



Developing Applications for the OpenFrame

A comprehensive guide to developing in the OpenPeak
OpenFrame application framework

OpenPeak Inc.
Boca Raton, Florida

March 23, 2011 Framework 4.0 Revision 29947

OpenPeak Confidential and Proprietary

Table of Contents

Introduction 1

Assumptions 1

Background Information 2

Application Requirements 3

The icon movie 3

The language.xml file 3

The application movie 4

The info movie 5

Accessing local files 5

The Global Stage Object 6

Interacting with the Framework 6

ActionScript Restrictions 6

The basic components 7

Keyboard 7

Alert window 9

Option List 9

Confirm Window 11

WaitAnimation 12

Date Entry Window 12

Time Entry Window 12

Sound 13

Playing Sounds 14

OPSound Events 14

Playback Status 15

OPSound Termination 15

Advanced Components 17

Movie Player 17

Slideshow 17

Volume Controller 19

Storing Application Information 20

Cookies 20

Text Files 20

Shared Objects 21

The Asynchronous Events Model 22

Establishing Background Processes 22

Handling Asynchronous Events from Background Processes 22

Handling Focus Changes 23

Telephony 25

Telephony Examples 26

Click2Dial 26

VoiceMail Notification 26

Testing Applications 28

Testing on a PC 28

Testing on an OpenFrame 29

Appendixes 30

Flex Development 30

The config.xml file 30

Core Component Paths 30

Flash Libraries 30

Default Settings 30

Available Languages 31

The op.utils package 31

OPBtn 31

OPBtnI 31

OPBtnMM 32

OPBtnToggle 32

OPBtnToggleWTI 32

OPCellCheckBox 33

OPCellOption 33

OPEvent 34

OPList 34

OPLoader 34

OPScrollPane 34

OPWindow 35

The OPLink Class 35

application [read-only] 35

applicationDirectory [read-only] 35

applicationID [read-only] 35

dateFormat [read-write] 36

language [read-write] 36

region [read-only] 36

timeFormat [read-write] 36

addScreensaverHold 37

classDefintion 37

getLocal 37

getGlobal 37

loadApplication 38

loadMoviePlayer 38

loadSlideShow 38

loadVolumeControls 39

removeScreensaverHold 39

write 39

remove 39

Shared Libraries 40

system 40

The fl package 44

Assets.flc 44

Introduction

The goal of this document is get you, the developer, creating applications for the OpenPeak OpenFrame as quickly as possible. Therefore we have organized the bulk of this document more like a quick start guide then a detailed perspective. In-depth descriptions of the framework's architecture and libraries can be found the appendixes at the end of this document.

Assumptions

This document assumes that you have at least an intermediate level of experience with Adobe Flash version CS4 and ActionScript 3. The goal of this document is not to teach you how to use the Flash IDE or program in ActionScript but how to develop an application for the OpenFrame application framework and run it on the OpenFrame.

Background Information

The OpenFrame Application Framework runs on the Adobe Flash Player version 10 and has been developed using the Adobe Flash CS4 toolset. The foundation of the framework is a collection of Flash movies (.swf files) that present, manage and interact with a series of externally defined Flash applications. The framework provides a rich set of APIs and RIA features that allow each external application developed under the proceeding guidelines to interact with the core movies and the OpenFrame hardware.

Application Requirements

Developing an application for the OpenFrame is relatively simple. We have strived to minimize the application development requirements as much as possible. Each application must contain at least the following files:

- `main.swf`
- `icon.swf`
- `language.xml`
- `info.swf`

The icon movie

All applications intent on loading from the main menu must supply an icon movie. Icon movies have the following requirements:

- Dimensions of 100×100 pixels.
- Display an appropriately styled button instance with up and down states on the stage at all times.
- Do not add items directly to the `.stage` property of the document level class.
- Keep all visible attributes inside the region defined by the 100×100 pixel stage.
- Do not edit any properties at the document level that will affect the scale, position, orientation, or visibility of the movie in the main menu.

In general, the Icon movie only needs to contain one frame in the timeline with the interactive button already placed on the stage. More advanced applications that require asynchronous events (i.e., a phone that needs to display and incoming call window) may need to add code into the icon movie's document class to handle these situations. Handling asynchronous events is explained in detail further in this document.

The language.xml file

For the sake of internationalization, all strings that need to be displayed in the application should be added to the `language.xml` file. The framework loads this file when the main menu is constructed. All strings should be added to the file in the following format:

```
<[string id] [language id]="[string]"/>
```

You may add as many language IDs as needed. An ID is the ISO 639-1 two-letter name of the language. The provider of the OpenFrame platform you are developing for should dic-

tate this. The following is an example of a `language.xml` file that contains English and French.

```
<copy>
  <mm en="Calculator" fr="Calculatrice"/>
</copy>
```

To access this in an application you need to reference the `translate` method in the static `OPLang` Class located in the `op.framework` package. Here is an example using the `language.xml` example from above:

```
package
{
    import flash.display.MovieClip;
    import op.framework.OPLang;
    import op.framework.OpenFrameApplication;

    public class Example extends OpenFrameApplication;
    {
        function Example()
        {
            super();
            trace( "mm" = OPLang.translate("mm")); // traces Calculator
        }
    }
}
```

All `language.xml` files must contain the "mm" node. This value will be displayed under the icon on the main menu.

Strings that are used often can be added to the framework translation string table, to access it use:

```
OPLink.translate("stringName");
```

The application movie

This is the actual Flash application that will load when an icon on the main menu is depressed. The application must adhere to the following requirements:

- Dimensions of 800x410 pixels.
- Do not add items directly to the `.stage` property of the document level class.
- Do not edit any properties at the document level that will affect the scale, position, orientation, or visibility of the movie in the main menu.
- Do not unload the application using the document level `loaderInfo.loader` attribute.

Each application in the framework must have a document class that extends the `OpenFrameApplication` class found in the `op.framework` package as in the following example:

```
package
{
    import op.framework.OpenFrameApplication;
    import op.framework.OPLang;

    public class Example extends OpenFrameApplication
    {
        public function Example()
        {
            //class constructor
            super();
        }

        public override function onApplicationUnload(e:*)
        {
            /*
             * This event handler called when the application is unloaded
             * Your application should clean up all references and pointers here.
             */
        }
    }
}
```

The info movie

The `info.swf` is used within the OpenPeak AppShop as an overview of the content of the application. This overview is viewed by the end user as part of the AppShop browsing experience. The info movie can be as simple as a screen capture or can contain a limited animation. The specifications for the `info.swf` movie are:

- Contents: Screen shot, slide show or other limited animation
- Resolution: 400x205
- File size: Optimal file size is between 30-50k.

Accessing local files

Some applications will need to load or access additional support files from within their application's directory on the file system. At runtime, an application will not be able to access these files through a standard relative path. Instead, they must add the relative path of the support file to the application's directory path by accessing the static `OPLink.applicationDirectory` property. For example, the following block of code will load a `config.xml` file located in a given application's directory on the file system:

```
var ldr:URLLoader = new URLLoader();  
ldr.addEventListener(Event.Complete, onComplete, false, 0, true);  
ldr.load( new URLRequest ( OPLink.applicationDirectory + "config.xml" ) );
```

The Global Stage Object

The application framework does *not* support the global Stage instance provided by the Adobe Flash interface and does not support any applications that use it. The Stage Object, referenced from `.stage` in any Display Object, must not be used for any reason to prevent positioning, layering, and reference errors. It is important to note that an application or icon will not be on the Stage when the document level constructor is executed so all Display Objects in the document hierarchy will have a `.stage` attribute equal to null.

Interacting with the Framework

Many features of the framework will require you to interact with the classes defined in the “op” package. Reference classes are physically located in the root directory of the SDK distribution. In order to properly access them during development and avoid compilation errors, the root directory of the SDK must be added to either the global `classpath` (ActionScript 3.0 only) or to each application `classpath` found under “Publish Settings”. The latter is usually more desirable to ensure that you are compiling against the correct classes. Look at the “Publish Settings” in the examples provided under apps in the root directory of the SDK distribution.

These are the basic requirements to develop an application in the OpenFrame framework. Advanced features of the framework, such as keyboard input, pop-up windows and lists, non-Flash supported media, and asynchronous event will be explained in the following sections.

ActionScript Restrictions

Currently all ActionScript packages and features are supported with the following exceptions:

- The “mx” package distributed with Flex SDK Distribution is not supported.
- All Adobe AIR specific features or packages are not supported.

The basic components

Several components have been provided in the SDK to help present a unified look and feel for tasks such as keyboard input, alert windows, option lists, etc. These are incorporated in the `opcomponents.swf` file which can be found in the `components` directory. To leverage them, you will need to become familiar with the `OPLink.classDefinition` API. The following steps outline how to leverage a common component.

1. Obtain the component's class definition using `OPLink.classDefinition` API.
2. Create an instance of the component from the obtained class definition
3. Register event listeners with the class instance accordingly.
4. Add the component to your application's local display stack.
5. Utilize the component
6. Remove the component from the stage and release for garbage collection.

In general, all basic `OpenFrame` components should be loaded, used once, and unloaded on their respective `BACK` and/or `SAVE` events. Descriptions of each component with usage examples can be found in the proceeding sections. Additionally extensive working examples and source code have been included in the "apps" directory of the SDK distribution.

Keyboard

The keyboard component provides an interface for all of your basic text input needs. The following attributes define the type of keyboard as well as its run time behavior.

- `type:String` – Dictates the available keyset using the following values.
 - `ABC`: Standard complete keyset
 - `123`: Numeric only keyset
 - `IP`: Numeric keyset with "."key.
 - `hex`: Keyset containing the 0-9 and A-F characters
 - `phone`: Numeric keyset formatted like a dial pad.
- `multi:Boolean` – Allows for scrolling and carriage return input. This only applies to the standard keyboard.
- `txt:String` – The starting text for the keyboard.
- `rtnCmnd:Number` – Optional numeric value that will with returned in keyboard events.
- `password:Boolean` – Masks the user-entered characters as "*". Only applies to the standard keyboard.
- `maxChars:Number` – Maximum number of character that a user may enter.
- `autoComplete:Boolean` – Set to true to present a keyboard with an auto complete list.

To access the keyboard component in an application, instantiate an instance of it as demonstrated in the following block of code.

```
var params:Object = new Object()
params.type = "ABC";
params.muilt = false;
params.txt = "";
params.rtnCmd = 1;

var OPKeyboard:Class = OPLink.classDefinition("components", "OPKeyboard") as
Class;
var kb:* = new OPKeyboard(params);
kb.name = "keyboard";
addChild(kb);
```

Additionally, you must register appropriate handlers for the BACK and SAVE events. Constant values for these events are in the `op.utils.OPEvent` class. The SAVE and BACK events will surface when a user clicks on the "Save" and "Back" buttons respectively. You may also optionally register for a CHANGE event which is dispatched when the user edits the keyboard text field.

```
kb.addEventListener(OPEvent.SAVE, onKeyboardSave, false, 0, true)
kb.addEventListener(OPEvent.BACK, onKeyboardBack, false, 0, true)
kb.addEventListener(OPEvent.CHANGE, onKeyboardChange, true, 0, true)
```

```
private function onKeyboardSave(e:*):void
{
    var args = e.args;

    //assign the text to the textfield on the stage
    txtOnStage.text = args.rtnTxt;

    //unload the keyboard
    onKeyboardBack(e);
}

private function onKeyboardBack(e:*):void
{
    //unload the keyboard
    var kb:* = getChildByName("keyboard");
    removeChild(kb);
    kb = null;
}
```

The example above uses the wildcard (*) type for the keyboard parameter. This is done to avoid runtime definition conflict errors associated with surfacing events of the same class

type from different application domains. You should always use the wildcard definition for framework component event handlers.

Alert window

The alert component displays a text pop-up message. Similar to the keyboard and all basic framework components the definition is located in the “components” shared library and may be obtained locally through the `OPLink.classDefinition` API. The alert component dispatches an `OPEvent.BACK` event when the “OK” button is pressed. The following example implements the set of functions necessary to load and unload an alert window:

```
private function launchAlert(e:MouseEvent):Void
{
    trace("OPEXAlert::launchAlert");

    var alertInfo = new Object();
    alertInfo.rtnCmd = 1;
    alertInfo.msg = "You have opened an Alert window.";
    alertInfo.closeBtnTxt = "Close";

    var OPAlert:Class = OPLink.classDefinition("components", "OPAlert") as
Class;
    var alert:* = new OPAlert(alertInfo);
    alert.addEventListener(OPEvent.BACK, onAlertClose, false, 0, true);
    alert.name = "alert";
    addChild(alert);
}

private function onAlertClose(e:*) :void
{
    trace("OPEXAlert::onAlertClose");
    var args = e.args;

    for(var nm:String in args)
        trace("OPEXAlert::onAlertClose::args."+nm+"="+args[nm]);

    var confirm = getChildByName("alert");
    removeChild(confirm);
    confirm = null;
}
```

Option List

The Option List component provides a uniform way for users to select an item from a pop-up list. An Object containing the following attributes must be passed into the constructor of the `OPOptionList` class.

- `tle`: String – Header text for the pop up.
- `lst`: Array – A list of objects.
- `displayPath`:String – Name of the attribute to display in the list. The default is “lbl”.
- `rtnCmd`:uint – Numeric indicator to be returned with the SAVE and BACK event.

The options component will dispatch the `OPEvent.SAVE` event when a user presses an item in the list. The `OPEvent.args` attribute will contain an Object with the following attributes:

- `rtnCmd`:unit – The numeric indicator passed to the initialize method.
- `itm`:Object – The data Object associated with the selected item from the list.

The options component will dispatch the `OPEvent.BACK` event when the “Back” button is pressed. The component should be programmatically removed as deemed appropriate by your application.

The following block of code implements the necessary methods to load and handle an Option List component:

```
private function launchOptions(args:Object):void
{
    //create the options argument object
    var args = new Object();
    args.tle = "Options Example";
    args.rtnCmd = 1;
    args.lst = [{lbl:"item 1", val:1} , {lbl:"Item 2", val:2}];

    var OPOptionList:Class = OPLink.classDefinition("components",
"OPOptionList") as Class;
    var optionList:* = new OPOptionList (args);
    optionList.name = "optionlist";
    optionList.addEventListener(OPEvent.BACK, onAlertClose, false, 0, true);
    optionList.addEventListener(OPEvent.BACK, optionsBack, false, 0, true);
    optionList.addEventListener(OPEvent.SAVE, optionsSave, false, 0, true);
    addChild(optionList);
}

private function optionsSave(e:*) :void
{
    //handle selection here
    var itm:* = e.args.itm

    for(var nm:String in itm)
        trace("itm." + nm + " = " + itm[nm].toString() );

    //remove the component
    optionsBack(e);
}
```

```
private function optionsBack(e:*):void
{
    var optionList = getChildByName("optionlist");
    removeChild(optionList);
    optionList = null;
}
```

Confirm Window

The confirm window allows you to present a question and solicit a yes/no response from the user. An Object with the following attributes must be passed into the constructor of the OPConfirm Class definition.

- `msg:String` – Question to display.
- `noBtnTxt:String` – Text to display on the left button, will produce a *false* response.
- `yesBtnTxt:String` – Text to display on the right button, will produce a *true* response.
- `rtnCmd:uint` – Numeric indicator to be returned with CONFIRM event.

The confirm component will dispatch only one event: `OPEvent.CONFIRM`. The `args` parameter will contain an object with the following attributes:

- `confirmed:Boolean` – Indicates the users response
- `rtnCmd:uint` – Numeric indicator from the initialize method.

The following block of code illustrates the necessary methods to instantiate, display, and process events from a confirm component pop up:

```
private function launchConfirm(e:MouseEvent):void
{
    trace("OPEXConfirm::launchConfirm")

    var params:Object = new Object();
    params.rtnCmd = 1;
    params.msg = "Are you sure you want to close this application?";
    params.noBtnTxt = "No";
    params.yesBtnTxt = "Yes";

    var OPConfirm:Class = OPLink.classDefinition("components", "OPConfirm") as
Class;
    var confirm:* = new OPConfirm(params);
    confirm.addEventListener(OPEvent.CONFIRM, onConfirmClose, false, 0, true);
    confirm.name = "confirm";
    addChild(confirm);
}

private function onConfirmClose(e:*):void
{
    trace("OPEXConfirm::onConfirmClose");
}
```

```

var args:* = e.args;
var confirm:* = getChildByName("confirm");
removeChild(confirm);
confirm = null;

if(args.confirmed)
    OPLink.closeApplication();
}

```

WaitAnimation

The components library includes a standard wait animation for using in applications. The OPWaitAnimation class constructor supports the following parameters:

- fullscreen:Boolean – If true, a translucent background will prevent key presses to controls below the spinner. Default value is false.
- x:Number – x coordinate of the spinner
- y:Number – y coordinate of the spinner

The OPWaitAnimation constructor takes the above arguments as individual parameters as in the following declaration statement:

```

public function OPWaitAnimation(fullscreen:Boolean = false, x:Number = 387,
y:Number = 190)

```

The following block of code will instantiate an OPWaitAnimation class and place it on the stage:

```

var WaitAnimation:Class = OPLink.ClassDefinition("components",
"OPWaitAnimation") as class;
var spinner:* = new WaitAnimation(true) // returns an 800x410 animation
spinner.name = "spinner";
addChild(spinner);

```

Date Entry Window

TBA

Time Entry Window

TBA

Sound

OpenPeak provides an alternative to standard Adobe Sound support. This alternative is provided to address Adobe issues FP-2250 and FP-4320, which could result in corrupted audio over time.

To use the OpenPeak methods of producing sound you will need to use the OPSoundManager class. This class is pre-loaded during the startup of the framework. To access the manager use the OPLink.classDefinition method as follows:

```
private var manager:*;  
private var soundObj:*;  
  
try {  
    var temp:Class = OPLink.classDefinition("media", "OPSoundManager") as Class;  
    manager = Object(temp).getInstance();  
}catch (e:Error) {  
    OPTrace.debug("Failed to get OPSound manager:" + e.errorID);  
}
```

Once you've executed the *Object(temp).getInstance()* method you will have a reference to the OPSoundManager. OPSoundManager controls all OPSound objects which play sound. The next step is to allocate an OPSound object from the manager:

```
// get a new sound object  
try {  
    soundObj = manager.getSoundObject();  
}catch (e:Error) {  
    OPTrace.debug("Failed to get sound object:" + e.errorID);  
}
```

The sound manager manages a finite number of OPSound objects. **There is no guarantee that your call to manager.getSoundObject will return an object.** If an available OPSound Object is not available a null value is returned. You should always check for a null value and take the appropriate action. Currently, only seven application space OPSound objects are available.

Playing Sounds

The OPSound object currently supports the mp3 format. You can specify mp3 files found locally on the frame or remotely via the Internet. Please note, the OPSound object is not intended to stream radio broadcast or other sound sources which are not a simple mp3 format. There are two steps involved in playing sound: 1. Load the sound file and 2. Execute the start method.

To load and play the sound file use the load and start methods:

```
soundObj.load("/media/mymp3file.mp3");  
soundObj.start();
```

Playback of your mp3 sound starts after the start method is executed. To stop playback use the *stop()* method.

The volume can be controlled via the *set_volume()* method. This method receives a value from 0 to 100. Example:

```
soundObj.set_volume(23);
```

Sound looping can be accomplished by setting the loop property to "true". Sound looping will cause the sound to OPSound object to begin playback automatically when a sound complete event occurs.

```
soundObj.loop = true;
```

OPSound Events

The OPSound object dispatches several events during and after the playback process has begun. You can listen for the following events from the OPSound object:

Available Events:

- OPSoundEvents.PLAY_START – dispatched when the mp3 file has begun playing.
- OPSoundEvents..PLAY_STOP – dispatched when playback of your mp3 has been stopped.
- OPSoundEvents..PLAY_ERROR – dispatched when a playback error has occurred.

- OPSoundEvents.PLAY_COMPLETE – dispatched when an mp3 has completed playing.

Example:

```
soundObj.addListener(OPSoundEvents.PLAY_START,onPlayStart, false, 0, true);
soundObj.addListener(OPSoundEvents.PLAY_STOP,onPlayStop, false, 0, true);
soundObj.addListener(OPSoundEvents.PLAY_ERROR,onPlayError, false, 0, true);
soundObj.addListener(OPSoundEvents.PLAY_COMPLETE,onPlayComplete, false, 0, true);
```

Playback Status

OPSound provides methods for obtaining the playback position and duration of a playing mp3. This information allows you to build UI components which can reflect the playback position of the audio file being played. Use the *get_duration()* method to get the length of the track in seconds and the *get_current_position()* method to determine how many seconds have played in the audio file.

```
soundObj.get_duration();
soundObj.get_current_position()
```

OPSound Termination

All OPSound objects are terminated and reclaimed for reallocation when any application terminates. This normally happens when the user touches the home button in the titlebar. In other case, like when a phone call is received and ends, your OPSound objects will be terminated and reclaimed because the phone app was terminated. To best deal with this situation, we recommend testing the value of your sound object before attempting to access any of its properties and methods.

For example, the following code tests to see if the soundObj object is null and attempts to obtain a new instance if so.

```
if (soundObj == null) {
    // get a ref to the OPSoundManager
    try {
        var temp:Class = OPLink.classDefinition("media", "OPSoundManager") as
        Class;
        manager = Object(temp).getInstance();
    }catch (e:Error) {
        OPTrace.debug("failed to get OPSound singleton:" + e.errorID);
    }
}
```

```
// get a new sound object
try {
    soundObj = manager.getSoundObject();
} catch (e:Error) {
    OPTrace.debug("failed to get sound object:" + e.errorID);
}

if (soundObj == null) {

    lastEvent.text = "[no available channels]";
    positionEventStatus();

} else {

    // add listeners
    try {
        OPTrace.debug("LEE OPSound::ffplayItem adding listeners");
        soundObj.addEventListener(OPSoundEvents.PLAY_START, onPlayStart,
false, 0, true);
        soundObj.addEventListener(OPSoundEvents.PLAY_STOP, onPlayStop,
false, 0, true);
        soundObj.addEventListener(OPSoundEvents.PLAY_ERROR, onPlayError,
false, 0, true);
        soundObj.addEventListener(OPSoundEvents.PLAY_COMPLETE,
onPlayComplete, false, 0, true);

    } catch (e:Error) {
        OPTrace.debug("failed to get sound object:" + e.errorID);
    }

}

}
```

Advanced Components

Unlike the basic components that are loaded and managed locally by your application, the advanced components are accessed directly from the framework through static methods in the `OPLink` Class. These components are tightly integrated with application framework and function as standalone `OpenFrame` applications.

Movie Player

The Adobe Flash player only supports `.flv` based codecs natively for video playback. The Movie Player component will play video files for all media codecs supported by the `OpenFrame`. To load this component call the `OPLink.loadMoviePlayer(args:Object)` from your application with a parameter Object containing the following attributes:

- `lst:Array*` – of the movie information Objects.
- `displayPath:String*` – Name of the attribute in the information Object to use for the URI to the video.
- `filePath:String*` – Name of the attribute in the information Object to use for the title of the video.
- `idx:uint*` – Starting index of the playlist.
- `loop:Boolean` – Whether or not the playlist will loop or stop at the end.
- `randomize:Boolean` – Whether videos are played in order or at random.
- `usb:Boolean` – Indicating Whether or not the media comes from a USB key.

*These are required

Slideshow

The slideshow offers a structured way to present a list of photos on the frame in full screen mode. The slideshow component allows the user to make basic adjustments to the aspect, interval, and transitions. User settings are maintained across applications. To use this component, call the `OPLink.loadSlideShow(args:Object)` method from your application with a parameter Object containing the following attributes:

- `lst:Array*`
 - Array of objects each of which holds information for each photo.
- `filePath:String*`
 - Name of the property of each photo object, which holds URL of the photo.
 - The property must be String type.
 - Default is "path".
- `idx:int*`

- Index of first photo to be displayed in the array list.
- Default is 0.
- `appName:String`
 - If this parameter is provided, user settings for zoom, delay and caption show/hide would be remembered specifically for the app.
 - If this parameter is not provided, user settings would be shared across all the apps and screensaver that do not require separate settings.
- `loop:Boolean`
 - If `true`, first photo will be loaded at the end of the array during slideshow. Slideshow will go on until user stops it.
 - If `false`, slideshow will stop at the end of the array.
 - Default is `true`.
- `randomize:Boolean`
 - If `true`, photos in the array will be shuffled.
 - If `false`, photos will be in original sequence.
 - Default is `false`.
- `delay:int`
 - Delay between each photo during slideshow
 - Value is in milliseconds.
 - User can change this setting. Previously user selected setting will be used ignoring value provided in parameter.
 - Default is 3000.
- `zoom:int`
 - 1 for ZOOM. Small photo will be enlarged and large photos will be reduced to fit with screen width or height. Aspect ratio won't be changed.
 - 2 for FIT. Large photos will be reduced to fit with screen width or height. Aspect ratio won't be changed. Small photos will be modified.
 - 3 for STRETCH. Photo's width will be the same as screen width and photo's height the same as screen.
 - User can change this setting. Previously user selected setting will be used ignoring value provided in parameter.
 - Default is 1.
- `autostartSlideshow:Boolean`
 - If `true`, slideshow will play automatically once loaded.
 - If `false`, photo at `idx` will be loaded, and user needs to start slideshow manually.
 - Default is `false`.
- `enableCaption:Boolean`
 - If `true`, "Caption" button and caption function will be enabled.
 - If `false`, "Caption" button and caption function will be disabled.
 - Default is `"false"`.
- `captionPath:String`
 - Name of the property of each photo object, which holds caption text.
 - The property must be String type.

- Default is "caption".
- `autostartCaption: Boolean`
 - If `true`, captions will be displayed when slideshow is loaded. `enableCaption` must be `true` to get this working.
 - If `false`, captions will not be displayed when slideshow is loaded. If `enableCaption` is `true`, user can make captions visible by pressing "Caption" button.
 - User can change this setting. Previously user selected setting will be used ignoring value provided in parameter.
 - Default is `true`

*These are required

Volume Controller

The volume controller provides a slider to adjust the master volume for Flash and our external media player. Using this component will ensure that the following settings persist beyond a reboot of the OpenFrame. The Volume Controller requires no incoming parameters and can be loaded by calling the `OPLink.loadVolumeContols()` method.

Storing Application Information

Some applications will need to store persistent data in order to function properly or improve the user experience. The OpenPeak framework offers two methods for storing data on the OpenFrame. These methods are beyond the scope of the Adobe FlashPlayer and will only execute correctly on the OpenFrame hardware.

Note, a common directory is provided by the OpenPeak framework. This common directory is used for maintaining content across system updates and as well, provides for sharing information between applications.

Cookies

OpenFrame Cookies are very similar to SharedObjects. Native Adobe Flash data types are appended to the `Cookie.data` property and written to the file system when the `Cookie.flush()` method is executed. To create or load a cookie from within the framework, call either the `OPLink.getLocal()` or `OPLink.getGlobal()` methods, which are defined as follows:

```
OPLink.getLocal(id:String = "localCookie", enc:String = "");  
OPLink.getGlobal(id:String = "localCookie", enc:String = "");
```

The “id” parameter will dictate the name of the cookie to be read or created. Cookies are saved in the appropriate application directory and destroyed when an application is removed. The “enc” parameter serves as the encryption key. A longer key will yield a stronger level on encryption. An empty (zero length) string will yield no encryption. Both parameters are optional. For example, the assignment statement:

```
var mycookie = OPLink.getLocal();
```

will read or create an unencrypted cookie named “localCookie” in the installation directory of a given application. Applications using encryption will need to ensure the encryption string is well hidden within their ActionScript code. Cookies created with the `OPLink.getGlobal()` API will be accessible by all applications while cookies created with the `OPLink.getLocal` API will be accessible only by the application that created it.

Text Files

Some applications may want to store persistent unencrypted information in a standard text file. The following methods will allow you to create and destroy plain text files in the installation directory of your application from within the framework:

```
OPLink.write(file:String, data:String, overwrite:Boolean = true);
OPLink.remove(file:String, overwrite:Boolean = true);
```

The “file” parameter is the full name of the text file (i.e. “info”, “info.txt”, “info.xml”, etc). The “Data” parameter is the String that will be written to the file. The entire file is overwritten each time OPLink.write is executed. To prevent undesired overwriting or removal, set the “overwrite” parameter to false. To read a file that was created with the OPLink.write API use the native URLLoader class with the following URLRequest string:

```
OPLink.applicationDirectory + <file name>
```

For example to load a file “info.xml” saved in your application directory use the following block of code:

```
var ldr:URLLoader = new URLLoader();
ldr.load(new URLRequest(OPLink.applicationDirectory + “info.xml”));
```

The following method will allow you to create plain text files in the common directory within the framework:

```
OPLink.writeGlobal(file:String, data:String, overwrite:Boolean = true);
```

For example to load a file “info.xml” saved in your common directory use the following block of code:

```
var ldr:URLLoader = new URLLoader();
ldr.load(new URLRequest(OPLink.commonDirectory + “info.xml”));
```

Files created with the OPLink.writeGlobal() API will be maintained across system updates and as well, be accessible by all applications while files created with the OPLink.write API will not be maintained with a system updated and will only be accessible to the application that created it.

Shared Objects

Adobe Flash offers its native Shared Object Class to save persistent data on a file system. Applications should avoid using Shared Objects for security concerns and the possibility of collision errors. Instead, application developers should use one of the proprietary methods described in the previous sections.

The Asynchronous Events Model

Some applications will need to maintain a persistent presence in the background to listen for asynchronous events from one or more services and display appropriate content to the user. For instance, a telephony application may need to present an incoming call event or an alarm application may need to present a pop up notification to the user at a predetermined interval.

Establishing Background Processes

The framework establishes and maintains a separate application domain (an Adobe Flash class definition container) at startup for each application. These application domains persist for the life of the framework such that all class definitions and static members are persevered even if an applications main.swf movie has been unloaded. At any time, an application may add data or a process (i.e., Socket Connection, Polling Timer instance, etc.) to a static class member or singleton definition in its local application domain. In order to initiate a background process at startup an application will need to create a document class for its Icon movie and add all registration code into the respective constructor.

Handling Asynchronous Events from Background Processes

Once an asynchronous event is dispatched to your background listener, you will likely want to display some sort of notification on your screen. To load your application from your icon movie or background process, call the static method:

```
OPLink.loadApplication(file:String, args:* = null)
```

The “file” parameter is the name of the movie to load from the applications installation directory. It is not a complete path. The “args” parameter can be of any data type and will be passed to the application in its “onInitialLoad” handler, which is called after the document-level constructor is executed. The following example defines a document class that uses the onInitialLoad handler to process an asynchronous event:

```
package
{
    import op.framework.OpenFrameApplication;

    public class eventExample extends OpenFrameApplication
    {
        public function eventExample()
        {
```

```

        /*
         * Do nothing in the constructor since this application
         * may handle asynchronous events
         */
    }

    public override function onInitialLoad(args:* = null)
    {
        if (args.async)
            //load display popup window
        else
            //start application as normal
        }
    }
}

```

The framework will only recognize one instance of the `OpenFrameApplication` class in a given application domain. If your application is actively running in the foreground, `OPLink.loadApplication()` should not be called and the asynchronous event should be handled within your application. To access your application from a background processes, use the read-only `OPLink.application` property. If your application is not loaded, the `OPLink.application` property will be set to null. If you call `OPLink.loadApplication` while your application is currently on the display stack it will surface the application to the top of the stack and attempt to call the `onSurface()` method in your applications document class.

Handling Focus Changes

Processing an asynchronous event may cause an application that is currently running in the foreground to lose focus. Some applications will need to perform appropriate actions when focus changes occur. For example, a video game may want pause and/or save game information or video application may want to save position information. To handle these situations the following callback methods should added to your applications document level class definition:

```

public override function onFocusOut()
{
    //surfaced when the application loses focus
}

public override function onFocusIn()
{
    //surfaced when the application regains focus
}

```

Since these methods are defined in the `op.framework.OpenFrameApplication` class that your application must extend, you will need to override the methods as shown in the example above.

Telephony

Telephony support is accessible via the `op.extensions.phone.CallControlAS3` class. This class exposes the following set of APIs. These APIs enable the creation of a multi-provider, multi-account, and multi-line user interface.

```
function initialize();
function setDefaultLine(line:CallControlLine);
function offhook(line:CallControlLine);
function onhook(line:CallControlLine);
function digit(line:CallControlLine, digit:String);
function dial(line:CallControlLine, digits:String);
function redial(line:CallControlLine);
function click2dial(digits:String);
function ignore(line:CallControlLine);
function hold(line:CallControlLine);
function resume(line:CallControlLine);
function atransfer(line:CallControlLine, targetLine:Number);
function utransfer(line:CallControlLine, teleNumber:String);
function conference(line1:CallControlLine, line2:CallControlLine);
function message(line:CallControlLine, msg:String);
function volume(level:Number);
function mute();
```

The following asynchronous events will be surfaced accordingly. The class definitions can be found at `op.extensions.phone.events`.

- `CallControlErrorEvent`
- `ConfigurationErrorEvent`
- `ConfigurationEvent`
- `ConnectedEvent`
- `DialPlanErrorEvent`
- `DisconnectedEvent`
- `HookEvent`
- `IncomingCallEvent`
- `InformationalEvent`
- `MediaCapabilitiesEvent`
- `MessageEvent`
- `NetworkErrorEvent`
- `RegistrationEvent`
- `ReminderEvent`
- `RemoteAlertingEvent`
- `VoiceMailEvent`

Telephony Examples

Click2Dial

The following snippet illustrates how to enable an application with click-to-dial capabilities.

```
function click2dial(dialString:String):void
{
    private var callControl:* = null;
    private var CC_LIB:*;
    private var dialString:String;
    private var CallControl:Class = OPLink.classDefinition("callcontrol",
"libCallControl") as Class;

    try {
        CC_LIB = new CallControl();
        callControl = CC_LIB.getInstance(CallControl.CC_RESIDENTIAL);
        callControl.click2dial(dialString, true); // true = launch phone dialer
    }
    catch(e:Error)
    {
        trace("Failed to instantiate CallControlAS3.");
    }
}
```

VoiceMail Notification

The following example illustrates how to gain access to the telephony component and register for asynchronous events.

```
package
{
    import flash.display.*;
    import flash.events.*;
    import flash.text.TextField;
    import flash.net.*;
    import op.framework.*;
    import op.extensions.phone.*;
    import op.extensions.phone.events.*;

    public class PhoneExample extends MovieClip
    {
        private var callControl:* = null;
        private var CC_LIB:*;
        private var CallControl:Class = OPLink.classDefinition("callcontrol",
"libCallControl") as Class;

        public function PhoneExample ()
        {
            try {
```


Testing Applications

Cross-platform support is one of the key advantages of developing with Adobe Flash. There are no special rules, publish settings, or compilation requirement for the OpenFrame. The majority of your development and test effort can comfortably take place on a PC. However, you will ultimately want to thoroughly test the application on the OpenFrame to ensure all aspects of your application are usable and accessible.

Testing on a PC

A complete executable version of the framework can be located in the “published” folder located in the root directory of the SDK distribution. The following must be done to run your application in the OpenFrame framework:

1. Create a directory for your application under “published/apps/”.
2. Place your icon movie, application movie, and language.xml file into your newly created application directory.
3. Create an entry for your application in the applications.xml file located in the published directory. This entry must adhere to the following format:

```
<app static="1" loc="<application directory>" icon="<icon movie>" app="<application movie>" />
```

The applications.xml file will already have several entries that you can use as examples.

4. Run openframe.swf. *

*If this is the first time you are running the OpenFrame Framework in the standalone Flash player you may need to add an exception to the Flash Global Security Panel:

1. Launch the standalone Flash player
2. Right click on the player and choose “settings”.
3. Go to the “Privacy” tab and choose “advanced”.
4. Choose “Global Security Panel” from the left menu on the web page
5. Add the complete path to the root directory of the SDK to your list of security exceptions.
6. Close your web browser and run openframe.swf.

The standalone Flash player will not display run-time errors or trace statements. For the sake of development you should install the debug version of the Flash player available at www.adobe.com. This will present alert windows when runtime errors are not properly

handled. Additionally, you can run `openframe.swf` from within the Adobe Flash IDE to see all of your trace statements and debug output.

Testing on an OpenFrame

See Release Notes

Appendixes

Flex Development

Flex 2.0 applications or applications published in Adobe FlexBuilder or the OpenSource Flex SDK are not supported by the OpenPeak framework.

The config.xml file

The core configuration file for the framework, config.xml, is located in root directory of the published application framework (the “published directory” in the Windows SDK distribution). The file contains basic information required to run the framework including the core component paths, default settings, available languages, etc. This is explained in detail in the following breakdown:

Core Component Paths

<splash>, <titlebar>, <mainmenu>, <language_xml>, and <apps>, etc define the URIs to core files representative of the tags descriptive name. In general editing or removing these sections will break the framework and should not be done.

Flash Libraries

The <flash_libraries> section defines all of the runtime shared ActionScript libraries that are accessible via the OPLink.classDefinition() API. The components library was described in the Basic Components section earlier in this document. The remaining libraries distributed with the SDK are explained in proceeding appendixes.

Default Settings

The <default> section contains the default session information for provider and regional specific configurations. These setting can be edited under the following guidelines:

- <dateFormat> – “na” or “eu” for MM.DD.YYYY and DD.MM.YYYY respectively.
- <timeFormat> – “tm12” or “tm24” for a 12- or 24-hour clock, respectively.
- <region> – default region code used by applications, if necessary.
- <language> – default language value must match an available language as defined in the proceeding <languages> section.

Available Languages

The <languages> section must contain a series of language identifier nodes to specify the available languages that users may select in the *Settings* application. Each node must contain attributes for “lbl” and “val” to specify the regional display string and identifier respectively. The “val” attribute must correspond to a valid identifier in the `language.xml` files throughout the framework and applications.

The `op.utils` package

The `op.utils` package contains a set of utility classes designed to extend standard flash definitions and override the base classes of library assets. This section intends to provide a basic description of the definitions included in the `op.utils` package. The source code for these classes is included in the SDK distribution package in case a more detailed understanding is necessary.

To use the base classes in this package you will need to first create a `MovieClip` in your application’s library that contains the required assets described in one of the definitions below. Then you will need to export the `MovieClip` and replace the standard `MovieClip` base class tag with a reference to the desired base class in the `op.utils` package. For examples please reference the libraries in the sample applications included in the SDK distribution package.

OPBtn

Description: A base class definition for a library `MovieClip` that implements a button with a dynamic label. This class assumes that the following assets are sitting on the stage at frame 1 of the asset’s local timeline:

- `TextField`, `id = lbl_txt`
- `SimpleButton`, `id = btn`

Properties:

- `lbl:String` – Display Text
- `enabled:Boolean` – Setting to `true` will disabled mouse events and reduce the alpha value to 0.25

OPBtnI

Description: A base class definition for a library MovieClip that implements a button with a dynamic label that alternates black and white colors on mouse events. This class assumes that the following assets are sitting on the stage at frame 1 of the asset's local timeline:

- TextField, id = lbl_txt
- SimpleButton, id = btn

Properties:

- lbl:String – Display Text
- enabled:Boolean – Setting to true will disable mouse events and reduce the alpha value to 0.25

OPBtnMM

Description: A base class definition for the SimpleButton asset that represents the background of an application's main menu icon. This class has no methods and only serves to maintain a consistent filter treatment across the backgrounds of main menu icons.

OPBtnToggle

Description: A base class definition for a library MovieClip that implements a button with two states. This class assumes the library asset has key frames at frame 1 and 2 of the asset's local timeline representing the inactive and active states respectively.

Properties:

- enabled:Boolean – Setting to true will disable mouse events and reduce the alpha value to 0.25
- isOn:Boolean – Toggles the active and inactive states

OPBtnToggleWTI

Description: A base class definition for a library MovieClip that implements a button with two states and a dynamic label with multiple color states. This class assumes the library asset has key frames at frame 1 and 2 of the asset's local timeline representing the inac-

tive and active states respectively. Additionally the asset must contain the following on a layer that spans the state key frames:

- TextField, id = lbl_txt

Properties:

- lbl:String – Display Text
- enabled:Boolean – Setting to true will disabled mouse events and reduce the alpha value to 0.25
- isOn:Boolean – Toggles the active and inactive states

Methods:

- setColors(...colors) – sets the text color state according to the following rules:
 - 1 color – all four states are the same color
 - 2 colors – up states are the first color, down states are the second color
 - 3 colors – active up state is the first color, active down state is second color, inactive up and down states are the third color.
 - 4 colors – active up state is the first color, active down state is the second color, inactive up state is the third color, inactive down state is the fourth color.

OPCellCheckBox

Description: A base class definition for a library MovieClip that implements a custom cell renderer with an instance of OPBtnToggle for a flash List component using the ICellRenderer implementation. This class assumes that the following assets are sitting on the stage at frame 1 of the asset's local timeline:

- TextField, id = lbl_txt
- SimpleButtons, id = btn
- MovieClip, id = selected_mc
- MovieClip linked to OPBtnToggle, id = check_btn

OPCellOption

Description: A base class definition for a library MovieClip that implements a custom cell renderer for a flash List component using the ICellRenderer implementation. This class assumes that the following assets are sitting on the stage at frame 1 of the asset's local timeline:

- TextField, id = lbl_txt
- SimpleButtons, id = btn
- MovieClip, id = selected_mc
- MovieClip linked to OPBtnToggle, id = check_btn

The following styles can be set using the `setRendererStyle` method on the List component instance:

- `displayPath:String` – defines the object attribute name that will be used to set `lbl_txt`, the default value is “lbl”
- `showSelection:Boolean` – set to `false` to prevent the selected state from rendering; the default value is `true`

OPEvent

Description: Extends the `flash.events.Event` class in order to carry an argument.

OPList

Description: Subclass of the List component designed to enforce a common set of styles. This class assumes the following assets are in the application’s library:

- Flash List component
- A set of MovieClips that define the scrollbar skin. These must be exported for use in ActionScript as `OPSBTrack`, `OPSBThumb`, `OPSBThumbIcon`, `OPSBArrowDown`, `OPSBArrowDownH`, `OPSBArrowUp`, and `OPSBArrowUpH`

OPLoader

Description: Extends the `flash.display.Loader` class in order to carry an argument.

OPScrollPane

Description: Subclass of the ScrollPane component designed to enforce a common set of styles. This class assumes the following assets are in the application’s library:

- Flash ScrollPane component

- A set of MovieClips that define the scrollbar skin. These must be exported for use in ActionScript as OPSBTrack, OPSBThumb, OPSBThumbIcon OPSBArrowDown, OPSBArrowDownH, OPSBArrowUp, and OPSBArrowUpH

OPWindow

Description: A base class definition for MovieClips that represent pop-up window backgrounds. This class has no methods and only serves to maintain a consistent filter treatment across pop up window backgrounds.

The OPLink Class

The OPLink class contains a set of static attributes and methods that serve as the main bridge between applications and the application framework. To access OPLink in an application use the following import statement:

```
import op.framework.OPLink;
```

The OPLink class defines the following public attributes for use in an application:

application [read-only]

The application attribute returns a pointer to the loader object of the active application in a particular application domain space. If an application is not loaded in the domain space the return value will be null.

Implementation:

```
public static function get application():*
```

applicationDirectory [read-only]

The applicationDirectory attribute returns a string path to the installation directory of the associated application on the OpenFrame hardware.

Implementation:

```
public static function get applicationDirectory():*
```

applicationID [read-only]

The applicationID attribute returns a string identifier for the associated application.

Implementation:

```
public static function get applicationID():*
```

dateFormat [read-write]

The dateFormat attribute returns a string value of “eu” or “na” for the configured date format.

Implementation:

```
public static function set dateFormat(s:*):void
public function get dateFormat():*
```

language [read-write]

Gets or sets the current language across the device. Acceptable values are the ISO 639-1 two-letter language identifiers as defined in config.xml

Implementation:

```
public static function set language(s:*):void
public static function get language():*
```

region [read-only]

The region attribute returns the current region code for the device

Implementation:

```
public static function get region():*
```

timeFormat [read-write]

Gets or sets a string value of “tm12” or “tm24” for the configured time format.

Implementation:

```
public static function set timeFormat(s:*):void
public static function get timeFormat():*
```

The OPLink class defines the following public methods for use in an application:

addScreensaverHold

This method prevents the screensaver from loading while the hold is in place. If the screensaver is currently active it will be dismissed. An application must call **removeScreensaverHold** when it is unloaded.

Implementation:

```
public static function addScreensaverHold():void
```

classDefinition

Retrieves a class definition from a shared library

Params:

- lib:String – Name of the shared library as defined in config.xml
- className – Full path to the class definition in the library

Implementation:

```
public static function classDefinition(lib:String, className:String):Class
```

getLocal

Returns and/or creates an openpeak cookie in the associated applications installation directory.

Params:

- id:String – Name of the cookie to be created
- enc:String – Encryption string to the cookie

Implementation:

```
public static function getLocal(id:String = "localCookie", enc = ""):*
```

getGlobal

Returns and/or creates an openpeak cookie in the common data directory on the Open-Frame. All applications will have access to the cookies created with this method.

Params:

- id:String – Name of the cookie to be created
- enc:String – Encryption string to the cookie

Implementation:

```
public static function getGlobal(id:String, enc = ""):*
```

loadApplication

This method loads the specified application into the associated application domain. If an application is already loaded in the domain it will be surfaced to the top of the display stack.

Params:

- file:String – file name of the application .swf file in the applications installations directory
- args:* – optional arguments to pass to the loaded application via the onInitialLoad handler

Implementation:

```
public static function loadApplication(file:String, args:* = null):void
```

loadMoviePlayer

The loadMoviePlayer method loads the movie player component. See the advanced components section for more details

Params:

- args:* – argument set described in the advanced components section.

Implementation:

```
public static function loadMoviePlayer(args:*)void;
```

loadSlideShow

The loadSlideShow method loads the photo slideshow component. See the advanced components section for more details

Params:

- args:* – argument set described in the advanced components section.

Implementation:

```
public static function loadSlideshow(args:*)void;
```

loadVolumeControls

The loadVolumeControls method loads the volume control component. See the advanced components section for more details

Implementation:

```
public static function loadVolumeControls():void;
```

removeScreensaverHold

The removeScreensaver method removes an application hold on the screensaver.

Implementation:

```
public static function removeScreensaverHold():void
```

write

Writes a text file to the installation directory of an associated application

Params:

- file:String – Name of the file to write
- data:String – Text data to write
- overwrite:Boolean – Boolean value to restrict overwriting

Implementation:

```
public static function write(file:String, data:String, overwrite:Boolean = true):void
```

remove

The remove method removes a text file from the installation directory of an associated application

Params:

- file:String – Name of the file to write
- overwrite:Boolean – Boolean value to restrict overwriting

Implementation:

```
public static function remove(file:String, overwrite:Boolean = true):void
```

Shared Libraries

In addition to the framework features available via the static OPLink class the SDK includes a series of runtime shared libraries that provide universal access to system information and utilities via the OPLink.classDefinition() API. As previously described in this document all basic components are currently accessed through this API as follows:

```
var ClassName:Class = OPLink.classDefinition(<library name>, <full class path>);
```

For example to access and instantiate the OPKeyboard component use the following block of code:

```
var OPKeyboard:Class = OPLink.classDefinition("components", "OPKeyboard") as
Class;
var keyboard:* = new OPKeyboard({type:"ABC", txt:"", rtnCmd:0});
```

Notice that the keyboard instance is not strictly typed (*). Currently the framework requires this to avoid Application Domain conflicts.

The components library is described in detail in the "Basic components" section of this document. The remaining libraries included in the SDK distribution are described below:

system

The "system" library contains one class, SystemUtils, which presents read-only access to most hardware settings.

SystemUtils

Use the following block of code to access and instantiate the SystemUtils class:

```
var SystemUtils:Class = OPLink.classDefinition("system", "SystemUtils") as
Class;
var utils:* = new SystemUtils();
```

The SystemUtils class defines the following attributes:

brightness [read-write]:

uint value from 0 to 100 for the current screen brightness.

Implementation:

```
public function get brightness():uint  
public function set brightness(val:uint):void
```

ESSID [read-only]:

Returns an Object with the following attributes pertaining to the current ESSID:

- Name:String – Current network SSID Name
- Signal:Int – Integer value for the signal strength of the configured wireless network
- Encrypted:Boolean – Boolean value for whether or not the configured network is encrypted
- Connected:Boolean – Boolean value for whether or not the device is connected to the Network

Implementation:

```
public function get ESSID():Object
```

gateway [read-only]:

String value representing active gateway address.

Implementation:

```
public function get gateway():String
```

ipAddr [read-only]:

String value representing the active IP address.

Implementation:

```
public function get ipAddr():String
```

macAddr [read-only]:

String value representing the assigned MAC address of an OpenFrame.

Implementation:

```
public function get macAddr():String
```

primaryDNS [read-only]:

String value representing the primary DNS server address as configured on the Open-Frame.

Implementation:

```
public function get primaryDNS():String
```

primaryNTP [read-only]:

String value representing the current primary NTP server address as configured on the OpenFrame.

Implementation:

```
public function get primaryNTP():String
```

secondaryDNS [read-only]:

String value representing the secondary DNS server address as configured on the Open-Frame.

Implementation:

```
public function get secondaryDNS():String
```

secondaryNTP [read-only]:

String value representing the secondary NTP server address as configured on the Open-Frame.

Implementation:

```
public function get secondaryNTP():String
```

subnet [read-only]:

String value representing the active subnet address.

Implementation:

```
public function get subnet():String
```

timezone [read-only]:

String value representing the configured time zone.

Implementation:

```
public function get timezon():String
```

UID [read-only]:

String value representing the unique identifier for a given OpenFrame device.

Implementation:

```
public function get macAddr():String
```

The SystemUtils class defines the following methods:

log:

log application statistics to the centralize proprietor database.

Params:

- id:String – log identify for the application. In most cases this will be OP-Link.applicationID.
- actions:String – activity action identifier
- desc:String – optional description of log entry

Implementation:

```
public function log(id:String, action:String, desc:String = ""):void
```

The fl package

The third party fl package adds the following useful styles to the native Flash scroll bar component:

```
setStyle("scrollBarWidth", <size in pixels>);  
setStyle("scrollArrowHeight", <size in pixels>);
```

Without this package, the scrollbar will automatically be resized to the width of the default native component skin. This package is physically located in the root directory of the SDK distribution.

Assets fla

The assets fla file located in the root directory of the SDK distribution contains a set of core graphics representing the basic look and feel of the OpenPeak user interface. This file is not intended to designate a complete set graphics and/or assets required by your application but to provide a set of graphical building blocks. Some of the library assets in this file (i.e. buttons, toggles, windows, scrollbar skins, etc.) have been linked to base classes in the op.utils package to add functionality and runtime formatting.

© 2009-2010 OpenPeak Inc.

This document constitutes and contains information that is proprietary to OpenPeak Inc. and may be considered confidential. This document may not be reproduced or distributed to any party without the prior written consent of OpenPeak Inc.

Possession of this document does not constitute or effect, either expressly or impliedly, a transfer, conveyance, or license of any intellectual property rights or other rights in the information disclosed therein. The descriptions contained in this document do not expressly or impliedly grant any licenses to make, use, or sell equipment or software in accordance with such descriptions.

The information contained in this document is provided strictly "AS IS" and is subject to change at any time without notice or obligation by OpenPeak Inc. Neither OpenPeak Inc. nor any party through whom the user obtains this document guarantees that it is accurate or complete or makes any warranties with regard to the results to be obtained from its use.

Under no circumstance is this document to be used or considered as an offer to sell or an offer to buy any security in any state in which such offer to sell or buy would be unlawful prior to registration or qualification under the securities laws of any such state. This document may contain certain "forward-looking statements" about OpenPeak Inc.'s future expectations or other statements concerning other than historical facts. OpenPeak Inc. intends that such forward-looking statements be subject to the safe harbors created under applicable security laws. Since these statements involve risks and uncertainties and are subject to change at any time, actual results could differ materially from expected results.

OpenPeak, OpenFrame, OpenGateway, and ProFrame are registered and unregistered trademarks of OpenPeak Inc. Other trade brands, names, and marks used in this document are the property of their respective owners.