The Word Services Apple Event Suite

Michael D. Crawford
Product Development Manager
Working Software, Inc.

Abstract

The Word Services Apple Event Suite allows any application to link to a speller, grammar checker or other text service as if it was a built-in menu item. Working Software developed the protocol in cooperation with our competitors in the spelling business, as well as several grammar checker, database and word processor publishers, and Apple Computer, Inc. It is probably the simplest useful thing that a developer can do with Apple Events.

Word Services is designed to be very easy for client programs to implement – the client (or word processor) needs only to send a single Apple Event to a server program (or spellchecker) and then resume its event loop. The server takes over control of the protocol, retrieving and changing text in the client program's document by using a small subset of Core Suite Apple Events. The complete source code to Writeswell Jr. will be distributed along with the protocol specification. Writeswell Jr. is a simple word processor that supports Word Services, and is provided as an example that developers may use to add support for the protocol to their own applications.

Word Services applications have been shipping for over a year and a half. Spellers are available in a number of languages, and client programs ranging from a curriculum planner with a total of ten users to a major word processor have adopted the protocol. I will reflect on the long process of developing and promoting an industry standard protocol.

I. Introduction

The Word Services Apple Event Suite allows any application to link to a speller, grammar checker or other text service as if it is a built–in menu item. I will tell you how Word Services works and why you should support the protocol in your own applications. I will give you some examples of existing, shipping implementations. Now that Word Services has been in the market for over a year and a half, I will discuss what I have learned from developing and promoting the protocol. I have the hope, in doing so, that others will benefit from our difficult, though personally satisfying experience.

When System 7 was released at the 1991 World Wide Developer's Conference it was my first WWDC on my first job as a Macintosh developer. System 7 provides us developers with many new opportunities, with many new tools for application development, but in 1991 System 7 still required hard work from developers to be complete: hard work to develop the Apple Event protocols to allow programs to work together, and to write the programs that will use the protocols.

I came to the conference with the task of developing a spellchecking protocol. I meant to meet the other spellchecker vendors and start working with them. I soon discovered that several grammar checker vendors had the same idea, and several word processor, database and spreadsheet developers wished to work with us as well. The essential features of the protocol were hashed out over a sushi dinner during the developer's conference, hosted by my employer, Dave Johnson. This was a fine experience: competitors out for a night on the town, working together towards a common purpose.

We decided the bulk of the effort to implement the protocol must be the responsibility of the “server” application – that is, the speller or grammar checker developer should do the hardest work. The protocol should be easy for “client” application developers to implement. This is required for widespread adoption by word processor, database and communications program developers. These developers wish to focus their attention on the central purpose of their programs. They do not wish to devote expensive resources or time–consuming effort to adding a feature that is not essential to their product.

This decision has been a successful one. The amount of time required to implement Word Services in a client program has ranged from as little as two hours to a maximum of two weeks.

I imposed a second requirement on Word Services: the protocol had to be a practical one, that got finished in a reasonably short amount of time, and got real products into the market that customers would actually purchase and use. There were many requests to add features to Word Services to suit some particular application, but I generally denied these requests unless they fit this criterion. I feel this is why our protocol is succeeding where others have already failed.

Why should you support Word Services in your application? If you are a client application developer, the protocol:

• allows you to add features such as spellchecking without debugging someone else’s OEM code.

• allows you to provide for these features for free. We even give away free example source code.

• allows your users to save disk space. They no longer need several different spelling dictionaries, a different one for each program.

• provides your users with a single user interface among several different programs. The speller will appear the same no matter what application it is used in.

• allows your users to pick and choose their favorite services, in the language that they wish to use.

If you are a server program developer, the protocol:

• allows you to work with new applications, and new document formats, without any further work on your part. Speller developers no longer need update their programs to support new file formats.

• allows you to keep your source code private.

The protocol is public. No license fee and no nondisclosure agreement is required to use the protocol, or to use the sample code provided in the Word Services Software Development Kit. The Word Services SDK is available free from Working Software, Inc.

While several server applications have been developed independently, Working Software considers the internal operation of our Spellswell 7 application a proprietary trade secret. This means that other speller publishers must do the same hard work we did to write a Word Services server program. I will concentrate on the client side of the protocol in this paper. It is not impossible to write a server application – it merely requires more work than a client application.

II. How it Works

I will give a simple description of Word Services. Do not let the protocol specification document scare you: the entire Word Services specification is long and detailed, and really describes four closely related protocols: the server–interface Word Services protocol, the client–interface protocol, the batch protocol and the interactive protocol.[1]

So far no one has implemented the interactive Word Services protocol. I will not discuss it further here, except to say that we will implement it only if we receive a commitment to support it from a major client program vendor.

A form of the client–interface protocol has been implemented by the Communicate! terminal emulator from Mark/Space Softworks. It is especially suited for terminal emulators because they have read–only text: it makes no sense to use the Set Data event to change text that has scrolled by on the screen, or is being edited by a mainframe text editor such as **vi** or **emacs**.

We are concerned mainly with the "server interface batch mode" protocol. This means that the user interface, for example the "Skip or Replace" dialog for a speller, is provided by the server application. By "batch mode" we mean that the text is checked in bulk when the user requests it, rather than continuously as she types.

This is the essence of Word Services: send a request for batch processing to the server. Include a list of text blocks with the request. Resume your event loop. The server requests the contents of each text block, then changes strings of characters in each text block. When it is done, the server may quit, or may stay running and require the user to bring the client application to the front.

There are some issues to be understood, such as how to find the server initially, and how to create the list of text blocks. There are optional features, such as the use of background highlighting of erroneous text, and there are two different ways that the list of text blocks may be provided, but this explains the most important components of the protocol.

Object Specifiers

We must clearly understand "object specifiers."

Object specifiers are used in the Apple Object Model to denote, or to point to data items within a program. They serve the function that pointers do in C or Pascal, and the function that file pathnames do in file systems: they let a program know what data is the target of some operation.

Object specifiers are heirarchical constructions. They are thus more akin to file pathnames than they are to pointers. Because we know where the data is in some structural space, rather than where it is in some absolute address space, we can denote the data by giving a heirarchy that locates the data within the program. I say "a" heirarchy rather than "the" heirarchy because there usually is more than one heirarchy that will work. One chooses a heirarchy out of convenience, or to suit a particular problem.

Examples of such heirarchies are:

```
the second text block
   of the window named “foo”
      of the application.
```

and

```
the third text block from the end
   of the frontmost window
      of the application.
```

When an application receives an object specifier, it passes the specifier to the AEResolve function, which converts the heirarchical description into a "token" for the object. This changes a "name" for an object to a "pointer" to the object. Object specifiers give locations as a user might view them, with little or no regard to the format that the data is stored in. The token generally contains the address of the object, and the structure of the token object itself is not specified by the Apple Event Manager, as it must contain intimate knowledge of the memory format of your data.

When a client application requests batch processing, the list of text items that are sent is a list of object specifiers; it is not the text itself. That is, we do not say "spellcheck this text," rather we say "spellcheck the blocks of text that you can find here, and here, and here."

After a server receives a batch request, it picks out the first object specifier in the list and sends the client a Get Data event to get the first block of text. If the server is a spellchecker, and the user chooses to replace a word, the server creates a new object specifier which uses this first object specifier as a container. It might look something like this:

```
characters 30 through 25 from the end
    of the second text block
        of the window named “foo”
            of the application.
```

The server then uses this object specifier in a Set Data event to replace the text of the erroneous word.

Why do we say “from the end?” Why do we say “the second text block” in our example when this is the first text block in the list?

The list of text blocks does not necessarily give each text block in the entire document; instead it gives the specifiers for each text block that the user actually wishes to process. How this is done is left up to the client application developer, but I suggest that you process the entire document if there is no user selection, and process each paragraph that contains such a selection if one exists. This allows the user to choose different blocks of text to be submitted to different servers, or to spellcheck only the text that has been recently edited.

We give our ranges from the end of the text block because the length of the block may change when we correct the text. For example, if the correct the sentence “The redd dogg barks,” we will alter the length after we change “redd” but before we change “dogg.” A subsequent correction has the risk of changing the wrong characters.

A server could keep track of the offsets, but it is simpler to use object specifiers that are relative to the end of the container. This are given as negative integers; character -5 is the fifth from the end.

Finally, we must understand the formRange type of object specifier. We can use any of several different “key forms” at each level of our heirarchy. For example, formAbsolutePosition gives a position as a count from the beginning or end. FormName specifies an object by its name: “window named Foo.” A formRange specifier uses two object specifiers recursively; one is a specifier for the beginning of the range, the other for the end of the range.

Because of this recursive use of object specifiers for character ranges, your “object resolution” functions must call AEResolve recursively when they find a formRange specifier. Examples of this are given in the Word Services Suite specification, and in the Word Services SDK.

The clearest technique I have seen for handling Apple Events is the "Object First Approach" described by Richard Clark.[2]

Registering a Service

How do we start up a Word Services connection? We must register each service with each client application.

It would be nice to have a systemwide registry, but at the time Word Services was developed I could see no practical way to handle this. It would be best to do it within the Macintosh System software, but Apple’s engineer’s were busy doing other things. Since we decided to use the way that would get to market, rather than the most aesthetically pleasing way, we chose to leave the registration up to each client developer.

(Of course, sample code is given in the Word Services SDK. You can lift it right out of Writeswell Jr. if you like.)

There are four steps to register a service:

• locate the service,

• ask the service for its icon, location alias and menu string,

• save these items in your preferences file,

• display the icon and menu string in your menu.

When a user selects the menu item from your menu, launch the server using the location alias, and send it a “Batch Process My Text” event.

There are two ways to locate the service. You may use the PPCBrowser as Writeswell Jr. does, or you may use the Open dialog box from the Standard File package. There are advantages and disadvantages to each method. SFGetFile is already familiar to the users and does not require the server to be running already – the server is launched by the client application after it is selected. The PPCBrowser does not require the users to navigate the file system via a complicated dialog, and only displays applications that are Apple Event aware. Unfortunately there is no simple way to show only Word Services applications.

Once the server is located, use the Get Data event to ask it for its pBatchMenuString property. This is a string suitable for display in the client's menu that describes the service to be performed, such as "Check Spelling." Also ask the speller for its pMenuIcon, to display in the menu as well. This prevents the user from being confused by two servers that provide the same kind of service, and also gives the client application nice "System 7" styling.

These must be saved in your preferences file, along with the location alias that you may ask the server to supply as well.

When you start up, scan your preferences file for these saved menu strings and display them in your menu. When the user selects one of these menu items, check to see if the server is already running. If it is not, use the saved alias to launch the server, then send it a "Batch Process My Text" event, with an AEList containing object specifiers for each text block you wish to check.

Be sure that you use an AEList, even if there is only one text block. For example, if you wish to spellcheck your entire document as a single block, you must create an AEList containing a single object specifier. This is different from the usual practice of placing a single object specifier in the direct object of an event.

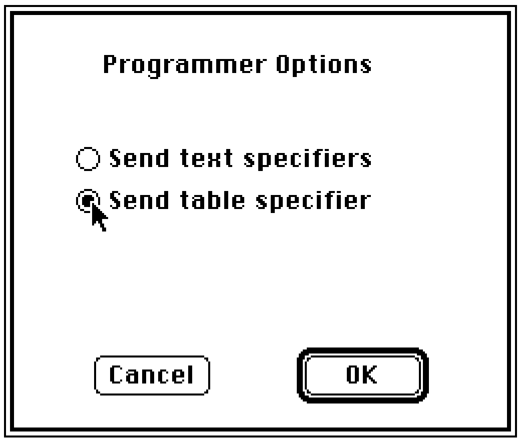
Send Table Specifier vs. Send Block Specifiers

Why do we require an AEList? In order to allow for checking large structured documents such as databases, we provide an optional method of specifying large numbers of text blocks. I call the method I have already described the "send block specifiers" method. This is troublesome if there are large numbers of text blocks in the document, as there will be for a database or spreadsheet: Apple Events may contain no more than 64K of data, and each object specifier may be 200 to 300 bytes. A database easily exceeds the limit of 200 or so text blocks that this imposes. The memory of the client, the system, or the server may be exceeded as well.

To resolve this problem we have the "send table specifier" method of requesting service. Instead of explicitly sending a list of text blocks, the client may send a single object specifier to a table that is maintained by the client. The elements of this table are also object specifiers. Each object specifier describes a single text block to be checked.

One may not need to maintain this table as real object specifiers. One can use any format for the table elements and convert them to object specifiers as they are requested.

To illustrate this concept, the Programmer Options dialog in Writeswell Jr. allows you to use either method to request service. Look at the sample code that refers to this preference to see what it does.


Programmer Options

○ Send text specifiers

[✓ Send table specifier]

Figure 1: The Programmer Options Dialog in Writeswell Jr.

How does a server know which option is being used? The server examines the data type of the direct object to the batch event. If it is typeAEList then the Send Text Specifiers method is used. If the direct object is of typeObjectSpecifier then the Send Table Specifier method is used: the server uses Get Data to get a table element, then retrieves the text for that element.

Highlighting Erroneous Words

Traditional OEM spellers highlight erroneous words in the original document as they are presented to the user. This is provided for by Word Services, but it is an optional feature. I highly recommend that you support highlighting.

The speller requests that the client highlight some text by setting the pBackgroundHilite property on a range of characters to True. If the client supports this operation, it returns noErr as the result of the Set Data event. If it does not support background highlighting, the client returns an error code.

A client might choose not to support background highlighting because it requires displaying a selection while the application is in the background. Many applications are hardwired to turn off their selections while in the background. If it is possible to implement this property, then do so, but your application will work as a Word Services client if you do not support background highlighting.

If the server detects that the client does not support background highlighting, it is the server's responsibility to display the erroneous word itself, with some surrounding text to supply meaningful context to the user.

III. Existing Implementations

A number of shipping applications now support Word Services. Several more are under development. Among the shipping applications are:

Writeswell Jr.

Writeswell Jr. is a simple "TeachText–like" word processor with a Services menu:

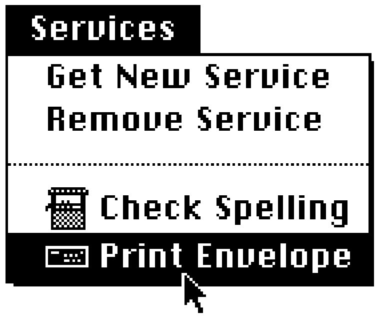


Figure 2: Writeswell Jr.'s Services Menu

Working Software gives away Writeswell Jr. for free as a demonstration of Word Services. We supply the source code on the Word Services SDK to aid developers in writing Word Services applications.

Info Depot

Info Depot, formerly Fair Witness, from Chena Corporation was the first commercial Word Services client. Jim Kaslik, President of Chena, was extraordinarily helpful in reviewing the protocol and in implementing it while the protocol was still being revised. He was quite good natured about changing his product to meet the changing specifications.

When developing a protocol it is important that someone else implement it independently. Early releases of the Word Services SDK had "compensating bugs" in Writeswell Jr. and the development speller. It was not until Jim wrote the code for his product that we discovered that a bug in the speller was compensated by a bug in Writeswell Jr., so that they both worked together but were both slightly off the Apple Events specification.

Info Depot, an "information spreadsheet" has gone on to support a rich implementation of AppleScript.

Eudora

The Eudora electronic mail program from QUALCOMM Inc. implements Word Services in the commercial, 2.0 release. Previous shareware releases support AppleScript but not Word Services. Word Services is one of the distinguishing features of the commercial version. We have just signed an agreement to bundle Spellswell 7 with Eudora.

It has a form of the "selection" checking, in that it will check the header of a mail message if the cursor is on the header; otherwise it will check the body of the mail message.

WordPerfect

WordPerfect from WordPerfect Corporation has a nice placement of the registration within its Preferences dialog (Figure 3).

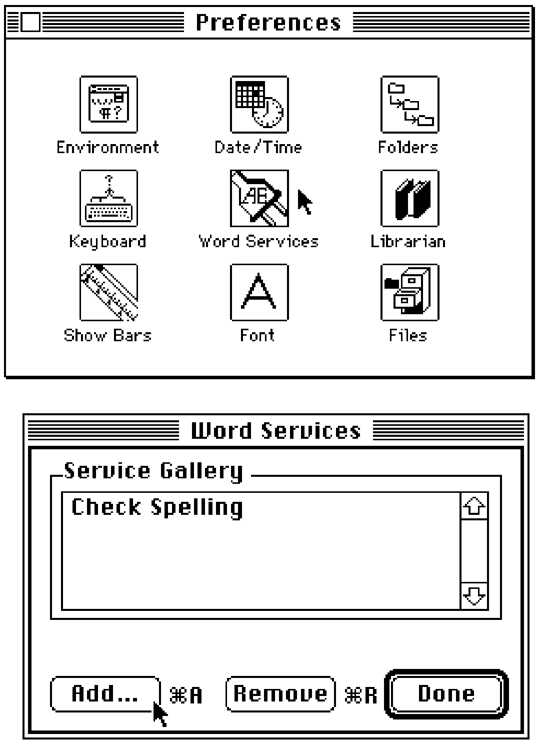


Figure 3: The WordPerfect Preferences Dialog

We worked quite closely with engineers from WordPerfect throughout the early development of Word Services, with extensive discussions about the possibility of harmonizing the Word Services Suite with the Writing Tools API, a cross–platform programming library that is philosophically similar to Word Services.

Unfortunately we lost touch with the engineer that we worked with when he was assigned to some other work on the 3.0 release of WordPerfect. While Word Services is supported by the product and is well–documented in the manual, their implementation actually works quite poorly. It is very slow, and the background highlighting does not work unless one forces updates by dragging the speller window around.

We have the hope that they will take interest in the protocol again and do the programming required to make it work well.

Omnis 7

Omnis 7, a database development program from Blyth Software of England, was also an early adopter of Word Services. It supports the protocol in a minimal way, in that it requires the user to select a service from the PPC Browser each time Word Services is used.

A Curriculum Planning Program

The University of Michigan has developed a curriculum planning program which is used by ten teachers. We found out about its existence *after* they had implemented Word Services, when they ordered ten copies of Spellswell 7.

While we have not made much money off of this program, it is interesting in that it shows that Word Services benefits small, vertical application developers by providing a way to spellcheck documents without paying large license fees. It also shows that the work required is small enough to be worthwhile to a small developer. I personally believe that we will make the most money by selling small numbers of spellers to many small developers, rather than selling many spellers to a single large developer.

World Write

World Write, from World Software in France, is sold largely in Eastern Europe. This Script Manager compatible word processor is bundled with a Czech, Hebrew and Russian speller, as well as the English language Spellswell 7.

Services are available in a number of different languages. Besides the spellers just mentioned, there is the English, French and German MacPrimus speller from Applied Technologies in Berlin, Germany, and MacKilavuz, a Turkish speller written by Professor Akif Eyler of Bilkent University in Ankara, Turkey.

Ralf Menssen, of Applied Technologies, and Dr. Eyler were quite helpful in the early suite development.

We have an envelope addressing program called Mailswell in development. Based on our QuickLetter correspondence program, Mailswell adds an "Address Envelope" menu item to client applications. A postal address will be filtered from document text and printed on preformatted stationery envelopes that are set up within Mailswell.

IV. Creating a New Protocol

Now I come to what I find new and interesting in this discussion. I have described the protocol many times, from writing the specification itself, to discussing it at innumerable meetings, to presenting it now at the MacHack '94 Conference. I feel that the protocol is mature enough that support for it is growing steadily. Now (I hope) I can sit back and watch it grow, and reflect upon this experience.

The questions I put to myself are: would I do it again? Would I do it differently? What have I learned?

I may not have much real wisdom to shed here, but at perhaps I can plant some seeds in your minds. If you are contemplating the development of protocols yourselves, perhaps you can start with what I have done and do it better than I – or if not better, at least faster, with more realistic expectations.

Would I do it again? If I had really understood how much effort would be required, how long it would take for any payoff to occur, I would have chosen to spend my time, and my company's money, doing something that was more immediately profitable. I believe that I could have chosen other projects that would have greater payoff. Perhaps I would have chosen to do it later, after I had grown some as an engineer and manager, and had a deeper understanding of the industry.

I am glad that I have had this experience. This protocol has made my work known in the industry as a whole, in a way that would not be available if I just wrote products. For some reason it is more important to me as a person to gain a professional reputation than it is to make money. This has been the personal reward of doing Word Services: my work has become known, and we have not made much money at all.

There is another personal reward to developing and promoting this protocol. I have learned a great deal about working with people. The technical work – the programming and testing, the writing of the protocol, the debugging – has been small in comparison to the work with people that has been required, first to gain some consensus on the protocol, and then to get it adopted.

In my early days of programming I was only interested in the technical problems faced by programmers. To some extent it did not matter to me whether a program was ever finished. My concern was the challenge faced and obstacles overcome. The effort required to get a program "in the box" often lacks this technical fascination.

As a young programmer I also generally avoided the company of other people, preferring instead the predictability and comfort of my computer. I believe this is a trait shared by many programmers: for us the computer is a comfortable substitute for the complexities and frustrations of social interaction.

One cannot really be a good programmer if one cannot relate to people. I believe that the most important trait a programmer can possess is the ability to communicate well. One must listen carefully to the needs of others, translate these needs into technical ideas, judge these ideas for feasibility and economics, than translate this judgement into common language that any non–programmer can understand.

One must also be willing to let go of the security of the "technological sandbox" that we play in as young programmers. To be really productive means to get a program finished, even if it means devoting time to uninteresting and repetitive work such as testing. In a small company it means devoting time to activities entirely outside development, such as sales, marketing and technical support.

Thus, for me, Word Services provided the exercise and training needed to grow from a youthful hacker to a seasoned engineer.

Would I develop the protocol differently?

I am happy with the protocol itself, but I would change the process by which I developed it. I would not have set out on the process unless I had stronger commitments from more companies than I did.

I found early on that many people expressed great enthusiasm for the protocol. I had the hope that it would be widely adopted within a year of the time that I began working on it. I was disappointed to find that this enthusiasm quickly evaporated when I asked others to devote real labor to writing or reviewing the protocol specification, or writing code for the Word Services SDK.

The protocol got developed because a small group of fanatics were willing to devote their time and money far out of proportion to the expected payoff. We did feel that we were doing the right thing in a moral sense, but we may have chosen wrong in a business sense – our time might have been better spent working on other things.

Instead of taking off on my own to write the protocol, I should have spent more time initially in building a group of early adopters, getting all to commit to support the protocol once written, and then starting to write the protocol after this commitment had been made. This would have been quite difficult, but may have resulted in earlier commercial success.

What have I learned in writing Word Services?

I have learned to be more realistic in my expectations.

At the 1991 WWDC I met Sue Layman, program manager of the Apple Events Developer's Association. The AEDA was established by Apple to coordinate protocol development efforts, with the hope that it would eventually be spun off as an independent organization. Sue and I had great hope that many suites would be developed and put to market in a year or two. It seems that our youthful enthusiasm has been tempered by hard experience: it has taken me three years to get the protocol adopted, and Sue has since moved on to another company, the AEDA now a thing of the past.

I think that we could all learn from the Internet Protocol community. Protocol standards documents are called "Requests For Comments," and protocols are not considered to be officially adopted until they have been in real use for some period of time. No single company has the power to establish a protocol on its own – this requires real cooperation among competitors.

V. Conclusion

Word Services is adopted widely enough that I hope I can sit back and watch it grow.

It is a useful protocol, and a simple one. It serves a real need for users and for application developers. The simplicity, and the presence of this demand combine to give the protocol its success.

As we enter the age of the Information Superhighway and the beginnings of compound document architectures such as OpenDoc, we will all work more and more with communications protocols. Whether we adopt protocols or define protocols, or just use programs that support protocols without our conscious knowledge, communications protocols will be ever–present in our lives as programmers and computer users.

I encourage you to adopt the Word Services Apple Event Suite in your own programs, and if you contemplate the writing of a new protocol, I encourage you to learn from my experience.

Getting the Word Services SDK

To obtain the Word Services Software Development Kit, send your postal address to Michael Crawford at:

CompuServe 76004,2072
AppleLink D1620
America Online WorkingSW
Internet 76004.2072@compuserve.com

Working Software, Inc.
P.O. Box 1844
Santa Cruz, CA 95061-1844

References

[1] Crawford, Michael, et. al. "The Word Services Apple Event Suite," *Word Services Software Development Kit*, Working Software, Inc. 1994.

[2] Clark, Richard. "Apple Event Objects and You," d e v e l o p, May 1992.