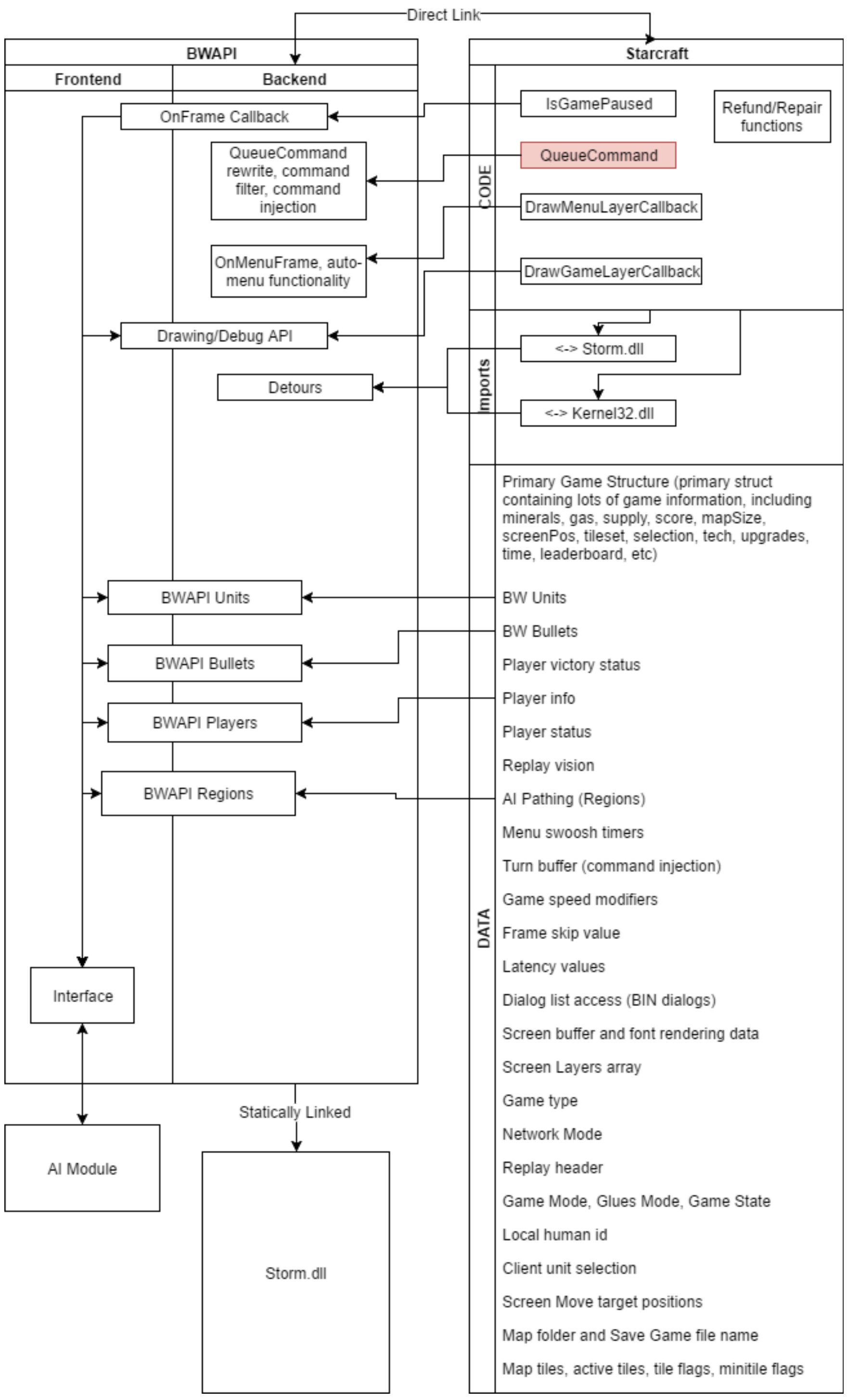
BWAPI 4.x.x Technical Overview



Partial diagram simplifying backend logic. Anything not linked are used for the Game interface or are too mundane to illustrate.

Table of Contents

Preface

This is a technical overview meant for developers who want to pick up developing BWAPI, code/propose a backend to support BWAPI, or learn about how it works. Let me know if I should change anything, restructure, or add to it. Also note that this is for BWAPI 4 (four) and ***not*** BWAPI 5, so it does not contain any planned changes (i.e. better infrastructure). This document is curated to only include items that the API is using (so no windowed mode, auto-menu, additional network protocols, and various other hacks). Note that in many cases, we are referring to internal names that are unknown to us.

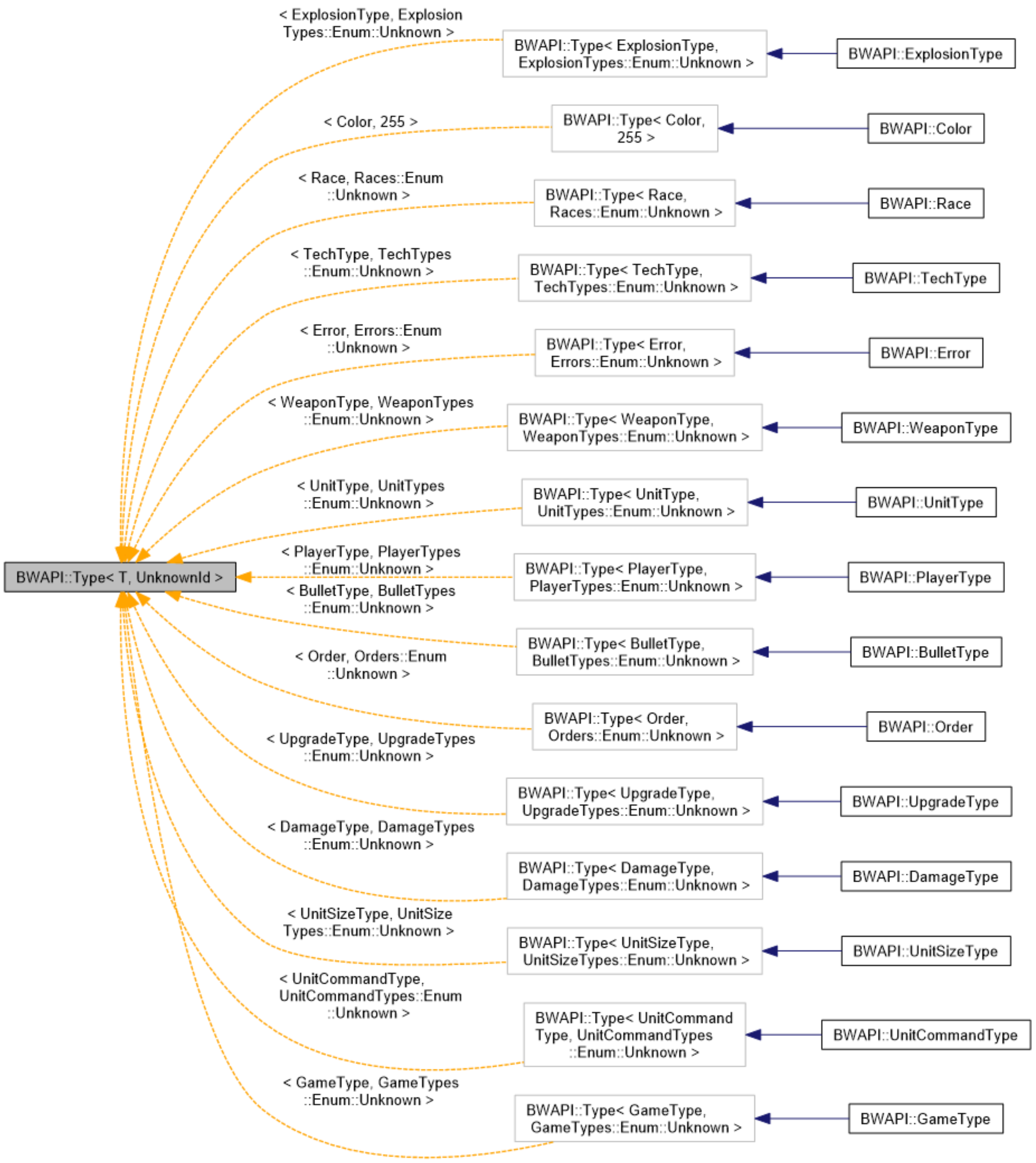
Frontend

BWAPILib

BWAPILib is an independent library and can exist without Starcraft and the BWAPI DLL. There are essentially two primary components: **Types** and **Interface Supplementation**. It contains all types in Broodwar, as well as various API level functions for improving the API (i.e. function overloads, convenience members, or definitions that don’t need to be part of the core).

Types and Type Classes

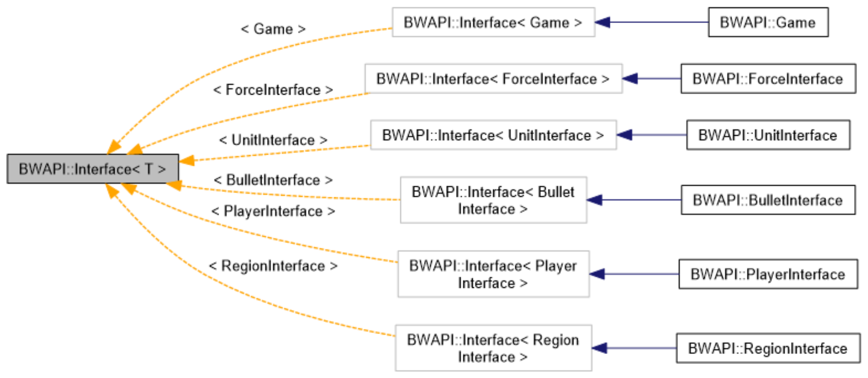
Seen here, along with their type classes. It is a rich API that allows developers to traverse tech trees and requirements, check type properties, and more. It only contains the default unit information, Use Map Settings and modding support is still in the issue tracker.


BWAPI::Type< T, UnknownId >
< ExplosionType, ExplosionTypes::Enum::Unknown >
BWAPI::Type< ExplosionType, ExplosionTypes::Enum::Unknown >
BWAPI::ExplosionType
< Color, 255 >
BWAPI::Type< Color, 255 >
BWAPI::Color
< Race, Races::Enum::Unknown >
BWAPI::Type< Race, Races::Enum::Unknown >
BWAPI::Race
< TechType, TechTypes::Enum::Unknown >
BWAPI::Type< TechType, TechTypes::Enum::Unknown >
BWAPI::TechType
< Error, Errors::Enum::Unknown >
BWAPI::Type< Error, Errors::Enum::Unknown >
BWAPI::Error
< WeaponType, WeaponTypes::Enum::Unknown >
BWAPI::Type< WeaponType, WeaponTypes::Enum::Unknown >
BWAPI::WeaponType
< UnitType, UnitTypes::Enum::Unknown >
BWAPI::Type< UnitType, UnitTypes::Enum::Unknown >
BWAPI::UnitType
< PlayerType, PlayerTypes::Enum::Unknown >
BWAPI::Type< PlayerType, PlayerTypes::Enum::Unknown >
BWAPI::PlayerType
< BulletType, BulletTypes::Enum::Unknown >
BWAPI::Type< BulletType, BulletTypes::Enum::Unknown >
BWAPI::BulletType
< Order, Orders::Enum::Unknown >
BWAPI::Type< Order, Orders::Enum::Unknown >
BWAPI::Order
< UpgradeType, UpgradeTypes::Enum::Unknown >
BWAPI::Type< UpgradeType, UpgradeTypes::Enum::Unknown >
BWAPI::UpgradeType
< DamageType, DamageTypes::Enum::Unknown >
BWAPI::Type< DamageType, DamageTypes::Enum::Unknown >
BWAPI::DamageType
< UnitSizeType, UnitSizeTypes::Enum::Unknown >
BWAPI::Type< UnitSizeType, UnitSizeTypes::Enum::Unknown >
BWAPI::UnitSizeType
< UnitCommandType, UnitCommandTypes::Enum::Unknown >
BWAPI::Type< UnitCommandType, UnitCommandTypes::Enum::Unknown >
BWAPI::UnitCommandType
< GameType, GameTypes::Enum::Unknown >
BWAPI::Type< GameType, GameTypes::Enum::Unknown >
BWAPI::GameType

BWAPI::Error is the only non-Broodwar type, containing error codes that are specific to BWAPI. The ugly superclass hierarchy is for integral and string conversions, and validation code. Each type class has a sister *enum* to be used in cases where a constant C++ object falls short (i.e. switch statement). The underlying values are a one-to-one mapping with internal Broodwar values.

Interface Classes

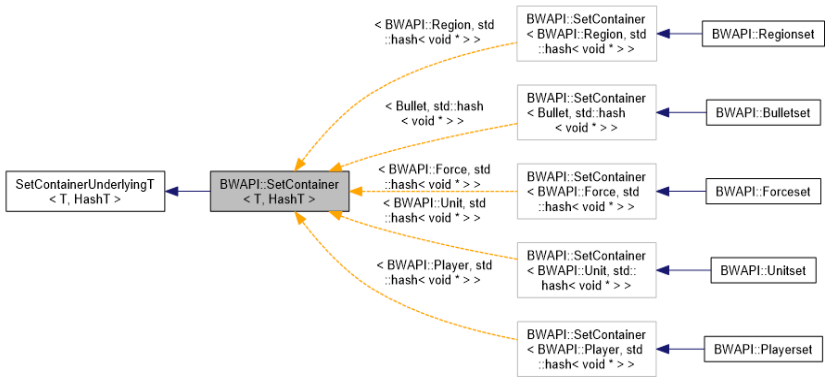
It may be an interface, but there is a bit of additional definition. Each interface class has several overloads and convenience members. **Broodwar**, the variable that wraps the **Game** interface even overloads ***operator<<*** for printing text.



The base interface class can even register events (fictional, just checks a condition each frame) and set client information (attached to the interface for ease of reference).

Containers

Each interface class also comes with its own *Interface* ***set*** class, a container with underlying type *std::unordered_set*, containing convenience members for querying information or giving orders (in case of units) to every element in the set (in groups of 12).


SetContainerUnderlyingT < T, HashT >
BWAPI::SetContainer < T, HashT >
< BWAPI::Region, std ::hash< void * > >
< Bullet, std::hash < void * > >
< BWAPI::Force, std ::hash< void * > >
< BWAPI::Unit, std ::hash< void * > >
< BWAPI::Player, std ::hash< void * > >
BWAPI::SetContainer < BWAPI::Region, std ::hash< void * > >
BWAPI::SetContainer < Bullet, std::hash < void * > >
BWAPI::SetContainer < BWAPI::Force, std ::hash< void * > >
BWAPI::SetContainer < BWAPI::Unit, std:: hash< void * > >
BWAPI::SetContainer < BWAPI::Player, std ::hash< void * > >
BWAPI::Regionset
BWAPI::Bulletset
BWAPI::Forceset
BWAPI::Unitset
BWAPI::Playerset

Position

The position class, and its siblings *WalkPosition* and *TilePosition*, each have rich operators and can arbitrarily be converted to one another. There are also special positions:

- **Invalid** - Generic invalid state.
- **None** - Indicates that there is no position, or it is unset.
- **Unknown** - Indicates that the information being queried is unknown to the player.
- **Origin** - The position (0, 0).

Filters

Filters were largely an experiment. They are basically composable function objects, that can be passed to finder and event functions. For example:

```
myCenter->getUnitsInRadius(512, BWAPI::Filter::IsWorker && BWAPI::Filter::IsIdle &&
BWAPI::Filter::IsOwned) // Returns list of owned, idle workers surrounding myCenter
```

BWAPI.dll

The BWAPI DLL simply implements the interface, exposing only information the player should know about. There are two global flags that can only be enabled at the beginning of the game, which will notify all players in the game that they've been activated:

- **CompleteMapInformation -** Gives the bot full map information, including for units it doesn't own and those in the fog of war.
- **UserInput -** Allows the player operating behind the bot to give unit commands.

BWAPI will also notify players that it has been injected at the start of each game.

Bot Load Mechanism

There are two ways to load a bot.

Native

Native library loading is the easiest solution, loading the AI module DLL into the Starcraft process with everything else, and calling an exported library function from the AI module. It is loaded on game start (*after* the lobby) and unloaded on game end. This allows developers to iterate quickly, compiling their bot while keeping the game open. It is also the most supported interface.

Client

There is also an alternative IPC (shared memory and named pipe bridge) approach. This approach has some advantages and some drawbacks. Note the actual interface provided by BWAPI is largely the same, but instead of exporting a function it cycles through an event loop.

Advantages

- Keeps the bot binary out of Broodwar so it can't cheat by reading BW memory.
- Doesn't crash the game if the bot crashes.

Disadvantages

- Not well supported/maintained.
- Prone to more issues than the native interface.
- Has major design flaws in its current state (i.e. constant-sized arrays that can overflow in sufficiently long games).

AI Module

This contains all of the user's bot logic. The AI module has these callbacks. See also: an example AI module.

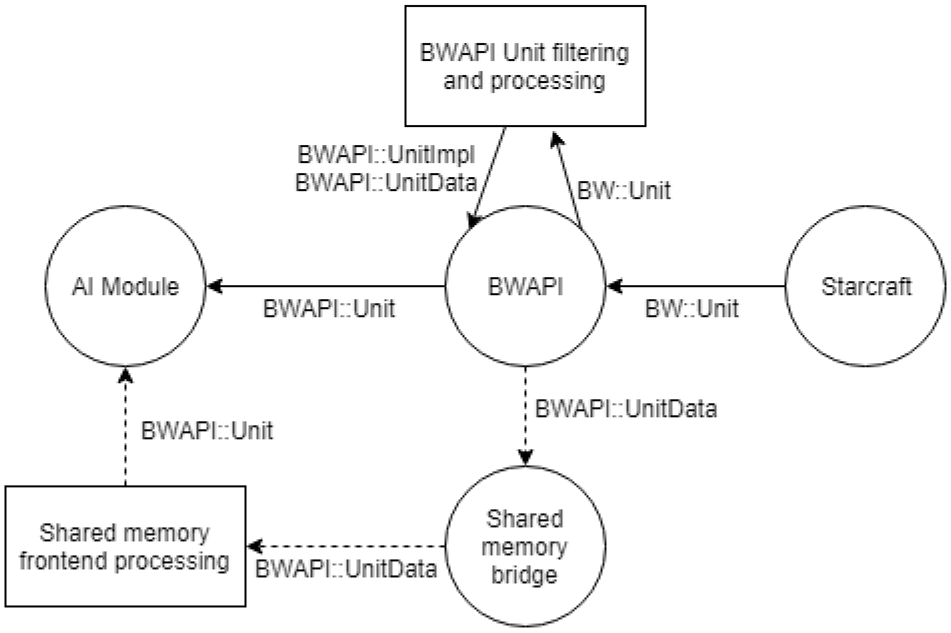
Tournament Module

The tournament module is an additional native-only library with all of the same callbacks as AI modules, but with the ability to apply restrictions, enforce rules, and record results in tournament settings. For example, the tournament module can prohibit the AI module from chatting, disqualify the bot for intentionally using a game glitch, or control the game camera as seen on the SSCAIT stream.

Backend

BWAPI.dll

BWAPI reads game and unit data from Starcraft, applying a filter for what a player should or should not be able to see or know about. It also converts BW types into BWAPI types, BW positions into BWAPI positions, and populates the shared memory for the client interface.



Flow of Starcraft unit data. An occurrence for every unit in every logical frame.

BWAPI will also often recalculate things to avoid additional hooks and memory addresses, if possible.

Low Level Hooking

In order to minimize the number of assembly hooks, BWAPI makes use of everything at its disposal. A lot of offsets were combined into larger known structures, or scrapped and code entirely re-implemented (i.e. drawText was reimplemented). BWAPI treats Starcraft memory as if it owns it.

Statically Linking Storm.dll

By reverse engineering Storm's exported ordinals, we built BWAPI to link against and interact with it. Calls and detours made include:

- **SFile*/SMem*** - Used to read map data to determine whether a player *could be* a non-observer in a Use Map Settings oriented melee map, that had makeshift observer slots (how observer slots were done before observer slots). This covered the general use case and doesn't rely on force names.
- **SEvtDispatch** - Turns out the event hook that **battle.snp** uses to display battle.net messages in the game is a really clean way to print text in game and lobby.
- **SNetSendServerChatCommand/SNetSendMessage** - Used our re-implementation of chat message handling.
- **SNetGetProviderCaps** - Used in a function that determines the number of latency frames. Also used in a reimplementation of "QueueGameCommand".
- **SNetGetTurnsInTransit** - Reimplementation of QueueGameCommand.
- **SNetDropPlayer** - Used to get rid of the "Waiting for players" dialog when the counter reaches 0, dropping the offending players immediately.
- **SDrawGetFrameWindow** - Used to get the window handle (HWND) for several window-based hooks, windowed mode (no longer needed), and send key and button presses (rarely used).
- **SErrGetErrorStr** - Error messages.
- **SFileDestroy** - Workaround for a failure in WINE (See #672 for detail)

Import Detours

Library calls that are intercepted by BWAPI.

Storm.dll

- **SNetLeaveGame** - This is our **onGameEnd** hook.
- **SStrCopy** - Used for both our **onSendText** and **onSaveGame** hooks.
- **SNetReceiveMessage** - Used for **onReceiveText**.
- **SNetSendTurn** - Used to track the last sent turn to calculate the next expected turn time.

Kernel32.dll

- **DeleteFile/GetFileAttributes/CreateFile** - Used for automatic replay saving, based on configurable name and location. Fix for replay filenames in multi-instance mode.
- **FindFirstFile** - Choosing desired map in auto-menu mode. Instead of injecting the name of the map, we trick Broodwar into believing that scanning the maps folder only has the desired map (in any location) and is automatically selected.
- **CreateThread** - Register thread names for Visual Studio debugging purposes.
- **CreateEvent** - Multi-instance hack.
- **GetSystemTimeAsFileTime** - Configurable fixed random seed hack.
- **GetCommandLine** - For nosound config option, since ChaosLauncher does not pass command line arguments.

Code Hooks

- **QueueGameCommand** - Rewritten to be callable from BWAPI and include a command filter, as well as manage/restore selection states for when commands are given.
- **DrawMenuLayerCallback** - Hooked by swapping out the callback in the layers structure. Used for interacting with menus.
- **DrawGameLayerCallback** - Hooked by swapping out the callback in the layers structure. Used for drawing debug data to the screen each frame.
- **IsGamePaused** - The cleanest hook that could be made to capture a callback once for every game frame.
- **RefundMinerals/Gas** - About 9 hooks to capture the refund of resources, and their use in repairing as well.
- **RandomizeRace/InitPlayerConsole** - Hooks to determine which players have chosen random, so that they can be marked as being “unknown” until discovered.
- **ExecuteGameTriggers** - Overrides a millisecond timer to correct a bad behaviour in UMS maps when the millisecondsPerFrame game speed modifiers are set to 0 (for ultra fast speed).
- **UpdateScreenPosition** - Not hooked, but called to update the screen position after changing the screen coordinates.

Starcraft Data Access

BWAPI treats Starcraft memory as if it were owned by it. Since 1.16.1 has a fixed base address, we cast memory addresses to C++ references using a macro.

Primary Structure Access

- **BWGame** - Glorious master structure verbatim copied to save-game files. Contains scores, supply, upgrade, tech, resources, player slots, races, vision, map title, etc.
- **Player Victory Bytes** - Used for determining who won at the end of a game.

- **Player Info Structure** - Used to get player type, race, team, and name.
- **Player Status Flags** - Used in drop player implementation.
- **Replay Vision** - Control behaviour of **setVision** API when in a replay.
- **Turn Buffer (and size)** - Used in **QueueGameCommand** implementation.
- **Game Speed Modifiers** - The list of milliseconds spent per frame for each game type. Overwritten as a hack when BWAPI uses the **setLocalSpeed** API.
- **LatencyFrames** - Used to determine the latency frames between commands.
- **FrameSkip** - Originally exclusively for 16x replay speed, BWAPI writes to this in the **setFrameSkip** API.
- **DialogList** - Pointer to the beginning of the loaded dialogs list. Allows BWAPI to traverse and interact with all game menus.
- **GameScreenBuffer** - Target buffer for rendering additional debug shapes and text.
- **ScreenLayers** - List of screen layers and their function callbacks. The callbacks are replaced with BWAPI's draw hooks.
- **Visible/Hidden Unit Lists and Full Array** - Access to game unit structures.
- **Bullet List and Array** - Access to game bullet structures.
- **GameType** - Determines game type.
- **InReplay** - Determines if a replay is being watched or not.
- **NetMode** - The .SNP network mode (i.e. BNET, IPXN, UDPN, etc). Determines if game is multiplayer or not, as well as if it is on Battle.net or not.
- **ReplayHeader** - Retrieval of replay total frame count, retrieval of the game's random seed, and used to set the replay frame count to avoid a bug that BWAPI introduces that causes replays to be written with 0 frames.
- **GameState/gwNextGameMode** - Used in **leaveGame** implementation to tell Starcraft which next state to enter.
- **gwGameMode** - Used to determine the game mode in some detours, as well as the *QueueGameCommand* implementation.
- **glGluesMode** - Used to determine which menu was loaded to traverse it via auto-menu.
- **g_LocalHumanID** - Determines the current player's ID.
- **Client Unit Selection Group** - Used to save/restore the user's selection so that BWAPI can give commands to other units that were not being selected by the UI.
- **isGamePaused** - The actual boolean value used in the **IsGamePaused** hook. Used in the reimplementation, API function **isGamePaused**, checked for frame update.
- **MoveToX/MoveToY** - Used by API function **setScreenPosition**.
- **MousePosition** - Used in debug drawing **CoordinateType::Mouse** as an anchor, as well as API's **getMousePosition**.
- **ScreenX/ScreenY** - Used for drawing **CoordinateType::Map** as an anchor (draw to map is offset by screen position), and **getScreenPosition** API.
- **CurrentMapFolder** - Written to by auto-menu for automatic map selection.
- **SaveGameFile** - Checks if the destination for an intercepted **SStrCopy** call is this to determine if the game was just saved.

- **TriggerVectors** - UMS triggers, traversed to determine if a player has the possibility of owning any units after the map has started. If not, then the player is considered an observer.
- **MapTileArray** - Used to get map tile IDs.
- **TileSetMap** - Used to get MegaTile properties (i.e. buildability and access minitiles).
- **MiniTileFlags** - Used for MiniTile properties (walkability, height).
- **ActiveTileArray** - ActiveTiles are the current properties of tiles on the map (i.e. hasCreep, visibility, explored, occupied, etc).
- **SAIPathing** - Megastructure containing regions, contours, and which tiles belong to which regions. It is currently only used for the regions, but general and ground pathing would be incredibly useful.

Unit Structure Access

List of members that currently have a meaning (That does not necessarily mean some unused members won't have a meaning in the future). See CUnit.h. (Note: more recent structure here, but this refers to BWAPI's version)

- **index** (non-member) - Used for identification and fabricating a unit command.
- **hitPoints** - getHitPoints()
- **Sprite** - Used to get access to the image and which iscript animation is currently being played (to find if the unit is attacking and check if it's in an attack frame). Used to get the visibility status of the sprite. isAttacking(), exists(), isVisible(), isAttackFrame(), unit death callback
- **moveTarget** - getTargetPosition()
- **movementFlags** - isMoving(), isAccelerating(), isBraking(), isVisible() for cloaked units
- **currentDirection1** - getAngle()
- **position** - getPosition()
- **current_speed** - getVelocity()
- **playerID** - Player identification. getPlayer()
- **orderID** - Primary order. exists(), unit death callback, nuke detected callback, isVisible() for cloaked units, check for zerg building morphing, getOrder()
- **orderState** - Unit death callback. Note this probably has more use cases, i.e. better check for if unit is attacking (rather than move to attack). Is there a list of order states that can be shared?
- **mainOrderTimer** - getOrderTimer()
- **groundWeaponCooldown** - isStartingAttack(), getGroundWeaponCooldown()
- **airWeaponCooldown** - isStartingAttack(), getAirWeaponCooldown()
- **spellCooldown** - getSpellCooldown()
- **orderTarget** - isAttacking(), getNukeDots(), getOrderTarget(), getOrderTargetPosition()
- **shieldPoints** - getShields()
- **unitType** - Various internal unit-specific logic. getType()
- **subUnit** - Used to obtain subunit weapon information, merged into the parent unit for simplification.

- **connectedUnit** - getTransport(), getHatchery()
- **previousUnitType** - Not used for anything yet, but there's a bug in BWAPI with vespene geysers that this may resolve.
- **lastEventTimer/lastEventColor** - Poor man's implementation for isUnderAttack(). This will be replaced in the future with a more accurate implementation.
- **killCount** - getKillCount()
- **lastAttackingPlayer** - getLastAttackingPlayer()
- **currentButtonSet** - Special check if placeCOP() can be used on flag beacons or not.
- **movementState** - isStuck()
- **buildQueue/buildQueueSlot** - getTrainingQueue()
- **energy** - getEnergy()
- **uniquenessIdentifier** - Generating proper unit IDs.
- **secondaryOrderID** - Check if silos are building nuclear missiles.
- **remainingBuildTime** - getRemainingBuildTime()
- **vulture.spiderMineCount** - getSpiderMineCount()
- **carrier.inHangerCount/outHangerCount** - getInterceptorCount(), getScarabCount()
- **fighter.parent** - getCarrier()
- **fighter.inHanger -**
- **building.addon** - getAddon()
- **building.upgradeResearchTime** - getRemainingUpgradeTime(), getRemainingResearchTime()
- **building.techType** - getTech()
- **building.getUpgrade** - getUpgrade()
- **building.larvaTimer** - getRemainingTrainTime() for hatcheries
- **worker.pPowerup** - getPowerup()
- **resource.resourceCount** - getResources()
- **resource.gatherQueueCount/nextGatherer** - isBeingGathered()
- **resource.resourceGroup** - getResourceGroup(), note this was discovered to be the resource base index (for computer players)
- **nydus.exit** - getNydusExit()
- **silo.bReady** - hasNuke()
- **statusFlags** - isCompleted(), isBurrowed(), isCloaked(), isGathering(), isLifted(), isPowered(), isInterruptible(), isInvincible(), isHallucination(), isVisible(), isDetected()
- **resourceType** - isCarryingGas(), isCarryingMinerals()
- **visibilityStatus** - isVisible(), isDetected()
- **currentBuildUnit** - getBuildUnit(), getAddon()
- **rally.*** - getRallyPosition(), getRallyUnit()
- **isBeingHealed** - isBeingHealed()
- **removeTimer** - getRemoveTimer()
- **defenseMatrixDamage** - getDefenseMatrixPoints()
- **defenseMatrixTimer** - getDefenseMatrixTimer()
- **stimTimer** - getStimTimer()
- **ensnareTimer** - getEnsnareTimer()

- … repeat for each status effect
- **stormTimer** - isUnderStorm()
- **acidSporeCount** - getAcidSporeCount()
- **posSortXL, posSortXR, posSortYT, posSortYB** - Used in unit/location search (replica), we can actually rewrite/own this so it's not necessary (there are more efficient algorithms).

Bullet Structure Access

- **targetPosition** - getTargetPosition()
- **type** - getType()
- **position** - getPosition()
- **velocity** - getVelocityX(), getVelocityY()
- **direction** - getAngle()
- **sourceUnit** - getSource()
- **targetUnit** - getTarget()
- **removeTimer** - getRemoveTimer()

Sprite Structure Access

Only used to supplement unit/bullet data.

- **visibilityFlags** - Visibility for units
- **position** - Visibility check for bullets
- **pImagePrimary** - isAttacking(), isAttackFrame()

Image Structure Access

Only used to supplement unit data (specifically isAttacking and isAttackFrame).

- **anim** - animation type, for isAttacking()
- **frameSet** - Used to check for "rest frames" on specific units, to determine if they are in an attack frame or not.