How the Contract Markup Language Works

The contract markup language expresses all the elements of a contract that ought to be modified by a designer. A parser reads the tags and data in the contract txt file, and inserts them as appropriate into the template JSON for that contract type.

In general, the contract text file expects to find exactly the right number of items in each section, and extra items will cause an error when it's parsed. This is because the encounter is full of guids, and those guids are not changeable, and they're also not exposed to the contract author. So each encounter has its own json template, which is a bare contract, and each encounter's json template is parsed into a text template containing the correct number of fields for everything.

Header

ID: The contract's ID
Name: The contract's friendly name
Template: Which encounter the contract is for
Difficulty: Easy, Medium or Hard; this is parsed into the actual difficulty number
DifficultyUIMod: The modifier to the displayed difficulty in the contract UI
ContractScope: The contract's Scope value - When this contract is available typically STANDARD - Other values include TUTORIAL, EARLY_CAMPAIGN, MID_CAMPAIGN, LATE_CAMPAIGN, POST_CAMPAIGN

Miscellaneous Data

This data is just arbitrary top-level true/false and int values.

ContractDisplayStyle: string
DisableNegotiations: bool
DisableLanceConfiguration: bool
DisableCancelButton: bool
DisableAfterAction: bool
ContractRewardOverride: int
TravelOnly: bool
UseTravelCostPenalty: bool
UsesExpiration: bool
ExpirationTimeOverride: int
NegotiatedSalary: int
NegotiatedSalvage: int

```
public enum ContractDisplayStyle
{
    INVALID_UNSET = 0,
    BaseCampaignNormal = 1, // Procedural Contracts
    BaseCampaignStory = 2,
    BaseCampaignRestoration = 3,
    BaseFlashpoint = 4,
}
```

Description

This data covers the description and value of the contract.

ShortDesc (or EmployerDesc):The employer's description of the contract
LongDesc (or DariusDesc): Darius's commentary on the contract
Salvage: The salvage potential for the contract -- always a multiple of 4
FOW: The contract's initial fog of war state

Contract Negotiation and After Action Report Data

[RepresentativeCastDefIDOverride]castDef_OstergaardDefault
[MissionSuccessStatementOverride]Good job!
[BadFaithStatementOverride]You call yourself a merc?
[GoodFaithStatementOverride]Nice try!

The RepresentativeCastDef is the employer that communicates with you on the Contract Negotiation and After Action Report screens. The statements are shown on the After Action Report depending on how the mission went.

AutoComplete Dialogue

[OverrideAutoCompleteDialogueId] guid or bucketdef
[OverwriteMissionCompleteWhenAutoComplete]False

If the EncounterLayer's AutoComplete fires, it normally just pulls something from the AutoComplete bucket, replaces the mission successful dialogue and ends the mission. **OverrideAutoCompleteDialogueId** is allowed to point at a Dialogue Guid in the scene, or a DialogueBucketDefId similar to the default file: "DialogBucketDef_Universal_AutoCompleteEscape". You can also use the value: **NONE** if you don't want AutoComplete to override the default mission success dialogue.

If **OverwriteMissionCompleteWhenAutoComplete** is false, it will play the AutoComplete dialogue, and then play the mission success dialogue before quitting the mission. (Default is true)

Requirements

The contract can have top-level requirements. If the ‘Requirements’ tag appears outside of the context of a previously specified Objective, it’s a top-level requirement. Requirements are formatted as a pipe-delimited string, as follows:

> [**Scope**]|[**Type (stat, tag, or notag)**]|
> For tag requirements: [**tag name**]
> For stat requirements: [**stat name**]|[**operator**]|[**value**]

Operators can be one of: eq, lt, gt, lte, gte, or ne; these correspond to Equals, LessThan, GreaterThan, LessThanOrEquals, GreaterThanOrEquals, and NotEquals.

Here are two examples illustrating the requirement format:

```
[Requirement]StarSystem|stat|Target.IsOwner|eq|1
[Requirement]Company|notag|chain_assassinate_HoloVidCelebrity0
```

Tonnage Limits

The contract can also require tonnage limits at the lance and/or the individual unit level.

```
[LanceMinTonnage]-1
[LanceMaxTonnage]-1
[MechMinTonnages]-1|-1|-1|-1
[MechMaxTonnages]-1|-1|-1|-1
```

-1 means there is no limit.

Objectives

The objectives section contains contract, primary, and optional objectives.

ContractObjective: the human-readable string for the contract objective
Description: A description for the contract objective. I don’t think we actually use this. However, ‘Description’, like ‘Requirement’, is context-sensitive; it applies to whatever section that can be described comes immediately before it. In this case, it will apply to the contract objective.

Objective: the human-readable string for an objective. Objectives are always set *in order*, meaning they map directly to the objectives in the json file, and expect those objectives to stay in the same order in the target json file.

Description: As above. We don't really use this, either.

OnSuccess: This indicates that the result that follows should be put in this objective's OnSuccess result set.

OnFailure: As above, but for the OnFailure result set.

OnSuccessDialogueGUID - The dialogue with this GUID will be played if this objective is successful.
OnFailureDialogueGUID - The dialogue with this GUID will be played if this objective is successful.

```
[Objective]Destroy Enemy Reinforcements
     [Description]
     [OnSuccess]
          [Result]Company|Stat|ContractBonusRewardPct|0.20
     [OnSuccessDialogueGUID]Dialogue_Congratulations
```

Results

Results, like Requirements, use a pipe-delimited format, as follows:

> [**Scope**]|[**type (stat, tag, or notag)**]|
>
> For tag results: [**tag name**]
> For stat results: [**stat name**]|[**value to add**]

Here are two examples illustrating the result format:

```
[Result]Company|Stat|ContractBonusRewardPct|0.25
[Result]Company|tag|chain_assassinate_HoloVidCelebrity0
```

ArtilleryObjective

You can specify which vfx types to use for the artillery. Currently we support the values, ArtilleryShellSingle, ArtilleryShellBarrage, Explosion, OrbitalPPC. Here are two examples:

```
[ArtilleryObjective]ArtilleryShellBarrage
[ArtilleryObjective]Explosion
```

OptionalChunk

Optional chunks are identified by name, which is expected to match the name of a specific optional chunk in the json. This name is never shown to the user, but it is intended to be comprehensible to a designer making a contract. The optional chunk is specified with a simple pipe-delimited format, as follows:

[**Chunk name**]|[**on,off, or requirement**]

Example:

```
[OptionalChunk]Chunk_Ambush (Bonus)|on
```

If you specify Requirement, then the following lines have to be requirements exactly how you do it for contract requirements.

```
[OptionalChunk]Chunk_Ambush (Bonus)|requirement
     [Requirement]StarSystem|stat|Target.IsOwner|eq|1
     [Requirement]Company|notag|chain_assassinate_HoloVidCelebr
     ity0
```

Dialogue

Dialogue: This identifies a specific piece of interrupt dialogue by name. The parser expects to find a dialogue item by that name in the json. It doesn't care what's inside that item, and in fact throws away any contents for the dialogue item, replacing them with its own contents. If the parser doesn't find the dialogue with the same name, then it throws an error.

```
[Dialogue]Dialogue_MissionSuccess
```

If you are adding new Dialogue to the contract you will also have to provide a unique identifier on the dialogue line. The unique identifier is just a string. I suggest using the name of the new dialogue to make things easy. If the ID you provide is not unique, the parser will display an error.

```
[Dialogue]Dialogue_Congratulations|Dialogue_Congratulations
```

Content: This must have a preceding Dialogue item, and will be attached to that item. Content is ordered; it will be added and displayed in the order it's written in the text. The value of the Content tag is the actual text that will be displayed. A contract can add as many of these as it likes to a given Dialogue item, regardless of what's in the template.

The following items all expect to follow a Content item, and will be attached to that item.

Color: A pipe-delimited string indicating R|G|B|A. This is always 1|1|1|1.
Cast: The castdef of the person saying this content item.
Emote: I don't think we use these at all.
Audio: Interrupt dialogue never uses this field.
Focus: A named camera focus target. The names are generally included in the templates so that the contract author can pick the one they want to use.

ExtractViaDropship

Sometimes it doesn't make sense in missions for extraction to occur via dropship (especially in consecutive Flashpoint missions). So this feature allows you to turn off extraction via dropship.

```
[ExtractViaDropship]Chunk_Dropship_Extraction|False
[ExtractViaDropship]EscapeRegionObjective|True
```

There are plans for different encounter layers to be tagged with things like escort_to_building, escort_to_tunnel, etc so that flashpoints can request these maps specifically, but encounter layers haven't been tagged appropriately yet (04/04/2019) and there isn't a method to request encounters by tags so for now, this should only be used in Flashpoint/curated contracts - don't use this feature with Procedural contracts. If you do, someone might escort the vehicles into an empty field and have them just disappear with an "ESCAPED" floatie.

If you turn off the PlayerEscapeRegion you'll want to make sure the AutoCompleteDialogue is overriden appropriately. Otherwise someone might mention Sumire coming to pick you up in the leopard.

Teams

Each team follows the same format: one Lance entry for each lance spawner in the encounter, below which is one Unit entry for each unit spawner in that lance spawner. The numbers of each are set by the encounter, and cannot be changed; the parser fails with an error if the contract author tries to add an unexpected lance or unit.

Team: the named team; one of player1Team, player2Team, employerTeam, targetTeam, targetsAllyTeam, neutralToAllTeam, hostileToAllTeam.

Lance: The named lance spawner being defined.

LanceDef: What specific lance should be spawned. There are two alternate values, ‘Manual’, indicating the Unit values below should be used; and ‘Tagged’, indicating a random lance should be chosen using the specified tags below.
Tags: A pipe-delimited string defining the tags that a lance must have to be chosen for this spawner.
ExcludedTags: The tags that a lance must *not* have to be chosen for this spawner.
SpawnEffects: A pipe-delimited string specifying the named effects that should be applied to each unit in this lance as it is spawned.
DifficultyAdjustment: The modifier to be applied to the contract’s difficulty during the spawn process.

Unit: Each unit in the spawner gets one entry here. The value here is one of Mech, Vehicle or Turret.
UnitDef: The specific unit to be spawned. There are three alternate values, ‘mechDef_None’, which means the spawner should not be used, ‘Tagged’, meaning a unit should be chosen randomly based on the specified tags below, and ‘mechDef_InheritLance’, which means the unit should receive its unitdef from the lance.
CustomName: If the unit should have a special name, it’s specified here (e.g. ‘Brock Armstrong’).
Tags: A pipe-delimited string defining the tags that should be used to randomly select a unit.
SpawnEffects: A pipe-delimited string specifying the named effects that should be applied to this unit as it is spawned.
PilotDef: The specific pilot that should be spawned in this unit. This has an alternate value, ‘pilotDef_InheritLance’, which indicates that the lance will provide a pilot.