Apple Events

The Æ Builder/Printer

Version 1.1

Jens Peter Alfke
12 August 1991

User Programming Group

Contents

OK, What Is It?

Even with the helpful new Object Support Library routines that assemble common Apple Event object descriptors, building descriptors is still a pain. I’ve written a library of two functions that make it quick and easy to build or display Apple Event descriptors.

The `AEBuild` function takes a format string — a description in a very simple language of an Apple Event descriptor — and generates a real descriptor (which could be a record or list) out of it. The `AEPrint` function does the reverse: given an Apple Event descriptor, list or record, it prettyprints it to a string. (The resulting string, if sent to `AEBuild`, would reproduce the original `AEDesc` structure.)

`AEBuild` can plug variable parameters into the structures it generates — as with `printf`, all you do is put marker characters in the format string and supply the parameter values as extra function arguments.

The benefits of using this library are fourfold:

- ❉ It’s easier for you to write the code to build Apple Event structures. You only have to remember one function call and a few simple syntax rules. Your resulting code is also easier to understand.
- ❉ As of version 1.1, your code is faster: `AEBuild` is at least twice as fast as the regular Apple Event Manager routines at constructing complex structures.
- ❉ Your code is smaller: the code for `AEBuild` and the AEStream library is about 5k in size, and the overhead for each call is minimal. (Most of the descriptor string consists of the same four-letter codes you’d be using in your program code anyway, and the strings can even be stored in resources for more code savings.)
- ❉ `AEPrint` helps in debugging programs, by turning mysterious `AEDesc` structures into human-readable text.

Just How Stable Is It, Anyway?

I’ve tested this code out, and verified that `AEBuild` runs reliably on a moderately complex expression, produces exactly the same Apple Event structure as does the original C code, and has no memory leakage. `AEBuild` has also been used in constructing test programs for the Object Support Library. These functions have not, however, been extensively tested. (See the disclaimer below.)

In addition, one bug was discovered and fixed since the first (1.0a3) release. See the release notes for more information.

In version 1.1, `AEBuild` now uses my nifty new `AEStream` library instead of the Apple Event Manager routines. This led to a threefold increase in speed, but be warned that the new code has *not* been as thorougly tested. If you encounter problems, try switching back to version 1.0a4 (available on Developer CD #8) and see if they go away.

I have written two programs so far that use `AEBuild` and `AEPrint`. The first is an MPW tool that converts an input string to an `AEDesc`, then converts it back and prints the results. (This tool is available in the distribution folder. It’s very useful for checking that your descriptor strings are syntactically correct. It does *not*, however, handle parameter substitution.)

The second is a timing tool that determines the average time to run the example code given later on in this document. The results of this test are given along with the sample code.

The libraries are being used in other places as well, such as Ed Lai’s AESend tool.

What Are All Those Files?

Here’s what you get:

Release Notes	Notes on the latest release.
AE Builder/Printer doc	This document.
AEBuild.o	MPW library of AEBuildfunction.
AEBuildWithGlobals.o	Ditto, but with the extra code and globals.
AEPrint.o	MPW library of AEPrint function.
AEBuild.h	Interface file for `AEBuild`.

AEBuild Globals.h	Global variable declarations, for use with the AEBuild/Print Global library.
AEPrint.h	Interface file for `AEPrint`.
AEBuildDemo	MPW tool for trying out AEBuild descriptor strings.
AEBuildDemo.c	Source code for the tool.

You'll also need:

AEStream.o	The AEStream library; should be available near where you found AEBuild. Look for a folder inside this folder or its parent folder.

Disclaimer

THIS SOFTWARE HAS NOT BEEN PAINSTAKINGLY TESTED BY APPLE'S RUTHLESSLY EFFICIENT QUALITY ENGINEERS. NEITHER APPLE COMPUTER, INCORPORATED, NOR THE AUTHOR OF THIS SOFTWARE MAKE ANY LEGALLY BINDING CLAIM THAT THIS SOFTWARE WILL DO ANYTHING IN PARTICULAR BESIDES USE UP VALUABLE SPACE ON A CD OR HARD DISK. IN THE EVENT THAT YOUR USE OF OR INABILITY TO USE THIS SOFTWARE RESULTS IN A VISITATION FROM MACSBUG, DAMAGE TO OTHER SOFTWARE OR HARDWARE, THE EXPLOSION OF YOUR MACINTOSH IN A SHOWER OF SPARKS (AS SEEN ON STAR TREK®) OR INDEED THE END OF WESTERN CIVILIZATION AS WE KNOW IT, YOUR ATTEMPTS TO ATTACH BLAME ONTO APPLE COMPUTER, INCORPORATED OR THE AUTHOR OF THIS SOFTWARE WILL BE EXPENSIVE AND UNSUCCESSFUL. HAVE A NICE DAY.

How To Call the Functions

The function interfaces are pretty trivial:

AEBuild

```
OSErr
       AEBuild(  AEDesc *desc, char *descriptorStr, ... ),
       vAEBuild( AEDesc *desc, char *descriptorStr, void *args );
```

`AEBuild` reads a null-terminated descriptor string (usually a constant, although it could come from anywhere), parses it and builds a corresponding `AEDesc` structure. (Don’t worry, I’ll describe the syntax of the descriptor string in the next section.) If the descriptor string contains magic parameter-substitution characters (“`@`”) then corresponding values of the correct type must be supplied as function arguments, just as with `printf`.

(`vAEBuild` is analogous to `vprintf`: Instead of passing the parameters along with the function, you supply a `va_list`, as defined in `<stdarg.h>`, that points to the parameter list. It’s otherwise identical.)

`AEBuild` returns an `OSErr`. Any errors returned by Apple Event manager routines while building the descriptor will be sent back to you. The most likely results are `memFullErr` and `errAECoercionFail`. Also likely is `aeBuildSyntaxErr`, resulting from an incorrect descriptor string. (Make sure to debug your descriptor strings, perhaps using the demo application, before you put them in programs!)

The basic version of `AEBuild` just reports that a syntax error occurred, without giving any additional information. If you want to know more (perhaps the string came from a user, to whom you’d like to report a helpful error message) you can use the other version of the library. This version includes a wee bit of extra code, and two global variables that will contain useful information after a syntax error:

```
extern AEBuild_SyntaxErrType
       AEBuild_ErrCode;
extern long
       AEBuild_ErrPos;
```

`AEBuild_ErrCode` is an enumerated value that will contain a specific error code. The error codes are defined in AEBuild.h. `AEBuild_ErrPos` will contain the index into the descriptor string at which the error occurred: usually one character past the end of the offending token.

△ **Important** `AEBuild` cannot be called from a Pascal program because it takes a variable number of arguments. `vAEBuild` could be called from Pascal, but you'd have to build the `va_list` by hand. △

AEPrint

```
OSErr AEPrint( AEDesc *desc, char *bufStr, long bufSize );
```

`AEPrint` reads the Apple Event descriptor `desc` and writes a corresponding descriptor string into the string pointed to by `bufStr`. It will write no more than `bufSize` characters, including the trailing null character. Any errors returned by Apple Event Manager routines will be returned to the caller; this isn't very likely unless the `AEDesc` structure is somehow corrupt.

The descriptor string produced, if sent to `AEBuild`, will build a descriptor identical to the original one. `AEPrint` tries to detect `AERecords` that have been coerced to other types and print them as coerced records. Structures of unknown type that can't be coerced to `AERecords` are dumped as hex data.

Descriptor-String Syntax

The real meat of all this, of course, is the syntax of the descriptor strings. It’s pretty simple: basic data types like numbers and strings can be described directly, and then built up into lists and records. I’ve even provided a pseudo-BNF grammar (next section) for those of you who actually enjoy reading those things.

Basic Types

The fundamental data types are:

Type	Examples	Type-code	Description

Integer	`1234`
	`-5678`
		`'long'` or
		`'shor'`
			A sequence of decimal digits, optionally preceded by a minus sign.

Enum/Type Code	`whos`
	`longint`
	`'long'`
	`<=`
	`'8-)'`
	`‘ZQ 5’`
	`m`
		`'enum'`
		
		(Use coercion to change to `'type'`)
			A magic four-letter code. Will be truncated or padded with spaces to exactly four characters. If you put straight or curly single-quotes around it, it can contain any characters. If not, it can’t contain any of: `@‘’'“”:-,([{}])` and can’t begin with a digit.

String	`“A String.”`
	`“Multiple lines are okay.”`
		`'TEXT'`	Any sequence of characters within open and close curly quotes. Won’t be null-terminated.

Hex Data	`«4170706C65»`
	`«0102 03ff e b 6 c»`
		??
		(Must be coerced to some type)
			An even number of hex digits between French quotes (Option-\, Option-Shift-\). Whitespace is ignored.

Coercion

Any basic element (except a hex string) by itself is a descriptor, whose descriptorType is as given in the table. You can coerce a basic element to a different type by putting it in parentheses with a type-code placed before it. Here are some examples:

```
sing(1234)
type(line)
long(CODE)
hexd("A String")
'blob'(«4170706C65»)
```

◆ Coercions of numeric values are effected by calling `AECoerceDesc`; if the coercion fails, you'll get an `errAECoercionFail` error returned to you. Coercions of other types just replace the `descriptorType` field of the `AEDesc`. ◆

◆ Hex strings *must* be coerced, because they have no intrinsic type. ◆

You can also coerce nothing, to get a descriptor with zero-length data:

```
emty()
```

Even the type can be omitted, leaving just `()`, in which case the type is `'null'`.

Lists

To make an `AEDescList`, just enclose a comma-separated list of descriptors in square brackets. For example:

```
[123, -456, "et cetera"]
[sing(1234), long(CODE),
 ["wheels", "within wheels"]]
[]
```

The elements of a list can be of different types, and a list can contain other lists or records (see below) as elements.

Lists cannot be coerced to other types; the type of a list is always `'list'`.

Records

An `AERecord` is indicated by a comma-separated list of elements enclosed in curly braces. Each element of a record consists of a keyword (a type-code, as described under Basic Types) followed by a "`:`", followed by a value, which can be any descriptor: a basic type, a list or another record. For example:

```
{x:100, y:-100}
{'origin': {x:100, y:-100}, extent: {x:500, y:500},
 cont: [1, 5, 25]}
{}
```

The default type of a record is `'reco'`. Many of the Apple Events Object Model structures are `AERecords` that have been coerced to some other data type, like `'indx'` or '`whos'`. You can coerce a record structure to any type by preceding it with a type code. For example:

```
rang{ star: 5, stop: 6}
```

Again, this coercion is done by calling `AECoerceDesc`, which has a general method for coercing an `AERecord` to any *non-primitive* type. Attempting to coerce a record to one of the primitive types, as in

```
bool{ star: 5, stop: 6 }
```
—Wrong-o, buckwheat!

will get you a loud raspberry from `AECoerceDesc`.

◆ Yes, you have to use the actual four-letter codes for keywords and object types, instead of the mnemonic constants. Luckily the codes are semi-mnemonic anyway. I did it this way to avoid the overhead, both in code size and execution speed, of a symbol table. You can find the definitions of the constants in the text file "AEObjects.p", which is part of the Apple Events Object Support Library. ◆

Substituting Parameters

To plug your own values into the midst of a descriptor, use the magic "`@`" character. You can use "`@`" anywhere you can put a basic element like an integer. Each "`@`" is replaced by a value taken from the parameter list sent to the `AEBuild` function. The type of value created depends on the context in which the "`@`" is used: in particular, how it's coerced.

Type Coerced to:	**Type of fn parameter read:**	**Comments:**
No coercion	`AEDesc*`	A plain "`@`" will be replaced with a descriptor parameter.
Numeric (`bool`, `shor`, `long`, `sing`, `doub`, `exte`)	`short`, `short`, `long`, `float`, `short double`, `double`	Remember that THINK C's `double` corresponds to type `'exte'`!
`TEXT`	`char*`	Pointer to a null-terminated C string.
Any other type	`long` followed by `void*`	Expects a length parameter followed by a pointer to the descriptor data.

△ **Important** Note particularly: that `TEXT` parameters must be null-terminated strings, although the resulting descriptor data will not be null-terminated; and that the general case expects *two* parameters: the data's size and location. △

This mechanism is still a bit limited, and may well be improved in the future.

Descriptor-String Grammar

Since no language, however small, can be taken seriously unless it comes fully equipped with a formidable-looking BNF grammar specification, I here present one. No attempt has been made to prevent Messrs. Backus and/or Naur from rolling over in their respective graves.

Character Classification:

whitespace	‘ ’, ‘\r’, ‘\n’, ‘\t’
digit	**0** … **9**
paren, bracket, braces	**(**, **)**, **[**, **]**, **{**, **}**
single-quote	**'**
double quotes	**“**, **”**
hex quotes	**«**, **»**
colon	**:**
comma	**,**
at-sign	**@**
identchar	*any other printable character*

Tokens:

ident ::=	**identchar (identchar | digit)*** **' character* '**	—Padded/truncated to exactly 4 chars
integer ::=	[-] **digit**+	—Just as in C
string ::=	**“ (character)* ”**	

hexstring ::=	**«** (**hexdigit** | **whitespace**)* **»**	—Even no. of digits, please

Grammar Rules:

formatstring ::=	obj	—This is the top level of syntax
obj ::=	data	—Single AEDesc; shortcut for **(**data**)**
	structure	—Un-coerced structure
	ident structure	—Coerced to some other type

structure ::=	**(** data **)**	—Single AEDesc
	[objectlist **]**	—AEList type
	{ keywordlist **}**	—AERecord type
objectlist ::=	*«blank»*	—Comma-separated list of things
	obj [**,** obj]*	
keywordpair ::=	**ident :** obj	—Keyword/value pair
keywordlist ::=	*«blank»*	—List of said pairs
	keywordpair [**,** keywordpair]*	
data ::=	**@**	—Gets appropriate data from fn param
	integer	—'shor' or 'long' unless coerced
	ident	—A 4-char type code ('type') unless coerced
	string	—Unterminated text; 'TEXT' type unless coerced
	hexstring	—Raw hex data; *must* be coerced to some type!

There. Now it’s all crystal-clear, right?

An Example & Timing Comparison

As an example, I’ll take a C function to generate an object descriptor (taken from a Pascal example in the Object Model ERS, fleshed out and with gobs of error checking added) and turn it into a call to `AEBuild`. The object descriptor we want to generate is:

> **First line of document 'Spinnaker' whose first word is 'April'**
> **and whose second word is 'is'**

Then I’ll execute both functions and compare their execution times.

C Code Using Object-Packing Library

```
OSErr
BuildByHand( AEDesc *dDocument, AEDesc *theResultObj )
{
	OSErr err;
	AEDesc dObjectExamined, dNum, dWord1, dWord2, dAprilText, dIsText,
		dComparison1, dComparison2, dLogicalTerms, dTheTest, dLineOne, dTestedLines;

	dObjectExamined.dataHandle =  /* Zero things to start out with so we can safely */
	   dNum.dataHandle =             /* execute our fail code if things don't work out */
	   dWord1.dataHandle =
	   dWord2.dataHandle =
	   dAprilText.dataHandle =
	   dIsText.dataHandle =
	   dComparison1.dataHandle =
	   dComparison2.dataHandle =
	   dLogicalTerms.dataHandle =
	   dTheTest.dataHandle =
	   dLineOne.dataHandle =
	   dTestedLines.dataHandle =
	      NIL;

	if( err= AECreateDesc( 'exmn', NIL, 0, &dObjectExamined ) )
	   goto fail;
```

```
if( err= MakeIndexDescriptor(1,&dNum) )
```

```
            goto fail;
        if( err= MakeObjDescriptor( 'word', &dObjectExamined, formIndex, &dNum,
                    false, &dWord1) )
            goto fail;
        if( err= AECreateDesc( 'TEXT', "April", 5, &dAprilText ) )
            goto fail;

        AEDisposeDesc(&dNum);
        if( err= MakeIndexDescriptor(2,&dNum) )
            goto fail;
        if( err= MakeObjDescriptor( 'word', &dObjectExamined, formIndex, &dNum,
                    true, &dWord2) )
            goto fail;
        if( err= AECreateDesc( 'TEXT', "is", 2, &dIsText ) )
            goto fail;

        if( err= MakeCompDescriptor( '=   ', &dAprilText, &dWord1, true, &dComparison1 ) )
            goto fail;
        if( err= MakeCompDescriptor( '=   ', &dIsText,   &dWord2, true, &dComparison2 ) )
            goto fail;

        if( err= AECreateList( NIL, 0, false, &dLogicalTerms ) )
            goto fail;
        if( err= AEPutDesc( dLogicalTerms, 1, dComparison1 ) )
            goto fail;
        if( err= AEPutDesc( dLogicalTerms, 2, dComparison2 ) )
            goto fail;

        AEDisposeDesc(&dComparison1);
        AEDisposeDesc(&dComparison2);

        if( err= MakeLogicalDescriptor( &dLogicalTerms, 'AND ', true, &dTheTest) )
            goto fail;

        if( err= MakeObjDescriptor(classLine,&dDocument,formTest,&dTheTest,true,
                                &dTestedLines) )
            goto fail;

        if( err= MakeIndexDescriptor(1,&dLineOne) )
            goto fail;
        if( err= MakeObjDescriptor( classLine, &dTestedLines, formIndex, &dLineOne,
                true, theResultObj ) )
            goto fail;
        return noErr;

fail:                                   /* Clean up in case we couldn't build it */
        AEDisposeDesc(theResultObj);
        AEDisposeDesc(&dObjectExamined);
        AEDisposeDesc(&dNum);
        AEDisposeDesc(&dWord1);
        AEDisposeDesc(&dWord2);
        AEDisposeDesc(&dAprilText);
        AEDisposeDesc(&dIsText);
        AEDisposeDesc(&dComparison1);
        AEDisposeDesc(&dComparison2);
        AEDisposeDesc(&dLogicalTerms);
        AEDisposeDesc(&dTheTest);
        AEDisposeDesc(&dLineOne);
        AEDisposeDesc(&dTestedLines);

        return err;
}
```

MPW 3.2b5 C compiled this into 816 bytes of object code.

The average time to execute this function, on a Mac IIfx, is 0.0184 seconds. Use this figure for comparison only; your times may vary. The timing is especially dependent on the number of blocks in the heap, since so many block allocations and disposals are happening.

Descriptor String

```
obj{ want:type('line'),
    from: obj{ want: type('line'), from: @, form: 'test',
               seld: logi{
                          term: [comp{ relo:=, obj1:“April”,
                                       obj2:
                             obj{ want:type('word'), from:exmn(), form:indx, seld:1 }},
                                 comp{ relo:=, obj1:“is”,
                                       obj2:
                             obj{ want:type('word'), from:exmn(), form:indx, seld:2 }}
                                ],
                          logc:AND
                        }
             },
    form: 'indx',
    seld: 1
}
```

AEBuild Call

```
char descriptor[] =           /* Same descriptor string as above. Note clever */
"obj{ want:type('line'),"     /* method used to break string across lines. */
   "from: obj{ want: type('line'), from: @, form: 'test',"     /* Note parameter here */
              "seld: logi{"
                         "term: [comp{ relo:=, obj1:“April”,"
                                      "obj2:"
                            "obj{ want:type('word'), from:exmn(), form:indx, seld:1 }},"
                                "comp{ relo:=, obj1:“is”,"
                                      "obj2:"
                            "obj{ want:type('word'), from:exmn(), form:indx, seld:2 }}"
                               "],"
                         "logc:AND"
                       "}"
            "},"
   "form: 'indx',"
   "seld: 1"
"}";

void PackWordDesc( AEDesc *dDocumentObject )   /* “Spinnaker” descriptor is a parameter */
{
       err = AEBuild(&theResultObj,
                    descriptorString,
                    dDocumentObject);                /* AEDesc* parameter for "@" */
}
```

MPW 3.2b5 C compiled this into 42 bytes of object code, plus 310 bytes of data storage for the string.

The average time to execute this function, on a Mac II, is 0.0097 seconds. Use this figure for comparison only; your times may vary. The timing is dependent on the number of blocks in the heap, since heap blocks are being allocated and resized.

Timing Conclusions

With previous versions of this library, there was a 70% increase in execution time when using the `AEBuild` routine. After delivering the bad news, I wrote:

> However, if speed does become an issue, there is always the option of turbocharging `AEBuild` by having it directly build descriptors without going through the Apple Event Manager functions at all. This would save an incredible number of Memory Manager calls and probably increase performance severalfold. Anyone using `AEBuild` will get all these improvements for free.

This is exactly what I did in version 1.1. In fact, I wrote a library (AEStream) to do it, so you can do it too. It's easy.

`AEBuild` is now almost twice as fast (89% faster) as the using the Apple Event Manager and/or Object Packing Library routines. (This means that AEStream was responsible for a threefold speed-up in AEBuild. Not bad, when you take into account other overhead like parsing the format string!)

Needless to say, if you were already using `AEBuild` you get this speed increase absolutely free. Enjoy!

The Demo Program

I’ve included a demonstration MPW tool in the distribution. This is a program I used to debug the library. It reads each command-line argument, uses AEBuild to translate it into an AEDesc, uses AEPrint to translate the AEDesc back into a string, and prints each resulting string. If you don’t give any command-line arguments it will ask you to enter a line from the standard input. Error codes are reported, including syntax-error messages. The source code is provided in case you want to see how the functions are called.

▲ **Warning** The demo tool does **not** handle parameter substitution (the “`@`” character). If you try to substitute parameters, messy and unpleasant things may happen. Use some numeric value in place of parameters, and then replace it with “`@`”s after you paste the string into your program. ▲

The Header Files

Here for your convenience are printouts of the header files as of 28 January 1991.

AEBuild.h

```
/*
*      AEBuild.h                            Copyright ©1991 Apple Computer, Inc.
*/

#define aeBuild_SyntaxErr  12345            /* Let's get an Official OSErr code someday */

typedef enum{                        /* Syntax Error Codes: */
       aeBuildSyntaxNoErr = 0,              /* (No error) */
       aeBuildSyntaxBadToken,               /* Illegal character */
       aeBuildSyntaxBadEOF,                    /* Unexpected end of format string */
       aeBuildSyntaxNoEOF,                  /* Unexpected extra stuff past end */
       aeBuildSyntaxBadNegative,            /* "-" not followed by digits */
       aeBuildSyntaxMissingQuote,           /* Missing close "'" */
       aeBuildSyntaxBadHex,                    /* Non-digit in hex string */
       aeBuildSyntaxOddHex,                    /* Odd # of hex digits */
       aeBuildSyntaxNoCloseHex,                /* Missing "»" */
       aeBuildSyntaxUncoercedHex,           /* Hex string must be coerced to a type */
       aeBuildSyntaxNoCloseString,             /* Missing "”" */
       aeBuildSyntaxBadDesc,                /* Illegal descriptor */
       aeBuildSyntaxBadData,                /* Bad data value inside (…) */
       aeBuildSyntaxNoCloseParen,           /* Missing ")" after data value */
       aeBuildSyntaxNoCloseBracket,         /* Expected "," or "]" */
       aeBuildSyntaxNoCloseBrace,           /* Expected "," or "}" */
       aeBuildSyntaxNoKey,                  /* Missing keyword in record */
       aeBuildSyntaxNoColon                    /* Missing ":" after keyword in record */
       aeBuildSyntaxCoercedList                /* Cannot coerce a list */
} AEBuild_SyntaxErrType;


OSErr
       AEBuild(  AEDesc *dst, char *src, ... ),
       vAEBuild( AEDesc *dst, char *src, void *args );
```

AEBuildGlobals.h

```
/*
*      AEBuildGlobals.h                      Copyright ©1991 Apple Computer, Inc.
*/


extern AEBuild_SyntaxErrType
       AEBuild_ErrCode;              /* Examine after AEBuild returns a syntax error */
extern long
       AEBuild_ErrPos;               /* Index of error in format string */
```

AEPrint.h

```
/*
*      AEPrint.h                                Copyright ©1991 Apple Computer, Inc.
*/

OSErr AEPrint( AEDesc *desc, char *bufStr, long bufSize );
```