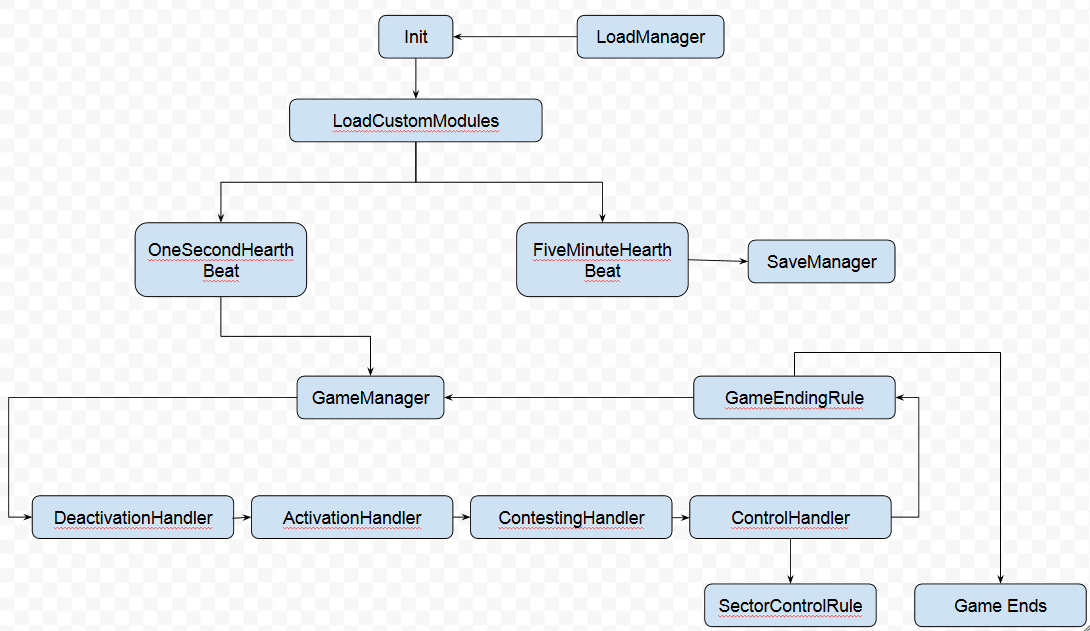
CTI template mission

# Description

The CTI template mission is a base set of tools for creating custom CTI missions. CTI at its heart is a game mode where you have an amount of sectors belonging to different sides. When one side conquers all territories the game ends declaring the side as the winner. Missions become more interesting then when other systems are introduced like custom spawning of enemies, custom spawning, custom arsenals and many other ideas. This can be done by adding modules to the template. This without adjusting any of the core code will give an easy environment for mission designers to develop and maintain CTI missions.

# Core design



* Init: Initialization of shared, client and server scripts. All sectors will be created here either from scratch or loaded from the savegame. All actions before the server initializes and after the server initializes can be extended by custom module scripts. Any respawn handlers will be added client side to the player object. These respawn handlers can be extended by custom module scripts.
* LoadManager: All data from the savegame is loaded here and made available. Can be extended by custom module scripts that run after the default loading.
* LoadCustomModules: Any modules need to be activated here by calling their own init scripts
* OneSecondHearthBeat: Each second any script added to the one second hearthbeat will be executed. These can be extended by custom module scripts.
* FiveMinuteHearthBeat: Each five minutes any script added to the five minute hearthbeat will be executed. These can be extended by custom module scripts.
* SaveManager: Attached to the five minute hearthbeat, this saves all data in the mission to the profilenamespace with the key defined in coreServerConstants.sqf. This key can be overridden in coreSettingOverride.sqf. After the default save mechanic custom module scripts can be added.
* GameManager: The game manager is the engine of the code, each second it triggers an entire sector scan cycle unless there is still a previous run running.
* DeactivationHandler: Decides if a sector needs to be deactivated. Can be extended by custom module scripts after deactivation has occurred.
* ActivationHandler: Decides if a sector needs to be activated. Can be extended by custom module scripts after activation for a sector from neutral to active has occurred.
* ContestingHandler: Decides if a sector is being contested. Can be extended by custom module scripts after it puts a sector in contested condition.
* ControlHandler: Decides if a sector needs to change hands. It decides by using the SectorControlRule as predicate. Can be extended by custom module scripts after control has switched.
* SectorControlRule: Rule that returns a boolean deciding if a sector should change side. It receives the invisible sector object as input. Mission designers can create their own rule script in the override folder and put the script in coreSettingOverride.sqf
* GameEndingRule: This rule is used directly by the GameManager. It returns a boolean deciding if the game should end. It receives three parameters as input. The total amount of sectors in OPFOR hands, the total amount of sectors in BLUFOR hands and the total amount of sectors. Mission designers can create their own rule script in the override folder and put the script in coreSettingOverride.sqf.

# Sector

Sectors in the template spawn from markers as invisible objects. They have certain properties to keep the mission running that shouldn’t be altered by a mission designer. However it’s perfectly fine to read them. A sector can be in a certain condition, which triggers or disables certain mechanics in the template and belongs to a certain side. The markers are placed in the editor as type empty, their name reflects if they become a sector <type>\_<name> fe: sec\_town\_telos.

## Attributes

1. markerName: The name of the marker this sector spawned from
2. condition: The condition the sector currently is in
   1. **neutral** : nothing special is going on with the sector
   2. **active**: the players came close enough to activate the sector that can trigger certain effects
   3. **deactivation**: the sector is marked to be deactivated after a set time it will turn back to neutral
   4. **contested**: BLUFOR and OPFOR forces are both in certain radius of the sector
3. side: The current side the sector belongs to

## Sector types

A sector can be of a certain type which reflects in the icon on the map. Mission designers could create their own sector types if desired. The format is always sec\_<four letter tag>.

1. sec\_town Town
2. sec\_fuel Fuel depot
3. sec\_base Military base
4. sec\_towr Radio tower

# Directory structure

1. <root>
   1. Core (contains all code core to CTI in principle a mission designer should never make changes in this directory and his childeren)
      1. Client (all core client code)
         1. Debug (anything that should work with DEBUG\_ENABLED true on client side)
         2. Functions (all client side functions)
      2. Server (all core server code)
         1. Sector (all sector related code, creating, loading, checking states etc…)
         2. Game (anything that could influence the game flow)
         3. Heartbeat (any heartbeats that should run while the game is active)
         4. Persistence (saving and loading of mission data)
         5. Functions (all server side functions)
      3. Shared (code that should be shared by client and server)
         1. Functions (all shared functions by client and server)
   2. Custom (space especially reserved for mission designers)
      1. Modules (all custom created module belong here)
         1. MyModule (example of a custom module, this directory should contain at least an initModule.sqf)
      2. Override (scripts that override constants from the core are placed here)

# Public functions

This is a list of functions a mission designer can use in their module designs. Some of them are wrappers around the expansion script arrays. These arrays should never be addressed directly but always through the functions.

## Server functions

### F\_addPreServerInitScript

Description: This method adds a custom script you want to run before the server initializes.  
Syntax: [scriptName] call F\_addPreServerInitScript;  
Parameters: scriptName: String  
Return: Nothing

### F\_addPostServerInitScript

Description: This method adds a custom script you want to run after the server initialized.  
Syntax: [scriptName] call F\_addPostServerInitScript;  
Parameters: scriptName: String  
Return: Nothing

### F\_addOneSecondHearthbeatScript

Description: This method adds a custom script that will run each one second.  
Syntax: [scriptName] call F\_addOneSecondHearthbeatScript;  
Parameters: scriptName: String  
Return: Nothing

### F\_addFiveMinuteHearthbeatScript

Description: This method adds a custom script that will run each five minutes.  
Syntax: [scriptName] call F\_addFiveMinuteHearthbeatScript;  
Parameters: scriptName: String  
Return: Nothing

### F\_addSectorActivationScript

Description: This method adds a custom script that will run after a sector activates from neutral to active. The first parameter given to the script being executed is the invisible sector object.  
Syntax: [scriptName] call F\_addSectorActivationScript;  
Parameters: scriptName: String  
Return: Nothing

### F\_addSectorDeActivationScript

Description: This method adds a custom script that will run after a sector actually deactivates and goes back to neutral state  
Syntax: [scriptName] call F\_addSectorDeActivationScript;  
Parameters: scriptName: String  
Return: Nothing

### F\_addSectorContestedScript

Description: This method adds a custom script that will run after a sector turns contested. The first parameter given to the script being executed is the invisible sector object.  
Syntax: [scriptName] call F\_addSectorContestedScript;  
Parameters: scriptName: String  
Return: Nothing

### F\_addSectorCreatedScript

Description: This method adds a custom script that will run after a sector is just created. The first parameter given to the script being executed is the invisible sector object.  
Syntax: [scriptName] call F\_addSectorCreatedScript;  
Parameters: scriptName: String  
Return: Nothing

### F\_addSectorLoadedScript

Description: This method adds a custom script that will run after a sector is loaded from the save. The first parameter given to the script being executed is the invisible sector object.  
Syntax: [scriptName] call F\_addSectorLoadedScript;  
Parameters: scriptName: String  
Return: Nothing

### F\_addSectorType

Description: This method adds a custom script that will run after a sector is loaded from the save  
Syntax: [sectorTag, sectorIcon] call F\_addSectorType;  
Parameters: sectorTag: String unique tag for the sector type fe: sec\_dock,  
sectorIcon: String the icon you want for the sector fe: loc\_Quay  
Return: Nothing

## Client functions

### F\_addRespawnScript

Description: This method adds a custom script to the Respawn eventhandler of the players  
Syntax: [scriptName] call F\_addRespawnScript;  
Parameters: scriptName: String  
Return: Nothing

## Shared functions

### F\_getUnitCount

Description: This method counts all units in a certain radius of a certain side  
Syntax: [position, distance, side] call F\_getUnitCount;  
Parameters: position: Position center of the radius, distance: Number size of the radius to look for units, side: Side only units of this side will be counted  
Return: Number: all counted units

### F\_log

Description: This method is for adding logging, however it will only log with DEBUG\_ENABLED  
Syntax: [[String, Params…]] call F\_log;  
Parameters: [String, Params…] log message to format   
fe [[“Logging data %1”, “myData”]] call F\_log;  
Return: Number: all counted units