Writeswell Jr.™ Code Notes

Michael D. Crawford
Working Software, Inc.
AppleLink D1620
Internet 76004.2072@compuserve.com

Contents

Introduction

This is a brief overview of the source code to the Writeswell Jr. program that is used to demonstrate the word-processor side of the Word Services Suite. Writeswell Jr. is a simple TextEdit based text editor. It is Apple Event aware, though, and can call a spelling checker, grammar checker, hyphenator, or other "word services" via the Word Services protocol.

This sample code is believed to match the Word Services Suite specification. Previous versions of the Word Services SDK had preliminary drafts of the specification and code that are no longer valid. Be sure that you are working with up-to-date code before actually putting this in your application. If you are not sure, contact me to get an update. I do try to keep the SDK current as I and other developers find bugs. The current code is quite robust as a result of extensive testing by many different people.

Running the Example Programs

Writeswell Jr. can be run off of a CD since it creates a preference file on your hard disk. IAC Spell Test will need to be on your hard disk since it saves its dictionary location in its own resource file.

You must install a debugger such as Macsbug when running Writeswell Jr., and the IAC Spell Test speller. If there is an error, the applications call the debugger with a helpful error message. If no debugger is installed, you will get an "Unimplemented A-Trap" System Error. If an error is returned from a trap call, the command "D7.W" will usually display the result code in Macsbug, and the command "g" will continue execution.

Writeswell Jr.™ Code Notes

How it Works

The program starts in main in TestBed.c. Main initialized the managers, calls functions to install the various handlers, and starts up the event loop.

There are two sets of Apple Event handlers. The first set, in MyHandlers.c, handle the required Apple Events. These are the OAPP, ODOC, PDOC, and QUIT handlers from the kCoreEventClass. The one notably useful code is MyOAPPHandler, which can open a file from disk. You can open a file at any time by dragging its icon onto Writeswell Jr.'s icon, as long as Writeswell Jr. has no windows open (it can only handle a single window). The handler calls MyOpenFile in MyFiles.c, which calls MakeNewWindow to create a window (with a TextEdit handle stuffed into the window's refCon), and then reads the text from the file into the TextEdit field.

The Core suite events are handled by a single "wildcard" handler, GenericHandler in GenHandlers.c, using an "inverted" method suggested to me by Richard Clark of Developer University. GenericHandler extracts the direct object from the event and calls AEResolve to identify the object that is specified by the event, and then calls a dispatcher for the given object type. This allows me to neatly separate all the code for each object class into different files for easy maintenance. There aren't many classes here yet, but I think this makes much more sense than having each event handler handle the events for each class.

(One can be easily confused by the "kCoreEventClass" constant for the required event class, and the "kAECoreSuite" constant for the Core Suite.)

The classes I support are cApplication, cWindow, cText, and typeObjectSpecifier. You may get and set a window's title, and you may count the cText and the typeObjectSpecifier elements of the window. Note that I use a coercion handler to convert typeChar data into "typePString" data in order to set the window title; this is a type that would be good to have defined in the Core suite. The coercion handler is TextPtrToPString in ObText.c. Note that the window code does not "know" about TextPtrToPString; it just asks for the data type, and gets it because the coercion handler is present.

I use a trick in my token data structures. A typical token body is as follows:

```
typedef struct {
   TEHandle    textH;
   short       startPos;   /* Short cuz TE only handles 32k o' text! */
```

```
   short       length;
   DescType    propertyCode;
} TETextTokenBody;
```

If my object accessors are given a key form of formPropertyID, the ID is placed in propertyCode; if they are not, then the desired data is the object itself (the window, the text, etc., rather than the title of the window or the font of the text), and I used a propertyCode of 'null'. Here I make the assumption that 'null' will never be a property; this may not be a valid assumption. The trick is handy in my event handlers; I just switch off the property code to determine what to do:

```
OSErr TETextSetDataHandler(... )...
switch ( propCode ){
   case typeNull:
         ...
         TEInsert( (Ptr)(*textValue.dataHandle), newTextLen, textH );
         break;
   case propBackgroundHilite:
         TESetSelect( (*tokHdl)->startPos,
                     (*tokHdl)->startPos + (*tokHdl)->length,
                     textH );
         TEActivate( textH );
         break;
   default:
               return errAENoSuchObject;
               break;
}
```

Though the Core Suite specifies a single cText class, the text in your application may be of different types - editable text in windows, non-editable text in menu titles or menu items and so on. You will need a number of different token types for your text to reflect the different kinds of text that you may have.

The key to supporting the Word Services suite is to support formRange and end-of-container formAbsolutePosition key forms. Specifying a position relative to the end of the container allows the text to be changed by several successive Set Data events without having to do extra work to calculate new offsets. One can tell that a formAbsolutePosition key is relative to the end of the container because it will be negative; -1 is the last object in the container. See the code in CharFromTEText in ObText.c.

The Object Support Library documentation is unclear about how to resolve formRange

specifiers. Note that to specify a range of characters, we use a formRange object specifier that consists of two formAbsolutePosition specifiers, which specify a single character at the beginning and end of the range. When an object accessor recieves a formRange key, the selectionData is a descriptor of type 'rang'. You can coerce this to typeAERecord, and then use AEGetKeyDesc to extract the keyAERangeStart and keyAERangeStop object specifiers. If you don't coerce to typeAERecord, the call to AEGetKeyDesc will fail - it does not recognize the the 'rang' data type is really an AERecord. You must then call AEResolve on each of the two specifiers to get the beginning and end of the range. Note that this is a recursive call - your object accessor has already been called by AEResolve; thus object accessors must be reentrant - they must not change global variables.

Following is the formRange code from CharFromTEText:

```
case formRange:

   err = AECoerceDesc( selectionData, typeAERecord, &rangeRecord );
   if ( err ) return err;

   err = AEGetKeyDesc( &rangeRecord, keyAERangeStart,
                           typeObjectSpecifier, &startSpec );
   if ( err ) return err;

   err = AEGetKeyDesc( &rangeRecord, keyAERangeStop,
                     typeObjectSpecifier, &endSpec );
   if ( err ) return err;

   err = AEResolve( &startSpec, kAEIDoMinimum, &startToken );
   if ( err ) return err;

   err = AEResolve( &endSpec, kAEIDoMinimum, &endToken );
   if ( err ) return err;
(... now make the token from the beginning and end tokens...)
```

The Services Menu

Word Services provides for a simple way to register new services with applications. Launch the speller (IAC Spell Test in this case). Launch Writeswell Jr. and select “New Batch Service” from the Services menu. Select “IAC Spell Test” from the PPCBrowser display.

Writeswell Jr. sends a Get Data event to request the “pBatchMenuString” property from

IAC Spell Test, and another Get Data to request the "pLocation" property. The menu string is saved in the Writeswell Jr. Preferences file, and added to the Services menu. The pLocation is an alias record. IAC Spell Test places its creator code in the userType field of the alias. This is important - the alias manager puts 0 in the userType - the value placed there is up to the application; the Word Services spec requires the actual signature of the speller, so that it is easily accessible to the word processor.

The preferences file also contains a record that records the type of service for each menu item - either batch service, interactive service, or no service. (Interactive service is not yet implemented). When the menu is built, a global array, gServItemID, stores the resource ID for each service menu item. When the service is selected from the menu, OpenSpeller in DoChecking.c looks up the resource ID in gServItemID. The alias record is read in. OpenSpeller calls FindAProcess to see if a process is running with that creator code. If it is not, then OpenSpeller calls LaunchSpeller to launch the speller application from the alias, and then returns a typeApplSignature descriptor for use in the AESend call to send the batch event.

You can simulate the presence of multiple Word Service programs on your hard disk. Make several copies of the "IAC Spell Test". Use ResEdit to give each copy a unique creator code. Change the 'STR#' 1300 resource to give each one a unique menu string. Then launch each one, and install it in Writeswell Jr.'s menu (or your own application's menu!) Each copy of the speller will need its own dictionary if they are to run all at the same time.

The code that services the Get Data event for the pLocation property of IAC Spell Test actually checks to see what its own creator code is. Normally, you just use a constant, but I wanted to allow for the simulated multiple servers.

Initiating a session

You may start a Word Services session by selecting "Check Spelling with IAC Spell Test" from the services menu, after installing the menu item as described above.
Selecting a service results in a call to DoSpellCheck in DoChecking.c. DoSpellCheck calls OpenSpeller to look for the server, launch it if necessary, and obtain its address as a "typeApplSignature" descriptor. Then DoSpellCheck calls either DoBatchCheck or DoBatchTableCheck to send a "Batch Process My Text" Apple Event to the server. The batch event contains a parameter that specifies what text is to be checked. The speller uses this parameter in subsequent Get Data and Set Data events to read and replace the text, and to set the background hilighting.

There are two ways that the text may be specified in the batch event. You can choose which is used by selection "Options…" from the Writeswell Jr. Edit menu.

The “Send text specifiers” option makes Writeswell Jr. send the object specifiers for the text explicitly. The direct object to the batch event is a list (a descriptor of typeAEList) which contains a single element, which is an object specifier for the first (and only) text field in the frontmost window.

Client programs that allow more than one text block, such as drawing programs, spreadsheets, and databases, may have several blocks checked at once by sending an object specifier to each block in the list.

This is easy to do and works well if there are not too many text blocks to be checked. It will not work if there are many text blocks, since an Apple Event may not contain more than 64K of data.

The “Send table specifier” option sends an object specifier for a table instead. The direct object is a descriptor of typeObjectSpecifier. The server will use the table specifier as a container to ask for elements of typeObjectSpecifier. That is, the server will ask for “object specifier number 27, which is contained within the first window of the application.” The object specifier that is gotten from the table is used in just the same manner as the object specifiers that are contained in the list from the “Send text specifiers” method.

A client application may choose to user either method. The method used will be invisible to the user - I allow the option here for so programmers may test either way. A server application **must** support **both** methods.

The server uses the descriptor type of the batch event’s direct object to determine which method to use. If the direct object is typeAEList, then the server expects the direct object to contain a list of object specifiers. If the direct object is typeObjectSpecifier, then the server expects that the direct object may be used access the elements of a table that is kept within the client.

In the Writeswell Jr. code, the table is just the document window. The elements of the window that are of typeObjectSpecifier are object specifiers to the text fields in the window that are to be spellchecked. (There’s only one text field, but there could be more in general). If one uses Get Data to get the first text element in the window, the text will be returned. If one uses Get Data to get the first object specifier element in the window, then an object specifier will be returned. This object specifier refers to the text element.

If you get this far, your head must be swimming. Go have a cup of tea and relax for a while. You have read the words “object specifier” entirely too many times.

It is up to you what you want to use for a container. I chose to use a window, but you could use the application (or null) container, or you could use some other object. The server does not examine the table specifier itself; it just uses it as container to get an element from.

Helpful Hints

Here are hints based on the experience of the other developers that have implemented the protocol in their own applications.

Be sure you have the SIZE resource correct. You will need the High Level Event Aware and the Can Background bits set. There is some possibility that you may need to change the logic of your program slightly if you have never used background processing before.

The AECreateAppleEvent function calls Random if you pass the kAutoGenerateReturnID constant for the return ID parameter. Random uses the random seed that is stored in the QuickDraw globals. This means that register A5 must be valid, or a crash might happen. The call to Random will be unlikely to crash immediately; instead, a crash will happen later because Random will place a new value into what it thinks is randSeed, overwriting some part of memory.

It is easy to neglect to dispose of all of your descriptors. This will cause a memory leak which may be serious if the leak occurs in code that is called frequently. The Leaks dcmd, available on the Apple Developer CD, is very useful for finding these.

Particularly watch out for functions that create a descriptor and return it to other functions that keep it around for a while. The token returned by AEResolve must be disposed of when you are done servicing the Apple Event.

The reply event passed to AESend must be disposed of whether you actually use it or not. Dispose of after the AESend if you are not using queued replies. If you are queing replies, dispose of it after the reply is actually recived.

Word Services is a stateless protocol. Once a client application sends the Batch Process My Text event to the speller, it continues its event loop as if nothing special was happening. There is no event to announce the termination of a Word Services session. If the client application really needs to know that the session has terminated, it can watch for suspend/resume events. The session is over when the client application returns to the front.

Future Directions

The demo code now substantially matches the current protocol specification. There are some extensions to the protocol that may be made in the future if there is sufficient interest from developers. Any such extensions will be upward compatible – old servers will work with new clients, and new servers will work with old clients.

The menu strings will be elements of the application rather than properties. There will be an optional parameter added to the batch event to specify which service is desired. This will allow a single server application to provide multiple services, each with their own menu string.

There are discussions under way on how to allow Word Services to support servers that use modeless dialogs. This will mainly be of use to grammar checkers, which will be able to tell the user to rephrase some text back in the original document, and then continue where they left off when the user selects the grammar checker's window. This will be an optional capability, and will require the client to maintain some state information during a Word Services session.