NWN2 Toolset Guide

Volume III – Scripting Appendices

Edition 6.1

If you find any corrections or have suggestions, edits and additions that would improve this document, I would enjoy hearing from you. Please send me a friendly e-mail. Thank you for taking the time to look through this work, and I hope you find it of some use.

—Bob Hall

September 1, 2013

Table of Contents

System Functions.

These are the built-in function calls that can be used in scripts without the need for an include file. For convenience, they have been organized into groups of common functionality, then listed in alphabetic order within each group. The complete library of functions is listed in the Script Assist panel of the script editor. In some cases questions or comments have been added in italics to indicate information that is unclear.[1]

Many of the arguments passed to these functions (or returned by the functions) are in the form of global constants, as listed under the Globals tab of the Script Assist panel. These are indicated by upper case names, such as 'ACTION_REST'. Where there are multiple such constants, a prefix is listed followed by an ellipsis. For example, ACTION_.... Some functions are used in pairs to iterate through a list of matching structure types: the GetFirst...() and GetNext...() style routines.

Actions

These calls are used to generate *action* states. By default, an action is assigned to the calling object. However, they can also be assigned to a different object, such as a creature, by passing the action command as an argument to an ActionDoCommand, AssignCommand or DelayCommand call. (See the Commands section.) Actions are also passed to some effect commands, such as EffectDispelMagicAll or EffectOnDispel. See the "ginc_wp" include file for examples of assigning actions to creatures.

1 More information on the following calls is needed than is provided in the function notes:

- GetBicFileName
- GetPlaceableIllumination
- GetSelectedMapPointTag
- LoadGlobalVariables
- PlayVoiceChat
- SaveGlobalVariables
- SetPlaceableIllumination

```
void ActionRest(
          int bIgnoreNoRestFlag = 0 )
```

This action will cause the creature to rest unless the area properties has the 'No Resting Allowed' property set to false. If *bIgnoreNoRestFlag* is set to true, the creature will rest regardless of the area properties, even if there are hostile creatures nearby. However, the creature will not rest if it is in combat.

```
void ActionWait( float fSeconds )
```

This will cause the subject to remain inactive for *fSeconds* seconds. The ActionWait command can be useful for adding an activity pause when the subject is performing fire-and-forget animations. These actions normally have an animation identifier larger than 100, although the ANIMATION_LOOPING_LISTEN animation also requires a wait.

Conversation

```
void ActionPauseConversation()
```

This pauses the current conversation. The conversation will be resumed when ActionResumeConversation is called.

```
void ActionPauseCutscene(
          int nTimeout,
          int bPurgeOnTimeout = FALSE )
```

This action pauses the current conversation cutscene. The *nTimeout* is the number of milliseconds before the cutscene automatically resumes, which also causes the pending cutscene actions to clear. If *nPurgeOnTimeout* is true, all pending cutscene actions are cleared.

This differs from ActionPauseConversation in that the conversation will resume once a cutscene action has been assigned, if there are no cutscene actions currently pending. Thus a call to ActionResumeConversation is unnecessary.

```
void ActionResumeConversation()
```

This will resume a conversation that was paused by a call to ActionPauseConversation.

```
void ActionSpeakString(
          string sStringToSpeak,
          int nVolume = TALKVOLUME_TALK )
```

The creature will speak the string *sStringToSpeak* at the volume *nVolume*. Valid values for the volume are defined by the TALKVOLUME_... constants.

```
void ActionSpeakStringByStrRef(
          int nStrRef,
          int nVolume = TALKVOLUME_TALK )
```

This will look up a string by reference *nStrRef* from the talk table then speak it as per ActionSpeakString at the volume *nVolume*. Valid values for the volume are defined by the TALKVOLUME_... constants.

```
void ActionStartConversation(
          object oObjectToConverseWith,
          string sDialogResRef = "",
          int bPrivateConversation = FALSE,
          int bPlayHello = TRUE,
          int bIgnoreStartDistance = FALSE,
          int bDisableCutsceneBars = FALSE )
```

Initiate a conversation with *oObjectToConverseWith*. This defaults to the creature's own conversation unless *sDialogResRef* is not empty and matches a conversation resource reference. If *bPlayHello* is false, the creature's greeting will not be played.

Interact

```
action ActionAttack(
          object oAttackee,
          int bPassive = FALSE )
```

This call causes the action subject to attack *oAttackee*. If *bPassive* is true, the attack is in passive mode. This means the subject will not move to attack *oAtackee* with a melee weapon.

```
void ActionCloseDoor(
          object oDoor )
```

The action subject closes the door object *oDoor*. See also ActionOpenDoor.

```
void ActionExamine( object oExamine )
```

This causes the creature to examine the object *oExamine*. When the action is assigned to the player's character, this will cause the Examination window to appear.

```
void ActionInteractObject(
          object oPlaceable )
```

This command will cause the subject to interact with the placeable *oPlaceable*. The net effect is that the subject runs up to the object and then stops.

```
void ActionLockObject( object oTarget )
```

If *oTarget* is a door or container, the subject will lock it.

```
void ActionOpenDoor(
          object oDoor )
```

The action subject opens the door object *oDoor*. See also ActionCloseDoor.

```
void ActionSit( object oChair )
```

If a creature is capable of sitting, it should seat itself in the chair *oChair*. Unfortunately this function does not work properly, so the target of this action will not sit down for more than a fraction of a second. What you can use instead is a set of custom looping animation files in combination with the PlayCustomAnimation call. These animation files are:

- "sitdrink"
- "sitdrinkidle"
- "sitfidget"
- "sitidle"
- "sittalk01"
- "sittalk02"

```
void ActionUnlockObject(
          object oTarget )
```

If *oTarget* is a door or a placeable container, the subject will unlock it.

Movement

```
void ActionPlayAnimation(
          int nAnimation,
          float fSpeed = 1.0,
          float fDuration = 0.0 )
```

This will cause the subject to move their body according to the animation *nAnimation*. Valid animations are specified by the ANIMATION_... constants, which are subdivided into fire-and-forget and looping-type animations. The speed of the animation is multiplied by *fSpeed*, and the duration of a looping animation is fDuration seconds. The fire-and-forget animations ignore the *fDuration* parameter.

Note that the PC races have most of the animations implemented, but not all creatures will perform all animations. The Wolf, for example, will only run the collapse, dodge duck, dodge side, standup and taunt fire-and-forget animations, as well as dead-back and dead-front looping animations.

The ANIMATION_LOOPING_SIT_CHAIR animation will only produce a brief crouching motion, rather than a seated position.

```
void ActionForceFollowObject(
          object oFollow,
          float fFollowDistance = 0.5f,
          int iFollowPosition = 0 )
```

When this action is activated, the subject will follow the creature *oFollow* until ClearAllActions() is executed. (See Commands.) The *fFollowDistance* is the distance to follow in meters, which defaults to 0.5f if less than 0.5f. The command notes indicate *iFollowPosition* is an offset position to run to behind the creature being followed, but I haven't noticed any effect from how this is set.

```
void ActionForceMoveToLocation(
          location locDestination,
          int bRun = FALSE,
          float fTimeout = 30.0f )
```

The subject walks to the location *locDestination*. If *bRun* is true, the subject will run instead. Note that if the path to the location is blocked, the subject may do nothing. You can address this by adding a DelayCommand of an ActionJump that occurs after the *fTimeout* period.

```
void ActionForceMoveToLocation(
          object oDestination,
          int bRun = FALSE,
          float fRange = 1.0f,
          float fTimeout = 30.0f )
```

The subject walks toward the object *oDestination* until it comes within *fRange* distance. If *bRun* is true, the subject will run instead. Note that if the path to the object is blocked, the subject may do nothing.

```
void ActionJumpToLocation(
          location locJumpTo )
```

The subject is instantly moved to the location *locJumpTo*, even if the location is in another area. This can be a bit jarring, so it is better used out of the player's sight.

```
void ActionJumpToObject(
          object oJumpTo,
          int bWalkStraightLine = TRUE )
```

The subject is instantly moved adjacent to the object o*JumpTo*. It is unclear what *bWalkStraightLine* is used for or why it is needed.

```
void ActionMoveAwayFromLocation(
          location locMoveAwayFrom,
          int bRun = FALSE,
          float fMoveAwayRange = 40.0f )
```

This command will cause the subject to move as far as possible away from the location *locMoveAwayFrom*, until a distance of *fMoveAwayRange* has been reached. The subject will stop once no further separation can be attained. If *bRun* is true, the subject will run.

```
void ActionMoveAwayFromObject(
          object oMoveAwayFrom,
          int bRun = FALSE,
          float fMoveAwayRange = 40.0f )
```

This command will cause the subject to move as far as possible away from the object *oMoveAwayFrom*, until a distance of *fMoveAwayRange* has been reached. This works even if there is no clear path between the subject and the object. The subject will stop once no further separation can be attained. If *bRun* is true, the subject will run.

```
void ActionRandomWalk()
```

This is a persistent action that causes the subject to generate a nearby random location, find a path there and move to it, then repeat. The ClearAllActions action must be called before any other actions will be executed.

Talents

```
action ActionCastFakeSpellAtLocation(
          int nSpell,
          location locTarget,
          int nProjectilePathType =
            PROJECTILE_PATH_TYPE DEFAULT )
```

The subject produces the conjuration and cast animations for the spell *nSpell* cast at location *locTarget* with projectile path type *nProjectilePath*. However, there is no other effect. The spell ID is one of the SPELL_... constants.

```
action ActionCastFakeSpellAtObject(
          int nSpell,
          object oTarget,
          int nProjectilePathType =
            PROJECTILE_PATH_TYPE_DEFAULT )
```

This is similar to ActionCastFakeSpellAtLocation, except the target is the object *oTarget*.

```
action ActionCastSpellAtLocation(
          int nSpell,
          int locTarget,
          int nMetamagic = METAMAGIC_ANY,
          int bCheat = FALSE,
          int nProjectilePathType =
            PROJECTILE_PATH_TYPE_DEFAULT,
          int bInstantSpell = FALSE )
```

This subroutine causes the spell *nSpell* to be cast at location *locTarget*. The spell type, *nSpell*, is given by one of the SPELL_... global constants. The *nMetaMagic* parameter is set to a METAMAGIC_... global constant value. Use -1 for METAMAGIC_ANY or 0 for METAMAGIC_NONE. If *bCheat* is true, then the caster does not necessarily need to be able to cast the spell. The *nDomainLevel* parameter defines the caster level. The *nProjectilePathType* parameter is set to one of the PROJECTILE_ PATH_TYPE_... global constants, or to PROJECTILE_ PATH_TYPE_DEFAULT if the value is zero. If *bInstantSpell* is true, cast the spell immediately.

```
action ActionCastSpellAtObject(
          int nSpell,
          int locTarget,
          int nMetamagic = METAMAGIC_ANY,
          int bCheat = FALSE,
          int nDomainLevel,
          int nProjectilePathType =
            PROJECTILE_PATH_TYPE_DEFAULT,
          int bInstantSpell = FALSE )
```

This call functions like ActionCastSpellAtLocation, except that the target is the object *oTarget*. The *nDomainLevel* parameter is undefined. This function is called by the ga_cast_spell_at_object script.

```
void ActionCounterSpell( object oCSTarget )
```

This causes the action subject to counter-spell the target *oCSTarget*. That is, the subject will attempt to use a spell to negate a matching spell being cast by the target. In order for this to succeed, the subject must be able to cast the spell being countered, must have the spell prepared, and must recognize the spell using a spellcraft skill check. This counts as a ready action, and it is activated at the start of the opponent's round. Any actions assigned to the creature following this command will negate the counterspell action.

```
void ActionUseFeat(
          int nFeat,
          object oTarget )
```

The subject uses the feat *nFeat* on the target *oTarget*. The feat is one of the FEAT... global constants. Feats in NWN2 can include class special abilities. See the Feats section.

```
int ActionUseSkill(
          int nSkill,
          object oTarget,
          int nSubSkill = 0,
          object oItemUsed =
                    OBJECT_INVALID )
```

The subject will use the skill *nSkill* on the target *oTarget*. Valid values of *nSkill* are SKILL_.... The possible values of *nSubSkill* are SUBSKILL_...TRAP, which apply to the disable trap skill. The *oItemUsed* can be passed an item object for use with the skill, such as a healer's kit for the healing skill. This routine returns true if the action was queued successfully.

```
void ActionUseTalentAtLocation(
          talent tChosenTalent,
          location locTarget )
```

The subject uses the talent *tChosenTalent* at the location *locTalent*. Talents can be feats, skills or spells.

```
void ActionUseTalentOnObject(
          talent tChosenTalent,
          object oTarget )
```

The subject will use the talent *tChosenTalent* on the target *oTarget*.

Combat

See also GetActionMode, GetLastAttacker, GetLastKiller and the Encounter section.

```
void ClearCombatOverrides(
          object oCreature )
```

This routine will clear any combat overrides that were established with the SetCombatOverrides call on the creature *oCreature*.

```
void DoWhirlwindAttack(
          int bDisplayFeedback = TRUE,
          int bImproved = TRUE )
```

The calling object performs a whirlwind attack against all enemies within a 10 ft. radius. If *bDisplayFeedback* is true, then feedback is printed in the chat window. If *bImproved* is true, the attack uses the improved version of a whirlwind attack.

This call is used solely in the spell script for the whirlwind attack. If you want to queue up a whirlwind attack action, the notes state you should use the ActionUseFeat action routine with the FEAT_WHIRLWIND_ATTACK.

```
object GetAttackTarget(
          object oCreature = OBJECT_SELF )
```

This function returns the object currently being attacked by the creature *oCreature*. If the creature is not in combat, this will return an invalid object.

```
object GetAttemptedAttackTarget()
```

Each combatant tracks the target of each attack, and this routine will return the object that was attacked. When the caller is not in combat, this will return an invalid object.

```
object GetAttemptedSpellTarget()
```

Each combatant tracks the target of each spell, and this routine will return the object that was targeted. When the caller is not in combat, this will return an invalid object.

```
int GetBaseAttackBonus( object oCreature )
```

This routine returns the current base attack bonus of the creature *oCreature*. See GetTRUEBaseAttackBonus.

```
int GetCurrentHitPoints(
          object oObject = OBJECT_SELF )
```

Fetch the current number of hit points for the object *oObject*. See GetMaxHitPoints.

```
int GetDamageDealtByType(
          int nDamageType )
```

This determines the amount of hit point damage of type *nDamageType* that has been dealt to the object calling this routine. The type is a DAMAGE_TYPE_... constant, which includes acid, bludgeoning, divine, and so forth.

```
object GetGoingToBeAttackedBy(
          object oTarget )
```

This specialized routine is intended for a henchman to process a “going to be attacked” shout from its master. It returns the object that is going to attack the target *oTarget*. This returns an invalid object if *oTarget* is not a valid creature or if combat has ended.

```
int GetIsCreatureDisarmable(
          object oCreature )
```

This returns true if the creature oCreature has the Disarmable property set to true. This would allow the use of a disarm action against the creature. (*Unfortunately, most of the default blueprints for weapon-wielding creatures have this set to false.*)

```
int GetIsInCombat(
          object oCreature = OBJECT_SELF )
```

If the creature oCreature is currently engaged in combat, this will return true.

```
int GetIsWeaponEffective(
          object oVersus = OBJECT_INVALID,
          int bOffHand = FALSE )
```

This returns true if the weapon equipped by the calling object is capable of inflicting damage against *oVersus*. If *bOffHand* is true, this will check the weapon in the off-hand slot instead.

```
int GetLastAttackMode(
          object oCreature = OBJECT_SELF )
```

This call returns the attack mode of the last attack by creature *oCreature*. The result is a COMBAT_MODE_... constant. If the creature is not in combat, this will return an invalid mode.

```
int GetLastAttackType(
          object oCreature = OBJECT_SELF )
```

When the object *oCreature* is in combat, this will return the type of the object's last attack. This is a global constant of the form SPECIAL_ATTACK_....

```
object GetLastDamager(
          object oObject = OBJECT_SELF )
```

This returns the last object that inflicted damage upon *oObject*.

```
object GetLastHostileActor(
          object oVictim = OBJECT_SELF )
```

This call returns the last object that committed a 'hostile' act against *oVictim*. This is the last object that was stored as a return value for GetLastAttacker, GetLastDamager, GetLastSpellCaster (when GetLastSpellHarmful was true), or GetLastDisturbed (as the result of a pick pocket attempt).

```
object GetLastWeaponUsed(
          object oCreature )
```

This returns the weapon that the creature *oCreature* used in an attack.

```
object GetPlayerCurrentTarget(
          object oCreature )
```

For a valid, player-controlled creature *oCreature*, this will return the object that the player currently has selected as the target of the PC.

```
float GetProjectileTravelTime(
          location locSource,
          location locTarget,
          int nProjectilePathType,
          int nSpellID = -1 )
```

This will return the time in seconds for a projectile to travel from *locSource* to *locTarget*. The projectile path type *nProjectilePathType* is a PROJECTILE_PATH_TYPE_... constant. If PROJECTILE_PATH_TYPE_SPELL is given and *nSpellID* is a valid spell identifier, SPELL_..., this routine will retrieve the spell's projectile path type from the 'spell.2da' file.

```
int GetTotalDamageDealt()
```

This returns the total hit points of damage that have been dealt to the object calling this routine.

```
int GetTRUEBaseAttackBonus( object oTarget )
```

This returns the base attack bonus of the object *oTarget* before any modifiers are applied.

```
void RestoreBaseAttackBonus(
          object oCreature = OBJECT_SELF )
```

This will restore the base attack bonus to its original level, thereby also restoring the number of attacks.

```
void SetCombatOverrides(
          object oCreature,
          object oTarget,
          int nOnHandAttacks,
          int nOffHandAttacks,
          int nMinDamage,
          int nMaxDamage,
          int bSuppressBroadcastAOO,
          int bSuppressMakeAOO,
          int bIgnoreTargetReaction,
          int bSuppressFeedbackText )
```

This routine is normally used during a cutscene combat round to cause a specific result by applying overrides on the creature *oCreature* attacking the object *oTarget*. The target can be invalid, in which case normal target selection will occur. The *nAttackResult* is a global constant of the form OVERRIDE_ATTACK_RESULT_..., that forces the attack outcome.

- The *nOnHandAttacks* and *nOffHandAttacks* sets the number of attacks per round. Setting these to -1 will use the default attacks for the round. Setting the total to a value between one and six (with each 0 to 6) will force that many attacks.
- The *nMinDamage* and *nMaxDamage* sets the random damage range, or set them to -1 for the default.
- If *bSuppressBroadcastAOO* is true, the creature may may cause an attack of opportunity from nearby creatures.
- If *bSuppressMakeAOO* is true, the creature will make attacks of opportunity when the circumstances allow.
- If *bIgnoreTargetReaction* is true, ActionAttack calls on hostile creatures are not rejected.
- If *bSuppressFeedbackText* is true, the creature's combat feedback will not be displayed.

```
void SpawnBloodHit(
          object oCreature,
          int bCriticalHit,
          object oAttacker )
```

This creates a visual effect that simulates blood emission from a combat hit. The blood will be of the type from the creature *oCreature*. If *bCriticalHit*, the blood will be from a critical hit, rather than a normal hit. The trajectory of the particles is determined by the placement of the attacker *oAttacker*.

```
void SpawnItemProjectile(
          object oSource,
          object oTarget,
          location locSource,
          location locTarget,
          int nBaseItemID,
          int nProjectilePathType,
          int nAttackType,
          int nDamageTypeFlag )
```

This creates the visual effect for a projectile. The source of the projectile *oSource* is at the location *locSource*. The target of the projectile *oTarget* is at location *locTarget*. The type of launcher *nBaseItemID* is a row in the 'baseitems.2da' file, which uses the AmmunitionType column to get the ammo type. (These match the BASE_ITEM_... constants.)

The *nProjectilePathType* is a global constant of the form PROJECTILE_PATH_TYPE_... that determines the path type of the projectile. The *nAttackType* parameter is a constant OVERRIDE_ATTACK_RESULT_... that is based on the attack result. The *nDamageTypeFlag* can be used to add a effect DAMAGE_TYPE_... that can be acid, cold, electrical, fire, divine or sonic.

```
int TouchAttackMelee(
          object oTarget,
          int bDisplayFeedback = TRUE,
          int nBonus = 0 )
```

The object calling this routine performs a touch melee attack on the target *oTarget* with a bonus *nBonus* to the attack roll. (This does not check whether the attacker is in range of the target.) It returns a global constant TOUCH_ATTACK_RESULT_... that gives the result of the attack. The result is printed to the player's chat window if the *bDisplayFeedback* parameter is true.

```
int TouchAttackRanged(
          object oTarget,
          int bDisplayFeedback = TRUE,
          int nBonus = 0 )
```

The object calling this routine performs a ranged touch attack on the target *oTarget* with a bonus *nBonus* to the attack roll. It returns a TOUCH_ATTACK_RESULT_... constant that gives the result of the attack. If the *bDisplayFeedback* parameter is true, the result is printed to the player's chat window.

Encounter

These routines apply to encounter trigger areas. See also the Combat section.

```
int GetEncounterActive(
          object oEncounter = OBJECT_SELF )
```

If *oEncounter* is a valid encounter object, this will return true of the encounter's Active property is set to true.

```
int GetEncounterDifficulty(
          object oEncounter = OBJECT_SELF )
```

If *oEncounter* is a valid encounter, this will return the difficulty level of the encounter as a global constant of the type ENCOUNTER_DIFFICULTY_.... The available types are very easy, easy, normal, hard and impossible.

```
int GetEncounterSpawnsCurrent(
          object oEncounter = OBJECT_SELF )
```

This will return the number of times that the encounter *oEncounter* has spawned. See GetEncounterSpawnsMax.

```
int GetEncounterSpawnsMax(
          object oEncounter = OBJECT_SELF )
```

This will return the maximum number of times that the encounter *oEncounter* can spawn.

```
int GetIsEncounterCreature(
          object oEncounter = OBJECT_SELF )
```

For a valid creature *oCreature*, this will return true if the creature was spawned by an encounter.

```
void SetEncounterActive(
          int bNewValue,
          object oEncounter = OBJECT_SELF )
```

This routine will control whether the encounter object *oEncounter* is active or note, based on the value of *bNewValue*.

```
void SetEncounterDifficulty(
          int nEncounterDifficulty,
          object oEncounter = OBJECT_SELF )
```

This call will set the difficulty ranking of the encounter *oEncounter* to *nEncounterDifficulty*, which is a global constant ENCOUNTER_DIFFICULTY_....

```
void SetEncounterSpawnsCurrent(
          int nNewValue,
          object oEncounter = OBJECT_SELF )
```

This will override the value that tracks the number of times that the encounter *oEncounter* has been triggered, and set it to *nNewValue*.

```
void SetEncounterSpawnsMax(
          int nNewValue,
          object oEncounter = OBJECT_SELF )
```

This will change the maximum the number of times that the encounter *oEncounter* will spawn to *nNewValue*.

```
void TriggerEncounter(
          object oEncounter,
          object oPlayer,
          int nCRFlag,
          float fCR )
```

This will cause the encounter *oEncounter* to be triggered by player *oPlayer*. The *nCRFlag* parameter is not implemented. The value of *fCR* will set the challenge rating of the encounter. A value of -1.0 will cause the function to compute the challenge rating based on nearby players.

Creatures

See also the Commands and Interaction sections.

```
string RandomName()
```

This generates a random character name.

Management

See also SetSoundSet.

```
void ForceRest( object oCreature )
```

The creature *oCreature* is forced to rest, thereby restoring it's hit points and spells, resetting it's feats, and so forth.

```
void GiveGoldToCreature(
          object oCreature,
          int nGP,
          int bDisplayFeedback = TRUE )
```

This call gives *nGP* gold pieces to the creature *oCreature*. If *bDisplayFeedback* is true, this will display feedback in the player's chat window.

```
void SetAILevel(
          object oTarget,
          int nAILevel )
```

This call will set the artificial intelligence level of an NPC oTarget to nAILevel, which is a AI_LEVEL_... constant. See GetAILevel.

```
void SetCreatureAppearanceType(
          object oCreature,
          int nAppearanceType )
```

This changes the appearance of the creature to *oCreature* to the type *nAppearanceType*, which can be any of the global constants APPEARANCE_TYPE_....

```
void SetDeity(
          object oCreature,
          string sDeity )
```

This sets the deity of the creature *oCreature* to the name *sDeity*. The deities used in the NWN2 campaign are listed in the 'nwn2_deities.2da' file.

```
void SetIsDestroyable(
          int bDestroyable,
          int bRaisable = TRUE,
          int bSelectableWhenDead = FALSE )
```

This call sets a flag that determines whether the calling object (OBJECT_SELF) is destroyed when slain. If the *bDestroyable* flag is false, then the corpse will remain behind after the caller is slain. Otherwise it will fade away.

If *bRaisable* is true, then the caller can be brought back to life via resurrection. If *bSelectableWhenDead* is true, then the caller is selectable when dead, allowing the description to be viewed by a player and so forth.

```
void SetImmortal(
          object oCreature,
          int bImmortal )
```

This will set the Immortal property on the creature *oCreature* to the boolean *bImmortal*. When a creature is immortal, it can be damaged but not slain. See GetImmortal and SetPlotFlag.

```
void SetLootable(
          object oCreature,
          int bLootable )
```

If *oCreature* is a living NPC, this will set the state of the Lootable property to the boolean *bLootable*.

```
void SetMovementRateFactor(
          object oCreature,
          float fFactor )
```

The movement factor of creature *oCreature* is changed to *fFactor*, and applied to the creature's speed. This factor is also modified by selected effects and encumbrance.

```
void SetSubRace(
          object oCreature,
          string sSubRace )
```

This sets the name of the creature *oCreature*'s subrace to *sSubRace*.

```
void TakeGoldFromCreature(
          int nGP,
          object oCreature,
          int bDisplayFeedback = TRUE )
```

This call removes *nGP* gold pieces from the creature *oCreature*. The gold is destroyed. If *bDisplayFeedback* is true, this will display feedback in the player's chat window.

Alignment

A character's alignment rating is measured along two scales: good-evil and law-chaos. These are integer values from 0 to 100, with each ALIGNMENT_... name matching a range along the scale. (See the notes on ginc_alignment for more details.) The alignment rating of a character can be modified as a result of behavior within the game. The current alignment determines how certain item properties or spells will impact a character.

```
void AdjustAlignment(
          object oSubject,
          int nAlignment,
          int nShift )
```

The alignment of *oSubject* is modified by an amount determined by the amount *nShift* in the direction specified by *nAlignment*. Valid values for nAlignment are the ALIGNMENT_... constants. For LAWFUL, CHAOTIC, GOOD or EVIL, the shift will be along a single axis in the direction specified. For ALIGNMENT_ALL, both the GOOD/EVIL and LAWFUL/CHAOTIC values are shifted by nShift. If ALIGNMENT_NEUTRAL is used, then the shift is toward true neutral.

See also the routines in the ginc_alignment include file.

```
int GetAlignmentGoodEvil(
          object oCreature )
```

This call returns a value that gives the general ethical alignment of the creature *oCreature* along the good-evil axis. The result matches an ALIGNMENT_... constant, or -1 if the creature is invalid. See GetGoodEvilValue.

```
int GetAlignmentLawChaos(
          object oCreature )
```

This call returns a value that gives the general ethical alignment of the creature *oCreature* along the lawful-chaotic axis. The result matches an ALIGNMENT_... constant, or -1 if the creature is invalid. See the GetLawChaosValue routine.

```
int  GetGoodEvilValue( object oAligned )
```

Returns a numerical value giving the current ethical alignment of the *oAligned* object along the good/evil axis. This function returns a value between 0 and 100, with 100 if the object is at the extreme end of goodness or 0 if the object is extremely evil.

```
int  GetLawChaosValue( object oAligned )
```

Returns a numerical value giving the current ethical alignment of the *oAligned* object along the law/chaos axis. This function returns a value between 0 and 100, with 100 if the object is very lawful or 0 if the object is very chaotic.

Associates

Each creature belonging to the party of a PC has an associate type. Each is either an animal companion, dominated, familiar, henchman or a summoned creature. Characters with druid or ranger classes can have animal companions, while those with a sorcerer or wizard class can have a familiar. These are creatures that share a special bond with the character. See also the Factions section.

```
void AddHenchman(
          object oMaster,
          object oHenchman = OBJECT_SELF )
```

If *oHenchman* is an NPC, make character a henchman to *oMaster*. If the addition would exceed the maximum number of henchmen, there is no effect.

```
int GetAnimalCompanionCreatureType(
          object oCreature )
```

If the creature *oCreature* has an animal companion, this returns a ANIMAL_COMPANION_CREATURE_TYPE_... constant giving the type.

```
string GetAnimalCompanionName(
          object oCreature )
```

If the creature *oCreature* has an animal companion, this will return a string containing the name of the companion. A null string is returned when there is no companion.

```
object GetAssociate(
          int nAssociateType,
          object oMaster = OBJECT_SELF,
          int nTh = 1 )
```

This routine will return the *nTh* associate of type *nAssociateType* belonging to *oMaster*, where the associate type is an ASSOCIATE_TYPE_... constant.

```
int GetAssociateType( object oAssociate )
```

If the creature *oAssociate* is an associate of another creature, this will return an ASSOCIATE_TYPE_... constant, or ASSOCIATE_TYPE_NONE if the creature is not an associate.

```
object GetControlledCharacter(
          object oCreature )
```

The notes for this command are unclear.

```
int GetFamiliarCreatureType(
          object oCreature )
```

If the creature *oCreature* has a familiar, this returns a constant FAMILIAR_CREATURE_TYPE_... giving the type.

```
string GetFamiliarCompanionName(
          object oCreature )
```

If the creature *oCreature* has a familiar, this will return a string containing the name of the creature. A null string is returned when there is no familiar.

```
object GetHenchman(
          object oMaster = OBJECT_SELF,
          int nNth = 1 )
```

Use this routine to return the *nNth* henchman belonging to *oMaster*. If the *nNth* henchman does not exist, this returns an invalid object.

```
int GetIsCompanionPossessionBlocked(
          object oCreature )
```

This call returns the state of the possession flag that is set by SetIsCompanionPossessionBlocked, for the creature *oCreature*.

```
int GetIsPossessedFamiliar(
          object oCreature )
```

If the creature *oCreature* is a familiar that is currently possessed by it's master, then this routine will return true.

```
object GetMaster( object oAssociate )
```

If *oAssociate* is an associate, this will return the object representing the creature's master.

```
int GetMaxHenchmen()
```

This function returns the maximum number of henchmen allowed. Henchmen are not the same as companions. See SetMaxHenchmen.

```
void RemoveHenchman(
          object oMaster,
          object oHenchman = OBJECT_SELF )
```

If oHenchman is a henchman in the service of oMaster, this will remove the henchman from the master's faction and restore its original faction.

```
void RemoveSummonedAssociate(
          object oMaster,
          object oAssicoate = OBJECT_SELF )
```

This will remove the associate *oAssociate* from the service of the master *mMaster*, restoring the associate to its original faction.

```
void SetIsCompanionPossessionBlocked(
          object oCreature,
          int bBlocked )
```

If the creature *oCreature* is not a player-owned character, setting the boolean *bBlocked* to true will allow a player to possess the creature as a companion.

```
object SetOwnersControlledCompanion(
          object oCurrentCreature,
          object oTargetCreature =
            OBJECT_INVALID )
```

This routine changes the creature controlled by a player from *oCurrentCreature* to *oTargetCreature*. If the creature *oTargetCreature* is not found, then the player is set to control their original, owned creature.

```
void SetMaxHenchmen( int nNumHenchmen )
```

This function sets the maximum number of henchmen allowed to *nNumHenchmen*. See GetMaxHenchmen.

```
void SummonAnimalCompanion(
          object oMaster = OBJECT_SELF )
```

If the creature *oMaster* has an animal companion, this routine will summon it into the game.

```
void SummonFamiliar(
          object oMaster = OBJECT_SELF )
```

If the creature *oMaster* has a familiar, this routine will summon it into the game.

```
void UnpossessFamiliar(
          object oCreature )
```

The player creature *oCreature* is made to unpossess it's familiar, if any.

Class Levels and Experience Points

These are routines related to character classes, class packages, gaining experience points and levelling up. See also GetFactionAverageXP and SetModuleXPScale.

```
float GetChallengeRating( object oCreature )
```

If *oCreature* is a valid creature, this routine will return the creature's challenge rating as a floating point value. If the input argument is not a valid creature, this will return 0.0.

```
int GetClassByPosition(
          int nClassPosition,
          object oCreature = OBJECT_SELF )
```

A creature can have as many as three different classes, and these are slotted in positions 1 through 3. This call returns the class type in position *nClassPosition* (1, 2 or 3) of the creature *oCreature*. The result matches a CLASS_TYPE_... constant, or CLASS_TYPE_INVALID if there is no class in that position.

```
int GetCreatureStartingPackage(
          object oCreature )
```

This returns a PACKAGE_... constant that gives the default package selected for the creature *oCreature* to use when levelling up.

```
int GetHitDice( object oCreature )
```

Return the number of hit dice of *oCreature*. If the object is invalid, return 0.

```
int GetLevelByClass(
          int nClassType,
          object oCreature = OBJECT_SELF )
```

Return the number of levels the creature *oCreature* has in the class *nClassType*, which is a CLASS_TYPE_... constant matching the various standard classes in 'classes.2da'.

```
int GetLevelByPosition(
          int nClassPosition,
          object oCreature = OBJECT_SELF )
```

A creature can have as many as three different classes, and these are slotted in positions 1 through 3. This call returns the number of class levels in position *nClassPosition* (1, 2 or 3) of the creature *oCreature*.

```
int GetLevelUpPackage(
          object oCreature )
```

This returns the level-up package for the creature *oCreature*. The result is a row in the 'packages.2da' file. See GetTotalLevels.

```
int GetMaxHitPoints(
          object oObject = OBJECT_SELF )
```

This routine will return the maximum allotted hit points of the object *oObject*.

```
int GetTotalLevels(
          object oCreature,
          int bIncludeNegativeLevels )
```

This routine returns the total number of levels across all three class positions for the creature *oCreature*. If the *bIncludeNegativeLevels* parameter is true, the result will also tally any negative levels, such as from the EffectNegativeLevel effect. See GetHitDice.

```
int GetXP( object oCreature )
```

This returns the total experience points of the creature *oCreature*. See SetXP and GiveXPToCreature.

```
void GiveXPToCreature(
          object oCreature,
          int nXpAmount )
```

This will award *nXpAmount* experience points, a positive integer, to the creature *oCreature*. See SetXP.

```
int LevelUpHenchman(
          object oCreature,
          int nClass = CLASS_TYPE_INVALID,
          int bReadyAllSpells = FALSE,
          int nPackage = PACKAGE_INVALID )
```

This causes the creature *oCreature* to increase a class level, even if it lacks the necessary experience points. It returns the new level of the creature. By default it will level up the first class, but this can be changed by setting *nClass* to a different class constant CLASS_TYPE_.... If *bReadyAllSpells* is true, then all spells will be instantly readied without resting. The *nPackage* causes the level up to use a non-default level-up package, PACKAGE_....

```
void ResetCreatureLevelForXP(
          object oTarget,
          int nExperience,
          int bUseXPMods )
```

This routine resets the level of the creature *oTarget* to zero, then sets the experience to *nExperience* and automatically levels up the creature to the new experience level. If *bUseXPMods* is true, then the creature's XP modifiers will be applied to the experience awarded before leveling up.

```
void SetLevelUpPackage(
          object oCreature,
          int nPackage )
```

This sets the level-up package for the creature *oCreature* to *nPackage*, which is a PACKAGE_... constant corresponding to a row in the 'packages.2da' file.

```
int SetUnrestrictedLevelUp(
          object oCreature )
```

Normally a package can apply restrictions to a creature levelling up, which are based upon the Starting Package parameter setting. This call will disable these restrictions for the creature *oCreature*, allowing level up in any eligible class.

```
void SetXP(
          object oCreature,
          int nXpAmount )
```

This changes the creature *oCreature*'s experience point total to *nXpAmount*.

Modes and States

```
int GetActionMode(
          object oCreature,
          int nMode )
```

A creature can have various action modes set, such as defensive casting or power attack. This will return the status of the mode *nMode* on the creature *oCreature*. The mode is an ACTION_MODE_... constant.

```
int GetAILevel(
          object oTarget = OBJECT_SELF )
```

This returns the level of the game's artificial intelligence that is being used to control the behavior of the creature *oTarget*. The result is an AI_LEVEL_... constant. See SetAILevel.

```
int GetDefensiveCastingMode(
          object oCreature )
```

This routine returns a DEFENSIVE_CASTING_MODE_... constant that gives the defensive casting mode of the creature *oCreature*. There are currently only two modes, activated and disabled.

```
int GetDetectMode(
          object oCreature )
```

This queries the creature *oCreature* and returns the current detection mode, which is a DETECT_MODE_... constant.

```
int GetEncumbranceState( object oCreature )
```

For a valid creature *oCreature*, this will return the current encumbrance state as an ENCUMBRANCE_STATE_...

constant. Valid states are normal, heavy and overloaded. See also GetWeight.

```
int GetIsDead( object oCreature )
```

Returns true if the creature *oCreature* is dead. If the creature is a PC, this function will also return true if the creature is dying.

```
int GetIsResting(
          object oCreature = OBJECT_SELF )
```

This routine will return true if a valid creature *oCreature* is currently resting.

```
object GetSittingCreature( object oChair )
```

If *oChair* is a valid placeable, this will return the creature object that is sitting upon it.

```
int GetStealthMode( object oCreature )
```

This returns a stealth mode of the creature *oCreature*. The result is a constant STEALTH_MODE_... that currently has two values: activated or disabled.

```
void SetActionMode(
          object oCreature,
          int nMode,
          int nStatus )
```

This sets the status *nStatus* of the action mode *nMode* for the creature *oCreature*. Valid modes are global constants ACTION_MODE_....

Player Characters

```
int GetIsOwnedByPlayer(
          object oCreature )
```

This returns true if the creature *oCreature* is an original player character belonging to a player. See GetIsPC.

```
int GetIsPC( object oCreature )
```

If *oCreature* is a player controlled creature, this will return true. It will return false if the player is currently controlling a different creature, even if *oCreature* is the player's original character. See GetIsOwnedByPlayer.

```
int GetIsPlayableRacialType(
          object oCreature )
```

If the creature *oCreature* is a playable racial type, then this call will return true. The playable races are dwarf, elf, gnome, halfling, half-elf, half-orc and human.

```
int GetIsPlayerCreated(
          object oCreature )
```

This returns true if the creature *oCreature* was created by a player.

```
object GetLastPCRested()
```

This call will fetch the object representing the last PC to rest in the current module.

```
object GetOwnedCharacter(
          object oControlled )
```

If the object *oControlled* is being controlled by a player, then this call will return that player's character.

Properties

See also GetCreatureHasTalent.

```
int GetAbilityModifier(
          int nAbility,
          object oCreature = OBJECT_SELF )
```

This returns the modifier for the ability *nAbility* of the creature *oCreature*. The *nAbility* is a global constant of the form ABILITY_.... The modifier is 0 for an ability score of 10 or 11, and changes by one per two points of ability score change.

```
int GetAbilityScore(
          object oCreature,
          int nAbilityType,
          int nBaseAttribute = FALSE )
```

The *nAbilityType* variable must be set to one of the ABILITY_... global constants. If *nBaseAttribute* is true, this function returns the base attribute score of the creature *oCreature*. Otherwise, it will return the modified attribute score of the creature.

The *nBaseAttribute* defaults to false if it is not set. On an error, this function returns 0.

```
int GetAC(
          object oObject,
          int nForFutureUse = 0 )
```

For a creature *oObject*, this will return the armor class. It returns zero for a door, item or placeable, and -1 otherwise. Use the GetHardness call to obtain the damage resistance of a door or placeable. At present, the *nForFutureUse* parameter is ignored.

```
int GetAge( object oCreature )
```

This returns the age setting of the creature *oCreature*. This parameter is not a configurable creature property, so this only returns a non-zero value for a PC.

```
int GetAppearanceType(
          object oCreature )
```

This returns the appearance type property of the creature *oCreature*. This is an APPEARANCE_TYPE_... constant.

```
int GetCharBackground( object oCreature )
```

This returns a constant of the form BACKGROUND_... that gives the character background of the creature *oCreature*. If the creature does not have a background, BACKGROUND_NONE is returned. Typically only a PC will have a background.

```
int GetCreatureSize( object oCreature )
```

This returns a constant CREATURE_SIZE_... that gives the size category of the creature *oCreature*. The valid values are tiny, small, medium, large, huge, or invalid. It does not include fine, diminutive, gargantuan or colossal.

```
string GetDeity( object oCreature )
```

This function returns a string containing the name of the creature's deity. If the *oCreature* object is invalid, an empty string is returned.

```
int GetGender( object oCreature )
```

This returns the gender of the creature *oCreature* as a constant of the form GENDER_....

```
int GetGold(
          object oTarget = OBJECT_SELF )
```

Return the amount of gold currently possessed by *oTarget*. If *oTarget* is not specified, it defaults to OBJECT_SELF.

```
int GetImmortal(
          object oTarget = OBJECT_SELF )
```

This returns true if the target *oTarget* is a creature that has the Immortal property set to true. See SetImmortal.

```
int GetIsDM( object oCreature )
```

If the creature *oCreature* is the Dungeon Master character, return true.

```
int GetIsDMPossessed( object oCreature )
```

If the creature *oCreature* is currently possessed by the Dungeon Master character, return true.

```
int GetIsImmune(
          object oCreature,
          int nImmunityType,
          object oVersus = OBJECT_INVALID )
```

This call returns true only if the creature *oCreature* has immunity of type *nImmunityType* against the race and alignment of object *oVersus*. The immunity is a constant of the form IMMUNITY_TYPE_.... If *oVersus* is an invalid object, then the routine only checks if the immunity exists.

```
int GetIsSpirit( object oCreature )
```

This routine will return true if the creature *oCreature* is a valid object and either the 'SpiritOverride' property is set to true or the racial type is an elemental or a fey.

```
int GetLootable( object oCreature )
```

This returns true if a valid creature *oCreature* is flagged as lootable.

```
int GetMovementRate(
          object oCreature )
```

This returns the speed of the creature *oCreature*.

```
float GetMovementRateFactor(
          object oCreature )
```

This returns to the movement rate factor that is applied to the creature *oCreature*'s speed. This factor is influenced by selected effects and encumbrance.

```
object GetNearestCreature(
          int nFirstCriteriaType,
          int nFirstCriteriaValue,
          object oTarget = OBJECT_SELF,
          int nNth = 1,
          int nSecondCriteriaType = -1,
          int nSecondCriteriaValue = -1,
          int nThirdCriteriaType = -1,
          int nThirdCriteriaValue = -1 )
```

This routine can be used to find the *nNth* closest creature to the object *oTarget* that satisfies a set of criteria. Up to three criteria type/value pairs can be specified that the creature must satisfy. The criteria type is a global constant CREATURE_TYPE_..., while the criteria value depends on the type suffix as follows:

- _CLASS – The value is a CLASS_TYPE_... constant that matches a character class.
- _DOES_NOT_HAVE_SPELL_EFFECT – The value

is a SPELL_... constant that matches a spell.

- _HAS_SPELL_EFFECT – The value is a SPELL_... constant that matches a spell.
- _IS_ALIVE – Use a CREATURE_ALIVE_... constant for the value to specify whether to check for creatures that are alive, dead or both.
- _PERCEPTION – A PERCEPTION_... constant for the value matches based on whether a creature has been perceived or not, and by what sense.
- _PLAYER_CHAR – The value is a global constant of the form PLAYER_CHAR_....
- _RACIAL_TYPE – A RACIAL_TYPE_... for value will match on a particular creature racial type.
- REPUTATION – The value must be a constant of the form REPUTATION_TYPE_..., indicating a friendly, neutral or enemy reputation status.
- SCRIPTHIDDEN – This can be used to select for creatures based on their Script Hidden property by setting value to a CREATURE_SCRIPTHIDDEN_... constant.

```
object GetNearestCreatureToLocation(
          int nFirstCriteriaType,
          int nFirstCriteriaValue,
          location locAt,
          int nNth = 1,
          int nSecondCriteriaType = -1,
          int nSecondCriteriaValue = -1,
          int nThirdCriteriaType = -1,
          int nThirdCriteriaValue = -1 )
```

This routine is similar to GetNearestCreature, except that the call returns creatures based on their proximity to the location *locAt*. See the GetNearestCreature notes for details on the valid parameter values.

```
int GetPolymorphLocked(
          object oCreature )
```

This call will return true only if *oCreature* is a valid creature and it has the polymorph locked flag set.

```
int GetRacialType( object oCreature )
```

Returns the racial type of a creature object *oCreature* as a constant integer. The result matches one of the RACIAL_TYPE_... variables. If the object is not a valid creature, return RACIAL_TYPE_INVALID. Note that there is no RACIAL_TYPE_PLANT constant. Instead check for the number '22', which corresponds to the Plant row of the 'racialtypes.2da' file.

```
int GetSpellResistance(
          object oCreature )
```

This will return the spell resistance of the creature *oCreature*.

```
int GetSubRace( object oCreature )
```

If *oCreature* is a creature, this will return the racial subtype as a value matching a RACIAL_SUBTYPE_... constant. See SetSubRace.

Effects

By default, the subtype of an effect is of subtype magical. This means the effect can be removed by a *dispel magic* spell or rest.

Management

```
void ApplyEffectAtLocation(
          int nDurationType,
          effect eEffect,
          location locAt,
          float fDuration = 0.0f )
```

This routine will apply the effect *eEffect* at the location *locAt*. Valid values for *nDurationType* are constants of the form DURATION_.... If the duration type is temporary, then the length is *fDuration* in seconds. Otherwise, *fDuration* is ignored.

```
void ApplyEffectToObject(
          int nDurationType,
          effect eEffect,
          object oTarget,
          float fDuration = 0.0f )
```

This routine will apply the effect *eEffect* to the object oTarget, which can be a creature or a non-static placeable. Valid values for *nDurationType* are constants of the form DURATION_.... If the duration type is temporary, then the length is *fDuration* in seconds. Otherwise, *fDuration* is ignored. To make an effect permanent, make the effect supernatural and use DURATION_TYPE_PERMANENT.

```
void RemoveEffect(
          object oCreature,
          effect eEffect )
```

This removes effect *eEffect* from creature *oCreature*. Most effects can also be removed by resting or a dispel.

```
void RemoveSEFFromObject(
          object oObject,
          string sSEFName )
```

This call removes an instance of a special effect file (SEF) with the name *sSEFName* from the object *oObject*. This essentially removes all instances of the effect from the object, since there can only be a single instance of an effect running on an object at once.

```
effect SetEffectSpellId(
          effect eEffect,
          int nSpellId )
```

This associates a spell identifier *nSpellId* with the effect *eEffect*, as well as any effects linked to it. This is useful for example, when there is a subsequent need to locate specific effects that do not have an EFFECT_TYPE_... constant. It is also useful for subsequent removal of effects that were applied by an item's tag-based On Equip script. Use the GetEffectSpellId call to find the *nSpellId* of a particular effect, if any. The function returns -1 if the effect does not have a spell identifier.

Restrictions

These calls apply restrictions to effects.

```
effect VersusAlignmentEffect(
          effect eEffect,
          int nLawChaos = ALIGNMENT_ALL,
          int nGoodEvil = ALIGNMENT_ALL )
```

For an effect *eEffect* belonging to a restricted set of effects, this will return an effect that is limited in it's effectiveness to alignments *nLawChaos* along the law-chaos axis and alignments *nGoodEvil* along the good-evil axis. The *nLawChaos* parameter is a ALIGNMENT_... constant that has a LAWFUL, CHAOTIC or ALL suffix. The *nGoodEvil* parameter is an ALIGNMENT_... constant that has a GOOD, EVIL or ALL suffix.

The notes for this routine list the effects that can be modified by this routine. Other effect types will be unmodified.

```
effect VersusRacialTypeEffect(
          effect eEffect,
          int nRacialType )
```

When passed a valid effect *eEffect*, this will return an effect that only functions against creatures belonging to the racial type *nRacialType*, which is a RACIAL_TYPE_... constant. This can be used, for example, on an EffectACIncrease to limit the benefits to attacks by creatures from that racial type. Example: 'nw_s3_herb'.

For this effect, the GetEffectInteger routine appears to return the racial type for the last integer value after all the values for the base effect type have been queried. Thus, a racial type effect for an AC increase will return the modify type in the first field, the AC increase in the second field, and the racial type in the third field.

```
effect VersusTrapEffect(
          effect eEffect )
```

For a valid effect *eEffect*, this will return an effect that is only valid against traps.

Subtypes

The subtype can be changed using the following functions. By default an effect is of subtype SUBTYPE_MAGICAL.

```
effect ExtraordinaryEffect( effect eEffect )
```

This routine returns the effect *eEffect* with its subtype changed to extraordinary. Extraordinary effects can not be removed by a *dispel magic* spell, but they can be removed by resting.

```
effect MagicalEffect( effect eEffect )
```

This routine returns the effect *eEffect* with its subtype changed to magical. These effects can be removed by a *dispel magic* spell or by resting.

```
effect SupernaturalEffect( effect eEffect )
```

This routine returns the effect *eEffect* with its subtype set to supernatural. Permanent supernatural effects can not be removed by resting.

Query

```
object GetEffectCreator(
          effect eEffect )
```

This call returns the object that created the effect *eEffect*. If the effect is produced by an ApplyEffect... call from an event handler script, then the object returned is what ran the script.

```
int GetEffectDurationType(
          effect eEffect )
```

This returns a DURATION_TYPE_... constant giving the duration type of the effect eEffect, or -1 if the effect is not valid.

```
int GetEffectInteger(
          effect eEffect,
          int nIndex )
```

This returns an integer value for the parameter *nIndex* of the effect *eEffect*. Normally, the first parameter has an *nIndex* value of 0, the second has an *nIndex* of 1, and so forth. The value returned depends on the type of effect.

```
int GetEffectSpellId(
          effect eEffect )
```

If an effect *eSpellEffect* was applied by a spell, or if a spell identifier was assigned using SetEffectSpellId, then this routine returns the spell identifier SPELL_... of the effect *eEffect*. This can be used to match up specific effects from a particular cause.

```
int GetEffectSubType(
          effect eEffect )
```

This routine returns the subtype of the effect *eEffect*, which matches a SUBTYPE_... constant. See the Subtypes section.

```
int GetEffectType(
          effect eEffect )
```

This call returns a constant EFFECT_TYPE_... that corresponds to the type of the effect *eEffect*. If an effect is invalid, or if a valid effect does not have a valid type, this function returns EFFECT_INVALIDEFFECT.

```
effect GetFirstEffect(
          object oCreature )
```

This returns the first of possibly multiple effects on the creature *oCreature*. This routine re-initializes the sequence of effects returned by repeated calls to the GetNextEffect function.

```
int GetHasFeatEffect(
          int nFeat,
          object oObject = OBJECT_SELF )
```

This call returns true only if the object *oObject* has any effects generated by the feat *nFeat*, which is a FEAT_... constant.

```
int GetHasAnySpellEffect(
          object oObject )
```

This routine returns true if the object *oObject* is valid and has any spell effects applied to it.

```
int GetHasSpellEffect(
          int nSpell,
          object oObject = OBJECT_SELF )
```

This returns true of the object *oObject* has any active effects applied by the spell with identifier *nSpell*, which is a

SPELL_... constant. This will not return true when an effect is created via a delayed command that generates a spell.

```
int GetIsEffectValid(
          effect eEffect )
```

If the effect *eEffect* is a valid effect and it has been applied to a creature, this should return true. Otherwise it will return false. An alternative is to call GetEffectType on the effect and test if it returns EFFECT_TYPE_INVALIDEFFECT.

```
effect GetNextEffect(
          object oCreature )
```

Calling GetFirstEffect will return the first effect on the creature *oCreature*. Thereafter, repeatedly calling this routine will return the next in the sequence of effects on the creature. Finally, EFFECT_TYPE_INVALIDEFFECT is returned when there are no additional effects.

```
int GetIsEffectValid( effect eEffect )
```

This routine should return true if the effect *eEffect* has been created by an Effect... routine and it has been applied to an object. However, this routine did not work properly even when the effects were properly applied, so I would not rely on it.

Constructors

These are the various effects that can be applied. Many of the effects have a type matching an EFFECT_TYPE_... constant, and the various types have one or more corresponding icons listed in 'effecticons.2da'.

Basic

```
effect EffectBlindness()
```

The subject of this effect is rendered blind. This gives the subject a -4 to attacks and a 50% chance to miss entirely. Example: 'nw_s0_blinddeaf'.

```
effect EffectCharmed()
```

This causes a charm effect, with the effected creature being charmed by the calling object. A charmed creature is unable to attack the charmer. Example: 'nw_s0_charmper'.

```
effect EffectConcealmentNegated()
```

When this goes into effect, all concealment and miss chance effects on the target are ignored. It effectively cancels out EffectConcealment and EffectMissChance effects while it is active.

```
effect EffectConfused()
```

The target of this effect behaves confused, per the 4th level wizard *confusion* spell. The subject will either wander aimlessly, stand still or attack the nearest visible target. Example: 'nw_s0_confusion'.

```
effect EffectDamageReductionNegated()
```

When this is in effect, all damage reduction on a creature is ignored. Example: 'x2_s1_psibarr'.

```
effect EffectDarkness()
```

Creates an effect like the *darkness* spell. A creature in darkness is blinded but is invisible to others. However, note that the 'nw_s0_darknessa' script uses EffectConcealment instead of this call.

```
effect EffectDarkVision()
```

Provides the recipient with dark vision, allowing it to see in complete darkness. There is no argument for setting a range. The description for the Half-Orc says that this ability allows the recipient to see in the dark up to 60 feet. Example: 'nw_s0_iseeunsen'.

```
effect EffectDazed()
```

The target is dazed, and can take no actions except move, but does not suffer an AC penalty. Examples: 'nw_s0_daze' and 'nw_s0_hammgods'.

```
effect EffectDeaf()
```

The subject is deaf. The game rules apply a -4 penalty to initiative, automatic failure on Listen checks and a 20% chance of spell failure. Example: 'nw_s0_blinddeaf'.

```
effect EffectDetectSpirits()
```

This allows spirits to be detected by the target. They will be displayed on the minimap. Spirits include fey, elementals and incorporeal undead such as wraith and shadows. Example: 'nx_s0_detectspirits'.

```
effect EffectDetectUndead()
```

This effect allows undead to be detected. They will be displayed on the minimap. Example: 'nw_s0_detctundd'.

```
effect EffectDominated()
```

The target is subject to a domination effect, such as that produced by a *dominate monster* spell. The target is dominated by the object applying the effect. A dominated PC is dazed, while a dominated NPC can be forced to attack an enemy of the dominator. Example: 'nw_s0_dompers'.

`effect EffectEntangle()`

The subject has it's movement restricted, takes a -2 penalty to all attacks and a -4 penalty to armor class. Example scripts: 'nw_s0_entangle' and 'x0_s3_tangle'.

`effect EffectEthereal()`

The subject of this effect is ignored by enemies. Example script: 'x0_s0_ether'.

`effect EffectFrightened()`

The target is subject to a fear effect, which causes it to run away from the source. A frightened creature suffers a -2 penalty to saving throws. Example: 'nw_s0_fear'.

`effect EffectHaste()`

This creates the effect of a *haste* spell, which increases movement by 50%, allows an extra attack action, and gives a +1 dodge bonus to AC. Spell times are halved while hasted. Example: 'nw_s0_haste'.

`effect EffectJarring()`

The notes say this creates a jarring effect. I observed no result from applying this to either a creature or a location.

`effect EffectKnockdown()`

The subject will be knocked off their feet then remain sitting until the effect wears off. The notes recommend applying this as a temporary effect lasting at least 3 seconds. This does not have a matching EFFECT_TYPE_... constant. Example: 'x0_s0_gustwind'.

`effect EffectInsane()`

The notes for this command say it causes the subject to attack the nearest target, whether friend or foe. Note that the *insanity* spell is a continual EffectConfused. The notes in 'nw_g0_insane' say that it is the heartbeat script for any creature subject to this effect.

`effect EffectLowLightVision()`

The effect recipient becomes the beneficiary of the low-light vision special ability. Example: 'nw_s0_lowltvisn'. This does not have a matching EFFECT_TYPE_... constant.

`effect EffectMaxDamage()`

Each time the subject makes a successful weapons-based attack, the maximum possible damage is applied. That is, if dice are rolled, the damage uses the highest number of pips on each dice. Example: 'nw_s2_furaslt'.

`effect EffectNWN2ParticleEffect()`

The notes say this creates a particle effect on an object or at a location. I saw no result from applying this effect. See the EffectNWN2ParticleEffectFile call.

`effect EffectPetrify()`

The effect description says the target is petrified, and is inflicted with a paralyze effect and a stoneskin visual covering. However, when I tested this there was no visual effect. This effect is used with the Burst of Glacial Wrath spell, as applied by the 'nx_s0_glacial' script.

Note that there is no EffectImmunity effect option that can negate a petrify effect. In the 'x0_i0_spells' include file, the spellsIsImmuneToPetrification routine grants immunity to certain creatures based upon their appearance.

`effect EffectResurrection()`

When applied as an instantaneous effect, the target is brought back to life after being slain. This likely requires that the creature's Resurrectable property be set to true. Example: 'nw_s0_raisdead'.

`effect EffectSeeInvisible()`

The target is able to see invisible creatures and target them with spells and attacks. Example: 'nw_s0_seeinvis'.

`effect EffectSeeTrueHPs()`

The effect notes say that this allows the hit points of the subject to be viewed. This does not have a matching EFFECT_TYPE_... constant. I observed no result from applying this effect.

`effect EffectSilence()`

This creates a silence effect like the cleric spell *silence*. The target can make no noise and can not hear anything. Most spells can not be cast while silenced. Example script: 'nw_s0_silencea'.

`effect EffectSleep()`

The recipient is subject to a sleep effect, like the wizard *sleep* spell. Some creatures are immune to sleep. Example script: 'nw_s0_sleep'.

`effect EffectSlow()`

The subject of this effect is slowed, per the wizard *slow* spell. The target moves at 50% of normal speed and suffer a -2 penalty to AC, reflex saves, and attack rolls. The number of attacks per round is reduced by one. Example script:

'nw_s0_slow'.

```
effect EffectStunned()
```

The effect's target is stunned. This can also be used on an animal to make it stand motionless and not turn to face the PC on a conversation attempt. Example: 'nw_s0_pwstun'.

```
effect EffectTimeStop()
```

This effect should stop all activities except for the target of the effect. Note that there is no *time stop* spell in NWN2. Example: 'nw_s0_timestop'.

```
effect EffectTrueSeeing()
```

This creates an effect like the *true seeing* spell that allows the subject target ethereal and invisible creatures. Example script: 'nw_s0_truesee'.

```
effect EffectTurned()
```

This effect causes the recipient to behave as turned, as per the clerical turn undead ability. Example: 'nw_s2_turndead'.

```
effect EffectUltravision()
```

The recipient gains the ultravision ability. The benefits of this effect are not described. Example: 'nw_s0_darkvis'.

```
effect EffectWildshape()
```

This effect flags the recipient as using the wild shape ability, but does not cause the polymorph. You can link this with an EffectPolymorph for a natural shape shifting effect.

Configurable

```
effect EffectAbilityDecrease(
          int nAbility,
          int nModifyBy )
```

The ability *nAbility* is one of the ABILITY_... constants. This effect will reduce the ability score by an amount *nModifyBy*. The latter is a positive integer. If the subject is a PC, this will cause a red icon to appear next to the portrait and an entry in the character window for the duration. Example script: 'nw_s3_alcohol'.

For this effect, the GetEffectInteger routine will return the ability type for the first integer value and the ability modifier for the second.

```
effect EffectAbilityIncrease(
          int nAbility,
          int nModifyBy )
```

The ability *nAbility* is one of the ABILITY_... constants. This effect will increase the ability score by an amount *nModifyBy*. The latter is a positive integer. If the subject is a PC, this will cause a blue icon to appear next to the portrait and an entry in the character window for the duration. Example: 'x0_s0_owlins'.

For this effect, the GetEffectInteger routine will return the ability type for the first integer value and the ability modifier for the second.

```
effect EffectAbsorbDamage(
          int nACTest )
```

The notes for this call indicate it creates a damage absorption effect. The *nACTest* parameter gives the armor class that must be exceeded to bypass the damage absorption. *How then does this differ from the normal armor class?*

For this effect, the GetEffectInteger routine will return the nACTest parameter for the first integer value. I tested the effect but observed no damage absorption. See also EffectDamageReduction.

```
effect EffectACDecrease(
          int nValue,
          int nModifyType = AC_DODGE_BONUS,
          int nDamageType =
            AC_VS_DAMAGE_TYPE_ALL )
```

This is similar to EffectACIncrease, except it reduces the armor class by the amount *nValue*. The cause of the reduction is *nModifyType* and it applies to damage of type *nDamageType*. The first is a valid global constant value of the type AC_..._BONUS. The second is a valid global constant of the type DAMAGE_TYPE_..., or else the default type above that is valid against all damage types. There is only one AC_VS_* constant.

For this effect, the GetEffectInteger routine will return the modification type for the first integer value and the AC reduction for the second.

```
effect EffectACIncrease(
          int nValue,
          int nModifyType = AC_DODGE_BONUS,
          int nDamageType =
            AC_VS_DAMAGE_TYPE_ALL,
          int bVsSpiritsOnly = FALSE )
```

This effect applies an *nValue* increase in armor class of type *nModifyType*, where the latter is a valid constant of type AC_..._BONUS. Note that armor class bonuses of the same type do not stack. If the *nDamageType* is a constant of type DAMAGE_TYPE_..., it only applies against attacks that inflict that damage type. If *bVsSpiritsOnly* is true, the bonus only applies against attacks by elementals, fey or creatures that return true to the GetIsSpirit routine.

Examples: 'nw_s0_barkskin' and 'x0_s0_shield'.

```
effect EffectAppear(
          int nAnimation = 1 )
```

The *nAnimation* setting for this call determines the animation that a creature will use to appear or disappear. The call notes mention that most creatures only have a single animation available. This does not have a matching EFFECT_TYPE_... constant.

```
effect EffectArcaneSpellFailure(
          int nPercent )
```

This effect will increase the probability of arcane spell failure by a percentage of *nPercent*. The effect stacks with the arcane spell failure effect caused by armor and shields. It is ignored by classes and special abilities that are not subject to the arcane spell failure rules. Thus it would not impact divine spells.

The GetEffectInteger routine will return the percentage value in the first integer for this effect.

```
effect EffectArmorCheckPenaltyIncrease(
          object oTarget,
          int nPenaltyAmt )
```

This effect will increase the armor check penalty by *nPenaltyAmt* for the creature target. *Why is oTarget being passed here?*

```
effect EffectAssayResistance(
          object oTarget )
```

The subject of this effect gains a bonus versus spell resistance against the target *oTarget*. According to the notes for the *assay resistance* spell, this effect gives a +10 bonus on caster level checks to overcome the target's spell resistance. Example script: 'nw_s0_assayrest'.

```
effect EffectAttackDecrease(
          int nPenalty,
          int nModifierType =
           ATTACK_BONUS_MISC )
```

This effect decreases the base attack of the subject by the amount *nPenalty,* a positive integer. The *nModifierType* is equal to one of the ATTACK_BONUS_... constants. The GetEffectInteger routine will return the value of the penalty in the first integer for this effect and the modifier type in the second. Example: 'x0_s0_flare'.

```
effect EffectAttackIncrease(
          int nBonus,
          int nModifierType =
            ATTACK_BONUS_MISC )
```

This increases the subject's base attack of the modifier type *nModifierType* by the amount *nBonus*. The *nModifierType* is equal to one of the ATTACK_BONUS_... constants. Example: 'x0_s0_truestrike'.

```
effect EffectBABMinimum( int BABMin )
```

The call notes say this creates a base attack bonus minimum effect *BABMin*, but it is unclear what this actually does. This does not have a matching EFFECT_TYPE_... constant.

```
effect EffectBardSongSinging(
          int nSpellID )
```

This creates the effect of one of the bard songs with spell ID *nSpellID*. *Should this be a FEAT_BARDSONG_... constant?*

```
effect EffectBeam(
          int nBeamVisualEffect,
          object oEffector,
          int nBodyPart,
          int bMissEffect = FALSE )
```

This effect causes a beam to be emitted from the object *oEffector* toward the effect subject, which must be an object. The *bBeamVisualEffect* is a constant of type VFX_BEAM_... that determines the beam type, such as VFX_BEAM_LIGHTNING for a continuous stream of lightning. The *nBodyPart* is a constant of the form BODY_NODE_.... If *bMissEffect* is true, the beam will miss the target and hit a random location nearby. This effect should be applied with a temporary or permanent duration. Examples: 'nw_s0_rayfrost' and 'nw_s0_chlightn'.

```
effect EffectBonusHitpoints(
          int nHitpoints )
```

This effect provides the subject an additional *nHitpoints*, just as if he had gained them from normal class level progression. While the effect continues, if the hit points are lost due to damage, they can be healed again.

```
effect EffectBreakEnchantment(
          int nLevel )
```

This effect functions like a spell that will free the target from curses, enchantments and transmutations, at a caster level determined by *nLevel*.

```
effect EffectConcealment(
          int nPercentage,
          int nMissType =
            MISS_CHANCE_TYPE_NORMAL )
```

The subject of this effect gains the benefits of concealment with the percentage *nPercentage*. The *nMissType* is a constant MISS_CHANCE_TYPE_..., which can be used to limit the effect to melee or ranged. The GetEffectInteger routine will return the value of the miss percentage in the first integer for this effect. Example: 'x0_s0_entrshield'.

```
effect EffectCurse(
          int nStrMod = 1,
          int nDexMod = 1,
          int nConMod = 1,
          int nIntMod = 1,
          intWisMod = 1,
          int iChaMod = 1 )
```

This effect applies a curse effect that results in a reduction of the subject's characteristics by the amounts passed in the arguments. Thus the *nConMod* determines the reduction in the Constitution ability score. Example: 'nw_s0_bescurse'.

Note that the *bestow curse* spell has multiple possible effects, one of which is a reduction of up to 6 in a single characteristic. A second effect is a -4 penalty on attacks, saves, ability checks and skill checks, which would require multiple combined effects.

```
effect EffectDamage(
          int nDamageAmount,
          int nDamageType =
            DAMAGE_TYPE_MAGICAL,
          int nDamagePower =
            DAMAGE_POWER_NORMAL,
          int bIgnoreResistances = FALSE )
```

When applied as an instantaneous effect, this inflicts *nDamageAmount* points of damage to the subject. The damage is of type *nDamageType*, which is a constant value of the form DAMAGE_TYPE_.... Creatures with energy resistance may ignore some of the damage of the matching type. The *nDamagePower* is a DAMAGE_POWER_... global constant. If *bIgnoreResistances* is true, the damage will bypass any damage immunity, reduction or resistance on the subject.

This does not have a matching EFFECT_TYPE_... constant.

```
effect EffectDamageDecrease(
          int nPenalty,
          int nDamageType =
            DAMAGE_TYPE_MAGICAL )
```

This effect will reduce the damage of type *nDamageType* that the subject can inflict by the amount *nPenalty*. The type is a constant of the form DAMAGE_TYPE_.... The value DAMAGE_TYPE_ALL reduces all damage, regardless of the type.

```
effect EffectDamageImmunityDecrease(
          int nDamageType,
          int nPercentImmunity )
```

This call decreases the subject's damage immunity by the percentage *nPercentImmunity* against the *nDamageType* type of damage. The damage type is a DAMAGE_TYPE_... global constant, with DAMAGE_TYPE_ALL applying to all damage types.

```
effect EffectDamageImmunityIncrease(
          int nDamageType,
          int nPercentImmunity )
```

This call increases the subject's damage immunity by the percentage *nPercentImmunity* against the *nDamageType* type of damage. The damage type is a DAMAGE_TYPE_... global constant, with DAMAGE_TYPE_ALL applying to all damage types. Example: 'nx_s2_immunityelectricity'.

```
effect EffectDamageIncrease(
          int nBonus,
          int nDamageType =
            DAMATE_TYPE_MAGICAL,
          int nVersusRace = -1 )
```

This effect will increase the amount of damage inflicted by *nBonus*, which must be one of the DAMAGE_BONUS_... constants. (The command notes that failure to use these constants can result in odd behavior.) The bonus is of type *nDamageType*, which is a DAMAGE_TYPE_... global constant. The *nVersusRace* parameter is undocumented.

```
effect EffectDamageOverTime(
          int nAmount,
          float fIntervalSeconds,
          int nDamageType =
            DAMAGE_TYPE_MAGICAL,
          int nIgnoreResistances = FALSE )
```

This effect inflicts *nAmount* of hit point damage over *fIntervalSeconds* seconds, rather than instantaneously. (Thus it works like a wounding effect.) The damage is of type *nDamageType*, which is a global constant DAMAGE_TYPE_.... If *nIgnoreResistances*, the damage will bypass the target's damage immunity, reduction and resistance. Note that there is no immunity effect or item property that works specifically against this effect.

```
effect EffectDamageReduction(
          int nAmount,
          int nDRSubType =
            DAMAGE_POWER_NORMAL,
          int nLimit = 0,
          int nDRType = DR_TYPE_MAGICBONUS )
```

This will reduce the amount of damage of type *nDRType* and sub-type *nDRSubType* by *nAmount*. If *nLimit* is not zero, this effect will absorb that amount of damage then end (as per the *stoneskin* spell). Otherwise it will absorb an infinite amount of damage.

The damage reduction type *nDRType* is a global constant DR_TYPE_.... The subtype depends on the type. Thus, if *nDRType* is DR_TYPE_ALIGNMENT, the subtype is an ALIGNMENT_... constant representing a particular location on the alignment grid, or ALIGNMENT_ALL. Thus, for 'damage reduction 5/magic', set the *nAmount* to 5, *nDRType* to DR_TYPE_MAGICBONUS, *nLimit* to 0 and *nDRSubType* to DAMAGE_POWER_NORMAL.

Example: 'nw_s0_protarrow'.

```
effect EffectDamageResistance(
          int nDamageType,
          int nAmount,
          in nLimit = 0 )
```

This will reduce the amount of damage by *nAmount* from a specific damage type *nDamageType*. The latter is a global constant DAMAGE_TYPE_.... This effect ends when *nLimit* points of damage have been absorbed, or it can absorb an infinite amount when *nLimit* is zero. Example script: 'nw_s2_shnshld'.

```
effect EffectDamageShield(
          int nDamageAmount,
          int nRandomAmount,
          int nDamageType )
```

When an attacker makes a successful attack of type *nDamageType* to the effect target, the attacker suffers a base *nDamageAmount* of damage plus a *nRandomAmount* global constant of the form DAMAGE_TYPE_... that determines the random damage. The *nDamageType* is a DAMAGE_TYPE_... constant.

For this effect, the GetEffectInteger routine will return the value of the damage amount with the first integer, the value of the random amount constant with the second, and the value of the damage type constant with the third.

```
effect EffectDeath(
          int nSpectacularDeath = FALSE,
          int nDisplayFeedback = TRUE,
          int nIgnoreDeathImmunity = FALSE,
          int bPurgeEffects = TRUE )
```

This effect can be applied to make a mortal creature (with 'plot' and 'immortal' properties set to false) appear to die. If *nSpectacularDeath* is true, the creature will die in a spectacular fashion. The *nIgnoreDeathImmunity* flag will ignore death immunity. The *bPurgeEffects* preserves visual effects on the body, but may cause the effect to fail in some circumstances (such as bonus HP).

This does not have a matching EFFECT_TYPE_... constant. Example script: 'nw_s0_destruc'.

```
effect EffectDisappear( int nAnimation = 1 )
```

This effect causes a creature to move away and then destroy itself. The *nAnimation* determines how the creature moves away. The command notes say that most creatures only have one animation mode.

```
effect EffectDisappearAppear(
          location locReappear,
          int nAnimation = 1 )
```

This effect functions like EffectDisappear, except that when the effect ends the creature will reappear at the location *locReappear*.

```
effect EffectDisease( int nDiseaseType )
```

The target of this effect is infected with the disease nDiseaseType, which is a constant DISEASE_.... The standard disease types are defined in the disease.2da file. Thus, DISEASE_DEMON_FEVER is a supernatural disease type with a DC 18 save to avoid infection, an incubation period of 1 hour, and inflicting 1d6 damage to Constitution. After 24 hours, the script nw_s3_demonfev is run, which can inflict a permanent 1 point Con loss. Example script: 'nw_s0_contagion'.

```
effect EffectDisintegrate( object oTarget )
```

This effect creates a disintegrate visual effect that is applied to the target *oTarget*. The command notes recommend that the creature already be dead when this is run.

```
effect EffectDispelMagicAll(
          int nCasterLevel,
          action aOnDispelEffect )
```

This creates a *dispel magic* effect at caster level *nCasterLevel* on all effects on the target. If any of the effects is removed, the action *aOnDispelEffect* will be called once. See the Actions section. For best results, this should be called using the 'spellsDispelMagic' function in 'x0_i0_spells'.

```
effect EffectDispelMagicBest(
          int nCasterLevel,
          action aOnDispelEffect )
```

This functions like EffectDispelMagicAll, except it is only applied to the 'best' effect. *It is unclear what criteria is used to select the 'best'.* For best results, this should be called using the 'spellsDispelMagic' function in 'x0_i0_spells'.

```
effect EffectEffectIcon( int nEffectIconId )
```

This places an effect icon above the subject creature. The value *nEffectIconId* is an EFFECT_TYPE_... constant that gives the row number in the 'effecticons.2da' file.

```
effect EffectHeal( int nDamageToHeal )
```

When applied as an instantaneous effect, this will heal *nDamageToHeal* points of damage to creatures, doors and placeables (as of patch v1.06). It has no effect if *nDamageToHeal* is below zero. This does not have a matching EFFECT_TYPE_... constant. Example script: 'nw_s2_wholeness'.

```
effect EffectHealOnZeroHP(
          object oTarget,
          int nDamageToHeal )
```

The target *oTarget* of this effect will be healed for *nDamageToHeal* points of damage when it falls to zero hit points or below.

```
effect EffectHideousBlow(
          int nMetaMagic )
```

This causes a hideous blow effect using the *nMetaMagic* modifier, which is a METAMAGIC_... constant. This implements the Warlock power of the same name. Example: 'nw_s0_ihideousb'.

```
effect EffectHitPointChangeWhenDying(
          float fHitPointChangePerRound )
```

If *fHitPointChangePerRound* is not zero, this changes that amount of hit points per round in the target. Per the rules books, PCs with hit points between -1 and -9 lose one hit point per round until bandaged.

```
effect EffectImmunity( int nImmunityType )
```

This creates an immunity of type *nImmunityType*, which is an IMMUNITY_TYPE_... constant. The GetEffectInteger routine will return the value of the immunity type in the first integer for this effect. Example scripts: 'nw_s0_freemove' and 'nw_s0_deaward'.

```
effect EffectInvisibility(
          int nInvisibilityType )
```

The subject gains invisibility of the type *nInvisibilityType*, which is an INVISIBILITY_TYPE_... constant. Example: 'nw_s2_invisib'.

```
effect EffectLinkEffects(
          effect eChildEffect,
          effect eParentEffect )
```

This call links the child effect *eChildEffect* as a child of the parent effect *eParentEffect*, returning the linked effect. This is useful, for example, when you want the removal of one of the effects to also cause the removal of the linked effects. See the call notes on what happens when the subject is immune to one of the effects.

This routine does not have a matching EFFECT_TYPE_... constant. The script 'nw_s0_freemove' contains an example of multiple linked effects.

```
effect EffectMesmerize(
          int nBreakFlags,
          float fBreakDist = 0.0f )
```

The effect recipient causes a mesmerizing effect within the radius *fBreakDist*. The *nBreakFlags* is a set of boolean flags, based on a sum of MESMERIZE_BREAK_ON_... constants, that determines what types of events can disrupt the mesmerize effect.

```
effect EffectMissChance(
          int nPercentage,
          int nMissChanceType =
            MISS_CHANCE_TYPE_NORMAL )
```

If *nPercentage* is a value between 1 and 99, this creates an effect that causes a miss chance during an attack of type *nMissChanceType*, which is a MISS_CHANCE_TYPE_... variable that determines the types of attacks that can miss. This can be a normal, melee or ranged attack.

This is essentially the same as EffectConcealment.

```
effect EffectModifyAttacks( int nAttacks )
```

This increases the number of attacks per round of the effected target by *nAttacks*. The nAttacks parameter is limited to 5 or less. This does not have a matching EFFECT_TYPE_... constant.

Normally, a character has a number of attacks equal to 1 + (BAB – 1)/5, where BAB is the base attack bonus.

```
effect EffectMovementSpeedDecrease(
          int nPercentChange )
```

This reduces the movement speed by *nPercentChange* percentage, which is an integer between 0 and 99. Thus a value of 99 will slow movement to a crawl.

```
effect EffectMovementSpeedIncrease(
          int nPercentChange )
```

This increases the movement by *nPercentChange* percent, which is an integer between 0 and 99. Thus, the maximum speed increase is 199% upon passing a value of 99. If GetEffectInteger is called for this effect with the second field set to zero, that function will return 100 + the speed increase from this effect.

```
effect EffectNegativeLevel(
          int nNumLevels,
          int bHPBonus = FALSE )
```

The effect lowers the subject's class levels by *nNumLevels* levels. This is only valid for values between 1 and 100. Example: 'nw_s0_enedrain'.

```
effect EffectNWN2ParticleEffectFile(
          string sDefinitionFile )
```

This function should create a particle emitter based upon the definition file *sDefinitionFile*.

```
effect EffectNWN2SpecialEffectFile(
          string sFileName,
          object oTarget = OBJECT_INVALID,
          vector vTargetPosition =
            [ 0.0, 0.0, 0.0 ] )
```

This creates an effect based on a special effects file *sFilename*. Valid files include those available under "Appearance (visual effect)" menu in the creature properties. See the Effect Files section for brief descriptions of the individual files, and the function notes for more details on this call.

```
effect EffectOnDispel(
          float fDelay,
          action aOnDispelEffect )
```

This is intended to be linked to another spell effect via the EffectLinkEffects routine. When the linked spell effect is dispelled by an EffectDispelMagic... effect, then the action script *aOnDispelEffect* is executed. The *fDelay* parameter is undocumented, but would seem to be used for the delay in seconds before the action is executed.

```
effect EffectParalyze(
          int nSaveDC = -1,
          int nSave = SAVING_THROW_WILL,
          int nSaveEveryRound = TRUE )
```

The target of this effect is paralyzed. The saving throw to avoid this effect is *nSave*, which is a constant of the type SAVING_THROW_.... If *nSaveEveryRound* is true, then each round the subject can make a save attempt at difficulty class *nSaveDC* to escape the effect. Example scripts: 'nw_s0_holdmon' and 'nw_s0_1carrion'.

```
effect EffectPoison( int nPoisonType )
```

The subject is poisoned by the toxin *nPoisonType*, which is a constant of the form POISON_.... Example script: 'nw_s0_poison'.

The effects of each poison are determined by data in the matching row of the 'poison.2da' file. Thus, POISON_DEATHBLADE on row 12 has a Save DC of 20, causes initial 1d6 Con damage, followed by 2d6 Con secondary damage. Some of the poisons use scripts to apply their effects, such as Sassone Leaf Residue which runs 'nw_s0_1sassone'.

```
effect EffectPolymorph(
          int nPolymorphSelection,
          int nLocked = FALSE,
          int bWildshape = FALSE )
```

The subject of this effect is polymorphed into the creature type *nPolymorphSelection*, which is a global constant of type POLYMORPH_TYPE_.... The *nLocked* parameter is undocumented, but it most likely determines whether the creature can voluntarily reverse the polymorph. This state can be obtained with a call to GetPolymorphLocked. If *bWildshape* is true, then the polymorph is treated as a Druid's wild shape ability. Example: 'nw_s0_shapechg'.

When called with this effect, the GetEffectInteger routine will return the value of the polymorph selection in the first integer.

```
effect EffectRegenerate(
          int nAmount,
          float fIntervalSeconds )
```

The subject of this effect will recover *nAmount* of hit point damage every *fIntervalSeconds* seconds. For this effect, the GetEffectInteger routine will return the amount regenerated for the first integer value and the time interval in milliseconds with the second integer value. Thus, for example, if RoundsToSeconds(1) is passed for the interval in seconds, the second integer value will be 6000. Example script: 'nw_s0_regen'.

```
effect EffectRescue(
          int nSpellId )
```

The notes for this routine state that it is a placeholder effect. The *nSpellId* is matched against the rescue effect. This effect is used with the Rescue epic feat, which provides the 'Rescue' spell ability defined on row #1075 of 'spells.2da'. See the 'nx_s2_rescue' script for details.

```
effect EffectSanctuary(
          int nDifficultyClass )
```

This effect is used to simulate the *sanctuary* spell, which causes other creatures to ignore the target unless they succeed in a saving throw with a *nDifficultyClass* difficulty class. Example: 'nw_s0_sanctuary'.

```
effect EffectSavingThrowDecrease(
          int nSave,
          int nValue,
          int nSaveType =
            SAVING_THROW_TYPE_ALL )
```

The target creature suffers a penalty of *nValue* to all saving throws *nSave*, which is a SAVING_THROW_... constant for fortitude, reflex, will or all. The penalty can be limited to saving throws against the type *nSaveType*, which is a constant SAVING_THROW_TYPE_... and applies to a particular spell descriptor. Example: 'nw_s1_gazedoom'.

```
effect EffectSavingThrowIncrease(
          int nSave,
          int nValue,
          int nSaveType =
            SAVINGTHROW_TYPE_ALL,
          int bVsSpiritsOnly = FALSE )
```

This is similar to EffectSavingThrowDecrease, except that it provides a bonus to saving throws. The additional parameter *bVsSpiritsOnly* will limit the effect to saving throws versus spirits when true. The GetEffectInteger routine will return the nValue modifier for the first integer value, nSave value for the second and the nSaveType for the third. Example script: 'nw_s0_heroism'.

```
effect EffectSetScale(
          float fScaleX,
          float fScaleY = -1.0,
          float fScaleZ = -1.0 )
```

This effect functions like the Scale property for a creature. If *fScaleY* and *fScaleZ* are left at their default values, all dimensions are scaled by the *fScaleX* multiple. Otherwise, the individual dimensions are scaled by the respective parameters. There is not a matching EFFECT_TYPE_... constant. Example: 'nw_s2_enlrgeper'.

```
effect EffectShareDamage(
          object oHelper,
          int nAmountShared = 50,
          int nAmountCasterShared = 50 )
```

With this effect, the creature *oHelper* will absorb the percentage *nAmountCasterShared* of the damage suffered by the recipient. The subject of the effect only receives *nAmountShared* percent of the total damage. Note that the two amounts do not need to add up to 100. Any extra damage vanishes, or any surplus is added to the total damage. Example: 'nw_s2_guardlord'.

```
effect EffectSkillDecrease(
          int nSkill,
          int nValue )
```

The effect reduces the skill *nSkill* by the positive integer *nValue*, where the skill is a valid SKILL_... constant. If SKILL_ALL_SKILLS is passed for the skill, then all skills will receive the same modifier. If GetEffectInteger is called for this effect, it will return the skill type for the first integer value and the value for the second.

```
effect EffectSkillIncrease(
          int nSkill,
          int nValue )
```

This will increase the skill *nSkill* by the positive integer *nValue*, where the skill is a valid SKILL_... constant. An *nSkill* value of SKILL_ALL_SKILLS will cause all skills to increase by *nValue*. Examples: 'nw_s0_lore' and 'nw_s0_heroism'.

```
effect EffectSpellFailure(
          int nPercent = 100,
          int nSpellSchool =
            SPELL_SCHOOL_GENERAL )
```

This effect gives a percentage chance *nPercent* for spells in the magic school *nSpellSchool* to fail, which is a constant SPELL_SCHOOL_.... This effect is independent of an arcane spell failure check, so an arcane spell caster can be subject to both spell failure effects applied in sequence.

```
effect EffectSpellImmunity(
          int nImmunityToSchool )
```

When a valid constant is passed as *nImmunityToSchool*, this effect will provide immunity to spells of that school. This can be a SPELL_SCHOOL_... constant, a row number in the 'spells.2da' file, or use SPELL_ALL_SPELLS for immunity to all spells. Example: 'x0_s0_shield'.

```
effect EffectSpellLevelAbsorption(
          int nMaxSpellLevelAbsorbed,
          int nTotalSpellLevelsAbsorbed = 0,
          int nSpellSchool =
            SPELL_SCHOOL_GENERAL )
```

This effect will absorb spells cast at the subject. The *nMaxSpellLevelAbsorbed* is the maximum spell level that can be absorbed. The *nTotalSpellLevelsAbsorbed* field gives the total spell levels that can be absorbed, or zero for all applicable spells. If *nSpellSchool* is set to a constant SPELL_SCHOOL_..., only spells in that school will be absorbed. Example: 'nw_s0_ghostvis' and 'nw_s0_globeinv'.

```
effect EffectSpellResistanceDecrease(
          int nValue )
```

This effect lowers the recipient's spell resistance by the positive integer *nValue*.

```
effect EffectSpellResistanceIncrease(
          int nValue,
          int nUses = -1 )
```

This effect increases the recipient's spell resistance by the positive integer *nValue*. If *nUses* is set to a positive integer, it defines the number of times this effect can be applied to spell resistance checks before it will end.

```
effect EffectSummonCopy(
          object oSource,
          int nVisualEffectId = VFX_NONE,
          float fDelaySeconds = 0.0f,
          string sNewTag = "",
          int nNewHP = 0,
          string sScript = "" )
```

This creates a duplicate copy of the existing creature *oSource* that appears *fDelaySeconds* seconds after the visual effect nVisualEffectId is played. If *nNewTag* is a non-null string, new creature is assigned that as it's tag; otherwise it uses the same tag as the original. If *nNewHP* is non-zero, the copy will be given that many hit points; otherwise it gains the starting hit point total of the original. The *sScript* can be the name of a script to run by the copy.

The creature copy appears with spells per day and memorized spells re-initialized. This effect can not make copies of creatures with their Plot or Immortal properties set to true.

```
effect EffectSummonCreature(
          string sCreatureResref,
          int nVisualEffectId = VFX_NONE,
          float fDelaySeconds = 0.0f,
          int nUseAppearAnimation = 0 )
```

This creates the creature with the resource reference *sCreatureResref* then places it in the party or faction of the effected target. The creature will appear *fDelaySeconds* after the visual effect *nVisualEffectId* is played. If the *nUseAppearAnimation* is 1, the creature will use it's 'appear' animation. (Some creatures have a second animation for appearance, which is set with a value of 2.) Example: 'nw_s0_animdead'.

```
effect EffectSwarm(
          int nLooping,
          string sCreatureTemplate1,
          string sCreatureTemplate2 = "",
          string sCreatureTemplate3 = "",
          string sCreatureTemplate4 = "" )
```

If *nLooping* is true, this will create a sequence of creatures that each replace the previously slain creature. It will cycle through the list of non-null *sCreatureTemplate...* parameters to create the creatures, returning to the *sCreatureTemplate1* at the end of the list.

```
effect EffectTemporaryHitpoints(
          int nHitPoints )
```

This effect creates *nHitPoint* temporary hit points that can not be recovered by sleep or healing. Example: 'nw_s0_aid'.

```
effect EffectTurnResistanceDecrease(
          int nHitDice )
```

The effectiveness of a turn undead attempt is determined by the number of hit dice of the undead. This causes the recipient to be subject to turn attempts as though they had *nHitDice* fewer hit dice, where *nHitDice* is a positive integer.

```
effect EffectTurnResistanceIncrease
          int nHitDice )
```

The subject becomes more difficult to turn, per the turn undead ability. They must be turned as though they had *nHitDice* additional hit dice, where *nHitDice* is a positive integer.

```
effect EffectVisualEffect(
          int nVisualEffectId,
          int nMissEffect = FALSE )
```

This creates the visual effect *nVisualEffectId* on the target object. If *nMissEffect* is true, the effect will be played at a random position near the effect target. The various effects correspond to rows in the 'visualeffects.2da' file, and many have corresponding VFX_* constants. The VFX_DUR_* constants are effects that are applied with durations.

For this effect, the GetEffectInteger routine will return the visual effect identifier for the first integer value.

Area of Effect

Area of effect objects are invisible instances that are used to assign an effect to a region. The following routines can be used to create and query these objects.

```
effect EffectAreaOfEffect(
          int nAreaEffectID,
          string sOnEnterScript = "",
          string sOnHeartbeatScript = "",
          string sOnExitScript = "",
          string sEffectTag = "" )
```

This creates an effect object in the area of the target creature, which remains at the location it was applied. Valid values for *nAreaEffectID* are the AOE_... constant values. If strings are passed into any of the *sOnEnterScript*, *sOnHeartbeatScript* and *sOnExitScript* fields, then the scripts with these names will be used as event handling scripts by the effect object. Otherwise, the default values for those fields are used, as defined in the 'vfx_persistent.2da' file, with the row matching the value of nAreaEffectID.[2] The *sEffectTag* can be passed to the object, allowing it to be referenced from a script via it's tag.

For a spell effect, the convention for the script names is to use the spell impact script name (or something similar to it) followed by "a", "b" and "c", respectively. See for example, the 'nw_s0_bladebar', 'nw_s0_acidfog', and 'nw_s0_silence' AOE scripts.

```
object GetAreaOfEffectCreator(
          object oAoEObject = OBJECT_SELF )
```

If *oAoEObject* is an area of effect object, this will return the creating object.

```
int GetAreaOfEffectDuration(
          object oAoEObject = OBJECT_SELF )
```

If *oAoEObject* is an area of effect object, this will return it's duration type DURATION_TYPE....

2 Note that the name in the ENTRY column is similar to the AOE_* constants, except that the 'AOE' prefix is replaced by 'VFX' in the LABEL column.

```
int GetAreaOfEffectSpellId(
          object oAoEObject = OBJECT_SELF )
```

If *oAoEObject* is an area of effect object, this will return the spell identifier that created the object.

```
object GetFirstInPersistentObject(
          object oPersistentObject =
            OBJECT_SELF,
          int nResidentObjectType =
            OBJECT_TYPE_CREATURE,
          int nPersistentZone =
            PERSISTENT_ZONE_ACTIVE )
```

This will return the first object within the boundary of the object *oPersistentObject* that matches the object type *nResidentObjectType*, which is an OBJECT_TYPE_... constant. Typically this is used for an Area of Effect object. An example of a persistent object is a trigger, so you can use this to return the first object in a trigger. The parameter *nPersistentZone* parameter is no longer used.

```
object GetNextInPersistentObject(
          object oPersistentObject =
            OBJECT_SELF,
          int nResidentObjectType =
            OBJECT_TYPE_CREATURE,
          int nPersistentZone =
           PERSISTENT_ZONE_ACTIVE )
```

Each time this is called, it returns the next object within the boundaries of the persistent object *oPersistentObject* that matches the type *nResidentObjectType*, which is an OBJECT_TYPE_... constant. This list is initialized by a call to GetFirstInPersistentObject using the same parameters. The parameter *nPersistentZone* parameter is no longer used.

Cutscene Effects

The following effects are intended for use with cutscenes, and they can not be resisted.

```
effect EffectCutsceneDominated()
```

The subject of this effect becomes dominated.

```
effect EffectCutsceneGhost()
```

The effect subject is able to walk through other creatures without collision.

```
effect EffectCutsceneImmobilize()
```

This will paralyze the legs of the effect target, preventing from walking. The subject is otherwise unaffected. Example: 'nw_s0_rejuvcocoon'.

```
effect EffectCutsceneParalyze()
```

This will automatically paralyze the subject creature.

Environment

Although the patched toolset provides fields for setting the weather conditions in an area, currently the weather routines appear to have no ability to change the settings. The settings are fixed at whatever they are set to during module save.

```
int GetWeather(
          object oArea,
          int nWeatherType )
```

This call returns the weather settings for the area *oArea*. The parameter *nWeatherType* is a WEATHER_TYPE_... constant to be queried. The function returns a constant of type WEATHER_POWER_....

```
void ResetNWN2Fog(
          object oTarget,
          int nFogType )
```

If the fog has been modified, this call will restore the fog settings to the initial value set for the area oTarget, or for all areas of oTarget is the module. The nFogType parameter is one of the FOG_TYPE_... constants that corresponds to one of the Day/Night Cycle Stage array indices. See the SetNWN2Fog command.

```
void SetFog(
          object oTarget,
          int nFogType,
          int nColor,
          float fFogStart,
          float fFogEnd,
          float fFarClipPlaneDistance )
```

See the SetNWN2Fog command.

```
void SetNWN2Fog(
          object oTarget,
          int nFogType,
          int nColor,
          float fFogStart,
          float fFogEnd )
```

This command is intended to change the fog settings for an area *oTarget*, or for all areas if *oTarget* is the module. The change can be reverted using the ResetNWN2Fog command. Note: when I tested this, the new fog settings only came into effect after I re-entered the area.

The *nFogType* is a constant FOG_TYPE_... that corresponds to one of the Day/Night Cycle Stage array indices. Thus, FOG_TYPE_NWN2_SUNRISE sets the fog for the sunrise time slot. The *nColor* is an integer that encodes a red-green-blue color setting. The fog will begin at the distance *fFogStart*, reaches maximum color at the distance *fFogEnd* and clips sight beyond *fFarClipPlaneDistance*.

Note that you can use the HexStringToInt function in ginc_math to convert a six-character hex string color code to an integer value suitable for passing into *nColor*.

The documentation for this command is identical to SetFog, except the latter list a *fFarClipPlaneDistance* parameter, which is the distance where sight is clipped.

```
void SetWeather(
          object oTarget,
          int nWeatherType,
          int nPower =
            WEATHER_POWER_MEDIUM )
```

This routine is intended to set the weather in an area *oTarget*. If *oTarget* is a module, the notes state it will set the weather on all areas in that module. The *nWeatherType* argument is a WEATHER_TYPE_... global constant, while *nPower* is a constant WEATHER_POWER_....

Lighting

```
void SetLightActive(
          object oLight,
          int bActive )
```

If *oLight* is a Light object, this will set the active state to the boolean *bActive*. Unfortunately there is not an equivalent GetLightActive call, so if you want to use this command to manage the state of the Light placeables in an area, you will need to store the state of the Light as a local boolean variable then update the variable with each call.

Note that a known bug causes the 'Tag' strings for 'Light' placeables not to be retained in the save file. This can be a problem for scripts that need to control lights with specific tags in an area using the SetLightActive call (across multiple game sessions). A work-around is to use the SetLocalObject command to store 'Light' placeables as local Object variables on the Area. These variables are stored during a save, so when the game is reloaded these variables can be used to access the lights.

The following calls made no appreciable change that I could detect.

- GetTileMainLight1Color
- GetTileMainLight2Color
- GetTileSourceLight1Color
- GetTileSourceLight2Color
- RecomputeStaticLighting
- SetTileMainLightColor
- SetTileSourceLightColor

Music

The following routines are used to manage the music tracks played in an area. The individual tracks are row numbers in the 'ambientmusic.2da' file. Thus, for example, the daytime music menu pick 'NWN – Aarin Gend' corresponds to row 53: 'mus_theme_argend'.

```
void MusicBackgroundChangeDay(
          object oArea,
          int nTrack )
```

This changes the daytime background music track for the area *oArea* to *nTrack*.

```
void MusicBackgroundChangeNight(
          object oArea,
          int nTrack )
```

This changes the night time background music track for the area *oArea* to *nTrack*.

```
int MusicBackgroundGetBattleTrack(
          object oArea )
```

This call returns the current battle track number for the area *oArea*.

```
int MusicBackgroundGetDayTrack(
          object oArea )
```

This call returns the current daytime track number for the area *oArea*.

```
int MusicBackgroundGetNightTrack(
          object oArea )
```

This call returns the current night time track number for the area *oArea*.

```
void MusicBackgroundPlay(
          object oArea )
```

This call tells the game to play the background music for the area *oArea*.

```
void MusicBackgroundSetDelay(
          object oArea,
          int nDelay )
```

For area *oArea*, this sets the delay *nDelay* in milliseconds between the end of a background music track and the time it begins playing the track.

```
void MusicBackgroundStop(
          object oArea )
```

Stop playing the background music for area *oArea*.

```
void MusicBattleChange(
          object oArea,
          int nTrack )
```

This changes the battle music for the area *oArea* to the track *nTrack*.

```
void MusicBattlePlay(
          object oArea )
```

This causes the current battle music for the area *oArea* to be played.

```
void MusicBattleStop(
          object oArea )
```

This stops the battle music for the area *oArea*.

Sounds

```
void AmbientSoundChangeDay(
          object oArea,
          int nTrack )
```

The ambient daytime track for area *oArea* is changed to *nTrack*. The tracks are listed in the 'ambientsound.2da' file.

```
void AmbientSoundChangeNight(
          object oArea,
          int nTrack )
```

The ambient night time track for area *oArea* is changed to *nTrack*.

```
void AmbientSoundPlay( object oArea )
```

This turns on the ambient sound for the area *oArea*.

```
void AmbientSoundSetDayVolume(
          object oArea,
          int nVolume )
```

This sets the ambient daytime track volume in area *oArea* to *nVolume*. Unlike the setting in the area properties, valid values for *nVolume* are zero to 100.

```
void AmbientSoundSetNightVolume(
          object oArea,
          int nVolume )
```

This sets the ambient daytime track volume in area *oArea* to *nVolume*. Valid values for *nVolume* are zero to 100.

```
void AmbientSoundStop( object oArea )
```

This turns off the ambient sound for the area *oArea*.

```
float GetDialogSoundLength(
          int nStrRef )
```

For a sound used for dialogue, *nStrRef*, this will return the length of the wave file in seconds.

```
int GetSoundFileDurationLength(
          string sSoundFile )
```

This will return the number of milliseconds needed to play the sound file sSoundFile. If the file is invalid, this will return 0.

```
float GetStrRefSoundDuration(
          int nStrRef )
```

This will return the number of seconds needed to play the sound file identified by string reference *nStrRef*. If the identifier is invalid, this will return 0.

```
void PlaySound(
          string sSoundName,
          int bPlayAs2D = FALSE )
```

This should cause the sound *sSoundName* to be played from the location of the object that calls it. If *bPlayAs2D* is true, this will play the sound two-dimensionally, rather than using 3D positional audio.

```
void PlaySoundByStrRef(
          int nStrRef,
          int nRunAsAction = TRUE )
```

If there is a sound associated with the string reference *nStrRef*, this will play it from the position of the object that calls it. If the *nRunAsAction* is set to false, the sound is played instantly rather than as a queued action.

```
void SetSoundSet(
          object oCreature,
          int nSoundSet )
```

This routine changes the sound set for the creature *oCreature* to the row *nSoundSet* in the 'soundset.2da' file. These rows correspond to the items in the Sound Set property menu for a creature blueprint.

```
void SoundObjectPlay( object oSound )
```

If *oSound* is a sound object, this routine will cause the object to play its sounds. See SoundObjectStop.

```
void SoundObjectSetPosition(
          object oSound,
          vector vPosition )
```

This routine changes the location of the sound *oSound* to the vector location *vPosition* in the current area.

```
void SoundObjectSetVolume(
          object oSound,
          int nVolume )
```

This call sets the volume of the sound object *oSound* to the value *nVolume*, which should be an integer between 0 and 127.

```
void SoundObjectStop( object oSound )
```

This causes the sound object *oSound* to cease playing its sounds. See SoundObjectPlay.

Events and Scripts

Events are conditions in the game that can trigger a compiled script listed in the Scripts properties of an object. These scripts are referred to as event handlers. The event type is associated with an identifier, with a value matching one of the following constant types as appropriate for the object type:

- CREATURE_SCRIPT_ON_...
- SCRIPT_AOE_ON_...
- SCRIPT_AREA_ON_...
- SCRIPT_DOOR_ON_...
- SCRIPT_ENCOUNTER_ON_...
- SCRIPT_MODULE_ON_...
- SCRIPT_PLACEABLE_ON_...
- SCRIPT_STORE_ON_...
- SCRIPT_TRIGGER_ON_...

However, some of the script types are missing constants. Thus, there is no constant for a creature's 'On Perception' or 'On Conversation' property fields.

```
void ExecuteScript(
          string sScript,
          object oTarget )
```

This call causes the object *oTarget* execute the *sScript*, then return to the current script. The script must have been compiled in the module or else nothing will happen.

```
int GetCustomHeartbeat( object oTarget )
```

If a custom heartbeat was set on the object *oTarget* using the SetCustomHeartbeat routine, this will return the heartbeat interval as a number of milliseconds. If no custom heartbeat has been set, it will return 0.

```
void SpawnScriptDebugger()
```

If the game is configured to run the Script Debugger, this call will trigger the debugger. For more details, see the Debugging section in the first volume.

```
void SetCustomHeartbeat(
          object oTarget,
          int nMSeconds )
```

Normally the heartbeat interval is 6 seconds. This routine will change the heartbeat interval for the object *oTarget* to *nMSeconds* milliseconds. The target must be a creature or placeable for this to have any effect. For creatures, a random time interval of up to a second is added to the allotted heartbeat interval. The command notes point out that applying this to many objects will have an impact on game performance.

Events

```
event EventActivateItem(
          object oItem,
          location locTarget,
          object oTarget = OBJECT_INVALID )
```

This returns an event where item *oItem* is activated with the target location *locTarget* and/or target object *oTarget*. The event must be triggered via SignalEvent.

```
event EventConversation()
```

This returns an event that triggers the conversation script by the calling object. The event must be triggered via SignalEvent.

```
event EventSpellCastAt(
          object oCaster,
          int nSpell,
          int bHarmful = TRUE )
```

This call returns an event that will cause the 'Spell Cast At' script to run. The caster is set to *oCaster* and the spell *nSpell*, which is a SPELL_... constant. If *bHarmful* is true, the spell is considered harmful. See the Spell section below. The event must be triggered via SignalEvent.

```
event EventUserDefined(
          int nUserDefinedEventNumber )
```

Thus routine generates a 'User Defined Event' triggering event with the type set to *nUserDefinedEventNumber*. The event must be triggered via SignalEvent. See the GetUserDefinedEventNumber routine.

```
string GetEventHandler(
          object oObject,
          int nEventID )
```

For a valid event identifier *nEventID*, this call will return the name of the script that the object *oObject* uses to handle events of that type. See SetEventHandler.

```
void SetCreatureScriptsToSet(
          object oCreature,
          int nScriptSet )
```

The creature event handler scripts are organized into sets within the 'NWN2_ScriptSets.2da' file. This will replace the creature *oCreature*'s default script with the set on the row *nScriptSet* of the 2DA file, which is equal to a constant SCRIPTSET_*. These scripts will only be used when the creature would normally run it's default scripts. Thus they would not be run when the creature is possessed or dominated.

Pass SCRIPTSET_PLAYER_DEFAULT in *nScriptSet* for party roster members.

```
void SetEventHandler(
          object oObject,
          int nEventID,
          string sScriptName )
```

This will replace an event handler script of the object *oObject* with the compiled script *sScriptName*. The script to be replaced is determined by the *nEventID* parameter. See GetEventHandler.

```
void SignalEvent(
          object oObject,
          event evToRun )
```

This causes the object *oObject* to run the event *evToRun*.

Script Subroutines

```
int FiredFromPartyTransition()
```

When executed from an 'On Client Enter' event handling script, this will return true only if a party entered the area via the JumpPartyToArea routine.

```
object GetClickingObject()
```

This call can be used in an 'On Left Click' or 'On Click' script to return the object performing the click. This is identical to the GetEnteringObject call.

```
object GetFirstEnteringPC()
```

This can be called in an 'On Client Enter' script to return the first player character to enter the area, sub-area or module.

```
object GetInventoryDisturbItem()
```

This returns the object that caused the 'On Inventory Disturbed' script of the caller to be run, such as during a pick pocket attempt.

```
int GetInventoryDisturbType()
```

This returns a INVENTORY_DISTURB_... constant that gives the type of disturbance that caused the caller's 'On Inventory Disturbed' script to run.

```
int GetIsPartyTransition( object oObject )
```

If *oObject* is a transition object, this will return true if it has the 'Party Transition?' property set to true.

```
int GetLastRestEventType()
```

When called from a module's 'On Player Respawn' script, this will return the type of the triggering rest event, as a constant REST_EVENTTYPE_REST_....

```
string GetMatchedSubstring(
          int nString )
```

When called from an 'On Conversation' script, this will return a substring matching the index *nString*. The total matches is returned by GetMatchedSubstringsCount. The listening patterns are established with a SetListenPattern call using wildcards. A double-asterisk wildcard will match any pattern.

```
int GetMatchedSubstringsCount()
```

In an 'On Conversation' script, this will return the number of substrings that matched a substring listening pattern.

```
object GetNextEnteringPC()
```

Each time this is called in an 'On Client Enter' script, it will return the next player character to enter the area, sub-area or module. The list is reset by GetFirstEnteringPC.

```
int GetUserDefinedEventNumber()
```

When called in a 'On User Defined Event Script', this will return the event number associated with the script call.

Perception

See also GetListenPatternNumber and GetLastPerceived.

```
int GetLastPerceptionHeard()
```

When called from an 'On Perception' script, this will return true if the object that triggered the script has been heard.

```
int GetLastPerceptionInaudible()
```

When called from an 'On Perception' script, this will return true if the object that triggered the script has become inaudible.

```
int GetLastPerceptionSeen()
```

When called from an 'On Perception' script, this will return true if the object that triggered the script was seen.

```
int GetLastPerceptionVanished()
```

When called from an 'On Perception' script, this will return true if the object that triggered the script has vanished from sight.

Spell

The following functions can be used in 'Spell Cast At' event scripts. See also GetLastSpellCaster and the Talents section.

```
int GetLastSpell()
```

This returns the identifier of the spell that was cast. The result is a SPELL_... constant.

```
int GetLastSpellCastClass()
```

This call returns the character class of the spell caster that cast the spell. The result is a CLASS_TYPE_... constant.

```
int GetLastSpellHarmful()
```

This returns true only if the spell that triggered the event was harmful in nature.

```
int GetMetaMagicFeat()
```

When executed in a 'Spell Cast At' script, this will return a constant METAMAGIC_... that gives the meta-magic type of the spell.

Triggering Objects

```
object GetEnteringObject()
```

This call can be used in an 'On Exit' script property script to return the object entering the calling sub-area.

```
object GetExitingObject()
```

When called from an 'On Exit' script, this will return the object that left the sub-area.

```
object GetLastAttacker(
          object oAttackee = OBJECT_SELF )
```

When called from the 'On ... Attacked' script of the creature, placeable or door *oAttackee*, this will return the object that instigated the attack.

```
object GetLastClosedBy()
```

When called from an 'On Closed' script for a door or placeable, this will return the object that closed the door or placeable.

```
object GetLastDisarmed()
```

When called from a trap script, this will return the last object to disarm the trap.

```
object GetLastDisturbed()
```

When called from the event handler script of an object with an inventory, this will retrieve the last object to disturb the inventory (such as by a pick pocket skill use).

```
object GetLastKiller()
```

This returns the last object to kill the calling object. It could be called, for example, from the 'On Death' script.

```
object GetLastLocked()
```

When called from the script of a door or placeable, this will return the object that last locked the calling object.

```
object GetLastOpenedBy()
```

When called from the script of a door or placeable, this will return the object that last opened the calling object.

```
object GetLastPerceived()
```

When run from an 'On Perception' script, this will return the object that triggered the event.

```
object GetLastPlayerDied()
```

When run from a module's 'On Player Death' script, this will return the PC that triggered the event.

```
object GetLastPlayerDying()
```

When run from a module's 'On Player Dying' script, this will return the PC that triggered the event.

```
object GetLastRespawnButtonPresser()
```

When run from a module's 'On Player Respawn' script, this will return the PC of the player that triggered the event.

```
object GetLastSpellCaster()
```

When run from a 'Spell Cast At' script, this will return the object that cast the spell. Spells can be cast by creatures, doors and placeables.

```
object GetLastUnlocked()
```

In an Unlock script for a door or placeable, this will return the object that unlocked the lock.

```
object GetPCLevellingUp()
```

In the module's 'On Player Level Up' script, this will return the PC that is levelling up.

```
object GetPlaceableLastClickedBy()
```

In a placeable's 'On Left Click' script, this will return the player-controlled object that triggered the event.

Factions

Each creature belongs to a faction, which is comparable to an alliance. Each member of a faction shares a common attitude toward members of other factions. Thus creatures in the hostile faction are allied with each other and generally dislike members of other factions. These routines are used to control creature membership in factions and the attitudes of factions toward creatures.

Management

```
void ChangeFaction(
          object oTarget,
          object oMemberOfFaction )
```

This routine will cause the NPC *oTarget* to change it's faction to the same as the NPC *oMemberOfFaction*.

```
void ChangeToStandardFaction(
          object oTarget,
          int nStandardFaction )
```

This call will cause the NPC *oTarget* to join the faction *nStandardFaction*, which must be set to one of the STANDARD_FACTION_... constants.

```
void SetFactionLeader(
          object oNewLeader )
```

If *oNewLeader* is a creature, this routine makes it the leader of it's faction. This creature becomes the speaker during a conversation.

Party

A player's party can consist of henchmen, NPC companions and one or more player-run characters, or PCs. The companions are selected from a roster of NPCs by the player, with each roster member being identified by a unique 10-character name. Each PC is owned by a separate player, so there is only one PC in a single-player game.

See also JumpPartyToArea, SetCreatureScriptsToSet, the Associates section and the "ginc_companion" include file.

```
int AddRosterMemberByCharacter(
          string sRosterName,
          object oCharacter )
```

This call will add the existing NPC *oCharacter* to the roster of characters that can be added to the player's party. The string *sRosterName* is a unique 10-character name that is used to refer to this character in the other roster calls. The routine returns true if the addition was successful, or false on an error. See also AddRosterMemberToParty.

```
int AddRosterMemberByTemplate(
          string sRosterName,
          string sTemplate )
```

This call will add the NPC identified by template sTemplate to the roster of characters that can be added to the player's party. The NPC needs to be created in order to appear in the game world. The string *sRosterName* is a unique 10-character name that is used to refer to this character in the other roster calls. The routine returns true if the addition was successful, or false on an error. See also AddRosterMemberToParty.

```
int AddRosterMemberToParty(
          string sRosterName,
          object oPC )
```

This will add the NPC with the roster name *sRosterName* to the party belonging to the player character *oPC*. This function will fail and return false if the roster member already belongs to another party. You can test for availability using GetIsRosterMemberAvailable. Note that this call is not restricted to the number of NPCs returned by GetRosterNPCPartyLimit.

If the added roster member is in the current module, it will be transported to a location near the PC. Otherwise it will be loaded in using the last saved condition. The faction of the new party member will be set to match the PC's faction.

```
void AddToParty(
          object oPC,
          object oPartyLeader )
```

In a multi-player game, this will add the player character *oPC* to the party being led by *oPartyLeader*. Only a maximum of two PCs can be added by this call.

```
int DespawnRosterMember(
          string sRosterName )
```

The roster member *sRosterName* is saved to preserve any changes, then is removed from the game. The command notes indicate this is the only routine that should be used to remove roster members from the game. To respawn a roster member, use the SpawnRosterMember routine.

```
object GetFirstPC( int iOwnedPC = TRUE )
```

If *iOwnedPC* is true, GetFirstPC returns the owned PC in the first player's list. Otherwise, it returns the first controlled object in the player's list. See also GetNextPC.

```
string GetFirstRosterMember()
```

This returns a string containing the roster name of the first member of the party roster. See GetNextRosterMember.

```
int GetIsRosterMember( object oMember )
```

If the object *oMember* is a roster member, this routine will returns true.

```
int GetIsRosterMemberAvailable(
          string sRosterName )
```

This function returns true only if the member with roster name *sRosterName* is available to join the party. It will return false if the member is claimed by another party or if the member is unavailable.

```
int GetIsRosterMemberCampaignNPC(
          string sRosterName )
```

A campaign NPC is persistent across modules, but can not be chosen as a party member by the player on the party selection user interface. However, a campaign NPC can be added to the party by a script. This routine will return true only if the NPC with the roster name *sRostername*, is a campaign NPC. See SetIsRosterMemberCampaignNPC.

```
int GetIsRosterMemberSelectable(
          string sRosterName )
```

This function returns true if the member *sRosterName* is selectable. It may return false if the member is not selectable for plot reasons, and so forth.

```
object GetNextPC( int bOwnedPC = TRUE )
```

Each call to GetNextPC will return an additional PC or controlled object in the list, depending on whether *bOwnedPC* is true or false. Once the list is exhausted, GetNextPC returns OBJECT_INVALID. The list is reset by a call to GetFirstPC.

```
string GetNextRosterMember()
```

Each time this is called, it returns a string containing the roster name of the next member of the party roster. The list can be reset by a call to GetFirstRosterMember.

```
object GetObjectFromRosterName(
          string sRosterName )
```

When passed a string containing the roster name *sRosterName* of a valid roster member, this will return the game world object that currently represents the roster member.

```
int GetPartyMembersDyingFlag()
```

This returns the campaign boolean 'Party Members Dying'.

```
string GetPartyMotto()
```

This call fetches the motto for the party calling this routine.

```
string GetPartyName()
```

This call returns the name of the party calling this routine.

```
object GetPCSpeaker()
```

Get the PC that is involved in a conversation. Returns OBJECT_INVALID on error.

```
string GetRosterNameFromObject(
          object oCompanion )
```

Returns a string with the roster name of the companion *oCompanion* from the roster list. If there is no match, an empty string is returned.

```
int GetRosterNPCPartyLimit()
```

This returns the number of roster NPCs that a player can add to the party using the party selection screen. This value can be modified using SetRosterNPCPartyLimit.

```
void RemoveFromParty( object oPC )
```

In a multi-p-layer game, this will remove a player character *oPC* from the party. See AddToParty.

```
int RemoveRosterMember( string sRosterName )
```

If *sRosterName* is the valid roster name of an NPC, this routine will remove the name from the roster of selectable NPCs and return true.

```
void RemoveRosterMemberFromParty(
          string sRosterName,
          object oPC,
          int bDespawnNPC = TRUE )
```

If a valid NPC with the *sRosterName* is currently a member of the party containing the player character *oPC*, this will detach the NPC from the party. If *bDespawnNPC* is true, the state of the NPC will be saved and it will be despawned.

```
int SetIsRosterMemberCampaignNPC(
          string sRosterName,
          int bCampaignNPC )
```

A campaign NPC is persistent across modules, but can not be chosen as a party member by the player on the party selection user interface. However, a campaign NPC can be added to the party by a script. This routine will mark or clear the campaign NPC status of the NPC with the roster name *sRostername*, depending on whether *bCampaignNPC* is true or false.

```
int SetIsRosterMemberSelectable(
          string sRosterName,
          int bSelectable )
```

Roster members can be marked as non-selectable, which prevents them from being selected via the party selection user interface by the player. This routine will change the selectability of the NPC with the roster name *sRosterName* to the state *bSelectable*. It will return true only if a matching roster member was found. Non-selectable NPCs that are already in the party can not be removed by the player.

```
void SetRosterNPCPartyLimit(
          int nLimit )
```

By default, a player can add up to three NPCs to a party using the party selection user interface. This routine will change the maximum number of selectable NPCs to *nLimit*.

```
object SpawnRosterMember(
          string sRosterName,
          location locAt )
```

This call will find (or create) an instance of the NPC with the roster name *sRosterName*. If successful, the NPC will appear at the location *locAt* and the function will return the object representing the NPC. Otherwise, the function will return an invalid object.

Query

In the following calls, *bMustBeVisible* is undocumented but it presumably restricts the search to visible creatures when true.

```
int GetFactionAverageGoodEvilAlignment(
          object oFactionMember )
```

This routine computes the average alignment, along the good-evil axis, of the faction containing *oFactionMember*. The result is an integer that ranges from 0 (evil) to 100 (good).

```
int GetFactionAverageLawChaosAlignment(
          object oFactionMember )
```

This routine computes the average alignment, along the law-chaos axis, of the faction containing *oFactionMember*. The result is an integer that ranges from 0 (chaotic) to 100 (lawful).

```
int GetFactionAverageLevel(
          object oFactionMember )
```

This function computes the average class level of the faction containing *oFactionMember*.

```
int GetFactionAverageXP(
          object oFactionMember )
```

This call returns the average number of experience points for the members of the faction containing *oFactionMember*.

```
object GetFactionBestAC(
          object oFactionMember =
            OBJECT_SELF,
          int bMustBeVisible = TRUE )
```

This routine returns the creature with the highest armor class rating among the faction containing *oFactionMember*. The *bMustBeVisible* is undocumented, but it presumably restricts the search to visible creatures when true.

```
int GetFactionEqual(
          object oFirstObject,
          object oSecondObject =
            OBJECT_SELF )
```

If the objects *oFirstObject* and *oSecondObject* belong to the same faction, then this routine will return true.

```
int GetFactionGold(
          object oFactionMember )
```

This call is used to determine the total gold being held by the faction that contains *oFactionMember*.

```
object GetFactionLeader(
          object oFactionMember )
```

This call returns the object that is the current leader of the faction containing *oFactionMember*. At present this only works with a party, and it returns an invalid object for other factions. The party leader is able to choose who to remove from the party.

```
object GetFactionLeastDamagedMember(
          object oFactionMember =
            OBJECT_SELF,
          int bMustBeVisible = TRUE )
```

This routine returns the creature with the least amount of damage among the faction containing *oFactionMember*.

```
object GetFactionMostDamagedMember(
          object oFactionMember =
            OBJECT_SELF,
          int bMustBeVisible = TRUE )
```

This is similar to GetFactionLeastDamagedMember, except the most damaged member of the faction containing *oFactionMember* is returned.

```
int GetFactionMostFrequentClass(
          object oFactionMember )
```

This call returns a value of type CLASS_TYPE_... that gives the most common character class type among the members of the faction containing *oFactionMember*.

```
object GetFactionStrongestMember(
          object oFactionMember =
            OBJECT_SELF,
          int bMustBeVisible = TRUE )
```

This call returns the object that is the strongest member of the faction containing *oFactionMember*. This is based on class level rather than the Strength attribute.

```
object GetFactionWeakestMember(
          object oFactionMember =
            OBJECT_SELF,
          int bMustBeVisible = TRUE )
```

This call returns the object that is the weakest member of the faction containing *oFactionMember*.

```
object GetFactionWorstAC(
          object oFactionMember =
            OBJECT_SELF,
          int bMustBeVisible = TRUE )
```

This routine returns the creature with the lowest armor class rating among the faction containing *oFactionMember*.

```
object GetFirstFactionMember(
          object oFactionMember =
            OBJECT_SELF,
          int bPCOnly = TRUE )
```

This returns the first member in the list of creatures of the faction that contains *oFactionMember*. If *bPCOnly* is true, this will return the first PC in the faction. To find other faction members, see the GetNextFactionMember routine.

```
object GetNextFactionMember(
          object oFactionMember =
            OBJECT_SELF,
          int bPCOnly = TRUE )
```

This returns the another member in the list of creatures of the faction that contains *oFactionMember*. If *bPCOnly* is true, this will return the first PC in the faction. This list is reset by a call to GetFirstFactionMember.

Reputation

The reputation of a creature in a faction is represented by an integer value between 0 and 100, with high values indicating a more favorable reputation. A low reputation can result in hostility and combat. See the Reaction section.

```
void AdjustReputation(
          object oTarget,
          object oFactionMember,
          int nAdjustment )
```

The attitude of the faction containing *oFactionMember* toward *oTarget* is adjusted by *nAdjustment*.

```
void ClearPersonalReputation(
          object oTarget,
          object oSource = OBJECT_SELF )
```

If the creature *oTarget* has gained a personal reputation with *oSource*, this routine will clear the settings.

```
int GetFactionAverageReputation(
          object oSourceFactionMember,
          object oTarget )
```

This call returns the average reputation of the creature *oTarget* in the faction containing *oSourceFactionMember*.

```
int GetReputation(
          object oSource,
          object oTarget )
```

This call will return an integer value that represents how favorably (or unfavorably) object *oSource* views object *oTarget*. Values from 0 to 10 represent hostility, while 90 to 100 represent friendliness. Values ranging from 11 to 89 indicate a neutral reputation. If either object is invalid, this call will return -1.

```
int GetStandardFactionReputation(
          int nStandardFaction,
          object oCreature = OBJECT_SELF )
```

This call returns the average reputation of the creature *oCreature* in the standard faction *nStandardFaction*, which is a STANDARD_FACTION_... constant.

```
void SetStandardFactionReputation(
          int nStandardFaction,
          int nNewReputation,
          object oCreature = OBJECT_SELF )
```

This call changes the reputation of the creature *oCreature* in the standard faction *nStandardFaction* to the value *nNewReputation*. The reputation is an integer between 0 and 100, inclusive, and the faction is a global constant of the form STANDARD_FACTION_....

Game Management

```
int dN( int nNumDice = 1 )
```

Returns the total from rolling an *N*-sided dice *nNumDice* times, where *N* is 2, 3, 4, 6, 8, 10, 12, 20 or 100. Thus, a call of d20(3) returns the total of three rolls of a 20-sided dice.

```
void ActivatePortal(
          object oTarget,
          string sIPAddress = "",
          string sPassword = "",
          string sWaypointTag = "",
          int bSeamless = FALSE )
```

This call attempts to send the object *oTarget* to the server with the IP address *sIPAddress* using the login password *sPassword*. Alternatively the IP address can be a alphanumeric resource location, and can include a port number. If the move is successful, the object will appear at the waypoint with the tag *sWaypointTag*. Normally the client will receive an information panel, but this can be turned off if *bSeamless* is set to true.

```
void BootPC( object oPlayer )
```

This "boots" the player *oPlayer* from the server.

```
void EndGame( string sEndMovie )
```

This call ends the current game by playing the movie file *sEndMovie* then returns all players to the menu interface.

```
int GetGameDifficulty()
```

This returns a value giving the game difficulty as a constant value GAME_DIFFICULTY_....

```
int GetIsSinglePlayer()
```

If this game is for a single player only, this returns true.

```
int GetOnePartyMode()
```

If this game is running in one party mode, this call will return true.

```
int GetPause()
```

If this returns true, then the game is currently paused.

```
string GetPCIPAddress( object oPlayer )
```

For a network name, this call will return the IP address of the client from which the player *oPlayer* is connecting to the server.

```
string GetPCPlayerName( object oPlayer )
```

This returns the name of the object *oPlayer*.

```
string GetPCPublicCDKey( object oPlayer )
```

This returns the public portion of the CD authentication key that the player *oPlayer* used to log on to a game.

```
object GetPrimaryPlayer()
```

In a multi-player game, the primary player is the player who is hosting the game. This routine will return the object representing the primary player. Note that a game may not have a primary player, such as when the hosting player is participating only as the DM.

```
location GetStartingLocation()
```

This routine returns the starting location of the current module.

```
int GetTalkTableLanguage()
```

This call returns a LANGUAGE_... constant that identifies the language being used in the game's talk table file.

```
void SetCustomToken(
          int nCustomTokenNumber,
          string sTokenValue )
```

This call will assign the string value *sTokenValue* to each use of the token <CUSTOM#####>, where 'XXXXX' is the string representation of *nCustomTokenNumber*. This will cause instances of the token, such as in conversations or journal entries, to be automatically converted to the string value. For consistency, this call is best made in during module startup, prior to the first use of the token.

Custom token numbers 0-9 are used by Bioware, so higher numbers should be chosen. (It's probably best to use a three digit number to avoid conflicts.)

```
void SetCameraMode(
          object oPlayer,
          int nCameraMode )
```

For a player-controlled character *oPlayer*, this will set the current viewing mode to *nCameraMode*, which is a CAMERA_MODE_... constant. The player can immediately override this setting with a shift-mouse drag. Overuse of this command will probably prove annoying to the player. See also StoreCameraFacing.

```
void SetPause( int bState )
```

This can be used to set the pause state of the game to the boolean *bState*.

Commands

```
void ActionDoCommand( action aActionToDo )
```

This will add the action *aActionToDo* to the execution stack of the object making the call. Thus, if there are other actions in the stack, those will be executed first. This function differs from AssignCommand in that the subject can not be set; only the action.

```
void AssignCommand(
          object oSubject
          action aActionToAssign )
```

This call will assign the action *aActionToAssign* to the object *oSubject*. The action is appended to the current action stack for the object. If the *aActionToAssign* is a command, it is run with *oSubject* set to OBJECT_SELF. This is useful for executing certain commands against *oSubject* such as ClearAllActions. If *aActionToAssign* is null, no command is assigned.

```
void ClearAllActions(
          int nClearCombatState = FALSE )
```

This call will clear the action queue of the caller. Despite the void return value, this call can be inserted as an action in the AssignCommand call. If *nClearCombatState* is true, it will also clear the combat state on a creature. This is called by the ga_clear_actions script.

```
void DelayCommand(
          float fSeconds,
          action aDelayedAction )
```

This call will cause an action *aDelayedAction* to take effect after *fSeconds* have passed. The notes for this function recommend that effects should be passed to the *aDelayedAction* function call as a variable rather than as an effect function within the DelayCommand arguments.

The script notes mention that a DelayCommand call causes an effect to lose its spellID, allowing effects to stack. See 'x2_s1_suckbrain' for example.

```
void DismountObject(
          object oDismountingObject,
          object oObjectToDismount )
```

The object *oDismountingObject* dismounts itself from the object *oObjectToDismount*. See MountObject.

```
int GetCommandable(
          object oTarget = OBJECT_SELF )
```

Return true if the object *oTarget*'s action stack can be modified.

```
int GetCurrentAction(
          object oObject = OBJECT_SELF )
```

This will return a constant ACTION_... that gives the current action being executed by the object *oObject*.

```
int GetLastAssociateCommand(
          object oAssociate = OBJECT_SELF )
```

This call returns the last command issued to the associate *oAssociate*. See GetAssociate.

```
int GetNumActions( object oObject )
```

This will return the current number of actions that are queued up for the object *oObject* to execute.

```
void MountObject(
          object oMountingObject,
          object oObjectToMount )
```

The object *oMountingObject* dismounts itself from the object *oObjectToMount*. See DismountObject.

```
void SetCommandable(
          int bCommandable,
          object oTarget = OBJECT_SELF )
```

If *bCommandable* is true, allow the object *oTarget's* action stack to be modified. Otherwise, disallow modification of the action stack. This call can be passed as an action.

Two-Dimensional Arrays

These routines manage access to the two-dimensional array files, which is a NWN2 data file with a '.2da' suffix. The name of the file that is passed as an argument should match a file name from the 'Pick 2DA' browser. However, it should not include the '.2da' suffix.

```
void Clear2DACache(
          string s2DAname = "" )
```

When a 2DA file is queried, it will be loaded into a memory cache. This function will clear out the cache for the 2DA file *s2DAname*, or all 2DA files if the string is empty.

```
string Get2DAString(
          string s2DA,
          string sColumn,
          int nRow )
```

This will return the value from a cell in the 2DA file named *s2DA*. The cell is on the row matching *nRow* and the column with the name matching *sColumn*. The column names can be found by opening the 2DA file using the '2DA File...' pick from the View menu. If the cell does not exist or the 2DA file is not found, a null string is returned.

The notes for this routine recommend not calling this routine inside a loop; most likely for performance reasons.

```
int GetNum2DAColumns( string s2DAname )
```

If the 2DA file *s2DAname* is found, this call will return the number of columns and will cache the data (if it isn't already cached). Otherwise it will return -1.

```
int GetNum2DARows( string s2DAname )
```

If the 2DA file *s2DAname* is found, this call will return the number of rows and will cache the data (if it isn't already cached). Otherwise it will return -1.

File Updates

```
void DoSinglePlayerAutoSave()
```

If the current game is a single-player game, this will cause an auto-save.

```
void ExportAllCharacters()
```

All currently active characters of players in the game are saved to their corresponding repositories.

```
void ExportSingleCharacter(
          object oPlayer )
```

The character belonging to the player *oPlayer* is saved to it's repository.

```
void PrintFloat(
          float fFloat,
          int nWidth = 18,
          int nDecimals = 9 )
```

This call writes a formatted floating point value *fFloat* to the log file. The output value has a width of *nWidth* characters and *nDecimals* decimals.

```
void PrintInteger( int nInteger )
```

This function will write the integer *nInteger* as a string in the log file.

```
void PrintObject( object oObject )
```

This function prints the object identifier of object *oObject* to the log file.

```
void PrintString( string sString )
```

This will write the string *sString* to the log file.

```
void PrintVector(
          vector vVector,
          int bPrepend )
```

This call will print the vector *vVector* to the log file. If *bPrepend* is true, "PRINTVECTOR:" will be inserted before the vector.

```
void WriteTimestampedLogEntry(
          string sLogEntry )
```

This routine will append a time-stamped copy of the string *sLogEntry* to the log file.

Geometry

These include math functions and game area geometry calls.

```
float FeetToMeters( float fFeet )
```

A distance *fFeet*, in units of feet, is passed to this call, and it returns the equivalent distance in meters. There is no equivalent call to convert meters to feet, but you can multiply the number of meters by 3.28.

```
float GetDistanceBetween(
          object oObjectA,
          object oObjectB )
```

This returns the separation in metres of object *oObjectB* from object *oObjectA*. If either of the objects is invalid, this routine will return 0.0f.

```
float GetDistanceBetweenLocations(
          location locA,
          location locB )
```

This call will return the distance from location *locA* to location *locB*.

```
float GetDistanceToObject(
          object oObject )
```

This returns the range to the object *oObject* from the object making the call.

```
float GetFacing( object oTarget )
```

This returns the direction that a valid object *oTarget* is facing as a number of degrees anti-clockwise from due east.

```
float GetFacingFromLocation(
          location locAt )
```

This obtains the orientation information from the location *locAt*. The result is a value between 0.0 and 360.0, giving the number of degrees anti-clockwise from due east.

```
vector GetPositionFromLocation (
          location locAt )
```

Given a location *locAt*, this will return the vector component within the location's area.

```
void SetFacing(
          float fDirection,
          int bLockToThisOrientation =
            FALSE )
```

The object calling this routine will be turned to face the direction *fDirection*, which is the number of degrees anti-clockwise from due east. If this is called during a conversation and *bLockToThisOrientation*, then the facing of the calling object will be left at this facing for the remainder of the conversation.

```
void SetFacingPoint(
          vector vTarget,
          int bLockToThisOrientation =
            FALSE )
```

This functions like the SetFacing routine, except the calling object is turned in the direction of the vector *vTarget*. To face toward an object, use the GetPosition call to return the vector position of that object.

Position

```
vector AngleToVector( float fAngle )
```

This functions accepts an angle *fAngle* in degrees and converts it to a unit vector. This vector can then be scaled by multiplying it with a floating point scalar.

```
location CalcPointAwayFromPoint(
          location locPoint,
          location locAwayFromPoint,
          float fDistance,
          float fAngularVariance,
          int bComputeDistFromStart )
```

This routine can be used to generate a new location that is in the opposite direction of *locAwayFromPoint* from the perspective of *locPoint*. If *bComputeDistFromStart* is true, then the value of *fDistance* determined how far away the new location will be placed from *locPoint*, (Per the notes, this can be a negative value, so that location is in the opposite direction.) If *bComputeDistFromStart* is false, then *fDistance* is measured from *locAwayFromPoint*. The parameter *fAngularVariance* is a value in degrees between 0 to 180, and it applies a random angle to the direction.

```
location CalcSafeLocation(
          object oCreature,
          location locTest,
          float fSearchRadius,
          int bWalkStraightLineRequired,
          int bIgnoreTestLocation )
```

This routine will try to find a location near *locTest* where the creature *oCreature* can stand. This search is limited to the radius *fSearchRadius* around *locTest*. If the *bWalkStraightLineRequired* is true, then this routine will only return a location that the creature can walk to. If the *bIgnoreTestLocation* argument is true, then *locTest* will be excluded from the possible safe locations.

If this call is unsuccessful then the routine will instead return the current location of the creature *oCreature*.

```
location GetLocation( object oObject )
```

Returns a location describing the position of the object *oObject*.

```
location Location(
          object oArea,
          vector vPosition,
          float fOrientation )
```

This routine will generate a location in the area *oArea* that is offset from coordinates [0.0, 0.0, 0.0] by an amount that is specified by the vector *vPosition*. The *fOrientation* parameter is an angle in degree relative to North for the area. Thus a value of 180.0f will face South.

```
vector Vector(
          float x = 0.0f,
          float y = 0.0f,
          float z = 0.0f )
```

This represents a vector offset. The vector parameters *x*, *y* and *z* are equivalent to the values in the 'Position No Snap' field upon selecting an item or placeable.

```
float VectorMagnitude( vector vVector )
```

This returns the magnitude (length) of the vector *vVector*. If this is the vector between two locations, the magnitude is the distance between the locations.

```
vector VectorNormalize( vector vVector )
```

This converts the vector *vVector* to a vector of unit length. Multiplying the normalized vector times the magnitude of *vVector* will produce the original vector.

```
float VectorToAngle( vector vVector )
```

This routine converts the vector *vVector* to an angle.

Math

The math functions are accurate to about 6-8 digits, which should be good enough for scripting purposes.

```
int abs( int nValue )
```

Returns the absolute value of *nValue*. That is, if *nValue* is negative it returns -*nValue*; otherwise it returns *nValue*.

```
float acos( float fValue )
```

Returns the inverse cosine (arccosine) of *fValue*. That is, cos(acos(*fValue*)) = *fValue*. This function will return an angle in degrees. Valid values for *fValue* are in the range: $-1 \leq fValue \leq 1$. Any other input will cause acos to return 0.

```
float asin( float fValue )
```

Returns the inverse sine (arcsine) of *fValue*. That is, sin(asin(*fValue*)) = *fValue*. This function will return an angle in degrees. Valid values for *fValue* are in the range: $-1 \leq fValue \leq 1$. Any other input will cause asin to return 0.

```
float atan( float fValue )
```

Returns the inverse tangent (arctan) of *fValue*. That is tan(atan(*fValue*) = *fValue*. This function will return an angle in degrees.

```
float cos( float fValue )
```

Returns the cosine of *fValue*, which has a value in degrees.

```
float fabs( float fValue )
```

Returns the absolute value of *fValue*, That is, it returns *fValue* if it is positive, otherwise it returns -*fValue*.

```
float log( float fValue )
```

Returns the natural (base *e*) logarithm of *fValue*. For the base 10 logarithm, multiply by $\log_{10}(e) = 0.434294$.

```
float pow( float fValue, float fExponent )
```

Returns *fValue* raised to the power of *fExponent*. If both *fValue* and *fExponent* are zero, it returns zero.

```
float sin( float fValue )
```

Returns the sine of *fValue*, which has a value in degrees.

```
float sqrt( float fValue )
```

Returns the square root of *fValue*. This is equivalent to pow(*fValue*, 0.5f).

```
float tan( float fValue )
```

Returns the tangent of *fValue*, which has a value in degrees.

```
float YardsToMeters( float fYards )
```

Returns the equivalent number of metres for the distance *fYards* in yards.

Interaction

This section covers routines that manage the interaction between creatures and other objects, including combat and conversation.

```
int GetTurnResistanceHD(
          object oUndead = OBJECT_SELF )
```

If *oUndead* is a creature with an undead racial type, this call will return the number of hit dice the creature has for the purpose of resisting a turn undead effect.

Collision

```
int GetBumpState( object oCreature )
```

This returns the bump state of the creature *oCreature* as a value matching a BUMPSTATE_... constant. This determines whether the target will move aside when another creature is moving through the target's position. See SetBumpState.

```
int GetCollision(
          object oTarget )
```

This will get the collision state for a creature or placeable *oTarget*. See SetCollision.

```
int GetScriptHidden(
          object oCreature )
```

This call will return the state of the boolean 'Script Hidden' property for a valid creature *oCreature*.

```
void SetBumpState(
          object oCreature,
          int nBumpState )
```

This call will set the bump state property of a creature *oCreature* to the value *nBumpState*, which is a constant of the form BUMPSTATE_.... See GetBumpState.

```
void SetCollision(
          object oTarget,
          int bCollision )
```

If *oTarget* is a creature, this will set the collision boolean to *bCollision*. Creatures do not count for collision when they are set to Script Hidden.

```
void SetScriptHidden(
          object oCreature,
          int bHidden,
          int bDisableUI = TRUE )
```

This routine sets the 'Script Hidden' property of a creature. When *bHidden* is true, the creature *oCreature* will not render, do not count when determining collision and can not be selected by a player on a client. Otherwise the creature appears in the game and can be interacted with normally. If *bDisableUI* is true, the AI is disabled while the creature is hidden.

Conversation

```
int BeginConversation(
          string sResRef = “”,
          object oObjectToDialog =
            OBJECT_INVALID,
          int bPreventHello = FALSE )
```

This command begins a conversation. If a resource reference string *sResRef* is passed, it will be used for the conversation dialog. Otherwise the default dialog is used. The *oObjectToDialog* can be used to specify the creature that will own the conversation, or it will default to the creature that triggered the event. Setting *bPreventHello* will prevent the speaker from saying it's hello message.

```
int GetCanTalkToNonPlayerOwnedCreatures(
          object oObject )
```

This returns true only if the object *oObject* is able to hold a conversation with creatures that have not been created by a player. This is set by the 'Can Talk to Non-Player Owned Creatures?' property field.

```
string GetNodeSpeaker()
```

Within a Conditional script of a conversation, this will return the tag of the current speaker.

```
object GetLastSpeaker()
```

In an Condition or Action script of a conversation, this will return the object of the creature with whom the calling character is conversing.

```
object GetPCSpeaker()
```

In an Condition or Action script of a conversation, this returns the participating PC.

```
int IsInConversation(
          object oObject )
```

If the object *oObject* is currently in a conversation, this will return true.

```
int IsInMultiplayerConversation(
          object oObject )
```

If the object *oObject* is currently in a conversation that is flagged as multiplayer, this will return true.

```
void SetCanTalkToNonPlayerOwnedCreatures(
          object oObject,
          int bCanTalk )
```

This routine can be used to set the 'Can Talk to Non-Player-Owned Creatures?' property. If *bCanTalk* is false, then the object *oOwned* is only allowed to talk to creatures that are owned by a player. If a creature is controlled but not owned by a player, then the conversation is switched to the player-controlled creature.

```
void SpeakOneLinerConversation(
          string sDialogResRef = “”,
          object oTokenTarget =
            OBJECT_INVALID,
          int nTalkVolume =
            TALKVOLUME_TALK )
```

This causes the calling object to speak the single line conversation named *sDialogResRef*. The line will appear as a string above the creature that floats upwards and fades. If the conversation contains tokens (strings inside '<' and '>' character pairs), their values are inserted based on the creature *oTokenTarget*. The *nTalkVolume* parameter is a TALK_VOLUME_... constant that sets the conversation volume. This determines how close a listener needs to be in order to see the message.

Cutscene

The 'ginc_cutscene' has additional routines for managing conversation cutscenes. See also the ActionPauseCutscene and EffectCutscene... routines.

```
void AssignCutsceneActionToObject(
          object oObject,
          action aAction )
```

This passes the action *aAction* to the object *oObject*, along with a cutscene flag. An ActionPauseCutscene can pause a dialogue, and the conversation will not resume until all actions with cutscene flags have been completed.

```
float GetCutsceneCameraMoveRate(
          object oCreature )
```

This routine will return the movement rate of the camera within the cutscene for the creature *oCreature*. The result is a floating point value between 0.1 and 2.0.

```
object GetLastPCToCancelCutscene()
```

In a multi-player game, a player can cancel from a cutscene. This returns the last player-owned character to do this.

```
int GetNumCutsceneActionsPending()
```

This returns the number of cutscene actions currently queued up. The notes suggest using this for troubleshooting why a cutscene pause did not resume when expected.

```
void RestoreCameraFacing()
```

This will restore the camera to the facing that was stored into memory using StoreCameraFacing, then purge the facing information. If no camera facing information has been stored then nothing will happen.

```
void SetCameraFacing(
          float fDirection,
          float fDistance = -1.0f,
          float fPitch = -1.0f,
          int nTransitionType =
            CAMERA_TRANSITION_TYPE_SNAP )
```

This will turn the camera to a different facing. The ground surface is an x-y plane with positive x facing east and positive y pointing north. The direction *fDirection* is the number of degrees anti-clockwise from the east, so a value of 90.0f is due north. The *fDistance* is the number of meters away from the location being filmed. The angle of the camera to the ground is set via the *fPitch* value, with 1.0f being almost directly overhead and 89.0f being nearly parallel to the ground. The *nTransitionType* determines how rapidly the camera switches to the new setting from the previous scene, and is a CAMERA_TRANSITON_TYPE_... constant.

Note that passing in a value of -1.0 for distance or pitch will cause the camera to retain its previous settings.

```
void SetCameraHeight(
          object oPlayer,
          float fHeight = 0.0f )
```

This causes the camera to be at the height *fHeight* for the player *oPlayer*. A height of zero will restore the camera to the the player's default racial height.

```
void SetCutsceneCameraMoveRate(
          object oCreature,
          float fRate )
```

This routine will set the movement rate *fRate* of the camera within the cutscene for the creature *oCreature*. A valid movement rate is a value between 0.1 and 2.0.

```
void SetCutsceneMode(
          object oCreature,
          int nInCutScene = TRUE )
```

If *nInCutScene* is true, this places the creature into cutscene mode. Within cutscene, the player can not use GUI or camera controls. Passing in false for the *nInCutScene* paraeter removes the cutscene mode from the creature.

```
void SetLookAtTarget(
          object oObject,
          vector vTarget,
          int nType = 0 )
```

The notes for this routine say it is mainly intended to fix bugs in cutscenes that control where the creature *oObject* is looking. This causes the creature *oObject* to look along the vector *vTarget*. The *nType* parameter is not implemented.

```
void SetOrientOnDialog(
          object oCreature,
          int bActive )
```

By default a creature's orientation will be modified during a conversation so that it is facing a speaker. Passing false in the *bActive* parameter will turn off this behavior for the creature *oCreature*.

```
void StoreCameraFacing()
```

This stores the current camera facing in memory. It can be restored using RestoreCameraFacing.

Reaction

These routines are used to measure and manage how an individual creature feels about another creature. See also the Factions section.

```
int GetIsEnemy(
          object oTarget,
          object oSource = OBJECT_SELF )
```

This will return true only if the object *oSource* considers *oTarget* an enemy, based solely on faction membership and personal reputation.

```
int GetIsFriend(
          object oTarget,
          object oSource = OBJECT_SELF )
```

This will return true only if the object *oSource* considers *oTarget* a friend, based solely on faction membership and personal reputation.

```
int GetIsNeutral(
          object oTarget,
          object oSource = OBJECT_SELF )
```

This will return true only if the object *oSource* considers *oTarget* as neutral, based solely on faction membership and personal reputation.

```
int GetIsReactionTypeFriendly(
          object oTarget,
          object oSource = OBJECT_SELF )
```

This routine takes into account reputation and the area's PvP setting to determine if the object *oSource* has a friendly reaction toward *oTarget*. If both are PCs, this will also factor in the *oSource* object's like or dislike setting for *oTarget*. See GetIsFriend.

```
int GetIsReactionTypeNeutral(
          object oTarget,
          object oSource = OBJECT_SELF )
```

This routine takes into account reputation and the area's PvP setting to determine if the object *oSource* has a neutral reaction toward *oTarget*. If both are PCs, this will also factor in the *oSource* object's like or dislike setting for *oTarget*. See GetIsNeutral.

```
int GetIsReactionTypeHostile(
          object oTarget,
          object oSource = OBJECT_SELF )
```

This routine takes into account reputation and the area's PvP setting to determine if the object *oSource* has a hostile reaction toward *oTarget*. If both are PCs, this will also factor in the *oSource* object's like or dislike setting for *oTarget*. See GetIsEnemy.

```
void SetIsTemporaryEnemy(
          object oTarget,
          object oSource = OBJECT_SELF,
          int bDecays = FALSE,
          float fDurationInSeconds =
            180.0f )
```

This causes *oSource* to become an enemy of *oTarget* based on personal reputation. If *bDecays* is true, the enmity will decay over a time interval of *fDurationInSeconds*. The animosity will continue as long as the faction reputation plus the total personal reputation of *oSource* is at or below REPUTATION_TYPE_ENEMY.

```
void SetIsTemporaryFriend(
          object oTarget,
          object oSource = OBJECT_SELF,
          int bDecays = FALSE,
          float fDurationInSeconds =
            180.0f )
```

This causes *oSource* to become a friend of *oTarget* based on personal reputation. If *bDecays* is true, the friendship will decay over a time interval of *fDurationInSeconds*. The friendship will continue as long as the faction reputation plus the total personal reputation of *oSource* is at or above REPUTATION_TYPE_FRIEND.

```
void SetIsTemporaryNeutral(
          object oTarget,
          object oSource = OBJECT_SELF,
          int bDecays = FALSE,
          float fDurationInSeconds =
            180.0f )
```

This causes *oSource* to become neutral toward *oTarget* based on personal reputation. If *bDecays* is true, the neutrality will decay over the interval *fDurationInSeconds* seconds. The neutrality will continue as long as the faction reputation plus the total personal reputation of *oSource* is above REPUTATION_TYPE_ENEMY and below REPUTATION_TYPE_FRIEND.

```
void SetPCDislike(
          object oPlayer,
          object oTarget )
```

This causes the player *oPlayer* and object *oTarget* to dislike each other.

```
void SetPCLike(
          object oPlayer,
          object oTarget )
```

This causes the player *oPlayer* and object *oTarget* to like each other.

```
void SurrenderToEnemies()
```

When this is run by an NPC, this will cause all NPCs within a 10 metre radius to cease their activities. Any enemies of the NPC within this radius will assume a neutral attitude toward the NPC.

Saving Throws

```
int GetFortitudeSavingThrow(
          object oTarget )
```

If *oTarget* is a creature, door or placeable, this routine will return the object's base fortitude saving throw. A zero will be returned for an invalid object. See FortitudeSave and SetFortitudeSavingThrow.

```
int GetReflexAdjustedDamage(
          int nDamage,
          object oTarget,
          int nDC,
          int nSaveType =
            SAVING_THROW_TYPE_NONE,
          object oSaveVersus = OBJECT_SELF )
```

This routine is used within scripts that apply damage from spell to adjust the total damage by reflex saving throws and evasion. The spell target *oTarget* makes a saving throw check of type *nSaveType* at difficulty class *nDC* against the object *nSaveType* to reduce the base damage *nDamage*. The saving throw type is a SAVING_THROW_TYPE_... constant, which includes various spell descriptors such as acid, death, fear, law, negative energy and sonic.

Example script: 'x0_s3_shurik'.

```
int GetReflexSavingThrow(
          object oTarget )
```

If *oTarget* is a creature, door or placeable, this routine will return the object's base reflex saving throw. A zero will be returned for an invalid object. See ReflexSave and SetReflexSavingThrow.

```
int GetWillSavingThrow(
          object oTarget )
```

If *oTarget* is a creature, door or placeable, this routine will return the object's base willpower saving throw. A zero will be returned for an invalid object. See WillSave and SetWillSavingThrow.

Checks

The following calls are used to determine whether the creature *oCreature* succeeds at a saving throw check against the source object *oSaveVersus* with the *nDC* difficulty class against an effect caused by *oSaveVersus*. The type *nSaveType* can be set to a SAVING_THROW_... global constant for saves against specific spell descriptors.

The returned result is a SAVING_THROW_CHECK_... constant. The value indicates whether the save was successful, a failure, or if the target was immune to the save type. See the notes when using these calls in an Area of Effect object script.

```
int FortitudeSave(
          object oCreature,
          int nDC,
          int nSaveType =
            SAVING_THROW_TYPE_NONE,
          object oSaveVersus = OBJECT_SELF )
```

This is used for a Fortitude saving throw check. See the section introduction, along with GetFortitudeSavingThrow and SetFortitudeSavingThrow.

```
int ReflexSave(
          object oCreature,
          int nDC,
          int nSaveType =
            SAVING_THROW_TYPE_NONE,
          object oSaveVersus = OBJECT_SELF )
```

This is used for a Reflex saving throw check. See the section introduction, along with GetReflexSavingThrow and SetReflexSavingThrow.

```
int WillSave(
          object oCreature,
          int nDC,
          int nSaveType =
            SAVING_THROW_TYPE_NONE,
          object oSaveVersus = OBJECT_SELF )
```

This is used for a Willpower saving throw check. See the section introduction, along with GetWillSavingThrow and SetWillSavingThrow.

Sense

See also the descriptions for GetMatchedSubstring and GetMatchedSubstringCount.

```
int GetIsListening( object oObject )
```

This will return true if the object *oObject* is listening. See SetIsListening.

```
int GetListenPatternNumber()
```

If an 'On Conversation' script is triggered by a listen pattern match (as established by the SetListenPattern call), this will return the index of the pattern heard. If none of the patterns matched, this will return -1. This is used, for example, in the nw_c2_default4 'On Conversation' script to check if the creature heard any of the predefined patterns.

```
int GetObjectHeard(
          object oTarget,
          object oSource = OBJECT_SELF )
```

This returns true only if creature *oSource* has heard the creature *oTarget*.

```
int GetObjectSeen(
          object oTarget,
          object oSource = OBJECT_SELF )
```

This returns true only if creature *oSource* has viewed the creature *oTarget*.

```
int LineOfSightObject(
          object oSource,
          object oTarget )
```

This call returns true only if there is a direct line of sight from *oSource* to *oTarget*. This can be blocked by terrain or placeables. Note that the call notes warn against frequent use of this function because it is computationally expensive.

```
int LineOfSightObject(
          vector vSource,
          vector vTarget )
```

This call returns true only if there is a direct line of sight from the vector v*Source* to the vector v*Target*. This can be blocked by terrain or placeables. Note that the call notes warn against frequent use of this function because it is computationally expensive.

```
void SetAssociateListenPatterns(
          object oTarget = OBJECT_SELF )
```

This will clear the listening patterns for the target *oTarget*, then set the listening patterns to the strings in the include file gb_setassociatelistenpatterns. These are used for henchmen and other associates.

```
void SetListening(
          object oObject,
          int bValue )
```

This call will establish whether or not the object *oObject* is listening, based on the boolean *bValue*. See GetIsListening.

```
void SetListenPattern(
          object oObject,
          string sPattern,
          int nNumber = 0 )
```

When the object *oObject* is listening, this will cause it to listen for the pattern *sPattern*, as spoken in the game via the SpeakString call. Objects can listen to multiple patterns, in which case *nNumber* is the index number of the pattern being set.

Store

See GetInfiniteFlag and SetInfiniteFlag.

```
int GetStoreGold( object oStore )
```

If *oStore* is a valid store that is using gold, this will return the amount of gold currently possessed by that store. If the store is not using gold, this returns -1. If *oStore* is not a valid store, this will return -2.

```
int GetStoreIdentifyCost( object oStore )
```

If the object *oStore* is a Store, this routine will return the cost in gold pieces that the store will charge for identifying a magic item. A result of -1 means the store does not identify magic items. If *oStore* is not a valid store, this will return -2.

```
int GetStoreMaxBuyPrice( object oStore )
```

This routine returns the maximum price in gold pieces that the store *oStore* will spend for an item. If -1 is returned, then there is no limit on the amount the store will spend (other than the total gold available). If *oStore* is not a valid store, this will return -2.

```
void OpenStore(
          object oStore,
          object oPC,
          int nBonusMarkUp = 0,
          int nBonusMarkDown = 0 )
```

This routine will open the interface of the store *oStore* for the PC *oPC*. The *nBonusMarkUp* is the percentage (from -100 to 100) the store will add to the base cost of an item for sale. The *nBonusMarkDown* is the percentage (from -100 to 100) that the store will deduct from the base cost of an item it will purchase.

```
void SetStoreGold(
          object oStore,
          int nGold )
```

This will set the amount of gold *nGold* that the store *oStore* has on hand for making purchases. If *nGold* is -1, then the store is set to not use gold and so will not purchase objects.

```
void SetStoreIdentifyCost(
          object oStore,
          int nCost )
```

This routine will change the price *nCost* that the store *oStore* will charge for identifying magic items. By default this amount is 100 gold pieces. A price of -1 will prevent the store from identifying items.

```
void SetStoreMaxBuyPrice(
          object oStore,
          int nMaxBuy )
```

This call will set the maximum amount *nMaxBuy* that the store *oStore* will spell to purchase an item that it is allowed to buy. The store must still have enough gold on hand to make the purchase. An *nMaxBuy* setting of -1 means the store can spend up the maximum gold it has available.

Interface

In the following calls, the colors must be in hexadecimal code, rather than an index into the nwn_colors.2da file.

```
void BlackScreen(
          object oCreature,
          int nColor = 0 )
```

This routine will cause the screen to instantly change to the color *nColor* for the player that controls *oCreature*. By default the color is black. This can be used prior to a FadeFromBlack call, and it is reverted by a StopFade call.

```
void FadeFromBlack(
          object oCreature,
          float fSpeed = FADE_SPEED_MEDIUM )
```

This causes the screen for the player running the creature *oCreature* to fade from black. The *fSpeed* determines the time period required for the fade to complete. There are several global constants FADE_SPEED_... that can be passed in this field.

```
void FadeToBlack(
          object oCreature,
          float fSpeed = FADE_SPEED_MEDIUM,
          float fFailsafe = 5.0,
          int nColor = 0 )
```

The screen for the player running the creature *oCreature* will fade to the color *nColor*, which defaults to black. The *fSpeed* determines the time period required for the fade to complete. If *fFailsafe* is not zero, then after *fFailsafe* seconds the fade will be removed even if a FadeFromBlack call is not executed.

```
void SetPanelButtonFlash(
          object oPlayer,
          int nButton,
          int bEnableFlash )
```

When *bEnableFlash* is set to true, this call will make the button *nButton* flash on the interface of player *oPlayer*. Valid button values are PANEL_BUTTON_... constants. Passing false in *bEnableFlash* will turn off the flashing.

```
void StopFade(
          object oCreature )
```

This call will remove any fading for the player controlling the creature *oCreature*.

Panels

See also OpenInventory and OpenStore.

```
void CloseGUIScreen(
          object oPlayer,
          string sScreenName )
```

If the graphical panel *sScreenName* is open on the client display of the player *oPlayer*, and the panel exists in the [ScriptGUI] section of the ingamegui.ini file, then this routine will close the panel.

```
void DisplayGuiScreen(
          object oPlayer,
          screen sScreenName,
          int bModal,
          string sFileName = “”,
          int bOverrideOptions = FALSE )
```

This routine runs the graphical user interface screen *sScreenName* on the client system of the player *oPlayer*. The valid screen names are listed in the [GuiScreen] section of the file *ingamegui.ini*, if it exists. Otherwise, a valid screen resource file can be passed in the *sFileName* field (as long as that is not already in use). This file will typically be of the form *myguiname*.xml. If *bModal* is true then the interface is modal and can not be bypassed. The *bOverrideOptions* field is not documented.

For details on how to communicate with this GUI screen, see the Components section below.

```
void DisplayInputBox(
          object oPC,
          int nMessageStrRef,
          string sMessage,
          sOkCB = “”,
          sCancelCB = “”,
          bShowCancel = TRUE,
          sScreenName =
            SCREEN_STRINGINPUT_MESSAGEBOX,
          nOkayStrRef = 66,
          sOkayString = “”,
          nCancelStrRef = 67,
          sCancelString = “”,
          sDefaultString = “” )
```

The notes for this command were mangled in the toolkit, so the parameter types above are best guesses. This runs a text input interface on the client system of the player running the character *oPC*. The dialogue includes a message, an input field and an okay and cancel buttons. The field names of the interface are set by the references in the '.tlk' file, unless they are overridden by passing a string as an argument:

Interface Element	String Reference	Override Field
Okay Button	nOkayStrRef	sOkayString
Cancel Button	nCancelStrRef	nCancelString
Message	nMessageStrRef	sMessage
Default input	—	sDefaultString

If a script is passed in the *sOkCB* field, and the name begins with 'gui', then that script will be run when the player clicks the okay button. Likewise, a *sCancelCB* script name string beginning with 'gui' be run when the player clicks cancel.

```
void DisplayMessageBox(
          object oPC,
          int nMessageStrRef,
          string sMessage,
          sOkCB = “”,
          sCancelCB = “”,
          bShowCancel = TRUE,
          sScreenName =
            SCREEN_STRINGINPUT_MESSAGEBOX,
          nOkayStrRef = 66,
          sOkayString = “”,
          nCancelStrRef = 67,
          sCancelString = “” )
```

The notes for this command were mangled in the toolkit, so the parameter types above are best guesses. This runs a text message interface on the client system of the player running the character *oPC*. The dialogue includes a message plus okay and cancel buttons. The field names of the interface are set by the references in the '.tlk' file, unless they are overridden by passing a string as an argument:

Interface Element	String Reference	Override Field
Okay Button	nOkayStrRef	sOkayString
Cancel Button	nCancelStrRef	nCancelString
Message	nMessageStrRef	sMessage

If a script is passed in the *sOkCB* field, and the name begins with 'gui', then that script will be run when the player clicks the okay button. Likewise, a *sCancelCB* script name string beginning with 'gui' be run when the player clicks cancel.

For a larger message box with a scrolling window, you can pass "SCREEN_MESSAGEBOX_REPORT" for the "sScreenName" value. This is the same dialog box that is used for the tutorial interface in the original campaign.

```
object GetPlayerCreatureExamineTarget(
          object oCreature )
```

If *oCreature* is a valid player-controlled creature, this will return the creature for which the player has opened the creature examine panel.

```
void PopUpDeathGUIPanel(
          object oPC,
          int bRespawnButtonEnabled = TRUE,
          int bWaitForHelpButtonEnabled =
            TRUE,
          int nHelpStringReference = 0,
          string sHelpString = "" )
```

This routine displays the user interface panel that is displayed when a player character *oPC* has died. If *bRespawnButtonEnabled* is true, the panel will display the Respawn button, allowing the character to be spawned back into the game. The *bWaitForHelpButtonEnabled* is true, the "Wait for Help" button will appear. *The remaining parameters are undocumented.*

```
void PopUpGUIPanel(
          object oPC,
          int nGUIPanel )
```

This call will cause the graphical user interface panel *nGUIPanel* to appear on the client of the player that controls the player character *oPC*. The *nGUIPanel* is a global constant of the form GUI_PANEL_.... Currently there is only one GUI panel: GUI_PANEL_PLAYER_DEATH.

```
void ShowWorldMap(
          string sWorldMap,
          object oPlayer,
          string sTag )
```

This displays the world map *sWorldMap* to the player *oPlayer*. The current location is identified by the string *sTag*.

Components

These calls are used for manipulating components in user interface (UI) panels. See the UI/default directory under the game install folder for various XML-based user interface definitions.

```
void SetGUIObjectText(
          object oPlayer,
          string sScreenName,
          string sUIObjectName,
          int nStrRef,
          string sText )
```

For the GUI named *sScreenName* opened for player character *oPlayer*, this call will send the string referenced by *nStrRef* to the UIText object named *sUIObjectName*. If *nStrRef* is set to –1, this call will send the text string *sText* instead.

- SetGUIObjectDisabled
- SetGUIObjectHidden
- SetGUIProgressBarPosition
- SetGUITexture
- AddListBoxRow
- ClearListBox
- ModifyListBoxRow
- RemoveListBoxRow
- SendNoticeText
- SetListBoxRowSelected
- SetLocalGUIVariable
- SetPlayerGUIHidden
- SetScrollBarRanges
- SetScrollbarValue

Journal

See the 'ginc_journal' include file for other scripted journal routines.

```
void AddJournalQuestEntry(
          string szPlotID,
          int nState,
          object oCreature,
          int bAllPartyMembers = TRUE,
          int bAllPlayers = FALSE,
          int bAllowOverrideHigher = FALSE )
```

This will add a journal entry from the journal editor to the owning player's journal of creature *oCreature*. The *szPlotID* is the journal category tag, while *nState* is the journal entry ID. If *bAllPartyMembers* is true, add the entry to all journals of players in the party. If *bAllPlayers* is true, the entry is added to every journal. Unless *bAllowOverrideHigher* the entry must be se to a *nState* that is higher than the current ID.

```
int GetJournalEntry(
          string szPlotID,
          object oCreature )
```

This returns the current journal entry identifier from the *szPlotID* journal category tag of the creature *oCreature*.

```
int GetJournalQuestExperience(
          string szPlotID )
```

This retrieves the experience points that are granted to the player for the completion of the quest with the journal category tag *szPlotID*.

```
void RemoveJournalQuestEntry(
          string szPlotID,
          object oCreature,
          int bAllPartyMembers = TRUE,
          int bAllPlayers = FALSE )
```

This routine will remove the a journal entry with the tag *szPlotID* from the journal of the creature *oCreature*. If *bAllPartyMembers* is true, remove the entry from the journals of all players in the party. If *bAllPlayers* is true, the entry is removed from every journal.

Messages

```
void DebugPostString(
          object oTarget,
          string sMessage,
          int nX,
          int nY,
          float fDuration,
          int nColor = 4294901760 )
```

This posts a debug message *sMessage* on the screen of the object *oTarget*. The *nX* and *nY* specify the screen coordinates, *fDuration* is the length of time the message will remain and *nColor* is the color used.

```
void FloatingTextStringOnCreature(
          string sStringToDisplay,
          object oCreatureToFloatAbove,
          int bBroadcastToFaction = TRUE,
          float fDuration = 5.0f )
```

This call will cause the string *sStringToDisplay* to float above the creature *oCreatureToFloatAbove*. If the boolean *bBroadcastToFaction* is true, then only players controlling creatures belonging to the same faction as the creature to float a message above will see the message, and then only if they are within 30 meters. The duration that the message will appear is set by *fDuration* in seconds.

```
void FloatingTextStrRefOnCreature(
          int nStrRefToDisplay,
          object oCreatureToFloatAbove,
          int bBroadcastToFaction = TRUE,
          float fDuration = 5.0 )
```

This call will cause a message to float above the creature *oCreatureToFloatAbove*. The *nStrRefToDisplay* integer is an identifier for a string in the Dialog.tlk file. If the boolean *bBroadcastToFaction* is true, then only players controlling creatures belonging to the same faction as the creature to float a message above will see the message, and then only if they are within 30 meters. The duration that the message will appear is set by *fDuration* in seconds.

```
string GetStringByStrRef(
          int nStrRef,
          int nGender = GENDER_MALE )
```

This call retrieves a string from a special file called a talk table. This consists of a set of language-translated strings that can be looked up via a reference ID. (This is the dialog.TLK file in the game folder.) This routine will return the string associated with reference *nStrRef*, with a gender determined by *nGender*.

As an example, the racialsubtypes.2da file lists the Tiefling on row 14. The Description column of this row has string reference 112092. Calling GetStringByStrRef with *nStrRef* set to this value returns the description that is presented during the character generation process for the Tiefling.

```
void SendMessageToAllDMs( string szMessage )
```

This sends the message *szMessage* to the DMs on the server.

```
void SendMessageToPC(
          object oPlayer,
          string sMessage )
```

This will cause the message *sMessage* to appear in the chat window of player controlling the PC *oPlayer*.

```
void SendMessageToPCByStrRef(
          object oPlayer,
          int nStrRef )
```

This is similar to SendMessageToPC, except the message to the player *oPlayer* is obtained from the string reference *nStrRef*.

Message Colors

The following color hex codes can be used to simulate the colors that the game engine uses in the chat window. (For more information, see the Font Format section in volume I.)

Color	Hex Code	Typical Uses
Red	`#eb0000`	Fire damage; pause
Dk. Orange	`#f16000`	Physical damage; combat msg.
Lt. Orange	`#ff9900`	Sonic damage
Yellow	`#ffff00`	Divine damage; informational
Green	`#00ff00`	Acid damage
Lt. Blue	`#99ffff`	Cold damage; PC name
Blue	`#60c1f1`	Opponent save
Dk. Blue	`#005eec`	Electric damage
Lt. Violet	`#cc99cc`	Opponent name
Violet	`#cc77ff`	Magical damage; casting spell
White	`#ffffff`	Positive energy damage
Gray	`#8e8e8e`	Negative energy damage

Inventory

Items are objects that can be placed in the inventory of a creature or container. The following routines are used for item management and the manipulation of an inventory.

Management

```
void ActionEquipItem(
          object oItem,
          int nInventorySlot )
```

Place the item *oItem* into the equipment slot *nInventorySlot*, which is specified by one of the INVENTORY_SLOT_... constants. If an error occurs, this will log (but not print) a message.

```
void ActionEquipMostDamagingMelee(
          object oTarget = OBJECT_INVALID,
          int bOffhand = FALSE )
```

The creature will equip itself with the melee weapon in its inventory that inflicts the highest damage. If none is available, the highest damage ranged weapon will be equipped instead. If *oTarget* is a creature, the most effective weapon against that creature will be selected (thus presumably taking advantage of 'Attack Bonus vs. ...' item properties). If *bOffhand* is true, put the weapon in the off hand. (The last didn't seem to work when I tested it.) The notes for this command warn that it should only be run in the 'End of Combat Round' scripts.

```
void ActionEquipMostDamagingRanged(
          object oTarget = OBJECT_INVALID )
```

The creature will equip itself with the ranged weapon in its inventory that inflicts the highest damage. If *oTarget* is a creature, the most effective weapon against that creature will be selected (thus presumably taking advantage of 'Attack Bonus vs. ...' item properties).

```
void ActionEquipMostEffectiveArmor()
```

The creature will equip the armor with the highest Armor Class in its inventory. If there are two pieces of armor with the same Armor Class, this routine appears to choose the one with the highest allowed dexterity bonus.

```
void ActionGiveItem(
          object oItem,
          object oGiveItemTarget,
          int bDisplayFeedback = TRUE )
```

The subject of this command will give the item *oItem* to the creature *oGiveItemTarget*. Nothing will occur if either *oItem* is not a valid item or *oGiveItemTarget* is not an existing object. If *bDisplayFeedback* is true then a message will be displayed in the chat window.

```
void ActionPickUpItem( object oItem )
```

The subject picks up the item *oItem* from the ground. If an error occurs, it will be logged to the log file.

```
void ActionPutDownItem( object oItem )
```

The subject puts item *oItem* on the ground. If an error occurs, it will be logged to the log file.

```
void ActionTakeItem(
          object oItem,
          object oTakeFrom,
          int bDisplayFeedback = TRUE )
```

This action will cause the subject to take the object *oItem* from the target *oTakeFrom*. If *bDisplayFeedback* is true, and the exchange involves a PC, a message will be printed to the chat window. If either *oItem* or *oTakeFrom* is invalid, nothing happens.

```
void ActionUnequipItem(
          object oItem )
```

If *oItem* is valid, it will be unequipped from its current slot and placed in the subject's general inventory.

```
void OpenInventory(
          object oCreature,
          object oPlayer )
```

This will open the inventory panel of the creature *oCreature* for the player *oPlayer*. For players this will only work for their owned player or the controlled creatures. If the *oPlayer* is the DM, this can be used to view the inventory of any creature.

```
void SetDroppableFlag(
          object oItem,
          int bDroppable )
```

This sets the Droppable property of an item *oItem* to the boolean state *bDroppable*.

```
void SetIdentified(
         object oItem,
         int bIdentified )
```

This will set the Identified property of the item *oItem* to *bIdentified*.

```
void SetInfiniteFlag(
         object oItem,
         int bInfinite = TRUE )
```

If *bInfinite* is true, this will set the Infinite property on an item *oItem* in a store inventory. This will cause the store not to run out of the item.

```
void SetItemCharges(
         object oItem,
         int nCharges )
```

The charges on an item are used to power it's charged-based special properties. This call sets the number of charges on the item *oItem* to *nCharges*. Setting the number of charges to zero will result in the item being destroyed.

```
void SetItemCursedFlag(
         object oItem,
         int bCursed )
```

This call sets the Cursed property on the item *oItem* to the boolean *bCursed*. When the Cursed property is true, an item can not be dropped.

```
void SetItemPropActivated(
         object oItem,
         int nPref )
```

The activation preference property determines how the special properties of item *oItem* are activated. A value for *nPref* of zero requires the item to be equipped; a value of 1 means the item will only be active when it is in the creature's repository (and not equipped), while a value of 3 will be active whether it is equipped or not.

```
void SetItemStackSize(
         object oItem,
         int nSize,
         int bDisplayFeedback = TRUE )
```

The size of the stack for item *oItem* is changed to *nSize*. If *nSize* is greater than the maximum stack size for the item, then the stack size will be set to the maximum. Values less than 1 will be set to 1. Setting *bDisplayFeedback* prevents feedback from being sent to a player.

```
void SetPickpocketableFlag(
         object oItem,
         int bPickpocketable )
```

If *oItem* is a valid item, this will set the state of the item's 'Pickpocketable?' flag to *bPickpocketable*.

```
void SetStolenFlag(
         object oStolen,
         int bStolenFlag )
```

This will set the stolen flag of the item *oStolen* to the boolean *bStolenFlag*. Stolen items can not be sold at stores that has the 'Black Market?' property set to false.

```
void SetWeaponVisibility(
         object oObject,
         int bVisible,
         int nType = 0 )
```

This sets the visibility on a weapon, helm or both to *bVisible* for object *oObject*. A *nType* of 0 sets the weapon visibility; a value of 1 sets the helm visibility, and 2 sets the visibility on both.

Creation

```
object CopyItem(
         object oItem,
         object oTarget = OBJECT_INVALID,
         int bCopyVars = FALSE )
```

If *oItem* is a valid item, this call will make a copy of the object and place it in the inventory of the object *oTarget*, and return the object copy. If *oItem* is invalid or a non-empty container, this will return OBJECT_INVALID. If *oTarget* is invalid, the copy will be placed at the same location as *oItem*. If *bCopyVars* is true, any local variables set on item *oItem* will be reproduced on the copy.

If there are stackable objects at the item's target location, then the copy will be merged with the existing copies and the combined stack will be returned by this call.

```
object CopyItemAndModify(
         object oItem,
         int nType,
         int nIndex,
         int nNewValue,
         int nCopyVars = FALSE )
```

This call makes copy of the object *oItem* while making a single change to the item's appearance. The modification is determined by the *nType*, *nIndex* and *nNewValue*, where *nType* and *nIndex* are ITEM_APPR_... constants as follows:

nType ITEM_APPR_TYPE_...	**nIndex** ITEM_APPR_...	**iNewValue**
SIMPLE_MODEL	N/A	Model #
WEAPON_COLOR	WEAPON_COLOR_...	1-4
WEAPON_MODEL	WEAPON_MODEL_...	Model #
ARMOR_MODEL	ARMOR_MODEL_...	Model #
ARMOR_COLOR	ARMOR_COLOR_...	0-63

Thus the following call:

```
object myCustomItem = CopyItemAndModify(
  oExistingItem,
  ITEM_APPR_TYPE_WEAPON_MODEL,
  ITEM_APPR_WEAPON_MODEL_TOP, 2 );
```

will change the color of the top part of a weapon to model number 2. The *iIndex* value is ignored for simple items and helmets. See GetItemAppearance.

```
object CreateItemOnObject(
          string sItemTemplate,
          object oTarget = TARGET_SELF,
          int nStackSize = 1,
          string sNewTag = "",
          int bDisplayFeedback = TRUE )
```

The item with the template *sItemTemplate* is created in the inventory of the object *oTarget*. It will set the stack size of the created object to the lower of *nStackSize* and the item's maximum stack size. If *nNewTag* is not an empty tag, the tag of the item will be set to this value. If *bDisplayFeedback* is false, no feedback will be printed in the chat window.

Query

See also GetHasInventory, GetInventoryDisturbItem, GetLastDisturbed and the Item properties section.

```
int GetArmorRank( object oItem )
```

If the item *oItem* is of base item type Armor, this will return the category as an ARMOR_RANK_... constant.

```
int GetBaseItemType( object oItem )
```

This call returns a constant BASE_ITEM_... that gives the Base Item property of the item *oItem*.

```
int GetDroppableFlag(
          object oItem )
```

This routine will return true only if the item *oItem* can be dropped. That is, the 'Droppable?' parameter is true.

```
object GetFirstItemInInventory(
          object oTarget = OBJECT_SELF )
```

If *oTarget* is a creature, item, placeable or store, this call will return the first item in the object's inventory. Subsequent calls to GetNextItemInInventory will return the remaining items.

```
int GetGoldPieceValue( object oItem )
```

For a valid item *oItem*, this will return the base gold piece value. If *oItem* is not an item, this will return 0.

```
int GetIdentified( object oItem )
```

This will return true only if *oItem* is a valid item and it has been identified. See SetIdentified.

```
int GetInfiniteFlag(
          object oItem )
```

If *oItem* is a valid store item and it has the infinite flag set to true, it will not run out when purchased. See SetInfiniteFlag.

```
int GetItemACValue(
          object oItem )
```

If the item *oItem* provides an armor class benefit, this routine will return the armor value.

```
int GetItemAppearance(
          object oItem,
          int nType,
          int nIndex )
```

This routine returns the appearance values for the type *nType* and index *nIndex* of item *oItem*. Valid values for the *nType* and *nIndex* parameters match the constants used in the CopyItemAndModify routine.

```
int GetItemCharges(
          object oItem )
```

This call will query the item *oItem* and return the number of charges remaining.

```
int GetItemCursedFlag(
          object oItem )
```

This will return true if the item *oItem* has the cursed property set. A cursed item can not be dropped by its owner.

```
int GetItemIcon( object oTarget )
```

This will return the icon number of the item *oTarget*. This is a row number in the 'nwn_icons.2da' file.

```
object GetItemInSlot(
          int nInventorySlot,
          object oCreature = OBJECT_SLOT )
```

This routine will return the item in the inventory slot *nInventorySlot* of the creature *oCreature*, if any. The slot type is a constant INVENTORY_SLOT_....

```
object GetItemPossessedBy(
          object oCreature,
          string sItemTag )
```

This returns the object instance of an item with the tag *sItemTag* that is owned by the creature *oCreature*, or OBJECT_INVALID if the creature lacks the item.

```
object GetItemPossessor( object oItem )
```

This will return the object that currently possesses the item *oItem*, or OBJECT_INVALID if none.

```
int GetItemPropActivation(
          object oItem )
```

This call returns a constant that gives the 'Item Property Activation Preference' setting for the item *oItem*. A return value of 0 indicates an item that is active only when equipped. If the result is 1, the item is active only when not equipped. The value of 2 indicates an item that is active regardless of whether it is equipped or not. See SetItemPropActivation.

```
int GetItemStackSize( object oItem )
```

This returns the maximum stack size of the object *oItem*.

```
object GetNextItemInInventory(
          object oTarget = OBJECT_SELF )
```

This call returns the next item in the inventory of object *oTarget*. This list can be re-initialized by a call to the GetFirstItemInInventory routine.

```
object GetNumStackedItems( object oItem )
```

This returns the number of items in the *oItem* stack. This is a value between 1 and the maximum stack size for the item.

```
int GetPickpocketable( object oItem )
```

If *oItem* is a valid item, this will return true if the item's 'Pickpocketable?' flag is set to true.

```
object GetSpellCastItem()
```

Some spells are configured to run a script when they are cast. When this call is run from a spell script, it will return the item that was used to activate the spell.

```
int GetStolenFlag( object oStolen )
```

If *oStolen* is an item, this call will return true if the object is flagged as stolen (such as via a successful pick pocket).

```
object GetWeaponRanged( object oItem )
```

This call returns true if the item *oItem* is a ranged weapon.

```
int GetWeaponType( object oItem )
```

If item *oItem* is a weapon, this routine will return a value WEAPON_TYPE_... that indicates the weapon type.

```
int GetWeight(
          object oTarget = OBJECT_SELF )
```

If *oTarget* is an item, this will return the item's weight in tenths of English pounds (1 pound = 0.45 kg). If *oTarget* is a creature, this will return the total weight being carried in tenths of pounds.

Module Item Scripts

These calls are useful in specific module scripts, which are set in the Scripts block of the Module properties.

```
object GetItemActivated()
```

This routine can be used in a module's 'On Activate Item' script to return the item activated.

```
object GetItemActivatedTarget()
```

This routine can be used in a module's 'On Activate Item' script to return the target of the item.

```
object GetItemActivatedTargetLocation()
```

This routine can be used in a module's 'On Activate Item' script to return the location of the item's target.

```
object GetItemActivator()
```

This routine can be used in a module's 'On Activate Item' script to return the creature that activated the item.

```
object GetModuleItemAcquired()
```

When called from a module's 'On Acquire Item' script, this routine will return the item acquired.

```
object GetModuleItemAcquiredBy()
```

When called from a module's 'On Acquire Item' script, this routine will return the object that acquired the item.

```
object GetModuleItemAcquiredFrom()
```

When called from a module's 'On Acquire Item' script, this routine will return the object that previously possessed the item.

```
object GetModuleItemAcquiredStackSize()
```

When called from a module's 'On Acquire Item' script, this routine will return the stack size of the item acquired.

```
object GetModuleItemLost()
```

When called from a module's 'On Unacquire Item' script, this routine will return the item lost.

```
object GetModuleItemLostBy()
```

When called from a module's 'On Unacquire Item' script, this routine will return the creature that lost the item.

```
object GetPCItemLastEquipped()
```

When called from a module's 'On Player Equip Item' script, this routine will return the item that was equipped.

```
object GetPCItemLastEquippedBy()
```

When called from a module's 'On Player Equip Item' script, this routine will return the PC that equipped the item returned by GetPCItemLastEquipped.

```
object GetPCItemLastUnequipped()
```

When called from a module's 'On Player Equip Item' script, this routine will return the item that was unequipped.

```
object GetPCItemLastUnequippedBy()
```

When called from a module's 'On Player Equip Item' script, this routine will return the PC that unequipped the item returned by GetPCItemLastUnequipped.

Item Properties

These routines can be used to modify properties on an item from a script, rather than configuring the item properties field. This is useful for transitory properties, such as those provided by a spell, or in-game enhancements using the crafting skills.

See the 'x2_inc_itemprop' file for a list of useful functions related to item properties. In particular, the function IPSafeAddItemProperty is useful when you want to prevent stacking of properties.

Management

```
void AddItemProperty(
          int nDurationType,
          itemproperty ipProperty,
          object oItem,
          float fDuration = 0.0f )
```

This will add a property to the item *oItem*. The *nDurationType* must be one of:

- DURATION_TYPE_PERMANENT
- DURATION_TYPE_TEMPORARY

The *ipProperty* can be created by a ItemProperty... routine. If temporary, the duration is specified by the *fDuration* value in seconds.

After using this call on an item in a character's inventory, the property may not become active until it is removed from the inventory. To activate right away, I copy the item using CopyItem, then destroy the original using DestroyObject.

```
void RemoveItemProperty(
          object oItem,
          itemproperty ipProperty )
```

This removes the item property *ipProperty* from the item *oItem*.

```
void SetItemPropActivation(
          object oItem,
          int nPref )
```

This sets the 'Item Property Activation Preference' property for an item *oItem*. Pasing a value of zero for *nPref* indicates activation when equipped, a one causes activation when not equipped, and a two results in activation whether the item is equipped or not.

Query

```
itemproperty GetFirstItemProperty(
          object oItem )
```

This will return an itemproperty result that is the first item property on a valid item *oItem*. Use GetNextItemProperty to view additional properties.

```
int GetIsItemPropertyValid(
          itemproperty nItemProp )
```

If the item property *nItemProp* is valid then this routine will return true.

```
int GetItemHasItemProperty(
          object oItem,
          int nProperty )
```

This routine returns true if *oItem* is a valid item and it has the property *nProperty*, which is a ITEM_PROPERTY_... constant.

```
int GetItemPropActivation(
          object oItem )
```

This returns the 'Item Property Activation Preference' setting for an item *oItem*. A zero indicates activation when equipped, a one is activated when not equipped, and a two is activated whether equipped or not.

```
int GetItemPropertyCostTable(
          itemproperty iProp )
```

Given an item property *iProp*, this routine will return a row in the 'iprp_costtable.2da' file. The name value on this row contains the name of a 2da file containing the item property cost information. GetItemPropertyCostTableValue is used to obtain the cost table value of the item property.

```
int GetItemPropertyCostTableValue(
          itemproperty iProp )
```

This returns an index in the item property cost table file, as determined by the GetItemPropertyCostTable call, for the item property *iProp*. Given the row of the cost table file, you can query the 2da file for the cost.

See the 'Two-Dimensional Arrays' section.

```
int GetItemPropertyDurationType(
          itemproperty iProp )
```

This call returns the duration type of the item property *iProp*. This is a DURATION_TYPE_... constant; either instant, permanent or temporary.

```
int GetItemPropertyParam1(
          itemproperty iProp )
```

The notes for this routine state that it returns the Param1 number of the item property *iProp* from the 2da file. It returns the row number in the iprp_paramtable.2da file. See GetItemPropertyParamValue1.

```
int GetItemPropertyParam1Value(
          itemproperty iProp )
```

This routine returns the Param1 value of the item property *iProp* from the 2da file. After finding the row of the iprp_paramtable.2da table using GetItemPropertyParam1, the entry in the TableResRef column gives the 2da file queried by this function. This call returns a row number in that 2da file.

For example, if row 4 of iprp_paramtable.2da is returned by GetItemPropertyParam1, the TableResRef column gives the file iprp_alignment.2da for a specific alignment. If this call returns row 5, then the resulting label is NE, or neutral evil.

```
int GetItemPropertySubType(
          itemproperty iProp )
```

This call fetches the subtype number of the item property from the iprop....2da files. For an item property type of damage vulnerability or damage immunity, it returns a constant of the form IP_CONST_DAMAGETYPE_*.

```
int GetItemPropertyType(
          itemproperty iProp )
```

This returns the type of the item property *iProp*. This is an ITEM_PROPERTY_... constant value.

```
itemproperty GetNextItemProperty(
          object oItem )
```

This routine will return the next item property in the list of properties for item *oItem*, or an invalid property when the end of the list is reached. Use the GetFirstItemProperty call to reset the list.

Properties

The following properties are equivalent to the properties that can be set in the 'Item Properties' field of an item blueprint.

```
itemproperty ItemPropertyAbilityBonus(
          int nAbility,
          int nBonus )
```

This item property provides a bonus *nBonus* to the ability *nAbility*, which is set to an IP_CONST_ABILITY_... global constant. The ability bonus is a positive integer between 1 and 12.

```
itemproperty ItemPropertyACBonus(
          int nBonus )
```

This item property gives a bonus *nBonus* to the owner's armor class. The bonus is a positive integer in the range from 1 to 20. The routine notes say that the type of bonus depends on the item, but not how.

```
itemproperty ItemPropertyACBonusVsDmgType(
          int nDamageType,
          int nACBonus )
```

This will add a property that gives an armor class bonus *nACBonus* against a type of damage *nDamageType*, which is a constant IP_CONST_DAMAGETYPE_.... The bonus is a positive integer between 1 and 20.

```
itemproperty ItemPropertyACBonusVsAlign(
          int nAlignGroup,
          int nACBonus )
```

This property provides an armor class bonus *nACBonus* against an alignment group *nAlignGroup*. The alignment is a constant IP_CONST_ALIGNMENTGROUP_.... The bonus is a positive integer from 1 and 20. The routine notes say that the type of bonus depends on the item, but not how.

```
itemproperty ItemPropertyACBonusVsRace(
          int nRace,
          int nACBonus )
```

This returns an item property that provides an armor class bonus *nACBonus* versus members of the racial group *nRace*, which is a IP_CONST_RACIALTYPE_... constant. (The values of these constants match those of the RACIAL_TYPE_... constants.) The bonus is a positive integer between 1 and 20.

Note that there is no IP_CONST_RACIAL_TYPE constant for incorporeal creatures, even though an item blueprint can be set with an AC bonus vs. incorporeal property.

```
itemproperty ItemPropertyACBonusVsSAlign(
          int nAlign,
          int nACBonus )
```

This property provides an armor class bonus *nACBonus* against an attacker with the alignment *nAlign*, which is a IP_CONST_ALIGNMENT_... constant. The bonus is a positive integer between 1 and 20.

```
itemproperty ItemPropertyArcaneSpellFailure(
          int nModLevel )
```

This property modifies the odds of an arcane spell failure, much like the penalty applied by armor and shields. The *nModLevel* parameter is a global constant of the form IP_CONST_ARCANE_SPELL_FAILURE_..., which gives penalties or bonuses in 5% increments between +50% and -50%.

```
itemproperty ItemPropertyAttackBonus(
          int nBonus )
```

This provides an attack bonus *nBonus*, which must be a positive integer between 1 and 20.

```
itemproperty ItemPropertyAttackBonusVsAlign(
          int nAlignGroup,
          int nBonus )
```

This item property provides the attack bonus *nBonus* when attacking a target belonging to the *nAlignGroup* alignment group, which is an IP_CONST_ALIGNMENTGROUP_... global constant. The bonus is a positive integer between 1 and 20.

```
itemproperty ItemPropertyAttackBonusVsRace(
          int nRace,
          int nBonus )
```

This will give an attack bonus *nBonus* when attacking a member of the racial group *nRace*, which is set by a global constant IP_CONST_RACIALGRUP_.... The bonus is a positive integer between 1 and 20.

```
itemproperty ItemPropertyAttackBonusVsSAlign(
          int nAlignment,
          int nBonus )
```

This is similar to ItemPropertyAttackBonusVsAlign, except the *nAlignment* is a IP_CONST_ALIGNMENT_... constant. It applies to an alignment rather than a group of alignments. The bonus is a positive integer from 1 to 20.

```
itemproperty ItemPropertyAttackPenalty(
          int nPenalty )
```

This causes the item to apply a penalty *nPenalty* to all attacks. The penalty is a positive integer between 1 and 5, which is then subtracted from the owner's attack bonus. There is no direct method to limit the penalty to specific races or alignments, although it could be mixed with ItemPropertyAttackBonus... properties.

```
itemproperty ItemPropertyBonusFeat(
          int nFeat )
```

The item owner gains the feat *nFeat*, which is a global constant IP_CONST_FEAT_.... corresponding to a row in 'iprp_feats.2da'. (Note that not all feats in 'feat.2da' are available as item property feats.) Unlike FeatAdd, this does not provide a means to check for prerequisites.

```
itemproperty ItemPropertyBonusHitpoints(
          int nBonusType )
```

This item property provides bonus hit points to the owner. The *nBonusType* is a row in the iprp_bonushp.2da file. A value of zero gives a random bonus, while from 1 to 20 it gives that many hit points. Thereafter it increases by +5 per row up to a maximum of +50 at row 26.

```
itemproperty ItemPropertyBonusSpellLevel(
          int nClass,
          int nSpellLevel )
```

This will add a bonus spell slot to the number of spells per day that can be prepared. The spell slot belongs to the class *nClass*, an IP_CONST_CLASS_... constant, and has spell level *nSpellLevel*.

```
itemproperty ItemPropertyBonusSavingThrow(
          int nBaseSaveType,
          int nBonus )
```

This adds a bonus *nBonus* to the base fortitude, reflex, or willpower saving throw *nBaseType*, which is a constant of the form IP_CONST_SAVEBASETYPE_.... A base type of IP_CONST_SAVEBASETYPE_ALL will apply the bonus to all saving throws. The bonus is a positive integer between 1 and 20.

```
itemproperty ItemPropertyBonusSavingThrowVsX(
          int nBonusType,
          int nBonus )
```

This provides a saving throw bonus *nBonus* against a particular effect or type of damage determined by *nBonusType*, which is a constant IP_CONST_SAVEVS_... that matches a row number in 'iprp_saveelement.2da'. The bonus is a positive integer between 1 and 20.

```
itemproperty ItemPropertyBonusSpellResistance(
          int nBonus )
```

This item property grants a spell resistance that is set by the IP_CONST_SPELLRESISTANCEBONUS_... constant that is passed as the *nBonus* parameter.

```
itemproperty ItemPropertyCastSpell(
          int nSpell,
          int nNumUses )
```

This allows an item to be used to cast a spell, identified by an IP_CONST_CASTSPELL_... constant for the *nSpell* parameter. The number of uses is set via *nNumUses*, which is a IP_CONST_CASTSPELL_NUMUSES_... constant that sets the uses per day or charges per use. Potions and wands can only be used to cast certain types of spells; these are documented in the command notes.

```
itemproperty ItemPropertyContainerReducedWeight(
          int nContainerType )
```

This property can be applied to a container item and it will reduce the weight of objects placed inside. The reduction is set by a IP_CONST_CONTAINERWEIGHTRED_... value for the *nContainerType* parameter.

```
itemproperty ItemPropertyDamageBonus(
          int nDamageType,
          int nDamage )
```

When applied to a weapon, this property will inflict increased damage *nDamage* of type *nDamageType*. The *nDamageType* value is a IP_CONST_DAMAGETYPE_... constant, but is limited to acid, bludgeon, cold, electrical, fire, pierce, slash or sonic damage types. The amount of damage *nDamage* is a IP_CONST_DAMAGEBONUS_... constant.

```
itemproperty ItemPropertyDamageBonusVsAlign(
          int nAlignGroup,
          int nDamageType,
          int nDamage )
```

This applies a damage bonus when attacking members of an alignment group. The alignment group *nAlignGroup* is a constant of type IP_CONST_ALIGNMENTGROUP_..., while *nDamageType* and *nDamage* are the same parameters used in ItemPropertyDamageBonus.

```
itemproperty ItemPropertyDamageBonusVsRace(
          int nRace,
          int nDamageType,
          int nDamage )
```

This applies a damage bonus when attacking members of a specific race. The race *nRace* is a constant of the form IP_CONST_RACIALTYPE_..., while the two parameters *nDamageType* and *nDamage* are the same as used in ItemPropertyDamageBonus.

```
itemproperty ItemPropertyDamageBonusVsSAlign(
          int nAlign,
          int nDamageType,
          int nDamage )
```

This applies a damage bonus when attacking members of a specific alignment. The alignment *nAlign* is a constant IP_CONST_ALIGNMENT_..., while *nDamageType* and *nDamage* are the same as the parameters used in ItemPropertyDamageBonus.

```
itemproperty ItemPropertyDamageImmunity(
          int nDamageType,
          int nImmuneBonus )
```

This property provides an *nImmuneBonus* percentage immunity to damage of the form *nDamageType*. The immunity is a IP_CONST_DAMAGEIMMUNITY_... constant that determines the percentage. The damage type is an IP_CONST_DAMAGETYPE_... constant, but only acid, bludgeon, cold, electric, fire, pierce, slash and sonic damage types are valid.

```
itemproperty ItemPropertyDamagePenalty(
          int nPenalty )
```

This inflicts a penalty to damage inflicted by the item owner. The penalty *nPenalty* is a positive integer between 1 and 5.

```
itemproperty ItemPropertyDamageReduction(
          int nAmount,
          int nDRSubType,
          int nLimit = 0,
          int nDRType = DR_TYPE_BONUS )
```

The item with this property will cause a reduction in the amount of damage *nAmount* suffered. It will reduce damage of the type *nDRType*, which is a DR_TYPE_... constant. Depending on the value of *nDRType*, the damage type can be further constrained using the *nDRSubType* field. (See the command notes for details) If *nLimit* is non-zero, this determines the maximum damage that can be absorbed before the property is eliminated.

```
itemproperty ItemPropertyDamageResistance(
          int nDamageType,
          int nHPResist )
```

The owner of this item becomes resistant to damage of the form *nDamageType*; a IP_CONST_DAMAGE_TYPE_... constant. A IP_CONST_DAMAGE_RESIST_... is passed as the *nHPResist* parameter to determine the hit points of damage resisted.

```
itemproperty ItemPropertyDamageVulnerability(
          int nDamageType,
          int nVulnerability )
```

The item causes the owner to suffer increased damage of the type *nDamageType*. The amount of additional damage is set by the IP_CONST_DAMAGEVULNERABILITY_... constant passed via the *nVulnerability* parameter.

```
itemproperty ItemPropertyDarkvision()
```

The owner gains the *darkvision* ability.

```
itemproperty ItemPropertyDecreaseAbility(
          int nAbility,
          int nModifier )
```

This causes a reduction *nModifier* in the ability score *nAbility*, which is a IP_CONST_ABILITY_... constant. The modifier *nModifier* is a positive integer between 1 and 10.

```
itemproperty ItemPropertyDecreaseAC(
          int nModifierType,
          int nPenalty )
```

This item property causes a decrease in armor class equal to *nPenalty*. The type of decrease *nModifierType* is a constant IP_CONST_ACMODIFIERTYPE_.... The penalty is a positive integer between 1 and 5.

```
itemproperty ItemPropertyDecreaseSkill(
          int nSkill,
          int nPenalty )
```

This property causes the item to decrease the skill *nSkill* of the owner by the amount *nPenalty*. The skill is a constant SKILL_..., while the penalty is a positive integer from 1 to 10.

```
itemproperty ItemPropertyEnhancementBonus(
          int nBonus )
```

This gives an enhancement bonus *nBonus* to a weapon, which adds the bonus to the base attack and to the damage inflicted. This bonus is a positive integer ranging from 1 to 20. As with other types of bonuses, multiple enhancement bonuses do not stack. Instead the largest bonus is applied.

```
itemproperty ItemPropertyEnhancementBonusVsAlign(
            int nAlignGroup,
            int nBonus )
```

This provides a weapon enhancement bonus *nBonus* when attacking creatures belonging to the alignment group *nAlignGroup*, which is a global constant of the form: IP_CONST_ALIGNMENTGROUP_.... This bonus is a positive integer ranging from 1 to 20.

```
itemproperty ItemPropertyEnhancementBonusVsRace(
            int nRace,
            int nBonus )
```

This adds a weapon enhancement bonus *nBonus* when attacking creatures from the racial group *nRace*, which is a IP_CONST_RACIALTYPE_... constant. This bonus is a positive integer ranging from 1 to 20.

```
itemproperty ItemPropertyEnhancementBonusVsSAlign(
            int nAlign,
            int nBonus )
```

This adds a weapon enhancement bonus *nBonus* when attacking creatures belonging to the specific alignment *nAlign*, which is an IP_CONST_ALIGNMENT_... constant. This bonus is a positive integer ranging from 1 to 20.

```
itemproperty ItemPropertyEnhancementPenalty(
            int nPenalty )
```

When this item property is applied to a weapon, it receives a penalty *nPenalty* to attack rolls and the amount of damage inflicted. The penalty is a positive integer from 1 to 5.

```
itemproperty ItemPropertyExtraMeleeDamageType(
            int nDamageType )
```

This item property can be applied to a melee weapon to change the type of the extra damage *nDamageType* inflicted on an successful melee attack. The bonus damage type is a IP_CONST_DAMAGETYPE_... constant, which is limited to the bludgeon, pierce or slash damage types. *Is this modifier only applied to damage inflicted from a Strength bonus?*

```
itemproperty ItemPropertyExtraRangeDamageType(
            int nDamageType )
```

This item property can be applied to a ranged weapon to change the type of the extra damage *nDamageType* inflicted on an successful ranged attack. The bonus damage type is a IP_CONST_DAMAGETYPE_... constant, which is limited to the bludgeon, pierce or slash damage types.

```
itemproperty ItemPropertyFreeAction()
```

The item gives the owner the free action ability, per that provided by the *freedom of movement* spell.

```
itemproperty ItemPropertyHaste()
```

The owner of this item gains the haste ability, per that provided by the wizard spell *haste*.

```
itemproperty ItemPropertyHealersKit(
            int nModifier )
```

This item functions as a healer's kit of level *nModifier*, which is a positive integer from 1 to 12.

```
itemproperty ItemPropertyHolyAvenger()
```

This item property makes the weapon function like a holy avenger item. *The specific benefits of this are not detailed.*

```
itemproperty ItemPropertyImmunityMisc(
            int nImmunityType )
```

The owner of the item imbued with this property gains immunity to a miscellaneous effect of type *nImmunityType*, which is a IP_CONST_IMMUNITYMISC_... constant. The immunity types include, paralysis, poison, knockdown, fear, disease and backstab.

```
itemproperty ItemPropertyImmunityToSpellLevel(
            int nLevel )
```

This property grants the item owner immunity to all spells up to level *nLevel*. The level is a positive integer from 1 to 9.

```
itemproperty ItemPropertyImprovedEvasion()
```

The item grants the owner the improved evasion special ability, as per the rogue special ability.

```
itemproperty ItemPropertyKeen()
```

A weapon with this property has its critical threat range doubled. Thus a threat range of 19-20 changes to 17-20.

```
itemproperty ItemPropertyLight(
            int nBrightness,
            int nColor )
```

This causes an item to radiate light with the intensity *nBrightness* and hue *nColor*. The brightness parameter is a constant of the form IP_CONST_LIGHTBRIGHTNESS_..., which defines the radius of the light:

- Dim – 5m
- Low – 10m
- Normal – 15m
- Bright – 20m

The hue is a IP_CONST_LIGHTCOLOR_... constant.

```
itemproperty ItemPropertyLimitUseByAlign(
          int nAlignGroup )
```

This property will limit the creatures that can use this item to members of the alignment group *nAlignGroup*, which is a IP_CONST_ALIGNMENTGROUP_... constant. If there are multiple such properties, the use is limited to members of the union of all the groups. It is possible to overcome this restriction with a suitable rank in the Use Magic Device skill.

```
itemproperty ItemPropertyLimitUseByClass(
          int nClass )
```

If a creature has a level in the class *nClass*, it can use this item. The class is a IP_CONST_CLASS_... constant. If there are multiple such properties, the use is limited to members of the union of all the groups. Note that it is possible to overcome this restriction with a suitable rank in the Use Magic Device skill.

```
itemproperty ItemPropertyLimitUseByRace(
          int nRace )
```

Creatures belonging to the race *nRace* can use this item. The class is a IP_CONST_RACIALTYPE_... constant. If there are multiple such racial restrictions applied, the item use is limited to members of the union of all the groups. Note that it is possible to overcome this restriction with a suitable rank in the Use Magic Device skill.

```
itemproperty ItemPropertyLimitUseBySAlign(
          int nAlignment )
```

This item property restricts the use of this item to creatures belonging to the alignment *nAlignment*, which is a constant of the form IP_CONST_ALIGNMENT_.... If there are multiple such alignment restrictions applied, the item use is limited to members of the union of all the groups. Note that it is possible to overcome this restriction with a suitable rank in the Use Magic Device skill.

```
itemproperty ItemPropertyMassiveCritical(
          int nDamage )
```

This item property inflicts additional damage *nDamage* on a successful critical hit. The damage is a global constant of the form IP_CONST_DAMAGEBONUS_..., which applies a constant or random damage bonus.

```
itemproperty ItemPropertyMaxRangeStrengthMod(
          int nModifier )
```

Application of this item property allows a strength modifier to a ranged attack. This modifier is limited to a maximum of *nModifier*, which is a positive integer between 1 and 20.

```
itemproperty ItemPropertyMonsterDamage(
          int nDamage )
```

This property is specifically for use with a monster's natural weapons, such as a claw or bite. It specifies the damage *nDamage* inflicted by the natural weapon as a global constant IP_CONST_MONSTERDAMAGE_....

```
itemproperty ItemPropertyNoDamage()
```

When applied to a weapon, this property causes the item to inflict no damage in combat. This is useful for balancing weapons that have special powers that are activated on a successful hit. See, for example, the two functions below.

```
itemproperty ItemPropertyOnHitCastSpell(
          int nSpell,
          int nLevel )
```

When this is applied to a weapon, it will cast the spell *nSpell* at level *nlevel* when an opponent is struck. For armor, it will cast the spell *nSpell* when the wearer is hit by an attack. The spell is a global constant of the form IP_CONST_ONHIT_CASTSPELL_....

Note that this item property can not be added to a ranged weapon such as a crossbow, even if it is applied via an On Acquire script.

```
itemproperty ItemPropertyOnHitProps(
          int nProperty,
          int nSaveDC,
          int nSpecial = 0 )
```

This item property applies an effect determined by the value *nProperty*, which is a subset of the global constants IP_CONST_ONHIT_.... (The list of effects is available in the command notes.) The save difficulty class to negate the effect is *nSaveDC*. Some of the effects require an additional parameter that is passed in the *nSpecial* field. For example, the IP_CONST_ONHIT_ABILITYDRAIN property will drain the IP_CONST_ABILITY_... ability passed in the *nSpecial* field.

```
itemproperty ItemPropertyOnMonsterHitProperties(
            int nProperty,
            int nSpecial = 0 )
```

This can be used to apply an effect to a natural monster weapon, such as slam or gore. The effect *nProperty* is a constant IP_CONST_MONSTERHIT_.... Some effects require that a second value be passed via the *nSpecial* parameter. For example, if *nProperty* is set to the constant IP_CONST_MONSTERHIT_DISEASE, the *nSpecial* is a DISEASE_... constant that determines the disease type.

```
itemproperty ItemPropertyReducedSavingThrow(
            int nBonusType,
            int nPenalty )
```

When applied to an item, this property will apply a penalty *nPenalty* to saving throws of the type *nBonusType*. The type is a IP_CONST_SAVEBASETYPE_... constant, while the penalty is a positive integer from 1 to 20.

```
itemproperty ItemPropertyReducedSavingThrowVsX(
            int nBaseSaveType,
            int nPenalty )
```

This property will reduce the saving throw by *nPenalty* against a type *nBaseSaveType* of effect or damage. The saving throw reduction is a positive integer from 1 to 20.

```
itemproperty ItemPropertyReduction(
            int nRegenAmount )
```

The owner of this item benefits from a regeneration ability that heals *nRegenAmount* per time interval.

```
itemproperty ItemPropertySkillBonus(
            int nSkill,
            int nBonus )
```

The item with this property grants a bonus *nBonus* to the skill *nSkill*. The skill is a SKILL_... constant, while the bonus is a positive integer from 1 to 50.

```
itemproperty ItemPropertySpecialWalk(
            int nWalkType = 0 )
```

This property causes the item owner to perform a special walk animation of the type *nWalkType*. A value of zero will cause a zombie walk animation. No other types are listed.

```
itemproperty ItemPropertySpellImmunitySchool(
            int nSchool )
```

The owner of this item gains immunity to spells from the *nSchool* magic school. The school is specified by a constant of the form IP_CONST_SPELLSCHOOL_.... This matches a row in the 'iprp_spellcost.2da' file.

```
itemproperty ItemPropertySpellImmunitySpecific(
            int nSpell )
```

This item property grants the owner immunity to the spell *nSpell*, which is a IP_CONST_IMMUNITYSPELL_... constant. Note that not every spell has a matching constant. The available immunities are listed in 'iprp_spellcost'. To provide immunity to a spell that is not on the list, you can apply an EffectSpellImmunity effect via a tag-based script.

```
itemproperty ItemPropertyThievesTools(
            int nModifier )
```

This provides a thieves tools item property with the modifier *nModifier*, which is a positive constant from 1 to 12. It provides a bonus while unlocking doors and chests.

```
itemproperty ItemPropertyTrap(
            int nTrapLevel,
            int nTrapType )
```

This returns a Trap item property. The *nTrapLevel* is one of the global constants IP_CONST_TRAPSTRENGTH_..., which can be minor, average strong or deadly. The *nTrapType* is a constant IP_CONST_TRAPTYPE_....

```
itemproperty ItemPropertyTrueSeeing()
```

This property gives the item owner true seeing, per the *true seeing* wizard spell.

```
Itemproperty ItemPropertySpellTurnResistance(
            int nModifier )
```

This gives a bonus *nModifier* to resist the cleric class turn undead ability.

```
Itemproperty ItemPropertyUnlimitedAmmo(
            int nAmmoDamage =
              IP_CONST_UNLIMITEDAMMO_BASIC )
```

This property provides unlimited use of an ammunition type. The *nAmmoDamage* parameter can be used to specify a special damage type by passing one of the IP_CONST_UNLIMITEDAMMO_... constants.

```
Itemproperty ItemPropertyVampiricRegeneration(
            int nRegenAmount )
```

This property provides vampiric regeneration to the maximum amount *nRegenAmount* per time unit, which is a positive integer from 1 to 20.

```
itemproperty ItemPropertyVisualEffect(
           int nEffect )
```

For a melee weapon, this will create a visual effect *nEffect*, which is a ITEM_VISUAL_... constant. The available types are acid, cold, electric, evil, fire, holy and sonic.

```
itemproperty ItemPropertyWeightIncrease(
           int nWeight )
```

This property causes an item weight increase *nWeight*, which is an IP_CONST_WEIGHT_INCREASE_... constant. This increase is a fixed number of weight in English pounds, rather than a proportion of the item weight. Note that this property to a suit of armor does not increase the weight.

```
itemproperty ItemPropertyWeightReduction(
           int nReduction )
```

This property causes an item weight decrease *nReduction*, which is an IP_CONST_REDUCED_WEIGHT_... constant. The reduction is as a percentage of the total weight.

Layout

The geography of the game is determined by a set of one or more areas that are organized into modules. A campaign consists of one or modules strung together in a sequence. The following routines are used to query and manage each of these organizational structures.

Areas

```
void ExploreAreaForPlayer(
           object oArea,
           object oPlayer,
           int nExplored = TRUE )
```

If this is called with *nExplored* set to true, it will expose the entire map of the area *oArea* to the player *oPlayer*. This only works for interior areas as exterior areas are automatically explored.

```
object GetArea( object oTarget )
```

This returns the area object where *oTarget* is currently located.

```
object GetAreaFromLocation( location locAt )
```

This returns the area object where the location *locAt* is positioned.

```
int GetAreaSize(
           int nAreaDimension,
           object oArea = OBJECT_IVNALID )
```

If *nAreaDimension* is set to AREA_HEIGHT, this will return the length of the area *oArea* as a number of tiles. Setting *nAreaDimension* to AREA_WIDTH returns the width in tiles. If the *oArea* is invalid, this routine will return the dimensions of the area that contains the object calling the routine.

```
object GetFirstArea()
```

This call returns the first area in the current module. It initializes the list of objects returned by GetNextArea.

```
int GetIsAreaAboveGround( object oArea )
```

If *oArea* is a valid area, this routine will return the constant AREA_ABOVEGROUND if the area has it's Underground property set to false. Otherwise it will return AREA_UNDERGROUND if the property is set to true.

```
int GetIsAreaInterior(
          object oArea = OBJECT_INVALID )
```

If *oArea* is a valid area, this will return true if either the Interior or Underground properties are set to true.

```
int GetIsAreaNatural( object oArea )
```

If the area *oArea* has the Natural property set to true, this will return AREA_NATURAL. Otherwise it will return AREA_ARTIFICIAL. If *oArea* is not a valid area, this routine will return AREA_INVALID.

```
int GetIsOverlandMap( object oArea )
```

If *oArea* is a valid area, this will return true if the area is flagged for use as an overland map.

```
object GetNextArea()
```

This routine will return the object representing the next area in the current module, or OBJECT_INVALID when it reaches the end of the list. The iteration is reset by a call to GetFirstArea.

```
object GetWaypointByTag(
          string sWaypointTag )
```

This call returns the first waypoint with the tag *sWaypointTag*, or an invalid object if there was no match.

```
void JumpPartyToArea(
          object oPartyMember,
          object oDestination )
```

The party containing the party member *oPartyMember* is jumped to the object *oDestination*. If this is in a different area, that area's Load Screen will be displayed while it is loading, then the 'On Client Enter Script' will be run once the party has finished moving to the destination. If the party is already in the area, the result is a jump to *oDestination* that does not trigger the 'On Client Enter Script'. See the FiredFromPartyTransition routine.

```
void SetAreaTransitionBMP(
          int nPredefinedAreaTransition,
          string sCustomAreaTransitionBMP =
            "" )
```

This causes a custom bitmap to be displayed during an area transition. The *nPredefinedAreaTransition* parameter is a AREA_TRANSITION_... constant that causes the matching predefined bitmap to be displayed. However, if it is set to AREA_TRANSITION_USER_DEFINED, the bitmap filename passed via *sCustomAreaTransitionBMP* is used instead.

```
void SetMapPinEnabled(
          object oMapPin,
          int nEnabled )
```

If *oMapPin* is a waypoint object, then this call will set the map note enabled tag state to *nEnabled*. For an example, see the 'gtr_enable_map_note' script.

```
void SetRenderWaterInArea(
          object oArea,
          int bRender )
```

If *bRender* is true, this routine causes the water planes in the area *oArea* to be displayed. Otherwise the water planes are hidden.

Subareas

A sub-area is an encounter region, trigger region or an area of effect object. See the 'Area of Effect' section of Effects.

```
object GetFirstSubArea(
          object oArea,
          vector vPosition )
```

This call returns the first sub-area in the area *oArea* at the vector location *vPosition*. Multiple overlapping sub-areas can occur at a position.

```
int GetIsInSubArea(
          object oCreature,
          object oSubArea = OBJECT_SELF )
```

If the creature *oCreature* has triggered the 'On Enter Script' event for a sub-area *oSubArea*, this routine will return true. Note that the creature does not need to remain located in the sub-area for this to return true.

```
object GetNextSubArea(
          object oArea )
```

Once GetFirstSubArea has been called in the area *oArea*, this routine will return the next sub-area at the same vector position as the first.

Campaigns

In the following calls, if you pass a player object as *oPlayer*, the variable will pertain to that player.

```
void DeleteCampaignVariable(
          string sCampaignName,
          string sVarName,
          object oPlayer = OBJECT_INVALID )
```

This will flag the variable with the name *sVarName* from a campaign database *sCampaignName*. The variable type does not matter, as the name must be unique.

```
void DestroyCampaignDatabase(
          string sCampaignName )
```

This call will delete the campaign database with the name *sCampaignName*. If the database does not exist, nothing happens.

```
{type} GetCampaign{Type} (
          string sCampaignName,
          string sVarName,
          object oPlayer = OBJECT_INVALID )
```

This will read a variable *sVarName* of type {type} from a campaign database with the name *sCampaignName*. Valid types are float, int, location, string and vector. The variable name must be unique in the database; you can not use different variable types with the same name. If *oPlayer* is provided, then the variable will pertain to that particular player.

```
void PackCampaignDatabase(
          string sCampaignName )
```

This removes any records that have been marked for deletion from the campaign database *sCampaignName*. The DeleteCampaignVariable routine can be used to delete individual campaign variables.

```
object RetrieveCampaignObject(
          string sCampaignName,
          string sVarName,
          location locLocation,
          object oOwner = OBJECT_INVALID,
          object oPlayer = OBJECT_INVALID )
```

This call will retrieve a creature or item object that was stored under the name *sVarName* from the campaign database *sCampaignName*. If a creature *oOwner* is specified, the call will try to place the object in the creature's inventory. Otherwise it will be placed on the ground. When *oObject* is invalid, the object will appear at the location *locLocation*. See StoreCampaignObject.

```
{type} SetCampaign{Type} (
          string sCampaignName,
          string sVarName,
          {type} tValue,
          object oPlayer = OBJECT_INVALID )
```

This will write a variable *sVarName* of type {type} to a campaign database with the name *sCampaignName*. Valid types are float, int, location, string and vector. The variable name must be unique in the database; you can not use different variable types with the same name. If you pass a player object as *oPlayer*, the variable will pertain to that player.

```
int StoreCampaignObject(
          string sCampaignName,
          string sVarName,
          object oObject,
          object oPlayer = OBJECT_INVALID )
```

This routine can be used to store a creature or item object *oObject* as a variable with the name *sVarName* in the campaign database *sCampaignName*. If the operation is successful, this routine will return true. Use the routine RetrieveCampaignObject to recover the object.

Modules

See also the Module Item Scripts section.

```
object GetModule()
```

Get the object representing the module. It returns OBJECT_INVALID on an error.

```
string GetModuleName()
```

This returns the module name for the language of the server where it is running.

```
int GetModuleXPScale()
```

This returns the 'XP Scale' property for the current module. The default value is 10.

```
void LoadNewModule(
          string sModuleName,
          string sWaypoint = “” )
```

This routine saves out the state of the currently running module then shuts if down and launches the *sModuleName* module. All of the connected players are moved to the starting location of the new module. If *sWaypoint* is not a null string, then the party is positioned at the waypoint that has this tag. See StartNewModule.

```
void SetModuleXPScale(
          int nXPScale )
```

This modifies the experience point scale for the running module to *nXPScale*, which is an integer between 0 and 200. The default value is 10, but this can be changed in the XP Scale property field. The modified XP scale is preserved across saves.

```
void StartNewModule(
          string sModuleName,
          string sWaypoint = "" )
```

This is similar to the LoadNewModule routine, except that the currently running module is not saved.

Objects

This sections describes functions that are used for non-static objects, including especially doors and placeables. Some object types are covered in more detail within their own sections. See Creatures, Inventory and Layout.

```
object CreateObject(
          int nObjectType,
          string sTemplate,
          location locAt,
          int bUseAppearAnimation = FALSE
          string sNewTag = "" )
```

This call will create an object of type *nObjectType* at location *locAt* using a blueprint template resource reference string equal to *sTemplate*. The *nObjectType* is a constant of type OBJECT_TYPE_..., but this is limited to a creature, item, placeable, store or waypoint. (Note that this does not include a door type.) If *bUseAppearAnimation* is true, then the object will gradually fade into view rather than suddenly appearing. If *sNewTag* is a non-empty string, then the object will be given that as its tag rather than the default.

In the particular instance of the 'Ipoint' placeable from the MISC PROPS, note that there is an extra space at the end of the resource reference (and the tag) in the blueprint. Thus, when calling CreateObject to create an 'Ipoint', the *sTemplate* string should be set to "`plc_ipoint `". Otherwise, you will get an invalid object.

```
object CopyObject(
          object oSource,
          location locAt,
          object oOwner = OBJECT_INVALID,
          string sNewTag = "" )
```

If object *oSource* is a valid creature or item, this call will produce a duplicate at location *locAt*. If the object is an item and *oOwner* is valid, then the item will be placed into *oOwner*'s inventory. If *sNewTag* is a non-empty string, then the object will be given that as its tag rather than the default.

```
void DestroyObject(
          object oDestroy,
          float fDelay = 0.0f,
          int nDisplayFeedback = TRUE )
```

After a delay of *fDelay* seconds, this will irrevocably destroy a non-static object *oDestroy*, such as a creature or item. If the item is in the inventory of a PC, setting *nDisplayFeedback* to true will cause a notice to appear in the chat window. This call has no effect on areas or campaigns.

For sound objects that are currently playing, you will first need to call SoundObjectStop before destroying the object. Otherwise, the sound will keep running in the game.

```
int PlayCustomAnimation(
          object oObject,
          string sAnimationName,
          int nLooping,
          float fSpeed = 1.0f )
```

This will cause a valid object *oObject* to play the animation file *sAnimationName*, which should be a gr2 file. If *nLooping* is true, then the animation will loop. The *fSpeed* parameter controls the rate of play for the animation, with the default at 1.0 and negative values causing the animation to play in reverse.

The 'ga_play_custom_animation' script describes several special symbols that can be passed in the *sAnimationName* field. In particular, '*' at the start of the name will fill in the prefix of an animation. A '%' will reset the creature to its default animation. A list of potential animation files can be seen in 'nwn2_animstan.2da' and combat animations in 'nwn2_animcom.2da. (The actual gr2 files are located under the lod-merged folders of the NWN2 installation directory, with names such as 'P_HHM_sneak.gr2'.)

Get Objects

```
object GetFirstObjectInArea(
          object oArea = OBJECT_INVALID )
```

This will return the first member of the list of objects located within the area *oArea*. If *oArea* is invalid, this will use the area where the calling object is located. See GetNextObjectInArea.

```
Object GetFirstObjectInShape(
          int nShape,
          float fSize,
          location locTarget,
          int bLineOfSight = FALSE,
          int nObjectFilter =
                    OBJECT_TYPE_CREATURE,
          vector vOrigin = [0.0,0.0,0.0] )
```

This function finds the first object inside a geometric volume defined by *nShape* and *fSize*.

nShape	fSize
SHAPE_SPHERE	Sphere radius
SHAPE_SPELLCYLINDER	Cylinder length
SHAPE_CONE	Widest radius of cone
SHAPE_CUBE	½ the length of a side

The *locTarget* parameter defines the center of the shape. If *bLineOfSight*, only objects that are within the line of sight of the *locTarget* are returned. The *nObjectFilter* is a integer containing OR'd bit flags, as defined by OBJECT_TYPE_..., that are used to filter the selected objects. For example:

```
nObjectFilter =
   OBJECT_TYPE_CREATURE | OBJECT_TYPE_DOOR
```

will only select creatures or doors. The *vOrigin* is used for cylinders and cones to establish the origin of the effect.

Example script: 'nw_s0_fireball'.

```
object GetNearestObject(
          int nObjectType = OBJECT_TYPE_ALL,
          object oTarget = OBJECT_SELF,
          int nNth = 1)
```

This returns the *nNth* nearest object of type *nObjectType* to the object *oTarget*. The object type is a global constant of type OBJECT_TYPE_.... If no object matching the criteria is found, this returns an invalid object.

```
object GetNearestObjectByTag(
          string sTag,
          object oTarget = OBJECT_SELF,
          int nNth = 1 )
```

This returns the *nNth* closest object with the tag *sTag* to the object *oTarget*.

```
object GetNearestObjectByLocation(
          string sTag,
          location locAt,
          int nNth = 1 )
```

This returns the *nNth* closest object with the tag *sTag* to the location *locAt*.

```
object GetNextObjectInArea(
          object oArea = OBJECT_INVALID )
```

After GetFirstObjectInArea has been called on the area *oArea*, this will return the next object in the area each time it is called. When the list is exhausted, it will return an invalid object.

```
Object GetNextObjectInShape(
          int nShape,
          float fSize,
          location locTarget,
          int bLineOfSight = FALSE,
          int nObjectFilter =
                    OBJECT_TYPE_CREATURE,
          vector vOrigin = [0.0,0.0,0.0] )
```

This function finds the next object inside a geometric volume defined by *nShape* and *fSize*. See GetFirstObjectInShape for an explanation of the parameters. Calling GetFirstObjectInShape reinitializes the search sequence. Example script: 'nw_s0_fireball'.

```
object GetObjectByTag(
          string sTag,
          int nNth=0 )
```

There can be multiple instances of objects with tag string *sTag*. This function returns the Nth instance of an object with tag *sTag*. This returns a valid result for *nNth* equal to one of 0, 1, ... n-1, where n is the number of objects with the matching tag.

If there is no match, it returns OBJECT_INVALID. If *nNth* is not specified, 0 is used.

Management

```
void SetDescription(
          object oTarget,
          string sDescription )
```

This will change the description of the object *oTarget* to *sDescription*. This only functions for creatures, items and placeables.

```
void SetFirstName(
          object oTarget,
          string sFirstName )
```

If *oTarget* is a valid object, this routine will set the first name of that object to *sFirstName*. This can be useful when you want to change the name of a creature as a result of a conversation. For an example, see the 'ga_first_name_set' script.

```
void SetHardness(
          int nHardness,
          object oObject )
```

If *oObject* is a door or placeable, this will set the hardness rating to *nHardness*, which must be an integer between 0 an 250.

```
void SetLastName(
          object oTarget,
          string sLastName )
```

If *oTarget* is a valid object, this routine will set the last name of that object to *sLastName*. For an example, see the 'ga_last_name_set' script.

```
void SetPlotFlag(
          object oTarget,
          int bPlotFlag )
```

This call will change the 'Plot' property of the object *oTarget* to *bPlotFlag*.

```
void SetScale(
          object oObject,
          float fX,
          float fY,
          float fZ )
```

This will change the base scale for the object *oObject* to the *fX*, *fY*, and *fZ* scales. By default these are set to 1.0 unless modified in the blueprint or another SetScale call.

Immediate Actions

```
void JumpToLocation(
          location locDest )
```

The calling object is jumped to the location *locDest*. This action is inserted at the top of the action queue.

```
void JumpToObject(
          object oJumpTo,
          int nWalkStraightLineToPoint = 1 )
```

The calling object is jumped to the location of the object *oJumpTo*. The variable *nWalkStraightLineToPoint* is undocumented. This action is inserted at the top of the action queue.

```
void PlayAnimation(
          int nAnimation,
          float fSpeed = 1.0f,
          float fSeconds = 0.0 )
```

This call functions like ActionPlayAnimation, except the calling object executes the animation immediately. It causes the subject to move their body according to the animation *nAnimation*. Valid animations are specified by the ANIMATION_... constants, which are subdivided into fire-and-forget and looping-type animations. The speed of the animation is multiplied by *fSpeed*, and the duration of a looping animation is *fDuration* seconds. The fire-and-forget animations ignore the *fDuration* parameter.

Note that the PC races have most of the animations implemented, but not all creatures will perform all animations.

```
void SpeakString(
          string sStringToSpeak,
          int nVolume = TALKVOLUME_TALK )
```

The creature will immediately speak the string *sStringToSpeak* at the volume *nVolume*. Valid values for the volume are defined by the TALKVOLUME_... constants.

```
void SpeakStringByStrRef(
          int nStrRef,
          int nVolume = TALKVOLUME_TALK )
```

This will look up a string by reference *nStrRef* from the talk table then immediately cause the caller to speak it as per SpeakString at the volume *nVolume*. Valid values for the volume are defined by the TALKVOLUME_... constants.

Properties

```
string GetDescription(
          object oTarget )
```

This returns a string containing a description of the valid object *oTarget*. This only functions for creatures, items and placeables.

```
string GetFirstName( object oTarget )
```

This will return the first name of object *oTarget*. Valid objects are creatures, items and placeables. See also GetLastName.

```
int GetHardness( object oObject )
```

If *oObject* is a door or placeable, this will return the hardness rating. For other objects, this will return 0.

```
int GetHasInventory( object oObject )
```

This returns true if the object *oObject* has an inventory. Creatures, stores and containers can have an inventory.

```
int GetIsObjectValid( object oObject )
```

This function returns true only if *oObject* is a valid object.

```
string GetLastName( object oTarget )
```

This will return the last name of object *oTarget*. Valid objects are creatures, items and placeables. See also GetFirstName.

```
string GetName( object oObject )
```

For a valid object *oObject*, this will return the object's full name.

```
int GetObjectType( object oTarget )
```

This call will return the type of a valid object *oTarget* as a constant OBJECT_TYPE_.... If the object is invalid it will return -1.

```
vector GetPosition( object oTarget )
```

If *oTarget* is a valid object, this will return the position of the object as a vector. Otherwise, it will return a zero-length vector, [0.0, 0.0, 0.0].

```
int GetPlotFlag(
          object oTarget = OBJECT_SELF )
```

If *oTarget* is a valid target, this will return true if the object has the 'Plot' property set to true.

```
string GetResRef( object oObject )
```

This call returns the name of the resource reference of the template that was used to create the object *oObject*. An empty string is returned when there is no template, such as when the object is invalid.

```
float GetScale(
          object oObject,
          int nAxis )
```

When *nAxis* is set to a constant SCALE_..., this call will return the scale of the base object along that axis. Thus, for an *nAxis* value of SCALE_X, this will return the scale along the X axis. This does not include the effects of a EffectSetScale on the object dimensions.

```
string GetTag( object oObject )
```

This call returns the identifying tag of the object *oObject*. If *oObject* is not a valid object, this call returns an empty string.

Obstacles

These routines apply to doors, placeables and traps. See also the Saving Throws section.

```
void DoPlaceableObjectAction(
          object oPlaceable,
          int nPlaceableAction )
```

This routine causes the object calling this routine to perform the action *nPlaceableAction* on the placeable object *oPlaceable*. Valid actions are PLACEABLE_ACTION_... constants.

```
int GetIsOpen( object oObject )
```

If *oObject* is a door or placeable, this will return true if it is currently open.

```
int GetIsPlaceableObjectActionPossible(
          object oPlaceable,
          int nPlaceableAction )
```

This routine will return true only if the action *nPlaceableAction* can be performed on the placeable *oPlaceable*. The action is a PLACEABLE_ACTION_... constant.

```
object GetTransitionTarget(
          object oTransition )
```

If *oTransition* is a door or a trigger that is configured for a transition, this call will return the destination. The resulting object is a door or a waypoint.

```
int GetUseableFlag(
          object oObject = OBJECT_SELF )
```

If *oObject* is a valid object, this will return true if the 'Usable?' property is set to true.

```
void SetFortitudeSavingThrow(
          object oObject,
          int nNewFortSave )
```

This is used to change the fortitude saving throw modifier *nNewFortSave* on a door or placeable *oObject*. Valid values for the fortitude save are integers from 0 to 250. This has no effect on other object types. See GetFortitudeSavingThrow and FortitudeSave.

```
void SetReflexSavingThrow(
          object oObject,
          int nNewRefSave )
```

This is used to change the reflex saving throw modifier *nNewRefSave* on a door or placeable *oObject*. Valid values for the reflex save are integers from 0 to 250. This has no effect on other object types. See GetReflexSavingThrow and ReflexSave.

```
void SetUseableFlag(
          object oObject = OBJECT_SELF,
          int bUseableFlag )
```

For a non-static object *oObject*, this will set the 'Useable?' property to *bUseableFlag*. When true, the player will be able to click on the object and interact with it.

```
void SetWillSavingThrow(
          object oObject,
          int nNewWillSave )
```

This is used to change the willpower saving throw modifier *nNewWillSave* on a door or placeable *oObject*. Valid values for the willpower save are integers from 0 to 250. This has no effect on other object types.

Doors

See also GetLastClosedBy.

```
void DoDoorAction(
          object oDoor,
          int nDoorAction )
```

The action *nDoorAction*, a DOOR_ACTION_... constant, is performed on the door *oDoor*. Valid actions include bash, knock, open and unlock.

```
object GetBlockingDoor()
```

This call returns the door object that last blocked the progress of the calling object. That is, the last door that was opened, unlocked, disabled or bashed.

```
int GetIsDoorActionPossible(
          object oTargetDoor,
          int nDoorAction )
```

This routine will return true only if the action *nDoorAction* can be performed on the door *oTargetDoor*. The action is a DOOR_ACTION_... constant.

```
void UnlinkDoor( object oDoor )
```

This routine modifies a door *oDoor* so that it no longer serves as a transition link to another area. Once the door is unlinked, it can be deleted.

Lock

The following routines apply to doors and certain placeables. See also GetLastLocked and GetLastUnlocked.

```
string GetKeyRequiredFeedbackMessage(
          object oObject )
```

This will return the feedback message that a player will receive when they try to open the object *oObject* without having the required key in their inventory.

```
int GetLocked( object oTarget )
```

This will return true only if *oTarget* is a valid lockable object and is currently locked.

```
int GetLockKeyRequired( object oTarget )
```

This will return true if the door or placeable *oTarget* has the KeyRequired property set.

```
string GetLockKeyTag( object oTarget )
```

This returns the tag of an object that is required to unlock the object *oTarget*. For a door this is the 'Key Tag' property, while for a placeable this is the KeyName.

```
int GetLockLockable( object oObject )
```

Both doors and placeables have a Lockable property that indicates whether the object is lockable. This routine will return the current state of the Lockable property for the object *oObject*.

```
int GetLockLockDC( object oObject )
```

This call will return the 'Open Lock DC' of the door or placeable *oObject*. This is the difficulty class of an open lock skill check to unlock the object.

```
int GetLockUnlockDC( object oObject )
```

This call will return the 'Close Lock DC' of the door or placeable *oObject*. This is the difficulty class of an open lock skill check to lock the object.

```
void SetKeyRequiredFeedbackMessage(
          object oObject,
          string sFeedback )
```

If *oObject* is a door or placeable that requires a key to open, this call will cause the message *sFeedback* to be displayed when a player tried to open the door without the proper key in their inventory.

```
void SetLocked(
          object oObject,
          int bLocked )
```

This call sets the Locked property of the door or placeable object *oObject* to the boolean *bLocked*.

```
void SetLockKeyRequired(
          object oObject,
          int bKeyRequired = TRUE )
```

This will set the KeyRequired property of the door or placeable *oTarget*. If true, a key is required to open the object. See SetLockKeyTag.

```
void SetLockKeyTag(
          object oObject,
          string sKeyTag )
```

If the object *oObject* has been configured to require a key to open, this routine will set the tag *sKeyTag* of the key required to unlock the door or placeable.

```
void SetLockLockable(
          object oObject,
          int bLockable = TRUE )
```

This call sets the state of the Lockable property for a door or placeable *oObject* to *bLockable*. When true, the object can be locked.

```
void SetLockLockDC(
          object oObject,
          int nNewLockDC )
```

This call will set the 'Open Lock DC' of the door or placeable *oObject* to *nNewLockDC*, a value between 0 and 250. This is the difficulty class of an open lock skill check to unlock the object.

```
void SetLockUnlockDC(
          object oObject,
          int nNewUnlockDC )
```

This call will set the 'Close Lock DC' of the door or placeable *oObject* to *nNewUnlockDC*, a value between 0 and 250. This is the difficulty class of an open lock skill check to lock the object.

Traps

Only doors, placeables or triggers can be trapped objects. See also ItemPropertyTrap in the Item Properties section and the GetLastDisarmed routine.

```
object CreateTrapAtLocation(
          int nTrapType,
          location locAt,
          float fSize = 2.0f,
          string sTag = "",
          int nFaction
           = STANDARD_FACTION_HOSTILE,
          string sOnDisarmScript = "",
          string sOnTrapTriggered = "" )
```

This creates a square trap of type *nTrapType* with the tag *sTag* centered on the location *locAt*. Valid types are global constants TRAP_BASE_TYPE_.... The trap belongs to the faction *nFaction*, which is a global constant STANDARD_FACTION_.... If *sOnDisarmScript* is not an empty string, the object will run this script when the trap is disarmed. If *sOnTrapTriggered* is not an empty string, the object will run this script when the trap is triggered.

The properties of the created trap is determined by the values in the traps.2da file.

```
object CreateTrapOnObject(
          int nTrapType,
          object oObject,
          int nFaction,
           = STANDARD_FACTION_HOSTILE,
          string sDisarmScript = "",
          string sOnTrapTriggered = "" )
```

This will create a trap on the door or placeable *oObject* of type *nTrapType*. Valid types for the latter are global constants TRAP_BASE_TYPE_.... The trap belongs to the faction nFaction, which is a global constant STANDARD_FACTION_.... If *sOnDisarmScript* is not an empty string, the object will run this script when the trap is disarmed. If *sOnTrapTriggered* is not an empty string, the object will run this script when the trap is triggered.

The properties of the created trap is determined by the values in the traps.2da file.

```
int GetIsTrapped( object oObject )
```

This returns true if the door or object *oObject* is trapped.

```
object GetLastTrapDetected(
          object oTarget = OBJECT_SELF )
```

This returns the last trap to be detected by the target *oTarget*.

```
object GetNearestTrapToObject(
          object oTarget = OBJECT_SELF,
          int nTrapDetected  = TRUE )
```

This will find the nearest trap to the target *oTarget*, in the current area. If *nTrapDetected* is true, this will only return the nearest trap that has been detected by *oTarget*.

```
int GetTrapActive( object oObject )
```

If *oObject* is an active trap, then this will return true. Use SetTrapActive to change the active setting.

```
int GetTrapBaseType( object oTrap )
```

This call will return the base trap type on the object *oTrap*. This is a constant matching one of TRAP_BASE_TYPE_....

```
object GetTrapCreator( object oTrap )
```

This will return the creature object that created the trap *oTrap*. Traps created by the toolset will return a value of OBJECT_INVALID.

```
int GetTrapDetectable( object oTrap )
```

This returns true if the 'TrapDetectable?' property on the object *oTrap* is true. Use SetTrapDetectable to change the property.

```
int GetTrapDetectDC( object oTrap )
```

This returns the value of the 'Trap Detect DC' property of the trapped object *oTrap*. This is the difficulty class of a Search skill check to find the trap. Use SetTrapDetectDC to change this value.

```
int GetTrapDetectedBy(
          object oTrap,
          object oCreature )
```

This returns true if the creature *oCreature* has detected the trap *oTrap*.

```
int GetTrapDisarmable( object oTrap )
```

This returns true only if the 'Trap Disarmable?' property is set to true on the trapped object *oTrap*.

```
int GetTrapDisarmDC( object oTrap )
```

This call will return the difficulty class for disarming the trap *oTrap*.

```
int GetTrapFlagged( object oTrap )
```

The notes for this script say that it returns true if the trap *oTrap* has been flagged as visible to all creatures. *How is a trap flagged to be visible to all?*

```
string GetTrapKeyTag( object oTrap )
```

This returns the tag of the key that will disarm the trap oTrap. The tag can be set by a call to SetTrapKeyTag. *Is this the same as the key needed to open a door or placeable? What about a key for a trigger?*

```
int GetTrapOneShot( object oTrap )
```

This returns true if the trap *oTrap* does not reset after it has been triggered. This can be changed by a call to the function SetTrapOneShot.

```
int GetTrapRecoverable( object oTrap )
```

If the TrapRecoverable property is set to true on the trap *oTrap*, this call will return true. Per the game manual, a trap can be recovered on a Disable Trap skill check at the base disarm DC plus 10. Use SetTrapRecoverable to change this property.

```
void SetTrapActive(
          object oTrap,
          int bActive )
```

This call sets the active state of the trap *oTrap* to *bActive*. Use GetTrapActive to get the current state.

```
void SetTrapDetectable(
          object oTrap,
          int bDetectable = TRUE )
```

This will set the detectability of the trap *oTrap* to the value of *bDetectable*. Use GetTrapDetectable to get the current setting.

```
void SetTrapDetectDC(
          object oTrap,
          int nDetectDC )
```

This call with change the difficulty class of a Search skill check needed to detect the trap *oTrap*. Use the GetTrapDetectDC call to get the current value.

```
int SetTrapDetectedBy(
          object oTrap,
          object oDetector )
```

This call will mark the creature *oDetector* as having detected the trap *oTrap*. Use the GetTrapDetectedBy call to find out who detected a trap.

```
void SetTrapDisabled(
          object oTrap
```

This will disable the trap *oTrap*. As a consequence the trap is deleted on a trigger, or removed from a placeable or door.

A call to GetIsTrapped on the object will return false.

```
void SetTrapDisarmable(
          object oTrap,
          int bDisarmable = TRUE )
```

This call will set the 'Trap Disarmable?' property on a trapped object *oTrap* to *bDisarmable*. A call to GetTrapDisarmable will return the current setting.

```
void SetTrapDisarmDC(
          object oTrap,
          int nDisarmDC )
```

This routine will change the disarm difficult class of the trap *oTrap* to *nDisarmDC*. Use GetTrapDisarmDC to get the current value of the Disarm DC.

```
void SetTrapKeyTag(
          object oTrap,
          string sKeyTag )
```

Use this routine to make the object with the tag *sKeyTag* a key to disarm the trap *oTrap*. The current key tab is returned by GetTrapKeyTag.

```
void SetTrapOneShot(
          object oTarget,
          int bOneShot = TRUE )
```

If *bOneShot* is true, this call will cause the trap *oTarget* to only fire once. Otherwise it will reset after being triggered. Use the GetTrapOneShot call to determine if a trap is currently one-shot.

```
void SetTrapRecoverable
          object oTrap,
          int bRecoverable = TRUE )
```

If *bRecoverable* is true, a character can use a successful Disable Trap as DC plus 10 skill check to recover the trap *oTrap*. Use GetTrapDetectable to get the current setting.

Talents

See also the descriptions for ActionUseTalentAtLocation and ActionUseTalentOnObject.

```
int GetCreatureHasTalent(
          talent tTalent,
          object oCreature = OBJECT_SELF )
```

This routine is intended to return a boolean state that indicates whether the creature *oCreature* has the talent *tTalent*. *However, this call crashed the game when passed a talent generated by TalentSpell.*

```
talent GetCreatureTalentBest(
          int nCategory,
          int nCRMax,
          object oCreature = OBJECT_SELF,
          int nExcludedTalentsFlag = 0 )
```

This call searches through the talents possessed by the creature *oCreature* in the category *nCategory* and looks for the talent with the highest challenge rating that does not exceed *nCRMax*. The category is a global constant TALENT_CATEGORY_.... The *nExcludedTalentsFlag* parameter is a sum of TALENT_EXCLUDE_... boolean constants that can be used to exclude ability, item or spell talents.

```
talent GetCreatureTalentRandom(
          int nCategory,
          object oCreature = OBJECT_SELF,
          int nExcludedTalentsFlag = 0 )
```

This call will return a random talent belonging to the creature *oCreature* in the category *nCategory*, which is a global constant TALENT_CATEGORY_.... The parameter *nExcludedTalentsFlag* is a sum of TALENT_EXCLUDE_... boolean constants that can be used to exclude ability, item or spell talents.

```
int GetIdFromTalent( talent tTalent )
```

If *tTalent* is a valid talent, this will return a constant identifier. Depending on the talent type, this could be a FEAT_..., SKILL_... or SPELL_... The type can be obtained using GetTypeFromTalent.

```
int GetIsTalentValid( talent tTalen )
```

This returns true if *tTalent* is a valid talent.

```
int GetTypeFromTalent( talent tTalent )
```

This call returns a type TALENT_TYPE_... constant for the talent *tTalent*.

```
int Talent( talent tTalent )
```

For a valid talent *tTalent*, this will return a constant of the form TALENT_TYPE_..., which identifies the talent as a feat, skill or spell.

```
talent TalentFeat( int nFeat )
```

This creates a feat talent instance with a type *nFeat* matching the global constant FEAT_....

```
talent TalentSkill( int nSkill )
```

This creates a skill talent instance with a type *nSkill* matching the global constant SKILL_....

```
talent TalentSpell( int nSpell )
```

This creates a spell talent instance with a type *nSpell* matching the global constant SPELL_....

Feats

In the following calls, the feat identifiers correspond to the FEAT_... global constants, which match the row numbers in the feat.2da file. See also TalentFeat, ActionUseFeat, GetHasFeatEffect and ItemPropertyBonusFeat.

```
int FeatAdd(
          object oCreature,
          int nFeatID,
          int bCheckRequirements,
          int bFeedback = FALSE,
          int bNotice = FALSE )
```

This call can add the feat *nFeatID* to the creature *oCreature*, then return true if the feat was successfully added. If *bCheckRequirements* is true, the creature must satisfy the requirements before the feat can be added. If *bFeedback* is true, feedback will be printed in the owning layer's chat window. If *bNotice* is true, a notice message will be printed on the player's screen.

```
void FeatRemove(
          object oCreature,
          int nFeatID )
```

This routine strips the feat *nFeatID* from the creature *oCreature*.

```
int GetHasFeat(
          int nFeat,
          object oCreature = OBJECT_SELF,
          int nIgnoreUses = FALSE )
```

This returns true if the creature *oCreature* has the feat *nFeat*, and the feat has uses remaining. The feat parameter is a FEAT_... constant. If *nIgnoreUses* is true, the routine ignores whether the creature has any feat uses remaining.

```
int GetMetaMagicFeat()
```

If the last spell cast by the caller had a metamagic type, this routine will return it's value as a METAMAGIC_... constant. A value of METAMAGIC_NONE indicates no metamagic type was used.

```
int GetSpellFeatId()
```

If a spell ability is being used that is provided by a feat, this script will return the feat identifier, FEAT_.... This is intended for use in a spell script.

Use Limited Feats

```
void DecrementRemainingFeatUses(
          object oCreature,
          int nFeat )
```

For a feat *nFeat* that has a number of uses per day, this routine will decrement the total available to the creature *oCreature* by one. The *nFeat* parameter is a global constant FEAT_... that has a number of uses per day. For example, FEAT_EXTRA_RAGE.

```
void IncrementRemainingFeatUses(
          object nCreature
          int nFeat )
```

This allows the creature *nCreature* to use the *nFeat* feat one additional time during the day. The *nFeat* is a global constant FEAT_....

```
void ResetFeatUses(
          object oCreature,
          int nFeatID,
          int bResetDailyUse,
          int bResetLastUseTime )
```

If the feat *nFeatID* has a number of uses per day, this call can reset the number of uses available to the creature *nFeatID*. If the feat *nFeatID* has a cool down time, this call will reset the last use time. The *nFeatID* is a global constant FEAT_..., or use FEAT_INVALID to reset all. The *nResetDailyUse* and *bResetLastUseTime* parameters are not documented, but may be used with the FEAT_INVALID option to reset feat categories.

Skills

See also ActionUseSkill.

```
int GetHasSkill(
          int nSkill,
          object oCreature = OBJECT_SELF )
```

This will return true only if the creature *oCreature* has the skill *nSkill* and the skill is usable. The skill is a SKILL_... constant.

```
int GetIsSkillSuccessful(
          object oTarget,
          int nSkill,
          int nDifficulty )
```

Return true if the result of a 1d20 roll plus the *nSkill* rank of creature *oTarget* is at or above the *nDifficulty* score. The skill *nSkill* is a SKILL_... constant.

```
int GetSkillRank(
          int nSkill,
          object oTarget = OBJECT_SELF,
          int bBaseOnly = FALSE )
```

This returns the number of skill ranks the object *oTarget* has in the skill *nSkill*, which is a SKILL_... constant. If *bBaseOnly* is true, this returns the base skill ranks without modifiers. If the target is untrained, this call will return 0. If the target doesn't have the skill, this returns -1.

Spells

See also ActionCastSpell..., ActionCastFakeSpell..., ActionCounterSpell, TalentSpell, SetEffectSpellId, GetAreaOfEffectSpellId, GetHasAnySpellEffect and GetSpellResistance.

```
void DecrementRemainingSpellUses(
          object oCreature,
          int nSpell )
```

If the creature *oCreature* has a number of uses per day of a spell *nSpell*, then this will decrement the total by one. The spell identifier is a SPELL_... global constant.

```
int GetArcaneSpellFailure(
          object oCreature )
```

This routine returns the arcane spell failure percentage for the creature *oCreature*.

```
int GetCasterLevel( object oCreature )
```

This routine returns the caster level for the last spell or spell-like ability cast by the creature *oCreature*. If the creature has not cast a spell, this returns zero.

```
int GetDefensiveCastingMode(
          object oCreature )
```

This call fetches the defensive casting mode of the creature *oCreature*. This is a global constant of the form DEFENSIVE_CASTING_MODE_..., which at present only returns true when the mode is activated or false when disabled.

```
object GetAttemptedSpellTarget()
```

This returns the creature last targeted by a spell cast by the object that is calling the routine. The result will change each time a spell is cast, and is cleared when combat ends.

```
int GetHasSpell(
          int nSpell,
          object oCreature = OBJECT_SELF )
```

This returns the number of instances of the spell *nSpell* that the creature *oCreature* has prepared. The *nSpell* parameter is a constant of type SPELL_....

```
int GetHasSpellEffect(
          int nSpell,
          object oObject = OBJECT_SELF )
```

This returns true only if the object *oObject* has any active effects that were applied by the spell *nSpell*, a constant of type SPELL_....

```
int GetSpellFeatId()
```

When this is run from a spell script, if the spell is being cast as a feat ability, this will return the feat identifier as a FEAT_... constant.

```
int GetSpellId()
```

When this is run from a spell script, this will return the spell identifier as a SPELL_... constant.

```
int GetSpellLevel()
```

When this is run from a spell script, this will return the spell level (rather than the caster level) of the spell being cast.

```
int GetSpellSaveDC()
```

When called from the spell script of a creature or an area of effect object, this will return the difficulty class to save against the spell.

```
location GetSpellTargetLocation()
```

This returns the location of the last spell cast by the caller.

```
object GetSpellTargetObject()
```

This will return the object that was the target of the last spell cast by the caller.

```
void RefreshSpellEffectDurations(
          object oTarget,
          int nSpellId,
          float fDuration )
```

This call will reset the duration of all effects on the target *oTarget* that originated with spell *nSpellID* to *fDuration* seconds. The spell identifier *nSpellId* is a SPELL_... constant.

```
int ResistSpell(
          object oCaster,
          object oTarget )
```

This routine will perform a spell resistance check by *oTarget* against a spell cast by *oCaster* and return true if the spell was resisted. Spell resistance functions like an armor class against spell attacks, and the actual spell level is not a consideration.. The return value includes information about how the spell was resisted. See the call notes for more details.

```
void SpawnSpellProjectile(
          object oSource,
          object oTarget,
          location locSource,
          location locTarget,
          int nSpellID,
          int nProjectilePath )
```

This creates the visual effect for a spell projectile. The source of the projectile *oSource* is at the location *locSource*. The target of the projectile *oTarget* is at location *locTarget*. The *nSpellID* is a SPELL_... constant that specifies the spell type. The *nProjectilePathType* is a global constant of the form PROJECTILE_PATH_TYPE_... that determines the path type of the projectile.

Time

```
void DayToNight(
          object oPlayer
          float fTransitionTime = 0.0f )
```

In an area that is not using the day/night cycle, this call transforms the sky's appearance to night for the player *oPlayer*. According to the call notes, the *fTransitionTime* parameter is not currently used.

```
int GetCalendarDay
```

This returns the current in-game day of the month.

```
int GetCalendarMonth()
```

This returns the current in-game month of the year.

```
int GetCalendarYear
```

This returns the current in-game year number.

```
int GetIsDawn()
```

Return true if the in-game clock is in the dawn time range.

```
int GetIsDay()
```

Return true if the in-game clock is in the daytime range.

```
int GetIsDusk()
```

Return true if the in-game clock is in the dusk time range.

```
int GetIsNight()
```

Return true if the in-game clock is in the night time range.

```
int GetTimeHour()
```

Returns the current game hour. This is an integer value ranging from 0 through 23, inclusive. Each game hour lasts a number of game minutes equal to the 'Minutes To Game Hour' property of the module; by default this is set to 2. The hour labels can be found in 'time.2da'.

```
int GetTimeMillisecond()
```

Returns the current game millisecond.

```
int GetTimeMinute()
```

Returns the current game minute. A game minute is equal to a real-world minute, but the number of minutes in an hour is determined by the 'Minutes To Game Hour' property of the module. At the default value of 2 minutes per game hour, this routine will return an integer value of 0 or 1.

```
int GetTimeSecond()
```

Returns the current game second. A game second is equal to a real-world minute, and there are sixty game seconds in a game minute.

```
float HoursToSeconds( int nHours )
```

This uses the 'Minutes To Game Hour' property of the module to convert *nHours* to a number of seconds. If the default value of 2 minutes to the game hour is set, the result will be *nHours* × 2 × 60 seconds.

```
void NightToDay(
          object oPlayer,
          float fTransitionTime = 0.0f )
```

In an area that does not use the day/night cycle, this changes the current day/night cycle to daylight for the player *oPlayer*. The *fTransitionTime* is non-functional.

```
float RoundsToSeconds(
          int nRounds )
```

This routine converts *nRounds* into a number of seconds, where a round is 6 seconds in length.

```
void SetCalendar(
          int nYear,
          int nMonth,
          int nDay )
```

Add *nYear* years, *nMonth* months and *nDay* days to the current date. Valid values are 0<= *nYear* <= 32000, 0 <= *nMonth* <= 12 and 0 <= *nDay* <= 28.

```
void SetTime(
          int nHour,
          int nMinute,
          int nSecond,
          int nMillisecond )
```

This routine will change the in-game time to the settings passed as arguments. According to the documentation, valid values are as follows:

- *nHour* – From 0 to 23 hours
- *nMinute* – From 0 to 59 minutes
- *nSecond* – From 0 to 59 seconds
- *nMillisecond* – From 0 to 999 milliseconds

Values larger than these range will be set to modulus the time range, with the excess being pushed to the next field.

Note that if the module is configured to use the default value of 2 'Minutes per Game Hour', a value of *nMinute* greater than two will be halved then added to the number of hours. Thus, a SetTime of 5 hours and 45 minutes will be converted to:

hours = (*nHour* + int(*nMinute*/2)) % 24 = 27 % 24 = 3

minutes = *nMinute* % 2 = 1

Hence, to set the time to three quarters of an hour past five a.m., *nMinute* should be set to `int(minutes / 30) % 2 = 1` and *nSecond* to 30 `2 × (minutes % 30) = 30`, where `minutes` is 45.

```
float TurnsToSeconds(
          int nTurns )
```

This routine will convert a number of *nTurns* turns to a number of seconds, with a single turn equal to 60 seconds.

Variables

Variables can be stored on objects within the game for persistence between game sessions. Each variable has a name string, which is limited to 32 characters.

```
int Random( int nMaxInteger )
```

This generates a random integer in the range from 0 to nMaxInteger-1 inclusive.

Strings

The following calls are used for string manipulation.

```
int CharToASCII( string sString )
```

This will take the first character of the string *sString* and convert it to the numerical ASCII value.

```
int FindSubString(
          string sString,
          string sSubString )
```

If the string *sSubString* exists within the string *sString*, this call will return the position of the first matching character. (The first character is at position zero.) If there is no match found, the call will return -1.

```
string GetStringLeft(
          string sString,
          int nCount )
```

This returns a string containing the left most *nCount* characters from the string *sString*.

```
int GetStringLength( string sString )
```

This returns the number of characters in the string *sString*.

```
string GetStringLowerCase(
          string sString )
```

This call returns a string containing a lower case version of the string *sString*.

```
string GetStringRight(
          string sString,
          int nCount )
```

This returns a string containing the right-most *nCount* characters from the string *sString*.

```
string GetStringUpperCase(
          string sString )
```

This call returns a string containing a upper case version of the string *sString*.

```
string GetSubString(
          string sString,
          int nStart,
          int nCount )
```

This routine will return a string containing *nCount* characters from *sString*, beginning with the *nStart* character. The first character in *sString* is at position 0.

```
string InsertString(
          string sDestination,
          string sInsert,
          int nPosition )
```

This will return a string containing a copy of *sDestination* with the string *sInsert* at the character position *nPosition*. The first character in *sDestination* is at position 0. If *sInsert* is past the end of *sDestination*, this will return *sDestination*.

```
int StringCompare(
          string sString1,
          string sString2,
          int bCaseInsensitive = FALSE )
```

If strings *sString1* and *sString2* are identical, this returns 0. Otherwise, this function will perform an ASCII sort order comparison, returning a negative value if *string1* is before *string2*, and a positive value otherwise. Thus, if you want to check for an exact match, compare the result against zero:

```
  if ( StringCompare( str1, str2 ) == 0 )
```

If *nCaseInsensitive* is true, the case of the string characters is ignored while performing the comparison.

```
int TestStringAgainstPattern(
          string sPattern,
          string sStringToTest )
```

The notes for this call state that it returns true if the string *sStringToTest* matches the pattern *sPattern*. It doesn't appear to use regular expression pattern matching or match simple sub-strings.

Storage

The following function descriptions represent multiple function calls that are used for variable storage and recovery. The notation {Type} indicates a variable type, which can be Float, Int, Location, Object or String. The valid types are listed for each function description. For campaign variables, see the Campaign section.

```
void DeleteLocal{Type}(
          object oObject,
          string sVarName )
```

This call is available for variables of type Float, Int, Location, Object and String. If the local variable of the same type with the name *sVarName* has been assigned to the object *oObject*, this call will remove it. Thus, DeleteLocalInt will delete a local integer.

```
{type} GetGlobal{Type}(
          string sVarName )
```

This call is available for variables of type Bool, Float, Int, and String. The call will return the value of the global variable with the name *sVarName*. Thus, GetGlobalBool will return a boolean value.

```
{type} GetLocal{Type}(
          object oObject
          string sVarName )
```

This call is available for variables of type Float, Int, Location, Object and String. The call will return the value of the local variable with the name *sVarName* that is assigned to the object *oObject*. Thus, GetLocalString will return a string value.

```
string GetVariableName(
          object oTarget,
          int nPosition )
```

Local variables are assigned to an object sequentially. This call will return the name of the variable at position *nPosition* on the object *oTarget*. If the position is invalid, this will return an empty string.

```
int GetVariableType(
          object oTarget,
          int nPosition )
```

This call will return the type of the variable at position *nPosition* on the object *oTarget*. The result is a global constant VARIABLE_TYPE_..., or -1 if the position is invalid.

```
void SetGlobal{Type}(
          string sVarName,
          {type} tValue )
```

This call is available for variables of type Bool, Float, Int and String. The call will set the global variable *sVarName* of type {type} to the value *tValue*. Thus, SetGlobalInt will set an integer value.

```
void SetLocal{Type}(
          object oObject
          string sVarName,
          {type} tValue )
```

This call is available for variables of type Float, Int, Location, Object and String. The call will set the object's *oObject* variable with the name *sVarName* to the value *tValue*. Thus, SetLocalLocation will set a location value.

Type Conversion

The following routines can be useful for various purposes, such as converting numbers to strings for presentation in the chat window.

```
int FloatToInt( float fFloat )
```

This function will convert the floating point value *fFloat* to the nearest integer value.

```
string FloatToString(
          float fFloat,
          int nWidth = 18,
          int nDecimals = 9 )
```

This call will return a string representation of the floating point *fFloat*. The string is *nWidth* characters wide and has *nDecimals*.

```
float IntToFloat( int nInteger )
```

This function will convert the integer *nInteger* to a nearby floating point value.

```
string IntToHexString( int nInteger )
```

This will convert the integer *nInteger* to a hexadecimal string, beginning with '0x' and followed by eight hex digits. See also the HexStringToInt function in ginc_math.

```
object IntToObject( int nInteger )
```

This call converts the integer value *nInteger* to an object. However, the object may be invalid.

```
string IntToString( int nInteger )
```

This will return a string representation of the integer *nInteger*.

```
int ObjectToInt( object oObject )
```

This call takes an object *oObject* and returns an integer value.

```
string ObjectToString( object oObject )
```

This will convert the object *oObject* to a hexadecimal string. See StringToObject.

```
float StringToFloat( string sNumber )
```

This call converts a string representation of a value to a floating point number.

```
int StringToInt( string sNumber )
```

This will convert the string *sNumber* to an integer.

```
object StringToObject( string sString )
```

The string *sString* is converted to an object. However, the object may be invalid. See ObjectToString.

Arrays.

The two-dimensional arrays provide tables of information that is accessed while the game is running. The individual tables can be viewed by selecting “2DA File...” from the “View” menu. The identifiers in some of the tables correspond to sets of global constants. Scripts can retrieve values from these files by means of the Get2DAString routine. The Data\2DA folder under the install directory contains some text files that describe the corresponding 2DA file format.

Some of the columns in the 2da files contain reference numbers to entries on the dialog.tlk file under the game's install directory. To view these entries, you can use a tool such as 'tlkedit', which is available for download from the internet.

Here are some of the 2da files available:

ambientsound – the ambient sound tracks. These correspond to the Ambient Sound menu items in the area properties. Thus, “Barrow Ambience” is al_mg_spiritbarrow_01 (= #201073).

armorrulestats – the game statistics for the armor and shield types. These are set via the Armor type field under Behavior.

backgrounds – the various backgrounds that can be selected during the character creation process. The values are passed as a parameter to the gc_background script. The identifiers match the BACKGROUND_... global constants.

baseitems – A table of base item types and associated parameters. The identifiers match the BASE_ITEM_... global constants.

cls_... – These are tables of data that are referenced by the class.2da file.

classes – This includes all of the classes used in the game. See the classes.txt file on the 2DA folder for more details.

crafting – This table gives crafting recipes for producing various items. For example, row 47 is the recipe for item tag NW_IT_MBRACER002, which is the Bracers of Armor +1. The required items are gem_03, n2_crft_ingiron and n2_crft_ingiron. These tags correspond to an Obsidian gem and two Iron Ingots.

creaturespeed – The movement types and rates set with the 'Walk Rate' properties field.

des_crft_spells – This file is used to determine which of the spells are allowed to be cast on a 'Blank Scroll', 'Magical Potion Bottle' or 'Magic Wand' item to create a new magic item. The *magic missile* spell, for example, can not be used to create a potion.

des_restsystem – This file is used for the wandering monster system.

disease – Parameter definitions for the available disease types, per the DISEASE_... constants. See EffectDisease.

domains – This array lists the available cleric domains and the spell powers provided at each spell level. The domains are selected by the Class Ability creature feats. The spells correspond to the SPELL_... constants and the rows in spells.2da. The following domains are included in this table but are not listed in the game manual: chaos, darkness, dream, fury, law, luck, time and undeath.

effecticons – The effect types that have icons defined by the EFFECT_TYPE_... constants. These are used to display active effects next to the characters in the party bar. See EffectEffectIcon.

feat – This long table defines the various feats available to party members and creatures. The Constant column corresponds to the FEAT_... constants. Other columns determine the prerequisites, whether the feat can be taken multiple times, and whether the feat is active and persistent.

iprp... – Various item property tables, including cost tables.

itemprops – the available item property types available for each item type. See the far right column for the item properties.

loadscreens – the load screens available during the transition to an area. The identifiers match the AREA_TRANSITON_... global constants.

nwn2_colors – the valid text color identifiers. These labels can be used to change the display color of a text string.

nwn2_deities – These are the various deities used in the game. The columns include the subraces (when AllRaces is 0), alignments, classes and genders that follow the deity. The subraces are defined in the racialsubtypes.2da file. Favored weapons are defined in the far right column.

placeables – This table lists data for the various placeables. Click on the NWN2_ModelName column to sort by tag.

polymorph – the available polymorph effects, as used in the ga_effect_polymorph script.

prioritygroups – the ranking of the sound priorities.

racialsubtypes – This lists the various racial subtypes as well as the monster types. It includes the ability adjustments, favored class, race-specific feat (per the RACE_... constants) and the matching RACIAL_... constants.

ranges – a list of ranges used with the creature 'Perception Range' setting.

reputation – this table is used to determine the reputation shift with the ga_reputation script.

spells – a long table of the spells and associated data. Among the columns are the spell levels for various classes, school, range, the visual and auditory effects, and spell immunities.

traps – contains values for detecting and disarming the various trap types.

visualeffects – the available visual effect identifiers, labels and parameters. The identifiers corresponds to the VFX_... global constants. See the visualeffects.txt file

- Label – The tag that corresponds to the global constant used in the ApplyEffect commands.
- Type_FD – D: duration effect/persistent; F: fire and forget effects that don't use duration; B: beam effects that can't be used like fire-and-forget
- ..._HeadCon_Node – the '.sef' files played at the source.
- ..._Impact_Node – the '.sef' files played at the impact.
- Shake... – used when screen shaking is associated with the effect.

Action Scripts.

The following scripts can be run as actions within a conversation. They are available from the conversation action tab's script menu when a new action is added. Each script name may be followed by one or more parameters. The script's action is described immediately after.

```
ga_align_good-evil
          int nActType
          int nAlignmentShiftRule
```

This script modifies the alignment of the speaker along the good-evil axis. The *nActType* parameter determines how much the act affects the alignment on a scale of -3 (fiendish) to +3 (saintly). The *nAlignmentShiftRule* determines how the shift is applied to the current alignment rating.

For *nAlignmentShiftRule* = 0, apply the shift normally. For a value of 1, a good act shifts toward good to a maximum alignment of 50; an evil act shifts toward evil to a maximum alignment of 50. This prevents the shift from moving the creature's alignment to the opposite end of the axis by a single act. Passing a 2 causes the shift to move toward neutral regardless of whether it was good or evil.

```
ga_align_law-chaos
          int nActType
          int nAlignmentShiftRule
```

This script functions like the ga_align_good-evil script, except that it modifies the alignment of the speaker along the law-chaos axis.

```
ga_alignment
          int nActType
          int bLawChaosAxis
```

This script changes the alignment of the player. The shift is determined by the value of *nActType*, which has the same range of values as for the ga_align_good-evil script. If *bLawChaosAxis* is true, the alignment shift is along the law-chaos axis. Otherwise the change is along the good-evil axis.

```
ga_area_transition
          string sDestination
          int bIsPartyTransition
```

This script causes a transition of the speaker to the area identified by *bIsPartyTransition*. If *bIsPartyTransition* is true, cause the associated party to transition as well.

```
ga_attack
          string sAttacker
          int bMaintainFaction
```

This causes the creature identified by *sAttacker* to initiate an attack against the speaker. If *bMaintainFaction* is false, change the faction of the attacker to Hostile. This should be placed on an [END DIALOG] entry.

```
ga_attack_target
          string sAttacker
          string sTarget
          int bMaintainFaction
```

The script will trigger an attack identified by *sAttacker* on *sTarget*. If *bMaintainFaction* is false, *sAttacker* will change to the Hostile faction.

```
ga_bark_trigger_reset
```

Reset the last bark trigger.

```
ga_blackout
```

This causes the screen to instantly fade to black.

```
ga_camera_facing_point_party
          string sTarget
          float fDistance
          float fPitch
          int nTransitionType
```

This script will turn the party cameras to face the target object identified by *sTarget*. The *fDistance* is the camera distance in meters. This is usually from 1 to 25, with 5 being optimal. An *fDistance* of 0 will use the current camera distance setting. The *fPitch* is an angle from 1 (overhead) to 89 (ground). A reasonable value is 60-70, and a value of 0 uses the current value. The *nTransitionType* sets the transition rate to between 1 and 100, with 0 being instantaneous.

```
ga_cast_fake_spell_at_object
          float fSecondsDelay
          string sCaster
          int nSpell
          string sSpellTarget
          int nProjectilePathType
```

This script causes the caster *sCaster* to execute the motions used to cast a spell at *sSpellTarget*, but without the actual spell effects. The *fSecondsDelay* gives the number of seconds before casting starts. The spell type, *nSpell*, is given by one of the SPELL_... global constants. If *sSpellTarget* is an empty string, the effect is targeted at the PC. The *nProjectilePathType* parameter is set to one of the PROJECTILE_PATH_TYPE_... global constants, or to PROJECTILE_PATH_TYPE_DEFAULT if the value is zero.

```
ga_cast_spell_at_object
          float fSecondsDelay
          string sCaster
          int nSpell
          string sSpellTarget
          int nMetaMagic
          int bCheat
          int nDomainLevel
          int nProjectilePath
          int bInstantSpell
```

This causes the target *sCaster* to cast a real spell at the target *sSpellTarget*. Most of the parameters are identical to those used with the *ga_cast_fake_spell_at_target* script.

The *nMetaMagic* parameter is set to a METAMAGIC_... global constant value. Use -1 for METAMAGIC_ANY or 0 for METAMAGIC_NONE. If *bCheat* is true, then the caster does not necessarily need to be able to cast the spell. The *nDomainLevel* parameter defines the caster level. If *bInstantSpell* is true, cast the spell immediately.

```
ga_clear_actions
          string sTarget
          int bClearCombatState
```

This script removes any actions from the *sTarget* creature's queue. If *bClearCombatState* is true, it will also stop combat.

```
ga_clear_comp
```

Remove any roster members from the speaker's party. The removed members are not despawned.

```
ga_clock_off
          int bAllPlayers
```

Turn off the clock for the PC speaker. If *bAllPlayers* is true, turn off the clock for everybody. This will allow a pre-combat discussion without expiring running spells.

```
ga_clock_on
          int bAllPlayers
```

As ga_clock_off, but turn the clock on.

```
ga_commandable
          string sTarget
          int bCommandable
```

Set the commandable state for the creature identified by *sTarget*. If *bCommandable* is true, make the creature commandable; otherwise creature is non-commandable.

```
ga_compshift
```

Currently this does nothing.

```
ga_conversation_self
          string sConversation
```

This starts a conversation *sConversation* with yourself; without another creature being involved. Possibly useful for narrative.

```
ga_create_object
          string sObjectType
          string sTemplate
          string sLocationTag
          int bUseAppearAnimation
          string sNewTag
          float fDelay
```

This script calls ActionCreateObject using the parameters. The *sObjectType* is a single character string indicating the object type: C=creature; P=placeable; I=item; W=waypoint; S=store; V=visual effect and L=light. The *sLocationTag* is the tag of a waypoint where the object will be created. This defaults to OBJECT_SELF.

```
ga_cutscene_move
          string sWPTag
          int nMilliseconds
          int bRun
          int string sMoverTag
```

This will delay the conversation for *nMilliseconds*/1000 seconds, and move the creature identified by *sMoverTag* to *sWPTag*.

```
ga_date_advance
          int nYear
          int nMonth,
          int nDay
```

Cause the date to advance by *nYear* = [0,32000] years, *nMonth* = [0,12] months and *nDay* = [0,28] days.

```
ga_date_set
          int nYear
          int nMonth
          int nDay
```

This sets the game calendar to a new date. Valid values are the year *nYear* between 1340 and 32001, the month *nMonth* between 1 and 12, and the day *nDay* between 1 and 28. Each year has 12 months and each month has 28 days.

```
ga_death
          string sTag
          int nInstance
```

Make *nInstance*'s of objects with tag *sTag* appear dead. Multiple, comma-separated tags can be passed in *sTag*.

```
ga_description_append
          string sTarget
          string sText
          int nStrRef
```

This script will append the string *sText* to the end of the target *sTarget*'s description, with no space. If sTarget is empty, the target is the PC speaker. If *nStrRef* is not zero, this script will then append the talk table string with reference *nStrRef* to the end of the description. (See *GetStringByStrRef*.)

```
ga_description_prepend
          int nStrRef
          string sText
          string sTarget
```

If *nStrRef* is not zero, this script will then insert the talk table string with reference *nStrRef* at the beginning of target *sTarget*'s description. (See *GetStringByStrRef*.) If sTarget is empty, the target is the PC speaker. The string *sText* will be inserted between the table string and the description, without spaces.

```
ga_description_set
          string sTarget
          string sText
          int nStrRef
```

This script will set the target *sTarget*'s description to the string *sText*, with no space. If sTarget is empty, the target is the PC speaker. If *nStrRef* is not zero, this script will then append the talk table string with reference *nStrRef* to the end of the description. (See *GetStringByStrRef*.)

```
ga_destroy
          string sTagString
          int iInstance
          float fDelay
```

This script will destroy the instance *iInstance* of the object with the tag *sTagString*. If *iInstance* is -1, all instances will be destroyed. The *sTagString* can include a comma-separated list (with no spaces) to destroy multiple objects with matching tags. The *fDelay* sets the time before the objects are destroyed.

```
ga_destroy_item
          string sTagString
          int nQuantity
          int bPCFaction
```

This script will remove *nQuantity* items matching *sTagString* from the PC's inventory. If *bPCFaction* is true, the items are removed from all members of the PC faction.

```
ga_destroy_party_henchmen
```

Destroy all henchmen belonging to the party.

```
ga_disable_scripts
          string sTarget
```

Save and clear the event handlers for the target *sTarget*. See *ga_enable_scripts*.

```
ga_donothing
```

This does nothing except print a debug message.

```
ga_door_close
          string sTag
          int nLock
```

Close the door with the tag *sTag*. If *nLock* is true, lock the door after it is closed.

```
ga_door_open
          string sTag
```

Unlock and open the door identified by sTag.

```
ga_effect
          string sEffect
          string sParams
          string sDuration
          int iVisualEffect
          string sTarget
```

Execute an effect on target *sTarget*. The effect *sEffect* can be one of the following strings: “AbilityIncrease”, “AbilityDecrease”, “Blindness”, “Damage”, “Death”, “Disease”, “Heal”, “Paralyze”, “Poison”, “Raise” or “Visual”. The *sParams* string provides the appropriate parameters for selected effects, as follows:

- AbilityIncrease or AbilityDecrease: ability_type, modify_by; The ability_type is Str=0, Dex=1, Con=2,Int=3, Wis=4, Cha=5. The modify_by is an integer amount by which to modify the ability.
- Damage: amount, type, power. The type is one of the DAMAGE_TYPE_... constant values; the power is one of the DAMAGE_POWER_... constant values.
- Disease: type. The type is one of the DISEASE_... constant values.
- Poison: type. The type is one of the POISON_... constant values.

The remaining effects do not require parameters. The duration string is either “I” for instant, “P” for permanent or “T,{duration}”. The last consists of a f-value constant. (For example: “T,10.0f”.) The *iFVisualEffect* is -1 for no visual, 0 for a standard visual or select a value from the visualeffects.2da file.

See the script comments for examples.

```
ga_effect_polymorph
          string sTarget
          int nPolymorphSelection
          int nLocked
          string sDurationType
          float fTemporaryDuration
```

Perform a polymorph effect on the target *sTarget*, which defaults to the speaker. The *nPolymorphSelection* is selected from the polymorph.2da table. If *nLocked* is 1, the player is unable to cancel the polymorph. The *sDurationType* is the duration type: “I”=instant, “P”=permanent and “T”=temporary. The last is set to the floating point duration specified by *fTemporaryDuration*.

```
ga_enable_scripts
          string sTarget
```

Restore the saved event handlers for target *sTarget*. See *ga_disable_scripts*.

```
ga_end_game
          string sEndMovie
```

End the current game, play the movie *sEndMovie* and then return all players to the main menu.

```
ga_equip_slot
          string sTarget
          int nSlot
          string sItemTag
```

Equip the *sSlot* inventory slot of creature *sTarget* with the item identified by tag *sItemTag*. The *sSlot* is one of the INVENTORY_SLOT_... global constants. These correspond to the equipment slots in the edit inventory dialog for a creature blueprint.

```
ga_explore_current_area
          int bWholeParty
```

This script reveals the map for the speaker, or, if *bWholeParty* is true, for the entire party. It exposes the entire map of the PC's current area.

```
ga_face_target
          string sFacer
          string sTarget
          int bLockOrientation
```

This script causes the creature *sFacer* to orient themselves toward the target *sTarget*. If *bLockOrientation* is true, the facing will remain in effect for the rest of the conversation.

```
ga_faction_join
          string sTarget
          string sTargetFaction
```

The target *sTarget* is made to join faction *sTargetFaction*. This can either identify a member of the faction, or be one of the standard factions: $COMMONER, $DEFENDER, $HOSTILE or $MERCHANT.

```
ga_faction_rep
          string sTarget,
          string sTargetFaction
          string sChange
```

This script adjusts the attitude of the faction *sTargetFaction* toward the target *sTarget* by the amount sChange. The target faction can be one of the standard factions: $COMMONER, $DEFENDER, $HOSTILE or $MERCHANT. Alternatively, it can be a creature that belongs to the target faction.

```
ga_fade_from_black
          float fSpeed
```

If the screen had been faded to a color using the ga_fade_to_black script, then this causes each player's screen to return from that color at the speed specified by *fSpeed*.

```
ga_fade_to_black
          float fSpeed
          float fFailsafe
          int nColor
```

This script will fade each player's screen to a color *nColor* at the speed *fSpeed*. The color is an integer equivalent to the hexadecimal values in the NWN2_Colors.2da file, with 0 for black and 1677725 for white. On a PC running Vista, the color hexadecimal values can be converted to integers as follows:

1. Choose the All Programs menu on the OS toolbar.
2. Launch the Calculator from the Accessories menu.
3. Select the View menu and choose Scientific.
4. Click on the Hex radio button.
5. Enter the six character hexadecimal value.
6. Click on the Dec radio button.

```
ga_first_name_append
          string sTarget
          string sText
          int nStrRef
```

This script can be used to append *sText* and the string reference *nStrRef* to the first name of the object *sTarget*. If no *sTarget* is provided, the first name of the PC_SPEAKER is modified. The script notes recommend setting sText to a space if this is only being used to append the string reference. See also the ga_last_name_... scripts.

```
ga_first_name_prepend
          string sTarget
          string sText
          int nStrRef
```

This is similar to ga_first_name_append, except that the strings are prepended to the first name. See also the ga_last_name_... scripts.

```
ga_first_name_set
          string sTarget
          string sText
          int nStrRef
```

This script changes the first name of the target *sTarget* to the concatenation of *sText* and the string reference *nStrRef*. If the sTarget string is empty, the first name of the PC Speaker is set instead. See also the ga_last_name_... scripts.

```
ga_flag_worldmap_autosave
          int bOn
```

This script changes the setting of the global integer CAMPAIGN_SWITCH_WORLD_MAP_AUTO_SAVE to the value of *bOn*. This variable is used in the DoShowWorldMap routine in the ginc_worldmap include file. If *bOn* is true, the DoShowWorldMap routine will perform a single player autosave .

```
ga_floating_str_ref
          int iStrRef
```

This will place a floating string above the PC speaker. The string is referenced by *iStrRef* using a GetStringByStrRef call.

```
ga_floating_text
          string sText
```

This script places the floating string *sText* above the PC speaker. This string will remain for the default duration of 5 seconds and can be seen by members of the same faction within 30 meters.

```
ga_force_exit
          string sCreatureTag
          string sWPTag
          int bRun
```

The creature identified by tag *sCreatureTag* will walk to the location of waypoint *sWPTag* then self-destruct by fading away. If *bRun* is true, the creature will run instead of walking.

```
ga_force_rest
          int bAllPartyMembers
```

The PC speaker is provided the benefits of a rest, including restoring hit points, recovering spells, resetting limited use feats, and so forth. If *bAllPartyMembers* is true, then the entire party is provided the same benefits.

```
ga_gint_max
          string sVariable
          string sChange
          int iMax
          int iRule
```

This script will set the value of the global integer variable *sVariable*. The *sChange* parameter can be a new value or it can perform an operation on the existing value. (See ga_global_int for details.) The *iMax* parameter specifies a maximum value for the variable, so if, for example, sChange causes an incremental increase, the value of *iMax* can be used to provide a cap to the value.

The parameter *iRule* determines the action taken when both the original value and modified value of *sVariable* exceed *iMax*:

iRule	**Result**
0	Ignore *iMax* and just apply the *sChange*.
1	Set the variable to *iMax*.
2	Use the lower of the original or final value for *sVariable*.

```
ga_gint_min
          string sVariable
          string sChange
          int iMin
          int iRule
```

This script will set the value of the global integer variable *sVariable*. The *sChange* parameter can be a new value or it can perform an operation on the existing value. (See ga_global_int for details.) The *iMin* parameter specifies a minimum value for the variable, so if, for example, *sChange* causes an incremental decrease, the value of *iMax* can be used to provide a cap to the value.

The parameter *iRule* determines the action taken when both the original value and modified value of *sVariable* exceed *iMin*:

iRule	**Result**
0	Ignore *iMin* and just apply the *sChange*.
1	Set the variable to *iMin*.
2	Use the greater of the original and final value for *sVariable*.

```
ga_give_feat
          string sTarget
          int nFeat
          int bCheckReq
          int bAllPartyMembers
```

Use this script to award the target creature *sTarget* the feat *nFeat*, which is equal to one of the FEAT_... constants (or see the row numbers in the feat.2da file). If *bCheckReq* is true, the feat is only awarded if the target meets the requirements. If *bAllPartyMembers* is true, then the feat is awarded to each member of the party. See also ga_remove_feat.

```
ga_give_gold
          int nGP
          int bAllPartyMembers
```

The PC speaker is given *nGP* gold pieces. If *bAllPartyMembers* is true, each member of the party is awarded *nGP* gold pieces.

```
ga_give_inventory
          string sSource
          string sTarget
          int iInventory
```

This script is used to transfer inventory from the creature *sSource* to *sTarget* en masse. The *sSource* parameter defaults to the conversation owner while *sTarget* defaults to the PC. If *iInventory* is 0, all of sSource's possessions and gold are given to *sTarget*. When *iInventory* is 1, only the non-equipped inventory is transferred, while for *iInventory* set to 2 only the equipped inventory is given.

```
ga_give_item
          string sTemplate
          int nQuantity
          int bAllPartyMembers
```

This script will provide the PC speaker with *nQuantity* items that have the Template ResRef string *sTemplate*. If the *nQuantity* is less than 1, it is set to 1. If the *bAllPartyMembers* parameter is true, then all members of the party will receive the same items.

```
ga_give_item_global_int
          string sItemRR
          string sGlobalNum
          int bAllPartyMembers
```

This script will provide the PC will one or more copies of the item specified by the resource reference *sItemRR*. The number of copies is determined by the value of the global integer with the name passed in the *sGlobalNum* argument. If *bAllPartyMembers* is true, then all party members receive the same number of the items.

```
ga_give_partial_quest_xp
          string sQuestTag
          int nPercentXP
```

This script will award all members of the party a percentage of the experience point award that is specified for the quest with the tag *sQuestTag*. Both the tag and the XP award should be listed in the module or campaign journal as a category. The value of *nPercentXp* is a percentage value between 0 and 100. If it is above 100, the value is limited to 100. The notes for this script suggest this for use with a quest that can be completed in several stages, providing a partial award at the end of each stage. See also ga_give_quest_xp.

```
ga_give_quest_xp
          string sQuestTag
```

Each member of the party will be awarded the experience for completion of the quest with the tag sQuestTag. The tag must be listed in the module or campaign journal as a category. See also ga_give_partial_quest_xp.

```
ga_give_xp
          int nXP
          int bAllPartyMembers
```

This script will award the PC with *nXP* experience points. If *bAllPartyMembers* is true, all members of the party will receive the experience points.

```
ga_global_float
          string sFloatName
          string sChange
```

This script modifies the value of the float global variable *sFloatName*. The *sChange* parameter can be a new value or it can perform an operation on the existing value. For example:

"3.2"	Set the value to 3.2f.
"++"	Add 1.0f to the current value.
"--"	Subtract 1.0f from the current value
"=-1.5"	Set the value to -1.5f.
"-10.1"	Subtract 10.1 from the value.
"-"	Subtract 1.0f from the value.

```
ga_global_int
          string sIntName
          string sChange
```

This script modifies the value of the integer global variable *sIntName*. The *sChange* parameter can be a new value or it can perform an operation on the existing value. For example:

"3"	Set the value to 3.
"--"	Subtract 1 from the current value
"-4"	Subtract 4 from the value.
"=-31"	Set the value to -31.

```
ga_global_string
          string sStringName
          string sChange
```

This script modifies the contents of the string global variable *sStringName* to *sChange*.

```
ga_group_formation_bma
          string sGroupTarget
          float fSpacing
          int iMoveType
```

This will cause the group containing member *sGroupTarget* to be placed into a staggered marching formation called "BenMa". Each member is separated by *fSpacing* from his neighbor. The *iMoveType* is a constant that defines how the members of the group will move to their destination. The valid constants are of type MOVE_... as defined in the ginc_group include file. The latter has a number of routines for creating groups and running various commands on the collective membership.

```
ga_heal_pc
          int nHealPercent
          int bAllPartyMembers
```

The PC is healed by an amount *nHealPercent* percent of his total hit points. Thus if the PC has lost half his hit points and *nHealPercent* is 50, the character will be fully healed. This is accompanied by the visual effect of a cure critical wounds spell. If *bAllPartyMembers* is true, then each party member is healed by the same percentage. The default value of 0 for *nHealPercent* will heal 100% of the damage.

```
ga_henchman_add
          string sTarget
          int bForce
          string sMaster
          int bOverrideBehavior
```

This script will attempt to add the creature *sTarget* to the party as a henchman of the character *sMaster*. The latter defaults to the PC speaker. If *bForce* is true, the creature will be added even if it will exceed the maximum allowed henchmen for the PC. If *bOverrideBehavior* is true, the creature's scripts will be replaced by the standard associate scripts (gb_assoc_...).

```
ga_henchman_remove
          string sTarget
          string sOptionalMasterTag
```

This will remove a henchman *sTarget* from the PC's party. If *sTarget* is a companion, this will have no effect. The *sOptionalMasterTag* parameter is a vestigial argument that has no effect.

```
ga_henchman_replace
          string sOld
          string sNew
          int bForce
          int bOverrideBehavior
          string sOptionalMasterTag
```

An existing henchman *sOld* of *sOptionalMasterTag* is removed from the party and *sNew* is added as a new henchman. The optional parameter *sOptionalMasterTag* defaults to the master used in the ga_henchman call for *sOld*, or else to the PC speaker. If *bForce* is true, the creature will be added even if it will exceed the maximum allowed henchmen for the PC. If *bOverrideBehavior* is true, the creature's scripts will be replaced by the standard associate scripts (gb_assoc_...).

```
ga_henchman_setmax
          string sChange
```

This will change the maximum number of henchmen allowed for the party. The parameter *sChange* is used in the same manner as *sChange* for the ga_global_int script. Thus an *sChange* value of "++" will increment the maximum henchmen by one.

```
ga_hotspot_vis
          int bVisible
          string sMap
          string sHotspot
```

This script is used in campaigns when you have built a world map with the World Map Editor. It sets the visibility of a hot spot on the map. The visibility state is determined by the value of *bVisible*, with 0 for false and 1 for true. The string *sMap* is the value of the WorldMapName in the map properties. The *sHotSpot* is the value of the Name in the map point property.

```
ga_hotspots_match_range_vis
          int bVisible
          string sMap
          string sCol
          int nMin
          int nMax
```

This script will set the visibility of a group of hotspots on a world map based on their resource reference number. The visibility state is set by the boolean *bVisible*. The string *sMap* is the value of the WorldMapName in the map properties. The string *sCol* is the name of the column of the map hotspots 2da file, which has the value <sMap>_hs.2da. All hotspots that have a resource reference number at or above *nMin* and less than *nMax* will be flagged. (This is true even if *nMin* > *nMax*.)

```
ga_hotspots_match_str_vis
          int bVisible
          string sMap
          string sCol
          string sMatchValue
          sMatchSpecifications
```

This script will set the visibility of a group of hotspots on a world map. The visibility state is set by the boolean *bVisible*. The string *sMap* is the value of the WorldMapName in the map properties. The string *sCol* is the name of the column of the map hotspots 2da file, which has the value <sMap>_hs.2da. If the string *sMatchValue* matches any values in the column, those hot spots will have their visibility state modified. The *sMatchSpecifications* parameter is currently unused.

```
ga_journal
          string sCategoryTag
          int nEntryID
          int bAllPartyMembers
          int bAllPlayers
          int bAllowOverrideHigher
```

This script will add an entry from the module or campaign to a player's in-game journal. The *sCategoryTag* must matches the Tag of a category in the journal, and *nEntryID* needs to match the ID of an entry within the category.

If *bAllPartyMembers* is true, the entry is added to the journal of all party members. If *bAllPlayers* is true, the journal entry is added to the journal of all players in the module. If *bAllowOverrideHigher* is true, the journal entry is allowed to override an existing journal entry from the same category that has a higher ID number.

Note that if this is this journal entry is marked as an end point, the journal entry will be placed in the completed section of the player's journal. However, you will need to award the experience points for quest completion using a separate script, such as with ga_give_quest_xp.

```
ga_jump
          string sDestination
          string sTarget
          float fDelay
```

The object *sTarget* will perform a direct jump to the object or waypoint *sDestination* after a delay of *fDelay* seconds. This jump functions like a magical teleport, skipping the intervening terrain. By default *sTarget* is the conversation owner.

```
ga_jump_faction
          string sTagOfMember
          string sTagOfWayPoint
          int iFormation
          string sFormationParams
```

This script uses the group functions defined in the ginc_group include file. It creates a group consisting of all members of the faction that includes *sTagOfMember*. The members of the group are then placed into a staggered marching formation called “BenMa”, which is essentially a filled diamond formation. The group is then jumped to the waypoint with the tag *sTagOPfWayPoint*.

The parameters *iFormation* and *sFormationParams* are not used in the script and can be ignored.

```
ga_jump_party_in_formation
          string sWaypoint
          int iFormationNum
          float fFormationTightness
          int bDisableNoise
```

This script will jump the entire party to the location of the waypoint *sWaypoint*. The *iFormationNum* is a parameter specifying a formation type, as follows:

iFormationNum	Formation
0	Filled diamond
1	Line
2	Half circle, facing out
3	Full circle, facing out
4	Rectangle

The *fFormationTightness* sets the spacing between the party members, which has a default value of 1.5f. If the *bDisableNoise* parameter is false, then the party members may be randomly shifted out of a perfect formation.

```
ga_jump_party
          string sDestTag
          int bWholeParty
          int bOnlyThisArea
```

This script will perform a jump movement of the party to the location of an object or waypoint identified by the tag *sDestTag*. If *bWholeParty* is false and this is a module only, only the PC makes the transition to the destination. The *bOnlyThisArea* parameter is unused and can be ignored.

```
ga_last_name_append
          string sTarget
          string sText
          int nStrRef
```

This script can be used to append *sText* and the string reference *nStrRef* to the last name of the object *sTarget*. If no *sTarget* is provided, the last name of the PC_SPEAKER is modified. The script notes recommend setting *sText* to a space if this is only being used to append the string reference. See also the ga_first_name_... scripts.

```
ga_last_name_prepend
          string sTarget
          string sText
          int nStrRef
```

This is similar to ga_last_name_append, except that the strings are prepended to the last name. See also the ga_first_name_... scripts.

```
ga_last_name_set
          string sTarget
          string sText
          int nStrRef
```

This script changes the last name of the target *sTarget* to the concatenation of *sText* and the string reference *nStrRef*. If the *sTarget* string is empty, the last name of the PC Speaker is set instead. See also the ga_first_name_... scripts.

```
ga_load_mod
          string sModule
          string sWaypoint
```

This script causes the module *sModule* to be loaded, then places the players at the module waypoint *sWaypoint*. The sModule must be the file name of a valid module without the '.mod' extension. If *sWaypoint* is not provided, the party will be placed at the default start location for the module. Note that this differs from the ga_start_mod script in that the module state is preserved from the last time the PCs visited.

```
ga_local_float
          string sVariable
          string sChange
          string sTarget
```

This will change the value of the floating point variable *sVariable* belonging to the target *sTarget*. If *sTarget* is undefined, it defaults to the conversation owner. The *sChange* parameter can be a new value or it can perform an operation on the existing value. See ga_global_float for examples.

```
ga_local_int
          string sVariable
          string sChange
          string sTarget
```

This will change the value of the integer variable *sVariable* belonging to the target *sTarget*. If *sTarget* is undefined, it defaults to the conversation owner. The *sChange* parameter can be a new value or it can perform an operation on the existing value. See ga_global_int for examples.

```
ga_local_string
          string sVariable
          string sValue
          string sTarget
```

This will change the value of the string variable *sVariable* belonging to the target *sTarget* to the string *sValue*. If *sTarget* is undefined, it defaults to the conversation owner.

```
ga_lock
          string sDoorTag
          int bLock
```

If *bLock* is true, then the door *sDoorTag* will be set to locked. Otherwise the door will be set to unlocked.

```
ga_lock_orientation
          string sTarget
          int bLock
```

If *bLock* is true, this script calls SetOrientOnDialog to lock the facing of the target creature *sTarget* so that it won't turn to face the current speaker during a conversation. Passing false for *bLock* enables the default conversation behavior.

```
ga_mapnote
          string sTag
          int bActive
```

This script can be used to control the visibility of a map note on the mini-map for an area. If *bActive* is true, the map note with the tag *sTag* will be enabled, otherwise it will be disabled.

```
ga_move
          string sWP
          int nRun
          string sTagOverride
```

This will cause a creature to perform a move action and travel to the waypoint *sWP*. If *nRun* is true, the creature will run instead of walking. The *sTagOverride* is the tag of the creature that will perform the move. If this field is blank, then the conversation owner performs the move.

```
ga_move_exit
          int nExitNumber
          int nRun
          string sWaypoint
          string sObject
```

This will cause the creature *sObject* to move to a waypoint with the tag *sWaypoint*, then exit. If *sWaypoint* is not specified, the script will search for a waypoint with the tag “nwc_exit” plus the string value of *nExitNumber*. Thus, if *nExitNumber* is 1, the expected tag is “nwc_exit1”. If nRun is true, the creature will run instead of walking.

```
ga_music_battle_play
```

This will cause the area's battle music to play.

```
ga_music_battle_restore
```

This script restores the area's battle music that was saved with the ga_music_battle_save script. The combination allows the battle music to be changed with the ga_music_battle_set, then restored to the original setting.

```
ga_music_battle_save
```

This script saves the name of the current battle music track to a local string variable attached to the area. It allows the music track to be restored with ga_music_battle_restore after it has been changed.

```
ga_music_battle_set
          int nTrack
```

This script will change the area's battle music track to a theme identified by the track number *nTrack* in the ambientmusic.2da file. The notes for this script list the full names of the tracks. See also the TRACK_BATTLE_... global constants.

```
ga_music_battle_stop
```

This deactivates the area's battle music.

```
ga_music_play
```

This will cause the area's background music to play.

```
ga_music_restore
```

This script restores the area's background music that was saved with the ga_music_save script. The combination allows the background music to be changed with the ga_music_set, then restored to the original setting.

```
ga_music_save
```

This script saves the name of the current background music track to a local string variable attached to the area. It allows the music track to be restored with ga_music_restore after it has been changed.

```
ga_music_set
          int nTrack
```

This script will change the area's background music track to a theme identified by the track number *nTrack* in the ambientmusic.2da file. The notes for this script list the full names of the tracks.

```
ga_music_stop
```

This deactivates the area's background music.

```
ga_notice_text
          string sText
          int nStrRef
```

This script will construct a string consisting of the contents of *sText* concatenated with the string reference identified by *nStrRef*. It will then post the resulting string to the player's “Notice Window GUI”. When I tested this out the string appeared on the screen in a manner similar to a notice about a Journal Entry.

```
ga_object_events_clear
          string sObjectTag
```

This script causes the various event handler scripts for the object *sObjectTag* to be stored as local variables on the object. All script properties are then cleared on the object, so that no events are handled. This script will only work on objects within areas. It fails when applied to areas or the module. See also ga_object_events_restore.

```
ga_object_events_restore
          string sObjectTag
```

If ga_object_events_clear was used to clear the event handler scripts on the object *sObjectTag*, this script will restore the properties to their original value. It loads the event handler scripts from local variables on the object.

```
ga_open_inventory
          string sCreatureTag
```

This script causes the inventory panel of the creature *sCreatureTag* to open for access by the player. This should only be run at the end of a cut scene conversation, or from a Neverwinter Nights 1-style dialogue. It could be used, for example, when selecting a container causes a conversation launch that determines what actions the player will take.

```
ga_open_store
          string sTag
          int nMarkUp
          int nMarkDown
```

This is the standard script for opening a store from a conversation with, say, a merchant. Typically it should be run as the last step in a conversation, or from a Neverwinter Nights 1-style dialogue. The store *sTag* must exist as a Store object within the module, or the dialog will fail. (Typically the store object is located with the merchant.) The *nMarkUp* and *nMarkDown* parameters are percentages that are used to modify the base price of an bought or sold item, respectively. These are modified by the result of a opposed appraise check by the current speaker, which can be adjusted by certain spells such as *charm person*.

```
ga_party_add
          string sRosterName
```

For a creature to be a member of the player's roster, it must have been assigned a roster name, such as by the ga_roster_add script. The ga_party_add script will add the creature with the roster name *sRosterName* to the PC's party. This will fail if the creature would exceed the maximum party size, or if the creature is not available.

```
ga_party_face_target
          string sFactionMember
          string sTarget
          int bLockOrientation
```

The members of the faction that includes *sFactionMember* will turn to face the subject *sTarget*. This is useful, for example, if the party is approached by an NPC because of a SpeakTrigger, and you want the remainder of the NPC's faction to turn and face the PC. If *bLockOrientation* is true, then the faction members will remain locked facing the same orientation for the remainder of the dialog.

```
ga_party_freeze
```

This causes all members of the PC's party to stand their ground. It sets the NW_ASC_MODE_STAND_GROUND variable to true, which is used in the gb_comp_heart companion heartbeat script to tell the companion not to follow the PC or enter combat. You would typically call this at the start of a conversation.

```
ga_party_size
          int nSize
```

This script will set the number of roster members that the player has available with the Party Selection interface to *nSize*. The default roster size is 3, not including henchmen.

```
ga_party_unfreeze
```

This will restore movement to the party after they were frozen by a call to the ga_party_freeze script. It does not cause the party members to continue the actions they were performing prior to the freeze. This script would typically be called at the end of a conversation where ga_party_freeze was run.

```
ga_play_animation
          string sTarget
          int nAnim
          float fSpeed
          float fDurationSeconds
          float fDelayUntilStart
```

The target *sTarget* will perform the animation with the tag nAnim, which should be set to an ANIMATION_... constant value. This will allow fire and forget, looping and placeable animations. It can be useful when the conversation node Camera Settings are configured to use a static camera. If *sTarget* is not set, the animation is applied to the conversation owner.

The speed of the animation is set by *fSpeed*, with 1.0 setting the speed to normal. For animations other than fire and forget, *fDurationSeconds* sets the length of the animation in seconds. (You may want to synchronize this with the Delay setting for the conversation mode.) If *fDelayUntilStart* is not zero, the target will wait this many seconds before running the animation.

```
ga_play_custom_animation
          string sTarget
          string sAnim
          int bLoop = 0
          float fDelayUntilStart = 0.0f
```

This will cause the target *sTarget* to play a custom animation per the PlayCustomAnimation function. The *sAnim* string is the name of a gr2 file that will be played. If *bLoop* is true, the animation will loop; otherwise it will run once. The *fDelayUntilStart* parameter is the number of seconds before the animation is played.

```
ga_play_sound
          string sSound
          string sTarget
          float fDelay
```

This script makes a sound *sSound* play at the location of the target *sTarget*. The sound parameter must be a valid '.wav' file without the extension. (For example, you could use the sounds from the 'Sounds' blueprints.) The sound will run after a delay of *fDelay* seconds. As near as I can tell, the sound runs at the peak volume setting.

Note that the script notes advise only using this on a creature; it may not play if assigned to an area or the module. It does work when played by a conversation with a door.

```
ga_play_voice_chat
          int nVoiceChat
          string sTag
```

This will play the voice chat *nVoiceChat* from the creature with the tag *sTag*. The *nVoiceChat* is one of the VOICE_... global constants.

```
ga_reequip_all_items
          string sTarget
```

The ga_remember_equipped script can be used to mark equipped items as local variables on the target. If any items were unequipped by *sTarget* as the result of a call to ga_unequip_hands or ga_unequip_slot, this script will re-equip the remembered items to their original inventory slots.

```
ga_refresh_timedate_tokens
```

This script updates date and time tokens for use in a conversation, based on the game world's date and time settings.

```
ga_remember_equipped
          string sCreature
```

This script will store the object in each inventory slot as a local variable on the *sCreature*. The objects can then be restored by a call to ga_reequip_all_items.

```
ga_remove_aoe
```

Any OBJECT_TYPE_AREA_OF_EFFECT objects in the current area are destroyed. These are objects that are created as the result of a spell.

```
ga_remove_comp
          string sHenchmanTag
```

This removes the henchman with the tag *sHenchmanTag* from the party. Note that if the creature is not a henchman but is on the roster, this will remove the creature from the party roster.

```
ga_remove_effects
          string sTarget
```

If *sTarget* is not blank, this call will remove all effects from the target. Otherwise it removes all effects from the party.

```
ga_remove_feat
          string sTarg
          string nFeat
          int bAllPartyMembers
```

This script removes the feat *nFeat* from the creature *sTarg*, where *nFeat* is the value of a FEAT_... global constant. If *bAllPartyMembers* is true, then all members of the party are stripped of the feat. See also ga_give_feat.

```
ga_replace_comp
          string sCompToRemove
          string sCompToAdd
```

This script removes the companion with the tag *sCompToRemove* from the party then adds the companion with the tag *sCompToAdd*. It does not abort if wither variable is not a valid tag for a companion.

```
ga_reputation
          int nActLevel
          int nRepOver
```

The player's reputation is tracked as a local variable on the PC, which determines whether the character is liked or disliked. This script is used to adjust the value of the player's reputation based on a particular act. The *nActLevel* variable is an integer variable in the range from +4 to -4, depending on how noble or villainous the act they performed. The shift in the reputation is determined by a lookup in the 'reputation.2da' file, with the column set to *nActlevel* and the row determined by the current reputation.

For example, if the current reputation is 54 then the reputation level is row 2 and the character is “Liked”. If *nActlevel* is set to -2, then the column named “Disliked” is used, for an adjustment of -6. The new reputation will be

48, which drops the character to “Known”. If the *nActlevel* was set to -3, the table lookup gives an adjustment of “S:1”. This causes an alignment shift to “Known” with a value of 38.

If *nRepOver* is passed to the script then the reputation is set to that value, overriding the current value and whatever was passed to *nActLevel*.

```
ga_reset_level
          string sCreature
          int bUseXPMods
```

Typically this would be run when a creature joins the party. The creature with the tag *sCreature* is set to the average party experience level. The creature is also force rested, thereby recovering hit points and spells. If *bUseXPMods* is true, then the creature's experience modifiers will be applied to the total before awarding the experience to the creature.

```
ga_reset_level_by_xp
          string sCreature
          int nXP
          int bUseXPMods
```

This script completely resets the creature *sCreature* to level zero, then grants the creature *nXP* experience points and automatically levels the creature to the maximum allowed level allowed for the creature's level-up package. If nXP is set to -1, then the current experience point total of the creature will be used instead. If *bUseXPMods* is true, then the creature's experience modifiers will be applied to the total before awarding the experience to the creature.

```
ga_rest
```

The current speaker performs a rest action.

```
ga_rest_convo
          int nAction
```

The script notes describe this as the master script for a conversation “rest”. The only value that will do anything is *nAction* equal to 100.

```
ga_restoration
          string sTarget
          int bFactionWide
          int bGroupWide
          int bSuppressVFX
```

This script will cast the spell 'greater restoration' on the creature *sTarget*, curing the subject of most negative effects. If *bFactionWide* is true, all members of the target's faction are cured. This can be used to cure the party, for example. The *bGroupWide* is used for groups, per the ginc_group include file calls. Setting *bSuppressVFX* to true will prevent the spell visual effect from playing.

```
ga_restore_equipped
          string sCreature
```

If the equipped items of the creature sCreature were recorded by a call to ga_remember_equipped, this script will restore the items to their previous slots.

```
ga_rm_go_to_hangout
          string sRosterName
```

The party member with the roster name *sRosterName* is removed from the party and sent to the hangout. The latter can be set using the ga_rm_set_hangout script. The default hangout spot for a roster member is “hangout_” plus the character's roster name. It tries to find an object with this tag (such as a waypoint) then dispatches the party member to that location.

```
ga_rm_set_hangout
          string sRosterName
          string sHangOutWPTag
```

This script changes the hangout spot for roster member *sRosterName* to the tag *sHangOutWPTag*. This tag is used with the ga_rm_to_hangout script.

```
ga_roster_add
          string sRosterName
          string sTemplate
```

This adds the creature with the template *sTemplate* to the roster using the 10-character name *sRosterName*. This name is used with other roster-related scripts to refer to this creature.

```
ga_roster_add_template
          string sRosterName
          string sTemplate
```

This is identical to ga_roster_add except that it has no debugging entries.

```
ga_roster_add_object
          string sRosterName
          string sTarget
```

This script will add the creature *sTarget* to the roster using the 10-character name *sRosterName*. The creature remains in the game world but is available for addition to the player's party. If *sTarget* is not specified, it defaults to the conversation owner.

```
ga_roster_campaignnpc
          string sRosterName
          int nCampaignNPC
```

If *sRosterName* is a valid member of the roster, this will make the creature persistent across all modules in the campaign. The creature will be selectable as a party member throughout the campaign.

I am uncertain about the purpose of *nCampaignPC*.

```
ga_roster_despawn
          string sRosterName
```

This function saves the creature with the roster name *sRosterName*, then despawns it. The notes for this script recommend that this should be the only script used to despawn party members. See also ga_roster_spawn.

```
ga_roster_despawn_all
          int bExcludeParty
```

This function will despawn all members of the roster, effectively performing a ga_roster_despawn on each member. If *bExcudeParty* is true, only roster members that are not currently in the party will be despawned.

```
ga_roster_gui_screen
```

This activates the graphical tool that allows the player to select the party members. Normally, party members can only be added or removed if they are selectable. See the ga_roster_selectable script.

```
ga_roster_party_add
          string sRosterName
```

This script will add the roster member with the roster name *sRosterName* to the player's party. For a multi-player game, this will fail if the roster member is already in a different player's party. When the roster member is added, it will appear at the PC's location.

```
ga_roster_party_remove
          string sRosterName
```

The party member with the roster name *sRosterName* will be removed from the party. The creature is not removed from the game and it's current state is not saved. Compare to ga_roster_despawn.

```
ga_roster_party_remove_all
          int bDespawnNPC
          int bIgnoreSelectable
```

This script will remove all selectable roster members from the player's party. If *bDespawnNPC* is true, the characters will also be despawned. If *bIgnoreSelectable* is true, all non-selectable party members will also be removed. See ga_roster_selectable.

```
ga_roster_remove
          string sRosterName
```

This script will remove the creature with the roster name *sRosterName* from the list of characters that can be selected as party members.

```
ga_roster_selectable
          string sRosterName
          int bSelectable
```

The roster graphical tool can add or remove party members can added to the party if they are selectable. This script will set the selectable state of roster member *sRosterName* to the boolean *bSelectable*. If a current party member is set to non-selectable, it can not be removed from the party.

```
ga_roster_spawn
          string sRosterName
          string sTargetLocationTag
```

This script will cause an instance of the roster member *sRosterName* at the location of the object with identifying tag *sTargetLocationTag*. If the creature already exists in the game, it will be moved to that location. See also ga_roster_despawn.

```
ga_roster_spawn_rand_loc
          string sRosterNameList
          string sTargetLocationTag
          float fRadius
```

The parameter *sRosterNameList* consists of a comma-separated list of roster names. This script will spawn each valid roster member in this list at a random location within a radius *fRadius* of the location of an object with the tag *sTargetLocationTag*.

```
ga_scripthidden
          string sTag
          int bHide
```

If *bHide* is true, this script causes the nearest creature with the tag *sTag* to disappear. Otherwise it will cause the creature to appear. The concealment is implemented by setting the creature's ScriptHidden property to true. Thus it will not render, does not interact via a collision and can not be selected by a player.

```
ga_set_animation_condition
          int nFlag
          int bState
          string sTarget
```

This script updates the value of the target sTarget's local string variable "NW_ANIM_CONDITION" that is used to store various animation flags. Valid flags are shown in the table below, and are defined as the NW_ANIM_FLAG_... hexadecimal constants in the 'x0_i0_anims' include file.

Value	Animation flag
1	NPC has been initialized
2	Animate each heartbeat
4	Use voicechats
8	NPC is animating
16	Interacting with a placeable
32	Has gone inside an interior area
64	The NPC has a home waypoint
128	Currently talking
256	The NPC is mobile
512	The NPC is mobile in a close range
1024	The NPC is civilized
2048	Will close an open door

Thus, passing a value of nFlag = 4 + 128 = 132 will set the the NPC to chatter and behave as though talking. If *bState* is true, the flag will be activated in the variable; otherwise it will be turned off.

```
ga_set_associate_state
          int nFlag
          int bState
          string sTarget
```

This script is used to set behavioral flags for the associate *sTarget*. The script notes say that some flags should not be modified by this script, but not which flags.

```
ga_set_flag
```

This script sets a global integer variable named "GlobalFlagName" to '1'. It's purpose is unknown to me.

```
ga_set_weapon_visibility
          string sTarget
          int bVisible
          int nType = 0
          int bForWholeGroup = 0
          int bForWholeFaction = 0
```

This script will set the visibility for a weapon belonging to *sTarget* by a call to SetWeaponVisibility. If *bVisible* is true, the weapon is set to visible; otherwise it is set to invisible. The script notes say that *nType* is not currently used, but this is passed as the third argument to SetWeaponVisibility. If *bForWholeGroup* is true, the script is applied to all members of a group, per the ginc_group include file. If *bForWholeFaction* is true, the script is applied to all members of a faction.

```
ga_set_wwp_controller
          string sTarget
          string sWalkWayPointsTag
```

A creature *sTarget* can be set to walk a set of waypoints with the tag that begins "WP_", followed by the creature tag, then by a numerical suffix. (See the section on Walk Path in the first volume.) This script will cause the creature *sTarget* to walk the set of way points that were intended for the creature with the tag *sWalkWayPointsTag*, rather than walking it's own set of waypoints.

```
ga_setbumpstate
          string sTarget
          int nBumpState
```

This will set the Bump State flag of the creature *sTarget* to *nBumpState*. For example, if you made a guard unbumpable so that it would stay in position and block access through an opening, this can be used to allow the guard to move aside and let a PC pass through.

```
ga_setimmortal
          string sTarget
          int bImmortal
```

This script can be used to change the Immortal property on the creature *sTarget* to the state *bImmortal*. An immortal creature can not be reduced below 1 hit point or slain. Making a creature immortal can be useful when you want an opponent to remain alive long enough to perform some action, such as teleporting away in the nick of time, surrendering to the party, or making a dying speech.

```
ga_setplotflag
           string sTarget
           bPlotFlag
```

This will set the Plot flag on creature *sTarget* to the state *bPlotFlag*. Changing a creature to a plot object makes it invulnerable to attacks.

```
ga_setrecipes
```

This script will set up all of the recipes for crafting items in global memory. This only needs to be run once in a module.

```
ga_sound_object_play
           string sTarget
```

This will activate the sound object *sTarget*, causing it to play it's sound. The object is still subject to the behavior limits regarding when the sound can be played, so it should not play at times when it is set not to make a sound. See also ga_play_sound, which allows a delay before playing the sound.

```
ga_sound_object_set_position
           string sTarget
           string sPositionTag
           float fXoffset
           float fYoffset
           float fZoffset
```

This script will set the location of a sound object *sTarget* to the location of *sPositionTag*, with an offset vector defined by (*fXoffset*, *fYoffset*, *fZoffset*). If *sPositionTag* is invalid or empty, the position is set to the origin of the area plus the offset vector.

```
ga_sound_object_set_volume
           string sTarget
           int iVolume
```

This script sets the volume of the target *sTarget* to the value *iVolume*. Valid values for *iVolume* are integers in the range 0 to 127, inclusive.

```
ga_sound_object_stop
           string sTarget
```

This will cause the sound object *sTarget* to stop playing.

```
ga_start_convo
           string sTarget
           string sConversation
           int bPrivateConversation
           int bPlayHello
           int bIgnoreStartDistance
           int bDisableCutsceneBars
```

This will immediately launch a new conversation *sConversation* with the target *sTarget*. It can be used to immediately jump from one conversation to another. If *bPrivateConversation* is true, then the conversation will only be audible to the PC. If *bPlayHello* is true, then the conversation owner will play an initial greeting sound.

If *bIgnoreStartDistance* is true, the conversation can start at any separation distance between the conversation owner and the PC. If *bDisableCutsceneBars* is true, cutscene bars will be disabled in NWN2-style conversations. This causes the camera display to fill the screen and text strings to appear on whatever background the camera provides.

```
ga_start_mod
           string sModuleName
```

This script runs the command that will launch the module *sModuleName*. The current module is shut down and all currently-connected players are shifted to the new module.

```
ga_summon_animal_companion
           string sTarget
```

If the target *sTarget* has an animal companion, this script will summon it into play. If *sTarget* is blank, this will summon the animal companion of the conversation owner.

```
ga_summon_familiar
           string sTarget
```

If the target *sTarget* has a familiar, this script will summon it into play. If *sTarget* is blank, this will summon the animal companion of the conversation owner.

```
ga_take_gold
           int nGold
           int bAllPartyMembers
```

This script will cause the amount *nGold* to be removed from the gold possessed by the player and passed to the conversation owner. If *bAllPartyMembers* is true, the gold is removed from each member of the party. This script does not check whether the characters actually have the gold.

You can use the condition script gc_check_gold to determine if a party member has the required amount of gold.

```
ga_take_item
          string sItemTag
          int nQuantity
          int bAllPartyMembers
```

This will remove the item with the tag *sItemTag* from the possession of the player. If *nQuantity* is greater than 1, that many items of the same type will be removed. Setting *nQuantity* to -1 will remove all such items. If the parameter *bAllPartyMembers* is true, the same number of the item will be removed from the inventory of all party members.

The ga_give_item script can be used to give the party members a temporary item before using ga_take_item to remove it later.

```
ga_talkto
          string sTarget
```

This will set a local variable on the conversation owner, indicating that you have talked to it. You can use the gc_talkto script to check if this variable has been set.

```
ga_time_advance
          int nHour
          int nMinute
          int nSecond
          int nMillisecond
```

This script will increment the game clock by *nHour* hours, *nMinute* minutes, *nSecond* seconds and *nMillisecond* milliseconds. Overflow values will be added to the next time field. Thus if the current time is 2000 hours and you add 10 hours, the clock will be advanced to 0600 on the following day. Negative values will have no effect.

```
ga_travel
          string sDestination
          string sTarget
          int bRun
          float fDelay
```

The target *sTarget* moves to the destination *sDestination* after a delay of *fDelay* seconds. If the *sDestination* is in the same area, the target will walk there. If *bRun* is true, the target will run rather than walk. When *sDestination* is in a different area, the target will move to the nearest door or waypoint, then perform a jump action to the destination.

```
ga_unequip_hands
          string sOwner
```

The creature *sOwner* will unequip items in the left and right hands. These items will be remembered so that they can be re-equipped by the ga_reequip_all_items script.

```
ga_unequip_slot
          string sTarget
          int nSlot
```

The creature *sTarget* will unequip the item in the slot *nSlot*. The *nSlot* parameter can be the value of any of the INVENTORY_SLOT_... global constants. The unequipped item is remembered so that it can be restored by the ga_reequip_all_items script.

```
ga_xp_award_2da_entry
          int iXPAwardID
          int bIndividualOnly
```

This script looks up an experience point award in the *iXPAwardID* row of the campaign's 'k_xp_award.2da' file. If *bIndividualOnly* is true, the experience will be awarded to the player. Otherwise the experience is awarded to the entire team.

Condition Scripts.

These scripts can be called as conditions within a conversation. The boolean return value is used to control whether a branch in the conversation tree is followed.

Some of these scripts use a mathematical rule that is passed as a string argument. These rules consist of an comparison operator followed by a numerical constant. The operator can be one of: '=', '<', '>' or '!', which test for 'equals', 'less than', 'greater than' and 'not equals', respectively. Thus, “>20” is used to check if an integer variable is above 20, “!5” is used to test if a value is not equal to five, and so forth.

```
gc_align_chaotic
```

This script returns true if the speaker has a chaotic alignment.

```
gc_align_evil
```

This script returns true if the speaker has an evil alignment.

```
gc_align_good
```

This script returns true if the speaker has a good alignment.

```
gc_align_lawful
```

This script returns true if the speaker has a good alignment.

```
gc_alignment
          int nCheck
          int bLawChaosAxis
```

This script returns true if a numerical value of the player's ethical alignment is greater than *nCheck*. The scale is from 3 to -3, with 0 being neutral. If the *bLawChaosAxis* variable is zero, check the good/evil axis. Otherwise, check the law/chaos axis.

As the good/evil and law/chaos values are normally stored on a scale of 0 to 100, the scale is converted to the -3 to +3 scale as follows:

Alignment 3 to -3 Value	**Equivalent 0 to 100 Value**
3	99
2	85
1	75
0	50
-1	25
-2	15
-3	5

```
gc_area
          string sArea
          string sCreature
```

This returns true if the creature with tag *sTarget* is in the area with the tag *sArea*. The *sCreature* variable defaults to OBJECT_SELF.

```
gc_background
          int nBackground
```

This returns true if the player has the specified value of *nBackground*. The background can be any of the BACKGROUND_... global constants.

```
gc_check_gold
          int nGold
          int bMP
```

If *nMP* is false, return true if the PC has more gold than *nGold*. If *nMP* is true, return true if every PC has more gold than *nGold*.

```
gc_check_item
          string sItemTag
          int bCheckParty
```

If *bCheckParty* is false, return true if the PC has the item identified by *sItemTag* in their inventory. If *bCheckParty* is true, return true only if every PC has the item in their inventory.

```
gc_check_level
          int nLevel
          int bMP
```

If *bMP* is false, return true if the PC has *nLevel* or more hit dice. If *bMP* is true, return true only if every PC has *nLevel* or more hit dice.

```
gc_check_race
          string sTarget
          string sRaceType
```

The string *sRaceType* contains the name of a racial type. This function returns true if the object specified by *sTarget* is of the same racial type. It prints an error string if an invalid race parameter is specified.

The possible values of *sRaceType* are: 'aberration', 'animal', 'beast', 'construct', 'dragon', 'dwarf', 'elemental', 'elf', 'fey', 'goblinoid', 'giant', 'gnome', 'halfelf', 'halfling', 'halforc', 'human', 'humanoid', 'magical_beast', 'monstrous', 'orc', 'ooze', 'outsider', 'reptilian', 'shapechanger', 'undead' or 'vermin'. The 'humanoid' type is a group that encompasses goblinoid, monstrous, orc and reptilian.

```
gc_check_race_pc
          string sRace
```

The racial type of the speaker is compared with the *sRace* parameter, and the script returns true if they match. This script recognizes a fixed set of race names or an equivalent number.

Name	Number
dwarf	1
elf	2
gnome	3
halfelf	4
halfling	5
halforc	6
human	7
outsider	8

This function will also recognize certain racial subtypes, as follows:

Name	Number
shielddwarf	11
moonelf	12
rockgnome	13
sr-halfelf	14
lightfoothalf	15
sr-halforc	16
sr-human	17
golddwarf	18
duergar	19
drow	20
sunelf	21
woodelf	22
svirfneblin	23
stronghearthalf	26
aasimar	27
tiefling	28

Finally, the following strings can be used to identify selected the so-called civilized racial subtype groupings.

- "civdwarves" or "41" – all of the dwarf subraces except the duergar.
- "civelves" or "42" – all of the elf subraces, except drow and wild.
- "civhalflings" or "43" – all of the halfling subraces, except ghostwise.

```
gc_check_stats
          int nStat
          int nValue
```

The *nStat* must set to one of the ABILITY_... global constants. This script will return true only if the value of the speaker's attribute score is at or above the *nValue* variable.

```
gc_class_level
          int nClass
          string sLevelCheck
          string sTarget
```

This script checks the number of class levels the character *sTarget* has in the class *nClass*. The *sLevelCheck* string is a integer comparison; see the CompareInts function in ginc_var_opts for details. The *nClass* must be equal to one of the CLASS_... global constants.

Example: for *nClass* = CLASS_TYPE_HARPER and *sLevelCheck* = ">2", the script returns true if the target has at least 3 levels in the Harper prestige class.

```
gc_comp_remove
          string sCompanion
```

If the creature with tag *sCompanion* is currently in the party and can be removed, return true. Otherwise return false.

```
gc_dead
          string sCreatureTag
```

Return true only if the creature with tag *sCreatureTag* is neither living nor undead.

```
gc_deity
          string sDeity
```

Returns true only if the deity of the speaking character is the same as *sDeity*. The deity name is case insensitive.

```
gc_distance
          string sTagA
          string sTagB
          string sCheck
```

The *sTagA* and *sTagB* arguments are the tags of two objects. This script finds the distance from the object with tag *sTagA* to the nearest object with tag *sTagB*. The *sCheck* variable is a floating point comparison statement consisting of a mathematical operator and a floating point value. If the distance from A to B satisfies the comparison statement, this script returns true. See the *CheckVariableFloat* function for more on the comparison operator.

```
gc_distance_pc
          string sTag
          string sCheck
```

This script is similar to gc_distance, except that the distance is between the object with tag *sTag* and the nearest PC.

```
gc_equipped
          string sItem,
          string sTarget,
          int bExactMatch
```

This script returns true only if the creature with tag *sTarget* has the item with tag *sItem* in their inventory. If *sTarget* is an empty string, use the speaking PC. If *sTarget* is the non-standard constant "$PARTY", then the entire party is checked. If *bExactMatch* is false, sItem can be a tag substring.

```
gc_false
```

This script will always return false. It can be used as a placeholder in a dialog.

```
gc_global_float
          string sVarName
          string sCheck
```

This script checks the value of the global floating point variable defined by *sVarName* using the *sCheck* mathematical comparison, returning true if the comparison is correct. See the *CheckVariableFloat* function for more on the comparison operator.

```
gc_global_int
          string sVarName
          string sCheck
```

This script checks the value of the global integer variable defined by *sVarName* using the *sCheck* mathematical comparison, returning true only if the comparison is correct. See the CompareInts function in ginc_var_opts.

```
gc_global_string
          string sVariable
          string sCheck
```

Return true if the value of the global string variable *sVariable* matches the contents of the string *sCheck*.

```
gc_has_feat
          string sTag
          int nFeat
          int bIgnoreUses
```

Check if the creature identified by *sTag* has the feat numbered *nFeat*. If *sTag* is blank, check the speaker. If *bIgnoreUses* is true, only check if the player has the feat. Otherwise, also check the number of uses left of the feat.

The feat number can be obtained from the value of the corresponding FEAT_... global constant. Alternatively, the lists under the feats tab for a character's property window will list the ID number for each feat.

```
gc_henchman
          string sTarget
          string sMaster
```

This script returns true if the creature identified by *sTarget* is a henchman of the creature identified by *sMaster*. If *sMaster* is blank, use the speaker. It returns false if either *sTarget* or *sMaster* is invalid for the area containing OBJECT_SELF.

```
gc_henchmen_max
          string sCheck
```

This script returns true only if the maximum number of henchmen allowed satisfies the integer comparison statement *sCheck*. See the CompareInts function in ginc_var_opts.

```
gc_is_enemy_near
          float fRadius
          int bLineOfSight
```

This script returns true if an enemy creature is within radius *fRadius* of the speaker. If *bLineOfSight* is true, the enemy creature must also be within the line of sight of the speaker for this function to return true.

```
gc_is_female
```

This script returns true if the speaker is female.

```
gc_is_in_combat
```

This script returns true if the speaker is currently in combat.

```
gc_is_in_party
          string sTargetTag
```

This script returns true if the creature identified by the tag *sTargetTag* is in the part of the speaker.

```
gc_is_male
```

This script returns true if the speaker is male.

```
gc_is_near
          string sTag
          string sObjTypes
          float fRadius
          int bLineOfSight
```

This script will return true if there is an object belonging to a set of types within a spherical radius *fRadius* of the speaker. The object must contain the string *sTag* and belong to the object type specified by *sObjTypes*. The string *sObjTypes* can be set to "ALL" or contain one or more of the following letters:

Letter	Object Type
C	Creature
I	Item
T	Trigger
D	Door
A	Area of Effect
W	Waypoint
P	Placeable
S	Store
E	Encounter

If *bLineOfSight* is true, the script will only return true if the object is within the line of sight of the speaker.

```
gc_is_open
          string sObjectTag
```

Return true if the object identified by *sObjectTag* is open. The object must be a placeable or a door.

```
gc_is_owner
          string sTag
```

This script will return true if the owner of the conversation has identifying tag *sTag*.

```
gc_is_skill_higher
          int nSkillA
          int nSkillB
          int bStrictlyGreaterThan
```

This script compares two of the speaker's skills and determines which one is greater. The value of *nSkillA* and *nSkillB* are determined by the following table:

Skill	Value
Appraise	0
Bluff	1
Concentration	2
Craft Alchemy	3
Craft Weapon	4
Diplomacy	5
Disable Device	6
Heal	7
Intimidate	8
Listen	9
Lore	10
Move Silently	11
Open Lock	12
Parry	13
Perform	14
Ride	15
Search	16
Craft Trap	17
Sleight of Hand	18
Spellcraft	19
Spot	20
Survival	21
Taunt	22
Tumble	23
Use Magic Device	24

Note that *Ride* is not a valid skill. If *bStrictlyGreaterThan* is true, then this script returns true only if the value of *nSkillA* is greater than *nSkillB*.

```
gc_item_count
          string sItemTag
          string sCheck
          int bPCOnly
```

This script checks the number of items with the tag *sItemTag* in the inventory of the PC or the party, then performs an integer comparison using the rule passed by *sCheck*. See the CompareInts function in ginc_var_opts. If *bPCOnly* is true, only check the speaker; otherwise check the party.

Example: if *sCheck* is "<5", return true if there are fewer than 5 items matching *sItemTag* in the inventory.

```
gc_journal_entry
          string sQuestTag
          string sCheck
```

This script will examine the state of the quest matching *sQuestTag*, then perform a check using the *sCheck* condition. This check is identical to the *sCheck* parameter in the gc_local_int call.

```
gc_local_float
          string sVariable
          string sCheck
          string sTarget
```

This script will compare the value of a local float variable *sVariable* belonging to *sTarget* using the math rule *sCheck*. However, '==' operations will usually fail because of accuracy. If sTarget is blank, use OBJECT_SELF. See the *CheckVariableFloat* function for more on the comparison operator.

```
gc_local_int
          string sVariable
          string sCheck
          string sTarget
```

This script will compare the value of a local integer variable *sVariable* belonging to *sTarget* using the math rule *sCheck*. If *sTarget* is blank, use OBJECT_SELF. See the CompareInts function in ginc_var_opts.

```
gc_local_string
          string sVariable
          string sCheck
          string sTarget
```

This script will compare the value of a local string variable *sVariable* belonging to *sTarget* against *sCheck*. It will return true if the strings match.

```
gc_module
          string sTag
```

This script compares the string *sTag* against the tag of the module and returns true only if they match.

```
gc_node
          int iNodeIndex
```

A variable *sn_NodeIndex* is set on a creature by gtr_speak_node. If the value of this variable is equal to *iNodeIndex*, set *sn_NodeIndex* to zero and return true.

```
gc_num_comps
          string sCheck
```

This script compares the number of companions in the party using the rule sCheck, returning true if the match comparison is correct. See the CompareInts function in ginc_var_opts.

```
gc_num_pcs
          string sCheck
```

This script compares the number of PCs in the party using the rule *sCheck*, returning true if the match comparison is correct. See the CompareInts function in ginc_var_opts.

```
gc_obj_valid
          string sTag
          int bAreaOnly
```

If the object specified by tag *sTag* is valid for the current area, return true. If *bAreaOnly* is false, check the entire module.

```
gc_rand_1of
          int iMax
```

This script will return true if the result of a random integer generation equals 1. This can be used to choose a random response by decrementing *iMax* by 1 for each response. For example:

- Response 1: iMax = 4.
- Response 2: iMax = 3.
- Response 3: iMax = 2.
- Response 4: Fall-through response.

```
gc_range_day
          int iStartDay
          int iEndDay
```

This script will return true if the current day is in the range *iStartDay* to *iEndDay*. The variables have a value between 1 and 28.

```
gc_range_hour
          int iStartHour
          int iEndHour
```

This script will return true if the current hour is in the range *iStartHour* to *iEndHour*. The variables have a value between 1 and 23.

```
gc_range_month
          int iStartMonth
          int iEndMonth
```

This script will return true if the current month is in the range *iStartMonth* to *iEndMonth*. The variables have a value between 1 and 12.

```
gc_range_year
          int iStartYear
          int iEndYear
```

This script will return true if the current month is in the range *iStartYear* to *iEndYear*. The default start year is 1372.

```
gc_reputation
          int nRepLevel
          int nRepNumber
```

If *nRepNumber* is not equal to zero, return true only if the player's reputation is at or above *nRepNumber*.

If *nRepNumber* is zero, this script gets the value of the local “Player's Reputation” variable and converts it to a scale between -4 and 4. It then returns true if *nRepLevel* is at or above the converted scale value.

Reputation	Level
To -110	-4
-109 to -80	-3
-79 to -50	-2
-49 to -25	-1
-24 to 24	0
25 to 49	1
50 to 79	2
80 to 109	3
110 or higher	4

See also ga_reputation.

```
gc_rm_in_hangout_area
          string sRosterName
```

This script determines if the roster member specified by *sRosterName* is in the same area as the PC hangout spot. See ga_rm_set_hangout.

```
gc_roster_available
          string sRosterName
```

If the roster member *sRosterName* is available to be added to the party, return true. This script will return false if the roster member is in another party or is currently unselectable.

```
gc_roster_campaignnpc
          string sRosterName
```

Return true only if the CampaignNPC flag of *sRosterName* is set to true. If *sRosterName* is not found, this script returns false. See also ga_roster_campaignnpc.

```
gc_roster_selectable
          string sRosterName
```

This script returns true if the roster member *sRosterName* is selectable. An NPC can be set to non-selectable for plot reasons, etc.

```
gc_singleplayer
```

This script returns true if this is a single-player game.

```
gc_skill_dc
          int nSkill
          int nDC
```

This script performs a skill check against skill *nSkill* at DC equal to *nDC*, then returns true if the check is successful. See gc_is_skill_higher for the list of skill indices.

```
gc_skill_rank
          int nSkill
          int nRank
```

This script returns true if the speaker has a rank in skill *nSkill* at or above *nRank*. See gc_is_skill_higher for the list of skill indices.

```
gc_talkto
          string sTarget
```

Return true if this is the first time the PC has talked to the target identified by *sTarget*. If *sTarget* is blank, the target is the owner of the dialog.

```
gc_time_of_day
          string sTimeOfDay
```

This script checks if the time of the day is within a range specified by *sTimeOfDay*. This string can have one of the following values:

String	Time interval
“DAWN”	The hour of dawn
“DAY”	The time between dawn and dusk
“DUSK”	The hour of dusk
“NIGHT”	The time between dusk and dawn
“NOON”	The hour of noon
“MIDNIGHT”	The hour of midnight

```
gc_true
```

This script will always return true. It can be used as a placeholder in a dialogue.

Include Files.

The standard toolset scripts access a set of include files via an include statement near the top. For example:

```
#include "ginc_actions"
```

is used to include 'ginc_actions' calls and constants. These include files can be examined by scrolling down the menu of available scripts to the set that begins with "ginc".

- **ginc_2da** – routines used for constants and data lookup functions for the 2DA files.
- **ginc_actions** – various calls to have an object perform an action. In general these are applied to the object that runs them, rather than having the object passed as a parameter.
- **ginc_ai** – artificial intelligence calls for determining behaviors.
- **ginc_alignment** – functions for adjusting character alignment. This is treated on a scale of 0 to 100, with subdivisions of 0–30, 31-69 and 70–100. When an alignment crosses into a different subdivision, it is adjusted to the middle of that subdivision. If the alignment of the PC is changed, characters in the same party can also receive an alignment adjustment.
- **ginc_autosave** – the auto-save functions for single player mode.
- **ginc_baddie_stream** – routines for steady streams of evil doers.
- **ginc_behavior** – routines for checking and changing the current creature behavior, such as determining if a creature is too busy to perform a task or how they react to a damaging spell.
- **ginc_companion** – utility functions for companions to the PC. These includes retaining appropriate companions when loading a module, transitioning all companions when travelling to a new area, adding or removing a companion, and spawning companions at a certain area.
- **ginc_crafting** – routines related to the crafting skills. These are used for the workbench placeables.
- **ginc_cutscene** – routines that are used in cut scenes, such as in NWN2-style conversations.
- **ginc_death** – routines for processing death or unconsciousness.
- **ginc_debug** – specialized routines that are used when debugging mode is active.
- **ginc_effect** – used to create or remove visual effects.
- **ginc_event_handler** – routines used to manage the event handling scripts for creatures, associates or objects.
- **ginc_group** – used to work with groups of objects, especially creatures. After a group has been created, various calls can be applied to all members. They can also be arranged in a formation.
- **ginc_gui** – calls for displaying the load game dialog or party selection interface.
- **ginc_henchman** – routines used in the henchman action scripts, for adding, removing and replacing henchmen, or modifying the maximum number of henchmen.
- **ginc_ipspeaker** – calls used with the Ipoint Speaker placeable for running a location-based conversation.
- **ginc_item** – these are routines that are used for items and inventory management, including functions for stores.
- **ginc_item_script** – a handful of routines for item scripts.
- **ginc_journal** – various calls related to journal categories and entries, including gold, item and experience point rewards.
- **ginc_math** – various math utility functions, including random value generators, tests for proximity, vector comparators, and hex string to number converters.
- **ginc_misc** – a handful of miscellaneous calls.
- **ginc_nx1spells** – a function for creeping cold.
- **ginc_object** – calls for creating objects at a waypoint or a tagged object location, as well as counting or destroying objects.
- **ginc_param_const** – this includes routines for

parsing parameters, converting string patterns to constants, and obtaining objects by tag or special identifier. It also defines standard constants used in other include files.

- **ginc_reflection** – this has functions for rotation and reflection operations. The latter can be used for beams reflecting off objects.
- **ginc _restsys** – these constants and functions are used for the wandering monster rest system.
- **ginc_roster** – these routines are for management of a roster. See the Faction section of the system functions.
- **ginc_sound** – a few utility routines for saving music tracks as variables then restoring them later.
- **ginc_symbol_spells** – these routines are specific to the “Symbol of” spells, such as *symbol of death*.
- **ginc_time** – various time management functions, including routines for time-based events.
- **ginc_trigger** – these functions are related to triggers. It includes routines for managing speak triggers.
- **ginc_utility** – miscellaneous sundry routines. Mostly for creating and spawning objects.
- **ginc_var_opts** – some of these routines use a mathematical comparison consisting of a relational operator followed by a constant. The operator can be '=', '<', '>' or '!'. If the operator is not specified, '=' is assumed. This function applies the comparison defined by the string to the value, returning true if the comparison is correct and false otherwise.
- **ginc_vars** – various calls for local and global variables, including modifying values, arrays, global objects and retrieving the prefix or suffix of a string.
- **ginc_worldmap** – constants and functions related to world maps.
- **ginc_wp** – these functions are used with scripted waypoints. For example, making NPCs stop and chat with each other.

- **nw_o[0-2]_...** – object scripts.
- **nw_s[0-3]_...** – spell hook scripts.
- **nw_t[01]_...** – trap scripts.
- **nwn2_inc_spells** – Utility spell functions.
- **x2_inc_craft** – This is the central script for player crafting through feats or skills. For example, it includes the 'CIGetSpellWasUsedForItemCreation' call that is executed by the X2PreSpellCastCode, which is run in each spell script. If the base item type is BASE_ITEM_BLANK_POTION (101, or 'Potion Bottle'), this runs the brew potion routine. This allows a Wizard with the Brew Potion feat to cast a spell on a 'Magical Potion Bottle', creating a potion of the type of the spell.

Function Index.

The following index lists the functions defined in the toolset include files, followed by the name of the include file where the function is implemented. (Most of the private functions have not been included.) If you paste this list into a script called '___INDEX' then wrap the entire text in comment delineators (/* ...content... */), you can use it to quickly lookup function locations.

Function calls are included from the following files: *ginc_*, inc_*, kinc_spirit_eater, nw_i0_*, nwn2_inc_*, x0_i0_*, x0_inc_*, x1_inc_*, x2_am_inc*, *x2_i0_spells* and *x2_inc_**.

ActionCreateObject – *ginc_actions, x0_i0_transform*
ActionFireEvent – *ginc_actions*
ActionForceExit – *ginc_actions*
ActionForceMoveToTag – *ginc_actions*
ActionOrientToObject – *ginc_actions*
ActionOrientToTag – *ginc_actions*
ActionPlayAnimationUncommandable – *ginc_actions*
ActionPrintString – *ginc_actions*
ActionPsionicCharm – *inc_mf_combat*
ActionPsionicMB – *inc_mf_combat*
ActionRemoveMyself – *ginc_actions*
ActionSpeakNode – *ginc_actions*
ActionSpeakOneLiner – *ginc_wp*
ActionStartBeam – *ginc_reflection*
ActionTurnHostile – *ginc_actions*
ActivateFleeToExit – *nw_i0_generic*
AddCompanionsToRosterList – *ginc_roster*
AddHenchmanToCompanion – *ginc_companion*
AddHighlight – *x0_i0_highlight*
AddItemPropertyAutoPolicy – *ginc_crafting*
AddNearestWithTagToGroup – *ginc_group*
AddTemporaryHighlight – *x0_i0_highlight*
AddToGroup – *ginc_group*
AdjustAlignmentLawChaos – *ginc_alignment*
AdjustAlignmentGoodEvil – *ginc_alignment*
AdjustAlignmentOnAll – *x0_i0_partywide*
AdjustReputationWithFaction – *x0_i0_partywide*
AdvanceToNextToken – *x0_i0_stringlib*
AIAssignDCR – *ginc_ai*
AIAttackInOrder – *ginc_ai*
AIAttackPreference – *ginc_ai*
AIContinueInterruptedScript – *ginc_ai*
AIDamageSwitchEndStatement – *ginc_ai*
AIFinitePursuit – *ginc_ai*
AIGetIsSpellQueued – *ginc_ai*
AIIgnoreCombat – *ginc_ai*
AIInterceptEventHandler – *ginc_ai*
AIMakeCounterCaster – *ginc_ai*
AIMakeProtector – *ginc_ai*
AIMakeTank – *ginc_ai*
AIResetType – *ginc_ai*
AIRestoreInterruptedScript – *ginc_ai*
AISpellQueueEnqueue – *ginc_ai*
AITurnOffDamageSwitch – *ginc_ai*
AITurnOnDamageSwitch – *ginc_ai*
AmIAHumanoid – *nw_i0_spells*
AngryTalk – *ginc_wp*
AnimateWithOther – *ginc_wp*
AnimationNeedsWait – *ginc_wp*
AnimDebug – *x0_i0_anims*
AnimInitialization – *x0_i0_anims*
AppendGlobalList – *x0_i0_stringlib*
AppendSpellToName – *c2_inc_craft*
AppendToList – *x0_i0_stringlib*
AppendUniqueToList – *x0_i0_stringlib*
AppendUniqueToNVP – *ginc_param_const*
ApplyDeathToSpiritEater – *kinc_spirit_eater*
ApplyEncodedEffectToItem – *ginc_crafting*
ApplyEncodedEffectsToItem – *ginc_crafting*
ApplyFatigue – *nwn2_inc_metmag*
ApplyFriendlySongEffectsToArea – *nwn2_inc_metmag*
ApplyFriendlySongEffectsToParty – *nwn2_inc_metmag*
ApplyFriendlySongEffectsToTarget – *nwn2_inc_metmag*
ApplyHenchmanModifier – *ginc_companion*
ApplyHostileSongEffectsToArea – *nwn2_inc_metmag*
ApplyMetamagicDurationMods – *nwn2_inc_metmag*
ApplyMetamagicDurationTypeMods – *nwn2_inc_metmag*
ApplyMetamagicVariableMods – *nwn2_inc_metmag*
ApplySEFToLocation – *ginc_effect*
ApplySEFToObject – *ginc_effect*
ApplySEFToObjectByTag – *ginc_effect*
ApplySEFToWP – *ginc_effect*
ApplySongDurationFeatMods – *nwn2_inc_metmag*
ApplySpiritEaterFeatList – *kinc_spirit_eater*
ApplySpiritEaterStage – *kinc_spirit_eater*

Function Index.

Function Index.

Function Index.

Function Index.

Function Index.

Function Index.

GetBaddiesVar – *ginc_baddie_stream*
GetBeenHired – *nw_i0_henchman*
GetBehaviorState – *x0_i0_behavior*
GetBehind – *nw_i0_assoc*
GetBehindLocation – *x0_i0_position*
GetBestAOEBehavior – *x2_i0_spells*
GetBestEldritchAOETarget – *nwn2_inc_talent*
GetBestEldritchBlastFeat – *nwn2_inc_talent*
GetBestEldritchShape – *nwn2_inc_talent*
GetBestInvocationEssenceMetamagic – *nwn2_inc_talent*
GetBluntWeapon – *x2_i0_spells*
GetBMAFormationStartPosition – *ginc_group*
GetBMALocation – *ginc_group*
GetBooleanValue – *x0_i0_common*
GetCached2DAIndex – *ginc_2da*
GetCached2DAEntry – *ginc_2da*
GetCacheVarName – *ginc_2da*
GetCampaignBooleanValue – *x0_i0_common*
GetCampaignDBInt – *x0_i0_campaign*
GetCampaignDBFloat – *x0_i0_campaign*
GetCampaignDBLocation – *x0_i0_campaign*
GetCampaignDBName – *x0_i0_campaign*
GetCampaignDBString – *x0_i0_campaign*
GetCampaignDBVector – *x0_i0_campaign*
GetCanBardSing – *nwn2_inc_metmag*
GetCanCastHealingSpells – *nw_i0_plot*
GetCanLevelUp – *nw_i0_henchman, x0_i0_henchman*
GetCappedCasterLevel – *x0_i0_spells*
GetCasterAbilityModifier – *x0_i0_spells*
GetChangedPosition – *x0_i0_position*
GetChangeInX – *x0_i0_position*
GetChangeInY – *x0_i0_position*
GetChapter – *nw_i0_henchman, x0_i0_common*
GetCharacterLevel – *nw_i0_generic*
GetCharisma – *nw_i0_plot*
GetClassSpellLevelColumn – *c2_inc_craft*
GetClockDisplay – *ginc_time*
GetClockDisplayDate – *ginc_time*
GetClockDisplayTime – *ginc_time*
GetCombatCondition – *x0_i0_combat*
GetCombatDifficulty – *x0_inc_generic*
GetCrafting2DARecipeMatch – *ginc_crafting*
GetCraftingIntData – *ginc_crafting*
GetCraftingStringData – *ginc_crafting*
GetCreatureFlag – *x2_inc_switches*
GetCreatureHomeWaypoint – *x0_i0_anims*
GetCreatureOverrideAIScriptTarget – *x2_inc_switches*
GetCreatureTalent – *x0_inc_generic*
GetCreatureTalentBestStd – *x0_inc_generic*
GetCreatureTalentRandomStd – *x0_inc_generic*
GetCRMax – *x0_inc_generic*
GetCureDamageTotal – *nw_i0_spells*
GetCurrentCTimeDate – *ginc_time*
GetCurrentFriend – *x0_i0_anims*
GetCurrentInteractionTarget – *x0_i0_anims*
GetCurrentMaster – *x0_i0_assoc*
GetCurrentNegativeConditions – *x0_i0_talent*
GetCurrentTimeHash – *ginc_autosave*
GetCurrentWaypoint – *x0_i0_walkway*
GetCustomLeftDirection – *x0_i0_position*
GetCustomRightDirection – *x0_i0_position*
GetDayNightSwitch – *x0_i0_walkway*
GetDelayedSpellInfoSaveDC – *nwn2_inc_metmag*
GetDidDie – *nw_i0_henchman, x0_i0_henchman*
GetDidPersuade – *x0_i0_common*
GetDidQuit – *x0_i0_henchman*
GetDistillationRecipeVar – *ginc_crafting*
GetDoneFlag – *ginc_vars*
GetDoorFlag – *x2_inc_switches*
GetDurationType – *ginc_param_const*
GetEldritchBlastDmg – *nw_i0_invocatns*
GetEldritchBlastLevel – *nw_i0_invocatns*
GetEncodedEffectItemProperty – *ginc_crafting*
GetEpicSpellSaveDC – *x2_i0_spells*
GetEventPostfix – *x2_inc_switches*
GetEventRouter – *x0_i0_common*
GetEventRouterTag – *x0_i0_common*
GetEventsClearedFlag – *ginc_event_handlers*
GetFarLeftDirection – *x0_i0_position*
GetFarRightDirection – *x0_i0_position*
GetFeatImmunityType – *x0_inc_generic*
GetFirstAlpha – *x0_i0_stringlib*
GetFirstBeamTarget – *ginc_reflection*
GetFirstInGroup – *ginc_group*
GetFirstPCInFaction – *ginc_debug*
GetFirstTalentTarget – *x0_i0_talent*
GetFirstValidInGroupFromCurrent – *ginc_group*
GetFlankingLeftLocation – *x0_i0_position*
GetFlankingRightLocation – *x0_i0_position*
GetFleeToExit – *nw_i0_generic*
GetFloatParam – *ginc_param_const*
GetFollowDistance – *x0_i0_assoc*
GetFormerMaster – *nw_i0_henchman*
GetForwardFlankingLeftLocation – *x0_i0_position*

Function Index.

Function Index.

Function Index.

GetObjectTypes – *ginc_param_const*
GetOneLiner – *x0_i0_common*
GetOppositeDirection – *x0_i0_position*
GetOppositeLocation – *x0_i0_position*
GetPartyGroup – *ginc_group*
GetPartyMemberHasEquipped – *ginc_item*
GetPCAverageXP – *ginc_misc*
GetPCByUniqueID – *kinc_spirit_eater*
GetPCLeader – *ginc_companion*
GetPCTotalLevel – *x2_inc_plot*
GetPercentageHPLoss – *x0_i0_assoc*
GetPersuadeAttempt – *x0_i0_common*
GetPlayerHasHired – *x0_i0_henchman*
GetPlayerHasHiredInCampaign – *x0_i0_henchman*
GetPlayerQueuedTarget – *ginc_companion*
GetPLocalInt – *nw_i0_plot*
GetPlotItemTag – *x0_i0_plotgiver*
GetPointsInStage – *kinc_spirit_eater*
GetPreferredSpiritEater – *kinc_spirit_eater*
GetPreviousWaypoint – *x0_i0_walkway*
GetProjectileTrapOrigin – *x0_i0_projtrap*
GetQuestItemTag – *x0_i0_plotgiver*
GetQuestStatus – *x0_i0_plotgiver*
GetQuestTag – *x0_i0_plotgiver*
GetQuestVarname – *x0_i0_plotgiver*
GetQuestXPPercentRewarded – *ginc_journal*
GetRacialTypeCount – *x0_i0_enemy*
GetRandom2DVector – *ginc_math*
GetRandomDelay – *nw_i0_spells*
GetRandomFriend – *x0_i0_anims*
GetRandomHench – *x2_inc_banter*
GetRandomInvocationEssenceByLevel – *nwn2_inc_talent*
GetRandomLocation – *x0_i0_position*
GetRandomObjectByTag – *x0_i0_anims*
GetRandomObjectByType – *x0_i0_anims*
GetRandomObjectInArea – *ginc_object*
GetRandomQuasiFriend – *x0_i0_anims*
GetRandomStop – *x0_i0_anims*
GetRandomTextNumber – *x2_inc_banter*
GetRandomWaypoint – *ginc_wp*
GetRatioMissingOfStage – *kinc_spirit_eater*
GetRatioRemainingOfStage – *kinc_spirit_eater*
GetRayTargets – *x2_inc_beholder*
GetReady – *x2_am_inc*
GetRecipeIntElement – *ginc_crafting*
GetRecipeMatch – *ginc_crafting*
GetRecipeVar – *ginc_crafting*
GetRectangleLocation – *ginc_group*
GetRemovalSpell – *x0_i0_match*
GetRespawnLocation – *x0_i0_common*
GetRestMessageStrRef – *ginc_restsys*
GetResurrected – *x0_i0_henchman*
GetRewardItemTag – *x0_i0_plotgiver*
GetRightDirection – *x0_i0_position*
GetRightHandWeapon – *x2_inc_ws_smith*
GetRightLocation – *x0_i0_position*
GetRowIndexes – *ginc_crafting*
GetScaledCastingMagic – *nw_i0_generic*
GetScaledDuration – *nw_i0_spells*
GetScaledEffect – *nw_i0_spells*
GetSecretItemRevealed – *x0_i0_secret*
GetShift – *x1_inc_cutscene, x2_inc_cutscene*
GetSizeModifier – *x0_i0_spells*
GetSkillConstant – *ginc_param_const*
GetSlashingWeapon – *x2_i0_spells*
GetSlotOfEquippedItem – *ginc_item*
GetSortedItemList – *ginc_crafting*
GetSoundObjectByTag – *ginc_param_const*
GetSpawnInCondition – *x0_i0_spawncond*
GetSpecialAreaForDeckCard – *x0_i0_deckmany*
GetSpellBreachProtection – *nw_i0_spells*
GetSpellEffectDelay – *nw_i0_spells*
GetSpellImmunityType – *x0_inc_generic*
GetSpellLevelForClass – *c2_inc_craft*
GetSpiritBarScreenName – *kinc_spirit_eater*
GetSpiritBarXMLFileName – *kinc_spirit_eater*
GetSpiritEater – *kinc_spirit_eater*
GetSpiritEaterCorruption – *kinc_spirit_eater*
GetSpiritEaterPoints – *kinc_spirit_eater*
GetSpiritEaterStage – *kinc_spirit_eater*
GetSpiritEaterStageMaximumPoints – *kinc_spirit_eater*
GetSpiritEaterStageMinimumPoints – *kinc_spirit_eater*
GetStandardFaction – *ginc_param_const*
GetState – *ginc_math*
GetStepLeftLocation – *x0_i0_position*
GetStepRightLocation – *x0_i0_position*
GetStoryVar – *nw_i0_henchman*
GetStringPad – *x0_i0_stringlib*
GetStringParam – *ginc_param_const*
GetStringPrefix – *ginc_vars*
GetStringSuffix – *ginc_vars*
GetStringTokenizer -- *x0_i0_stringlib*
GetSymbolUniqueID – *ginc_symbol_spells*
GetTagNoPrefix – *x0_i0_common*

Function Index.

Function Index.

Function Index.

Function Index.

IWPlayRandomHitQuote – *x2_int_intweapon*
IWPlayRandomUnequipComment – *x2_int_intweapon*
IWSpawnInWeaponCreature – *x2_int_intweapon*
IWSetConversationCondition – *x2_int_intweapon*
IWSetCreatureHadOneLiner – *x2_int_intweapon*
IWSetEnhancementAndDrainLevel – *x2_int_intweapon*
IWSetQuestionAsked – *x2_int_intweapon*
IWSetTalkedTo – *x2_int_intweapon*
IWStartIntelligentWeaponConversation – *x2_int_intweapon*
IWSWrapper – *x2_int_intweapon*

JobBarPatron – *x2_am_inc*
JumpPartyToSpeaker – *ginc_cutscene*

KeepDead – *x0_i0_henchman*
KillAndExplode – *x0_i0_corpses*
KillAndReplaceLootable – *x0_i0_corpses*
KillAndReplaceDecay – *x0_i0_corpses*
KillAndReplaceRaiseable – *x0_i0_corpses*
KillAndReplaceSelectable – *x0_i0_corpses*
KillAndReplaceDecorative – *x0_i0_corpses*
KnockOutCreature – *ginc_death*

LawChaosAxisAdjustment – *ginc_alignment*
LevelHenchmanUpTo – *x0_i0_henchman*
LevelUpXP1Henchman – *x0_i0_henchman*
ListenToTalker – *ginc_wp*
ListMembersOfGroup – *ginc_group*
LoadActionModes – *ginc_cutscene*
LoadPartyActionModes – *ginc_cutscene*
LoadPartyAIState – *ginc_cutscene*
LocationToString – *x0_i0_position*
LookInCrate – *ginc_wp*
LookUpWalkWayPoints – *x0_i0_walkway*
LookUpWalkWayPointsSet – *x0_i0_walkway*
LootInventory – *x0_i0_corpses*
LootInventorySlots – *x0_i0_corpses*

MakeConversable – *ginc_cutscene*
MarkAsDone – *ginc_vars*
MarkAsUndone – *ginc_vars*
MatchAreaOfEffectSpell – *x0_i0_match*
MakeBaseItemList – *ginc_crafting*
MatchCombatProtections – *x0_i0_match*
MatchCrossbow – *x0_i0_match*
MatchDoIHaveAMindAffectingSpellOnMe – *x0_i0_match*
MatchDoubleHandedWeapon – *x0_i0_match*
MatchElementalProtections – *x0_i0_match*
MakeEncodedEffect – *ginc_crafting*
MatchHumanoidRacialType – *x0_i0_match*
MatchInflictTouchAttack – *x0_i0_match*
MakeItemUseableByClass – *c2_inc_craft*
MakeItemUseableByClassesWithSpellAccess – *c2_inc_craft*
MakeList – *ginc_crafting*
MatchMeleeWeapon – *x0_i0_match*
MatchMindAffectingSpells – *x0_i0_match*
MatchNonliving – *x0_i0_match*
MakeNonNegIntList – *ginc_crafting*
MatchPersonSpells – *x0_i0_match*
MatchNormalBow – *x0_i0_match*
MakeRepeatedItemList – *ginc_crafting*
MarkItemAsDone – *ginc_item_script*
MatchShield – *x0_i0_match*
MatchSingleHandedWeapon – *x0_i0_match*
MatchSpellProtections – *x0_i0_match*
MaxAB – *ginc_baddie_stream*
MaximizeOrEmpower – *x0_i0_spells*
MeanCheck – *x0_i0_common*
Message – *ginc_debug*
MessageLine – *ginc_debug*
ModifyGlobalInt – *ginc_vars*
ModifyLocalInt – *ginc_vars*
ModifyLocalIntOnFaction – *ginc_vars*
MoveAllAssociatesTo – *x2_inc_plot*
MoveToAndAttack – *ginc_behavior*
MoveToNewLocation – *x0_i0_position*
MoveToNextWaypoint – *x0_i0_walkway*
MutualGreeting – *ginc_wp*
MyGetCreatureTalent – *nwn2_inc_talent*
MyPrintString – *x0_i0_debug*
MyResistSpell – *nw_i0_spells*
MySavingThrow – *nw_i0_spells*

N2_GetNPCEasyMark – *ginc_item*
N2_AppraiseOpenStore – *ginc_item*
NewCTimeDate – *ginc_time*
newdebug – *x0_i0_debug*
NoInterrupt – *ginc_actions*
NoPlayerInArea – *x2_am_inc*
NormalizeCTimeDate – *ginc_time*
NotifyRegisteredObject – *ginc_time*
NotifyRegisteredObjects – *ginc_time*

oidCreateItemOnObject – *ginc_item*

Function Index.

Function Index.

Function Index.

Function Index.

Function Index.