Chapter4
What is a plug-in?

4.1 Overview

This chapter describes the runtime model for plugin modules: the structure of a plugin file, how plugin modules are located by the Arrange application and loaded into memory, and the environment in which they execute. You should at least skim the entire contents of this chapter, as it provides information which may be useful during debugging. Particular attention should be paid to sections 5.2.3 (toolbox calls) and 5.2.4 (memory management).

4.2 The plug-in operating environment

4.2.1 Arrange startup

When the Arrange application is launched, it tries to load all plug-in modules it can find. This is done by searching for resources of type “MDdf” in the Arrange application file, any files of type “APlg” in the same folder as the Arrange application, and (under System 7) any files of type “APlg” in the Arrange preferences folder. The modules are then loaded according to the following process, which can be considered to operate in parallel for all modules. The effect of this process is that Arrange loads the most recent available version of each module, subject to the module’s system, resource, and inter-module dependency requirements.

1. The module’s resource file is opened.
2. The “MDdf” resource, and any associated “MDdp” resource, is loaded.
3. The system and resource requirements described in the “MDdf” resource are checked. If the current system and application do not meet the requirements, the module is rejected (its load process halts at this point).
4. If a module with the same four-byte ID, and a higher version number, is available, the load process for this module is suspended until the other module is either loaded or rejected. If the other module is loaded, then this module is rejected; if the other module is rejected, then the load process for this module is allowed to proceed.
5. If the module has a “MDdp” resource, the load process is suspended for this module until all of the listed modules have been loaded. If any module listed in the “MDdp” resource cannot be loaded, or the required version is not available,

the module is rejected.

6. If the module's load-at-boot flag (located in the "MDdf" resource) is set, the module's root code resource is loaded into memory, locked, and its main entry point is called with an "initialize" message. The load-at-boot flag should normally be set for all modules except those intended only to be called from other modules (for example, a utilities package shared by several other modules).

4.2.2 Arrange exit

When Arrange exits, it reverses the load process for all loaded modules, by calling their main entry point with an "exit" message and closing their resource fork. If a plugin module has made changes to its resource fork (this is not recommended), it must explicitly call UpdateResFile, either when making the changes or in response to the "exit" message. Arrange will not call UpdateResFile itself before closing the module's resource fork.

4.2.3 Toolbox calls

This section describes the conditions under which plugin modules may make calls to the Macintosh Toolbox and operating system. (Rules for memory allocation are given in the next section.)

As of version 2.0, Arrange requires that System 7.0 (or higher) be installed, so modules can assume the presence of System 7 features such as the Process Manager and Alias Manager. Arrange does not require a 68020 processor or Color QuickDraw, so you should check for the presence of these features before using them.

In general, plugin modules are free to make calls to the toolbox and OS, as long as these calls are not "visible" to Arrange, i.e. as long as they do not interfere with Arrange's operation. In general this simply means following common sense; for example, don't dispose of a window which Arrange has created. Some specific rules:

- Do not make any direct calls to the Menu Manager to add or remove menus from the menu bar, or edit existing menus. (Creating popup menus for use in a modal dialog is acceptable.) To manipulate Arrange's menu bar, use callback functions such as AddMenuItem and SetMenuItem.
- Do not create or destroy nonmodal windows or dialogs. It is legal to create modal dialogs, but you must run the dialog and put it away before returning control to Arrange.
- Never call GetNextEvent, WaitNextEvent, or EventAvail, except during a modal dialog. (Calling these functions at other times may cause Arrange to miss a suspend or resume event.)
- Don't make drawing calls except in windows you create yourself (i.e. in modal dialogs).

- Arrange 2.0 implements the “required” AppleEvent suite, so the isHighLevelEventAware flag is set in Arrange’s SIZE resource, and therefore plugin modules can send and receive AppleEvents. Specific rules for adding event handlers and receiving events have not yet been defined.
- Arrange makes the following toolbox initialization calls at application startup time: InitGraf, InitFonts, InitWindows, InitMenus, TEInit, InitDialogs, and InitCursor. It also installs event handlers for the four required AppleEvents. If you use toolbox managers which require other initialization calls you must make them yourself (at this time there is no mechanism for avoiding conflicts between multiple plugins which initialize the same toolbox manager).
- Arrange uses a private scrap. This private scrap is only synchronized with the Scrap Manager at suspend and resume events. Therefore, plugins should not make use of the Scrap Manager, as this would cause the user to see an inconsistent clipboard. (This is recognized as a serious limitation and will be rectified in a future release.)

4.2.4 Memory management

This section describes the rules for memory allocation and usage within a plugin module. Plugin modules operate inside Arrange’s application heap; therefore, they must allocate memory from the same pool used by other modules and by Arrange itself.

Arrange allocates a substantial amount of memory at application startup time for nonpurgable code segments, a file buffer cache, and other permanent data structures. Additional memory is allocated and deallocated at runtime for purgable code segments, windows and dialogs, various additional caches, and so forth. Due to the extensive dynamic caching which Arrange performs, and the large amount of purgable resources (code resources and others) in the Arrange application, the amount of free space on the heap tends to be small. This does not mean that no more memory can be allocated, however, because much of the heap is purgable, and Arrange installs a GrowZone procedure to free up additional memory when needed (by releasing caches).

Modules are free to allocate memory by calling the Memory Manager directly (e.g. NewPtr and NewHandle), or by calling Toolbox routines which allocate memory (e.g. GetResource). As mentioned above, space will be made available by purging resources and freeing up caches in Arrange’s GrowZone procedure. However, when possible, modules should instead allocate memory using the AllocMem and DeallocMem callback routines. This allows Arrange to do better memory management, allows small nonrelocatable blocks to be allocated via malloc (which is much more efficient than NewPtr), and provides error checking and leak detection in debugging versions of Arrange. See the callback API documentation for a detailed description of AllocMem and DeallocMem. Also see the MemAvailable function, which gives an idea of how much memory is available for allocation (including purgable caches).

In the future, Arrange may provide support for plugin modules to create cache data structures which can be released by the GrowZone function. For now, plugin modules should try to minimize overall, memory usage, and not create large caches, since there is currently no provision for freeing these caches when needed.

At the present time, Arrange makes no use of MultiFinder temporary memory, and no guidelines are provided for use of temporary memory from a plugin module.

4.3 Plugin file structure

A plug-in module is stored in a Macintosh resource file as one or more resources. Multiple modules can be stored in a single file. The file can be the Arrange application file itself, or a separate "plug-in file" located in the same folder as the Arrange application, or in the Arrange preferences folder.

Each module is defined by a single "MDdf" (Module Definition) resource. This is not an executable code resource. It contains header information consisting of a unique name and ID for the module; version information; memory and system requirements; etc. Each "MDdf" resource defines a distinct plug-in module; Arrange uses the information in the resource at boot time to decide whether the module should be loaded.

> For information on registering completed plug-ins with Common Knowledge, see section 5.2, "Naming and registering your plug-in with Common Knowledge."

The unique ID for a module, stored in its MDdf resource, is a 32-bit value. It is unrelated to the module's resource ID, which can be assigned arbitrarily by the programmer (subject to the conventions listed below). The module name in the MDdf resource is also unique. Module names and IDs are used by Arrange to identify modules, and for modules to identify one another (e.g. if one module calls functions defined in another module). You must contact Common Knowledge to have a name and ID assigned to each module which you create. (For experimental use, use IDs in the range 0x70000000 - 0x7FFFFFFF.)

In addition to the "MDdf" resource, a module will normally contain at least one resource of type "MDcd" (Module Code), which contains the actual executable code for the module. The "MDdf" resource specifies a root "MDcd" resource; position 0 in this resource contains the module's main entry point.

Finally, modules can contain other resources such as PICTs, STR#s, and DITLs, for internal use. Except as noted in specific callback functions, Arrange will never directly reference any plug-in file resource of types other than "MDdf", "MDcd", or "MDdp" (described below).

The resource ID for a "MDdf" resource should be a negative multiple of 512 (i.e. it should be one of -512, -1024, -1536, ..., -32768.) All other resources in a plug-in module should have an ID between x and x+511, inclusive, where "x" is the ID of the "MDdf" resource. This convention will minimize resource ID conflicts between multiple plug-in modules stored in a single file (especially the Arrange application file).

If possible, a module should store all resource IDs relative to the ID of the MDdf resource. For example, if the MDdf resource has ID -512, and the module contains an STR resource with ID

-500, the code should reference this as baseID+12, where baseID is the ID of the MDdf resource. This ID can be obtained from the mod->defID field of the ModuleParamBlock. By storing resource IDs in this relative fashion, a module allows itself to be relocated at a different resource ID so that several modules can be placed in a single file without ID conflicts.

Certain standard resource types include references to other resources; for example, a DLOG resource contains the ID of its DITL. Obviously such references will need to be "patched" if a plugin module is to be relocated. At some point Common Knowledge will provide a utility program to relocate plugin modules and patch resources as needed. (This program would not be supplied to end users, but would be used by developers. For example, Common Knowledge is likely to use such a utility to embed plugin modules into the Arrange application itself as a means of implementing minor feature upgrades without recertifying the full application.)