[*Document goal* 1](#_Toc322427804)

[*Application Framework structure* 1](#_Toc322427805)

[*Initialization* 2](#_Toc322427806)

[*Registration of objects* 3](#_Toc322427807)

[*Views Creation* 3](#_Toc322427808)

[*Commands Creation* 6](#_Toc322427809)

Standalone Framework Architecture

July 11, 2011

# Document goal

This document presents the standalone framework architecture implementation. The architecture design is based on requirements that were presented in:

* DeveloperToolsApplicationFrameworkRequirements.docx – describes the framework requirements
* Standalone look and feel.docx – describes the standalone UI requirements

The goal of the documents is to describe the classes, their functionality and sequence of work that is needed to implement the specified requirements.

# Application Framework structure

The application layers structure:



Layers:

* Visual studio: The visual studio is not in our control and therefor it is above everything we do.
* Applications layers: The standalone application and the VS package are the two layers that expose the tools functionality. Both upload the tools and encapsulate them in the UI environment (as a standalone application or inside VS).
* The application layer use the new framework and the tool itself (in the example shown it is the gDEBugger tool set).
* UI components are defined in two libraries: the QtComponents derived library that uses the Qt, and the acComponents derived from Wx.
* Last layer is a set of libraries that give different basic functionality such as:
  + osWrapper: gives classes that are OS independent.
  + Performance counters that can be used in different engines
  + Process Debugger: used for deriving the engine such as kernel debugging, remote debugging and profiling.

## Initialization

When the application executes it will examine a “Tools” directory for installed tools. Each tool will be implemented as a set of modules (DLLs/dynamic libraries). The application will iterate through all the modules in the directory searching for an “amdInitTool” function. Using the amdInitTool functions the tool will be able to register its components.

The registration will be done through several objects managers:

* Register command: Based on the Qt menu item object
* Register views: The are two type of views that can be registered
  + Wx views that will be wrapped by the framework as a Qt view
  + Qt views that will be added also with some wrapping.
* Options: For each tool there will be an option page.
* Debug/Profile engines: Defines relation between the different engines
* Data Collectors: An engine can register the data collector it uses.

## Registration of objects

In order to make things clear different components will register in different managers. The views and commands creator will register in a main creators manager. Option will register in the options manager and engines and collectors will register in a Engines manager.

Managers are singleton objects that have a register interface for each type of object they allow to register:

For example Creators Manager:



Instance is the method to get the singleton object.

RegisterCommandsCreator – registers a single Commands Creator

Wx views and Qt views have different creators since different wrapping is needed.

A single views and commands creator can create one object or several objects, and a single DLL can implement several creators of the same type. There is no limitation on the implementation.

## Views Creation



The framework will pass through all views creators and creates all the views per creator

The main classes:

* CreatorsManager: holds all the creators as explained
* ViewsCreator: the abstract class for the views creator functionality
* WxViewsCreator, QtViewsCreator: abstract classes of views creator to create one of the two views type.

ViewsCreatorAbstract sample functions:

* numberViews: return the number of views that the creator can create
* titleString: the caption of the view and how the view will appear in the “views” menu
* type: type of view:
  + Dock: uses the docking mechanism of Qt
  + MDI: a view that will act as an MDI window
* initialSize: First time the view is created then the preferred initial size. This is a recommended size and might not be physically possible to allocate this size due to initial size of other views.
* dockArea: The initial docking site of the view (left, right, top, bottom).

All of the above functions (except the numberViews) get a view index so the creator knows which view it is answering about.

There are many other functions to get specific information on a view.

Visibility (initial visibility), iconAsPixmap (icon of the view), etc.

Other functions that supplied by the creator and are service functions:

* NumberViewsCreated: How many views were created by the creator. Useful when a creator is used to create several views of the same type.
* widget: get the QtWdiget that wraps the view.

When a view is created the following occurs:

* Create the view with its initial information
* Wrap it when needed:
  + Wx view needs Qt wrapping
  + Qt frame objects that is defined as dock type needs wrapping
* Create a view menu item that enables showing/hiding the view
* Initialize other view information (icons, size, caption).

There is a special case which is called dynamic views, those are views that can have multiple instances and created not during initialization of the application. More detailed information in following sections.

The two derived classes WxViewsCreator, QtViewsCreator are the classes from which the application specific creators need to be derived. If the application creates wx views then the creator must be derived from the WxViewsCreator, and if the application create Qt views then QtViewsCreator must be used.

Detailed sequence diagram



* Get the number of views the creator can create
* For each view
  + Create the wrapper
  + Create the content (Qt or wx)
  + Get view information (size, caption, icon..)
  + Get the type of it
  + Dock the view or add it as mdi according to type.
    - Set view size, icons and other information.
  + Create the wrapping. This can only be done AFTER the wrapper is in its proper position (docked or mdi)
  + Create menu view action
  + Notify the wrapper what it is wrapping
  + Set initial size

Once the view is available then its visibility toggle action can be added.

## Commands Creation



The framework will pass through all actions creators and creates all the actions per creator

The main classes:

* CreatorsManager: holds all the creators as explained
* actionCreator: the abstract class for the views creator functionality

ActionsCreator functions:

* numberActions: number of actions that can be created by the creator
* actionText: Get all action text items:
  + Caption in the menu
  + tooltip: tooltip in the toolbar/menu. If none is supplied the caption is used
  + keyboardShortcut: keyboard accelerator for this action
* iconFile: if supplied then the icon will appear in the menu item and a must if action is needed to appear in the toolbar
* menuPosition: position in the menus including submenu.
  + The format is menu/submenu1/submenu2…
  + The menus order is currently defined by the order in which they are created. It is possible to enhance the menu position to include position but currently there is no need for it.
  + Separator after the item can be defined. Exact format TBD
* toolbarPosition: position in the toolbars.
  + There is no sub toolbar as in menu
  + Separator after the item can be defined. Exact format TBD
* group: Set the group of the action
* handleTrigger: when an action is triggered then this function is called to let the creator handle the action
* handleUpdateUI: same as handleTrigger but for UI update
  + when the menu is opened
  + toolbar needs update
* action: get the specific QAction so it can be manipulated by the handleUpdateUI and group functionality.

All of the above functions (except the numbreActions) get an action index so the creator knows which action it is answering about.

Creation of an action is simpler then creation of a view. A QAction is created and its attributes are set using all the ActionsCreator abstract functions. Once an action was created then it is added to the menus and to the toolbars.

This functionality does not support Editbox and Combobox items but based on the current creator, it is possible to add the functionality by:

* Adding “should use edit/combobox” functions to the creators
* Add handler for the edit/combobox events

Dynamic views

There were several problems that needed addressing

* Creation of views not during initial startup of the application.
* Actions that work in the scope of the active view
* Actions with the same name
  + What happens if one action is “active view action” and the other not
  + What happens if several normal action with same name exist
* How to identify the action when it is no longer unique (can be registered by several actions creator)

**Implemented solution**

Creation of views not during initial startup of the application

A view creator will have an API “is a dynamic view creator” function. When initially passing through all views creators, creators that answer that API with true will not be asked to create their views.

The view creator is still accessible through the manager for creating the view later.

View creation event:

A new event will be defined which is responsible for creating dynamic views in run time. The dynamic view creators will use this event in order to register themselves and create new views in run time. These events should inherit the base event class, and implement the function “indexWhithinCreator” which means what index should be used within the creator, in order to create the view for this event.

## Application framework event listening:

The application framework will define an event listener which will handle events of type “view creation”. This listener will go through it’s mapping between event type and creator, and would create all of the dynamic views that are registered for this event type. The creation function is called with the index specified in the event.

The event and event observer (listener) base classes are currently implemented it gDEBugger framework and can be found in the repository in the GRAPIClasses project.

Actions that work in the scope of the active view

View creator includes a single action creator in it that will be used to create views related actions.

The action creator API will include “Is Active View Action” function. Theoretically it can be decided that action creator in view can only create “views action” and other action creator are not allowed to do so and this will remove the need for this API and its implementation.

“Handle action” and “Action Update UI” will be added to the view creator API (same as the action creator API), so the creator can handle view action.

Actions with the same name

When creating a “View action” it will be checked if the item already exists in the menu in the target position. If it exists then a new action will not be created.

In case of other actions, they will always be created. This means we can have menu items with the same name. Our goal is to add status bar info about each item that can be different between two actions with the same caption, clarifying which action is which.

How to identify the action when it is no longer unique (can be registered by several actions creator)

When registering an action, then the system allocates a unique serial number that will be returned to the creator. If the action was already created (in the case of view action that was already created) then the action id that was allocated in the creation will be used and a new one will not be allocated.

The action creator will store the new identifier so it can identify which action to execute when receiving the notification about an action.

**Action activation**

Normal actions, when activated, the creator receives the “Handle action” notification.

The action creator converts the identifier it receives with the original action identifier.

This is a simple mechanism.

View actions will behave differently when defined as “View actions”.

* Find active view
* Find view creator and view index
  + The action creator cannot be used since there maybe multiple actions for the same command.
* Send the view the “Handle action” and “Action Update UI” notification with the action unique identifier.
* The view creator can find the action through the map in its action creator table.

The view creator handles the notification.

Configurations

There are two types of items under the configuration definition

* Options: User defined setting for the different tools
* Screen layout: Position of the different UI components:
  + Views:
    - Position
    - Docking state
    - Visibility
  + Toolbars:
    - Visibility
    - Items added or removed
  + There can be several layouts based on current working mode (UI Context):
    - Debugging mode
    - Profiling
    - Not executing
    - Etc...

The mechanism suggested to store the configuration will use QSetting and groups in it.

Groups:

* Option group: Save the options per tool. Each tool will also have a subgroup so modifying a specific tool will not affect other tools.
* UI Context: Per UI context a set of all the views and toolbars setting will be stored. Each time there is a change in the UI context the current state is saved before changing the UI context.

## Engines & Data Collectors



When the application is initiated all the tools register the debug engines.

When the debugged application is started, it is controlled by the application. Before starting the debugged application the data collectors can be loaded and connected to the debug application.

For example replace a system DLL by changing the order DLL directory and forcing the replacement DLL to load first.

When registering the debug engine and data collectors the way of connection is defined. Based on it the proper action to do the preload/connection will be executed.

The debug engine controls its data collectors and gathers data from them.

## Project settings

The application framework implements projects management.

**Framework project**

A CodeXL project setting a defined in the class **apProjectSettings**

Following are some of the fields contained in this class. This list is not full since we add / remove / rename these settings often.

* Project name
* Application executable path
* Application working directory
* Command line arguments
* Environment variables list

A project can be created, opened, saved and edited by menu commands that are added by the framework.

Each of the projects created by CodeXL front-end, is saved in an XML file (with extension “cxl”). The projects are saved in the user application data folder, under a folder structure “AMD/CodeXL”.

**Project extension**

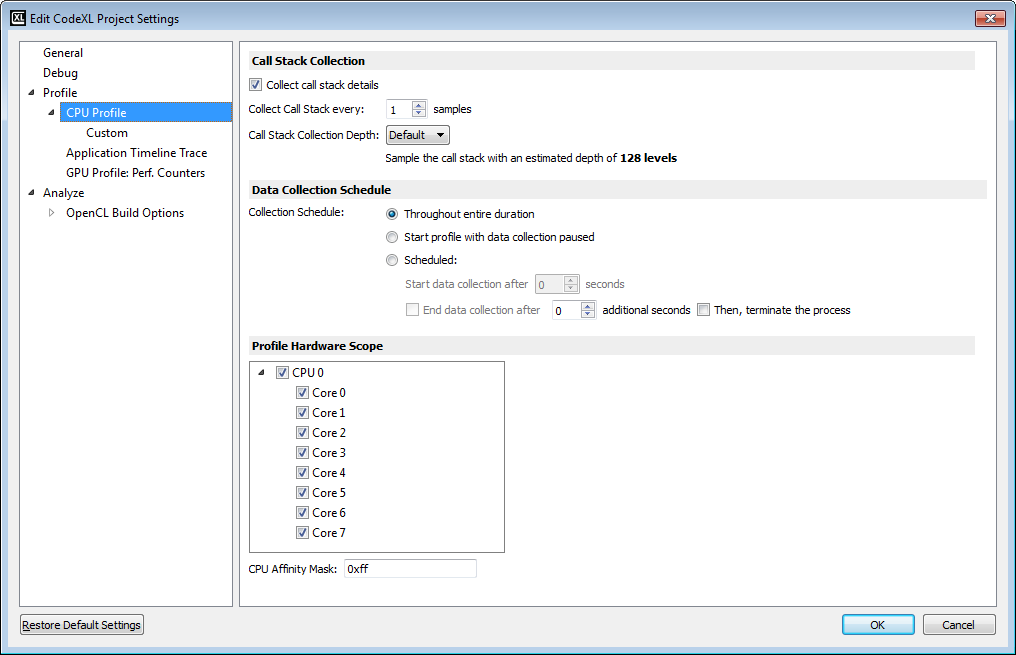
CodeXL can be extended by plug-ins. Therefore, each of the plug-in can also extend the definition of project settings.

Each of the plug-ins, when extending CodeXL project settings, inherits the class

**afProjectSettingsExtension**.

**afProjectSettingsExtension** inherits a Qt group box, and is defining the controls that queries the user for the extension specific project settings.

* **Initialize** – initializeQtwidgets structure.
* **ExtensionXMLString** – unique string defining the plugin (for example “CpuProfile”).
* **ExtensionTreePathAsString** – list of strings, separated by “,” defining the tree path in the project setting dialog tree. (See screenshot below – the CPU Profile settings extension should return "Profile, CPU Profile").
* **GetXMLSettingsString** – Build the current settings as an XML string. This settings will be saved to the XML file, and later will be used to load the project plugin settings
* **SetSettingsFromXMLString** – Extract the current settings from the input XML string, and set the Qt widgets in the dialog page accordingly.
* **RestoreDefaultProjectSettings –** restorecurrentsettings to default.
* **AreSettingsValid** – return true / false is the current settings are valid.



**afProjectManager**

This class is managing the projects in the application framework. The manager returns the current project settings and manages the project extensions.

Once a project extension is implemented,

**afProjectManager::instance().registerProjectSettingsExtension** should be called. This should be called while initializing the extension (in parallel to registering the views creators).

**NOTICE:**

This code segment has a major design problem that makes it difficult to implement changes in this area. The current mechanism assumes that there is only one “current project”. The framework, the plugin extensions, all are working on the “current project”.

In real time, there are sometimes 2 projects. The one that is currently closed and saved, and the new one that is opened. Due to events handled non-sequentially, many times it happens that the following set of operations can be called:

* Open project “B.cxl”
* Save the opened project “A.cxl”.

The interface created is not flexible enough, since the extension can only get / set XML settings string. The solution for this problem is to enable multiple projects.

This means that set / get XML functions will have to get a project name as a parameter, and then there will not be clashes as there are now.

## Options (not implemented yet)

The registration of options format: here are two possible way to register the options:

1: Register a Qt dialog that will be encapsulated in the main options dialog as a node in the options tree.

Advantages:

* Enables direct control on the changes from the dialog inside the tool itself. There is no need to send notification about option changes.
* Full Qt widgets inside the dialog

Disadvantages:

* Might not have a unified look & feel to different options dialog

2: Register specific items that are needed as option and their UI definition (option X as checkbox, option Y and Z as a radio button group, etc…)

Advantage:

* Single mechanics controlled by the application to all options. Everything is controlled in one place
* Unified look & feel to the whole application

Disadvantage:

* Might be a complex interface to define different type of UI components
* Not all UI widgets will be enabled
* Every change needs to send a notification event.

## 

## Properties view:

The framework publishes a properties view, which can be used whenever the extension needs to output formatted data to the user. In order to get an instance of the properties view, user should do the following:

1. Get an instance for the application command object:

afApplicationCommands\* pCommands = afApplicationCommands::instance();

1. Retrieve the pointer for the properties view:

afPropertiesView\* pPropertiesView = pCommands->propertiesView();

1. Once you have the properties view, you can do one of the following:
   1. Set the HTML text to the properties view – pPropertiesView->setText
   2. Use our object that formatting the HTML table to look in the CodeXL unified look.

Example:

afHTMLContent content(“My Page Title”);

// Add a sub title to the HTML table

content.addHTMLItem(afHTMLContent::AP\_HTML\_SUB\_TITLE, “Subtitle1”);

content.addHTMLItem(afHTMLContent::AP\_HTML\_LINE, “Row1-Col1”, “Row1-Col1”);

Add as many lines and subtitles as you like, and once you’re done, get the HTML as string

content.toString(myString);

pPropertiesView->setText(myString->asAsciiCharArray());

## Logging and assertion handling:

Logging

While using the framework, there would be few log files that would be initialized.

1. Client Log file the application framework log. This log is initialized by the framework, and any extension that is linked with the application framework dll can use the logging capabilities of the framework.
2. Engine log file - A log file initialized and being written by the debug / profile engine module inside the user application (the debugging / profiling server). This is initialized by the engine module, supplying a file name prefix (see below) to write into.  
   Only one engine module is required to initialize the log file, afterwards, all modules may use the log. It is recommended that the lowest-level (most independent in the linking hierarchy) module do this at the beginning of its module constructor, so that logging and assertions are available for the module construction and initialization.
3. Support application log files - Any support executable (e.g. Remote debugging server, process launcher, etc.) should have a log file, initialized and managed by it, similar to the Engine log file.  
   In this case as well, only one module in the support application is required to initialize the log.

**Client File path**: %TEMP%\gDEBugger\_%USERNAME%.log

* + The file name would change in the next framework version

**Engine File path**: %TEMP%\gDEBuggerServers\_%USERNAME%.log

* + The file name would change in the next framework version
  + Each engine could have a different log file, or they could all share the same.
    - CodeXLDebuggingServers\_%USERNAME%.log, CodeXLGPUProfilingServers\_%USERNAME%.log
    - CodeXLServers\_%USERNAME%

**Support Application File path**: %TEMP%\<Support Application Name>\_%USERNAME%.log

* + Each support application should have its own log, to avoid conflicts when more than one is running.

**File format:**

* + **Header:** contain system information details
  + **Sessions** -each session contain information logged between the last “Go” and “Terminate” commands (or the application’s normal exit). The session start and end time is displayed. Follow [these instructions](#_Start_a_session) to start your own session, when your extension engine is fired.  
    For the application framework and support applications, a session is considered a run of the application that is writing the log.
  + **Log Messages** the message structure is as follows:
    - ## [Time] [Time Stamp] [Message Severity] [Current Thread ID] [Function Name] [File Path] [line] [Message Text]

The message format could be difficult for reading, but we have a log viewer which parses the log display its data in a more friendly way

* + Message Severity - Error, info, debug, extensive. Setting the log level to a specific level will display all messages from that level and below (e.g. “info” log level will print error and info messages, but not debug or extensive messages)

Assertion failure handlers

The application framework uses objects defined as assertion failure handlers. Multiple assertion failure handlers can be registered in the application.

Upon initialization, the debug log is registered as an assertion failure handler, showing failed assertions as error-level messages.

In debug configuration, the application framework provides a dialog-based assertion failure handler, displaying the assertion message, and providing the option to ignore one instance of the assertion, ignore all instances of the specific assertion, and generate a breakpoint exception to allow debugging of the assertion failure location (if the framework is being debugged, the debugger will catch the breakpoint).

How to:

1. Start a session in the log file (in Engine module and support applications only - the framework does this for extensions):
   1. Instantiate the log class  
      osDebugLog& theDebugLog = osDebugLog::instance();
   2. Initialize the log file by supplying the file name (see above for the appropriate file name prefix)  
      theDebugLog.initialize(L"SupportApplication");  
      theDebugLog.initialize(L"CodeXLProfilingServers");
   3. Set the initial log severity (see below). It is recommended to pass the log severity as an environment variable to the server module and to support applications, using the osDebugLogSeverityToString and osStringToDebugLogSeverity functions.  
      For example:  
      The debugger module passes the logged severity as a string in the SU\_DEBUG\_LOG\_SEVERITY environment variable.  
      The remote debugging server passes the logged severity as a string in the RD\_DEBUG\_LOG\_SEVERITY environment variable.
2. Add a log message:
   1. Use the OS\_OUTPUT\_DEBUG\_LOG(wchar\_t\* logMessage, osDebugLogSeverity logMessageSeverity) macro.
3. Change log file severity:
   1. GUI - only available in next framework version
      1. Tools->Options->Advanced->Debug log level
   2. Code - use the setLoggedSeverity function on the osDebugLog class instance, e.g. osDebugLog::instance().setLoggedSeverity(OS\_DEBUG\_LOG\_DEBUG);
4. Add an assert condition in the code:  
   Use the assertion macros. Note that according to the coding standards, the booleanExpression field should not be a pointer or numerical value, but only a Boolean expression. Also note that comparing a Boolean variable to true or false is redundant.  
   E.g:  
   bool isValid, isInvalid; -> GT\_ASSERT(isValid);, GT\_ASSERT(!isInvalid);, GT\_ASSERT(false != isValid);, GT\_ASSERT(true == isValid);  
   int someNum; -> GT\_ASSERT(someNum);, GT\_ASSERT(0 != someNum);  
   void\* somePtr; -> GT\_ASSERT(somePtr);, GT\_ASSERT(NULL != somePtr);
   1. GT\_ASSERT(booleanExpression)  
      If booleanExpression is false, trigger the assertion failure handlers. The assertion message will be "Assertion failure (<booleanExpression as a string>)".  
      E.g, in the samples above, L" Assertion failure (isValid)", L" Assertion failure (0 != someNum)", L" Assertion failure (NULL != somePtr)"
   2. GT\_ASSERT\_EX(booleanExpression, failureMessage)  
      If booleanExpression is false, trigger the assertion failure handlers. The assertion message will be failureMessage
   3. GT\_IF\_WITH\_ASSERT(booleanExpression)  
      Convenience macro for  
      GT\_ASSERT(booleanExpression);  
      if (booleanExpression)  
      Note that booleanExpression is evaluated only once, so it is safe to use function calls with this macro.
   4. GT\_IF\_WITH\_ASSERT\_EX(booleanExpression, failureMessage)  
      Convenience macro for  
      GT\_ASSERT\_EX(booleanExpression, failureMessage);  
      if (booleanExpression)  
      Like GT\_IF\_WITH\_ASSERT, this evaluates booleanExpression only once.
   5. GT\_RETURN\_WITH\_ASSERT(booleanExpression)  
      Convenience macro for  
      GT\_ASSERT(booleanExpression);  
      return booleanExpression;
   6. GT\_ASSERT\_ALLOCATION(pointerExpression)  
      Should be called immediately after any time pointerExpression is assigne with malloc, new or new[] statement.  
      This macro does the following, if the allocation failed (i.e. pointerExpression is NULL):  
      - Prints a message to the std error  
      - Prints an assertion failure with "Allocation Failure" as the message.  
      - Exits the application.  
      When verifying a pointer variable that was received as a parameter, use GT\_IF\_WITH\_ASSERT(NULL != pointerExpression) instead, as this macro will exit the application on failure.
5. Add an assertion failure handler:
   1. Implement a class that inherits gtIAssertionFailureHandler.
   2. Implement the mandatory function onAssertionFailure(const wchar\_t\* functionName, const wchar\_t\* fileName, int lineNumber, const wchar\_t\* message) - which will be called in a case where an assertion failure had occurred.
   3. On the class's constructor, call  
      gtRegisterAssertionFailureHandler(this);
   4. On the class's destructor, call  
      gtUnRegisterAssertionFailureHandler(this);

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Changed by | Changes | Date |
| 1.0 | Gilad Yarnitzky | Initial Version | 14 August 2011 |
| 1.1 | Gilad Yarnitzky | Updates and Dynamic views added | 5 October 2011 |
| 1.2 | Sigal Algranaty | Projects extensions | 17 April 2012 |
| 1.3 | Uri Shomroni | Added Logging and assertiong handling section | 17 April 2012 |