Event Calls

A description of various eventCalls that can be used when building a custom encounter.

- **activateTarget**
- **clearCombatRoles**
- **clearFactionOverrides**
- **damageAI**
- **designerComment**
- **forceAIToFlee**
- **forceChargeOnAllAI**
- **maintainAICount**
- **makeAIAwareOfPlayer**
- **migrateAIFromExternalScript**
- **proceedToNextScript**
- **raiseEventFlagOnExternalScript**
- **removeAI**
- **setCombatRoles**
- **setFactionRelation**

- **setMusicState**
- **setNextScriptIndex**
- **setNextScriptIndexRandom**
- **spawnAI**
- **spawnArchvile**
- **spawnBuffPod**
- **spawnPossessedAI**
- **spawnSingleAI**
- **spawnSpirit**
- **staggeredAISpawn**
- **stopMaintainingAICount**
- **wait**
- **waitAIHealthLevel**
- **waitAIRemaining**
- **waitForEventFlag**
- **waitKillCount**
- **waitMaintainComplete**
- **waitMulitpleConditions**
- **waitRandomKillCount**
- **waitStaggeredSpawnComplete**

activateTarget

> An eventCall to activate another entity from an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "activateTarget";
		args = {
			num = 2;
			item[0] = {
				entity = ""; // target
			}
			item[1] = {
				string = ""; // designComment
			}
		}
	}
}
```

- `target` is the entity that is being activated.
- `designComment` is an optional parameter for the user, and does not show up in game.

Example Usage

```
entity { // You Can't Just Shoot a Hole into the Surface of Mars
	entityDef mod_soundentity_samuel_vo_shoot_mars {
	inherit = "sound/soundentity";
	class = "idSoundEntity";
	expandInheritance = false;
	poolCount = 0;
	poolGranularity = 2;
	networkReplicated = false;
	disableAIPooling = false;
```

```
□edit = {
□□temporarySoundEvent = true;
□□spawnPosition = {
□□□x = 4.20000553;
□□□y = 0.449982464;
□□□z = 12.1500015;
□□}
□□soundOcclusionBypass = true;
□□startEvents = {
□□□num = 1;
□□□item[0] = "play_vo_samuel_hubsamuel_hookup_you_cant_just_shoot__ghost52033";
□□}
□}
}
}
```

```
item[0] = {
□eventCall = {
□□eventDef = "waitAIRemaining";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY";
□□□}
□□□item[1] = {
□□□□int = 0; // desired_remaing_ai_count
□□□}
□□□item[2] = {
□□□□string = ""; // group_label
□□□}
□□}
□}
}
item[1] = {
□eventCall = {
□□eventDef = "activateTarget";
□□args = {
□□□num = 2;
□□□item[0] = {
```

```
□□□□entity = "mod_soundentity_samuel_vo_shoot_mars"; // target
□□□}
□□□item[1] = {
□□□□string = "Play Hayden's audio for the Mars Hole line"; // designComment
□□□}
□□}
□}
}
```

First we place an `idSoundEntity` with the name `mod_soundentity_samuel_vo_shoot_mars` in the .entities, and then within the encounter manager, we can use `activateTarget` on `mod_soundentity_samuel_vo_shoot_mars` after our **`waitAIRemaining`**, so that the "You can't just shoot a hole in the surface of Mars" line is triggered once no ai remain.

clearCombatRoles

> An eventCall to clear any combat roles set by a preceding **`setCombatRoles`** .

Usage

```
item[0] = {
	eventCall = {
		eventDef = "clearCombatRoles";
		args = {
			num = 2;
			item[0] = {
				eEncounterSpawnType_t = ""; // spawnType
			}
			item[1] = {
				string = ""; // group_label
			}
		}
	}
}
```

- `spawnType` are the **`eEncounterSpawnType_t`** whose role should be cleared.
- `group_label` is the label for the ai whose role should be cleared. This parameter is optional, and can be left blank.

Example Usage

```
item[0] = {
	eventCall = {
		eventDef = "spawnSingleAI";
		args = {
			num = 3;
			item[0] = {
				eEncounterSpawnType_t = "ENCOUNTER_SPAWN_MANCUBUS";
			}
```

```
    item[1] = {
        entity = "ai_target_spawn_454";
    }
    item[2] = {
        string = "defender";
    }
  }
 }
}
item[1] = {
 eventCall = {
  eventDef = "setCombatRoles";
  args = {
   num = 3;
   item[0] = {
    eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY"; // spawnType
   }
   item[1] = {
    string = "defender"; // group_label
   }
   item[2] = {
    encounterGroupRole_t = "ROLE_DEFEND"; // group_role
   }
  }
 }
}
item[2] = {
 eventCall = {
  eventDef = "waitAIHealthLevel";
  args = {
   num = 3;
   item[0] = {
    eEncounterSpawnType_t = "ENCOUNTER_SPAWN_MANCUBUS";
   }
   item[1] = {
    float = 0.5; // target_group_health
   }
   item[2] = {
    string = ""; // group_label
   }
```

```
□□}
□}
}
item[3] = {
□eventCall = {
□□eventDef = "clearCombatRoles";
□□args = {
□□□num = 2;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_MANCUBUS"; // spawnType
□□□}
□□□item[1] = {
□□□□string = "defender"; // group_label
□□□}
□□}
□}
}
```

In this example, we spawn a Mancubus which is set to `ROLE_DEFEND`. Once it reaches 50% HP, we use `clearCombatRoles` to clear the defend role.

See Also

- **setCombatRoles**

clearFactionOverrides

> An eventCall to clear any faction overrides set by a preceding **setFactionRelation** in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "clearFactionOverrides";
		args = {
			num = 0;
		}
	}
}
```

Example Usage

```
item[0] = {
	eventCall = {
		eventDef = "setFactionRelation";
		args = {
			num = 4;
			item[0] = {
				eEncounterSpawnType_t = "ENCOUNTER_SPAWN_BARON"; // instigatorSpawnType
			}
			item[1] = {
				string = ""; // groupLabel
			}
			item[2] = {
				eEncounterSpawnType_t = "ENCOUNTER_SPAWN_CYBER_MANCUBUS"; // targetSpawnType
			}
			item[3] = {
				socialEmotion_t = "EMOTION_DESTROY_AT_ALL_COSTS"; // relation
			}
```

```
□□}
□}
}
item[1] = {
□eventCall = {
□□eventDef = "setFactionRelation";
□□args = {
□□□num = 4;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_CYBER_MANCUBUS"; // instigatorSpawnType
□□□}
□□□item[1] = {
□□□□string = ""; // groupLabel
□□□}
□□□item[2] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_BARON"; // targetSpawnType
□□□}
□□□item[3] = {
□□□□socialEmotion_t = "EMOTION_DESTROY_AT_ALL_COSTS"; // relation
□□□}
□□}
□}
}
item[2] = {
□eventCall = {
□□eventDef = "waitKillCount";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterEventFlags_t = "ENCOUNTER_SPAWN_ANY";
□□□}
□□□item[1] = {
□□□□int = 1;
□□□}
□□□item[2] = {
□□□□string = "";
□□□}
□□}
□}
}
```

```
item[3] = {
	eventCall = {
		eventDef = "clearFactionOverrides";
		args = {
			num = 0;
		}
	}
}
```

In this example **setFactionRelation** is used to create infighting between Barons and Cyber-Mancubi. Once the player has killed 1 of any ai type, `clearFactionOverrides` is used to restore normal behaviour, having them attack the player instead.

See Also

- **setFactionRelation**

damageAI

> An eventCall to apply a damage type to the specified ai in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "damageAI";
		args = {
			num = 3;
			item[0] = {
				decl = {
					damage = ""; // damageType
				}
			}
			item[1] = {
				eEncounterSpawnType_t = ""; // aiType
			}
			item[2] = {
				string = ""; // groupLabel
			}
		}
	}
}
```

- `damageType` is the path to the decl for the damage type to be used.
- `aiType` are the **`eEncounterSpawnType_t`** that you want the damage to be applied to.
- `group_label` is the label assigned to the ai that will be damaged by this eventCall. This parameter is optional, and can be left blank.

Example Usage

```
item[0] = {
	eventCall = {
		eventDef = "damageAI";
		args = {
			num = 3;
			item[0] = {
				decl = {
					damage = "damage/triggerhurt/triggerhurt1000_instagib"; // damageType
				}
			}
			item[1] = {
				eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY"; // aiType
			}
			item[2] = {
				string = "fodder"; // groupLabel
			}
		}
	}
}
```

In this example, `triggerhurt1000_instagib` is applied to any ai type with the `fodder` label, killing them instantly.

designerComment

> An eventCall for leaving a comment in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "designerComment";
		args = {
			num = 2;
			item[0] = {
				string = ""; // designerComment
			}
			item[1] = {
				bool = true; // printToConsole
			}
		}
	}
}
```

- `designerComment` is the comment you wish to add.
- `printToConsole` determines whether said comment is displayed in the console.

Example Usage

```
item[0] = {
	eventCall = {
		eventDef = "designerComment";
		args = {
			num = 2;
			item[0] = {
				string = "\nExample text between line breaks\n"; // designerComment
			}
			item[1] = {
```

```
□□□□bool = true; // printToConsole
□□□}
□□}
□}
}
```

forceAIToFlee

> An eventCall to that forces spawned ai in an idEncounterManager to flee.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "forceAIToFlee";
		args = {
			num = 4;
			item[0] = {
				eEncounterSpawnType_t = ""; // spawnType
			}
			item[1] = {
				string = ""; // group_label
			}
			item[2] = {
				float = 0; // radius
			}
			item[3] = {
				bool = true; // immediateDespawn
			}
		}
	}
}
```

- `spawnType` are the **`eEncounterSpawnType_t`** that you want to flee.
- `group_label` is the label for the ai to flee
- `radius`
- `immediateDespawn` is if the ai will immediately despawn (with the despawn animation)

Example Usage

```
item[0] = {
□eventCall = {
□□eventDef = "forceAIToFlee";
□□args = {
□□□num = 4;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY"; // spawnType
□□□}
□□□item[1] = {
□□□□string = ""; // group_label
□□□}
□□□item[2] = {
□□□□float = 0; // radius
□□□}
□□□item[3] = {
□□□□bool = true; // immediateDespawn
□□□}
□□}
□}
}
```

forceChargeOnAllAI

> An eventCall to make all spawned ai in an idEncounterManager charge the player.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "forceChargeOnAllAI";
		args = {
			num = 0;
		}
	}
}
```

Example Usage

```
item[0] = {
	eventCall = {
		eventDef = "waitAIRemaining";
		args = {
			num = 3;
			item[0] = {
				eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY";
			}
			item[1] = {
				int = 0; // desired_remaing_ai_count
			}
			item[2] = {
				string = "priority"; // group_label
			}
		}
	}
}
item[1] = {
```

```
□eventCall = {
□□eventDef = "waitAIRemaining";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY";
□□□}
□□□item[1] = {
□□□□int = 3; // desired_remaing_ai_count
□□□}
□□□item[2] = {
□□□□string = "fodder"; // group_label
□□□}
□□}
□}
}
item[2] = {
□eventCall = {
□□eventDef = "forceChargeOnAllAI";
□□args = {
□□□num = 0;
□□}
□}
}
```

Here, we wait until all ai with the `priority` label are killed, then we wait until only 3 fodder remain, and we use `forceChargeOnAllAI`, so that the player does not have to go around hunting stray fodder to finish the encounter.

maintainAICount

> An eventCall to maintain a certain quantity of respawning idAI2 in an idEncounterManager.

Usage

```
item[0] = {
    eventCall = {
        eventDef = "maintainAICount";
        args = {
            num = 8;
            item[0] = {
                eEncounterSpawnType_t = ""; // spawnType
            }
            item[1] = {
                int = 5; // desired_count
            }
            item[2] = {
                int = 10; // max_spawn_count
            }
            item[3] = {
                float = 5; // min_spawn_delay
            }
            item[4] = {
                int = 1; // min_ai_for_respawn
            }
            item[5] = {
                entity = ""; // spawnGroup
            }
            item[6] = {
                string = ""; // group_label
            }
            item[7] = {
                float = 10; // max_spawn_delay
            }
        }
```

```
    }
}
```

- `spawnType` are the **`eEncounterSpawnType_t`** that you want to be spawned. Multiple types can be listed, and the spawned ai types will be randomly chosen from this list. `ENCOUNTER_SPAWN_ANY` can also be used.
- `desired_count` is how many ai this eventCall should spawn up to. To avoid overlapping spawns, `desired_count` should be <= the number of **`idTarget_Spawns`** listed under your `idTargetSpawnGroup`.
- `max_spawn_count` is the maximum number of ai that can be spawned by this eventCall in total. This can be set to `-1` for infinite respawns.
- `min_spawn_delay` is the minimum delay between each spawn, in seconds. This value also affects how soon before this eventCall spawns its first ai.
- `min_ai_for_respawn` triggers the ai respawns when they fall below or equal to the value it's set to. `min_ai_for_respawn` should also be < `desired_count`. If it is greater than `desired_count`, the quantity of demons maintained will not properly reflect `desired_count`, possibly exceeding it.
- `spawnGroup` is the `idTargetSpawnGroup` you want to use.
- `group_label` is what label you want to give to the ai spawned by this eventCall. This parameter is optional, and can be left blank.
- `max_spawn_delay` is the maximum delay between each spawn, in seconds.

Example Usage

```
item[0] = {
    eventCall = {
        eventDef = "maintainAICount";
        args = {
            num = 8;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ZOMBIE"; // spawnType
            }
            item[1] = {
                int = 10; // desired_count
            }
```

```
            item[2] = {
                int = 20; // max_spawn_count
            }
            item[3] = {
                float = 3; // min_spawn_delay
            }
            item[4] = {
                int = 5; // min_ai_for_respawn
            }
            item[5] = {
                entity = "ai_target_spawn_volume_76"; // spawnGroup
            }
            item[6] = {
                string = ""; // group_label
            }
            item[7] = {
                float = 5; // max_spawn_delay
            }
        }
    }
}
```

In this example, 10 Zombies will be spawned, with a 3-5s delay between each (re)spawn. When only 5 Zombies remain, they will begin respawning until there are 10 Zombies again. Once 20 Zombies have been spawned in total, this eventCall is complete.

Notes

This eventcall appears to have functionality to adjust its kill requirements if not enough enemies are available, to prevent the encounter from locking up, due to otherwise unattainable kill requirements.

If the quantity of currently alive required enemies are greater or equal to the required kills, then the eventcall functions as normal. However, if the quantity of currently alive required enemies are less than the required kills, then the amount of enemies you need to kill is lowered to match the quantity of currently alive required enemies.

This anti-softlock functionality fails if a `maintainAICount` is followed by a `waitKillCount` sometime later in the same encounter manager, while the `maintainAICount` is still active. If there are not enough enemies are available in this scenario, then the `waitKillCount` will use the completion of the `maintainAICount` as one of it's requirements. Simply killing the maintained demons until its `max_spawn_count` is depleted will avoid soft locking in this situation. Do note that if `max_spawn_count` is set to `-1`, then the soft lock cannot be avoided.

This functionality has only been tested for in Doom Eternal.

See Also

- **waitMaintainComplete**
- **stopMaintainingAICount**

makeAIAwareOfPlayer

> An eventCall to make any ai entities spawned in an idEncounterManager, aware of the player.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "makeAIAwareOfPlayer";
		args = {
			num = 4;
			item[0] = {
				bool = true; // allActive
			}
			item[1] = {
				bool = false; // onSpawn
			}
			item[2] = {
				string = ""; // groupLabel
			}
			item[3] = {
				bool = false; // restorePerception
			}
		}
	}
}
```

migrateAIFromExternalScript

> An eventCall to migrate ai entities from another idEncounterManager, into the current encounter manager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "migrateAIFromExternalScript";
		args = {
			num = 4;
			item[0] = {
				entity = ""; // encounterScript
			}
			item[1] = {
				eEncounterSpawnType_t = ""; // aiType
			}
			item[2] = {
				string = ""; // groupLabel
			}
			item[3] = {
				bool = false; // sharedBetweenScripts
			}
		}
	}
}
```

- `encounterScript` is the name of the external idEncounterManager to migrate ai from.
- `aiType` are the ai types from the external encounter manager to be migrated.
- `groupLabel` is the label given to the migrated ai.
- `sharedBetweenScripts` is if the migrated ai are shared between both encounter managers.

proceedToNextScript

> An eventCall to proceed to the next script in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "proceedToNextScript";
		args = {
			num = 2;
			item[0] = {
				bool = true; // bypassNextWaitForCommit
			}
			item[1] = {
				bool = true; // carryOverExistingUserFlags
			}
		}
	}
}
```

This eventCall should always be paired with **setNextScriptIndex**.

See Also

- **setNextScriptIndex**

raiseEventFlagOnExternalScript

> An eventcall to trigger a flag on an idEncounterManager outside of the one that this eventcall is used in.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "raiseEventFlagOnExternalScript";
		args = {
			num = 3;
			item[0] = {
				entity = "game_ai_encounter_script_encounter_script_pre_combat_imp_room"; // encounterScrip
			}
			item[1] = {
				eEncounterEventFlags_t = "ENCOUNTER_EVENT_FLAG_USER_EVENT"; // eventFlag
			}
			item[2] = {
				string = "blue_door_start"; // userFlag
			}
		}
	}
}
```

- `encounterScript` is the encounter manager that the user flag is for
- `eventFlag` is the `eEncounterEventFlags_t` (type) for the flag
- `userFlag` is the parameter of the name in the `idEncounterTrigger_RaiseUserFlag` being triggered

Note that using **`activateTarget`** directly on the `idEncounterTrigger_RaiseUserFlag` in question can also accomplish the same effect.

removeAI

> An eventCall to remove spawned ai entities from an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "removeAI";
		args = {
			num = 2;
			item[0] = {
				eEncounterSpawnType_t = ""; // spawnType
			}
			item[1] = {
				string = ""; // group_label
			}
		}
	}
}
```

- `spawnType` are the **`eEncounterSpawnType_t`** that you want to be removed.
- `group_label` is what label for the ai to be removed by this eventCall. This parameter is optional, and can be left blank.

AI entities removed with this command will be instantly removed, without any death or despawn animation. It is best used for ai types that are not affected by **`damageAI`** or **`forceAIToFlee`**.

setCombatRoles

> An eventCall to assign roles to idAI2 in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "setCombatRoles";
		args = {
			num = 3;
			item[0] = {
				eEncounterSpawnType_t = ""; // spawnType
			}
			item[1] = {
				string = ""; // group_label
			}
			item[2] = {
				encounterGroupRole_t = ""; // group_role
			}
		}
	}
}
```

- `spawnType` are the **`eEncounterSpawnType_t`** that you want to given a role.
- `group_label` is the label for the ai be given a role by this eventCall. This parameter is optional, and can be left blank.
- `group_role` is the **`encounterGroupRole_t`** (combat role) that you wish to give to the specified ai.

Example Usage

```
item[0] = {
	eventCall = {
```

```
□□eventDef = "spawnSingleAI";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_MANCUBUS";
□□□}
□□□item[1] = {
□□□□entity = "ai_target_spawn_454";
□□□}
□□□item[2] = {
□□□□string = "defender";
□□□}
□□}
□}
}
item[1] = {
□eventCall = {
□□eventDef = "setCombatRoles";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY"; // spawnType
□□□}
□□□item[1] = {
□□□□string = "defender"; // group_label
□□□}
□□□item[2] = {
□□□□encounterGroupRole_t = "ROLE_DEFEND"; // group_role
□□□}
□□}
□}
}
```

In this example, a Mancubus is spawned with the label `defender`, and then `setCombatRoles` is used to set any ai type with the `defender` label to `ROLE_DEFEND`.

See Also

- **clearCombatRoles**

setFactionRelation

> An eventCall to force an emotion between an instigating and target faction in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "setFactionRelation";
		args = {
			num = 4;
			item[0] = {
				eEncounterSpawnType_t = ""; // instigatorSpawnType
			}
			item[1] = {
				string = ""; // groupLabel
			}
			item[2] = {
				eEncounterSpawnType_t = ""; // targetSpawnType
			}
			item[3] = {
				socialEmotion_t = ""; // relation
			}
		}
	}
}
```

- `instigatorSpawnType` is the ai type for the instigator.
- `targetSpawnType` is the ai type for the target.
- `relation` is the **`socialEmotion_t`** (emotion) that the instigator feels towards the target.

> The set emotion is only for the instigator towards the target. The target does not automatically reciprocate the same set emotion.

Example Usage

```
item[0] = {
□eventCall = {
□□eventDef = "setFactionRelation";
□□args = {
□□□num = 4;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_BARON"; // instigatorSpawnType
□□□}
□□□item[1] = {
□□□□string = ""; // groupLabel
□□□}
□□□item[2] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_IMP ENCOUNTER_SPAWN_HELL_SOLDIER"; //
targetSpawnType
□□□}
□□□item[3] = {
□□□□socialEmotion_t = "EMOTION_DESTROY_AT_ALL_COSTS"; // relation
□□□}
□□}
□}
}
item[1] = {
□eventCall = {
□□eventDef = "setFactionRelation";
□□args = {
□□□num = 4;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_IMP ENCOUNTER_SPAWN_HELL_SOLDIER"; //
instigatorSpawnType
□□□}
□□□item[1] = {
□□□□string = ""; // groupLabel
□□□}
□□□item[2] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_BARON"; // targetSpawnType
□□□}
□□□item[3] = {
```

```
□□□□socialEmotion_t = "EMOTION_DESTROY_AT_ALL_COSTS"; // relation
□□□}
□□}
□}
}
```

In this example, Barons are set to `EMOTION_DESTROY_AT_ALL_COSTS` against Imps/Soldiers, and vice versa. This results in forced infighting between Barons and Imps/Soldiers.

See Also

- **clearFactionOverrides**

setMusicState

> An eventCall to change the state of the specified **idMusicEntity** in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "setMusicState";
		args = {
			num = 3;
			item[0] = {
				entity = ""; // target
			}
			item[1] = {
				decl = {
					soundstate = ""; // stateDecl
				}
			}
			item[2] = {
				string = ""; // designComment
			}
		}
	}
}
```

- `target` is the **idMusicEntity** to change the state of.
- `stateDecl` is the path to the decl for the `idSoundState` you want to change it to.

Example Usage

```
item[0] = {
	eventCall = {
		eventDef = "setMusicState";
```

```
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□entity = "sound_sound_musicentity_1"; // target
□□□}
□□□item[1] = {
□□□□decl = {
□□□□□soundstate = "music_ghost_states/main_heavy"; // stateDecl
□□□□}
□□□}
□□□item[2] = {
□□□□string = "Change music state to heavy combat music"; // designComment
□□□}
□□}
□}
}
```

See Also

- **idMusicEntity**

setNextScriptIndex

> An eventCall to set the index of the next script in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "setNextScriptIndex";
		args = {
			num = 1;
			item[0] = { // nextScriptIndex
				int = 1;
			}
		}
	}
}
```

- `nextScriptIndex` is the index of next script

Example Usage

```
entity { // Custom Encounter #1
	entityDef mod_encounter_manager_custom_1 {
	inherit = "encounter/manager";
	class = "idEncounterManager";
	expandInheritance = false;
	poolCount = 0;
	poolGranularity = 2;
	networkReplicated = true;
	disableAIPooling = false;
	edit = {
		enableAIHighlightOnFinish = true;
		disabledAITypeForHighlight = "AI_MONSTER_SPECTRE AI_MONSTER_BUFF_POD AI_MONSTER_TENTACLE";
		playerMetricDef = "encounter/player_metrics";
```

```
□□chargeCombatGrouping = "encounter/combat_role/charge_command";
□□aiTypeDefAssignments = "actorpopulation/default/default_no_bosses";
□□spawnPosition = {
□□□x = 0;
□□□y = 0;
□□□z = 0;
□□}
□□soundOcclusionBypass = true;
□□combatRatingScale = "COMBAT_RATING_SCALE_IGNORE";
□□encounterComponent = {
□□□entityEvents = {
□□□□num = 2;
□□□□item[0] = { // Part 0
□□□□□entity = "mod_encounter_manager_custom_1";
□□□□□events = {
□□□□□□num = 5;
□□□□□□item[0] = {
□□□□□□□eventCall = {
□□□□□□□□eventDef = "setMusicState";
□□□□□□□□args = {
□□□□□□□□□num = 3;
□□□□□□□□□item[0] = {
□□□□□□□□□□entity = "sound_sound_musicentity_1";
□□□□□□□□□}
□□□□□□□□□item[1] = {
□□□□□□□□□□decl = {
□□□□□□□□□□soundstate = "music_ghost_states/main_heavy";
□□□□□□□□□□}
□□□□□□□□□}
□□□□□□□□□item[2] = {
□□□□□□□□□□string = "";
□□□□□□□□□}
□□□□□□□□}
□□□□□□□}
□□□□□□item[1] = {
□□□□□□□eventCall = {
□□□□□□□□eventDef = "spawnSingleAI";
□□□□□□□□args = {
□□□□□□□□□num = 3;
```

```
□□□□□□□□□□item[0] = {
□□□□□□□□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_BARON";
□□□□□□□□□□}
□□□□□□□□□□item[1] = {
□□□□□□□□□□□entity = "ai_target_spawn_87";
□□□□□□□□□□}
□□□□□□□□□□item[2] = {
□□□□□□□□□□□string = "";
□□□□□□□□□□}
□□□□□□□□□}
□□□□□□□□}
□□□□□□□}
□□□□□□□item[2] = {
□□□□□□□□eventCall = {
□□□□□□□□□eventDef = "waitAIRemaining";
□□□□□□□□□args = {
□□□□□□□□□□num = 3;
□□□□□□□□□□item[0] = {
□□□□□□□□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY";
□□□□□□□□□□}
□□□□□□□□□□item[1] = {
□□□□□□□□□□□int = 0; // desired_remaing_ai_count
□□□□□□□□□□}
□□□□□□□□□□item[2] = {
□□□□□□□□□□□string = ""; // group_label
□□□□□□□□□□}
□□□□□□□□□}
□□□□□□□□}
□□□□□□□}
□□□□□□□item[3] = {
□□□□□□□□eventCall = {
□□□□□□□□□eventDef = "setNextScriptIndex";
□□□□□□□□□args = {
□□□□□□□□□□num = 1;
□□□□□□□□□□item[0] = { // index of next script
□□□□□□□□□□□int = 1;
□□□□□□□□□□}
□□□□□□□□□}
□□□□□□□□}
□□□□□□□}
```

```
□□□□□□□item[4] = {
□□□□□□□□eventCall = {
□□□□□□□□□eventDef = "proceedToNextScript";
□□□□□□□□□args = {
□□□□□□□□□□num = 2;
□□□□□□□□□□item[0] = {
□□□□□□□□□□□bool = true; // bypassNextWaitForCommit
□□□□□□□□□□}
□□□□□□□□□□item[1] = {
□□□□□□□□□□□bool = true; // carryOverExistingUserFlags
□□□□□□□□□□}
□□□□□□□□□}
□□□□□□□□}
□□□□□□□}
□□□□□□}
□□□□□item[1] = { // Part 1
□□□□□□entity = "mod_encounter_manager_custom_1";
□□□□□□events = {
□□□□□□□num = 2;
□□□□□□□item[0] = {
□□□□□□□□eventCall = {
□□□□□□□□□eventDef = "setMusicState";
□□□□□□□□□args = {
□□□□□□□□□□num = 3;
□□□□□□□□□□item[0] = {
□□□□□□□□□□□entity = "sound_sound_musicentity_1";
□□□□□□□□□□}
□□□□□□□□□□item[1] = {
□□□□□□□□□□□decl = {
□□□□□□□□□□□□soundstate = "music_ghost_states/main_ambient";
□□□□□□□□□□□}
□□□□□□□□□□}
□□□□□□□□□□item[2] = {
□□□□□□□□□□□string = "";
□□□□□□□□□□}
□□□□□□□□□}
□□□□□□□□}
□□□□□□□}
□□□□□□item[1] = {
```

```
□□□□□□□□eventCall = {
□□□□□□□□□eventDef = "activateTarget";
□□□□□□□□□args = {
□□□□□□□□□□num = 2;
□□□□□□□□□□item[0] = {
□□□□□□□□□□□entity = "game_target_count_cp11"; // target
□□□□□□□□□□}
□□□□□□□□□□item[1] = {
□□□□□□□□□□□string = ""; // designComment
□□□□□□□□□□}
□□□□□□□□□}
□□□□□□□□}
□□□□□□□}
□□□□□□}
□□□□□}
□□□}
□□}
□}
}
}
```

In this example we use `setNextScriptIndex` after all there are no ai remaning, followed by **`proceedToNextScript`** to advance to the next script. Note that the encounter manager has to be split into multiple scripts (line 26 and 110) in order for `setNextScriptIndex` to work properly. The next script index does not have to be in sequential order either, as in an encounter manager with 4 scripts, it is possible to advance from script 0 to script 3, skipping over scripts 1 and 2.

See Also

- **proceedToNextScript**

setNextScriptIndexRandom

> An eventCall to set a random index of the next script in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "setNextScriptIndexRandom";
		args = {
			num = 2;
			item[0] = {
				int = 1; // from event 0
			}
			item[1] = {
				int = 5; // to event 5
			}
		}
	}
}
```

spawnAI

> An eventCall to spawn multiple idAI2 entities, simultaneously, in an idEncounterManager.

Usage

```
item[0] = {
    eventCall = {
        eventDef = "spawnAI";
        args = {
            num = 4;
            item[0] = {
                eEncounterSpawnType_t = ""; // spawnType
            }
            item[1] = {
                int = 6; // spawn_count
            }
            item[2] = {
                entity = ""; // spawnGroup
            }
            item[3] = {
                string = ""; // group_label
            }
        }
    }
}
```

- `spawnType` is the ai type that you want to be spawned. Multiple types can be listed, and the spawned ai types will be randomly chosen from this list.
- `spawnCount` is the number of ai you want to be spawned. To avoid overlapping spawns, `spawnCount` should be <= the number of **`idTarget_Spawn`** entities listed under your `idTargetSpawnGroup`.
- `group_label` is the label you want to give this spawned ai. This parameter is optional and can be left blank.

Example Usage

```
item[0] = {
	eventCall = {
		eventDef = "spawnAI";
		args = {
			num = 4;
			item[0] = {
				eEncounterSpawnType_t = "ENCOUNTER_SPAWN_HELL_SOLDIER"; // spawnType
			}
			item[1] = {
				int = 6; // spawn_count
			}
			item[2] = {
				entity = "mod_spawngroup_1"; // spawnGroup
			}
			item[3] = {
				string = "fodder"; // group_label
			}
		}
	}
}
```

spawnArchvile

> An eventCall to spawn an Archvile.

Usage

```
item[0] = {
    eventCall = {
        eventDef = "spawnArchvile";
        args = {
            num = 4;
            item[0] = {
                entity = ""; // spawnTarget
            }
            item[1] = {
                entity = ""; // archvileTemplate
            }
            item[2] = {
                string = ""; // archvile_label
            }
            item[3] = {
                string = ""; // group_label
            }
        }
    }
}
```

- `spawnTarget` is the spawn target or spawn group to spawn the Archvile with.
- `archvileTemplate` is the **idArchvileTemplate** you want to use.
- `archvile_label` is the label given to the Archvile that is spawned. This parameter is optional and can be left blank.
- `group_label` is the label given to demons summoned by the Archvile. This parameter is optional and can be left blank.

Note that if an Archvile is spawned without an idArchvileTemplate, it will default to being able summon all (base game?) ai types, with the exemption of another Archvile.

Example Usage

```
item[0] = {
    eventCall = {
        eventDef = "spawnArchvile";
        args = {
            num = 4;
            item[0] = {
                entity = "ai_target_spawn_111"; // spawnTarget
            }
            item[1] = {
                entity = "mod_template_2"; // archvileTemplate
            }
            item[2] = {
                string = "super heavy"; // archvile_label
            }
            item[3] = {
                string = "summons"; // group_label
            }
        }
    }
}
```

See Also

- **idArchvileTemplate**

spawnBuffPod

> An eventCall to spawn a buff totem.

Usage

```
item[0] = {
    eventCall = {
        eventDef = "spawnBuffPod";
        args = {
            num = 3;
            item[0] = {
                entity = ""; // spawnTarget
            }
            item[1] = {
                string = ""; // buffpod_label
            }
            item[2] = {
                string = ""; // buffed_ai_label
            }
        }
    }
}
```

- `spawnTarget` is the spawn target to spawn the totem at.
- `buffpod_label` is the label for the totem itself.
- `buffed_ai_label` is the label for the ai to buff. If this is left blank, all ai will be buffed.

Example Usage

```
item[0] = {
    eventCall = {
        eventDef = "spawnSingleAI";
        args = {
```

```
            num = 3;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_IMP"; // spawnType
            }
            item[1] = {
                entity = "ai_target_spawn_46"; // spawnTarget
            }
            item[2] = {
                string = ""; // group_label
            }
        }
    }
}
item[1] = {
    eventCall = {
        eventDef = "spawnSingleAI";
        args = {
            num = 3;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_IMP"; // spawnType
            }
            item[1] = {
                entity = "ai_target_spawn_47"; // spawnTarget
            }
            item[2] = {
                string = "buff me pls"; // group_label
            }
        }
    }
}
item[2] = {
    eventCall = {
        eventDef = "spawnBuffPod";
        args = {
            num = 3;
            item[0] = {
                entity = "ai_target_spawn_451"; // spawnTarget
            }
            item[1] = {
                string = ""; // buffpod_label
```

```
            }
            item[2] = {
                string = "buff me pls"; // buffed_ai_label
            }
        }
    }
}
```

In this example, the totem will only buff the Imp with a `group_label` of `"buff me pls"`, because that matches the `buffed_ai_label` the totem is looking for.

spawnPossessedAI

> An eventCall to spawn an ai that is already possessed by a spirit in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "spawnPossessedAI";
		args = {
			num = 7;
			item[0] = {
				eEncounterSpawnType_t = ""; // ai_spawnType
			}
			item[1] = {
				entity = ""; // ai_spawnTarget
			}
			item[2] = {
				string = ""; // ai_groupLabel
			}
			item[3] = {
				entity = ""; // spirit_spawnTarget
			}
			item[4] = {
				eEncounterSpawnType_t = ""; // spirit_allowedAITypes
			}
			item[5] = {
				string = ""; // spirit_allowedGroupLabel
			}
			item[6] = {
				bool = true; // aiTypeExplicitFiltering
			}
		}
	}
}
```

- `ai_spawnType` is the **`eEncounterSpawnType_t`** for the ai to be spawned.
- `ai_spawnTarget` is the spawn target to be used for the ai to be spawned.
- `ai_groupLabel` is the label to be given to the spawned ai.
- `spirit_spawnTarget` is the spawn target to be used for the spirit.
- `spirit_allowedAITypes` are the **`eEncounterSpawnType_t`** that the spirit is allowed to possess.
- `spirit_allowedGroupLabel` is the label that the spirit is allowed to possess.
- `aiTypeExplicitFiltering` is whether the spirit will possess ai based on the specified restrictions.

Example Usage

```
item[0] = {
	eventCall = {
		eventDef = "spawnSingleAI";
		args = {
			num = 3;
			item[0] = {
				eEncounterSpawnType_t = "ENCOUNTER_SPAWN_PINKY";
			}
			item[1] = {
				entity = "ai_target_spawn_13";
			}
			item[2] = {
				string = "heavy";
			}
		}
	}
}
item[1] = {
	eventCall = {
		eventDef = "spawnPossessedAI";
		args = {
			num = 7;
			item[0] = {
```

```
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_REVENANT"; // ai_spawnType
□□□}
□□□item[1] = {
□□□□entity = "ai_target_spawn_21"; // ai_spawnTarget
□□□}
□□□item[2] = {
□□□□string = "beamer"; // ai_groupLabel
□□□}
□□□item[3] = {
□□□□entity = "ai_target_spawn_17"; // spirit_spawnTarget
□□□}
□□□item[4] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY"; // spirit_allowedAITypes
□□□}
□□□item[5] = {
□□□□string = "beamer"; // spirit_allowedGroupLabel
□□□}
□□□item[6] = {
□□□□bool = true; // aiTypeExplicitFiltering
□□□}
□□}
□}
}
```

In this example, we spawn a Pinky with the `heavy` label. Then we use `spawnPossessedAI` to spawn a Revenant with the `beamer` label, and the spirit with the ability to possess any ai type with the `beamer` label. This way, once the possessed Revenant is dead, there are no other ai that use the `beamer` label, so the spirit will be unable to possess anyone else, and will despawn.

See Also

- **spawnSpirit**

spawnSingleAI

> An eventCall to spawn a single idAI2 entity in an idEncounterManager.

Usage

```
item[0] = {
    eventCall = {
        eventDef = "spawnSingleAI";
        args = {
            num = 3;
            item[0] = {
                eEncounterSpawnType_t = ""; // spawnType
            }
            item[1] = {
                entity = ""; // spawnTarget
            }
            item[2] = {
                string = ""; // group_label
            }
        }
    }
}
```

- `spawnType` is what ai type is spawned. Multiple types can be listed, but only a single ai entity will be spawned. Its type will be randomly chosen from this list.
- `spawnTarget` is the **`idTarget_Spawn`** entity you want to use.
- `group_label` is the label you want to give this spawned ai. This parameter is optional and can be left blank.

Example Usage

```
item[0] = {
    eventCall = {
```

```
        eventDef = "spawnSingleAI";
        args = {
            num = 3;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_IMP"; // spawnType
            }
            item[1] = {
                entity = "ai_target_spawn_47"; // spawnTarget
            }
            item[2] = {
                string = "fodder"; // group_label
            }
        }
    }
}
```

spawnSpirit

> An eventCall to spawn a spirit in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "spawnSpirit";
		args = {
			num = 4;
			item[0] = {
				entity = ""; // spawnTarget
			}
			item[1] = {
				eEncounterSpawnType_t = ""; // allowedAITypes
			}
			item[2] = {
				string = ""; // allowed_group_label
			}
			item[3] = {
				bool = true; // aiTypeExplicitFiltering
			}
		}
	}
}
```

- `spawnTarget` is the spawn target that the spirit is to be spawned with.
- `allowedAITypes` are the **`eEncounterSpawnType_t`** that the spirit is allowed to possess.
- `allowed_group_label` is the label that the spirit is allowed to possess.
- `aiTypeExplicitFiltering` is whether the spirit will only possess the exact specified ai types/group label.

Example Usage

```
item[0] = {
	eventCall = {
		eventDef = "spawnSingleAI";
		args = {
			num = 3;
			item[0] = {
				eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ARACHNOTRON";
			}
			item[1] = {
				entity = "ai_target_spawn_17";
			}
			item[2] = {
				string = "spider";
			}
		}
	}
}
item[1] = {
	eventCall = {
		eventDef = "spawnSingleAI";
		args = {
			num = 3;
			item[0] = {
				eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ARACHNOTRON";
			}
			item[1] = {
				entity = "ai_target_spawn_21";
			}
			item[2] = {
				string = "possess me";
			}
		}
	}
}
item[2] = {
	eventCall = {
		eventDef = "spawnSpirit";
```

```
□□args = {
□□□num = 4;
□□□item[0] = {
□□□□entity = "ai_target_spawn_26"; // spawnTarget
□□□}
□□□item[1] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ARACHNOTRON"; // allowedAITypes
□□□}
□□□item[2] = {
□□□□string = "possess me"; // allowed_group_label
□□□}
□□□item[3] = {
□□□□bool = true; // aiTypeExplicitFiltering
□□□}
□□}
□}
}
```

In this example, two Arachnotrons are spawned, one with a `spider` label, and the other with the `possess me` label. `spawnSpirit` is then used for `ENCOUNTER_SPAWN_ARACHNOTRON` with the possess me label, that way only the latter Arachnotron will be possessed.

See Also

- **spawnPossessedAI**

staggeredAISpawn

> An eventCall to spawn multiple idAI2 entities, with a delay between each spawn, in an idEncounterManager.

Usage

```
item[0] = {
    eventCall = {
        eventDef = "staggeredAISpawn";
        args = {
            num = 6;
            item[0] = {
                eEncounterSpawnType_t = ""; // spawnType
            }
            item[1] = {
                int = 5; // spawn_count
            }
            item[2] = {
                entity = ""; // spawnGroup
            }
            item[3] = {
                string = ""; // group_label
            }
            item[4] = {
                float = 1; // minSpawnStagger
            }
            item[5] = {
                float = 5; // maxSpawnStagger
            }
        }
    }
}
```

- `spawnType` is the ai type that you want to be spawned. Multiple types can be listed, and the spawned ai types will be randomly chosen from this list.

- `spawnCount` is the number of ai you want to be spawned. To avoid overlapping spawns, spawnCount should be <= the number of **`idTarget_Spawns`** listed under your `idTargetSpawnGroup`.
- `spawnGroup` is the `idTargetSpawnGroup` you want to use.
- `group_label` is what label you want to give to the ai spawned by this eventCall. This parameter is optional, and can be left blank.
- `minSpawnStagger` is the minimum delay between each spawn, in seconds. This value also affects how soon the first spawn in this eventCall occurs.
- `maxSpawnStagger` is the maximum delay between spawns, in seconds.

> `minSpawnStagger` must be <= `maxSpawnStagger`

Example Usage

```
item[0] = {
    eventCall = {
        eventDef = "staggeredAISpawn";
        args = {
            num = 6;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ZOMBIE_TIER_1"; // spawnType
            }
            item[1] = {
                int = 5; // spawn_count
            }
            item[2] = {
                entity = "game_target_spawn_group_13"; // spawnGroup
            }
            item[3] = {
                string = ""; // group_label
            }
            item[4] = {
                float = 7.5; // minSpawnStagger
            }
            item[5] = {
                float = 10; // maxSpawnStagger
```

```
            }
        }
    }
}
```

In this example, 5 Zombies will be spawned, with a 7.5 - 10s delay between each one.

See Also

- **waitStaggeredSpawnComplete**

stopMaintainingAICount

> An eventCall to end a **`maintainAICount`** in an idEncounterManager.

Usage

> This eventCall should only be used after a **`maintainAICount`**.

```
item[0] = {
    eventCall = {
        eventDef = "stopMaintainingAICount";
        args = {
            num = 2;
            item[0] = {
                eEncounterSpawnType_t = ""; // spawnType
            }
            item[1] = {
                string = ""; // group_label
            }
        }
    }
}
```

- `spawnType` is the **`eEncounterSpawnType_t`** used in the preceding **`maintainAICount`**. `ENCOUNTER_SPAWN_ANY` may also be used instead.
- `group_label` is the label used in the preceding **`maintainAICount`**. This parameter is optional, and may be left blank.

Example Usage

```
item[0] = {
    eventCall = {
        eventDef = "maintainAICount";
        args = {
```

```
            num = 8;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_IMP"; // spawnType
            }
            item[1] = {
                int = 5; // desired_count
            }
            item[2] = {
                int = -1; // max_spawn_count
            }
            item[3] = {
                float = 1; // min_spawn_delay
            }
            item[4] = {
                int = 3; // min_ai_for_respawn
            }
            item[5] = {
                entity = "mod_spawngroup_1"; // spawnGroup
            }
            item[6] = {
                string = "fodder"; // group_label
            }
            item[7] = {
                float = 2; // max_spawn_delay
            }
        }
    }
}
item[1] = {
    eventCall = {
        eventDef = "waitKillCount";
        args = {
            num = 3;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY";
            }
            item[1] = {
                int = 10;
            }
            item[2] = {
```

```
                string = "fodder";
            }
        }
    }
}
item[2] = {
    eventCall = {
        eventDef = "stopMaintainingAICount";
        args = {
            num = 2;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY";
            }
            item[1] = {
                string = "fodder";
            }
        }
    }
}
```

In this example, the **maintainAICount** will spawn 5 infinitely respawning Imps. However, once the player has killed 10 Imps, the **maintainAICount** will be stopped. Any Imps that are present when `stopMaintainingAICount` was called, will remain.

See Also

- **maintainAICount**
- **waitMaintainComplete**

wait

> An eventCall that tells the idEncounterManager to wait a specified amount of time before executing the next eventCall.

Usage

```
item[0] = {
    eventCall = {
        eventDef = "wait";
        args = {
            num = 1;
            item[0] = {
                float = 0.5; // seconds
            }
        }
    }
}
```

This eventCall can be used before and/or after most other eventCalls. The unit of time used is in seconds.

waitAIHealthLevel

> An eventCall that tells the idEncounterManager to wait until the specified ai type reaches a certain HP level, before executing the next eventCalls.

Usage

```
item[0] = {
    eventCall = {
        eventDef = "waitAIHealthLevel";
        args = {
            num = 3;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_HELL_KNIGHT"; // spawnType
            }
            item[1] = {
                float = 0.5; // target_group_health
            }
            item[2] = {
                string = ""; // group_label
            }
        }
    }
}
```

- `aiType` is the **`eEncounterSpawnType_t`** whose HP level should be checked.
- `target_group_health` is the HP level, in decimal percentage, that you want to reach before proceeding.
- `group_label` is the label for the specified ai type(s). This parameter is optional, can be left blank if inapplicable.

Example Usage

```
item[0] = {
    eventCall = {
```

```
        eventDef = "spawnSingleAI";
        args = {
            num = 3;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_CYBER_MANCUBUS"; // spawnType
            }
            item[1] = {
                entity = "game_target_spawn_1312"; // spawnTarget
            }
            item[2] = {
                string = "check health"; // group_label
            }
        }
    }
}
item[1] = {
    eventCall = {
        eventDef = "waitAIHealthLevel";
        args = {
            num = 3;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_CYBER_MANCUBUS"; // spawnType
            }
            item[1] = {
□□□□float = 0.6; // target_group_health
            }
            item[2] = {
                string = "check health"; // group_label
            }
        }
    }
}
item[2] = {
    eventCall = {
        eventDef = "spawnAI";
        args = {
            num = 4;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_CACODEMON"; // spawnType
            }
```

```
            item[1] = {
□□□□int = 2; // spawn_count
            }
            item[2] = {
                entity = "mod_spawngroup_flying_7"; // spawnGroup
            }
            item[3] = {
                string = "heavy"; // group_label
            }
        }
    }
}
```

In this example, we spawn a Cyber-Mancubus with the label `check_health`. Then we call `waitAIHealthLevel` with said `spawnType` and `group_label`, with `target_group_health` set to `0.6`, so that when the Cyber-Mancubus is <= 60% of its HP, the encounter will proceed.

waitAIRemaining

> An eventCall to wait until a certain quantity of specified ai types remain in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "waitAIRemaining";
		args = {
			num = 3;
			item[0] = {
				eEncounterSpawnType_t = ""; // aiType
			}
			item[1] = {
				int = 0; // desired_remaing_ai_count
			}
			item[2] = {
				string = ""; // group_label
			}
		}
	}
}
```

- `aiType` are the **`eEncounterSpawnType_t`** that you want to wait on.
- The encounter will wait until the quantity of specified ai types present in the encounter are less than or equal to the value of `desired_remaing_ai_count`
- `group_label` is the label that you want to wait on. This parameter is optional, and can be left blank.

Example Usage

```
item[0] = {
	eventCall = {
```

```
□□eventDef = "spawnAI";
□□args = {
□□□num = 4;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_IMP ENCOUNTER_SPAWN_SHOTGUN_SOLDIER"; // spawnType
□□□}
□□□item[1] = {
□□□□int = 4; // spawn_count
□□□}
□□□item[2] = {
□□□□entity = "mod_spawngroup_15"; // spawnGroup
□□□}
□□□item[3] = {
□□□□string = ""; // group_label
□□□}
□□}
□}
}
item[1] = {
□eventCall = {
□□eventDef = "spawnAI";
□□args = {
□□□num = 4;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_HELL_SOLDIER ENCOUNTER_SPAWN_BEAM_SOLDIER"; //
spawnType
□□□}
□□□item[1] = {
□□□□int = 4; // spawn_count
□□□}
□□□item[2] = {
□□□□entity = "mod_spawngroup_21"; // spawnGroup
□□□}
□□□item[3] = {
□□□□string = ""; // group_label
□□□}
□□}
□}
}
item[2] = {
```

```
	eventCall = {
		eventDef = "waitAIRemaining";
		args = {
			num = 3;
			item[0] = {
				eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY"; // aiType
			}
			item[1] = {
				int = 0; // desired_remaing_ai_count
			}
			item[2] = {
				string = ""; // group_label
			}
		}
	}
}
```

In this example, 4 Imps/Shotgunners are spawned, and 4 Soldiers/Hell Razers. By using `waitAIRemaining` set to `ENCOUNTER_SPAWN_ANY` with a `desired_remaing_ai_count` of 0, this ensures that all enemies will need to be killed first.

waitForEventFlag

> An eventCall that waits until a trigger is activated, or a "flag" is raised.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "waitForEventFlag";
		args = {
			num = 4;
			item[0] = {
				eEncounterEventFlags_t = "ENCOUNTER_EVENT_FLAG_USER_EVENT";
			}
			item[1] = {
				string = "User_Flag"; // Flag Name
			}
			item[2] = {
				bool = false; // set to true if looking for flag before the encounter has reached this poin
			}
			item[3] = {
				bool = false;
			}
		}
	}
}
```

Flag Triggers can activate when entering a certain section of the map or when **activateTarget** for the entity is used.
This is useful if you want to activate Energy Barriers to lock you in an arena, or if you need to enter a section of the arena for the encounter to progress.

idEncounterTrigger_RaiseUserFlag is the entity that this eventCall looks for when it is activated.

> The Encounter Manager must contain the **idEncounterTrigger_RaiseUserFlag** entity for it to recognize the flag name.

waitKillCount

> An eventCall that tells the idEncounterManager to wait until a certain quantity of the specified ai type are killed.

Usage

```
item[0] = {
    eventCall = {
        eventDef = "waitKillCount";
        args = {
            num = 3;
            item[0] = {
                eEncounterEventFlags_t = ""; // aiType
            }
            item[1] = {
                int = 1; // desired_kill_count
            }
            item[2] = {
□□□□string = ""; // group_label
□□□}
        }
    }
}
```

- `aiType` is the ai type(s) that you want to wait on.
- `desired_kill_count` is how many of the specified ai type(s) you want to be killed before proceeding.
- `group_label` is the label for the specified ai type(s). This parameter is optional, can be left blank if inapplicable.

Example Usage

```
item[0] = {
    eventCall = {
```

```
        eventDef = "maintainAICount";
        args = {
            num = 8;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_IMP ENCOUNTER_SPAWN_HELL_SOLDIER";
// spawnType
            }
            item[1] = {
                int = 5; // desired_count
            }
            item[2] = {
                int = -1; // max_spawn_count
            }
            item[3] = {
                float = 1; // min_spawn_delay
            }
            item[4] = {
                int = 3; // min_ai_for_respawn
            }
            item[5] = {
                entity = "mod_spawngroup_1"; // spawnGroup
            }
            item[6] = {
                string = "fodder"; // group_label
            }
            item[7] = {
                float = 2; // max_spawn_delay
            }
        }
    }
}
item[1] = {
    eventCall = {
        eventDef = "waitKillCount";
        args = {
            num = 3;
            item[0] = {
                eEncounterEventFlags_t = "ENCOUNTER_SPAWN_ANY"; // aiType
            }
            item[1] = {
```

```
                int = 10; // desired_kill_count
            }
            item[2] = {
□□□□string = "fodder"; // group_label
□□□}
        }
    }
}
item[2] = {
    eventCall = {
        eventDef = "stopMaintainingAICount";
        args = {
            num = 2;
            item[0] = {
                eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY"; // aiType
            }
            item[1] = {
□□□□string = "fodder"; // group_label
□□□}
        }
    }
}
```

In this example, the **maintainAICount** will spawn 5 infinitely respawning Imps or Soldiers. However, once the player has killed 10 fodder, the **maintainAICount** will be stopped.

waitMaintainComplete

> An eventCall to wait until a preceding **`maintainAICount`** finishes.

Usage

```
item[0] = {
□eventCall = {
□□eventDef = "waitMaintainComplete";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterSpawnType_t = ""; // aiType
□□□}
□□□item[1] = {
□□□□int = 0; // remaining_spawn_count
□□□}
□□□item[2] = {
□□□□string = ""; // group_label
□□□}
□□}
□}
}
```

- `aiType` are the **`eEncounterSpawnType_t`** that you want to be wait on. `ENCOUNTER_SPAWN_ANY` can also be used.
- `remaining_spawn_countdesired_count` is how many remaining spawns this eventCall should wait on.
- `group_label` is the same as the label given to the preceding **`maintainAICount`** . This parameter is optional, and can be left blank.

> This eventcall only works if `max_spawn_count` is set to a finite amount, ex. not `-1` .

Example Usage

```
item[0] = {
□eventCall = {
□□eventDef = "maintainAICount";
□□args = {
□□□num = 8;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_BARON";
□□□}
□□□item[1] = {
□□□□int = 1; // desired_count
□□□}
□□□item[2] = {
□□□□int = 3; // max_spawn_count
□□□}
□□□item[3] = {
□□□□float = 5; // min_spawn_delay
□□□}
□□□item[4] = {
□□□□int = 0; // min_ai_for_respawn
□□□}
□□□item[5] = {
□□□□entity = "mod_spawngroup_encounter7B"; // spawnGroup
□□□}
□□□item[6] = {
□□□□string = "priority"; // group_label
□□□}
□□□item[7] = {
□□□□float = 5; // max_spawn_delay
□□□}
□□}
□}
}
item[1] = {
□eventCall = {
□□eventDef = "waitMaintainComplete";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_BARON"; // aiType
```

```
□□□}
□□□item[1] = {
□□□□int = 0; // remaining_spawn_count
□□□}
□□□item[2] = {
□□□□string = "priority"; // group_label
□□□}
□□}
□}
}
item[2] = {
□eventCall = {
□□eventDef = "spawnSingleAI";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_PAIN_ELEMENTAL";
□□□}
□□□item[1] = {
□□□□entity = "master_level_target_spawn_13";
□□□}
□□□item[2] = {
□□□□string = "priority";
□□□}
□□}
□}
}
```

We start with a **`maintainAICount`** that spawns 1 Baron at a time, with up to 3 spawns, each spaced 5s apart. Then we can use `waitMaintainComplete` with `ENCOUNTER_SPAWN_BARON` and `remaining_spawn_count` set to `0`, so that when all 3 Barons have spawned (ie. 0 spawns remaining), the encounter manager can proceed to the next eventCall and spawn the Pain Elemental.

See Also

- **maintainAICount**
- **stopMaintainingAICount**

waitMulitpleConditions

> An eventCall for using multiple wait eventCalls in an idEncounterManager.

> This eventCall is spelled incorrectly in the engine ("Mulitple"), and must be used this way.

Usage

```
item[0] = {
□eventCall = {
□□eventDef = "waitMulitpleConditions";
□□args = {
□□□num = 2;
□□□item[0] = {
□□□□int = 2; // condition_count
□□□}
□□□item[1] = {
□□□□encounterLogicOperator_t = ""; // logic_operator
□□□}
□□}
□}
}
```

- `condition_count` is the amount of wait conditions
- `logic_operator` is the logic operator used for the wait conditions. The only operators are `ENCOUNTER_LOGICAL_OP_OR` (any wait condition is fulfilled) and `ENCOUNTER_LOGICAL_OP_AND` (all wait conditions are fulfilled).

Example Usage #1

```
item[0] = {
□eventCall = {
□□eventDef = "waitMulitpleConditions";
□□args = {
```

```
□□□num = 2;
□□□item[0] = {
□□□□int = 2; // condition_count
□□□}
□□□item[1] = {
□□□□encounterLogicOperator_t = "ENCOUNTER_LOGICAL_OP_OR"; // logic_operator
□□□}
□□}
□}
}
item[1] = {
□eventCall = {
□□eventDef = "wait";
□□args = {
□□□num = 1;
□□□item[0] = {
□□□□float = 10;
□□□}
□□}
□}
}
item[2] = {
□eventCall = {
□□eventDef = "waitKillCount";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterEventFlags_t = "ENCOUNTER_SPAWN_ANY";
□□□}
□□□item[1] = {
□□□□int = 2;
□□□}
□□□item[2] = {
□□□□string = "";
□□□}
□□}
□}
}
```

In this example we have two conditions, set to `OR`. This way, when either 10s pass or any 2 ai types are killed, the encounter proceeds.

Example Usage #2

```
item[0] = {
□eventCall = {
□□eventDef = "waitMulitpleConditions";
□□args = {
□□□num = 2;
□□□item[0] = {
□□□□int = 3; // condition_count
□□□}
□□□item[1] = {
□□□□encounterLogicOperator_t = "ENCOUNTER_LOGICAL_OP_AND"; // logic_operator
□□□}
□□}
□}
}
item[1] = {
□eventCall = {
□□eventDef = "waitAIRemaining";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_BARON";
□□□}
□□□item[1] = {
□□□□int = 0; // desired_remaing_ai_count
□□□}
□□□item[2] = {
□□□□string = ""; // group_label
□□□}
□□}
□}
}
item[2] = {
□eventCall = {
□□eventDef = "waitAIRemaining";
```

```
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_MANCUBUS";
□□□}
□□□item[1] = {
□□□□int = 0; // desired_remaing_ai_count
□□□}
□□□item[2] = {
□□□□string = ""; // group_label
□□□}
□□}
□}
}
item[3] = {
□eventCall = {
□□eventDef = "wait";
□□args = {
□□□num = 1;
□□□item[0] = {
□□□□float = 5;
□□□}
□□}
□}
}
```

In this example, we have three conditions, set to `AND`. This way, there needs to be no Barons and Mancubi, and 5s must have passed as well, in order for the encounter to proceed.

waitRandomKillCount

> An eventCall to wait until a random quantity of specified ai types are killed in an idEncounterManager.

Usage

```
item[0] = {
	eventCall = {
		eventDef = "waitRandomKillCount";
		args = {
			num = 4;
			item[0] = {
				eEncounterEventFlags_t = ""; // aiType
			}
			item[1] = {
				int = 1; // kill_count_min
			}
			item[2] = {
				int = 1; // kill_count_max
			}
			item[3] = {
				string = ""; // group_label
			}
		}
	}
}
```

- `aiType` are the **`eEncounterSpawnType_t`** that will be tracked for the kill count.
- `kill_count_min` is the minimum value for the random kill count.
- `kill_count_max` is the maximum value for the random kill count.

A value will be randomly picked from between `kill_count_min` and `kill_count_max` (inclusive?). Once that kill count is achieved, the wait condition will be fulfilled.

Example Usage

```
item[0] = {
	eventCall = {
		eventDef = "waitRandomKillCount";
		args = {
			num = 4;
			item[0] = {
				eEncounterEventFlags_t = "ENCOUNTER_SPAWN_IMP ENCOUNTER_SPAWN_HELL_SOLDIER"; // aiType
			}
			item[1] = {
				int = 15; // kill_count_min
			}
			item[2] = {
				int = 20; // kill_count_max
			}
			item[3] = {
				string = "fodder"; // group_label
			}
		}
	}
}
```

In this example, once between 15 and 20 Imps/Soldiers are killed, the encounter will proceed.

waitStaggeredSpawnComplete

> An eventCall to wait until the preceding **staggeredAISpawn** finishes.

Usage

```
item[0] = {
□eventCall = {
□□eventDef = "waitStaggeredSpawnComplete";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterSpawnType_t = ""; // aiType
□□□}
□□□item[1] = {
□□□□int = 0; // remaining_spawn_count
□□□}
□□□item[2] = {
□□□□string = ""; // group_label
□□□}
□□}
□}
}
```

- `aiType` are the **eEncounterSpawnType_t** from the preceding **staggeredAISpawn** that you want to wait on. `ENCOUNTER_SPAWN_ANY` can also be used.
- `remaining_spawn_count` is how many remaining spawns this eventCall should wait on.

Example Usage

```
item[0] = {
□eventCall = {
□□eventDef = "staggeredAISpawn";
□□args = {
```

```
□□□num = 6;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_IMP"; // spawnType
□□□}
□□□item[1] = {
□□□□int = 3; // spawn_count
□□□}
□□□item[2] = {
□□□□entity = "mod_spawngroup_encounter2"; // spawnGroup
□□□}
□□□item[3] = {
□□□□string = "fodder"; // group_label
□□□}
□□□item[4] = {
□□□□float = 1; // minSpawnStagger
□□□}
□□□item[5] = {
□□□□float = 2; // maxSpawnStagger
□□□}
□□}
□}
}
item[1] = {
□eventCall = {
□□eventDef = "staggeredAISpawn";
□□args = {
□□□num = 6;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_HELL_SOLDIER"; // spawnType
□□□}
□□□item[1] = {
□□□□int = 3; // spawn_count
□□□}
□□□item[2] = {
□□□□entity = "mod_spawngroup_encounter2B"; // spawnGroup
□□□}
□□□item[3] = {
□□□□string = "fodder"; // group_label
□□□}
□□□item[4] = {
```

```
□□□□float = 1; // minSpawnStagger
□□□}
□□□item[5] = {
□□□□float = 2; // maxSpawnStagger
□□□}
□□}
□}
}
item[2] = {
□eventCall = {
□□eventDef = "waitStaggeredSpawnComplete";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_ANY"; // aiType
□□□}
□□□item[1] = {
□□□□int = 0; // remaining_spawn_count
□□□}
□□□item[2] = {
□□□□string = ""; // group_label
□□□}
□□}
□}
}
item[3] = {
□eventCall = {
□□eventDef = "spawnSingleAI";
□□args = {
□□□num = 3;
□□□item[0] = {
□□□□eEncounterSpawnType_t = "ENCOUNTER_SPAWN_HELL_KNIGHT";
□□□}
□□□item[1] = {
□□□□entity = ai_target_spawn_23";
□□□}
□□□item[2] = {
□□□□string = "priority";
□□□}
□□}
```

```
}
}
```

In this example, we have a **staggeredAISpawn** for 3 Imps, with a 1-2s delay between spawns, and ditto for Soldiers. We can then follow it up with `waitStaggeredSpawnComplete`, with `ENCOUNTER_SPAWN_ANY` and `remaining_spawn_count set` to `0`, so that all **staggeredAISpawn** eventcalls for any ai type will need to finish spawning (ie. 0 remaining spawns), before the encounter can proceed and spawn a Hell Knight.

See Also

- **staggeredAISpawn**