# Chapter3
# The Arrange API

## 3.1 Overview

In general, callbacks allow a module to examine the contents of any open document, make changes to a document, change the current program state, and customize the user interface. You should skim this section to find those callbacks which are relevant to your purpose, and read their descriptions in detail.

The Hooks section defines a number of points at which a plug-in module can interrupt the normal control flow of the Arrange application, in order to extend or replace its normal behavior. You should skim this section to find those hooks which your plug-in module will need to intercept, and read their descriptions in detail.

Plug-ins use calls in the API to have Arrange modify its internal data structures.

## 3.2 Hooks

A Hook function is a piece of code in a plug-in module, which the module registers with Arrange via a callback. Hook functions are registered for specific events, such as the user choosing a menu item or saving a file. Whenever one of these events occurs, Arrange calls all hook functions which are registered for that event.

Hook functions are the only way for a plug-in module to get control of the machine, aside from the calls to its main entry point at launch and exit time. Typically, a module's initialization function will register one or more hook functions for the events in which the module is interested. For example, a module which reformats data after it has been entered into a field, will register a hook function for the event of the user clicking out of that field.

Arrange supports hook functions for many classes of events, such as menu commands, field events (clicking in a field, typing a keystroke into a field, etc.), tick events, and file events (opening a file, saving a file, etc.). For each event class, there is a callback to register and deregister hook functions for events of that class. Some of these callbacks take additional parameters which can be used to filter the set of events for which the hook function will be called; for example, a field hook can be registered for a particular field, or for all fields. These callbacks are described in section 3.3.11,"UICalls,". This section also gives specific information about the parameters passed to each type of hook function.

### 3.2.1 Module parameter block

Whenever Arrange calls a plug-in module, whether at initialization time or through a hook function, it passes in a standardized parameter block as the first parameter.

The precise structure of the parameter block is defined in the file Module.h; the main components are summarized below. Note that, with the exception of the moduleRefcon field, the parameter block must not be altered by the plug-in module.

```
struct ModuleParamBlock // incomplete; the full definition
                        // is given in Module.h.
   {
   uInteger moduleRefcon;
   uInteger hookRefcon;
   const ModuleSysInfo* sys;
   const ModuleAppInfo* app;
   const ModuleInfo*    mod;
   const CallbackTbl*   calls;
   };
```

The moduleRefcon field is set to 0 when the module is first called (i.e. when its main entry point is called with the initialization message). If the module assigns a new value to this field, it will be stored and passed in the next time the module is called. Modules can therefore use this field to store a pointer to their global variables, allocated at boot time (in response to the initialization message) and released at quit time (in response to the quit message).

The hookRefcon field is only defined when calling a hook function. Its value is taken from the refcon parameter to the call used to register the hook function (e.g. SetFieldHook or SetQuitHook). Unlike moduleRefcon, hostRefcon should not be changed by the plug-in module.

The sys and app fields point to records describing the current system and application; see the definitions of the ModuleSysInfo and ModuleAppInfo records in Module.h for details. These records are guaranteed not to change while the application is running.

The mod field points to a record containing information about the module being called, such as its name, module ID (module IDs are not yet documented), and the resource file from which it was loaded.

The calls field points to a list of function pointers, one for each callback which Arrange makes available to plug-in modules. calls is declared as pointing to a CallbackTbl, but actually points to an ArrangeCallbackTbl, which adds Arrange-specific functions to the general functions defined in CallbackTbl. (CallbackTbl is defined in Module.h, which is application-neutral, and could be used to support plug-in modules for applications other than Arrange. ArrangeCallbackTbl is defined in ArrangeCallbacks.h, which adds declarations for Arrange-specific functionality.)

(Note: to help manage the large number of callbacks, they are grouped into smaller records. The calls field actually points to an outer record, which in turn has a pointer to each of these smaller records.)

## 3.3 Callbacks

Callbacks are routines in the Arrange application that can be called from a plug-in module. They are so named because the module “calls back” to the application after the application makes an initial call into the module. Callbacks are the means by which plug-in modules affect the execution of the program.

Plugin modules access callbacks through a vector of function pointers contained in the module parameter block (described below). This vector is provided every time Arrange calls a module, but its location and contents will not change while the application is running. Modules are free to store either the vector location, or specific function points (copied out of the vector), in their internal data structures.

A module is free to call any callback whenever it receives control of the machine from Arrange, subject to any limitations documented with the specific function. If it receives control in other ways (e.g. by posting a VBL task or patching a trap), it must not use any callbacks, as Arrange’s internal data structures may be in an inconsistent state. One exception is an AppleEvent handler: modules can post AppleEvent handlers and, since these are called at WaitNextEvent time, Arrange may be assumed to be in a consistent state at this point. Therefore it is legal to use callback functions from an AppleEvent handler.

The next few sections document the callback functions supported by Arrange 2.0. The functions are grouped according to the record structures in which they appear in Module.h and ArrangeCallbacks.h (i.e. the individual fields of the CallbackTbl and ArrangeCallbackTbl records). The types used for parameters and function results are either standard Mac types appearing in Types.h or Quickdraw.h, or are defined in one of the supplied interface files (ArrangeCallbacks.h, Module.h, Module.r.h, or ModuleSupport.h). All types defined by Arrange are documented in the interface file where they appear.

### 3.3.1 Conventions

- The keyword OUT, used in a pointer parameter declaration, indicates that the parameter is used to return a value from the function. The caller is not expected to supply a value in the item being pointed to. (OUT is actually an empty #define macro.)
- The keyword IO, used in a pointer parameter declaration, indicates that data passes in both directions through the parameter. The caller is expected to supply a value in the item being pointed to, and that value may be changed by the function. (IO is actually an empty #define macro.)
- The keyword OWN, used in a parameter or function result declaration, indicates that ownership of an object (e.g. an arListID, arPathID, or Handle) is changing hands. In front of a standard parameter declaration, it indicates that the called function is taking over ownership from the caller; in front of an OUT pararameter declaration or a function result, it indicates that the caller receives ownership from the function. “Ownership” refers, primarily, to the responsibility of disposing of the object when it is no longer needed. (OWN is actually an empty #define macro.)
- The types Short and Integer refer to signed 16-bit and 32-bit integers. uShort and uInteger are used to refer to unsigned 16-bit and 32-bit integers. These names are typedef’ed to the appropriate primitive integer types.

### 3.3.2 Data Callbacks

This section lists callback functions used for reading and writing the data in a field.

```
Integer GetFieldTextLen(arNoteID note, arFieldID field);
```

Return the length of the text in a field, or -1 if the field isn’t present or isn’t textual (in the last case, also log an error). For purposes of this function and GetFieldText, we consider file-reference fields to be “textual”; their value is the name of the referenced file (just the file name, not the full path).

```
Integer GetFieldText( arNoteID note, arFieldID field, Integer bufLen, OUT
void* buffer );
```

Copy the text in the given field to the buffer, truncating if necessary to fit in the buffer, and append a zero terminator (we truncate to bufLen-1 characters so as to leave room for the zero terminator within the buffer). Return the (non-truncated) length of the field, excluding zero terminator, as in GetFieldTextLen.

It is legal to call this function with a nil buffer and bufLen = 0; in this case we simply return the length of the text (equivalent to calling GetFieldTextLen).

```
arDate GetFieldDate(arNoteID note, arFieldID field);
```

Return the contents of the given date/time (or time) field, or 0 if the field isn’t present, contains an

unparsable date, or isn't a date/time or time field (in the last case, also log an error).

```
arFloat GetFieldNumber(arNoteID note, arFieldID field);
```

Return the contents of the given number field, or the error value -1E20 if the field isn't present, contains an unparsable number, or isn't a number field (in the last case, also log an error).

```
Integer GetFieldListLen(arNoteID note, arFieldID field);
```

Return the number of entries in a note-list field, or zero if the field isn't present or isn't a note-link field (in the last case, also log an error).

```
arNoteID GetFieldListEntry( arNoteID note, arFieldID field, Integer index );
```

Return a particular entry in a note-list field. If the field isn't present, this isn't a note-link field, or the index (which is zero-based) is out of bounds, log an error and return nil.

```
OWN arListID GetFieldList(arNoteID note, arFieldID field);
```

Return an arListID token representing the contents of the given note-link field, or an empty list if the field isn't present or isn't a note-link field (in the last case, also log an error).

```
OWN PicHandle GetFieldPict(arNoteID note, arFieldID field);
```

Get the contents of a picture field. If the field is empty, return nil. If the field isn't present or isn't a picture field, return -1 (in the last case, also log an error).

If there isn't enough free memory to create the PicHandle, return -2. If the picture is in a foriegn (non-QuickDraw) format, return -3.

```
OWN Handle GetFieldAlias(arNoteID note, arFieldID field);
```

Get the contents of a file-reference field. If the field is empty, return nil. If the field isn't present or isn't a file-reference field, return -1 (in the last case, also log an error).

If there isn't enough free memory to create the AliasHandle, return -2.

```
void SetFieldText( arNoteID note, arFieldID field, const char* text, Integer
textLen, arSetFieldFlags flags );
```

Set the contents of a field instance; if the note doesn't have this field, add it as an invisible field. If the field isn't textual, log an error and do nothing. If the sfDoParse flag is set, and this is a date or number field, then parse the text and update the field's numeric value; otherwise leave the numeric value alone. If the sfDisableModDate flag is set, then don't update the note's modification date and last editor name (this flag should always be set when making changes to system fields).

For text and popup fields, we remove any style information; thus the text will revert to the default style for the field and note type.

Note that the system date fields (date created and date modified) only store a numeric value, not text. For these fields, therefore, this function has no effect unless sfDoParse is set, in which case it stores the numeric value only.

```
void SetFieldDate( arNoteID note, arFieldID field, arDate date,
```

```
arSetFieldFlags flags );
```

Set the numeric value of the given date/time or time field; if the note doesn't have this field, add it as an invisible field. If this isn't a date/time or time field, log an error and do nothing. If the sfDoFormat flag is set, then format the date and update the field's text value; otherwise leave the text value alone. If the sfDisableModDate flag is set, then don't update the note's modification date and last editor name (this flag should always be set when making changes to system fields).

Note that the system date fields (date created and date modified) only store a numeric value, not text. For these fields, therefore, the sfDoFormat flag is ignored.

```
void SetFieldNumber( arNoteID note, arFieldID field, arFloat number,
arSetFieldFlags flags );
```

Set the numeric value of the given number field; if the note doesn't have this field, add it as an invisible field. If this isn't a number field log an error and do nothing. If the sfDoFormat flag is set, then format the number and update the field's text value; otherwise leave the text value alone.

If the sfDisableModDate flag is set, then don't update the note's modification date and last editor name.

```
void SetFieldList( arNoteID note, arFieldID field, arListID list,
arSetFieldFlags flags );
```

Overwrite the contents of a note-link field with a list of notes contained in a ListID; if the note doesn't have this field, add it as an invisible field. If the field isn't a note-link field, log an error and do nothing.

If the sfDisableModDate flag is set, then don't update the note's modification date and last editor name.

A nil value for the list parameter is interpreted as the empty list.

```
void AddListFieldEntry( arNoteID note, arFieldID field, Integer index,
arNoteID childNote, arSetFieldFlags flags );
```

Add an entry to a note-link field at the given (0-based) index. If the field isn't present, first add it as an invisible field with an empty list. If the field isn't a note-link field, or the index is out of bounds, log an error and do nothing. If the childNote was already present in the field, it is moved to the given index.

If the sfDisableModDate flag is set, then don't update the note's modification date and last editor name.

This function, along with DeleteListEntry and MoveListEntry (below), can be used to edit the contents of a topic or view. However, in a view, it will not cause the parent topic (or folder) to be edited. To get this behavior, you must invoke high-level operations such as Cut, Paste, and Drag.

```
void DeleteListFieldEntry( arNoteID note, arFieldID field, arNoteID
childNote, arSetFieldFlags flags );
```

Remove an entry from a note-link field. If the field isn't present, isn't a note-link field, or childNote isn't present in the field, log an error and do nothing.

```
void MoveListFieldEntry( arNoteID note, arFieldID field, arNoteID childNote,
Integer index, arSetFieldFlags flags );
```

Move an entry from one position to another in a note-link field. If the field isn't present, isn't a note-link field, childNote isn't present in the field, or the (0-based) index is out of bounds, log an error and do nothing.

If the sfDisableModDate flag is set, then don't update the note's modification date and last editor name.

```
void SetFieldPict( arNoteID note, arFieldID field, PicHandle pict,
arSetFieldFlags flags );
```

Set the contents of a picture field. The PicHandle remains the property of the caller. If a nil handle is passed, make the field be empty. If the note doesn't have this field, add it as an invisible field. If the field isn't a picture field, log an error and do nothing.

If the sfDisableModDate flag is set, then don't update the note's modification date and last editor name.

```
void SetFieldAlias( arNoteID note, arFieldID field, Handle alias,
arSetFieldFlags flags );
```

Set the contents of a file-reference field. The handle (actually an AliasHandle) remains the property of the caller. If a nil handle is passed, make the field be empty. If the note doesn't have this field, add it as an invisible field. If the field isn't a file-reference field, log an error and do nothing. Note: for convenience, we might also define a version of this routine which takes an FSSpec instead of an alias handle.

If the sfDisableModDate flag is set, then don't update the note's modification date and last editor name.

```
void ReplaceFieldText( arNoteID note, arFieldID field, Integer repStart,
Integer repEnd, const char* insText, arSetFieldFlags flags);
```

Overwrite a portion of the contents of a field instance. The field must already be present in the note and must be a text or pop-up field; if this isn't the case, we log an error and do nothing. If th sfDisableModDate flag is set, then don't update the note's modification date and last editor name (this flag should always be set when making changes to system fields).

Characters in the range [repStart, repEnd) in the target field are deleted and replaced with the given text. For example, if the target field contains the text "Run, Spot, run!", repStart is 5, repEnd is 9, and insText is "Dick and Jane", then after this call the target field will contain the text "Run, Dick and Jane, run!". If repStart is negative, repEnd is less than repStart, or repEnd is greater than the length of the target field, we log an error and do nothing.

In summary, this function is identical to SetFieldText, except that (a) instead of overwriting the entire field contents, it only overwrites a section of the field, (b) the text to be inserted must be zero-terminated, (c) the style information for the remaining text is retained, and (d) it is illegal to call this function for a field that isn't already present in the note, or for fields of types other than text and pop-up.

```
void GetFieldStyle( arNoteID note, arFieldID field, OUT arFieldStyle*
fieldStyle, Integer bufLen, OUT arCharStyle* charStyles );
```

Return the style information for the text in the given field. If the field is not textual, log an error and return 0.

Style parameters applying to the entire field are returned in the fieldStyle record. Style parameters which can vary from character to character are returned in the charStyles array. bufLen should be the number of records in the charStyles array; our function result is the number of records needed for the entire field. If the function result is greater than bufLen, some of the style information was dropped.

On return, each entry in the charStyles array describes the style of one or more characters of text in the given field. The number of characters described is given in the charCount field of each record.

```
void SetFieldStyle( arNoteID note, arFieldID field, const arFieldStyle*
fieldStyle, Integer charStyleCount, const arCharStyle* charStyles,
arSetFieldFlags flags );
```

Set the style information for the text in the given field. If the field is not a text or pop-up field, or isn't present in the note, log an error and do nothing. If the sfDisableModDate flag is set, then don't update the note's modification date and last editor name.

It is illegal to call this function for the creator name and last-editor name system fields; in this case we log an error and do nothing.

Style parameters applying to the entire field are passed in the fieldStyle record. Style parameters which can vary from character to character are passed in the charStyles array. charStyleCount should be the number of records in the charStyles array. The sum of the charCount fields of the entries in the charStyles array must equal the length of the text in the field.

### 3.3.3 NoteCalls

This section lists callback functions used for creating, destroying, and managing entire notes, and for adding and removing fields from a note.

```
arNoteID CreateNote(arTypeID type, Boolean doGather);
```

Create a new note. This does not place the note anywhere in the document, except in the appropriate auto-gather topics (if doGather is true). You will need to make additional calls (e.g. to AddListFieldEntry) to insert the note somewhere in the document.

It is illegal to use this function to create system objects such as topics and field definitions.

```
arNoteID DuplicateNote(arNoteID note, Boolean doGather);
```

Duplicate a note. This is equivalent to the Duplicate Note command, in that it copies the contents of the note's fields and makes new references to the note's subnotes (and to any notes referenced in a note-link field). If doGather is true, the new note is added to any appropriate auto-gather topics.

It is illegal to use this function to duplicate system objects such as topics and field definitions.

```
void ConvertNote(arNoteID note, arTypeID type, Boolean doGather);
```

Change a note's type (equivalent to the Convert Note command). If the note is already of the given type, do nothing.

If doGather is true, the new note is added to any appropriate auto-gather topics for its new types. Note that we do *not* remove the note from the auto-gather topics (if any) for its old type.

It is illegal to use this function to convert a note from or to one of the system object types.

```
void DestroyNote(arNoteID note);
```

Destroy a note. This is equivalent to the Destroy Note command - it rips the note out everywhere that it appears. This function can also be used to destroy system objects such as topics, field definitions, note definitions, and windows (so use it with care!).

```
void AddField(arNoteID note, arFieldID field, arFieldID nextField);
```

Add a visible field to a note, inserting it before "nextField" and giving it the appropriate default value. If the field was already visible, move it to before nextField; if it was present but invisible, then make it visible (without changing its value). If nextField is nil or isn't present, insert the field just before the

subitems field (in the last case, also log an error).

To add an invisible field to a note, use the appropriate SetFieldXXX call.

This function can be called for a note type definition; in this case, it adds the field to all notes of that type (notes already having the field are unaffected).

```
void RemoveField(arNoteID note, arFieldID field);
```

Remove a field (whether visible or invisible) from a note. If the field isn't present, or is a visible field of the note's type definition, log an error.

This function can be called for a note type definition; in this case, it removes the field from all notes of type (assuming that the field was part of the type definition; otherwise, it does nothing).

WARNING: all notes must have at least one visible textual field, or Arrange may crash. You should not use this function to remove the last visible textual field from a note or type definition.

```
void HideField(arNoteID note, arFieldID field);
```

Hide a field, leaving it present in the note but invisible. If the field was already hidden, do nothing. If the field isn't present, or is a visible field of the note's type definition, log an error.

Note that there is no explicit "ShowField" function, since AddField can be used for this purpose.

This function can be called for a note type definition; in this case, it hides the field in all notes of that type (assuming that the field was visible in the type definition; otherwise, it does nothing).

WARNING: all notes must have at least one visible textual field, or Arrange may crash. You should not use this function to hide the last visible textual field in a note or type definition.

```
void MoveField(arNoteID note, arFieldID field, arFieldID nextField);
```

Move one visible field of a note to be just before another. If either field is invisible or is not present at all, log an error and do nothing.

If called for a note type definition, this function only affects the field ordering for notes created in the future; existing notes are unaffected.

```
Integer CountNoteFields(arNoteID note);
```

Return the number of visible fields in the given note; this includes the subnotes field and (if present) the note-text field. It does NOT include any invisible fields. Invisible fields can only be accessed by calling functions such as NoteHasField and GetFieldText with the proper field ID.

```
arFieldID GetNoteField(arNoteID note, Integer index);
```

Return a particular field from a note's visible-field list. If the index (which is zero-based) is out of bounds, log an error and return nil. See CountNoteFields for a description of the visible-field list.

```
Boolean NoteHasField( arNoteID note, arFieldID field, Boolean
onlyIfVisible );
```

Return true if the given field is present in the given note. If onlyIfVisible is true, then only return true if the field is both present and visible.

```
Boolean NoteExists(arNoteID note);
```

Return true if a note (including system objects) with the given ID exists in the current document.

```
arTypeID GetNoteType(arNoteID note);
```

Return the note type for a note.

```
void GetNoteInfo(arNoteID note, IO arNoteInfo *info);
```

Fill in the info record for the given note. The caller must initialize the versNum field of the record before calling this function; GetNoteInfo fills in all remaining fields (including the pad field).

If this version of Arrange doesn't support the given version number, we set the version field of the record to a (smaller) value that we do support.

```
void SetNoteInfo(arNoteID note, const arNoteInfo *info);
```

Set the specified information for the given note. To use this function, you typically call GetNoteInfo, change one or more fields in the record it returns, and then call SetNoteInfo to store the new values.

Currently, this function accepts changes to the createDate, lastModDate, creatorName, editorName, hilightColor, and hilightIcon fields. Changes to other fields of the info record are ignored.

In Arrange 2.0, the hilightIcon field is not supported, and hilightColor can only be on or off (i.e. 0 or 1).

If this version of Arrange doesn't support the given version number, log an error.

### 3.3.4 SelCalls

This section lists callback functions used for inspecting and altering the current selection.

```
arWindowID GetActiveWindow();
```

Return the ID of the frontmost window for the current document. If the document has no open windows, return nil. Note that this function will return hoist windows as well as regular document windows. Most other functions in this section apply to the frontmost window.

```
arSelType GetSelection( OUT arNoteID *note, OUT arFieldID *field, OUT
Integer *selStart, OUT Integer *selEnd );
```

Return the state of the current selection in the active window. The OUT parameters give information about the selected object(s); their values are undefined except as mentioned in the definition of the arSelType enum.

```
OWN arPathID GetSelPath();
```

Return the path associated with the current selection (as defined by GetSelection), or nil if there is no current selection, or it does not have an associated path (i.e. either more than one object is selected, or the path to the one selected object is not known).

```
void GetSelEntry( Integer index, OUT arNoteID *note, OUT arFieldID *field,
OUT arNoteID *parentNote );
```

Get one entry from the list of selected notes or fields in the active window. The values returned depend on the nature of the selection (classified according to the value GetSelection would return); all values are undefined except as explicitly metioned.

stNone - signal an error.

stNote, stMultipleNotes - "note" is set to the selected note, "field" is set to the field in which it appears, and "parentNote" is set to the note owning that field. The index should be zero for stNote, or in the range [0, count-1] for stMultipleNotes; otherwise an error is signaled and the results are undefined.

stField, stMultipleFields - "note" is set to the selected field, "field" is set to nil, and "parentNote" is set to the note in which it appears. The index is treated as for stNote and stMultipleNotes.

stText, stFieldContents - behave as if the field itself were selected (stField).

Note that for stField and stMultipleFields, one might expect the ID of the selected field to be returned in the "field" parameter rather than the "note" parameter. However, we return it in the note parameter, to match the historical definition of a selection triple.

```
void SelectObject( arNoteID parent, arFieldID field, arNoteID note, Boolean
doDeselect );
```

Select the note or field instance defined by the triple (parent, field, note). See the section on “Selections” in the Data Model topic for an explanation of selection triples. If doDeselect is true, we remove the object from the selection; otherwise we add it. If the desired state is already in effect (doDeselect=false and the object is already selected, or doDeselect=true and the object isn’t selected), do nothing.

If the current selection is incompatible with this operation, log an error and do nothing. This occurs when some text or other field contents are selected; when notes are selected and you are trying to add a field instance; and when field instances are selected and you are trying to add a note.

When you wish to select a single note or field in a particular context, it is better to call ShowPath (below), since ShowPath function will make sure that the note or field is visible, and will record the path under which it was selected for future operations (such as GetSelPath).

```
Boolean ObjectIsSelected(arNoteID parent, arFieldID field, arNoteID note);
```

Return true if the note or field instance defined by the triple (parent, field, note) is part of the selection. If there is no selection, or it is of an unrelated type (e.g. some text is selected), return false. This function returns true if and only if there is some index for which GetSelEntry would return the given triple; except that GetSelEntry will treat a text selection as being equivalent to selecting the field instance in which the text appears, and this function will not.

```
Boolean FlushSelection(Boolean alsoDrop);
```

Flush any changes made to the active text field, if any. If alsoDrop is true, then drop the selection (whether it is of text or something else) afterward. Return true unless we are unable to flush the changes (e.g. because it contains some illegal value), in which case the alsoDrop parameter is ignored and we return false.

At the present time (Arrange 2.0), there are no circumstances under which we would be unable to flush the changes. However, such circumstances existed in earlier versions of the program (e.g. if you entered a bad name for a field in the Fields system topic), and I expect they will arise again in the future, especially with plugin modules (e.g. a module which enforces uniqueness of serial numbers in a product registration database).

It is a good idea to call FlushSelection(false) before beginning any command which depends on field

contents. For example, Arrange itself calls this function before executing commands like Find, Gather, or Update View. If FlushSelection return false, then abort the execution of your command. If you don't call FlushSelection, then the last editing operation the user made won't be visible in calls like GetFieldText.

```
void ShowPath( arPathID path, Boolean selectFlag, Integer textSelStart,
Integer textSelEnd );
```

Make visible the note or field instance at the end of the given path. This may involve switching current the topic, scrolling the notes list, expanding or opening notes, and possibly bringing a different window to the front (e.g. if the path is anchored in the shelf of a back window, or if a hoist window is currently active and the path doesn't fall within the hoist window). If selectFlag is true, then select the note or field; otherwise just make it visible. If the field is a textual field, and textSelStart is non-negative, then select the specified character range within the field; otherwise (the field isn't textual, or textSelStart is negative), select the field instance itself.

This function may fail to execute, if there is a pre-existing selection which refuses to be flushed.

```
Integer GetSelText( Integer bufLen, OUT void* buffer, OUT Integer *selStart,
OUT Integer *selEnd );
```

Copy the text in the active textual field to the buffer, truncating if necessary to fit in the buffer, and append a zero terminator (we truncate to bufLen-1 characters so as to leave room for the zero terminator within the buffer). Return the (non-truncated) length of the text, excluding zero terminator. Set selStart and selEnd to the range of selected text.

This function returns the latest state of the text as edited by the user, even if the changes have not yet been flushed.

If the current selection consists of anything other than the contents of a textual field, log an error and return -1; selStart and selEnd will be undefined.

It is legal to call this function with a nil buffer and bufLen = 0; in this case we simply return the length of the text.

```
void SetSelText( const char* text, Integer textLen, Integer selStart,
Integer selEnd );
```

Overwrite the contents of the selected field with the given text, and set the selection to the given range of characters. If text is nil, we ignore the textLen parameter and only change the selection range. If the current selection consists of anything other than the contents of a textual field, log an error and do nothing.

This function does not immediately change the contents of the field instance in the Arrange document database. It is equivalent to the user typing new text into the field. The changes will be stored when FlushSelection is called or the user clicks in a different field.

```
void ReplaceSelText( const char* text, const char* undoName );
```

Replace the contents of the current text selection (or insertion point) with the given zero-terminated text. If there is no selection, or the selection is not textual, log an error.

If undoName is not nil, then we set the name of the first item in the edit menu to "Undo xxx", where xxx is the contents of the undoName string; if the user chooses Undo, we revert the contents of the text selection to its state before this call. If undoName is nil, we leave the current undo setting unchanged.

Like SetSelText, this function does not immediately change the contents of the field instance in the

Arrange document database. It is equivalent to the user typing new text into the field. The changes will be stored when FlushSelection is called or the user clicks in a different field.

### 3.3.5 SearchCalls

This section lists callback functions used for searching, sorting, and filtering data in an Arrange document.

```
Boolean NextAppearance(IO arPathID path, arDirCode direction);
```

Alter the given path object to point to the next or previous appearance of the note at its tip, wrapping around the end of the document if necessary. If no other appearance is found, leave the path unmodified and return false. This function performs essentially the same action as the Next/Previous Appearance command in the Search menu.

```
Boolean FindText( IO arPathID path, arDirCode direction, IO Integer
*selStart, IO Integer *selEnd, const char* searchString, arSearchFlags flags
);
```

Search, starting from the given path, for an appearance of the given string. If a match is found, set path, selStart, and selEnd to reflect the matching text and return true; otherwise leave all parameters unmodified and return false. We search either all notes in the topic in which the path is rooted, or all topics in the document, depending on whether the sfTopicScope flag is set. On entry, the selStart and selEnd parameters define where to start searching from in the field pointed to by the path. If the path does not end in a textual field, the input values for selStart and selEnd are ignored.

NOTE: This function is not implemented in Arrange 2.0. It will always return false.

```
OWN arListID FilterList( arListID list, arTypeID matchType, arFilterFlags
flags, Short clauseCount, const arFilterClause* clauses );
```

Return a new list containing all notes in the input list whose type is equal to matchType, and which match each entry in the clauses list. To accept notes of all types, pass nil for matchType. If the ffOrClauses flag is set, we accept notes which match any clause, rather than only accepting notes which match all of the clauses. If the ffSearchSubnotes flag is set, we (recursively) examine the subnotes of all notes in the input list. If the ffInvert flag is set, then we return those notes which would otherwise not have been returned, and vice-versa (thus, if matchType is non-nil and ffInvert is set, only notes of types *other* than matchType will be returned).

```
OWN arListID FilterNotes( arTypeID type, arFilterFlags flags, Short
clauseCount, const arFilterClause* clauses );
```

This function is identical to FilterList, except that the list is taken to be all notes of the given type. If type is nil, we examine all notes of all types (in this case, the ffSearchSubnotes flag is ignored, since all notes are examine as root notes anyway). Note that setting type to nil can cause system objects such as topics and field definitions to be matched.

```
OWN arListID SortNotes( arListID list, Short clauseCount, const
arSortClause* clauses );
```

Return a new list containing the same notes as the input list, but sorted according to the given sorting rule. The first clause has primary importance; subsequent clauses are used to break ties. If two notes tie on all clauses, the retain the same relative order as in the input list.

## 3.3.6 SysObjCalls

This section lists callback functions used for creating and managing system objects.

```
arNoteID GetBuiltInObject(arBuiltInObject whichObject);
```

Return the ID of a built-in note type, field definition, folder, topic, or other system object. Note that many of these objects are not visible to the user. See the definition of the arBuiltInObject enum for a list of the objects which can be accessed through this function.

```
arNoteID LookupObjectName( arObjectClass objClass, const char* name, Boolean
allowPartial );
```

Look for a note type, field definition, topic, folder, or view having the given name (which type is searched for depends on the “class” parameter). If allowPartial is true, then we match any object whose name begins with the given string; otherwise we require an exact name match. Case is not considered to be significant. If no objects match the given name, return nil; if multiple objects match the name, return the one whose name is first alphabetically.

This function does not expand wildcards.

### 3.3.6.1 Topics, folders, and views

```
arTopicID CreateTopic( const char* name, arTopicID parent, arTopicID
successor, Boolean makeView );
```

Create a new topic or view with the given name. If makeView is true, then this function creates a new view of the given parent topic; otherwise it creates a new topic in the given parent folder. To create an invisible topic, place it in the System folder.

The successor parameter determines where the new topic or view will be inserted in the parent's list of child topics or views. If successor is nil, the new topic/view is placed at the end of the list; otherwise, it is placed just before the successor object. If successor is not nil and does not appear in the parent's list of topics or views, signal an error.

When creating a topic, if the parent folder is currently collapsed, it will be expanded so that the new topic is visible in the contents list (however, we do not automatically scroll the contents list if the topic's name is offscreen).

When the standard New Topic or New View commands create a new topic or view, they switch the front window to show the new topic/view and activate its name for editing. You can do the first half of this by calling SetCurrentTopic after calling CreateTopic. At the moment, there is no way for a plug-in module to activate a topic or view's name for editing.

NOTE: the successor parameter is currently ignored when creating views; a new view is always added at the end of its parent's list of views.

NOTE: the standard New View command imposes a limit of 200 views for a single topic or folder (for a folder, the limit is on the number of direct views of that folder, not the total number of views for all topics inside the folder). CreateTopic does not enforce that limit; you should not call CreateTopic to add a view to a topic or folder which already has 200 or more views. The 200-view limit is based on performance and memory- usage considerations; it is not a hard limit in the implementation.

```
arTopicID CreateFolder(const char* name, arTopicID successor);
```

Create a new folder with the given name, placing it before the given successor folder. If successor is nil, place it at the end of the list of folders in this topic.

When the standard New Folder command is chosen, it switches the front window to show the new folder and activates its name for editing. You can do the first half of this by calling SetCurrentTopic after calling CreateFolder. At the moment, there is no way for a plug-in module to activate a folder's name for editing.

```
void UpdateView(arTopicID view);
```

Recalculate the contents of a view (equivalent to the Update Now command). You should generally drop the selection in the front window (by calling FlushSelection(true)) before calling this function, if the view in question is currently selected.

```
void GetTopicInfo(arTopicID topic, IO arTopicInfo *info);
```

Fill in the info record for the given folder, topic, or view. The caller must initialize the versNum field of the record before calling this function; GetTopicInfo fills in all remaining fields (including the pad field).

If this version of Arrange doesn't support the given version number, we set the version field of the record to a (smaller) value that we do support.

```
void SetTopicInfo(arTopicID topic, const arTopicInfo *info);
```

Set the specified information for the given folder, topic, or view. To use this function, you typically call GetTopicInfo, change one or more fields in the record it returns, and then call SetTopicInfo to store the new values.

Currently, this function only accepts the defaultType field, and only for topics. For folders and views it does nothing.

If this version of Arrange doesn't support the given version number, log an error.

#### 3.3.6.2 Type definitions

```
arTypeID CreateNoteType(const char* name, Boolean userVisible);
```

Create a new note type with the given name. If userVisible is true, then the definition will be added to the shelf, appear in the notes catalog, and generally be made visible to the user; otherwise it will be invisible (like the built-in note types for system objects).

The note type will have two visible fields: the note-text field and the subnotes field. Additional fields can be added, and the note-text field can be removed, after the note type is created. The note type will have no summary line (unless changed by a call to SetTypeInfo); this differs from the default behavior when the user creates a new note type, where we give the note type an automatic summary line.

This function should never be called with a name that is already used for a note type in the current document; however, this condition is only checked in module-debugging versions of Arrange.

```
Integer CountTypeInstances(arTypeID type);
```

Return the number of notes having the given note type. If the type is nil, return the number of notes of all types (including system-defined types).

```
OWN arListID GetTypeInstances(arTypeID type);
```

Return a list of all notes of the given type. If the type is nil, return a list of all notes of all types (including system-defined types).

```
void GetTypeInfo(arTypeID type, IO arTypeInfo *info);
```

Fill in the info record for the given note type. The caller must initialize the versNum field of the record before calling this function; GetTypeInfo fills in all remaining fields (including the pad field).

If this version of Arrange doesn't support the given version number, log an error.

```
void SetTypeInfo(arTypeID type, const arTypeInfo *info);
```

Set the specified information for the given note type. To use this function, you typically call GetTypeInfo, change one or more fields in the record it returns, and then call SetTypeInfo to store the new values.

If this version of Arrange doesn't support the given version number, log an error.

#### 3.3.6.3 Field definitions

```
arFieldID CreateField( const char* name, arFieldType dataType, Boolean
userVisible );
```

Create a new field with the given name. If userVisible is true, then the field will be added to the fields catalog and generally be made visible to the user; otherwise it will be invisible.

This function should never be called with a name that is already used for a field in the current document; however, this condition is only checked in module-debugging versions of Arrange.

```
void GetFieldInfo(arFieldID field, IO arFieldInfo *info);
```

Fill in the info record for the given field definition. The caller must initialize the versNum field of the record before calling this function; GetFieldInfo fills in all remaining fields (including the pad field).

If this version of Arrange doesn't support the given version number, log an error.

```
void SetFieldInfo(arFieldID field, const arFieldInfo *info);
```

Set the specified information for the given field definition. To use this function, you typically call GetFieldInfo, change one or more fields in the record it returns, and then call SetFieldInfo to store the new values.

Note that changing a field's data type is potentially a very destructive operation, so use this routine with care.

If this version of Arrange doesn't support the given version number, log an error.

#### 3.3.6.4 Windows

```
arWindowID CreateWindow(const char* name, Rect bounds);
```

Create a new window for the current document, with the given name, placing it in front of all existing windows, and with the given bounds (in global coordinates). If the bounds rectangle is empty, or does not

intersect any active monitor, then we compute a default bounds rectangle.

Before creating a window, this function will flush the selection in the active window. If the selection refuses to be flushed (see FlushSelection), we return nil.

```
arTopicID GetCurrentTopic(arWindowID window, Boolean returnRoot);
```

Return the ID of the current (selected) topic or folder in this window. If the window is showing a view, and returnRoot is false, then return the ID of the view itself; otherwise return the ID of its parent topic or folder.

```
Boolean SetCurrentTopic( arWindowID window, arTopicID topic, Boolean
useLastView );
```

Switch this window to display the given topic, folder, or view. Return false if we are unable to do so because the current selection cannot be flushed. If the target is a view, and the view's "Update View on Entry" flag is set, then recalculate the view's contents. If the target is a topic or folder, the useLastView flag is set, and on the most recent occasion that this topic or folder was selected, we were displaying a view of that topic or folder, then switch into that view instead. You should normally pass true for useLastView (this is the standard Arrange behavior when, for example, the user clicks on a topic name in the Contents list).

```
Boolean SelectWindow(arWindowID window);
```

Bring the given window to the front, making it visible if it was hidden. If the window was already active, do nothing. If the current front window contains a selection which cannot be flushed, do nothing and return false.

If the specified window is currently hidden, we make it visible before activating it.

```
void GetWindInfo(arWindowID window, IO arWindInfo *info);
```

Fill in the info record for the given window. The caller must initialize the versNum field of the record before calling this function; GetWindInfo fills in all remaining fields (including the pad field).

If this version of Arrange doesn't support the given version number, log an error.

```
void SetWindInfo(arWindowID window, const arWindInfo *info);
```

Set the specified information for the given window. To use this function, you typically call GetWindInfo, change one or more fields in the record it returns, and then call SetWindInfo to store the new values. This function can be used to manipulate document windows and the Note and Field Catalogs.

If this version of Arrange doesn't support the given version number, log an error.

### 3.3.7 ListCalls

This section lists callback functions used for manipulating arListIDs and arPathIDs.

```
Integer GetListLen(arListID list);
```

Return the length of a list. If the list is nil, return 0.

```
arNoteID GetListEntry(arListID list, Integer index);
```

Return a particular entry from a list. If the (zero-based) index is out of bounds, or the list is nil, log an error and return nil.

```
Integer GetListEntries( arListID list, Integer index, OUT arNoteID* buffer,
Integer bufSize );
```

Copy a group of entries from a list into the given buffer. Return the number of entries actually copied; this is the size of the buffer, or the number of remaining entries in the list, whichever is smaller. If the index is out of bounds, or the list is nil, return zero and do *not* signal an error.

```
Integer SearchList(arListID list, arNoteID target);
```

Search through a list for a particular note. Return the smallest (0-based) index where the note appears, or -1 if the note does not appear anywhere in the list. If the list is nil, always return -1.

```
void SetListEntry(IO arListID *list, Integer index, arNoteID entry);
```

Overwrite a list entry. If the list is nil, or the index is out of bounds, log an error and do nothing. Note that this takes a pointer to the arListID, as the arListID value may change as a result of the operation (similar to calling realloc for a free-store block).

```
void SetListEntries( IO arListID *list, Integer index, const arNoteID*
entries, Integer count );
```

Overwrite a series of list entries. If the list is nil, or any of the entries are out of bounds, log an error and do nothing. As with SetListEntry, this takes a pointer to the arListID, which may change as a result of the operation.

```
void InsertListEntry(IO arListID *list, Integer index, arNoteID entry);
```

Insert a note into a list at the given (0-based) index. If the list is nil, or the index is out of bounds, log an error and do nothing. As with SetListEntry, this takes a pointer to the arListID, which may change as a result of the operation.

```
void InsertListEntries(IO arListID *list, Integer index, Integer count);
```

Insert a group of entries into a list. The new entries will be initialized to nil. If the list is nil, or the index is out of bounds, log an error and do nothing. As with SetListEntry, this takes a pointer to the arListID, which may change as a result of the operation.

```
void DeleteListEntries(IO arListID *list, Integer index, Integer count);
```

Delete a group of entries from a list. If the list is nil, or the index or count are out of bounds, log an error and do nothing. As with SetListEntry, this takes a pointer to the arListID, which may change as a result of the operation.

```
void ResizeList(IO arListID *list, Integer newSize);
```

Resize a list. If the list grows larger, the new entries are initialized to nil. If the list is nil, log an error and do nothing. As with SetListEntry, this takes a pointer to the arListID, which may change as a result of the operation.

```
OWN arListID CreateList(Integer length);
```

Create a new list of the given length.

```
OWN arListID DupList(arListID list);
```

Duplicate a list. If the list is nil, return nil.

```
void DisposeList(OWN arListID list);
```

Dispose of a list. If the list is nil, do nothing.

```
arPathType GetPathInfo(arPathID path, OUT arNoteID *root);
```

Return a value indicating where the given path is rooted, or ptInvalid if the path is invalid or undefined. If the result is not ptInvalid, root will be set to the ID of the path's root window or topic. If the path is nil, return ptInvalid.

```
Integer GetPathLen(arPathID path);
```

Return the length of the chain of notes and fields in this path. If the path is invalid or nil, return zero.

```
arNoteID GetPathEntry(arPathID path, Integer index);
```

Return the note or field at the given (0-based) index along the path. If the path is nil or invalid, or the index is out of bounds, log an error and return nil.

```
Boolean PushPath(arPathID path, arNoteID newEntry);
```

Append a note or field to the end of a path, increasing the path's length by one. If the path's current length is even, newEntry should be a note; if the path's current length is odd, newEntry should be a field. Return true unless the new path is invalid (i.e. the given note or field does not appear in the previous field or note on the path), in which case we replace the path by an invalid path and return false.

If the path is nil or invalid, log an error return false.

```
void PopPath(arPathID path);
```

Pop the last note or field off the end of the path, decreasing the path's length by one. If the path is nil, invalid, or empty, log an error and do nothing.

```
Boolean WalkPath( arPathID path, arDirCode direction, IO arWalkCache*
cache );
```

Advance a path to the next or previous position in the document. If we walk off the end of the document, leave the path in an undefined state and return false; otherwise return true.

If the path is nil or invalid, log an error and return false.

The cache parameter should point to a variable of type arWalkCache, which should be initialized to nil. WalkPath will fill it in with a pointer to a data structure used internally to optimize subsequent calls to WalkPath. You should dispose of this object by calling DisposeWalkCache after the last call to WalkPath. If you make any change to the path object except through WalkPath (e.g. by calling PushPath or PopPath), or if you reverse the direction in which you are walking, you must invalidate the cache by calling DisposeWalkCache and then setting your local variable to nil.

The following code snippet illustrates how to use WalkPath and DisposeWalkCache:

arWalkCache myCache = nil;

while (WalkPath(myPath, dForward, &myCache) && <some condition on myPath>)

{ ... }

DisposeWalkCache(cache);

```
void DisposeWalkCache(arWalkCache cache);
```

Dispose of a cache object created implicitly by a call to WalkPath.

```
arPathID PathToTopic(arTopicID topic, Boolean atEnd);
```

Create a new path pointing to the beginning or end of the given topic. If topic is nil, we return a path pointing to the beginning or end of the first or last nonempty topic in the document. If all topics in the document are empty, we return nil.

A path to the beginning of a topic is a length-one path to the first note in the topic. If the topic is empty, we return nil.

A path to the end of a topic is a path to the last field of the last subnote in the topic, i.e. the last field of the last note in the last field of the last note of... of the last note in the topic. If the topic is empty, we return nil.

```
Boolean EqualPaths(arPathID path1, arPathID path2);
```

Return true if two path objects are equal. If either path is nil, return true if and only if both paths are nil.

```
arPathID DupPath(arPathID path);
```

Return a new arPathID made by duplicating the given path. If the path is nil, return nil.

```
void DisposePath(arPathID path);
```

Dispose of the given path object. If the path is nil, do nothing.

### 3.3.8 ViewCalls

This section lists callback functions used to manipulate the current display state.

```
arNoteState GetNoteState(arNoteID note, arNoteID parentNote);
```

Return the state of the given note as it appears in the given parent note.

```
void SetNoteState( arNoteID note, arNoteID parentNote, arNoteState
newState );
```

Set the state of the given note as it appears in the given parent note.

### 3.3.9 DocCalls

This section lists callback functions used to manipulate documents.

```
arDocumentPtr GetCurrentDoc();
```

Return the arDocumentPtr value for the "current document". Unless explicitly changed by a call to SetCurrentDoc, this will generally be the active document, i.e. the document corresponding to the active window (or nil if there are no open windows).

```
void SetCurrentDoc(arDocumentPtr newDoc);
```

Make the given document be the “current document”. This remains in effect until the next call to SetCurrentDoc, or until control returns from the plugin module back to Arrange. (Note that when one plugin module calls another directly, returning to the previous plugin module does not restore the current document.)

If you pass a bad value for newDoc, Arrange will crash hard. However, this is not checked in non-debugging versions of Arrange.

```
Integer CountOpenDocs();
```

Return the number of currently open documents.

```
arDocumentPtr GetIndexedDoc(Integer index);
```

Return a pointer to the given entry in the list of open documents. If the index is out of bounds (it should range from zero to CountOpenDocs-1), log an error and return the current document.

```
arDocumentPtr GetPrefsDoc();
```

Return a pointer to the document used for storing global preferences information. If we were unable to open the preferences document at boot time (an unlikely event), return nil.

```
void GetDocInfo(IO arDocInfo *info);
```

Fill in the info record for the current document. The caller must initialize the versNum field of the record before calling this function; GetDocInfo fills in all remaining fields (including the pad field).

If this version of Arrange doesn't support the given version number, log an error.

```
arDocumentPtr NewDoc( const FSSpec* stationeryFile, const char* title );
```

Create a new document based on the given stationery file, and make it be the current document. To use the default, "built-in" stationery file, pass in the FSSpec of the Arrange application (this can be obtained from the ModuleAppInfo record in the module parameter block). To create a "raw" document with no stationery file, pass nil for the FSSpec parameter.

If you pass nil for the title parameter, a unique title of the form "Untitled n" will be generated.

If for some reason we are unable to create a new document (e.g. insufficient memory, or a file-system error occurs), we display an error message and return nil. Otherwise we make the new document be the "current document" and return a pointer to it.

NOTE: this function is not currently implemented in Arrange 2.0.

```
arDocumentPtr Open(const FSSpec* srcFile, Boolean readOnly);
```

Open a document stored in the given disk file, and make it be the current document. The file will be opened read-only if the readOnly parameter is true, or if we are unable to get write permission to the file (e.g. if it is on a locked disk). The file should contain either a native Arrange file or a TAIL file.

If for some reason we are unable to open the file (e.g. insufficient memory, or a file-system error occurs), we display an error message and return nil. Otherwise we make the document be the "current document" and return a pointer to it.

If the given file is already open in Arrange, we bring it to the front and return a pointer to the existing document object.

NOTE: this function is not currently implemented in Arrange 2.0.

```
void Close();
```

Close the current document file, and set the "current document" to nil. Any unsaved changes in the document will be lost.

```
arFileOpResult Save();
```

Write the latest state of the current document file to the disk. This is a low-level operation and does not, for example, export a copy of the document in TAIL format if the user has selected that option. If the current document is "untitled" and hence has no associated disk file, log an error.

```
arFileOpResult SaveAs(const FSSpec* targetFile);
```

Save the current document under the given name. This is similar to the Save As menu item, in that it causes the document to henceforth be associated with the given disk file; subsequent calls to Save will write to that file. However, like the Save call, it is a low-level operation and does not perform auxilliary tasks such as exporting a copy of the document in TAIL format.

NOTE: this function is not currently implemented in Arrange 2.0.

```
arFileOpResult Revert();
```

Revert the current document to the last state saved to disk. If the current document is "untitled" and hence has no associated disk file, log an error.

```
Integer GetGlobalData(const char* name, Integer bufLen, OUT void* buffer);
```

Look for a block of global data stored using SetGlobalData under the given name in the current document. If such a block exists, return its length, and copy as many bytes as will fit into the given buffer. Otherwise (if no block is stored under the given name), leave the buffer alone and return -1.

As described in SetGlobalData, a global data block can contain text or binary data. If the block contains text, then we place a zero terminator in the buffer after the last copied character; thus, the actual text will be truncated to dataLen-1 bytes to leave room for the terminator. The function result is the length of the text excluding the zero terminator.

It is legal to call this function with a nil buffer and bufLen = 0; in this case we simply return the length of the data block.

```
void SetGlobalData( const char* name, const void* data, Integer dataLen,
Boolean isText );
```

Store the given data in a global table attached to the current document. This data will be saved in the document file and can subsequently be retrived using GetGlobalData.

Data blocks are identified by their name, which is case-sensitive. Any existing data block under this name is overwritten. If dataLen is zero, then no data will be stored, but any existing block will be removed. The name should never be more than 63 characters long.

If isText is true, then the data is interpreted as a character array, and will be subject to character set translation when the file is opened under a different operating system. If isText is false, the data is interpreted as binary data and will not be subject to translation.

```
Boolean BringCurDocToFront();
```

Force the front window to be a window for the current document. If this is already the case, do nothing; otherwise, if there are any visible document windows for this document, bring one to the front; otherwise (if all of our document windows are hidden), make one visible and bring it to the front. If the current front window contains an unflushable selection, do nothing and return false.

```
void SetChangedFlag();
```

Set the "changed" flag for the current document. This controls whether the Save menu item is enabled, and whether the "do you want to save changes" message is displayed when the user closes the document.

Normally, rather than calling this function directly, you should simply pass “true” for the setChangedFlag parameter to SetupUndo.

```
void SetupUndo(const char* operationName, Boolean setChangedFlag);
```

Mark the beginning of an undoable operation with the given name. That name (prefixed with the string “Undo ” or “Redo ”) will appear in the Edit menu, and choosing it will cause the document to revert to its state at the present moment. If setChangedFlag is true, the document will be marked as having unsaved changes (see the SetChangedFlag function for details). If you pass nil for the name parameter, then Undo will be disabled (i.e. any existing undo state will be thrown away); the setChangedFlag parameter is still respected in this case.

You should generally make this call as late as possible in the execution of a command, just before you start to actually change the state of the document. In particular, you should almost always call FlushSelection before SetupUndo, and not call SetupUndo if FlushSelection returns false.

```
void OpenHiddenDoc( FSSpec* srcFile, Boolean readOnly, OUT arDocumentPtr*
doc, OUT Boolean* closeWhenFinished );
```

Open the given document file. This is similar to the Open call, but it acts invisibly to the user - no document windows will be created. This is used when we need to access a document file as part of some internal batch operation.

A pointer to the document is returned in the "doc" parameter, and the closeWhenFinished flag is set depending on whether the document had been open previously. If the document was already open, doc will be set to the existing document object, and closeWhenFinished will be false. If the document was not already open, doc will be set to a new document object, and closeWhenFinished will be true. In this case, you should call Close after you are finished accessing the document.

## 3.3.10 IOCalls

This section lists callback functions used to import and export data from other applications.

```
Boolean ImportTAIL(const FSSpec* srcFile, arTopicID destTopic);
```

Import the contents of the given TAIL file into destTopic in the current document. If destTopic is nil, import into the topic(s) specified in the TAIL file. Return true if we complete successfully, false if an error occurs (in this case, we will display an error dialog before returning).

```
Boolean ExportTAIL(const FSSpec* targetFile, arTopicID theTopic);
```

Export the contents of the given topic as a TAIL file, replacing any existing file of that name. If theTopic is nil, export the entire contents of the document. Return true if we complete successfully, false if an error occurrs (in this case, we will display an error dialog before returning).

### 3.3.11 UICalls

This section lists callback functions used to alter the Arrange user interface and to intercept user actions.

```
void AddMenu(const char* menuName, Short menuCode, Integer parentCode);
```

Add a new menu to the menu bar (if parentCode is zero) or as a submenu of the menu identified by parentCode (otherwise). The menu will have the given title and code.

```
void AddMenuItem( Short menuCode, const char* itemName, Short cmdChar,
Integer commandCode, Integer itemRefcon );
```

Add an item with the given name, command key (0 if none), command-code, and refcon to the end of the menu with the given code. By default the item will be enabled and unchecked. If itemName is "-", then the item will be disabled and will be drawn as a gray bar. No other Menu Manager metacharacters are supported.

In the Windows menu (menuCode == mWindows), new items are added before the variable part of the menu.

```
void DeleteMenuItem( Short menuCode, Integer commandCode, Integer itemRefcon
);
```

Delete the menu item with the given code and refcon. If no such item exists, log an error.

```
void SetMenuItem( Short menuCode, Integer commandCode, Integer itemRefcon,
const char* itemName, Boolean itemEnabled, Short markChar, Short
itemStyle );
```

Set the title, enabled state, mark character, and style for the menu item with the given code and refcon. If no such item exists, log an error.

If itemName is nil, then the item's title is unchanged.

```
void SetClickHook(ClickHook hook, uInteger refcon, Boolean addHook);
```

Add or remove a ClickHook. The ClickHook will be called for any mouse-down event in a variety of “interesting locations” in an Arrange document or hoist window, as defined by the arClickLocation enum.

When a ClickHook function is called, it is passed the following parameters:

```
ModuleParamBlock* pb; // The standard parameter block.
arClickLocation loc;  // What region the mouse is in.
Point where;          // Mouse position (global coords.)
Short modifiers;      // Event modifiers
Short clickCount;     // 1 for single-click, 2 for
                     // double-click, etc.
arNoteID note;        // The note that was click on.
arFieldID field;      // The specific field that was
                      // clicked on, if any.
arPathID path;        // Path to the note or field.
```

The last three parameters are only used for some arClickLocation values; see the definition of arClickLocation for details. Any unused parameters will be set to nil. Note that when clicking on a topic or folder in the Contents list, or a system object on the Shelf, the “note” parameter is used to pass the ID of the topic, folder, or system object. The “field” parameter is only used when clicking in a field of a specific note in the main part of the window; it is not used when clicking on a field definition on the shelf (the field definition is a system object and so is passed in the “note” parameter).

On entry to a ClickHook function, the current QuickDraw port will be the port for the window that was clicked in.

The hook function should return true if it handled the click, false if it did not. If it returns false, control will pass to the next registered ClickHook function, and eventually to the standard application code (if any).

```
void SetMenuHook( MenuHook hook, uInteger refcon, Boolean addHook, Integer
commandCode );
```

Add or remove a hook from the menu item(s) with the given code. If the code is nil, the hook will be notified whenever any menu item is chosen.

The hook function should return true if it handled the command, false if it did not. If it returns false, control will pass to the next registered hook function for the menu item, and eventually to the standard application code (if any).

```
void SetKeyHook( KeyHook hook, uInteger refcon, Boolean addHook, Short
charFilter, Short keyFilter, Short modFilter );
```

Add or remove a KeyHook. The KeyHook will be called for any key-down or auto-key event which matches the given charFilter, keyFilter, and modFilter.

An event matches the charFilter if charFilter is zero or the event's character code is equal to the charFilter. Thus a charFilter of 32 matches the space key, while a charFilter of 0 matches any key.

An event matches the keyFilter if keyFilter is zero or the event's key code is equal to the keyFilter.

An event matches the modFilter if every bit that is set in the modFilter is also set in the event's modifiers field.

The hook function should return true if it handled the key, false if it did not. If it returns false, control will pass to the next registered hook function for the key, and eventually to the standard application code (if any).

```
void SetFieldHook( FieldHook hook, uInteger refcon, Boolean addHook,
```

```
arFieldID field );
```

Add or remove a hook from the given field in the current document. If the field is nil, the hook will be called for actions in any field of any document.

The hook function should always return false, except when called with an action of faUpdate, in which case it can return true to prevent the new data from being written back to the data layer; or an action of faPopupChoice, in which case it can return false to prevent the pop-up text from being inserted into the field.

The choiceText parameter to a FieldHook function is unused (and will be set to nil) except for the faPopupChoice action, in which case it will point to the text of the selected item in the popup menu.

```
void SetTopicHook(TopicHook hook, uInteger refcon, Boolean addHook);
```

Add or remove a hook to be called whenever the user switches windows or changes the current folder/topic/view in the front window. The refcon parameter is ignored when removing a hook (addHook = false). Log an error if an attempt is made to add a hook twice or remove a hook which isn't present.

```
void SetTickHook(TickHook hook, uInteger refcon, Boolean addHook);
```

Add or remove a hook to be called whenever Arrange receives a null event. The refcon parameter is ignored when removing a hook (addHook = false). Log an error if an attempt is made to add a hook twice or remove a hook which isn't present.

```
void SetFileHook(FileHook hook, uInteger refcon, Boolean addHook);
```

Add or remove a hook to be called at various file transitions. The hook will be called before or after various file-level operations are performed, as specified by the arFileAction enum. The refcon parameter is ignored when removing a hook (addHook = false).

The hook function should always return true, except when called with an action of faClose, in which case it can return false to prevent the file from being closed.

Log an error if an attempt is made to add a hook twice or remove a hook which isn't present.

```
void SetQuitHook(QuitHook hook, uInteger refcon, Boolean addHook);
```

Add or remove a hook to be called when the user quits the application. The hook will be called immediately after asking if the user wishes to save changes in any open documents. If a QuitHook returns false, the quit will be aborted, just as if the user chose "Cancel" in the "do you want to save changes?" dialog.

Note that Quit hooks will not be called if the application exits involuntarily or due to an error condition.

Log an error if an attempt is made to add a hook twice or remove a hook which isn't present.

```
void SetATMHook(AboutToMenuHook hook, uInteger refcon, Boolean addHook);
```

Add or remove a hook to be called just before processing a command key or a click in the menu bar. The refcon parameter is ignored when removing a hook (addHook = false). Log an error if an attempt is made to add a hook twice or remove a hook which isn't present.

AboutToMenuHooks are used to "fix up" menu items to reflect the current state of the world (e.g. the contents of the selection) just before the user gets to see them.

```
void RegisterImportFormat( const char* name, Integer formatID, OSType*
fileTypes, Integer fileTypeCount, ImportHook hook, uInteger refcon );
```

Register an Import format under the given name and ID. If the user chooses this format in the Import format dialog, we will call the ImportHook to import the contents of the file into the current topic.

When the user chooses the Import command in the File menu, we bring up a SFP dialog to select the file to import from. Normally, only Arrange documents and 'TEXT' files are displayed in this dialog. If the given ImportHook can read files of other types, pass those types in the fileTypes and fileTypeCount parameters. Otherwise set fileTypeCount to zero.

If there is already an Import format registered under the given ID, log a debugging error and return.

An ImportHook's function result is defined as follows:

0 = success

positive = error; error dialog handled by the ImportHook

negative = OS error code, Arrange should put up an error dialog

This function should only be called at application boot time, before any documents have been opened.

NOTE: the documentation above describes the situation when an ImportHook is called to actually import a file; in this case the sniff parameter will be false. ImportHooks are also called with sniff = true to decide whether a particular file is in their format. I need to document this more clearly, but for now I'll simply describe the expected function result in this case:

negative = file is definitely not in this format, or an error occurred

zero = file does not appear to be in this format

1-9999 = file appears to be in this format

10000 = file is definitely in this format

```
void RegisterExportFormat( const char* name, Integer formatID, arEFType
type, ExportHook hook, uInteger refcon, OSType fileType, OSType
fileCreator );
```

Register a Save As or Export format under the given name and ID. If the user chooses this format in the Save As or Export dialog, we will create an empty file of the given type and creator and call the ExportHook.

If there is already a format of the given type (Save As or Export) registered under the given ID, log a debugging error and return.

An ExportHook's function result is defined as follows:

0 = success

positive = error; error dialog handled by the ExportHook

negative = OS error code, Arrange should put up an error dialog

This function should only be called at application boot time, before any documents have been opened.

```
void RegisterFieldProc( const char* name, Integer id, arFPType type, Integer
fieldTypes, FieldHook proc, uInteger refcon );
```

Register a field format or action under the given name and ID. When the user selects or deselects this format/action in the Define Field dialog, the hook will be called with one of the messages faSetFormat, faUnsetFormat, faAddAction, or faRemoveAction. It will also be called when opening or closing a document containing fields attached to this format/action, or when creating (e.g. via the Duplicate or TAIL import commands) or destroying such fields.

If there is already a format/action registered under the given ID, log a debugging error and return.

The fieldTypes parameter indicates which types of field the given proc can be used with. For example, if the proc should only be used with text and pop-up fields, pass (1 << arFTText) || (arFTPopup). In the Define Field dialog, this proc will be disabled for any field whose type is not set in fieldTypes.

This function should only be called at application boot time, before any documents have been opened.

```
Boolean TestFieldProc(arFieldID field, arFPType type, Integer procID);
```

Return true if the given field has the given procID attached as a format or action (according to the type parameter).

### 3.3.12 DialogCalls

This section lists callback functions provided to display various standard dialogs used in the Arrange user interface.

```
void StopAlert(const char* message);
```

Display a stop-alert dialog with the given message and a single "OK" button.

```
Boolean OKCancelAlert(const char* message);
```

Display an OK/Cancel alert with the given message. Return true if the user chooses OK, false if they choose Cancel.

```
Boolean PromptForString( const char* prompt, IO char* string, Short maxLen,
Boolean numbersOnly );
```

Display a dialog with the given prompt string, a TextEdit box, an OK button, and a Cancel button. On entry, place the contents of the "string" parameter in the TE box; on exit, copy the contents of the TE box, trimmed to maxLen-1 characters, back into the string. Return true if the user chooses OK, false if they choose Cancel.

If numbersOnly is set, only allow the digit characters (0-9), plus and minus keys, and decimal point to be typed.

```
Boolean PromptForField( const char* prompt, IO arFieldID* field, const
arFieldType* allowedTypes, arChoiceDialogFlags flags );
```

This function displays a dialog box in which the user can select a field by typing its name or choosing it from a menu. The given prompt string is displayed at the top of the dialog. We return true if the user chooses OK to dismiss the dialog, false if they choose Cancel.

The "field" parameter is used on entry to set the default choice in the dialog, and (if the user chooses OK) is set on exit to the chosen field.

If allowedTypes is not nil, it should point to an array of arFieldType values, terminated by a value of arFTNull. Only fields of the listed types will be allowed. If allowedTypes is nil, fields of all types are allowed.

The flags parameter controls whether the user is allowed to select the options "Any Field" (represented by an arFieldID of -1), no field (i.e. typing a blank field name, represented by an arFieldID of 0), and a newly created field (defined from within the choice dialog). Any or all of the flag bits may be set.

```
Boolean PromptForType( const char* prompt, IO arTypeID* type,
arChoiceDialogFlags flags );
```

This function displays a dialog box in which the user can select a note type by typing its name or choosing it from a menu. The given prompt string is displayed at the top of the dialog. We return true if the user chooses OK to dismiss the dialog, false if they choose Cancel.

The "type" parameter is used on entry to set the default choice in the dialog, and (if the user chooses OK) is set on exit to the chosen type.

The flags parameter controls whether the user is allowed to select the options "Any Type" (represented by an arTypeID of -1), no type (i.e. typing a blank type name, represented by an arTypeID of 0), and a newly created type (defined from within the choice dialog). Any or all of the flag bits may be set.

```
void DisplayNotify(const char* message, arNotifyFlags flags);
```

Display a "notify dialog" with the given message text. The dialog will be centered on either the menu-bar monitor, or the monitor containing the front window, according to the forFrontWindow flag. If the showImmediately flag is not set, the dialog won't be displayed until the next call to TickNotify or BackgroundTick more than 1/2 second from now (thus, if the dialog is destroyed within 1/2 second, it will never be displayed at all; this avoids annoying dialogs for operations which take less than 1/2 second to complete).

The notify dialog should eventually be removed by a call to ClearNotify. These calls can *not* be nested; if DisplayNotify is called twice with no intervening call to ClearNotify, an error will be loggeed.

```
void ClearNotify();
```

Remove the notify dialog created by a call to DisplayNotify. If DisplayNotify has not been called, or if ClearNotify has already been called in the meantime, log an error.

```
void SetNotifyText(const char* newMessage);
```

Change the text displayed in the current notify dialog. If there is no current notify dialog (DisplayNotify hasn't been called, or ClearNotify has been called since), log an error.

```
Boolean TickNotify(Integer amountDone, Integer totalAmount);
```

This function should be called periodically between calls to DisplayNotify and ClearNotify. It checks to see whether the user has pressed command- period to abort the operation, and returns false if so, true otherwise. It also causes the notify window to be displayed if the showImmediately flag was set to false and 1/2 second has elapsed since the call to DisplayNotify.

If the dialog contains a progress indicator (the useThermometer flag was set), and you wish to update the progress indicator, pass in the amount of the job that has been completed as the fraction amountDone/totalAmount; totalAmount must be greater than zero. Otherwise (there is no progress indicator, or you don't have a new value to display) pass 0 for totalAmount; amountDone will be ignored.

Note that this function will in turn call BackgroundTick (below).

```
void BackgroundTick();
```

This function should be called periodically during any long operation. It gives small amounts of time to background applications, and performs other background tasks as appropriate

### 3.3.13 ModMgrCalls

This section lists callback functions provided to load, unload, and access other modules. Most plugins will not need to use the functions in this section.

```
ModuleID FindModuleName(const char* name, uInteger minVers);
```

Search for a loaded module with the given name (taken from the name field of the ModuleDef resource). Name comparisons are case-sensitive. If there is a loaded module with this name, return its current ModuleID; otherwise return 0.

```
ModuleID FindModuleID(uInteger id, uInteger minVers);
```

Search for a loaded module with the given unique ID (taken from the id field of the ModuleDef resource). If there is a loaded module with this id, return its ModuleID; otherwise return 0.

```
const ModuleInfo* GetModInfo(ModuleID module);
```

Return a pointer to the ModuleInfo record for the given loaded module. Signal an error if the given ID is not the ID of a currently loaded module. The pointer returned is valid until the next Module Manager call.

```
OSErr OpenModFile(const FSSpec *file, Short fileRefnum);
```

Open the given resource file and open all modules inside it. The modules (if any) will not actually be loaded until the next call to LoadOpenModules. If you have already opened the resource fork for this file, pass in its refnum; otherwise pass -1, in which case we open the resource fork ourselves. In the latter case (when OpenModFile must open the resource fork itself), if the file does not contain any module definition resources, we close the resource fork again.

If an error occurs, this function returns the (negative) OS error code. Otherwise, it returns the (non-negative) number of modules which were opened. If the result is zero, and you opened the resource file yourself, you might wish to close it again (since it didn't contain any usable modules).

```
ModuleEntryPoint GetModuleEntry(ModuleID module, uInteger entryID);
```

Return a ModuleEntryPoint record for an entry point in the given module. The entryID parameter is forwarded to the module's root function. Signal an error if the ModuleID is invalid or the module has no code resource.

If the module is valid but does not implement the desired entry point (its root function returns a nil address in response to the mrFindEntry call), return a record with a nil ModuleID.

```
ProcPtr ModuleCallPrelude( ModuleEntryPoint entry, OUT ModuleParamBlock
*pb );
```

This function is used to call a routine in a plugin module. The entry parameter should be a valid entry point record for the module, obtained by calling GetModuleEntry. On exit, all fields of the ModuleParamBlock record will be initialized, and the function result will be the address at which to call the module's code. Note that this function sets the hookRefcon field of the ModuleParamBlock to zero; if a different value is desired, you must insert it before calling the module function itself. Also note that the ModuleParamBlock record must be the exact block of storage you pass to the module function itself - problems will occur if you make a copy of the record and call the module function with the copy.

To use this function, you should first do any work necessary to prepare the parameters you intend to pass to the module routine. Then call ModuleCallPrelude, followed immediately by the function pointer it returns, followed immediately be ModuleCallEpilogue. You *must* call the function pointer and ModuleCallEpilogue, and should not make any other function calls in between.

After calling the function pointer, discard it - if you want to call the function again, you must make another call to ModuleCallPrelude.

The way to call a plugin module is as follows:

```
  ModuleEntryPoint entryPoint = GetSomeEntryPoint();
   Integer extraParam = CalculateSomeValue();

   ModuleParamBlock pb;
   ProcPtr procAddr = ModuleCallPrelude(entryPoint, pb);
   // optionally fill in pb.hookRefcon

   Integer functionResult = SomeProcPointerType(procAddr)(&pb,
extraParam);

   ModuleCallEpilogue(entryPoint, pb);

   // procAddr should no longer be used
```

ModuleCallPrelude / ModuleCallEpilogue use an internal stack which can hold 64 entries. If module calls are nested more than 64 levels deep, the application will abort with a fatal error.

```
void ModuleCallEpilogue( ModuleEntryPoint entry, const ModuleParamBlock
*pb );
```

This function should be called immediately after calling a plugin module using a ProcPtr taken from ModuleCallPrelude. See ModuleCallPrelude for details.

## 3.3.14 MemCalls

This section lists callback functions used to allocate, deallocate, and manage memory.

```
void* AllocMem(uInteger size, AllocMode mode);
```

Allocate a block of free store memory, a heap pointer, or a heap handle, according to the mode parameter. If the amClear flag is set, initialize the storage to zeros. If we are unable to allocate the requested storage, we return nil unless the amErrIfNoMem flag is set, in which case we display an error dialog and terminate the application.

NOTE: the amFriendly flag is not currently implemented (we always act as if it were false).

NOTE: the amErrIfNoMem flag is not currently implemented for amFreeStore allocation - it is always

treated as true. That is, if an amFreeStore allocation fails, we display an error dialog and terminate the application even if amErrIfNoMem is not set. This will be fixed at some future time, so if you can't handle an allocation failure, be sure to set the amErrIfNoMem flag for amFreeStore calls.

```
void DeallocMem(void* block, AllocMode mode);
```

Release a block of memory which was allocated using CBAllocMem. The mode parameter must reflect the type of storage allocated (free store block, heap pointer, or heap handle). The flag bits (friendly, clear, and errIfNoMem) are ignored.

```
uInteger MemAvailable();
```

Return the amount of memory which is available for "friendly" allocation. This is equal to the amount of free space on the heap, plus the amount of memory which can be released by our grow-zone function, minus a small "buffer zone" which we try to keep free for minor uses. The result of this function is only approximate, and at the moment does not take into account any purgable resources in the heap.

### 3.3.15 UtilCalls

This section lists some miscellaneous utility functions.

```
const char* GetRString(Short rsrcID, Short stringIndex);
```

Look up the string at the given (0-based) index in the STR# resource with the given ID. Return the string in C format (i.e. zero terminated).

If the given resource does not exist, or stringIndex is out of bounds, log an error and return an empty string.

The string which is returned is stored in a 1K circular buffer maintained internally by Arrange. It will therefore remain valid for “a little while”; you should make your own copy if you will be using the string for any length of time (i.e. for anything more than a few local function calls). It is strictly illegal to write to the pointer returned by this function.

Note that the stringIndex parameter is 0 based, and so should range from 0 to 1 less than the number of strings in the resource. This is different than the toolbox routine GetIndString, which takes a 1-based index.

```
void LowerString(IO char* string, Integer length);
```

Convert the given string to all lower case. This function is reasonably efficient, and correctly supports all Roman scripts.

```
void PrintToLog(const char* message);
```
Print the given (zero-terminated) string to the error log.