deal with one discrete event or action, so you might have a script to check if the Player has entered a specific area and another to check if the Player's health has dropped dangerously low. Technically this could all be done in a single script file per level but we've decided to separate them to make the whole debugging process as easy as possible.

A typical script will look something like this:



1. At the bottom of the screen are all the script files that are currently open.

2. To the right is the selector bar. It lists various bits of script that you will use over and over again. By double clicking on one you can insert that code into your script and just change the relevant details. There are various categories of sample code, stored in

'asl commands', 'asl examples' and 'asl natives'. You can move between them by double clicking on the top line (the two dots).
Note that if you don't have the bar to the right push ALT-F1 or F2 until it appears.

3. At the top we have three lines that appear in every script file. To insert them into a new file use the selector window on the right.

   The first line is standard variables. These are 'things' that you want the code to keep track of during the level. They are identified by a single word. So here we have a variable (also called VARs) called 'placemodem'. This will be used to see if the Player has placed the modem, so its state is 0 at first, changing to 1 when they have done so. Multiple variables are separated by commas.

   Next are saved variables. These are variables which are identical to the ones above except that they will be saved by the game (onto the memory card / hard disk / whatever). Apparently we won't need these much on Heist but they may be useful to Blue Pandora.

   Usage is basically the rest of the script (i.e. everything else) so just ignore it.

4. Below this is the bulk of the script itself. Basically the code runs through the entire script file from top to bottom, executing each command in order unless something prevents it (a time delay for instance). When the code gets to the bottom of your script file it stops and waits. This means that you have to specifically add a LOOPFOREVER command to the script to make it keep checking something over and over, otherwise it will check it once and then ignore it. More on this later.

   Text in blue is stuff that is recognised as being code.

   Text in black is the names of things in the level / script (names of commands, names of Max objects, names of variables, etc.).

   Text in pink is parameters (some things use these for specifics – exactly which speech line to play, exactly how long to delay, etc.).

   Text in green is 'commented out'. Any text with two // symbols in front of it is completely ignored by the script. Use this to fill your scripts with notes reminding you what does what and, more importantly, allowing other designers to read your scripts in case you're run over by a bus / wildebeest. You can also use it to add scripting to your levels even if the code isn't ready for it yet. Just comment it out until the code is ready to go.

SPEECH.

As mentioned earlier, each level has its own ‘master’ script file. This file currently contains the speech (well, the subtitles anyway) for that level. At some point the speech will be moved to a separate file so the script file will only contain pointers to that.

For the moment the speech is set up like this:

First you need to set up the level’s ‘speech banks’. Each bank contains one or more lines of speech, and every time you call a bank it will play the next line of speech in it. So you might have one bank for the level’s time limit and simply call it every minute so that it plays the next line saying “another minute gone.” Another bank might keep track of the Player’s health and would be called every time their health drops below a certain amount.

Note: There is a maximum of 16 speech banks in a level. This number can be increased so ask a programmer if you need it done.

Each bank has two parameters. The first is the speech banks number (so calling for this number will make the next line in it play). The second is the ‘setting’ for that speech bank. This can either be 0 or 1. Zero will make the speech in that bank loop when it reaches the last line (so play the first line again), whereas 1 will make that bank shut down when it has played its last line (and do nothing when called from then on).

```
standard variables:

saved variables:
    /* saved variables go here (separated by commas) */
usage:
    /* script code goes here */

    //////////////////////////////////
    //
    // Mission speech banks set up below here.
    //
    //////////////////////////////////

    SetMissionSpeechParams( 1, 1 )
    SetMissionSpeechParams( 2, 1 )
    SetMissionSpeechParams( 3, 1 )
    SetMissionSpeechParams( 4, 1 )
    SetMissionSpeechParams( 5, 1 )
    SetMissionSpeechParams( 6, 1 )
    SetMissionSpeechParams( 7, 1 )
    SetMissionSpeechParams( 8, 1 )

    //////////////////////////////////
    //
    // All mission speech set up below here.
    //
    //////////////////////////////////

    SetMissionSpeech( 1, "Ok, I'm in the lobby. I'll let you know when I've got the key from the guard."
    SetMissionSpeech( 1, "How's it going Carrie? You've only got a minute left.", 142, 86, 155 )
    SetMissionSpeech( 1, "You've only got 20 seconds left! Hurry!", 21, 56, 235 )
    SetMissionSpeech( 1, "We're taking too long! Abort the mission and get out of there!", 21, 56, 235 )
```

Below the speech banks you list all of the speech in that level using the command ‘SetMissionSpeech’. It has the following parameters:

First is the number of the speech bank that the line belongs to (so calling that bank will play this line if it's next in that bank's queue).
Next is the subtitle that will be displayed on screen (needs to be written inside speech marks). There is a maximum of 128 characters for this – including spaces.
Finally there is an RGB value telling it which colour to display the subtitle.

When you've run out of speech to include the rest of the script file is taken up with the actual level script.

MISSION SCRIPT.

Assuming you're using messaging objects inside your Max file the actual mission scripts for each level should be fairly small. Unlike Alias, most of the work is done by the objects inside max, leaving the mission scripts to deal with a few specific functions or things that are difficult to set up with messaging objects.

The scripts are also useful for dealing with special cases or odd events that crop up on a level by level basis. The script can be used to do absolutely anything to the game, without being bound by the 'normal' behaviour of an object or entity.

Scripting note 1:
One or more of the Max objects will point to the level's script files by path / name. If you move it or change the name then they will fail to compile.

You can see whether the level is successfully compiling its script by looking at the NEWCONV tool after its asked you if you want to skip an Xbox build.

Scripting note 2:
In addition to the required script file for each level, levels will use other 'mini-scripts' to deal with self-contained game events and report the results back to the main script or a Max object. Mini-scripts are useful for dealing with one off events in a level, but which might occur in another level too. By copying the mini-script and changing the names of bits in it you can avoid rewriting the whole thing.

Script codes and symbols.
There are various code commands and symbols that you may need when writing scripts. These include:

- Greater than is > symbol (so is X > Y?).
- Greater than or equal to is >= symbol.
- Less than symbol is < (so again, is value X < value Y?).
- Less than or equal to is <= symbol.

- TRUE / FALSE (these are used to see if something has happened or not – true can be written as 1 and false as 0, the code doesn't care either way).

- AND (are both value X and value Y true – if either is false then nothing happens, if both are true then something goes off).
- OR (is either of X or Y true – if so do something).

- IS (use this with AND / OR and the greater, less than stuff above. So you might say is X > Y?).

- WHILE (also used with the stuff above – so while X > Y do something, meaning that as soon as X is equal to or less than Y do nothing).

- NOT (this is a strange one, as basically it does the opposite of what the particular piece of code would normally do). I don't really understand it so ask a coder if you want to use it (you great big freak you).

WRITING SCRIPTS.

Instead of requiring us to write the script by hand each time, the most common commands have been set up for us to cut-and-past into the file. These are listed in CodeGenie under:

- ASL Natives.

These are the most commonly used functions and commands that will appear in scripts again and again. I've gone into each of these in more detail below so that you know what each one does, and can hopefully extrapolate what new ones do based on existing knowledge.

- ASL commands.

These are lower level script commands that will be used in combination with the above natives to fill out the script. Several of these probably won't be used but I've detailed the ones we might use below.

- ASL Examples.

These are programmer written examples of some of the commands. They're worth checking out at some point.

Note 1: Scripts will frequently refer to objects in the level's Max file by name. If you want to refer to the Player then the name to use is **PlayerCharacter**.

Note 2: The scripting language IS case sensitive so be careful. It's recommended that you do all scripting in lower case so there's no chance of a problem.

Note 3: Any time delays mentioned below are in milliseconds, so just add 3 000's to the number of seconds you want the delay to be.

Note 4: To use the natives and commands below in a script you would double click to insert one and then replace the text in green with your own names. The names are separated by commas so make sure you don't delete them.

ASL NATIVES.

SendTriggerMessage.

```
SendTriggerMessage( /* from */, /* to */, /* integer param */, /* delay */ )
```

SendTriggerMessage is used to send a message from one Max object to another (it can actually send a message from one object to itself if you want to be picky). Its main use is to trigger speech samples or events are set times.

If you send a Max object a trigger message then that object will perform its default action. A door will open (or close), a spawn manager will spawn NPCs, etc.

SendTriggerMessage has 4 parameters to fill in. The first is the Max object sending the message and the second is the object receiving it.

The third parameter is a number that will be fed into the receiving object. So if you were sending a command to play speech then the number here is the speech bank that will play. If the receiving object doesn't need a number parameter then it will ignore it so enter the number 1 for convenience.

Note:

Trigger messages will only work on 'active' objects. If the object in question is not active then the trigger message will be ignored. More on active

The final parameter is a time delay in milliseconds. The trigger message will not go off until this time has passed. Note though that even if you put a time delay in a command the script will read the command and set it off as soon as the level starts. What this means is that you have an unstoppable command which will go off at the specified time and there's nothing you can do to stop it (it's like the Terminator, only shit). You can get round this by using Max objects to trigger things, rather than setting them off at the start of a level with a delay. That way if you don't want the speech to play you just deactivate the Max object before it sends the signal.

SendActivateMessage.
SendDeactivateMessage.

```
SendActivateMessage( /* from */, /* to */, /* integer param */, /* delay */ )
SendDeactivateMessage( /* from */, /* to */, /* integer param */, /* delay */ )
```

These two are listed together as they perform two halves of the same function. Basically objects in the Max file can either be active (receiving signals, watching what's going on, possibly doing their default actions – like doors say). Or they can be inactive (completely shut down and won't respond to trigger commands). These are the commands you use to swap objects between the two states.

Note that if you deactivate something with an in-game model (i.e. a door) then it will stop responding to commands but will still be visible in the world. If you want things to deactivate and disappear then use the create and destroy commands below.

The differences between activate and trigger can get confusing. The rule is that things have to be active before they can be triggered. Think of it like a gun with a safety. The trigger does nothing until the gun is 'activated' (yes it's a crap analogy, just shut up).

SendCreateMessage.
SendDestroyMessage.

```
SendCreateMessage( /* from */, /* to */, /* integer param */, /* delay */ )
SendDestroyMessage( /* from */, /* to */, /* integer param */, /* delay */ )
```

These are similar to the activate and deactivate messages above. The difference is that if you activate or deactivate an object that object won't change from being visible or invisible. Whereas if you 'create' something it will become active and physically appear in the world, and if you 'destroy' something it will deactivate and be removed from the game world.

You can use this on any Max object including NPCs, making it ideal for removing NPCs that you don't need anymore (if you used deactivate on an NPC then they will still be visible in the world but won't react to anything).

SendDamageMessage.

```
SendDamageMessage( /* from */, /* to */, /* integer param */, /* delay */ )
```

This is similar to the destroy message above. The target object (usually an NPC) will take the amount of damage entered in the parameter section. If the target object can't take damage then it will simply ignore this command.

Note that the damage amount entered is NOT based on a percent, it's simply an amount, so you need to know how much damage each NPC can take for this to be useful.

SendPickUpMessage.

```
SendPickUpMessage( /* from */, /* to */, /* integer param */, /* delay */ )
```

This is one that will generally be handled in the code automatically but can be used manually. Basically the object specified (in the 'to' will be placed in the Player's inventory – and presumably removed from the game world.) You might use it if you want the Player to have gained a shotgun after a cutscene or something like that.

Script Timers.

```
ScriptTimerReset( /* timer */ )
ScriptTimerStart( /* timer */ )
```

The following few commands all deal with ScriptTimers. Any given script has 4 timers available to it at a time (you can reset a timer and use it again later). These timers are labelled 1 to 4 and can be started and reset using the commands above. You just enter the number of the timer you want to use in the parameter at the end.

```
/* elapsed time */ ScriptTimerElapsedTime( /* timer */ )
```

This will return the amount of time that a specified timer has been running for (in milliseconds). Again, it doesn't do anything itself but will be used by other code.

```
/* true/false */ ScriptTimerCheck( /* timer */, /* limit */ )
```

This will check whether a specified timer has reached a specified time limit and will return a true or false response (which can trigger other code or not). This command is useful because rather than just saying has timer X reached Y milliseconds you can actually specify a range of times. So you might say > 90000, < 100000 to check if the time is currently between 90 and 100 seconds, and do something very specific if it is.

SetMissionSpeech.

```
SetMissionSpeech( /* bank */, /* text */, /* r */, /* g */, /* b */ )     )
```

This one has been covered in the speech section several pages above.

CountXMessage.

```
/* number */ CountCreateMessage( /* object */ )
/* number */ CountDestroyMessage( /* object */ )
/* number */ CountActivateMessage( /* object */ )
/* number */ CountDeactivateMessage( /* object */ )
/* number */ CountPickUpMessage( /* object */ )
/* number */ CountTriggerMessage( /* object */ )
/* number */ CountDamageMessage( /* object */ )
```

These commands all keep track of the number of times a certain command has been sent to a certain Max object. It doesn't actually do anything by itself but is useful for including in other mini-scripts.

You might for instance keep activating and deactivating an object, but want that behaviour to stop when its been repeated a certain number of times. By including this command in a script with an increment (more on these later), the code would eventually count up to the required number and get the signal to stop.

To use a count message native you would probably put it in an 'is' command. So 'is countdamagemessage (max_object) = 1?' would do 'yes' if you had sent the item a damage message earlier in the level, or 'no' if not.

DistanceBetween.

```
/* distance */ DistanceBetween( /* object1 */, /* object2 */, /* 0 or 1, where 0 = 3D and 1 = 2D */ )
```

This command is used to check whether two objects are within a certain distance of each other. Again, this command doesn't do anything itself but it can be used in scripts to activate other commands (or whatever).

The distance at the start is in game units which, apparently, are centimetres. You provide the name of the two objects to watch as usual. Finally, the last parameter tells it whether to check in 3D space or on a 2D plane (you'd use the 2D option to stop the Player getting close to an object on the building floor above / below him).

ObjectInsideVolume.

```
/* success */ ObjectInsideVolume( /* volume */, /* object */ )
```

Another wotsit that doesn't do anything but can be used by other things. This simply checks whether a specified object has entered an <objTriggerVolume> proximity box. I've no idea what would happen if you entered the volume name as something that isn't actually a volume – probably nothing.

Ignore the word success at the start of the line, that's just the coders having a laugh.

ActionButtonPressed.

```
/* success */ ActionButtonPressed()
```

This simply watches to see if the Player has pushed the action button (defaults to X on the PS2 pad). Obviously this isn't any good on its own but if combined with the ObjectInsideVolume command above (and an AND statement) it can be used to check whether the Player has got close to something and pressed X to activate it.

Again, the word success at the start can be ignored.

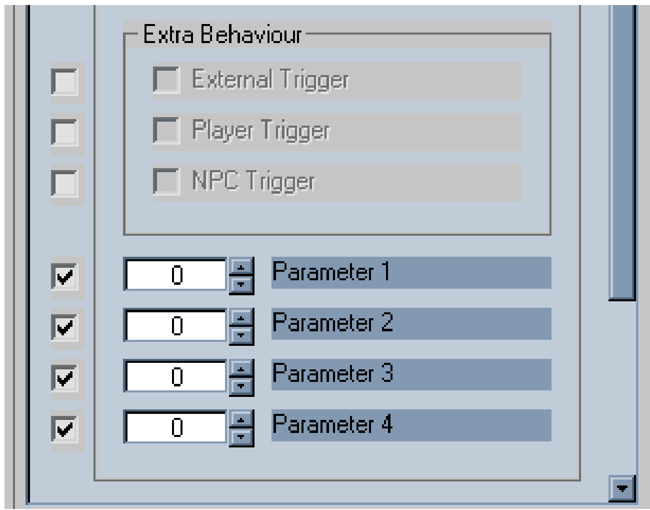
MaxXParam.

```
/* parameter */ MaxIntegerParam( ir, /* index */ )
/* parameter */ MaxRealParam( /* object */, /* index */ )
```

Right at the start of the scripting section I mentioned setting up any parameters you want the script to 'watch' during the gameplay. MaxParams are variants of that.

There are two versions but both are set up in the same way. If you edit the Darwin properties of a Max object you will find the 4 parameters (see diagram). Each of the parameters can be set to a different number and by entering that number in the 'index' section of the command you can watch that particular parameter.

The difference between an Integer number and a Real number is that Integers are whole numbers only (so no decimal points). Whereas Real numbers can have decimals, be fractions, etc. If it doesn't make any difference to you just go with Integers.



The word parameter at the start can be ignored as usual.

Cash.

```
/* number */ GetCarriedCash()
/* number */ GetDroppedCash()
/* number */ GetDroppedCashThisPhase()
DropCarriedCash()
GiveCash( /* amount */ )
```

There are several commands dealing with cash collection / carrying / dropping / etc. The first three will all return a number which can be used by the script to do other things (play some speech if the Player hasn't achieved a certain target for instance).

- GetCarriedCash is the amount of cash the Player character is currently holding. It doesn't count any money you've dropped off, lost, etc.
- GetDroppedCash is the maount of money you've dropped in this phase plus the money you may have dropped in the rest of the level. So in effect it's used for concurrency so that if you dropped $10 in the last phase you could use this and it would increment your money in this phase by $10 at the right time.
- GetDroppedCashThisPhase is only the money you've dropped in this phase. It doesn't include any of the money from the rest of the level. You would need this information if you wanted the script to react when the Player drops off a certain amount of money in this phase (so obviously you wouldn't want the money from previous phases getting in the way).
- DropCarriedCash will immediately force the Player character to dump any cash he's carrying. Any money dropped by this command WILL be added to the Player's money total for this phase. So if you include this command at the end of the phase the Player's money total for that phase will be money dropped off plus money carried. If you want to be evil don't call this and the money they're carrying won't get counted.
- GiveCash will immediately increase the amount of cash the Player is carrying by a specified amount.

As usual the word number at the start can be ignored.

ASL COMMANDS.

The following commands are used to 'fill in' the rest of the scripts and perform specific functions. After all a lot of the natives above will spit out information when requested, but you may need a commands to actually use that information to do something.

The following are the most commonly used commands:

Set var to value.

```
set /* var */ to /* value */
```

At the start of the scripting section I mentioned that at the beginning of a script you have to specify any vars (variables) that you want the code to keep track of. You give each var a (single word) name and for the rest of that level the code will be able to tell you what (number) value that var is currently on.

For instance you might set up a var called ‘pickups’. You would then set that var to 0 (see below) and increase it by 1 every time the Player picked up a power-up (see below for this too). A separate piece of script would be triggered when ‘pickups > 10’ and would play some speech telling the Player he’s a greedy bastard.

> Note: If you name a var the same name as an object in the level’s Max file then that var will automatically be set to ‘watch’ that object. You would generally only use this with the ‘elements’ and ‘reference name’ commands below, but you just need to be careful with what you call your variables.

So assuming you’ve defined some vars at the start of your script the following commands all deal with what you can actually do with those vars.

Firstly you can ‘set var to value’. This would be used near the start of a script to set one of the vars you set up to a specific number (usually 0 but not necessarily). You could then be certain that each time you increased or decreased that var it was definitely starting from a certain number.

Inc / Dec.

```
inc( /* var */, /* amount */ )
dec( /* var */, /* amount */ )
```

These two commands simply add a specified amount to a specified var. So ‘inc(X, 1)’ would add 1 point to whatever value the var called X was currently on.

Variable elements.

```
/* variable name */ has /* number of */ elements
```

Normally each variable consists of a name and a single number. By using the above line in a script you can ‘break up’ a single variable number into several numbers, all of which will be maintained by the code. You put the name of a variable in the first section and the number of elements you want in that var in the second bit.

Once you’ve done this you can refer to the elements in that var by using var [X], with the number in the square brackets being the element number you want. Other than this it follows all the normal var commands, scripts, etc. perfectly normally.

So for instance you might specify that a variable called 'spoon' had 10 elements in your script. You could then do things like 'set spoon [3] to 7' which would set the 3rd element of spoon to the value 7. Then you could 'inc(spoon [3], 1)' which would increase that particular element to 8.

Variable / Reference name.

```
/* variable name */./* reference name */
```

This command allows you to 'watch' a particular value from one of your variables – providing that variable is an object. Let me explain. The first part of the command is the name of the variable in question. In this case the variable has to be the name of an object in the Max file (remember if a variable matches the name of an object in the Max file then the two are linked together).

The second part of the command is the value you want the code to watch. The value you choose will not work until you've asked a coder to set it up for you. But once it is set up you will be able to use the script to find out what that particular value is at any time.

So for instance you might set up a var called 'guard01' and ask the coders to set up a reference called 'health'. By using the command 'guard01.health' you can find out what that guard's current health level is.

Is statement.

```
is /* conditional test */?
    yes:
        /* condition true script code goes here */
    no:
        /* condition false script code goes here */
end
```

This chunk of code will check to see if something specific has happened and set off whatever command you like if it has (and / or if it hasn't).

The 'is' statement at the start can check either a variable (which you defined earlier) or a function (another piece of script).

To test a variable you simply put 'is var (replace this with the name of the var) > X?' (where X is a number). So 'is spoon <= 4?' will check to see if the variable called spoon is equal to or less than 4 and do the 'yes' command if it is and the 'no' command if it isn't. You can also use the variable elements in here (so 'is spoon [4] > 10?').

To test a function (another ASL native or command) you can simply cut-and-paste some other ASL native / command into the 'is' bit. For instance you might put into it 'is

ObjectInsideVoulme (volume name, object name)?' To test whether a certain object was inside the volume at the time (and do the yes command) or not (do the no command).

Note 1: You do need a 'yes' command for the script to do if the result is yes. You don't have to use a 'no' command if you don't want. If you delete the word 'no' then the script will simply do nothing if the result is not yes.

The 'yes' and 'no' script commands can be any other scripting you like, so you can just cut-and-paste something else in either one. Continuing the example above you might do the following:

```
'is spoon <= 4?
        yes: SendTriggerMessage( mission_start, mission speech, 7, 2000 )
        no: dec( spoon, 1 )
end'
```

Which would check to see if the variable called spoon is less than or equal to 4. If it was then two seconds later it would play some speech from bank 7. If it wasn't then it would decrease the value of spoon by one. Assuming you set this script up to loop (see below) then sooner or later spoon will drop to 4 and the speech will play.

Note 2: Like the rest of the script this 'is' check will only happen once (and probably right at the start of the level at that) and will not repeatedly check. To make it check over and over you need to use the loopforever command below too.

Loopforever.

```
loopforever
    /* looped script code here */


end
```

If you put some scripting inside this command then that piece of scripting will literally loop forever. The code will check what's inside and do whatever its told and then instantly start doing it again. If you set up one of these in a script you can use it to check if something particular has happened, whereas if you don't put a check in a loopforever then the code will check it once and then forget about it.

Something important to be aware of regarding loopforever is that it will completely shut down the rest of the game if you just use it as is. Because the code is running over and over again nothing else can get a look in to run too. This means that everything will

simply freeze until the loopforever has finish. Obviously this is A Very Bad Thing. To get around this problem use the pause command below.

Pause.

```
pause( /* value or variable amount */ )
```

The pause command will immediately make the script wait for the designated number of milliseconds. The main reason to use it is placing it inside a loopforever command so that the rest of the game gets a look in (every time the loopforever hits the pause command it will be shut down – the rest of the script can then execute before the loopforever has another go). If you want to use the pause command for this reason then it's fine to set the delay to 0 milliseconds, just the fact that its been told to pause is enough for a loopforever, the actual delay is irrelevant.

Make sure you put the pause command inside the loopforever command (in front of the word 'end') for it to work.

While.

```
while /* loop condition here */ do
    /* looped script code here */
end
```

This command is the same as a loopforever but with an added condition at the top. Basically while the condition entered into the while bit is true then the script below will be executed. As soon as the condition becomes false then the looped code inside will be ignored. Don't forget that as this is a loopforever in a fancy hat you need to include the pause command in it somewhere or the game will fall over.

The condition can be any of the usual commands such as var < X, or object inside volume, and the looped script is the same as the code you would include in a loopforever.

Output.

```
output( /* output values here */ )
```

When called by a script this command will output whatever you've typed into the brackets. This output won't appear on screen (use the speech commands for that) but will appear in the debugger. It's mainly for debugging (such as watching to see when a specific piece of code has activated, etc.).

SCRIPTING EXAMPLES.

The following are a few examples of the script actually doing something useful. Everything in these scripts should be covered in the various natives, commands, etc. above so hopefully you can get an idea of how everything hangs together. The actual saved ASL files for these scripts are in: H:\Resource\All_level_scripts\Misc_scripts so you can just cut-and-paste them into your levels if you like.

Note that most of these examples could be set up without using much scripting at all. By using Max messaging objects you could get most of this stuff working without typing anything at all. A string of objects that trigger and deactivate each other could manage all of these functions one way or another. If you do decide to set everything up in Max make sure you add comments to all the objects so that others can debug your work.

Example 1:

Setting off a group of enemies after the Player steps into an area. Less enemies are created if the Player is badly hurt when this kicks off.

```
standard variables:
    gotkey
saved variables:
    /* saved variables go here (separated by commas) */
usage:
    /* script code goes here */

set gotkey to 1

while gotkey do

    is ObjectInsideVolume( enemies1_go, PlayerCharacter )?

        yes:
        SendTriggerMessage( enemies1_go, mission_speech, 1, 4000 )

        SendTriggerMessage( enemies1_go, spawn_01, 1, 6000 )
        SendTriggerMessage( enemies1_go, spawn_02, 1, 11000 )
        SendTriggerMessage( enemies1_go, spawn_03, 1, 16000 )

        SendDeactivateMessage( mission_start, enemies1_go, 1, 0 )

        dec( gotkey, 1 )

        no:
        // Do nothing.

    end

    pause( 0 )

    is PlayerCharacter.health <40?

        yes:
            SendDeactivateMessage( mission_start, spawn_03, 1, 0 )
        no:
            // Do nothing.

    end

    pause( 0 )

end
```

First up we establish a var called ‘gotkey’. This will be used to disable the script once the Player has got they key (or whatever) by making the loop a ‘while’. If we used a loopforever instead then the script would keep looping for the rest of the level, even after it had done its job.

Next we set gotkey to 1. This means that the ‘while’ will trigger (because while only triggers on ‘true’ results and won’t on ‘false’ results). Once the script has done it’s thing we will set gotkey to 0 (false) and the while will shut down next time it tries to run.

So while gotkey is 1 the loop is checking two things. First it's looking to see if the Player has stepped into a proximity called enemies1_go. If he has then it will trigger some speech (set up in the level's master script file). Next it will trigger 3 spawn managers that each create one or more enemies to attack the Player. Finally it will deactivate the proximity called enemies1_go because it's done its job and is no longer needed (you don't need to do this but it's good practice). Each of these things occurs after a delay.

The other thing that the loop is checking is if the Player's health has dropped below 40. It does this using the reference 'Health' from the object 'PlayerCharacter'. Don't forget you need to get programmers to set up any references you need.

When the Player health does drop below 40 then the script will send a 'deactivate' message to one of the spawn managers. This means that when the Player steps into the proximity only two spawners will go off instead of three, making the fight easier. This works because objects have to be 'active' to receive 'trigger' messages. If an object is 'deactive' then it will completely ignore all 'trigger' messages.

Another thing the loop does when it goes off is to dec (decrement) the var called gotkey by 1. As gotkey was 1 (true) it now drops to 0 (false) and that means that when the loop starts up again it will find that gotkey is false and will shut down.

Note that the loop includes the pause command so the rest of the game gets a look in.

Example 2:
Triggering some speech if the Player is getting on with things too slowly, but not triggering it if they're on schedule.

```
standard variables:
    slowhacker
saved variables:
    /* saved variables go here (separated by commas) */
usage:
    /* script code goes here */

set slowhacker to 1

ScriptTimerStart( 4 )

while slowhacker do

    is ScriptTimerCheck( 4,  60000 )?

        yes:
            SendTriggerMessage( mission_start, mission_speech, 2, 0 )
            ScriptTimerReset( 4 )
            dec( slowhacker, 1 )

        no:
            // Do nothing.
    end

pause( 0 )

end
```

First of all we set up a var called slowhacker and then set it to 1 (true). Next we start one of the level's four timers going (timer number 4 arbitrarily). Next we use a while loop to check if timer 4 has reached 60 seconds. If it has then we do several things. First play some speech (from bank 2). Next reset the timer (so we can use it again later), and finally decrement the var slowhacker by 1 (taking it to 0 and meaning that the while loop won't run next time).

What this script does is to play some speech saying something like "Hurry up! You're going too slowly!" when an amount of time has passed. We could do this much more easily by simply using the SendTriggerMessage command with a suitable delay. However, this will mean that the speech will always play at the specified time and there's no way to stop it once the command is sent (don't forget the command is sent immediately – it just doesn't actually play the speech until the delay has passed). The result is that you can't stop the speech from playing even if the Player's doing really well and the speech is no longer relevant.

By using the script you can simply use the command 'dec(slowhacker, 1)' anywhere in your script and it will turn the while loop off before it can say the line. That way you can shut the whole thing down if the Player is doing well.

TROUBLESHOOTING.

The following section will list some of the common reasons why a script won't compile / execute / etc.

- **The script doesn't appear to execute and the debugger complains about an ASB file not being present.**

If you're using script files in your level, you have to put two lines in the params section of the .tbs file you use to build the level. The lines are something like this...

PS2_SCRIPT_DIR C:\resBuilt\ps2\<level directory>
XB_SCRIPT_DIR C:\resBuilt\xb\<level directory>

where <level directory> is the name of the directory that the level is named after e.g nudojo

Basically aconv needs to know where to place the compiled script files (a file called lscripts.all) and will output a compile failure if it can't find these lines.

Note that you *might* also need to add the line below down at the bottom of the file. It goes after the commented out bit about copying to the Xbox, but before the several XBCopy lines.

COMPILE_SCRIPTS

- **When I build my level the converter crashes right at the end. The output of the converter says that it failed to build the ASB file.**

This error crops up if you are using a Max object with a number at the start of its name and this name appears in the script. So you could call something second_door but calling it 2nd_door will crash the compiler when it tries to build the ASB file.

- **I keep making changes to my script file but when I run the game nothing has changed at all.**

Sometimes the ASB file (which is a binary version of your script compiled when you build the level) can become kind of 'read only'. This means that it starts ignoring your changes from then on. Just delete the ASB file and do a build to rebuild it from scratch. The ASB is in the same folder where you save the ASL too.

Back to start of the document.