

The AppleEvent Builder/Printer

Apple Events

The \AE Builder/Printer

Version 1.3.3

Jens Peter Alfke

11 October 1993

The AppleEvent Builder/Printer

The AppleEvent Builder/Printer

AppleScript Team

© Apple Computer, Inc. 1991–1993

The AppleEvent Builder/Printer

Contents

Contents.....	iii
Introduction.....	1
OK, What Is It?	1
What's New?	2
How To Call the Functions.....	3
AEBuild	3
AEBuildParameters	4
AEBuildAppleEvent	4
AEPrint	5
Descriptor-String Syntax.....	6
Basic Types	6
Coercion	7
Lists	8
Records	8
Apple Events	9
Substituting Parameters	10
Descriptor-String Grammar.....	11
An Example & Timing Comparison.....	13
C Code Using Object-Packing Library	13
Descriptor String	15
AEBuild Call	16
Timing Conclusions	17
The Demo Program.....	18
The Header Files.....	19
AEBuild.h	19
AEBuildGlobals.h	20
AEPrint.h	20

OK, What Is It?

Even with the helpful Object Support Library routines that assemble common Apple event object descriptors, building descriptors and events 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 and the Apple events themselves.

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 or event) 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:

- i 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.
- i As of version 1.2, your code is even faster: `AEBuild` is three to four times as fast as the regular Apple Event Manager routines at constructing complex structures. (Your mileage may vary.)
- i Your code is smaller: the code for `AEBuild` and the `AESTream` library is about 6k 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.)
- i `AEPrint` helps in debugging programs, by turning mysterious `AEDesc` structures into human-readable text.

What's New?

In version 1.3:

- A new function, `AEPrintSize`, computes how long the string built by `AEPrint` would be, without actually creating it. This is useful if you want to allocate storage for the string dynamically.
- `AEPrint` no longer truncates hex dumps (of unknown descriptors) after 32 bytes.

And in version 1.3.2:

- `AEPrint` no longer uses the `stdio` library. This should help reduce code size (and eliminate some problems in code resources) but to do it I had to cripple floating-point display. Floating-point descriptors now print as the integer part followed by “.XXXX”.

And in version 1.3.3:

- After a brilliant suggestion by Rob Dye, `AEPrint` now uses the built in float-to-text coercion to display floating-point descriptors.
- Fixed a possible problem with `AEBuild` input strings containing Return characters, in the MPW version of the library.

How To Call the Functions

These are all C functions. They all take variable number s of arguments, so they'd be difficult or impossible to call from Pascal, anyway. (And remember, kids: *there are no Pascal compilers for the PowerPC chip...*)

AEBuild

```
OSErr  
    AEBuild( AEDesc *desc, const char *descriptorStr, ... ),  
    vAEBuild( AEDesc *desc, const 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:

The AppleEvent Builder/Printer

```
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.

AEBuildParameters

```
OSErr AEBuildParameters( AppleEvent *event, const char *descriptorStr, ... );
```

`AEBuildParameters` adds parameters and/or attributes to an existing Apple event. `descriptorStr` specifies the parameters (required and optional) and attributes. Its syntax is described below (see especially the Apple Event Descriptor Strings subsection); it's almost the same as the syntax for `AEBuild`, with a few additions and modifications.

(`vaAEBuildParameters` 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.)

AEBuildAppleEvent

```
AEBuildAppleEvent( AEEEventClass theClass, AEEEventID theID,
    DescType addressType, void *addressData, long addressLength,
    short returnID, long transactionID,
    AppleEvent *event,
    const char *descriptorStr, ... );
```

`AEBuildAppleEvent` is like `AEBuild` but builds an Apple event, including parameters and attributes. Or, you could say that it's like `AEBuildParameters` but creates the event from scratch.

The AppleEvent Builder/Printer

The AppleEvent Builder/Printer

Most of the parameters are just like the parameters to `AECreatAppleEvent`, although you pass the target address data directly, instead of via a pre-built descriptor. The resulting Apple event will appear in the `event` parameter.

`descriptorStr` specifies the parameters (required and optional) and attributes. The syntax is described below (see especially the Apple Event Descriptor Strings subsection); it's almost the same as the syntax for `AEBuild`, with a few additions and modifications.

(`VAEBuildAppleEvent` 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.)

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.

`AEPrint` can also print complete Apple events as well as regular descriptors. The syntax of the resulting string for an event is like that used by `AEBuildParameters` and `AEBuildAppleEvent`, except that:

- The string begins with the event class and ID separated by a backslash.
- the parameter list is surrounded by curly braces.
- Attributes are also displayed; they look like parameters but are preceded by "&"s.

u The builder functions do **not** accept this event syntax yet. u

AEPrintSize

```
OSErr AEPrintSize( AEDesc *desc, long *bufSizeNeeded );
```

`AEPrintSize` computes the buffer size that `AEPrint` would require if given the same descriptor. (The size is equal to the string length, plus 1 byte for the trailing null.) This is handy for pre-fighting `AEPrint`, if you want to allocate the buffer dynamically instead of relying on one of fixed size .

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 'short'	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.

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

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»)
```

- u 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`. u
- u Hex strings *must* be coerced to something, since they have no intrinsic type. u

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

```
empty()
```

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}
```

§ **Warning** Coercing to an existing type, such as 'bool' or 'TEXT', is a bad idea. Anyone parsing the descriptor (including AEPrint) will recognize the type and assume that the data has the normal interpretation, which in this case it wouldn't. Bad to awful things would happen. Don't do it.

Apple Events

The syntax of the formatting string for an entire Apple event (as passed to AEBuildAppleEvent) is almost identical to that of a record. Each keyed element specified in the string becomes a parameter or attribute of the event. The differences are:

- There are no curly-braces at the beginning and end of the string.
- The character “~” before a parameter keyword makes it optional.

Here's an example of how to construct an Open Selection event for the Finder:

```
AliasHandle parent, itemToOpen;
const OSType finderSignature = 'MACS';
AppleEvent event;
OSErr err;

// Construct the aliases here (not shown)

err= AEBuildAppleEvent(
    'FNDR', 'SOPE',
    typeApplSignature, @finderSignature, sizeof(finderSignature),
    kAutoGenerateReturnID, kAnyTransactionID,
    &event,                                     // Event to be created
    "----: alis(@@), fsel: [alis(@@)]",        // Format string
    parent,                                    // param for 1st @@
    itemToOpen                                 // param for 2nd @@
);
```

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	<code>AEDesc*</code>	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	<code>char*</code>	Pointer to a null-terminated C string.
Any other type	long followed by <code>void*</code>	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. §

In addition, you can substitute data from a handle by using two @ signs. An “@@” parameter will read a single handle from the parameter list and use the data pointed to by that handle as the value of the descriptor. The “@@” must be coerced so that `AEBuild` will know what type to make the descriptor; however, the type coerced to can be anything (the table above is ignored.)

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	<i>any other printable character</i>

Tokens:

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

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

Grammar Rules for AEBuild:

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

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

Grammar Rules for AEBuildAppleEvent:

eventstring ::=	evtkeywordlist	—Top level syntax for AEBuildAppleEvent
evtkeywordpair ::=	[~] ident : obj	—Keyword/value pair
evtkeywordlist ::=	«blank» evtkeywordpair [, evtkeywordpair]*	—List of said pairs

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= AECreatDesc( 'exmn', NIL, 0, &dObjectExamined ) )
        goto fail;
}
```

The AppleEvent Builder/Printer

```
if( err= MakeIndexDescriptor(1, &dNum) )  
    goto fail;
```

The AppleEvent Builder/Printer

```
if( err= MakeObjDescriptor( 'word', &dObjectExamined, formIndex, &dNum,
                           false, &dWord1) )
    goto fail;
if( err= AECreatDesc( '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= AECreatDesc( '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= AECreatList( 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);
```

The AppleEvent Builder/Printer

The AppleEvent Builder/Printer

```
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.

I found that the average time to execute this function was 0.0188 seconds (Quadra 700) or 0.0113 seconds (IIfx).^{1†} 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
}
```

^{1†} Yes, it really took half again as long on a Quadra! I think that cache flushing during the PACK call is responsible. (It barely slows down at all when you disable the caches.)

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.

I found that the average time to execute this function was 0.0049 seconds (Quadra 700) or 0.0070 seconds (IIfx). 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 (`AESStream`) to do it, so you can do it too. It's easy.

`AEBuild` is now 1.5 to 4 times as fast (depending on CPU) as the using the Apple Event Manager and/or Object Packing Library routines. (This means that `AESStream` 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 program in the distribution. This is a program I used to debug the library. It reads a line of input, uses AEBuild to translate it into an AEDesc, uses AEPrint to translate the AEDesc back into a string, and prints each resulting string. Error codes are reported, including syntax-error messages. The source code is provided in case you want to see how the functions are called.

s **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. s

The Header Files

Here for your convenience are printouts of the header files as of 21 July 1992.

AEBuild.h

```
#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 */
    aeBuildSyntaxUncoercedDoubleAt       /* "@" substitution must be coerced */
} AEBuild_SyntaxErrType;

// In all the "v..." functions, the "args" parameter is really a va_list.
// It's listed as void* here to avoid having to #include stdarg.h.

// Building a descriptor:

OSErr
    AEBuild( AEDesc *dst, const char *src, ... ),
```

The AppleEvent Builder/Printer

```
VAEBuild( AEDesc *dst, const char *src, const void *args );
```

The AppleEvent Builder/Printer

// Adding a parameter to an Apple event:

```
OSErr
    AEBuildParameters( AppleEvent *event, const char *format, ... ),
    vAEBuildParameters( AppleEvent *event, const char *format, const void *args );
```

// Building an entire Apple event:

```
OSErr
    AEBuildAppleEvent( AEEEventClass theClass, AEEEventID theID,
                      DescType addressType, const void *addressData, long addressLength,
                      short returnID, long transactionID, AppleEvent *result,
                      const char *paramsFmt, ... ),
    vAEBuildAppleEvent( AEEEventClass theClass, AEEEventID theID,
                      DescType addressType, const void *addressData, long addressLength,
                      short returnID, long transactionID, AppleEvent *resultEvt,
                      const char *paramsFmt, const 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 );
```