This chapter describes the resource structure used by both the `'aeut'` and `'aete'` resources and explains how to create an `'aete'` resource for your application. It also explains how applications that support additional plug-in modules, each with its own `'aete'` resource, can write a handler for the Get AETE event that collects the `'aete'` resources from the modules that are currently running.

Before you read this chapter, you should read the chapter "Introduction to Scripting" in this book and the chapters about the Apple Event Manager that are relevant to your application.

The first section in this chapter describes how the AppleScript component interprets AppleScript statements that trigger Apple events. The first section also explains how to define both Apple events and the corresponding user terminology for your application in a way that translates easily into AppleScript statements. If you implement Apple events so that they translate into logical and useful AppleScript scripts, your implementation will probably work well with other scripting components that resemble AppleScript.

The next two sections describe how to

- create an `'aete'` resource
- handle the Get AETE event

For details about the structure of the data in an `'aeut'` resource and an `'scsz'` resource, see "Reference to Apple Event Terminology Resources," which begins on page 8-26.



# Defining Terminology for Use by the AppleScript Component

You should keep two principles in mind when you are defining the Apple event object hierarchy and corresponding terminology for your application:

- Avoid defining new Apple events unless absolutely necessary. For example, instead of defining a custom Find event, use the Get Data event with whose tests. (For more information about whose tests, see the chapter "Resolving and Creating Object Specifier Records" in this book.)
- Use existing object classes, or if you must define your own, define them in a general fashion.

## Apple Event Terminology Resources

This section describes how the terms you specify in your application's `'aete'` resource are used in AppleScript statements that control your application. Before you implement the Apple event object hierarchy for your application, try out your proposed user terminology in AppleScript statements that use the standard syntax forms described here. This will help you discover some of the advantages and disadvantages of both your proposed object hierarchy and the human-language terminology you are planning to use.

Some AppleScript commands, such as `if`, `repeat`, and `tell`, are executed directly by the AppleScript component and do not correspond to Apple events. Other commands trigger Apple events when the AppleScript component evaluates them.

| AppleScript command | Corresponding Apple event |
|---|---|
| `open` | Open |
| `close` | Close |
| `save` | Save |
| `move` | Move |
| `delete` | Delete |
| `set` | Set Data |

The AppleScript component interprets the terms used in scripts according to rules defined by the AppleScript language. For example, the `open` command must be followed by an argument that specifies the objects to open, and the `save` command must be followed by an argument that specifies the objects to save. The AppleScript component uses the information in an application's `'aete'` resource to map the human-language terms used in these arguments to specific Apple event keywords and codes, so that it can construct object specifier records that describe the objects on which the Open and Save events act.

In general, the syntax for AppleScript commands that trigger Apple events follows this pattern:

*event name expression parameter name expression . . . parameter name expression*

The underlined terms are supplied by the AppleScript component's `'aeut'` resource, by the application's `'aete'` resource, or by the `'aeut'` resource available on the current computer. The argument that follows *event name* corresponds to the direct parameter for the event, if there is one. Each subsequent argument corresponds to an additional parameter.

An argument that corresponds to a direct parameter can use any of the syntax forms shown in Table 8-1. These forms correspond to the key forms that can be used to identify the key data in an object specifier record.

**Table 8-1** Syntax for AppleScript arguments that correspond to direct parameters

| Syntax of argument | AppleScript example | Key form |
|---|---|---|
| *property name* | `the font` | `formPropertyID` |
| *class name expression* | `table "Fred"`<br>`table 4` | `formName`<br>`formAbsolutePosition` |
| *class name* `before` \| `after` *expression* | `word after table 2` | `formRelativePosition` |
| *class name expression* `thru` *expression* | `words 1 thru 30` | `formRange` |
| `every` *class name* `whose` *expression* | `every word whose¬`<br>`  font = "Palatino"` | `formWhose` |
| *expression* of *expression* | `first row of¬`<br>`  table "Fred"` | Any key form; sets container for elements or properties |

If the Apple event object hierarchy for your application requires you to specify terms in your `'aete'` resource that are not included in the `'aeut'` resource, make sure those terms read naturally when they appear in AppleScript statements that use the syntax shown in Table 8-1. Any of the underlined terms in the table may be supplied by your application's `'aete'` resource.

For example, in the AppleScript statement

`copy` *name* `to` *expression*

the argument *name* corresponds to a direct parameter that can use any of the syntax variations shown in Table 8-1. The word `to` and the expression that follows it correspond to an additional parameter that describes the location to which to copy the objects described by the direct parameter.

Many AppleScript commands, including the `copy` command, take additional arguments that correspond to ***insertion location descriptor records*** , which are descriptor records of type `typeInsertionLoc` defined as part of the Core suite. An insertion location descriptor record is a coerced AE record that consists of two keyword-specified descriptor records with the following keywords:

| Keyword | Description |
|---|---|
| `keyAEObject` | An object specifier record that identifies a single container |
| `keyAEPosition` | A constant that specifies where to put the Apple event object described in an Apple event's direct parameter in relation to the container specified in the descriptor record with the keyword `keyAEObject` |

You can specify one of these constants for the data in a descriptor record identified by the keyword `keyAEPosition`:

| Constant | Meaning |
|---|---|
| `kAEBefore` | Before the container |
| `kAEAfter` | After the container |
| `kAEBeginning` | In the container and before all other elements of the same class as the object being inserted |
| `kAEEnd` | In the container and after all other objects of the same class as the object being inserted |
| `kAEReplace` | Replace the container |

The syntax that corresponds to an insertion descriptor record can take any of the forms shown in Table 8-2.

**Table 8-2** Syntax for AppleScript arguments that correspond to insertion location descriptor records

| Syntax of argument | AppleScript example | `keyAEPosition` constant |
|---|---|---|
| `before` \| `after` *expression* | `before Figure 1` | `kAEBefore` \| `kAEAfter` |
| `beginning of` \| `end of` *expression* | `end of window 2` | `kAEBeginning` \| `kAEEnd` |
| *expression* | `Figure 1` | `kAEReplace` |

For example, in the AppleScript statement

```
copy Chart 1 of document "Sales Chart" to before Figure 1
```

the term `copy` corresponds to a Clone event, and `Chart 1 of document "Sales Chart"` corresponds to the direct parameter for the Clone event. The term `to` is the human-language name specified by the `'aeut'` resource for the additional parameter identified by the keyword `kAEInsertHere`, which always consists of an insertion location descriptor record. The term `before` corresponds to the constant `kAEBefore` in the descriptor record identified by the keyword `keyAEPosition`, and `Figure 1` corresponds to the object specifier record identified by the keyword `keyAEObject`.

The AppleScript component handles statements that describe the replacement of one object with another differently from statements that specify an insertion location before, after, at the beginning of, or at the end of an object.

For example, in the statement

```
copy Chart 1 of document "Sales Chart" to Figure 1
```

the term `to` is the human-language name for the additional parameter identified by the keyword `kAEInsertHere`. When `to` is followed immediately by an element expression like `Figure 1`, the Clone Apple event sent by the AppleScript component includes an additional parameter that consists of an object specifier record for Figure 1. When your application requests the parameter as an insertion location descriptor record, a system coercion handler installed by the AppleScript component converts the object specifier record to an insertion location descriptor record that specifies `kAEReplace` in the descriptor record identified by the keyword `keyAEPosition`.

If your application defines any extensions to the standard Apple events or object classes that require the use of insertion locations, use standard insertion location descriptor records to specify them, and make sure your Apple event object hierarchy and the corresponding human terminology in your `'aete'` resource allow the AppleScript component to translate insertion location descriptor records into meaningful statements in an AppleScript dialect.

Unlike most other AppleScript commands, the `copy` command causes the AppleScript component to send different Apple events under different circumstances. In the examples just discussed, the `copy` command corresponds to a Clone event. However, after evaluating the statements

```
tell application "SurfWriter"
      copy table "Summary of Sales" of document¬
         "Sales Report" to Totals
end tell
```

the AppleScript component sends a Get Data event and sets the variable `Sales91` to the value of the returned data; and the statements

```
tell application "SurfCharter"
      copy Totals to Chart 1 of document "Sales Chart"
end tell
```

cause the AppleScript component to send a Set Data event that sets the data in the specified chart to the value of the variable `Totals`.

All scriptable applications should support the Get Data, Set Data, and Clone events for all Apple event objects that a user might want to manipulate from a script with the `copy` command. Scriptable applications should also support the other core events and any appropriate functional-area events.

If you find it difficult to come up with meaningful AppleScript statements based on your proposed implementation of Apple events, you may need to rethink your implementation.

# Structure of Apple Event Terminology Resources

Table 8-3 summarizes the resource structure used by both the `'aeut'` and `'aete'` resources. Each asterisk (*) in the table indicates the beginning of an array. Each array can contain any number of items, including both additional arrays and specific definitions (■).

**Table 8-3** Structure of the `'aeut'` and `'aete'` resources

- ■ Template version
- ■ Language code
- * Array of suites:
    - ■ Suite information
    - * Array of events:
        - ■ Event information (including information about the direct parameter)
        - * Array of other parameters:
            - ■ Parameter information
    - * Array of classes:
        - ■ Class description
        - * Array of properties:
            - ■ Property information
        - * Array of elements:
            - ■ Element information
            - * Array of key forms:
                - ■ Key form information
    - * Array of comparison operators:
        - ■ Comparison operator information
    - * Array of enumerations:
        - ■ Enumeration information
        - * Array of enumerators:
            - ■ Enumerator information

Listing 8-1 shows the resource type declaration in Rez format for the `'aeut'` resource, which can also serve as a template for an `'aete'` resource. (Rez is a resource compiler available with the MPW programming environment.) For complete descriptions of all the fields shown in Listing 8-1, see "Reference to Apple Event Terminology Resources," beginning on page 8-26.

**Listing 8-1** Resource type declaration for the `'aeut'` resource

```
type 'aeut' {
   hex byte;                          /*major version in binary-coded */
                                      /* decimal (BCD)*/
   hex byte;                          /*minor version in BCD*/
   integer  Language, english = 0, japanese = 11;  /*language code*/
   integer  Script, roman = 0;        /*script code*/
   integer = $$Countof(Suites);
   array Suites {
      pstring;                        /*human-language name of suite*/
      pstring;                        /*suite description*/
      align word;                     /*alignment*/
      literal longint;                /*suite ID*/
      integer;                        /*suite level*/
      integer;                        /*suite version*/
      integer = $$Countof(Events);
      array Events {
         pstring;                     /*human-language name of event*/
         pstring;                     /*event description*/
         align word;                  /*alignment*/
         literal longint;             /*event class*/
         literal longint;             /*event ID*/
         literal longint   noReply = 'null';    /*reply type*/
         pstring;                     /*reply description*/
         align word;                  /*alignment*/
         boolean  replyRequired,      /*if the reply is */
                  replyOptional;      /* required*/
         boolean  singleItem,         /*if the reply must be a list*/
                  listOfItems;
         boolean  notEnumerated,      /*if the type is enumerated*/
                  enumerated;
         boolean  reserved;           /*these 13 bits are reserved; */
         boolean  reserved;           /* set them to "reserved"*/
         boolean  reserved;
         boolean  reserved;
         boolean  reserved;
```

```
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved,         /*if event is verb event or nonverb */
         nonVerbEvent;     /* event; used by Japanese dialect*/
literal longint   noParams = 'null'; /*direct param type*/
pstring;                   /*direct param description*/
align word;                /*alignment*/
boolean  directParamRequired, /*if the direct param is required*/
         directParamOptional;
boolean  singleItem,       /*if the param must be a list*/
         listOfItems;
boolean  notEnumerated,    /*if the type is enumerated*/
         enumerated;
boolean  doesntChangeState, /*if the event changes server's state*/
         changesState;
boolean  reserved;         /*these 12 bits are reserved; */
boolean  reserved;         /* set them to "reserved"*/
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
boolean  reserved;
integer = $$Countof(OtherParams);
array OtherParams {
   pstring;                /*human-language name for parameter*/
   align word;             /*alignment*/
   literal longint;        /*parameter keyword*/
   literal longint;        /*parameter type*/
   pstring;                /*parameter description*/
   align word;             /*alignment*/
   boolean  required,      /*if param is required*/
            optional;
```

```
      boolean  singleItem,    /*if the param must be a list*/
               listOfItems;
      boolean  notEnumerated, /*if the type is enumerated*/
               enumerated;
      boolean  isNamed,       /*indicates if this should be the */
               isUnnamed;     /* unnamed parameter; only one */
                              /* parameter can be so marked; set to */
                              /* reserved if not required*/
      boolean  reserved;      /*these 9 bits are reserved; */
      boolean  reserved;      /* set them to "reserved"*/
      boolean  reserved;
      boolean  reserved;
      boolean  reserved;
      boolean  reserved;
      boolean  reserved;
      boolean  reserved;
      boolean  reserved;
      boolean  notFeminine,   /*feminine; set to reserved if not  */
               feminine;      /* required*/
      boolean  notMasculine,  /*masculine; set to reserved if not */
               masculine;     /* required*/
      boolean  singular,
               plural;        /*plural*/
   };
};
integer = $$Countof(Classes);
array Classes {
   pstring;                   /*human-language name for class*/
   align word;                /*alignment*/
   literal longint;           /*class ID*/
   pstring;                   /*class description*/
   align word;                /*alignment*/
   integer = $$Countof(Properties);
   array Properties {
      pstring;                /*human-language name for property*/
      align word;             /*alignment*/
      literal longint;        /*property ID*/
      literal longint;        /*property class*/
      pstring;                /*property description*/
      align word;             /*alignment*/
      boolean  reserved;      /*reserved*/
      boolean  singleItem,    /*if the property must be a list*/
               listOfItems;
```

```
      boolean  notEnumerated, /*if the type is enumerated*/
               enumerated;
      boolean  readOnly,      /*can only read it*/
               readWrite;     /*can read or write it*/
      boolean  reserved;      /*these 9 bits are reserved; */
      boolean  reserved;      /* set them to "reserved"*/
      boolean  reserved;
      boolean  reserved;
      boolean  reserved;
      boolean  reserved;
      boolean  reserved;
      boolean  reserved;
      boolean  reserved;
      boolean  notFeminine,   /*feminine; set to reserved if not  */
               feminine;      /* required*/
      boolean  notMasculine,  /*masculine; set to reserved if not */
               masculine;     /* required*/
      boolean  singular,
               plural;        /*plural*/
   };
   integer = $$Countof(Elements);
   array Elements {
      literal longint;        /*element class*/
      integer = $$Countof(KeyForms);
      array KeyForms {        /*list of key forms*/
         literal longint
         formAbsolutePosition = 'indx',
         formName = 'name';   /*key form ID*/
      };
   };
};
integer = $$Countof(ComparisonOps);
array ComparisonOps {
   pstring;                   /*human-language name for */
                              /* comparison operator*/
   align word;                /*alignment*/
   literal longint;           /*comparison operator ID*/
   pstring;                   /*comparison operator description*/
   align word;                /*alignment*/
};
integer = $$Countof(Enumerations);
array Enumerations {          /*list of enumerations*/
   literal longint;           /*enumeration ID*/
```

```
            integer = $$Countof(Enumerators);
            array Enumerators {         /*list of enumerators*/
               pstring;                 /*human-language name for enumerator*/
               align word;              /*alignment*/
               literal longint;         /*enumerator ID*/
               pstring;                 /*enumerator description*/
               align word;              /*alignment*/
            };
         };
      };
   };
};
```



# Creating an Apple Event Terminology Extension Resource

Scriptable applications must include an Apple event terminology extension (`'aete'`) resource. You use an `'aete'` resource to inform scripting components about the extent of your application's support for the standard Apple event suites, any custom Apple events or Apple event objects defined by your application, and the corresponding human-language terms for use in scripts that control your application.

The format of the `'aete'` resource is identical to that of the `'aeut'` resource, although it serves a different purpose. The `'aeut'` resource maps human-language names to IDs, keywords, and other codes used in the Apple events described by the current edition of the *Apple Event Registry: Standard Suites*. The `'aete'` resource for an application uses the same format to accomplish the following:

- Indicate when a set of definitions for a particular suite included in the `'aeut'` resource is supported in its entirety by the application. For example, an application can indicate that it supports all of the `'aeut'` resource definitions for the Required and Core suites simply by identifying the suite as a whole in its `'aete'` resource; the detailed information for each standard suite does not need to be repeated.
- Describe extensions, if any, to the definitions included in the `'aeut'` resource, such as additional parameters for standard Apple events, additional properties and element classes for standard object classes, and additional key forms for each element class. For example, an application can indicate that it supports all of the definitions for the Required and Core suites included in the `'aeut'` resource, an additional parameter for one of the core events defined in the `'aeut'` resource, and an additional property for one of the core object classes defined in the `'aeut'` resource.
- Describe the standard Apple events and object classes that belong to suites the application does not support in their entirety.
- Describe the application's custom suite—that is, the application's custom Apple events and object classes, if any.

By specifying a suite ID, suite level, and suite version, your application can indicate that it supports an entire suite. Because the `'aeut'` resource provided by each scripting component lists the human-language terms for all the standard suites, you do not have to repeat this information if you support a suite in its entirety. If you support a subset of a standard suite, you must list all the Apple events, Apple event parameters, object classes, and so on and equivalent human-language terms for the parts of the suite your application does support.

You can include at most one `'aete'` resource per application or per module. The language code for this resource must match the language code of the language for which you are developing your application. Applications that support additional modules with their own `'aete'` resources must provide an `'scsz'` resource and handle the Get AETE event as described in "Handling the Get AETE Event," which begins on page 8-23.

**IMPORTANT**

Each human-language term supported by an application should correspond to a unique Apple event ID, keyword, or other code in either the application's `'aete'` resource or the `'aeut'` resource. For example, since the `'aeut'` resource defines "size" as the human-language equivalent for the property identified by the four-character code `'ptsz'` (the `pPointSize` property of text objects), an application's `'aete'` resource must not define "size" as the human-language equivalent for some other part of an Apple event or object class. However, more than one human-language term can correspond to the same Apple event ID or code. For example, an application's `'aete'` resource can define a second human-language term, "point size," that corresponds to the Apple event identifier `'ptsz'`. ▲

The *AppleScript Software Developers' Kit* (available from APDA) includes a tool that allows you to specify your application's support for Apple events and creates the equivalent `'aete'` resource. The previous section, "Structure of Apple Event Terminology Resources," describes the basic format used by both the `'aeut'` and `'aete'` resources.

The sections that follow provide examples of `'aete'` resources that can be generated with the tools in the *AppleScript Software Developers' Kit*.

## Supporting Standard Suites Without Extensions

To indicate that your application supports a standard suite in its entirety, without any extensions, your `'aete'` resource needs to provide only the information that identifies the suite. For example, Listing 8-2 shows the Rez input for an `'aete'` resource provided by an application that supports the entire Required and Core suites with no omissions or extensions.

Every `'aete'` resource must provide the major and minor version numbers for the content of the resource (1 and 0 in Listing 8-2) and the language code (English in Listing 8-2). For each suite that an application supports in its entirety, without extensions, the `'aete'` resource provides only the name, suite description, suite ID, suite level (1 for all current suites), and suite version (1 for all current suites). If the `'aete'` resource provides an empty string as the human-language name for such a

suite, a scripting component uses the name provided for the corresponding suite by the `'aeut'` resource. If an application does not extend or omit any of the definitions in a standard suite, a scripting component can get the rest of the information about the suite—its event and object class definitions, comparison operators, and enumerated groups—from the `'aeut'` resource. The corresponding arrays in the `'aete'` resource can therefore be left empty.

Note that the Rez input for resources does not include the `align word` fields shown in the `'aeut'` resource type declaration in Listing 8-1. Rez takes care of word alignment automatically.



**Listing 8-2** Rez input for an `'aete'` resource for an application that supports the Required and Core suites in their entirety

```
resource 'aete' (0, "JustTwoSuites") {
   1,                                    /*major version in BCD*/
   0,                                    /*minor version in BCD*/
   english,                              /*language code*/
   roman,                                /*script code*/
   {  /*array Suites: 2 elements*/
      /*[1]*/
      "",                                /*human-language name for suite; */
                                         /* 'aeut' supplies "Required Suite"*/
      "Events that every application should support", /*suite description*/
      kAERequiredSuite,                  /*suite code*/
      1,                                 /*suite level*/
      1,                                 /*suite version*/
      {  /*array Events: 0 elements*/
      },
      {  /*array Classes: 0 elements*/
      },
      {  /*array ComparisonOps: 0 elements*/
      },
      {  /*array Enumerations: 0 elements*/
      },
      /*[2]*/
      "",                                /*human-language name for suite; */
                                         /* 'aeut' supplies "Core Suite"*/
      "Suite that applies to all applications", /*suite description*/
      kAECoreSuite,                      /*suite code*/
      1,                                 /*suite level*/
      1,                                 /*suite version*/
      {  /*array Events: 0 element*/
      },
```

```
        {  /*array Classes: 0 elements*/
        },
        {  /*array ComparisonOps: 0 elements*/
        },
        {  /*array Enumerations: 0 elements*/
        }
    }
};
```

## Extending the Standard Suites

If, like the `'aete'` resource shown in Listing 8-2, your application's `'aete'` resource indicates that you support an entire standard suite, the scripting component automatically makes use of all the terminology for that suite provided by its `'aeut'` resource. For this reason, you can easily extend the definitions for a suite that your application supports in its entirety: just provide arrays in the `'aete'` resource for the definitions not already included in the `'aeut'` resource. For example, if you're extending the definition of an event to support a single additional parameter, you should provide an array of parameters that includes one item: the description of the new parameter. Similarly, if you're not extending the contents of a particular array, you don't need to include the array in the `'aete'` resource.

Although an array of descriptions in an `'aete'` resource need not include descriptions that are already provided by the `'aeut'` resource, you must include information that defines the position of the array in relation to the other information in the `'aete'` resource. As Table 8-3 on page 8-8 shows, you can nest the arrays in an `'aete'` resource within other arrays: for example, an array of parameters is part of the description of an event, and the event description is, in turn, part of the array of event descriptions for a suite.

To add a description of a single new parameter to the definition of an Apple event in a suite that your application supports in its entirety, you need to provide

- an array of parameters containing one element: the description of the new parameter
- information that identifies the event definition to which you're adding the parameter
- information that identifies the suite containing the event

Listing 8-3 illustrates how this works. This Rez input adds two new parameters ("number of copies" and "print quality") to the Print Documents event in the Required suite, one enumeration (three values for the "print quality" parameter of the Print Documents event) to the Required suite, and a new property ("first indent") to the `cParagraph` class in the Text suite. It also adds a plural synonym for the `cParagraph` class: the word "paragraphs."

**Listing 8-3** Rez input for an `'aete'` resource that extends the definitions of the Required, Core, and Text suites

```
#define keyMyNumberOfCopies   'numc'
#define keyMyPrintQuality     'prtq'
#define typePrintQuality      'pqen'
#define kFast                 'fast'
#define kNormal               'nrml'
#define kHighQuality          'hiql'
#define pFirstIndent          'indt'

resource 'aete' (0, "SuiteExtensions") {
   1,                                   /*major version in BCD*/
   0,                                   /*minor version in BCD*/
   english,                             /*language code*/
   roman,                               /*script code*/
   {  /*array Suites: 3 elements*/
      /*[1]*/
      "",                               /*human-language name for suite; */
                                        /* 'aeut' supplies "Required Suite"*/
      "Events that every application should support", /*suite description*/
      kAERequiredSuite,                 /*suite code*/
      1,                                /*suite level*/
      1,                                /*suite version*/
      {  /*array Events: 1 element*/
         /*[1]*/
         "",                            /*human-language name for event; */
                                        /* 'aeut' supplies "Print Documents"*/
         "Print the specified list of documents",     /*event description*/
         kCoreEventClass,               /*event class*/
         kAEPrintDocuments,             /*event ID*/
         noReply,                       /*reply type*/
         "",                            /*reply description*/
         replyOptional,                 /*reply is optional*/
         singleItem,                    /*reply must be single item*/
         notEnumerated,                 /*type is not enumerated*/
         reserved,                      /*these 13 bits are reserved*/
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
         reserved,
```

```
reserved,
reserved,
reserved,
reserved,
reserved,
typeAlias,                          /*direct parameter type*/
"List of documents to print",       /*direct parameter description*/
directParamRequired,                /*direct parameter is required*/
listOfItems,                        /*direct parameter must be list*/
notEnumerated,                      /*type is not enumerated*/
doesntChangeState,                  /*event does not change */
                                    /* server's state*/
reserved,                           /*these 12 bits are reserved*/
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
reserved,
{  /*array OtherParams: these fields describe the additional */
   /* parameters Number of Copies and Print Quality to the */
   /* standard Print Documents event*/
   /*[1]*/
   "number of copies",          /*human-language name for parameter*/
   keyMyNumberOfCopies,            /*parameter keyword*/
   typeShortInteger,               /*parameter type*/
   "Number of copies to print",    /*parameter description*/
   optional,                       /*parameter is optional*/
   singleItem,                     /*parameter must be single item*/
   notEnumerated,                  /*type is not enumerated*/
   reserved,                       /*these 13 bits are reserved*/
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
```

```
      reserved,
      reserved,
      reserved,
      reserved,
      reserved,
      /*[2]*/
      "print quality",           /*human-language name for parameter*/
      keyMyPrintQuality,            /*parameter keyword*/
      typePrintQuality,             /*parameter type*/
      "The quality of the printing",/*parameter description*/
      optional,                     /*parameter is optional*/
      singleItem,                   /*parameter must be single item*/
      enumerated,                   /*type is enumerated*/
      reserved,                     /*these 13 bits are reserved*/
      reserved,
      reserved,
      reserved,
      reserved,
      reserved,
      reserved,
      reserved,
      reserved,
      reserved,
      reserved,
      reserved,
      reserved
   }
},
{  /*array Classes: 0 elements*/
},
{  /*array ComparisonOps: 0 elements*/
},
{  /*array Enumerations: 1 element*/
   /*these fields add the 'pqen' enumeration to the Required suite*/
   /*[1]*/
   typePrintQuality,                /*enumeration ID*/
   {  /*array Enumerators: 3 elements*/
      /*[1]*/
      "Fast",                       /*enumerator name*/
      kFast,                        /*enumerator ID*/
      "Print as quickly as possible",/*enumerator description*/
      /*[2]*/
      "Normal",                     /*enumerator name*/
```

```
      kNormal,                      /*enumerator ID*/
      "Print at normal speed",      /*enumerator description*/
      /*[3]*/
      "High-Quality",               /*enumerator name*/
      kHighQuality,                 /*enumerator ID*/
      "Print at highest quality possible" /*enumerator description*/
   }
},
/*[2]*/
"",                                 /*human-language name for suite; */
                                    /* 'aeut' supplies "Core Suite"*/
"Suite that applies to all applications",       /*suite description*/
kAECoreSuite,                       /*suite code*/
1,                                  /*suite level*/
1,                                  /*suite version*/
{  /*array Events: 0 elements*/
},
{  /*array Classes: 0 elements*/
},
{  /*array ComparisonOps: 0 elements*/
},
{  /*array Enumerations: 0 elements*/
},
/*[3]*/
"",                                 /*human-language name for suite; */
                                    /* 'aeut' supplies "Text Suite"*/
"A set of basic classes for text processing",   /*suite description*/
kAETextSuite,                       /*suite code*/
1,                                  /*suite level*/
1,                                  /*suite version*/
{  /*array Events: 0 elements*/
},
{  /*array Classes: 1 element*/
   /*[1]*/
   "paragraph",                     /*human-language name for class*/
   cParagraph,                      /*class ID*/
   "A paragraph",                   /*class description*/
   {  /*array Properties: 1 element*/
      /*[1]*/
      "first indent",           /*human-language name for property*/
      pFirstIndent,                 /*property ID*/
      cLongInteger,                 /*property class*/
      "First indent of paragraph in points",/*property description*/
```

```
   reserved,                    /*reserved*/
   singleItem,                  /*property is single item*/
   notEnumerated,               /*type is not enumerated*/
   readWrite,                   /*property can be modified*/
   reserved,                    /*these 12 bits are reserved*/
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved
},
{  /*array Elements: 0 elements*/
},
"paragraphs",                 /*human-language name for class*/
cParagraph,                   /*class ID*/
"Every paragraph",            /*class description*/
{  /*array Special Properties: 1 element*/
   /*[1]*/
   "",                        /*human-language name for property*/
   kAESpecialClassProperties,    /*property ID*/
   cType,                        /*property class*/
   "",                           /*property description*/
   reserved,                     /*reserved*/
   singleItem,                   /*property is single item*/
   notEnumerated,                /*type is not enumerated*/
   readOnly,                     /*property cannot be modified*/
   reserved,                     /*these 11 bits are reserved*/
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
   reserved,
```

```
            plural                          /*human-language name is */
                                            /* plural form*/
         },
         {  /*array Elements: 0 elements*/
         },
      },
      {  /*array ComparisonOps: 0 elements*/
      },
      {  /*array Enumerations: 0 elements*/
      }
   }
};
```

In Listing 8-3, the possible values for the "print quality" parameter belong to an enumeration. This is indicated by the term `enumerated` in the parameter description. For this reason, the parameter type field contains the ID for the enumeration—`typePrintQuality`.

Listing 8-3 also adds a plural synonym for "paragraph" to the array of classes: the word "paragraphs." Note that this is listed as if it were an additional class, except that it also specifies `cParagraph` as the class ID. The first property listed for the synonym has property ID `kAESpecialClassProperties`. This property describes characteristics of the class as a whole; the last flag bit for this property is set to plural, indicating that the term `paragraphs` is a plural term for the specified class. This property must always be the first property listed for a class. For more information about the `kAESpecialClassProperties` property, see "Property Data," which begins on page 8-38.

An enumeration is described only by its ID; its declaration does not include a name or description field. However, a name, value, and description must be provided for each of the enumerators in an enumeration.

You can use the method illustrated in Listing 8-3 only to add to the definitions of Apple events and Apple event object classes, not to support subsets of them. For example, to support only a subset of the parameters of an Apple event or only some of the elements or properties of an existing object class, you must list all the definitions from that suite that you do support. The next section, "Supporting Subsets of Suites," provides more information about how to do this.

Human-language names for Apple events, object classes, and so on (including extensions) can include both uppercase and lowercase letters and spaces. For comparison purposes, case doesn't matter. However, note that the human-language names defined in Listing 8-3 are all lowercase. This convention ensures that scripts in which these terms appear won't have capital letters in unexpected places.

Scripting components that get identifiers or strings from user terminology resources are free to change the identifiers or strings as necessary (eliminating spaces, converting identifiers to all uppercase or lowercase, or changing the identifiers altogether) to meet the requirements of a particular task.

### Supporting Subsets of Suites

Your application is not required to support all the definitions in a suite. If you wish to support a subset of the definitions in one or more standard suites, you can collect individual definitions from any number of suites in a placeholder suite whose suite ID is the application's signature or `typeWildCard` (`'****'`). When you support a subset of a suite, you must provide all the definitions you want to support in your `'aete'` resource.

### Supporting New Suites

If your application defines its own custom Apple events or other Apple event constructs, you should include a separate suite section for the suite in the `'aete'` resource. You should use your application's signature for both the suite ID and the class ID of all events in the suite.

## Handling the Get AETE Event

A scripting component sends the Get AETE event to an application when it needs information about the user terminology specified by the application. For example, the AppleScript component sends the Get AETE event when it first attempts to compile a `tell` statement that specifies a particular application. If your application does not handle the Get AETE event, the scripting component reads the terminology information it needs directly from your application's `'aete'` resource. Applications that support additional plug-in modules, each with its own `'aete'` resource, must provide an `'scsz'` resource and a handler for the Get AETE event that collects the `'aete'` resources from the modules that are currently running.

If your application does provide separate plug-in modules, the Get AETE event allows it to gather information from the `'aete'` resources for the modules that are currently running and return the terminology information along with your application's built-in terminology information to the scripting component in the reply event.

Here is a summary of the structure of a Get AETE event:

**Get AETE—Get an application's `'aete'` resource**

| | |
|---|---|
| Event class | `kASAppleScriptClass` |
| Event ID | `kGetAETE` |
| Required parameter | |
| Keyword: | `keyDirectObject` |
| Descriptor type: | `typeInteger` |
| Data: | Language code |
| Required reply parameter | |
| Keyword: | `keyDirectObject` |
| Descriptor type: | `typeAEList` or `typeAETE` |
| Data: | The application's terminologies |
| Description | Sent by a scripting component to an application when the scripting component needs information about the application's user terminology |

Your application can't handle the Get AETE event unless it is running. If your application doesn't provide a handler for the Get AETE event, the scripting component can obtain terminology information directly from your application's `'aete'` resource even if your application is not running.

If your application handles the Get AETE event, it must also provide a scripting size resource. A scripting size resource is a resource of type `'scsz'` that provides information about an application's capabilities for use by scripting components. It allows your application to declare whether it needs the Get AETE event and to specify preferences for the sizes of the portion of your application's heap used by a scripting component. For information about the `'scsz'` resource, see "The Scripting Size Resource" on page 8-45.

A handler for the Get AETE event should perform the following tasks:

- Obtain the language code specified by the event.
- Create a descriptor list to hold the `'aete'` resources.
- Collect the `'aete'` resources from all the application's plug-in modules that are currently running, including the application itself, and add them to the list.
- Add the list to the reply Apple event.

Listing 8-4 provides an example of a handler for the Get AETE event.

**Listing 8-4** A handler for the Get AETE event

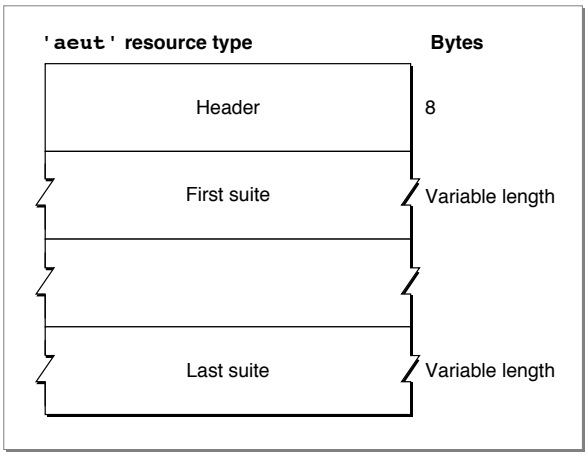
```
FUNCTION MyGetAETE (theAE: AppleEvent; theReply: AppleEvent;
                    refCon: LongInt): OSErr;
VAR
   theList:       AEDescList;
   returnedType:  DescType;
   actualSize:    Size;
   languageCode:  Integer;
   myErr:         OSErr;
BEGIN
   MyGetAETE := errAEEventNotHandled;
   languageCode := 0;
   {if a reply was not requested, then don't handle}
   IF theReply.dataHandle = NIL THEN
      Exit(MyGetAETE);
   {get the language code that AppleScript is requesting so that }
   { this function can return the aete of a specified language}
   myErr := AEGetParamPtr(theAE, keyDirectObject,
                          typeLongInteger, returnedType,
                          @languageCode, sizeOf(LongInt),
                          actualSize);
   IF myErr <> noErr THEN
      Exit(MyGetAETE);
   {create a list}
   myErr := AECreateList(NIL, 0, FALSE, theList);
   IF myErr <> noErr THEN
      Exit(MyGetAETE);
   {get the requested 'aete' resources and put in the list--the }
   { MyGrabAETE application-defined function does this}
   {your code should iterate all of your installed code }
   { extensions and add the aete for each that matches the }
   { language code requested}
   myErr := MyGrabAETE(languageCode, theList);
   IF myErr <> noErr THEN
   BEGIN
      myErr := AEDisposeDesc(theList);
      Exit(MyGetAETE);
   END;
   {add list to reply Apple event}
   myErr := AEPutParamDesc(theReply, keyDirectObject, theList);
   myErr := AEDisposeDesc(theList);
   myGetAETE := myErr;
END;
```

The `MyGetAETE` handler in Listing 8-4 begins by setting the function result to `errAEEventNotHandled`. The function is set to this result if for any reason the handler doesn't successfully handle the event, so that a system handler provided by the scripting component can at least read the terminology information directly from the application's own `'aete'` resource. The handler in Listing 8-4 then checks the language code specified by the event. After checking to make sure the reply exists, the handler creates a list and uses the application-defined function `MyGrabAETE` to collect all the appropriate terminology information and append it to the list. The `MyGetAETE` handler then adds the list to the reply event.

# Reference to Apple Event Terminology Resources

Listing 8-1 on page 8-9 shows the complete resource type declaration in Rez format for the `'aeut'` resource. The same resource structure is used by both the `'aeut'` and `'aete'` resources. Figure 8-1 shows the format of a compiled `'aeut'` or `'aete'` resource.

**Figure 8-1** Structure of an `'aeut'` or `'aete'` resource




An `'aeut'` or `'aete'` resource contains the following:

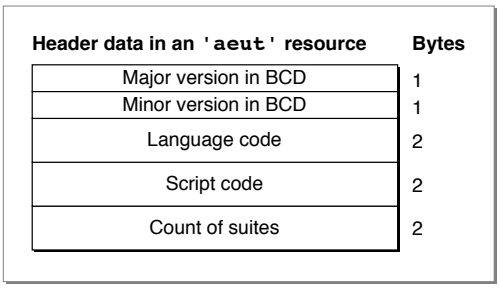
- a header containing the version and language code of the template and a count of the number of suites the resource describes
- a variable number of suite descriptions

The sections that follow describe the content of the header and each suite description in detail.

## Header Data for an Apple Event Terminology Resource

The header for an `'aeut'` or `'aete'` resource specifies the version of its contents, the language of the human-language equivalents contained in the resource, a script code, and a count of the number of suites the resource describes. Figure 8-2 shows the header format.

**Figure 8-2** Structure of the header data in an `'aeut'` or `'aete'` resource



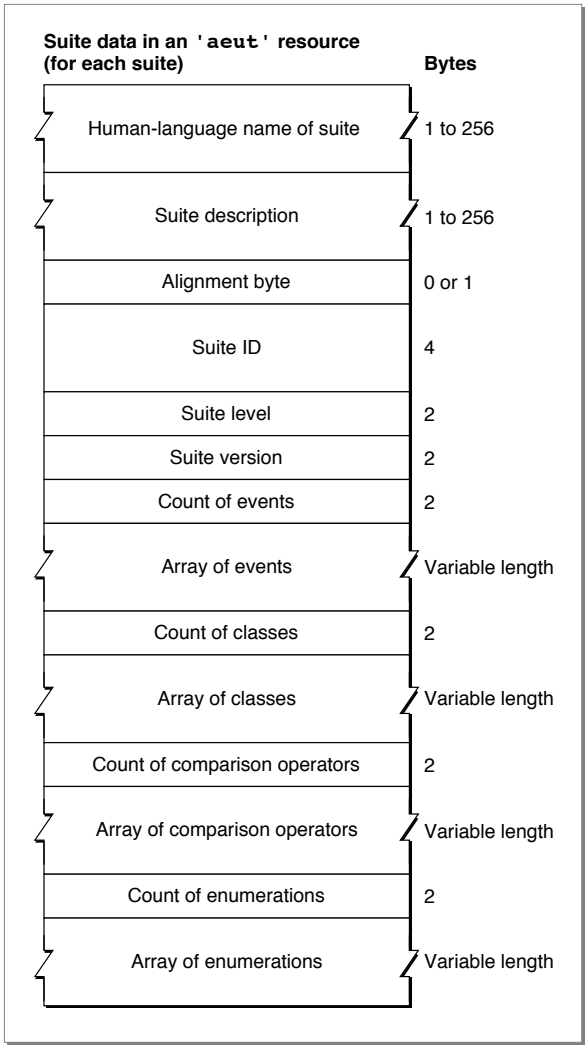

The header contains the following items:

- The major version number of the content of the resource in binary-coded decimal (the major version number for the first release of the `'aeut'` resource is 1). The major and minor versions describe the content of the resource, not its template. You can use these fields to provide version numbers for the content of your application's `'aete'` resource.
- The minor version number of the template in binary-coded decimal (the minor version number for the first release of the `'aeut'` resource is 0).
- The language code for the resource. *Inside Macintosh: Text* provides a list of language codes. This code must be the same as the resource ID for the resource.
- The script code for the resource, taken from the list of script codes provided in *Inside Macintosh: Text*.
- A count of the number of suites described by the resource.

## Suite Data for an Apple Event Terminology Resource

Each item in the array of suites for an `'aeut'` or `'aete'` resource includes information about the suite ID, level, and version and four arrays that specify the events, object classes, comparison operators, and enumerations for that suite. Figure 8-3 shows the format of this suite data.

**Figure 8-3** Structure of suite data in an `'aeut'` or `'aete'` resource




The data for each suite consists of the following items:

- The human-language name of the suite. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. If the `'aete'` resource specifies the name as an empty string, the scripting component looks up, in its `'aeut'` resource, the suite name and other suite data that correspond to the specified suite ID, suite level, and suite version. This strategy simplifies specification of an entire suite and facilitates localization, since the human-language name is provided by the `'aeut'` resource.

If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same suite ID, suite level, and suite version, the scripting component uses the new name with the same suite data from the `'aeut'` resource. Unless you are defining a custom suite, you should specify an empty string for the name of a suite.

- A human-language description of the suite. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.
- A four-character ID that distinguishes the suite from all other suites defined in either the `'aeut'` or `'aete'` resources. This value is normally the same as the event class for the Apple events in the suite.

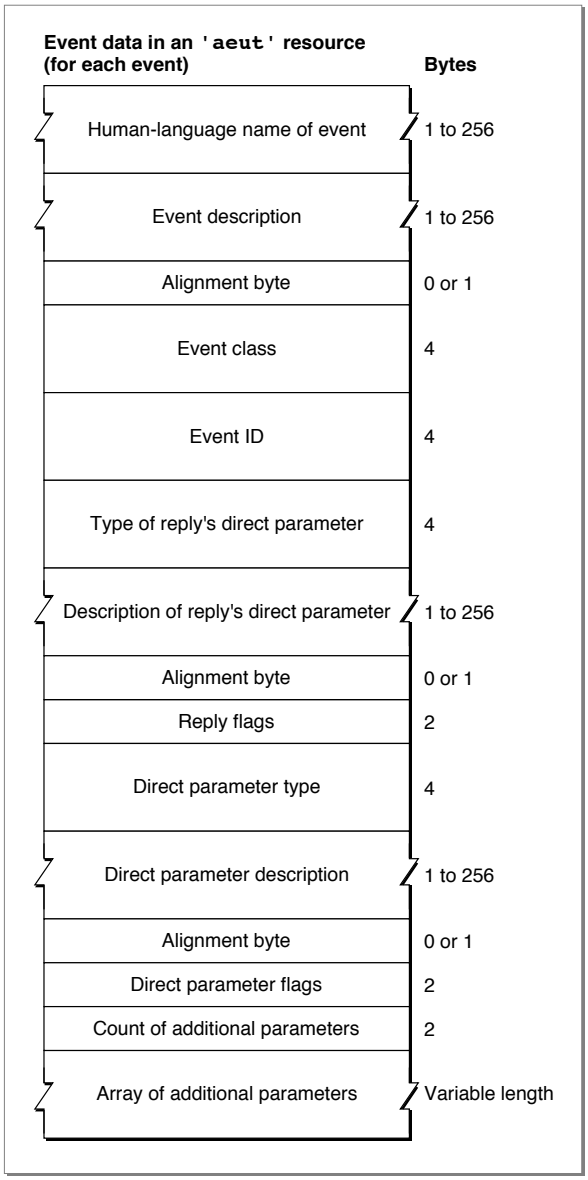
  If the `'aete'` resource specifies a standard suite name but a suite ID that is different from the suite ID for the standard suite of that name described in the `'aeut'` resource, the scripting component uses the new suite ID with the standard suite data for the specified name. In general, you should use the standard suite ID for any standard suite that you support.

  If your application uses a custom suite, you should use your application's signature as the event class for the events in the suite and, in addition, as its suite ID. When you register your application's signature with Developer Technical Support, the corresponding event class is automatically registered for your application, and only you can register events that belong to that event class. For information about registering Apple events, contact the Apple Event Registrar.
- The level and version of the suite. For the first version of any suite, the level is usually 1 (indicating that it is the suite that contains the most basic definitions) and the version is 1 (the version of this suite level). More advanced suites (such as a suite for performing more sophisticated text manipulation than the current Text suite allows) will have level numbers greater than 1. All currently defined suites have a level of 1 and a version of 1.
- A count of the events defined for this suite and an array of event definitions.
- A count of the object classes defined for this suite and an array of class definitions.
- A count of the comparison operators defined for this suite and an array of comparison operator definitions.
- A count of the enumerations defined for this suite and an array of enumeration definitions.

## Event Data

Each item in the array of events for a suite specified in an `'aeut'` or `'aete'` resource includes information about the event, the reply, and the direct parameter, and an array that specifies the additional parameters for the event. Figure 8-4 shows the format of this event data.

**Figure 8-4** Structure of event data in an `'aeut'` or `'aete'` resource



The data for each event consists of the following items:

- The human-language name of the event. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. If the `'aete'` resource specifies the name as an empty string, the scripting component looks up, in its `'aeut'` resource, the event name and other event data that correspond to the specified event class and event ID. This strategy facilitates localization, since the human-language name is provided by the `'aeut'` resource. In this case the scripting component will use the standard data from the `'aeut'` resource for the event plus the data provided by the `'aete'` resource for any additional parameters.

  If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same event class and event ID, the scripting component uses the new name with the same suite data from the `'aeut'` resource. You should specify an empty string for the name of any standard event that your application lists explicitly in its `'aete'` resource.

- A human-language description of the event. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

- The four-character event class for the event. If the `'aete'` resource specifies a standard event name and an event class other than the event class for the equivalent standard event, the scripting component uses the new event class with the standard event data for the specified name. You should specify the standard event class for any standard event that your application lists explicitly in its `'aete'` resource.

- The four-character event ID for the event. If the `'aete'` resource specifies a standard event name and an event ID other than the event ID for the equivalent standard event, the scripting component uses the new event ID with the standard event data for the specified name. You should specify the standard event ID for any standard event that your application lists explicitly in its `'aete'` resource.

- A four-character descriptor type for the direct parameter of the reply. If the event never needs a reply, or if the reply does not include a direct parameter, this value must be `typeNull`. Otherwise, the meaning of this field varies according to the values of two of the flags that follow. One flag specifies whether the parameter is a list (`singleItem` or `listOfItems`), and the other specifies whether the values for the parameter are enumerated (`enumerated` or `notEnumerated`):
  - If the parameter is not a list and its values are not enumerated, this value is the descriptor type for the direct parameter.
  - If the parameter is a list and its values are not enumerated, this value is the descriptor type for each of the items in the list. (If not all the items in the list are of the same descriptor type, the flag specifying whether the value is a list must have the value `singleItem`, and the value of this field must be `typeAEList`.)

- If the parameter is not a list and its values are enumerated, this value is the four-character code for the enumeration defined in either the `'aete'` or `'aeut'` resource that contains the allowable values for the parameter. (If the values are enumerated but the enumeration is not defined in either the `'aete'` or `'aeut'` resource, the flag specifying whether the parameter's values are enumerated must have the value `notEnumerated`, and the value of this field must be `typeEnumerated`.)
- If the parameter is a list and its values are enumerated, this value is the four-character code for the enumeration defined in the same resource that contains the allowable values for all of the items in the list. All items in the list must have one of these enumerated values.

- A human-language description of the direct parameter of the reply. This is a Pascal string that can include any characters. Although the reply may include other parameters, only the direct parameter of the reply is described here. When the resource description is compiled, the resource compiler aligns the string on a word boundary.
- Flags that specify the following as Boolean values:
  - Whether the direct parameter of the reply is required (`replyRequired`) or optional (`replyOptional`).
  - Whether the direct parameter of the reply is a single item (`singleItem`) or a list of items (`listOfItems`). (See the earlier description of the reply event's four-character descriptor type for information about how this value changes the meaning of the reply type.)
  - Whether named constants, called enumerators, are specified as the only valid values for the direct parameter of the reply (`enumerated` or `notEnumerated`). (See the earlier description of the four-character descriptor type for the reply event's direct parameter for information about how this value changes the meaning of the direct parameter type.) For information about specifying enumerators, see "Enumeration and Enumerator Data" on page 8-43.
  - Following 5 bits are reserved for future use. The values of these bits must be set to `reserved`.
  - Following 7 bits are reserved for future use as dialect-specific flags. The values of these bits must be set to `reserved`.
  - Whether the event is a nonverb event (`nonVerbEvent`). This bit is used by dialects such as the AppleScript Japanese dialect that make this distinction. For all other dialects, set the value of this bit to `reserved`.
- A four-character descriptor type for the direct parameter of the event. If the event never has a direct parameter, this value must be `typeNull`. Otherwise, the meaning of this field varies according to the values of two of the flags that follow. One flag specifies whether the parameter is a list (`singleItem` or `listOfItems`), and the other specifies whether the values for the parameter are enumerated (`enumerated` or `notEnumerated`):

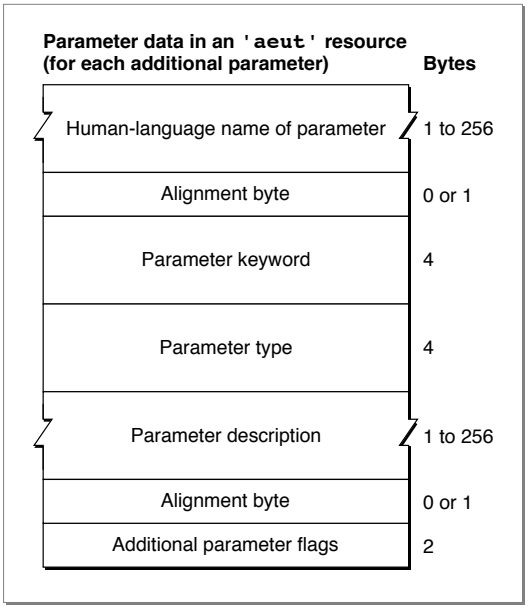
- If the parameter is not a list and its values are not enumerated, this value is the descriptor type for the direct parameter.
- If the parameter is a list and its values are not enumerated, this value is the descriptor type for each of the items in the list. (If not all the items in the list are of the same descriptor type, the flag specifying whether the value is a list must have the value `singleItem`, and the value of this field must be `typeAEList`.)
- If the parameter is not a list and its values are enumerated, this value is the four-character code for the enumeration defined in either the `'aete'` or `'aeut'` resource that contains the allowable values for the parameter. (If the values are enumerated but the enumeration is not defined in either the `'aete'` or `'aeut'` resource, the flag specifying whether the parameter's values are enumerated must have the value `notEnumerated`, and the value of this field must be `typeEnumerated`.)
- If the parameter is a list and its values are enumerated, this value is the four-character code for the enumeration defined in the same resource that contains the allowable values for all of the items in the list. The values of the items in the list must all be one of these enumerated values.

- A human-language description of the direct parameter. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.
- Flags that specify the following as Boolean values:
  - Whether the direct parameter of the event is required (`directParamRequired`) or optional (`directParamOptional`).
  - Whether the direct parameter of the event is a single item (`singleItem`) or a list of items (`listOfItems`). (See the earlier description of the direct parameter's four-character descriptor type for information about how this value changes the meaning of the direct parameter type.)
  - Whether named constants, called enumerators, are specified as the only valid values for the direct parameter (`enumerated` or `notEnumerated`). (See the earlier description of the direct parameter's four-character descriptor type for information about how this value changes the meaning of the direct parameter type.) For information about specifying enumerators, see "Enumeration and Enumerator Data" on page 8-43.
  - Whether receiving this event changes (`changesState`) or doesn't change (`doesntChangeState`) the internal state of the receiving application. Events that only get information do not change the state of the application, whereas events such as Cut and Move do.
  - Following 4 bits are reserved for future use. The values of these bits must be set to `reserved`.
  - Following 8 bits are reserved for future use as dialect-specific flags. The values of these bits must be set to `reserved`.
- A count of the additional parameters described for this event and an array of additional parameter definitions.

### Additional Parameter Data

Each item in the array of additional parameters for an event specified in an `'aeut'` resource includes information about a single additional parameter. Figure 8-5 shows the format of additional parameter data in an `'aeut'` or `'aete'` resource.

**Figure 8-5** Structure of additional parameter data in an `'aeut'` or `'aete'` resource




The data for each additional parameter consists of the following items:

- The human-language name of the parameter. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

  If the `'aete'` resource specifies the name of an additional parameter as an empty string, the scripting component looks up, in its `'aeut'` resource, the parameter name and other parameter data that correspond to the specified parameter keyword. If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same parameter keyword, the scripting component uses the new name with the same parameter data from the `'aeut'` resource. You should specify an empty string for the name of any standard additional parameter that you list explicitly in an `'aete'` resource.

- The four-character keyword for the parameter. If the `'aete'` resource specifies a standard parameter name and a parameter keyword other than the keyword for the equivalent standard parameter, the scripting component uses the new parameter keyword with the standard parameter data for the specified name. You should specify the standard parameter keyword for any standard additional parameter that you list explicitly in an `'aete'` resource.
- A four-character descriptor type for the parameter. The meaning of this field varies according to the values of two of the flags that follow. One flag specifies whether the parameter is a list (`singleItem` or `listOfItems`), and the other specifies whether the values for the parameter are enumerated (`enumerated` or `notEnumerated`):
  - If the parameter is not a list and its values are not enumerated, this value is the descriptor type for the direct parameter.
  - If the parameter is a list and its values are not enumerated, this value is the descriptor type for each of the items in the list. (If not all the items in the list are of the same descriptor type, the flag specifying whether the value is a list must have the value `singleItem`, and the value of this field must be `typeAEList`.)
  - If the parameter is not a list and its values are enumerated, this value is the four-character code for the enumeration defined in either the `'aete'` or `'aeut'` resource that contains the allowable values for the parameter. (If the values are enumerated but the enumeration is not defined in either the `'aete'` or `'aeut'` resource, the flag specifying whether the parameter's values are enumerated must have the value `notEnumerated`, and the value of this field must be `typeEnumerated`.)
  - If the parameter is a list and its values are enumerated, this value is the four-character code for the enumeration defined in the same resource that contains the allowable values for all of the items in the list. The values of the items in the list must all be one of these enumerated values.
- A human-language description of the parameter. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.
- Flags that specify the following as Boolean values:
  - Whether the parameter is required (`required`) or optional (`optional`).
  - Whether the parameter is a single item (`singleItem`) or a list of items (`listOfItems`). (See the earlier description of the additional parameter's four-character descriptor type for information about how this value changes the meaning of the parameter type.)
  - Whether named constants, called enumerators, are specified as the only valid values for the parameter (`enumerated` or `notEnumerated`). (See the earlier description of the parameter's four-character descriptor type for information about how this value changes the meaning of the parameter type.) For information about specifying enumerators, see "Enumeration and Enumerator Data" on page 8-43.

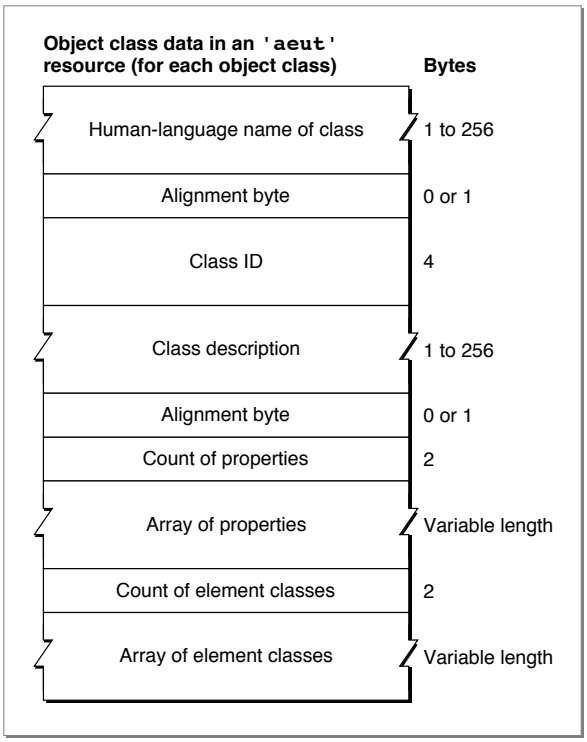
- Whether the parameter is the event's only unnamed parameter (`isUnNamed`) or is named (`isNamed`). This bit is used by dialects such as AppleScript Japanese that make this distinction. For all other dialects, set the value of this bit to `reserved`.
- Following 4 bits are reserved for future use. The values of these bits must be set to `reserved`.
- Following 8 bits are reserved for future use as dialect-specific flags. The values of these bits must be set to `reserved`.

"Extending the Standard Suites," which begins on page 8-16, includes sample Rez input for an `'aete'` resource that adds new parameters to a standard Apple event.

## Object Class Data

Each item in the array of object classes for a suite includes information about the class and arrays that specify the properties and elements for that class. Figure 8-6 shows the format of the object class data in an `'aeut'` or `'aete'` resource.

**Figure 8-6** Structure of object class data in an `'aeut'` or `'aete'` resource



The data for each object class consists of the following items:

- The human-language name of the object class. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

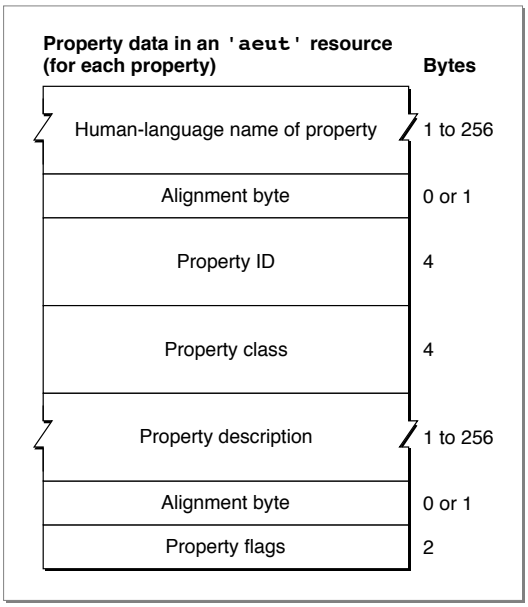
  If the `'aete'` resource specifies the name of an object class as an empty string, the scripting component looks up, in its `'aeut'` resource, the class name and other object class data that correspond to the specified class ID. If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same class ID, the scripting component uses the new name with the same object class data from the `'aeut'` resource. You should specify an empty string for the name of any standard object class that you list explicitly in an `'aete'` resource.
- The four-character class ID for the object class. If the `'aete'` resource specifies a standard object class name and a class ID other than the class ID for the equivalent standard object class, the scripting component uses the new class ID with the standard object class data for the specified name. You should specify the standard class ID for any standard object class that you list explicitly in an `'aete'` resource.
- A human-language description of the class. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.
- A count of the properties described for this class and an array of property definitions.
- A count of the element classes described for this class and an array of element class definitions.

To define characteristics of an object class (for instance, whether an object of that class is a single item or a list of items, whether it is singular or plural, and so on), your application's `'aete'` resource must define a special property of property ID `kAESpecialClassProperties` as the first property in the array of properties. Because object class data does not include flag bits, the flag bits of this property are used to specify attributes for the class to which the property belongs. The next section describes how this property is defined and used.

### Property Data

Each item in the array of properties for an object class includes information about a single property. Figure 8-7 shows the format of the property data in an `'aeut'` or `'aete'` resource.

**Figure 8-7** Structure of property data in an `'aeut'` or `'aete'` resource




The data for each property consists of the following items:

- The human-language name of the property. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

  If the `'aete'` resource specifies the name of a property as an empty string, the scripting component looks up, in its `'aeut'` resource, the property name and other property data that correspond to the specified property ID. If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same property ID, the scripting component uses the new name with the same property data from the `'aeut'` resource. You should specify an empty string for the name of any standard property that you list explicitly in an `'aete'` resource.

- The four-character property ID for the property. If the `'aete'` resource specifies a standard property name and a property ID other than the property ID for the equivalent standard property, the scripting component uses the new property ID with the standard property data for the specified name. You should specify the standard property ID for any standard property that you list explicitly in an `'aete'` resource.
- A four-character class ID for the object class to which the property belongs. The meaning of this field varies according to the values of two of the flags that follow. One flag specifies whether the property is a list (`singleItem` or `listOfItems`), and the other specifies whether the values for the parameter are enumerated (`enumerated` or `notEnumerated`):
  - If the property is not a list and its values are not enumerated, this value is the class ID for the property.
  - If the property is a list and its values are not enumerated, this value is the class ID for each of the items in the list. (If not all the items in the list are of the same descriptor type, the flag specifying whether the value is a list must have the value `singleItem`, and the value of this field must be `cAEList`.)
  - If the property is not a list and its values are enumerated, this value is the four-character code for the enumeration defined in either the `'aete'` or `'aeut'` resource that contains the allowable values for the property. (If the values are enumerated but the enumeration is not defined in either the `'aete'` or `'aeut'` resource, the flag specifying whether the property's values are enumerated must have the value `notEnumerated`, and the value of this field must be `typeEnumerated`.)
  - If the parameter is a list and its values are enumerated, this value is the four-character code for the enumeration defined in the same resource that contains the allowable values for all of the items in the list. The values of the items in the list must all be one of these enumerated values.
- A human-language description of the property. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.
- Flags that specify the following as Boolean values:
  - The first bit is reserved for future use. Its value must be set to `reserved`.
  - Whether the property is a single item (`singleItem`) or a list of items (`listOfItems`). (See the earlier description of the property's four-character class ID for information about how this value changes the meaning of the class ID.)
  - Whether named constants, called enumerators, are specified as the only valid values for the property (`enumerated` or `notEnumerated`). (See the earlier description of the property's four-character class ID for information about how this value changes the meaning of the class ID.) For information about specifying enumerators, see "Enumeration and Enumerator Data" on page 8-43.

- Whether the property's value can (`readWrite`) or cannot (`readOnly`) be set by the Set Data Apple event.
- Following 4 bits are reserved for future use.
- Following 5 bits are reserved for future use as dialect-specific flags.
- Whether the human-language name of the property is feminine (`feminine`) or not (`notFeminine`). This bit is used by dialects such as the AppleScript French dialect that make this distinction. For all other dialects, set the value of this bit to `reserved`.
- Whether the human-language name of the property is masculine (`masculine`) or not (`notMasculine`). This bit is used by dialects such as AppleScript French that make this distinction. For all other dialects, set the value of this bit to `reserved`.
- Whether the human-language name of the property is singular (`singular`) or plural (`plural`). This bit is used by dialects such as AppleScript French that make this distinction. If you set this bit to `reserved`, the scripting component will assign it the value `singular`.

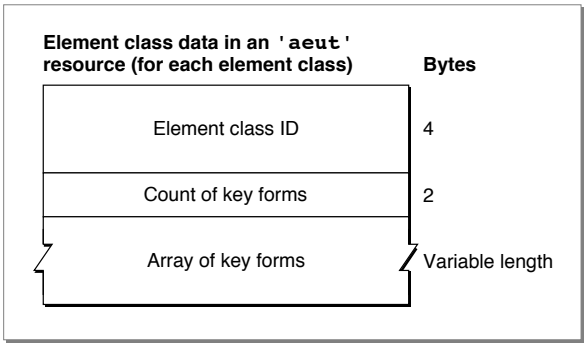
"Extending the Standard Suites," which begins on page 8-16, includes sample Rez input for an `'aete'` resource that adds a new property to a standard object class.

The array of properties in an `'aeut'` resource begins with a definition of a special property that describes characteristics of the class as a whole using the flags in the definition of that property. A property used in this way to define characteristics of a class must be defined first in the array of properties for that class and must specify `kAESpecialClassProperties` (`'c@#!'`) as the property ID, `cType` as the property class, and an empty string for the property name and property description. If you don't define such a property for a class in your application's `'aete'` resource, the scripting component will assign that class the default values specified by the first constant for each flag bit in the Rez declaration for the `'aeut'` resource. (See Listing 8-1, which begins on page 8-9, for the `'aeut'` resource type declaration.)

### Element Class Data

Each item in the array of elements for an object class includes information about a single element class and an array of key forms for that element class. Figure 8-8 shows the format of the object class data in an `'aeut'` or `'aete'` resource.

**Figure 8-8** Structure of element class data in an `'aeut'` or `'aete'` resource




The following statements are included for each element class in the array of element classes for an object class:

- The four-character class ID for the element's object class.
- A count of key forms that apply to elements of this class within objects of the class for which these element classes are defined, followed by an array of key forms. Each item in the array must be a value from a special `'kfrm'` enumeration. (The `'aeut'` resource includes enumerators for the standard key forms defined in the *Apple Event Registry: Standard Suites;* an `'aete'` resource can contain `'kfrm'` enumerators for additional key forms that are specific to an application. For information about defining enumerators and enumerations, see "Enumeration and Enumerator Data" on page 8-43.) The enumerators for a `'kfrm'` enumeration can include `'indx'` (for the key form `formAbsolutePosition`), `'name'` (for the key form `formName`), `'ID  '` (for the key form `formUniqueID`), `'prop'` (for the key form `formPropertyID`), `'rang'` (for the key form `formRange`), `'rele'` (for the key form `formRelativePosition`), and `'test'` (for the key form `formTest`).

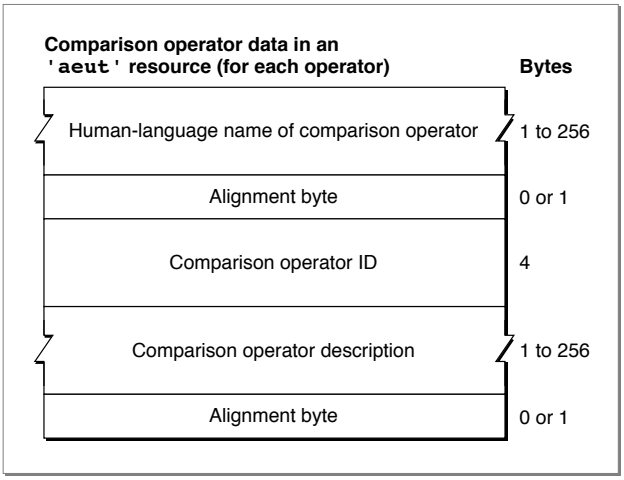
No names or descriptions are provided for element classes, because elements are specified by their object classes, and the declaration of each object class includes the name and description of the class.

## Comparison Operator Data

Each item in the array of comparison operators for a suite includes information about a single comparison operator. Figure 8-9 shows the format of the comparison operator data in an `'aeut'` or `'aete'` resource.

**Note**
The AppleScript component currently doesn't use information about comparison operators. Other scripting components may use this information. ◆

**Figure 8-9** Structure of comparison operator data in an `'aeut'` or `'aete'` resource




The data for each comparison operator consists of the following items:

- The human-language name of the comparison operator. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

  If the `'aete'` resource specifies the name of a comparison operator as an empty string, the scripting component looks up, in its `'aeut'` resource, the comparison operator name and other comparison operator data that correspond to the specified comparison operator ID. If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same comparison operator ID, the scripting component uses the new name with the same comparison operator data from the `'aeut'` resource. You should specify an empty string for the name of any standard comparison operator that you list explicitly in an `'aete'` resource.

- The four-character comparison operator ID for the property. If the `'aete'` resource specifies a standard comparison operator name and a comparison operator ID other than the comparison operator ID for the equivalent standard comparison operator, the scripting component uses the new comparison operator ID with the standard comparison operator data for the specified name. You should specify the standard comparison operator ID for any standard comparison operator that you list explicitly in an `'aete'` resource.
- A human-language description of the comparison operator. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

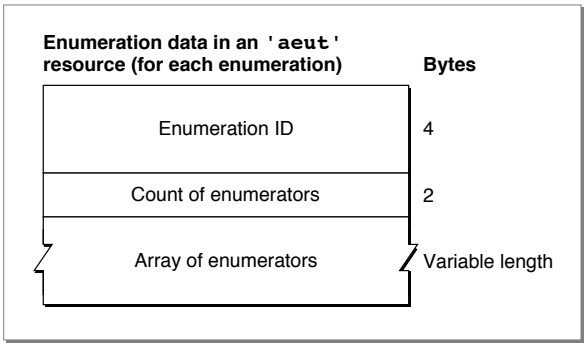
"Extending the Standard Suites," which begins on page 8-16, includes sample Rez input for an `'aete'` resource that adds a comparison operator to a standard suite.

## Enumeration and Enumerator Data

Each item in the array of enumerations for a suite includes information about a single enumeration and an array of enumerators for that enumeration.

Figure 8-10 shows the format of the enumeration data in an `'aeut'` or `'aete'` resource.

**Figure 8-10** Structure of enumeration data in an `'aeut'` or `'aete'` resource

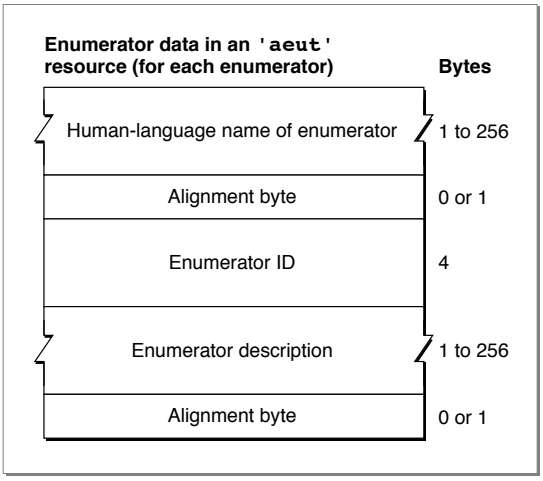



The data for each enumeration consists of the following items:

- a four-character enumeration ID
- a count of constants, known as enumerators, that specify the allowable values for the enumeration, and an array of enumerators

Figure 8-11 shows the format of the enumerator data.

**Figure 8-11** Structure of enumerator data in an `'aeut'` or `'aete'` resource




The data for each enumerator consists of the following items:

- The human-language name of the enumerator. This is a Pascal string that can include any characters, including uppercase and lowercase letters and spaces. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

  If the `'aete'` resource specifies the name of an enumerator as an empty string, the scripting component looks up, in its `'aeut'` resource, the enumerator name and other enumerator data that correspond to the specified enumerator ID. If the `'aete'` resource specifies a name other than the name provided by the `'aeut'` resource for the same enumerator ID, the scripting component uses the new name with the same enumerator data from the `'aeut'` resource. You should specify an empty string for the name of any standard enumerator that you list explicitly in an `'aete'` resource.
- The four-character enumerator ID for the enumerator. If the `'aete'` resource specifies a standard enumerator name and an enumerator ID other than the enumerator ID for the equivalent standard enumerator, the scripting component uses the new enumerator ID with the standard enumerator data for the specified name. You should specify the standard enumerator ID for any standard enumerator that you list explicitly in an `'aete'` resource.
- A human-language description of the enumerator. This is a Pascal string that can include any characters. When the resource description is compiled, the resource compiler pads the string and aligns the next field on a word boundary.

"Extending the Standard Suites," which begins on page 8-16, includes sample Rez input for an `'aete'` resource that specifies an enumeration and an array of enumerators.

## The Scripting Size Resource

If your application handles the Get AETE event, you must provide a scripting size resource. A scripting size resource is a resource of type `'scsz'` that provides information about an application's capabilities for use by scripting components. It also allows your application to specify preferences for the sizes of the portion of your application's heap used by a scripting component for its application-specific heap and stack.

Listing 8-5 shows the resource type declaration in Rez format for the `'scsz'` resource.

**Listing 8-5** Resource type declaration for the `'scsz'` resource

```
type 'scsz' {
   boolean  dontReadExtensionTerms, /*if application needs */
            readExtensionTerms;     /* Get AETE event*/
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;
   boolean  reserved;

/*memory sizes are in bytes; 0 means use default*/
   unsigned longint  minStackSize;        /*minimum stack size*/
   unsigned longint  preferredStackSize;  /*preferred stack size*/
   unsigned longint  maxStackSize;        /*maximum stack size*/
   unsigned longint  minHeapSize;         /*minimum heap size*/
   unsigned longint  preferredHeapSize;   /*preferred stack size*/
   unsigned longint  maxHeapSize;         /*maximum heap size*/
};
```

The data for an `'scsz'` resource consists of the following items:

- Flags that specify Boolean values:
  - Whether the scripting component should (`readExtensionTerms`) or shouldn't (`dontReadExtensionTerms`) read the application's terminology information directly from its `'aete'` resource. If the application is not running, this flag allows a scripting component to determine whether it should read the application's terminology information without sending it a Get AETE event.
  - The following 15 bits are reserved for future use. Their values must be set to `reserved`.
- The minimum size for the portion of the application's heap used by the scripting component's application-specific stack
- The preferred size for the portion of the application's heap used by the scripting component's application-specific stack
- The maximum size for the portion of the application's heap used by the scripting component's application-specific stack
- The minimum size for the portion of the application's heap used by the scripting component's application-specific heap
- The preferred size for the portion of the application's heap used by the scripting component's application-specific heap
- The maximum size for the portion of the application's heap used by the scripting component's application-specific heap

If you specify 0 for any of the fields that specify memory size or number of script IDs, the scripting component uses its own default values for those fields.

The AppleScript component provides a function, `ASInit`, that allows your application to initialize the component with desired values for memory sizes or number of script IDs. If your application doesn't call `ASInit`, the AppleScript component initializes itself using either the values specified in the application's `'scsz'` resource or, for those values not provided by the `'scsz'` resource, default values provided by the AppleScript component. For more information about `ASInit`, see "Initializing AppleScript" on page 10-80.