© Artly There Software, January 2007
—

This Developer document describes the v1.0.0 implementation of Externals/Plugins for Artly There Software's Compositor v2.9.3 and higher for OS X and is intended for those wishing to develop an external processing feature for Compositor. If you are a developer working with this spec, and encounter a problem or have a need, contact <artlythere@kagi.com>. Confirmed plugin developers are entitled to a substantial discount for any registration code needed (contact Artly There). And yes, one is aware that this spec may indeed appear to have been written by a team of internet monkeys sitting at word processors. :) It will get better.

Overview of Plugins/Externals and Compositor

Externals or plugins consist of the plugin application and an accompanying .info file in the same folder as the plugin application.

Plugins or Externals as referred to here are small freestanding image processing applications which do a single or multiple image processing task using Apple events to communicate back and forth with Compositor to handle the image transfer and information needs. Any work done to make a Compositor plugin does not limit that application to some use with Compositor alone. Any existing application could be made to work with Compositor as a plugin by just adding the necessary Apple events and data record handling required.

Required: Ability to write a Compositor Plugin/External is to be able to write an image processing filter or application, and know how to send and receive Apple Events. You'll need to know how to get and send PSN (Process Serial Numbers), and how to access and encase data sent over by Apple Events. Examples in Futurebasic are below, and doing same in C or objective C is likely easier.

For ease of development : A single Compositor external can currently own up to 32 menu items for that single plugin as variants, and the plugin will only need to process the particular variant selected when run (the variant

number is passed along with the image to be worked on via Apple event). Thus, one does not need to create multiple plugins if one would rather have a single external application which conditionally filters one way or another. For instance, you can have an invert filter which accepts variant 1, 2 or 3, where 1 may be the usual, 2 would be Green only inversion, or 3 might be red and blue only, and so forth, on up to 32 varieties, or just the default minimum single action. The variant information is listed as outlined in the .info format below. The variant number will be the rightmost 2 digits in the filename of the offloaded temp file (01 to 32), or you can access it from the Plugdata record as well after that is received. Thus, Plugin_WorkFile_12.psd might be sent over, the variant would be 12. For Plugin_WorkFile_03.psd, the variant would be 3. And for a single use plugin, the file might be named Plugin_WorkFile_01.psd.

Structure: Externals or plugins consist of the plugin application and an accompanying .info file, which is a text file outlining the plugin's information. This will likely be set up as a STR# resource approach, or xml approach eventually.

Plugins in Compositor operate differently from say Photoshop, as they are not code resources or bundles loaded up by the parent application. They are instead external applications which accept an offloaded file from Compositor, do the work, then return a file back, optionally with some flags depending on what was changed. They communicate with the Parent Application (Compositor) via Apple Events. This is in part for ease of implementation, and because the developers of Futurebasic never delivered on the promise of such wonders. A future version will soon allow for such, or Compositor might be rewritten one day in objective C, at the next major life depression. (that's what it would take...(cause?)) :)

Parent-Client Interaction:

Compositor and the external communicate via a set of Apple Events. The required events and structure are listed under Required Apple Events further down below.

User Notification during filter operations:

Plugins may remain in the background and utilize the progress screen from the parent application (sending back a 1-100 integer at various points during processing , Compositor updates the view) or they may be brought to the front, as in the case the plugin has some parameters to be set via a dialog, etc. In that case, you must create and update your own progress bar. When complete, bring Compositor to the front, and it will load the result. Such an approach as this may be a bit faster given I/O times for AE progress events.

Your plugin will note if it will be using Compositor's progress dialog or its own in its .info params file. See the note regarding Foreground or background settings.

How Plugins work with Compositor:

Compositor scans the Externals folder on application menu init, gets the current plugin applications available and their info files, then lists their filters (menu item names) according to the .info params in the plugin info file (assuming it was formatted and parsed correctly). Externals are checked for existing locally when asked to run, in case a user moves one after application init. If you add an external to the Externals folder you will want to restart the application so that it appears in the menu.

When the user chooses an External...

A. The filter is selected by the user accessing the Externals menu, including any variant.

B. Next, Compositor writes a file in the system's temporary items folder (in Photoshop .psd format currently as Quicktime can import these easily and quickly), and then using Launch Services (LSOpenFromRefSpec), Compositor launches the plugin either behind, or foreground depending on the .info setting. The open will include the reciept of an FSRef for the file to be loaded by the plugin (which may arrive as an FSSpec) and you can access this via the usual AEOpenDoc processor for your application. After processing the file information you need to send back the 'Aliv' ack containing your plugin's PSN, and then Compositor will immediately ship

back the current image record as given in the plugdata format.

C. The plugin processes the file, and according to Plug.info params, Compositor will either reload the new image, open a new document with it, or nothing, as in the case of an exporter. At the end of processing, a file writing notification AE is sent (if it is doing such) , followed by a return of the plugdata record (noting success or failure and more), then if nothing else, the 'FDun' event is required. This lets Compositor know you are all done. This event must be sent at any time of plugin abort as well.

Compositor offers an AppleEvent fed symbiotic progress bar, or the plugin can use its own. This is set in the params file, see below.

It is all very simple in practice and implementation. Plugin/External developers need only write the usual application, and implement a basic AppleEvent setup to do what is needed. The AE's needed are outlined below.

So in short, User selects plugin, Compositor writes a work file and then launches the plugin with an AEOpenDoc event. The plugin upon successful load of the image, sends an AE 'Aliv' ack, Compositor then sends an image record over which gives the image information and such, the plug does the work, then sends an equivalent record back (via the AE), whereupon Compositor reloads and adjusts to the new reality. Your plugin could also just export a file, and send back nothing, in that case Compositor will desire you send back the FSSpec of the new image for save of the file name into the recent items menu, etc.

The below is the format needed to currently specify an external/plugin's INFO file.

This file should be in the same folder as the app (plugin). For example, if the plugin is Crinkle.app, then the params file Compositor will seek should be alongside as Crinkle.info. The actual menu name you wish to have show up should be the folder name..so if you wished it to be your company, you might called the folder holding the plugin and info file, "Crinklyware"

EXAMPLE PARAMS FILE, to be titled Crinkler.info in the case of Crinkler.app

BEGIN is first line:

```
BEGIN
VERSION
100
TYPE
1
SEEKS
1
RETURNS
1
USEPROGRESS
1
WILLFOREGROUND
0
ABOUT1
Crinkler .... Applies funky textures
ABOUT2
By Winkly Beans Software,  http://www.crinkleywinkly.com
MENU
Crinkler Extreme  // Funky crinkle effect
Crinkler Maximus // Triple Funky crinkle effect
Crinkler Absconded // An image stolen from former glory
Crinkler Inverted  // Upside Down
Crinkler Do The Whammy Jammy // Las Vegas isn't this bad
END
```

The MENU flag must can be have at least 1 item (default can be the app name), but otherwise can be for multiple items, up to 32 as variants.

This enables one application to display multiple filter options in the menu, so one application/external can offer many filters. In the case of the MENU tag, you must always have at least 1, and this is also the name which will be shown in the menu (thus you can tailor your plugin's name in the menu). Compositor sends over the work file with the item number of the menu item appended, and you can know which version of your filter to apply by numerical index, 1 to 32.

When more than one item exists, a hierarchical menu will be created for that particular external.

To add comments about filters: The // denotes a comment to follow, and this line will appear in an about box for that line item (future). Note : the maximum for each line is 255 characters, spaces inclusive, and line wrapping is not allowed as of the current spec. (2 lines for About1 for instance). The About lines are not parsed for comments, thus any http:// link strings will not suffer on info load.

Params/Flags...

EndCaps : Must have BEGIN and END

A missing field ends the parse. Make sure you are thorough. Start from a copy of the template file provided.

The field or item being specified is above, and its value is below.

Spaces are stripped from tags both left and right, and the parser uppercases when checking, so don't worry about case.

Param and Tag details:

BEGIN
Must have begin.

VERSION
is which version of this Compositor external spec you are matching. This is a triplet, 100 means v1.0.0...101 would mean v1.0.1 of the spec you are meeting.

TYPE
is what function you are providing to the program... currently this means filters or other (vs Export, etc) 1 = Filter, 2 = Export/Save/Conversion. An int, or constant. Default is 'filter', so use 1 or 0 if unsure. If your external

alters the pixels of bitmap given, then it's a filter. If it changes the image form to another format, such as from PICT to LunarBomb format, then it's an Export/Save/Conversion type.

SEEKS

tells Compositor what kind of work file to provide your external. PICT, or Photoshop Each of those formats provides alpha channel information as well as the RGB. PICT is currently not supported (can take longer to load), so assume you'll be getting a PHOTOSHOP format image, .psd and return same. Use 1 in this field for now.

RETURNS

tells Compositor what you will be returning. Assumed is the same back (Photoshop, .psd) as 1, kReturnsImage. Set to 0 (kReturnsNothing) if returning Nothing, as with an exporter/converter. Set to 2 (kReturnsNewDocument) if you want the parent to open up a new window with the result. Be mindful that you write any new document in a place where the user can get to it, not to the temporary items folder you picked up the work file from!)

PROGRESS

as True means you will be sending Compositor updates as you filter. (double) Progess as False, means you will provide the user an update notification using your own progress dialog Progress bars need to allow for Canceling of your filter. Such should in general reflect the current UI of Compositor, good or bad as that may be, or just any mac standard. You may use the progress bar PICT resources within the parent program if you wish. Note please the below option for Foreground as regards progress dialogs.

FOREGROUND

True (1) or False (0) means that your app/plug will be switching app layers on top of Compositor and Compositor will launch your plugin in front, rather than keep it in back. If you set the FOREGROUND flag, because you need to set some options or otherwise utilize a full blown interface, then you must utilize your own progress dialogs, so set the progress flag to True. (1)

ABOUT1
is for your software company name, limit of 255 characters.
ABOUT2
is for what your filter does briefly, limit of 255 characters.

Currently, a plugin's about alert is seen if using the option key when choosing an external item. This will be upgraded in the future to something nicer.

This specification will be augmented in prior plugin specs assuming any interest is there and as the hooks are more implemented throughout Compositor. If your app provides its own splash/about box, unless absolutely necessary, do not make the user view it before operating on the work file but as a closing reminder...for instance, if your plugin is a demo and you wish people to register it.

MENU

Means your filter names...use the name you wish to see in the menu, or if you have several variants, the list you'd like to see listed. I.E.:

MENU
Crinkly One
Crinkly Two
Crinkly Three

Or Simply

MENU
New Improved Longer Named Crinkly Thing

The app can be called Joe Diddly...won't matter.

END
Must have the END tag.

OVERVIEW OF INTEGRATION BETWEEN COMPOSITOR AND A PLUGIN/

EXTERNAL

HOW THIS ALL WORKS:

1. When a plugin/external is chosen, Compositor will check to verify the plugin is available (not busy, etc), and that it exists (hasn't been moved since menu build time).

2. Iff all is well, Compositor will offload the current top document or selection as a file in the format the plugin request. The default is Photoshop. PICT will work (in the future) but takes longer to load and create and is limited to 4096 pixels per dimension maximum.

3. Compositor will then launch the plugin/external with an FSRef/FSSpec for the file in question, generally in the background, and your plugin/external should handle an Open Document request from Launch Services or AE's to anticipate this. Now that you have the image, it's your job to do your mojo behind the scenes. The variant is accessed by taking the rightmost 2 characters from the incoming work file's name.

If you are cycling through a loop, and are using Compositor's Progress alert per the param spec above, then your app will be expected to be sending back Apple Events containing an integer 1-100 indicating the current progress every now and then (every half second or so)... This integer will be current progress as a percentage...thus a loop of 15,000 cycles where you will be doing 45,000 total would send back a 33 , or one third completed. This will be displayed on Compositor's progress bar. You need not send back a huge number of progress events, less is more as far as speeding up the process. Just send enough so that the user, or program, isn't left hanging. :)

4. When you are done, write the image the Temporary items folder, or your own scratch location. Once your output is complete send back an Apple Event per the specification below containing the FSSpec of the new image. Quit your app, and Compositor will load up your result upon receipt of notification.

NOTE: Your plugin must write a new image file when outputting its result.

DO NOT overwrite the original work file. It may be needed for restore purposes should the filtering fail or other need in the future (layers, history, etc) , and you may need to reload it during some interim processing.

For v1.0, please limit operations to returning back a same sized (pixels w x pixels h) image. If you crop the image or resize it currently, it will open up as a plugin return document, not as the current image.

* * * Required Apple Events * * * for receiving and sending

The Application creator for Compositor is "CPtr" if for any reason you need to find the process number manually.

Your Plugin must Receive:

AEOpenDoc

Your plugin will be launched, and you will receive an item list (1 file)..as an FSSpec, or however you wish to access it.

'PRec'

Plugin Record event...this will contain a data block with various image information. The record is outlined below. This will be utilized more in a future specification and is subject to change at that time. This block may contain FSSpec's to other images (masks, selections/layers) which your plugin may wish to load and work from.

'Quit'

Compositor is requesting that you quit. *Clean up what you are doing as fast as possible, and* ***Quit.*** The user has either canceled the filter, or quit Compositor. You'll want to check for receipt of this during filtering if the operation takes more than a couple seconds or longer.

Your Plugin must be able to Send:

‘Aliv’

On plugin open after the AEOpenDoc receipt, send back'Aliv' along with the plugin’s PSN (Process serial number). So, after securing your PSN:

Snippet (In FutureBasic)

```
osErr  = fn AECreateAppleEvent(_"CPtr", _”Aliv”, myAEDesc,
_kAutoGenerateReturnID, _kAnyTransactionID, myAEvent )
ignore = fn AEDisposeDesc( myAEDesc )

osErr  = fn AECreateDesc( _"psn ", PlugPSN, sizeof( ProcessSerialNumber
), myAEDesc )
err = Fn AEPutParamDesc( myAEvent, _"PSNf", myAEDesc )
```

'Eror'

report any problem initializing your plugin or loading the image using 'Eror'

```
osErr  = fn AECreateAppleEvent(_"CPtr", _"Eror”, myAEDesc,
_kAutoGenerateReturnID, _kAnyTransactionID, myAEvent )
ignore = fn AEDisposeDesc( myAEDesc )
err = Fn AECreateDesc( _typeLongInteger, @info, Sizeof( long ),
myAEDesc )
err = Fn AEPutParamDesc( myAEvent, _"pInt", myAEDesc )\

Long If err = _noErr
err = Fn AESend( myAEvent, #_nil, _kAENoReply _kAEneverInteract,
0,0,0,0 )
End If
ignore = Fn AEDisposeDesc( myAEvent )
ignore = Fn AEDisposeDesc( myAEDesc )
```

and so on..

'Prog'

while filtering (If using the parent app's progress bar and not your own)...send back a progress integer from 1-100.

```
osErr  = fn AECreateAppleEvent(_"CPtr", _"Prog”, myAEDesc,
_kAutoGenerateReturnID, _kAnyTransactionID, myAEvent )
ignore = fn AEDisposeDesc( myAEDesc )
err = Fn AECreateDesc( _typeLongInteger, @info, Sizeof( long ),
myAEDesc )
err = Fn AEPutParamDesc( myAEvent, _"pInt", myAEDesc )\

Long If err = _noErr
err = Fn AESend( myAEvent, #_nil, _kAENoReply _kAEneverInteract,
0,0,0,0 )
End If
ignore = Fn AEDisposeDesc( myAEvent )
ignore = Fn AEDisposeDesc( myAEDesc )
```

and so on..

Send :

'FWrt'

just before writing the output file

Example, if the filter action was completed, set some variable ‘info’(for instance) to True (1) then:

```
osErr  = fn AECreateDesc( _"psn ", gParentPSN, sizeof(
ProcessSerialNumber ), myAEDesc )
osErr  = fn AECreateAppleEvent(_"CPtr", whichAE, myAEDesc,
_kAutoGenerateReturnID, _kAnyTransactionID, myAEvent )
ignore = fn AEDisposeDesc( myAEDesc )

Select whichAE
case _"Prog" , _"Eror", _"FWrt"

err = Fn AECreateDesc( _typeLongInteger, @info, Sizeof( long ),
```

```
myAEDesc )
err = Fn AEPutParamDesc( myAEvent, _"pInt", myAEDesc )

Long If err = _noErr
err = Fn AESend( myAEvent, #_nil, _kAENoReply _kAEneverInteract,
0,0,0,0 )
End If
ignore = Fn AEDisposeDesc( myAEvent )
ignore = Fn AEDisposeDesc( myAEDesc )
```

Next, after updating any necessary information, send back the image datablock (gPlugData in this case) to Compositor, after -at least- having set plugdata.result to some value. (true (1) if the plugin completed the operation, other if not (canceled via 'Quit' or error)

```
'PRec' // Data outcome (can remain essentially unchanged for v1.0)

osErr  = fn AECreateAppleEvent(_"CPtr" , _"PRec", myAEDesc,
_kAutoGenerateReturnID, _kAnyTransactionID, myAEvent )
ignore = fn AEDisposeDesc( myAEDesc )

osErr  = fn AECreateDesc( _"Stat", gPlugData , sizeof( plugData ),
myAEDesc )
err = Fn AEPutParamDesc( myAEvent, _"Data", myAEDesc )
```

And lastly, ALWAYS Send :

```
'FDun'
```

In this case, info will be True (operation completed) or False (incomplete)

At this point the output FSSpec of the written file is also included for reuptake by Compositor (assuming it exists)

```
case _"FDun"

osErr  = fn AECreateAppleEvent(_"CPtr", _"FDun", myAEDesc,
```

```
_kAutoGenerateReturnID, _kAnyTransactionID, myAEvent )
ignore = fn AEDisposeDesc( myAEDesc )


// Info here is typically gPlugdata.result (1 = Successful, 0 = false, or
aborted)

err = Fn AECreateDesc( _typeLongInteger, @info, Sizeof( long ),
myAEDesc )
err = Fn AEPutParamDesc( myAEvent, _"pInt", myAEDesc )

long if info // Good Return!
ignore = Fn AEDisposeDesc( myAEDesc )
err = Fn AECreateDesc( _typeFSS, @gPlugReturnSpec, Sizeof( FSSpec ),
myAEDesc )
err = Fn AEPutParamDesc( myAEvent, _"PRtn", myAEDesc )
end if
```

THAT'S IT!

:)

For now in the below Plugdata record, worry only about imageOut$, imageIn$, new$,err1$,err2$, iRect, and variant, and in particular, result. As of Compositor v2.9.3, only a subset of this is yet supported. Time and hopefully some developer feedback will help make it clear which is a better specification and approach.

```
CONSTANTS

// Internal to Compositor
begin enum
_FullImage // 0
_PartialImage // 1
_FloatingImage // 2
_FloatingText // 3
end enum
```

```
// Internal to Compositor via Info params file
begin enum
_ReturnsNothing // Export, save modules.
_ReturnsImage
_ReturnsNewDocument
end enum

// Support these

_OperationFailed = 0 nothing gained, or error, or same as initial.
_OperationComplete = 1 True, was completed.

_PICTFormat = 0
_PhotoshopFormat = 1

// _NoErr = 0

begin enum
_NoError // 0 nothing, no error. Cool.
_GW1Error // work buffer
_GW2Error // work buffer
_BadFileError // the work file was not found
_WorkFileLoadError // File found, but could not load the file for
some reason
end enum

_RectError = 8 // Image had a bad rectangle specified (empty, or
invalid).

STRUCTURE OF PLUGIN RECORD DATA BLOCK sent back and forth via AE
on Plugin init, and end (returned back to Compositor)


Begin Record Plugdata

dim version$ // “100” in this case

dim imageOut$ // Filename of image from Parent
dim imageIn$// Filename of image from External if different

// The next are in case some new image is also created alongside
```

```
the prior.
dim new$ // Filename of new image from External
dim mask$ // mask file name...255 max.

dim err1$ // anything you wish to note...255 max.
dim err2$ // anything you wish to note...255 max.

dim iRect as rect // image rect out and back
dim cRect as rect // subrect of current selection in Compositor
(if different)
dim mRect as rect // maskrect same as iRect for mask ; if
different you -must- spec mx,my below
dim nRect as rect // newrect of any new image created (versus
returning a filtered one)

dim variant     as int// filter variant sent, used
dim result      as int// good result?
dim format      as int // format of returned images

dim resized       as int // cropped or scaled..Do not resize with
v1.0, if you do,it will be opened as a new document.
dim hasAlpha       as int // alpha channel exists in work file or
returned file
dim alphaEdit      as int // set _AlphaSame if nothing done, set
_AlphAdded if added, set _AlphaChanged if changed, set
_AlphaDeleted if deleted.

dim depth         as int// usually 24, 32 if Alpha channel added
dim newImage      as int // 1 = new image returned, open in new
window , 2 = The plug exported one...just sending FSSpec. Add to
Recent Items.
dim newMask       as int // Set newMask flag if making new mask
for a selection. Be sure to fill MaskSpec


dim mx          as int // mask offset for sub image sized mask
dim my          as int // mask offset for sub image sized mask

dim res3          as int
dim res4          as int
```

```
dim res5   as long
dim res6   as long
dim res7   as long
dim res8   as long

dim res9   as long
dim res10  as long

dim res11  as double
dim res12  as double
dim res13  as double
dim res14  as double

dim fSpec     as fsSpec // fspec of scratch file out, on Return,
the fsspec of returned file back (optional)
dim newSpec   as fsSpec // Additional file created
dim maskSpec  as fsSpec // Additional mask file created
dim alphaSpec as fsSpec // Additional alpha channel file created

// Future consideration
dim brushTSpec  as FSSpec // Brush tool spec
dim brushMSpec  as FSSpec // Brush mask spec

End Record
```