Monte

Software Development Kit

version 1.0.3

April, 2001

About This Document

This document describes the Monte data editing technology.

1. Introduction

Monte is a library of functions for decomposing packed binary data structures into their constituent pieces for the puposes of editing, localisation, or meta-data creation.

The Monte library is designed to help developers of data editing and localisation tools by providing a consistent interface to a wide variety of binary data formats.

Overview

The key features of the Monte library are:

- *Object-oriented data description*: Monte data translators are specified using a simple, object-oriented, data description language which supports encapsulation and inheritance. These features make the Monte data description language particularly suitable for describing binary structures which contain serialised objects, such as those used by many popular development frameworks.

- *Unicode/ISO 10646 support*: Monte understands data structures containing Unicode/ISO 10646-encoded text.

- *Platform-independent internal data representation*: Once decomposed, data is presented to the calling application in a platform independent manner. For example, all textual data is presented as Unicode/ISO 10646-encoded strings, regardless of its original, packed, encoding.

- *Data Validation*: The data-editing features of Monte provide a mechanism whereby changes to data are validated before being applied. For instance, text which is too long for its field, or unencodable in the field's native text encoding, is prevented from being written to that field. Such measures dramatically reduce the chance of editing tools inadvertently corrupting the structures they edit.

- *High-level data properties*: The Monte translator language provides a mechanism whereby high-level "data properties" can be generated from the unpacked data. Viewing the data as a collection of properties rather than as a series of individual fields offers an even greater degree of abstraction to tool-writers.

Unicode/ISO 10646 Support

When decomposing a data structure, the Monte library converts all extracted text into a standard Unicode/ISO 10646 representation. All subsequent inspection, editing or other manipulation of the structure's text is performed through the medium of Unicode/ISO 10646-encoded text. When the data is finally re-packaged into its original format, Monte will automatically re-encode textual data into the data-structure's native format.

This use of a common intermediary text encoding is particularly useful when performing localisation between languages which use different character encodings, as shown in the example in figure 2 below:

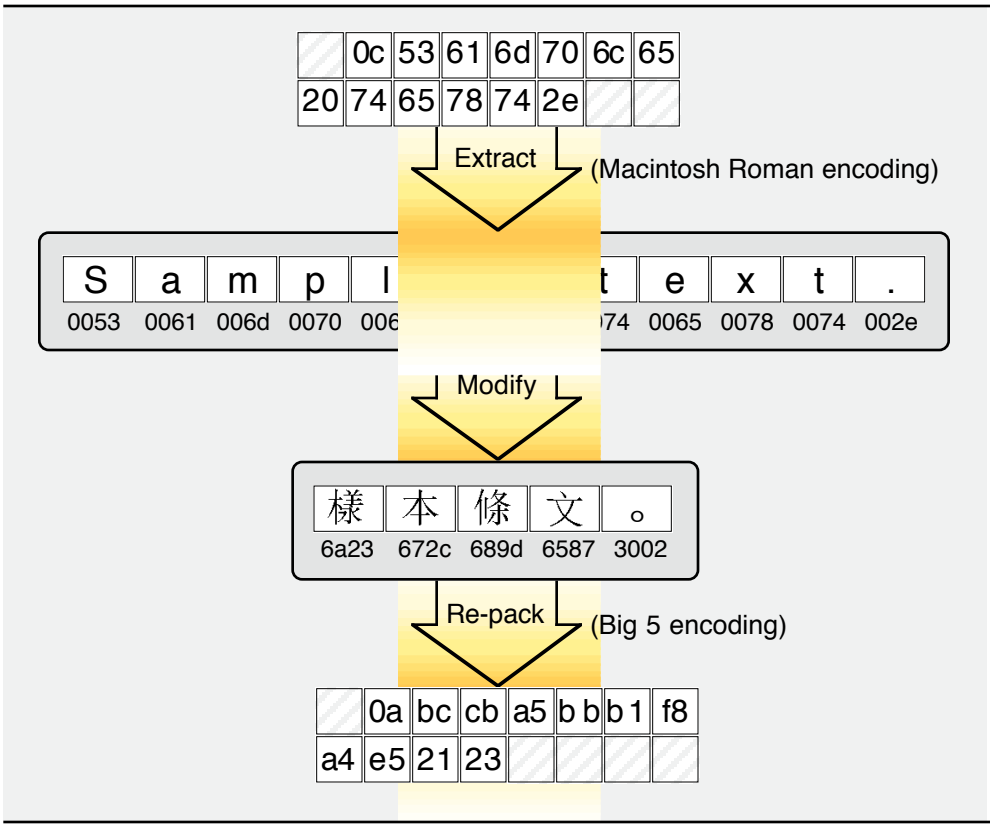

figure 1. editing text using Unicode/ISO 10646

(for the purpose of clarity, some steps have been omitted from the sequence shown in this diagram).

Data Translators

To get from a packed data block to an editable structure, Monte requires a description of the data to be decomposed. Traditionally on the Macintosh platform, ResEdit 'TMPL' ("template") resources have been used to describe data structures. However, when faced with the heterogenous container structures used to specify user interface elements in modern object-oriented frameworks, the 'TMPL' model has proved to be extremely cumbersome.

For this reason, Monte takes a different approach. To describe a binary data structure, Monte may use more than one *data translator*. A data translator is a textual description of all or part of particular data structure. For any given data format, Monte may use one or more of these data translators: where more than one translator is used, each describes a specific part of the entire data structure (for example, items in a list are described using a different data translator to that used for the entire structure).

As an example of the Monte translator language, figure 2 shows a simple data structure, and its equivalent Monte data translator:

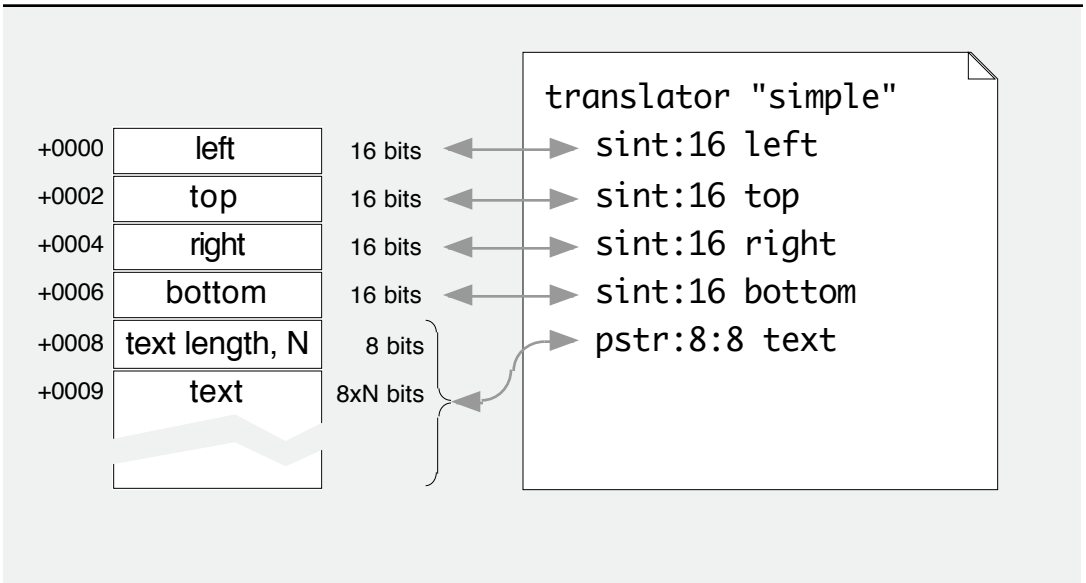

figure 2. a simple data structure and its data translator

The format of the translator language is described in more detail in Chapter 2 ("Writing Data Translators").

Object-oriented data description

In a departure from traditional data description mechanisms (eg., Rez, ResEdit templates), Monte takes a more fine-grained approach to the data being described. Instead of being a standalone definition of an entire data structure, a Monte data translator may describe only a part of the data to be processed. For all but the simplest data structures, several data translators are combined to describe the entire data structure. This approach reduces the amount of repetition required to describe closely related data types.

In addition, Monte allows translators to be specified as being variants of other, more basic, types. A variant translator may also have an identifying rule which Monte will use to choose between possible variants when translating packed data blocks, as illustrated in figure 3.

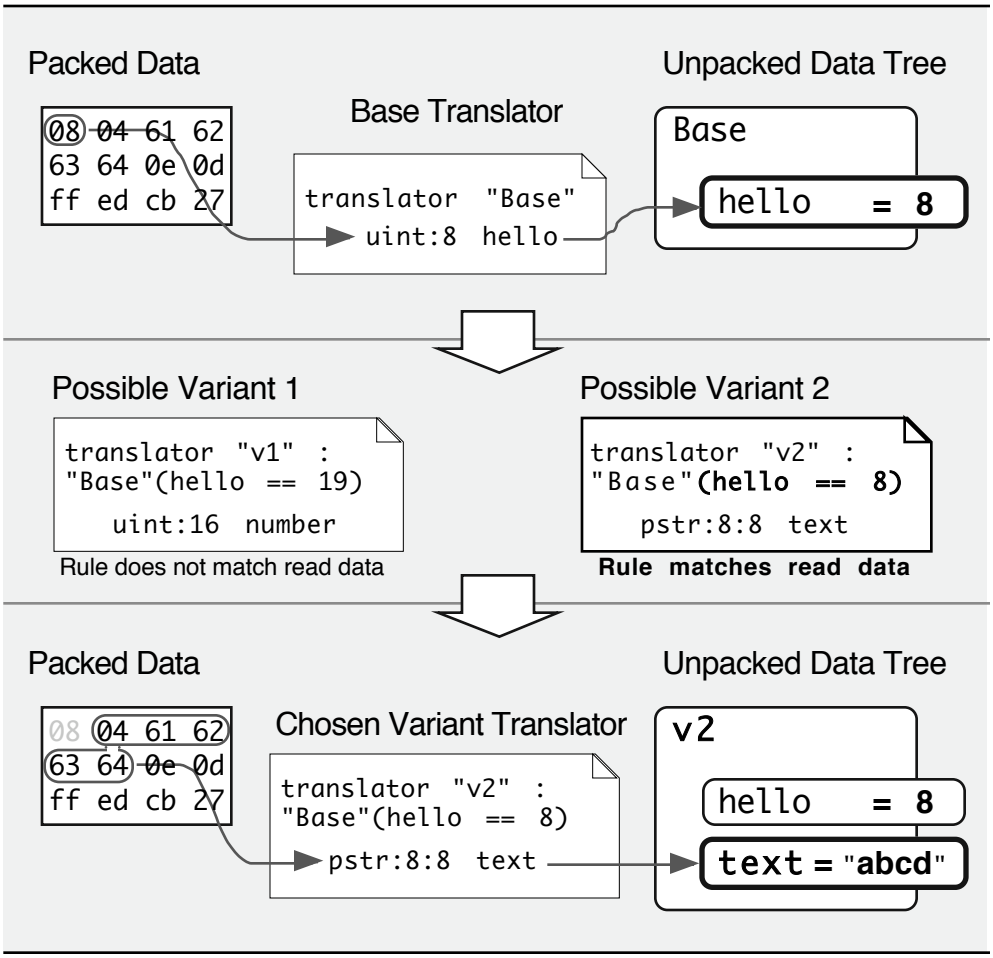

figure 3. data decomposition using translator variants

2. Writing Data Translators

This chapter provides an informal guide to the Monte data translator language. This language is formally described in Appendix B.

Introduction

A data translator is a text-based description of a data structure. Monte uses one or more data translators to convert a packed block of data into an editable, unpacked, data tree. This chapter provides information to allow you to write data translators.

File Encoding

Because Monte is a Unicode-based technology, its data-translators must also be Unicode compliant. This means that a "regular" 8-bit Mac-encoded text file cannot be used as a Monte data translator.

All translator files used by Monte must be Unicode or ISO 10646 UCS-2 compliant. This format is described in detail in Appendix B, but as a summary:

- Files must use the Unicode/ISO-10646 character encoding scheme.
- Files must begin with a Unicode byte-order-mark (character code U+FEFF)
- Files may be UTF-8, UTF-2BE (high-byte:low-byte) or UTF-2LE (low-byte:high-byte) format. UTF-2BE is preferred for PowerPC platforms.

There are several third-party text editors available which conform to these requirements.

Structure of Translator Files

A Monte data translator file has the following format:

- A header, containing version information, then
- Zero or more included files, followed by
- Zero or more translator definitions

File Headers

Before Monte can interpret a data translator file's contents, it must first know which version of the translator language the file is written in. Thus, all data translator files must begin with the 'monte' keyword followed by a version number, as shown below:

```
monte 1.0
```

Currently, there is only one version of the Monte translator language, but in future there may be others. By insisting on a language version declaration, newer versions of Monte will still be able to use older translators.

If the 'monte' keyword is omitted, Monte reports an error.

The following is the shortest valid translator file:

```
monte 1.0
```

Alternatively, comments may be added before or after the version declaration, as shown below:

```
/*
    veryShort.mtpl

*/
monte 1.0  // that's it!
```

Comments

From the example above, it can be seen that Monte allows comments to be included in data translator files. A comment can be a multi-line comment, delimited by the character sequences '/*' and '*/', as shown below:

```
/* This is a multi-line comment
and spans two lines */

/* another comment, that happens to be on only one line! */
```

Single-line comments can also be added to data translator files by using the character sequence '//'. All remaining text on the line is taken to be a comment:

```
uint:16 hello   // this is a comment!
// so is this
```

Note that, unlike languages such as C or C++, Monte does not permit comments to occur where spaces can occur. Comments in Monte translators must end with a new line. Thus, the following three comments are all illegal:

```
uint:16 /*can't replace spaces*/ hello

/* can't precede instructions */ uint:16

/* multi-line comments
are not allowed to end on the same
line as an instruction */ uint:16
```

Translator Worlds

Monte allows translators to be grouped into separate namespaces, known as translator worlds. This allows more understandable names to be used for the translators within a world.

To place translators within a particular world, use the 'world' directive:

```
world world-specification
```

The world specification is similar in structure to a UNIX file path: a series of worlds are specified, each separated by a forward-slash character. This allows users of Monte to ask for translators in a particular world, such as in "`mac/rsrc/MacApp/MyApp2.0`", or to specifiy a more general world, such as "`mac`", which will cause Monte to search in subordinate translator worlds for the requested translator.

World names are case sensitive: "`Macintosh`" and "`MacIntosh`" are two separate worlds.

Including Other Files

If a translator needs to use other translators in order to work, these may be included using the 'include' directive:

```
include "(file name here)"
```

The file specified by 'include' must reside in the same directory as the file including it. Monte does not permit path specifications of any sort in include directives.

Defining Translators

A data translator file can contain any number of data translators. Each of these is defined by using a 'translator' instruction, followed by any number of data instructions. The end of the translator definition is marked by the next 'translator' instruction, or the end of the file.

The syntax for the translator instruction is as follows:

```
translator "(translator name here)"
```

Export Translators

A Translator can be marked as an "export" translator by preceding its 'translator' instruction with the 'export' keyword, as shown below:

```
export translator "(translator name here)"
```

Applications which use the Monte library can discriminate between export and private (those without the 'export' tag) when presenting translator lists to their users, or when choosing translators themselves.
Generally, an export translator should refer to a complete data type, while any translators which are used by that export translator should be left private. So, for example, the 'DITL' translator would be an export translator, but 'CheckboxDITLItem' probably would not.

Variants

Many data structures in modern software make use of inheritance, whereby a new structure is defined as being an existing structure with some additional fields.To accomodate this, Monte allows translators to be specified as being variants of existing translators.

To specify that a translator is a variant translator, place a colon after the new translator's name, and follow it with the name of the translator from which you wish to derive, as shown below.

```
translator "(new translator's name)" : "(existing translator's name)"
```

For example, a structure "Extra", which is extends an existing structure "Basic", can be specified as follows:

```
translator  "Basic"
    sint:16  aElement


translator  "Extra"  :  "Basic"
    uint:32      extraElement
```

Note In this example, “Extra” has been defined as the **default variant** for “Basic”. This means that unpacking a structure with translator “Basic” will *always* result in translator “Super” being used as well. The reason for this is described below.

Variant Rules

When unpacking data structures, Monte uses a scheme called “greedy inheritance” to match the input data against the user-specified data translator. In effect, Monte will always try to use a variant of the translator it was given if one exists.

This is a difference from the way many high-level languages handle inheritance, but is more useful in decomposing packed data.

What this means for the example above is that whenever a user of Monte requests that a data block should be unpacked using the translator “Basic”, it will in fact be unpacked using “Basic”, then the translator “Extra”. This is because, when the translator “Extra” was defined, no rule governing when the translator should be used was specified.

A translator, like “Extra”, which contains no governing rule, is known as a *default variant*, as it is used by default whenever its base translator is used. A given translator is allowed to have only one default variant.

It is far more useful to specify variant translators which are used only sometimes, for example, when a field which has beeen read from the packed data block is found to have a certain, special value. As an example of this, the following should be familiar to many MacOS programmers:

```
        // General stub dialog item. Variants extend this as
        // appropriate.
    translator "DITLItem"
        fill:32
        sint:16 left/"left edge"
        sint:16 top/"Top edge"
        sint:16 right/"Right edge"
        sint:16 bottom/"Bottom edge"
        uint:1 enable/"Enabled? (1 for yes)"
        uint:7 type/"Item type"

        // Button
    translator "Button" : "DITLItem" ( {type}==4 )
        pstr:8:8 title/"Button title"
        align:16
```

The important thing to note in the above example is the rule specification ("`( {type}==4 )`") in the second translator, "Button". This states that, while "Button" is a variant of "DITLItem", it is a variant which should only be used when the value read for the field 'type' (the last field in the "DITLItem" translator) is 4.

A base translator, such as "DITLItem", may have any number of variant translators of this kind, provided that their rules are all different.

Although shown with a numeric field above, it is also possible to specify variant rules which refer to textual elements:

```
translator  "BaseText"
   pstr:8:8    hello

translator  "ExtraText"  :  "BaseText"(  {hello}="%")
   pstr:8:16   uniHello
```

In this example, if the 8-bit Pascal-style string 'hello', is found to read "%", then Monte will use the "ExtraText" translator to attempt to read a Pascal-style string with 16-bit characters from the packed data block.

(The pstr instruction, which represents Pascal-style strings, is described later)

Instructions

Instructions are the basic building blocks of Monte data translators. It is the instructions within a translator definition which actually describe the data to be unpacked.

In version 1.0, Monte provides nineteen instructions. These are:

`uint`	`sint`	`char`	`string`
`cstr`	`pstr`	`listcount`	`zerolistcount`
`sizeof`	`offset`	`location`	`align`
`fill`	`skip`	`insert`	`sub`
`list`	`repeat`	`prop`	

Instruction Labels

Each instruction in a data translator can be given a label. This label, which is an unquoted text string, is used in variant rule definitions (see previously) and property defintions. The syntax for specifying a label is as follows:

```
uint:8 (label goes here)
```

Labels must start with a letter. Digits and the characters '.' (U+002E, full stop/period) and '_' (U+005F, underscore) are also permitted in labels, but only as second or subsequent characters.

Note Because Monte data translators are Unicode/ISO-10646 based, "letter" here means any letter or character of any script system, excluding punctuation. Non-arabic-roman digits (ie., 0..9) are also permitted, although not strictly "letters". Be aware that Chinese/Japaese/Korean ideographs can—and probably will—be used in element labels.

Instruction Descriptions

Instruction labels are intended primarily for cross-referencing elements within a data translator. However, this means that instruction labels are largely unsuitable for presenting to an end user as a field description. To get around this problem, Monte allows a descriptive name to be given to an instruction's field. This is specified as follows:

```
uint:8 (label goes here)/"(descriptive text)"
```

An instruction must posess a label before it can be given a description. The description is intended as a field label for editor applications rather than reminders for the translator author. The descriptive text of an instruction is equivalent to the textual description field of a 'TMPL' resource item.

Numeric Data Instructions

Numeric Types

Monte supports signed and unsigned integer data types of up to 64 bits in length. To specify elements of this type, the 'sint' and 'uint' commands are used. 'sint' represents a signed integer value; 'uint', an unsigned value.

```
uint(:optional-data-size ) (label, etc.)
sint(:optional-data-size ) (label, etc.)
```

For example, consider the C-language structre 'myStruct' shown below:

```
struct  myStruct
    {
    short  w;
    unsigned  long  x;
    unsigned  short  y;
    };
```

This structure can be represented using the following data translator[1]

```
translator  "myStruct  translator"
   sint:16    w
   uint:32    x
   uint:16    y
```

Numeric data elements can be any size between 1 and 64 bits. This can be useful when dealing with structures in which several different fields are packed into a memory word or longword.

As an example, the MacOS (Carbon) Control Manager often specifies on-screen controls using a 'CDEF' (Control definition) number and variation code. These two values are packed into a 16-bit word in which the upper 12 bits hold the CDEF number and the lower four the variant code. In Monte, this arrangement can be expressed using the following instruction:

```
   uint:16        cdefAndVariation  // upper 12 bits are 'CDEF' ID
```

…however, it is more useful to split the physical 16-bit value into its component fields to allow each to be edited more easily, as shown below:

```
   uint:12        controlCDEFid
   uint:4         controlVariation
```

When writing data translators, you should try to break compound data elements (such as the 'cdefAndVariant') into their individual parts wherever possible.

Constant Numeric Values

Often, a data format will by definition contain some constant value elements. These may be placeholders, marker fields, "magic numbers", format version identifiers or any other type of fixed-value items. Monte allows these to be specified as follows:

```
translator  "everything  is  constant"
   sint:32    version = $100
   sint:32    minusOne = -1
   uint:17    answer = 42
   uint:15    filler = 0
```

[1] Assuming that the size of 'short' is 16 bits, and the size of 'long' is 32 bits, as is the case on virtually all Macintosh C compilers.

Note how hexadecimal constants can be specified using the $ symbol. Constant values may also be specified using packed characters, octal or binary as shown in the table below:

constant specified as…	*constant value is…*
$xxxxx… (x=0…9, a…f or A…F)	Hexadecimal
\xxxxx… (x=0…7)	Octal
%xxxxx… (x=0 or 1)	Binary
'xxxxx…**'** (x=any character)	Packed characters
xxxxx… (x=0…9)	Decimal

note Packed character constants are subject to a few rules. First, the data in a packed character constant is assumed to be made of 8-bit characters. In version 1.0 of Monte, only characters found in the MacOS Roman character set are permitted in packed character constants.

Second, although the MacOS Roman character encoding is used to calculate the value of the constant, the constant is specified using Unicode/ISO10646 encoding in the translator file. For example, the constant 'í' is stored in the translator file as its Unicode/ISO10646 character value of U+00ED, but the value of the constant is calculated from the MacOS Roman encoding's value for 'í', which is EA (hexadecimal).

If you use a Unicode-compliant editor, then all of these machinations will be completely hidden from you.

If, when reading data, Monte encounters a different value in a field which was marked as constant, an error message will be passed to the user. Using constant-value items thus allows Monte to detect badly-formed data, or even data of the wrong format.

Representing Text

String Instructions

Monte offers a range of instructions to describe text elements. The simplest of these is the 'string' instruction:

```
string(:optional-character-size) (label, etc.)
```

This represents an unbounded string of characters, as for example in the MacOS (Carbon) 'TEXT' resource type

```
translator  "TEXT"
   string     theText
```

The example above represents a string containing 8-bit-wide characters. However, Monte allows other character widths to be used: the size of the characters in a string can be specified by placing a size-specifier after the 'string' keyword, as shown below:

```
translator "packed ASCII text"
   string:7       theText
```

In the example above, each character is packed into seven bits. While this example is hardly commonplace, the same mechanism can be used to represent wide-character strings, which are far more useful:

```
translator "utxt"
   string:16 theText
```

For character-widths greater than 8 bits, Monte will assume that the Unicode/ISO10646 text encoding scheme is being used for character values.

For strings whose characters less than or equal to 8 bits in size, Monte will use a user-specified text encoding scheme to conver the smaller characters into Unicode/ISO10646: this text encoding is specified outside of the data translator to allow the same translator to be used for a data structure regardless of the encoding of its textual elements.

C-style Strings

C-style strings are made up of a finite sequence of characters, the last of which has the value 0. This zero-value character must always be present to terminate the string. To specify a C-style string in a translator, use the 'cstr' instruction:

cstr(:*optional-character-size) (label, etc.)*

The following translator shows the use of the 'cstr' instruction without a character-size field.

```
translator "standardCStringPair"
   cstr   hello
   cstr   goodbye
```

The example above represents two C-style strings containing 8-bit-wide characters. As for 'string', other character widths can used with C-style strings: the size of the characters in the string can be specified by placing a size-specifier after the cstr keyword, as shown below:

```
translator "packed ASCII Greetings"
   cstr:7     hello
```

Note that because the terminator value is just another character in a C-style string, the terminator will be the same length as the other characters in the strings (in this case, seven bits).

Naturally, wider characters are also possible: the example below shows how 16-bit Unicode/ISO10646-encoded C-style strings may be represented in a Monte translator.

```
translator  "wideCStringPair"
   cstr:16   hello
   cstr:16   goodbye
```

Note Because its terminator is actually character, any C-style string with 16-bit characters will have a 16 bit wide zero-value as a terminator.

Pascal-style Strings

Pascal-style strings are made up of a string-length field, followed by a finite sequence of characters. The number of characters in the string is specified by the value in the string-length field. Monte allows both the size of this length field and that of the characters in the string to be specified independently of each other.

To represent Pascal-style strings, Monte provides the 'pstr' keyword. The syntax for Pascal-style strings is:

```
pstr(:optional-length-field-size:character-size) (label, etc.)
```

The translator below shows the use of the 'pstr' instruction without specifying character or length-field sizes:

```
translator  "standardPascalStringPair"
   pstr   hello
   pstr   goodbye
```

The example above represents two Pascal-style strings containing 8-bit-wide characters. The length counter value is also eight bits. Pascal-style strings of this type are by far the most common way of representing text on the MacOS (Classic) platform.

The limitation of such strings is that because an 8-bit value is being used to hold the character count, the string is limited in length to 255 characters (255 being the largest number which can be represented using 8 bits). This limitation is often overcome by applications using a larger length field — sixteen or thirty-two bits, for example. Luckily, Monte can also understand these kinds of string:

```
translator  "wordPascalString"  // pstr with 16-bit length field
   pstr:16:8  hello

translator  "longPascalString"  // pstr with 32-bit length field
   pstr:32:8  hello
```

Although the length fields of the strings represented above are 16 and 32 bits, respectively, both strings still use 8-bit characters. Monte can, however, represent strings with any size of character up to 16 bits:

```
translator  "uPascalString" // pstr with 16-bit characters
   pstr:32:16 hello
```

Fixed-length Strings

Often, data structures will contain as string field which is of fixed-length, regardless of the length of the field's text. To represent these, Monte allows any string instruction to be used in a fixed-length form. The syntax for this is:

string-instruction(:*optional-character-size)*[*character-count*] *(label, etc.)*

... where *character-count* is the fixed length of the string. Examples of fixed-length strings are shown below

```
translator  "SomeStrings"
   string:8[32]  thirtyTwoBytes
   pstr:8:8[27]  twentyEightBytes
```

Note that for Pascal-style strings the length-counter field is not included in the string's character count. This is for the simple reason that Monte allows the length field of a Pascal-style string to be of a different size than that of the string's characters.

String lengths are specified in encoded characters. Thus, if the characters in the string are larger than eight bits, the space occupied by the string will be correspondingly larger, as the examples below show:

```
translator  "moreStrings"
   cstr:16[10]  twentyBytes
   cstr:8[10]  tenBytes
```

Also, because the length of a fixed-length string is specified in encoded characters, which Monte converts to Unicode/ISO10646 internally, there will often be no direct correspondance between the size of the string in the original data, and its length in characters when editing it with Monte. Monte will, however, ensure that no changes are made to a fixed-length string which would cause its encoded text to exceed the maximum length specified in the data translator.

Terminating Strings

Sometimes a structure contains variable-length strings without "normal" terminators or length counts. Fortunately, Monte can deal with these: if a string instruction is followed by a constant-value integer instruction, then Monte will terminate the string when that constant-value is found in the input data.

For example, for a string ending with the 16-bit value "-1":

```
translator  "terminatedString"
   string:8   text
   sint:16    terminator = -1
```

The terminator may also be the first instruction in a following substructure, if that instruction is a constant numeric instruction. This is shown below:

```
translator  "substructure"
   sint:16    terminator = -1

translator  "terminatedString  II"
   string:8   text
   sub("substructure")
```

The 'sub' instruction, which is described fully later, is used to insert one translator's data elements into another.

Constant Strings

Just as constant numeric instructions can be specified, so can constant strings. The syntax for specifying constant value strings is:

(string instruction) = *constant-value*

The constant value must be a quoted string, as shown below:

```
translator  "constStringStructure"
   sint:4     a
   sint:4     b
   pstr:8:8   hello="Hi  There!"
   string:8   goodbye/"text  used  to  say  goodbye"="Slán!"
```

Any type of string instruction can be given a constant value in this way.

Note that the text of the constant value comes *after* the instruction's label or description text.

Structures

Monte provides two instructions for building new data translators from existing translators. The 'sub' instruction is used to create sub-containers in the decomposed data tree, whereas the 'insert' instruction provides a simple "macro-expansion" mechanism for Monte data translators.

Insertion

To insert the contents of one translator into another, use the 'insert' instruction:

```
insert( "name-of-translator" )
```

The *name-of-translator* parameter, a quoted text string, should be the name of the translator whose elements you wish to insert into the current translator.

Note It is not possible to assign a label or description to 'insert' instructions, simply because the 'insert' instruction does not itself create an entry in the unpacked data tree.

The example below shows how the insert instruction is used.

```
translator  "rectangle"
   sint:16    left
   sint:16    top
   sint:16    right
   sint:16    bottom

translator  "button"
   insert("rectangle")
   pstr:16:16 text

translator  "icon"
   insert:("rectangle")
   sint:16    iconID
```

In this example, the elements from the 'rectangle' translator are inserted into both the 'button' and 'icon' translators. This way, additional rectangle-related information (such as properties, see later) may be specified once, in the 'rectangle' translator, without need for repetition in all translators which use rectangles.

Insertion is invaluable when describing multiply-inherited data structures: this topic is dealt with in more detail in the section "Translator Variants" below.

Encapsulation

To describe substructures within the packed data tree, as opposed to just using a preset sequence of items, Monte provides the 'sub' instruction:

```
sub( "name-of-translator" ) label, description, etc.
```

The 'sub' instruction creates a container in the output decomposed data tree, and fills that container with items translated by the specified sub-translator.

As for 'insert', previously, *name-of-translator* is the translator which will describe the substructure. Unlike 'insert', however, it is possible to assign a label and description to the results of the 'sub' instruction.

Encapsulation vs. Insertion

The main difference between the 'sub' and 'insert' instructions is that the 'sub' instruction will create a sub-container for its results in the unpacked data tree, whereas the 'insert' instruction does not. This behaviour is shown in figure 5. overleaf:

Figure 5a. shows the result of using a 'sub' instruction to include the 'rectange' translator. In the diagram of the packed data tree (Figure 5a., right), the elements from the "rectangle" translator have been placed within a separate container, called "bounds" (the label of the 'sub' instruction).

In contrast, figure 5b. shows the action of the insert instruction on the same translator. Note that the insert instruction creates no elements in the unpacked data tree, but rather causes the elements fromthe "rectangle" translator to be placed directly into the unpacked data tree. From figure 5b. it should also be obvious why the 'insert' instruction cannot be given a label: there is nothing to label.

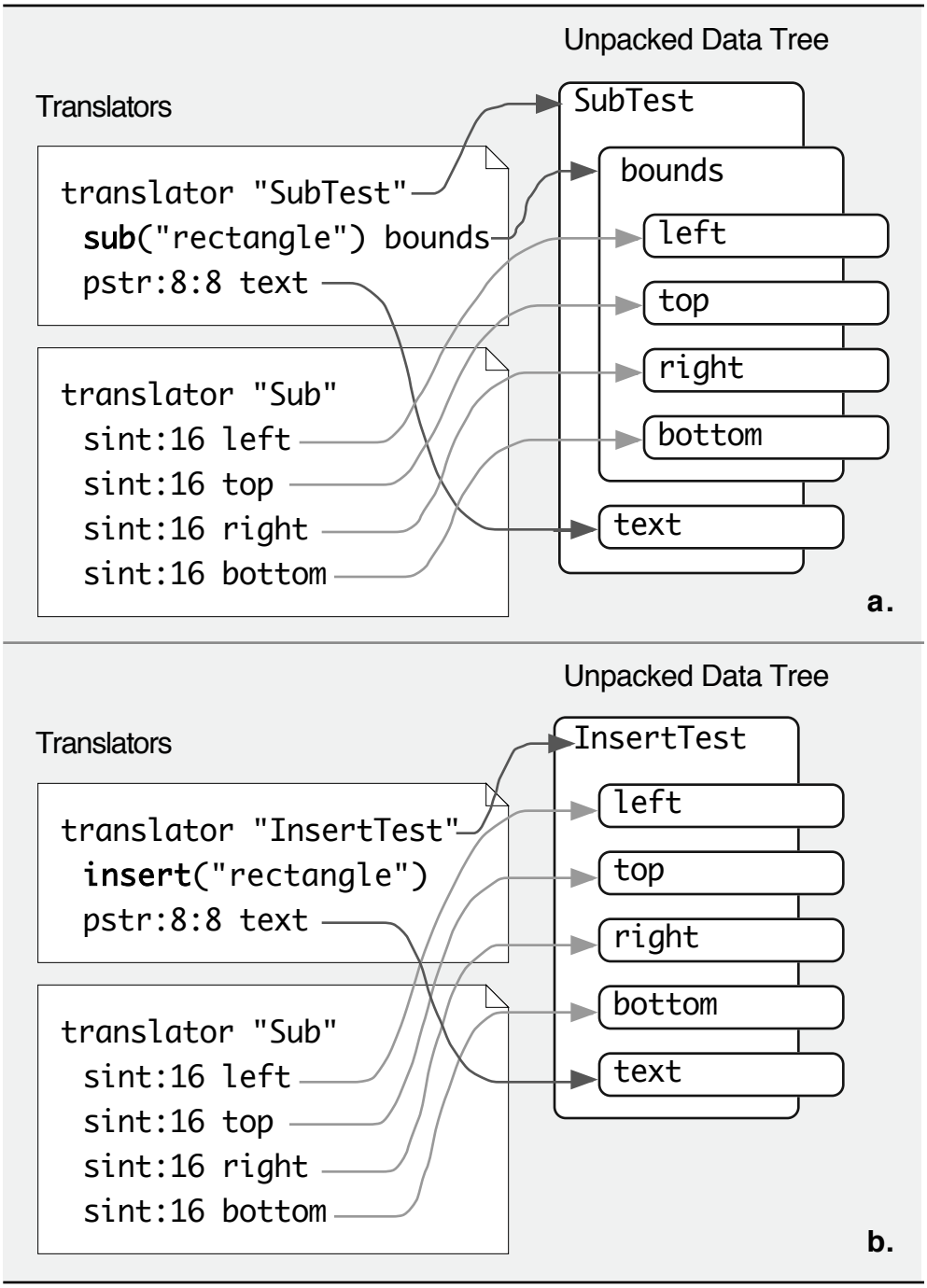

figure 5. behaviour of sub and insert instructions

Lists

To describe lists of items, Monte provides the 'list' and 'repeat' keywords. The former describes a list with any number of items, which may or may not be counted; the latter describes lists in which the number of items is fixed.

Variable-length Lists

By far the more common type of list is the list with a variable number of items. This is represented by the 'list' instruction:

> `list(` *name-of-translator* `)` *label, description, etc.*

The 'list' instruction creates a container—the "list container"— then fills it with further containers, each of which is described by the specified translator.

The parameter *name-of-translator* is the name of the translator which will describe the items of the list. Note that if this translator has variants, then the list can contain items of different types (ie., a heterogenous list): the variants of the selected translator will describe each possible list type.

The default behaviour of the 'list' instruction is to keep reading as many items as possible until the input data runs out. This behaviour can be changed, however, by the use of list counters (see below).

List Counters

To allow more control over lists of items, Monte implements two kinds of list counter, the 'zerolistcount' and 'listcount' instructions.

> `zerolistcount:`*size*`(` *list-label* `)` *label, description, etc.*
> `listcount:`*size*`(` *list-label* `)` *label, description, etc.*

The *size* parameter is identical to that of the 'uint' and 'sint' instructions and describes the width of the list counter field, in bits. The *list-label* parameter is the label of a subsequent 'list' instruction, and indicates which list's items are being counted by this list count instruction.

The 'listcount' instruction counts the items of the list from zero (an empty list) to *n* (however many items).

Several existing data structures (most notably the MacOS Classic 'DITL' resource), adopt a different numbering for counters, in which the value zero means the list has *one* item. To accomodate these, Monte provides the 'zerolistcount' instruction.

Fixed-length Lists

In some cases, a data structure may contain a fixed number of objects. Such structures can be described using the 'repeat' instruction:

> `repeat[`*number-of-items*`](` *name-of-translator* `)` *label, description, etc.*

As its name implies, the 'repeat' instruction creates a list container, then fills it with a fixed number of whose contents are described by the specified translator. The *number-of-items* parameter is used to specify how many items should be present in the list.

As for the 'list' instruction, *name-of-translator* specifies the translator to be used to describe the list's items. Note that even though the *number* of items in the list is fixed, there is no reason why the *type* of these items must be the same: if the specified translator has variants, then these variants may be used to describe the list's items.

Data Alignment and Structuring

Sometimes, data structures contain data which is necessary to preserve the data structure itself, rather than convey any information. Items must sometimes be aligned to a specific multiple of bytes, structures may contain empty fields, or the location or size of specific elements within the structure needs to be recorded. Fortunately, Monte can cater for all of these situations.

Aligning Structures

To align following data to a specific bit-multiple, Monte provides the 'align' instruction:

> `align:`*word-size*`(:`*bit-offset*`)` *label, description, etc.*

When reading data, the 'align' command causes a number of bits of data to be ignored such that the next element read is aligned to a multiple of *word-size* bits. If a data structure needs to be aligned to a specific bit within this larger word size, the *bit-offset* parameter can be used to specify which bit (0, the default value, means the first bit in the word).

As an example, to align to an "odd-byte", use the following:

```
align:16:8
```

Locations

To record the position of another element in the data structure, use the 'location' instruction:

> `location:`*size*`(` *target-label* `)` *label, description, etc.*

The *target-label* parameter identifies the element whose location is to be recorded (the "target element"). The location of this element will be expressed as a number of *bytes* (not bits!) from the beginning of the packed data block.

When the packed data is being read, the value corresponding to a 'location' instruction is ignored by the current version of Monte. When data is to be re-packed, the correct location value will be generated and written to the output.

Note If a target element spans more than one byte (ie., a misaligned field), the location instruction records the location of the first byte containing a any of the data.

Offsets

Wheras the 'location' instruction specifies the absolute position of an element, it is also possible to represent the relative position of an item, using the 'offset' instruction:

```
offset:size( target-label ) label, description, etc.
```

Unlike 'location' (above) which calculates the position of its target from the beginning of the packed data block, 'offset' records the number of bytes between the first byte of the 'offset' instruction's data and the first byte of the target element's data (the target element being that element whose label is *target-label*).

When the packed data is being read, the value corresponding to an 'offset' instruction is ignored by the current version of Monte. When data is to be re-packed, however, the correct offset value will be generated and written to the output.

Note The size of the 'offset' field itself is also included when the offset is calculated.

For example, the following shows a use of the 'offset' instruction:

```
translator  "jumpItem"
   offset:32( itemEnd ) offsetToResID   // value will be 8
   uint:32  type
   sint:16    resID
```

Sizeof

To determine the size, in bytes, of an element (which may also be a substructure or list), the 'sizeof' instruction is used:

```
sizeof:size( target-label ) label, description, etc.
sizeof:size  label, description, etc.
```

If present, the parameter inside the brackets is the label of the element whose size is to be measured. This can be any element at the same level in the data tree as the 'sizeof' command. If no target element is specified, then the element to be measured is the container in which the 'sizeof' instruction itself occurs.

'sizeof' instructions can be used to explicitly size unterminated strings (see previouslty) or 'skip' fields (see below). If the string or skip field is the subject of a 'sizeof' instruction, then the value read for that 'sizeof' will be used to calculate the length of the string, or the amount of data to skip. This holds true even if the 'sizeof' field refers not to the skip or string itself, but to a container which holds the skip or string. The following example illustrates both cases:

```
translator  "textItem"
   uint:16  typeSize
   string:8   theText      // unterminated, but see below.
```

```
translator  "sizeofDemo"
sizeof:16(theItem)          // terminates string in container
sub("textItem")  theItem
sizeof:16(alternateText)    // terminates string at this level
string:8   alternateText
```

Incidentally, the last two lines of the "sizeofDemo" translator are equivalent to:

```
pstr:16:8  alternateText
```

Unused Fields

The 'fill' instruction is used to represent an empty data field:

```
fill:size    label, description, etc.
```

Data corresponding to a 'fill' field is ignored when reading the packed data block, and written as a series of zero-bits when recreating the packed data.

Skipping Data

The 'skip' instruction is used to ignore data in the packed data block:

```
skip(:size )    label, description, etc.
```

Unlike 'fill' (above), the 'skip' instruction remembers the data which was read, and writes it back verbatim when the data block is being recreated.

If the size field of the 'skip' instruction is omitted, the remainder of the packed data block is skipped, unless the 'skip' instruction (or its parent container) is the target of a preceding 'sizeof' instruction, in which case the amount of data to be skipped is calculated based on the value read for the 'sizeof' field.

Properties

In order to facilitate a higher-level representation of the input data than that afforded by the simple decomposition of each data field, Monte provides a set of standard *data properties* to which values may be assigned by the data tranlsator. This allows the decomposed data to be viewed at a higher level, as a set of standard properties, rather than as a complex list of data fields.

Property values are assigned by the translator, using the 'prop' instruction:

```
prop  (label) =  target-label
prop  (label) =  "(property  definition)"
```

For example, the translator for 'DITL'-resource buttons may define the "LocaleText" (Localisable or user-visible text) property of a given Button sub-object to be the same as the "title" field of that decomposed Button object, as shown below:

```
translator "Button" : "DITLItem" ( {type}==6 )
   pstr title
   align:16
   prop LocaleText = title
   prop Name = "Button {Index1} ""{title}"""
```

…or for a string list, where the element representing the localisable text is different:

```
translator "StringEntry"
   pstr text
   prop LocaleText = text
```

This allows user of the Monte API to change the localisable text for a more than one type of structure consistently by simply asking to change the "LocaleText" property of each object. However, to be modified in this way, a property must correspond to the value (text or numeric) of *one* data field only: it is not possible to define modifiable properties using expressions involving more than one data field.

Non-Modifiable Properties

Certain properties of a decomposed data structure correspond directly to data fields in the decomposed data structure (for instance "LocaleText"). Because of this direct correspondence, such properties may be modified by callers of the Monte API just as the data fields themselves can. Properties like this are called Modifiable Properties.

However, there is another kind of property description, such as "Name", which is usually built up from one or more other properties or data elements. For example:

```
translator "Button" : "DITLItem" ( {type}==6 )
   pstr title
   align:16
   prop LocaleText = title
   prop Name = "Button {Index1} ""{title}"""
```

In this case, changing the value of the property could have no predictable effect on the underlying data, as it is in many cases impossible to determine which underlying data elements must change—or by how much— to generate the new value for the property.

Because of this difficulty, Monte considers properties which refer to more than one data-element to be non-modifiable: once defined, a non-modifiable property cannot be changed in the way a data-element can. However, a non-modifiable property will remain coherent with the state of its constituent data elements: a change in a data-element used by a non-modifiable property will be reflected in the property (changing the value of "Title" or "LocaleText" will cause the text of the Button's "Name" property to change).

Non-modifiable properties are still useful for tools which do not need to modify structured data, and can also provide additional information to those that do.

Element Insertion in Properties

If a non-modifiable property is created, elements are specified in the quoted string as shown below:

```
prop MyProperty = "One: {element1}. Two: {element2}"
```

Element names in braces (curly brackets) will be replaced by the value of the specified elements once data has been decomposed. To specify braces themselves, the brace characters can be doubled up:

```
prop CoOrdinates = "{{({x1},{y1}),({x2},{y2})}}"
   // would read as "{(10,10),(13,20)}"
```

Property names may be used interchangeably with element names in property defintions:

```
pstr:8:8 string
prop Name = "Item {Index1} : {string}"
       // would read as "Item 1 : Hello"
```

Note Because element and property names are interchangeable in this manner, it is not possible to use the same label for a property and another instruction.

It is also possible to perform self-references in order to extend a property string:

```
prop Counties = "Limerick, Waterford, Cork, Tipperary"
…
prop Counties = "{Counties}, Kerry, Clare"
```

A self-reference to a property which has not been previously defined is not an error: an empty string is used to represent the undefined property in such cases.

Note that property defintions are hierarchical – a sub-tree inherits the property definitions of its parent. Thus if one container had the property "`Path`" defined as "`abc/def/g`", then in a subcontainer of that container, the definition:

```
prop Path = "{Path}/z"
```

…would redefine `Path` as "`abc/def/g/z`" for that sub-container. Although shown using a self-reference here, this rule applies even when the property definiton is not self-referential.

Implied Properties

Certain properties become assigned to objects simply because the objects are in a particular situation. For instance, all items of a list are given an "Index" property which records the object's position within the list. In fact, two Index properties are generated: one, called Index0, starts counting items from zero; the other – Index1 – starts counting items from one. Thus for the first item, 'Index1' is 1 and 'Index0' is 0; for the second, 'Index1'=2 and 'Index0'=1, and so on.

Properties such as Index1 in this example are called *implied* properties, because they do not need to be explicitly set in a translator.

3. Sample Code

Strings

For compatibility, Monte uses MacOS memory handles to pass Unicode/ISO10646-encoded text strings into and out of the API.

The following functions show how to convert these handles into C++ Standard Template Library wide-strings (std::wstring).

First, to convert the text of a std::wstring object into an already-allocated handle:

```
void StringToHandle( const std::wstring& source, Handle dest )
    {
    long size = static_cast<long>(source.size()) << 1;

    SetHandleSize( dest, static_cast<long>(size) );

    if (size>0 && GetHandleSize( dest ) == size)
        {
        HLock( dest );
        BlockMove( source.data( ), *dest, size );
        HUnlock( dest );
        }
    }
```

Second, to convert the contents of a handle back to a std::wstring object:

```
void HandleToString( Handle source, std::wstring& dest )
    {
    unsigned long characters
        = static_cast<unsigned long>(GetHandleSize( source )) >> 1;

    if ( characters>0)
        {
        HLock( source );
        dest.assign( reinterpret_cast<wchar_t*>(*source),
                     characters );
        HUnlock( source );
        }
    else
        dest.clear( );
    }
```

ReportingErrors

Error Codes

Error codes (type Error) are used by many of the Monte APIs. The following code snippet shows how to interpret the fields of the Error structure.

```
void DumpErrorCode( Error error )
    {
        // only report if error is present
    if ( error.l != 0L )
        {
        gConsoleOutput << std::dec;

            // report error severity
        switch ( error.e.severity )
            {
            case kInformationOnly:
                gConsoleOutput << "{ Information: ";
                break;

            case kWarning:
                gConsoleOutput << "{ Warning: ";
                break;

            case kErrorSuggestTerminate:
                gConsoleOutput << "{ Fatal Error: ";
                break;

            case kErrorSuggestAbort:
                gConsoleOutput << "{ Serious Error: ";
                break;

            case kError:
                gConsoleOutput << "{ Error: ";
                break;
            }
            // dump OS error code and private (Monte) code
        std::cout << "(OS Code " << error.e.osError
                <<") Error Code " << error.e.privateCode << " }";
        }
    }
```

Error Logs

The following code produces a textual error dump from the contents of a Monte ErrorLog:

```
void DumpErrors( ErrorLogRef errors )
   {
       // Only report if an error exists in log
   if ( ErrorOccurred( errors ) )
       {
       short count, i;
       count = CountErrors( errors );
       Handle file, lineText;
       file = NewHandle( 0 );
       lineText = NewHandle( 0 );
       unsigned long lineNumber;
       unsigned long errorPosition;
       std::wstring line, filename;

       for (i= 0; i< count; i++)
           {
           std::cout << std::endl << std::dec;

               // report the error severity
           switch (GetIndErrorSeverity( errors, i) )
               {
               case kInformationOnly:
                    std::cout << "{ Information: ";
                    break;

               case kWarning:
                    std::cout << "{ Warning: ";
                    break;

               case kErrorSuggestTerminate:
                    std::cout << "{ Fatal Error: ";
                    break;

               case kErrorSuggestAbort:
                    std::cout << "{ Serious Error: ";
                    break;
```

```
    case kError:
        std::cout << "{ Error: ";
        break;
    }
    // report OS error code and private (Monte) code
std::cout <<"(OS Code "<<GetIndErrorCode( errors, i )
        <<") Error Code "
        <<GetIndPrivateErrorCode( errors, i )
        <<" }";

    // Check for a parser error - these have more
    // information than normal errors
if (IsIndErrorParserError( errors, i ) )
    {
    // extract filename, line and other information
    // for parser error
    GetIndParserErrorInfo( errors, i, file,
            &lineNumber, lineText, &errorPosition );
    HandleToString( lineText, line );
    HandleToString( file, filename );

        // highlight error position on line
    if (line.size()>0)
        {
        if (errorPosition< line.size()-1 )
            line.insert( errorPosition+1,
                 std::wstring("<-- ") );
        else
            line.append( std::wstring("<-- ") );

        line.insert( errorPosition, String(" -->") );
        }

    std::cout << std::endl << "      " << filename
            << " line "<< std::dec << lineNumber
            <<std::endl << "      “"<<line<<"”";
    }
}

// clear supplied error log
```

```
    FlushErrorLog( errors );

        DisposeHandle( file );
        DisposeHandle( lineText );
        }
```

Decomposing Binary Data

```
void ExtractResource( const std::wstring& translatorName,
        unsigned long resType, short resID   )
    {
    Handle tworld = NewHandle( 0 );
    Handle tname = NewHandle( 0 );

        // look for translator in “mac/rsrc” world
    StringToHandle( translatorName, tname );
    StringToHandle( std::wstring("mac/rsrc"), tworld );

        // Get the resource to be unpacked
    Handle res = Get1Resource( resType, resID );

        // only continue if resource was found
    if ( res != 0L )
        {
        TranslatorRef theTranslator = 0L;
        NodeRef tree;
        Error error;

        HLock( res );

        GetTranslator( tworld, tname, eAnyTranslator,
                &theTranslator, &error);

            // dump information about call
        std::cout << std::endl << "GetTranslator: ";
        DumpErrorCode( error );

            // create error log object to hold decomposition errors
        ErrorLogRef errors = NewErrorLog( );
```

```
        // Unpack the data
    DecomposeDataBlock( theTranslator, *res,
            GetHandleSize( res ), kTextEncodingMacRoman, &tree,
            errors );

        // dump error information
    std::cout << std::endl << "DecomposeDataBlock: ";
    DumpErrors( errors );

        // don't need error log anymore
    DisposeErrorLog( errors );

        // Dump contents of data tree (see later)
    DumpDataTree( tree, 1);

        // Dispose of data tree
    DisposeDataTree( tree, &error );
    std::cout << std::endl << "DisposeDataTree: ";
            DumpErrorCode( error );

        // Release data translator
    ReleaseTranslator( &theTranslator, &error );

    std::cout << std::endl << "ReleaseTranslator: ";
    DumpErrorCode( error );

    HUnlock( res );
    ReleaseResource( res );
    }

DisposeHandle( tname );
DisposeHandle( tworld );
}
```

Data Trees

The following examples show how to navigate and inspect a decomposed data tree, as produced by the 'DecomposeDataBlock' API call.

Node Information

The following function, 'DumpNodeInfo' produces a textual dump of the contents of a node of the decomposed data tree:

```
void DumpNodeInfo( NodeRef theNode, short indentLevel )
    {
    Error error;
    ENodeType type;
    ENodeDataType dType;
    Handle text = NewHandle( 0 );
    std::wstring conv;
    Numeric num;
    unsigned long val;
    Boolean sign, con;

        // Cosmetic stuff - ensure that the output is indented
        //      correctly. Indentation is used by "DumpDataTree"
        //      function (see later)
    if (indentLevel<1)
        indentLevel=1;
    String indent("\n");
    indent.append( String("\r") );
    indent.append( indentLevel << 2, Character(' ') );
    String shortIndent("\n");
    shortIndent.append( String("\r") );
    shortIndent.append( (indentLevel << 2) -3, Character(' ') );

        // Output the node's label
    std::cout << shortIndent << std::dec << " • ";
    GetNodeLabel( theNode, text, &error );
    DumpErrorCode( error );
    HandleToString( text, conv );
    std::cout << ""<< conv <<"";

        // output the node's descriptive text
    std::cout << " (description=";
    GetNodeDescription( theNode, text, &error );
```

```
DumpErrorCode( error );
HandleToString( text, conv );
std::cout << ""<< conv <<"")";

    // Find and report the kind of node this is
GetNodeType( theNode, &type, &error );
DumpErrorCode( error );
switch ( type )
    {
    case eEmptyNode:
        std::cout << "[empty, ";
        break;

    case eDataNode:
        std::cout << "[data, ";
        break;

    case eSubcontainerNode:
        std::cout << "[subcontainer, ";
        break;

    case eContainerNode:
        std::cout << "[container, ";
        break;

    case eListItemNode:
        std::cout << "[item, ";
        break;

    case eConstValueNode:
        std::cout << "[const, ";
        break;

    case eAutomaticNode:
        std::cout << "[generated value, ";
        break;

    case eListNode:
        std::cout << "[list, ";
        break;
```

```
    case  ePropertyNode:
        std::cout  <<  "[property,  ";
        break;

    default:
        std::cout  <<  "[**unknown  type**,  ";
        break;
    }

GetNodeDataType(  theNode,  &dType,  &error  );
DumpErrorCode(  error  );

if  (dType&eReadOnlyData)
    {
    std::cout  <<  "read-only  ";
    }

switch  (dType&(~eReadOnlyData))
    {
    case  eNoData:
        std::cout  <<  "no  data]";
        break;

    case  eTextData:
        std::cout  <<  "textual  data]";
        break;

    case  eNumericData:
        IsNodeSigned(  theNode,  &sign,  &error  );
        DumpErrorCode(  error  );
        if  (sign==true)
            std::cout  <<  "signed  ";
        else
            std::cout  <<  "unsigned  ";
        std::cout  <<  "numeric  data]";
        break;
```

```
    default:
        std::cout << "**unknown data**]";
        break;
    }

    // TEXTUAL NODES
    //  Get node's text, display it.
    //  Report size of text, as Unicode chars and encoded words
    //   If node's text is of limited size, report that size
if ( dType == eTextData || dType==eReadOnlyTextData )
    {
        // Get text of node
    std::cout << ":" << indent << "Text value=";
    GetNodeTextualData( theNode, text, &error);
    if (error.l != 0L )
        {
        DumpErrorCode( error );
        }
    else
        {
        HandleToString( text, conv );
        std::cout << "“" << conv <<"”";
        }

        // Get size of text in unicode characters
    std::cout << indent << "Text size " << std::dec;
    GetNodeTextCharacterCount( theNode, &val, &error);
    if (error.l != 0L )
        {
        DumpErrorCode( error );
        }
    else
        {
        std::cout << val <<" unicode chars ";
        }

        //  Get size of text in encoded words - may
        //  differ substantially from “character count” above
    std::cout << "(=";
    GetNodeTextEncodedCharacterCount( theNode, &val,
```

```
        &error);
if (error.l != 0L )
    {
    DumpErrorCode( error );
    }
else
    {
    std::cout << val <<" encoded words)";
    }

    //  Determine whether or not node's text is
    //  constrained by some upper size limit
std::cout << indent << "Limited text size? ";
IsNodeTextConstrained( theNode, &con, &error);
if (error.l != 0L )
    {
    DumpErrorCode( error );
    }
else
    {
    std::cout << (con? "yes" : "no");
    }

    // Get character limit and display it
    // (even "non-constrained" text nodes have a text
    // size limit)
std::cout << " text size limit=";
GetNodeTextMaxEncodedCharacterCount( theNode, &val,
        &error);
if (error.l != 0L )
    {
    DumpErrorCode( error );
    }
else
    {
    std::cout << val << " encoded word(s)";
    }
}
```

```
    //  NUMERIC  NODES
    //      Get numeric value, then ask node for
    //      this value presented as text.
if ( dType == eNumericData   )
        {
            // Get node value, display it as hex
        std::cout << indent << "Numeric value = ";
        GetNodeNumericData( theNode, &num, &error );
        if (error.l != 0L )
            {
            DumpErrorCode( error );
            }
        else
            {
            std::cout << std::hex << num.unsignedValue << std::dec;

                // Get node's value as text, by asking the node
                // (format may be binary, hex, octal or decimal)
                // note that this is the same call as used for
                // textual nodes above
            std::cout << " ( presented as text = ";
            GetNodeTextualData( theNode, text, &error);
            if (error.l != 0L )
                {
                DumpErrorCode( error );
                }
            else
                {
                HandleToString( text, conv );
                std::cout << """ << conv <<""" )";
                }
            }
        }

        // Has node any children? (specifically for containers,
        // but all nodes will respond correctly to the request)
    GetNodeChildCount( theNode, &val, &error );
    DumpErrorCode( error );
    if ( val>0)
        {
```

```
        gConsoleOutput << indent << " -> " << val << " children";
        }

    DisposeHandle( text );
    }
```

Properties

Property information is attached to container nodes. The following function shows how this information can be extracted and displayed:

```
void DumpPropertyInfo( NodeRef container, short indentLevel )
    {
    Error error;
    Handle text = NewHandle( 0 );
    PropertyRef prop;
    std::wstring conv;
    Boolean mod;

        // cosmetic indentation
    if (indentLevel<1)
        indentLevel=1;
    String indent("\n");
    indent.append( String("\r") );
    indent.append( indentLevel << 2, Character(' ') );


        // Find first property, then iterate through properties
        // until none are left
    GetFirstProperty( container, &prop, &error );
    DumpErrorCode( error );
    while ( (prop.l1|prop.l2) != 0L )
        {
        std::cout << indent << "property ";
        GetPropertyName( prop, text, &error );
        DumpErrorCode( error );
        HandleToString( text, conv );
        std::cout << conv ;
```

```
            // report whether property is modifiable or not
        IsPropertyModifiable( prop, &mod, &error );
        DumpErrorCode( error );
        if (mod)
            {
            std::cout << " (modifiable) ";
            }
        else
            {
            std::cout << " (non-modifiable) ";
            }

            // get and display property's value
        GetPropertyValue( prop, text, &error );
        DumpErrorCode( error );
        HandleToString( text, conv );
        gConsoleOutput << " = " << conv;

        GetNextProperty( prop, &prop, &error);
        }

    DisposeHandle( text );
    }
```

Navigating Data Trees

The following code shows how to navigate a decomposed data tree using the Monte API:

```
void DumpDataTree( NodeRef root, short indentLevel )
    {
    NodeRef current = { 0L, 0L};
    Error  error;
    unsigned long count =0;

        // Dump properties attached to the root node
    DumpPropertyInfo( root, indentLevel );

        // Get first child of the root node
    GetChildNode( root, &current, &error );
```

```
        // for each child, dump the information for that node
    while ( (current.l1|current.l2) != 0L )
        {
        gConsoleOutput << std::endl;
        DumpNodeInfo( current, indentLevel);   // see above.

            // If this node has children, dump their info too
        GetNodeChildCount( current, &count, &error );
        if ( count )
            {
            DumpDataTree( current, indentLevel+1 );
            }

            // move to next node
        GetNextNode( current, &current, &error );
        }
    }
```

Re-packing Data Trees

The following code shows how to reproduce a packed data block from a decomposed data tree:

```
void RepackData( NodeRef tree )
        {
        ErrorLogRef errors = NewErrorLog( );

        FlushErrorLog( errors );
        Handle myHand = NewHandle( 0L );
        std::cout << std::endl << "PackDataBlockToHandle: ";
        PackDataBlockToHandle( tree, myHand, errors );
        std::cout << "done. ";
        DumpErrors( errors );

        DisposeErrorLog( errors );
        }
```

4. Programmers' Reference

Data Types

Numeric

```
typedef union
   {
   unsigned long long unsignedValue;
   signed long long signedValue;
   } Numeric;
```

All numeric data types in Monte are represented using the 64-bit Numeric data type. Values of type Numeric may be signed or unsigned depending on context. (See **IsNodeDataSigned** function).

Textual data

All textual data in Monte is passed or returned using MacOS memory handles. The text itself is encoded as Unicode/ISO10646 characters in UTF-2 format (one 16-bit word per character, or two 16-bit words for surrogate character pairs).

Byte-order of the characters is platform native: for PowerPC based systems, character words are presented with Most Significant Bytes first in memory.

To find the length of a string, use the Memory Manager **GetHandleSize** function and divide the result by two:

```
long stringCharCount = GetHandleSize( theTextHandle ) >>1;
```

TranslatorRef

```
typedef unsigned long TranslatorRef;
```

The opaque TranslatorRef type is used to store a reference to a data translator. TranslatorRef objects should be initialised to 0, which represents "no translator". Certain translator management functions may return a zero-value TranslatorRef.

NodeRef

```
typedef struct
   {
   unsigned long l1;
   unsigned long l2;
   } NodeRef;
```

The NodeRef type is used to store a reference to a particular node of the decomposed data tree. NodeRef is an opaque type and should not be modified directly by users of the Monte API.

A null NodeRef is one in which both long values are zero. This can be detected as follows:

```
if ( (myNodeRef.l1 | myNodeRef.l2) ==0L)
   {
   // NULL node...
   }
```

Checking for a null node-reference is the only condition where callers of the Monte API should access the internal members of NodeRef. Under no circumstances should callers *modify* the internal members of a NodeRef value.

PropertyRef

```
typedef NodeRef PropertyRef;
```

The PropertyRef type is used to represent properties in the decomposed data tree. The same rules apply to PropertyRef objects as to NodeRef (above).

ErrRecord

```
struct ErrRecord
   {
   unsigned int severity:3, privateCode:13;
   signed int osError:16;
   };
```

The ErrRecord structure is used throughout the Monte API to encode error or status information. The fields of the ErrRecord structure have the following meanings:

ErrRecord Field	*Meaning*
severity	severity of the error (see below)
privateCode	MonteAPI error code (see Errors for details)
osError	Closest equivalent OS Error code.

If no error, warning or other information was returned by an API call, all of these fields will be zero. Only when all fields are zero has no error occurred: do not test the fields in isolation to determine if an error has occurred!

Severity codes are used by Monte to distinguish between errors, warnings and mere informational messages. The possible severity codes are:

Error Severity	*Meaning/Suggested Action*
kInformationOnly	This is an information message: no action should be taken.
kWarning	A warning. Should be logged, but there is no need to act.
kError	Error, but benign enough just to be noted for later remedial action.
kErrorSuggestAbort	Serious error: caller should abort their current operation
kErrorSuggestTerminate	Fatal error: caller should quit before bad things happen.

The privateCode member contains the internal, Monte-specific error code which describes the error condition. These codes are listed in the section "Errors" (below). Note that some erroneous operations will set the osError member of the ErrRecord structure in preference to the privateCode member.

The osError member contains the MacOS error code which occurred. This field may or may not be zero when an error occurs.

Error

```
typedef union
   {
   unsigned long l;
   struct ErrRecord e;
   } Error;
```

The Error type is the main means of obtaining the status of calls to the Monte API. The `l` member of the Error structure provides a quick means of checking for a no-error condition:

```
if ( theError.l ==0L )
   {
   // No error...
   }
```

ErrorLogRef

```
typedef void* ErrorLogRef;
```

The ErrorLogRef type is used for calls which can produce multiple errors. ErrorLogRef represents an Error Log. Error Logs must be created before they are used, and should be disposed of when no longer needed (See **Error Reporting**).

Any call requesting an ErrorLogRef parameter may also be passed the special **kNoErrorLogging** constant which will supress logging of errors.

Error Codes

General Errors

code	*meaning*
kErrUnknownFault	Should never occur. Indicates an internal fault.

Translator Archiving and Retrieval Errors

code	*meaning*
kErrBadTranslatorReference	TranslatorRef does not point to a valid translator.
kErrUnknownTranslatorWorld	Tried to specify a world which is not present in the translator archive.
kErrUnknownTranslatorName	Caller asked for a translator which is not in archive.
kErrTranslatorIsNotOfCorrectType	Caller asked for a translator which is in archive, but the archive's version had different "export" or "loaded" flags.
kErrTranslatorIsInUse	Caller tried to remove a translator from the archive while it was still being used. Bear in mind that the decomposed data tree requires its corresponding data translators.
kErrNotABaseTranslator	Caller passed a variant translator reference to a base-translator specific function.
kErrNotAVariantTranslator	Caller passed a base translator reference to a variant-translator specific function.
kErrTranslatorNotFound	Requested translator could not be found.
kErrWorldIndexOutOfRange	Caller passed an invalid world index to **GetIndWorld**.

Translator Syntax Errors

code	*meaning*
kErrUnknownVariantHeadTranslator	Translator file tried to declare a variant of a non-existant translator.

kErrMultipleDefaultVariants	Translator file tried to declare two or more default variants (ie., ones with no rule) for a given translator.
kErrExpectedMonte	Translator file did not begin with the 'monte' keyword.
kErrExpectedTranslator	Expected a translator definition.
kErrKeywordNotValidInHeader	Translator file contained a keyword in the header section (ie., that before the first 'translator' definition) which was not valid there.
kErrMissingWorldSpecifier	The 'world' keyword was not followed by a world specification.
kErrBadWorldSpecifier	The 'world' keyword was followed by a badly-formed world specification.
kErrMissingTranslatorKeyword	The keyword 'translator' was expected.
kErrBadTranslatorName	A badly-formed translator name followed the 'translator' keyword.
kErrBadVariantSpecification	Monte encountered a badly-formed variant specification.
kErrExpectedInstruction	An instruction was expected.
kErrBadRule	A translator variant description contained a badly-formed variant rule specification.
kErrBadEqualityRule	Monte encountered a badly-formed equality rule in a translator varant description.
kErrMissingLiteralValue	Equality rule specifications must contain a literal value.
kErrPresentationNotValidHere	The data-presentation keywords ('oct', 'bin', 'hex', 'dec') were placed in front of an instruction which cannot use them.
kErrBadIntegerInstruction	A badly-formed integer instruction was encountered
kErrIncompleteNoLabel	Expected an element name defintion.
kErrIncompleteNoDescription	Expected an element description.
kErrInvalidLabel	Specified instruction label is badly-formed or illegal in this position.

kErrInvalidDescription	Specified descriptive text is badly-formed or illegal.
kErrSyntaxError	Syntax error.
kErrBadIntegerConstant	Translator contained a badly-formed integer constant definition.
kErrMalformedCharInstruction	Translator contained a badly-formed character instruction.
kErrBadCharacterConstant	Translator contained a badly-formed character constant definition.
kErrMissingSizeSpecifier	Instruction must take a size specifier, but this was omitted in the translator.
kErrMissingElementReference	Instruction must take an element referece, but this was omitted in translator.
kErrMissingPstrSizeSpecifier	Pascal-string instruction was missing a size specifier.
kErrMissingSquareBracket	Square bracket expected (for 'repeat', fixed-length strings, etc.)
kErrMissingTranslatorName	Expected name of translator (for sub-structures, insertion, lists, etc)
kErrIncompleteStringInstruction	String definition was expected
kErrIntSizeSpecBracket	Expected size specification or label.
kErrOptionalQuotedString	Expected a descriptive text string.
kErrEof	End of file reached prematurely.
kErrMissingTranslatorVersion	Translator file is missing a translator-language version.
kErrExpectedLinefeed	Expected new line.
kErrMissingCloseParenthesis	Missing close-bracket ')'.
kErrMissingEquals	Missing equals sign '='.
kErrMissingCloseBrace	Missing close-curly-bracket '}'.
kErrMissingQuotedString	Missing quoted string.
kErrMissingSizeSpec	Missing data-size specification.
kErrMissingLabel	Missing label.
kErrMissingPositiveInteger	Missing numeric value
kErrMissingCloseSquareBracket	Missing close-square-bracket ']'.
kErrMissingAssignment	Missing assignment in instruction.

kErrUnrecognisedCharacter	Invalid character found in file.
kErrMultiCommentEnd	Instructions may not follow on the same line as the end of a multi-line comment.
kErrIllegalLineBreakInString	Strings may not contain line-breaks.
kErrIllegalLineBreakInPackedArray	Packed-character arrays may not contain line-breaks.
kErrBadUnaryMinus	Unary minus not valid in this position.
kErrTranslatorVersionIsUnsupported	The version of the translator language in the file is not supported by this version of Monte.
kErrUnsupportedPascalStringSize	Character-size specified in a pascal-style string instruction is not supported in this version of Monte.
kErrCouldNotOpenFile	A file specified by an 'include' instruction could not be found, or if found, could not be opened.

Data Decomposition Errors

code	*meaning*
kErrNotEnoughData	Translator did not finish decomposition before data stream ran out: either the wrong translator was used, or the input data is damaged or not of the format expected by the data translator.
kErrMismatchedConstantValue	The data translator specified a constant numeric or text value, but the supplied data didn't contain that value.

Data Repacking Errors

code	*meaning*
kErrOutputBufferTooSmall	Overflow of caller's output buffer occurred when repacking.

Navigation and Inspection Errors

code	*meaning*

kErrBadNodeReference	Supplied NodeRef does not point to a valid node.
kErrNodeIsNotAContainer	Function was expecting a NodeRef for a container node.
kErrNodeIsNotAList	Function was expecting a NodeRef which referred to a list.
kErrNodeIsNotWritable	The supplied NodeRef referred to a read-only node.
kErrNodeIsNotNumeric	Function expected to be passed a numeric data node.
kErrNodeIsNotWritableNumeric	Supplied NodeRef is numeric, but not writable (some numeric nodes generate their own values, and are thus read-only).
kErrNodeIsNotTextNode	Function requires a textual data node.
kErrNodeHasNoData	The supplied node has no data (it may be a container, or a filler node).
kErrNotRootNode	Function call required root node of the decomposed data tree, but some child node was passed to it.

Property Errors

code	*meaning*
kErrBadPropertyReference	Supplied PropertyRef does not point to a property
kErrPropertyIsNotModifiable	Tried to change the value of non-modifiable property.

Data Modification Errors

code	*meaning*
kErrValueTooLarge	Attempted to set the value of an integer node to a value which is too big for the node.
kErrNodeCannotEncodeSuppliedText	Attempted to set the value of a text node to text which cannot be encoded in the data tree's current destination text encoding.

kErrTextIsTooLong	Attempted to exceed a textual node's maximum text length (remember that the maximum length is in *encoded* words).
kErrCodePointOutOfRange	The supplied text contained out-of-range encoded characters. This occurs when the text itself is encodable in the destination encoding, but the encoded values are too big to fit in the destination *words* (eg. trying to squeeze 8-bit characters into a 7-bit string).

Warning Codes

Translator Archiving Warnings

code	*meaning*
kWarningAtEndOfTranslatorList	Caller tried to move past end of translator list.

Data Decomposition Warnings

code	*meaning*
kWarningTooMuchData	Translator did not have to read all of the data in supplied stream in order to create data tree.

Data Inspection Warnings

code	*meaning*
kWarningAtEndOfNodeList	Caller tried to move past end of the node list.
kWarningAtEndOfPropertyList	Caller tried to move past end of property list.
kWarningNodeHadNoParent	Caller attempted to get the parent node of the root node of a tree.
kWarningTargetNotFound	For offset/sizeof/location nodes: could not find the node to be measured when asked to calculate node value.

Data Modification Warnings

code	*meaning*
kWarningNodeTextNotEncodable	After changing encoding of the data tree, this node's text is no longer encodable.

Error Management Functions

Some functions in the Monte API are capable of returning more than one error or warning message. To deal with this, the Monte API uses *error logs*. An error log is a list of error, warning or status messages as generated by one or more API calls.

Error logs are opaque objects, represented by the ErrorLogRef data type.

The following functions are used to create, manage and inspect error logs:

NewErrorLog

```
pascal ErrorLogRef NewErrorLog( void );
```

Creates a new Error Log. Use the returned ErrorLogRef to pass to functions which may return more than one warning or error.

Errors below the severity "kError" will not be logged to this log.

This function returns kNoErrorLog if an error occurs. The most likely cause of an error is a low-memory condition.

NewErrorLogThreshold

```
pascal ErrorLogRef NewErrorLogThreshold( enum EErrorSeverity
    lowerThreshold );
```

Creates a new error log. The returned ErrorLogRef can be passed to functions which may return more than one warning or error.

Errors below the severity specified in lowerThreshold will not be logged to the new error log. This function is useful for operations where the end user wishes to receive warnings as well as errors (by default, error logs as created with NewErrorLog will accept on messages of greater severity than kError).

This function returns kNoErrorLog if an error occurs. The most likely cause of such a failure is a low-memory condition.

DisposeErrorLog

```
pascal void DisposeErrorLog( ErrorLogRef errors );
```

This function disposes of an error log, and releases the memory used internally for the log's items.

The input ErrorLogRef should not be used again after calling DisposeErrorLog.

FlushErrorLog

```
pascal void FlushErrorLog( ErrorLogRef errors );
```

This function removes all errors, warnings and messages from an error log, but keeps the internal log object for further use.

ErrorOccurred

```
pascal Boolean ErrorOccurred( ErrorLogRef errors );
```

This function returns true if log contains at least one entry whose severity is kError or higher.

ErrorLevelOccurred

```
pascal Boolean ErrorLevelOccurred( ErrorLogRef errors, enum
    EErrorSeverity severity );
```

This function returns true if the log “errors” contains a message at least as serious as the severity code“severity”.

A false result from ErrorLevelOccured does not mean that the error log is empty: there may be messages in the log whose severity was below the selected level.

CountErrors

```
pascal short CountErrors( ErrorLogRef errors );
```

This function returns the number of entries in the error log. Testing the result of CountErrors against zero (or a previous known error count) is an acceptable method of determining whether a function call has produced errors.

GetIndErrorCode

```
pascal OSErr GetIndErrorCode( ErrorLogRef errors, short index );
```

Use this function to return the MacOS error code for an individual entry in an error log. Be aware that not all entries in the log will return an OS error code: a warning, for instance, could simply return noErr.

index should be between zero and the result of CountErrors (see above) minus one.

GetIndPrivateErrorCode

```
pascal short GetIndPrivateErrorCode( ErrorLogRef errors, short index );
```

This function returns the internal Monte error code for an individual entry in an error log. Be aware that not all entries in the log will return an Monte error code, although most do. Sometimes, the OS error code is sufficient to indicate the problem.

index should be between zero and the result of CountErrors (see above) minus one.

GetIndErrorSeverity

```
pascal enum EErrorSeverity GetIndErrorSeverity( ErrorLogRef errors,
    short index );
```

This function is used to determine how severe a given log message is (see "ErrRecord" in "Data Types" above for the severity codes).

index should be between zero and the result of CountErrors (see above) minus one.

IsIndErrorParserError

```
pascal Boolean IsIndErrorParserError( ErrorLogRef errors, short index
    );
```

This function returns true if an entry in the error log is a parser error. Parser errors contain additional information which can be extracted using GetIndParserErrorInfo (see below).

index should be between zero and the result of CountErrors (see above) minus one.

GetIndParserErrorInfo

```
pascal void GetIndParserErrorInfo( ErrorLogRef errors, short index,
    Handle filename, unsigned long* lineNumber, Handle lineText,
    unsigned long* errorPosition );
```

This function returns additional information for parser errors. The return values are as follows:

parameter name	*description*
filename	will be filled with name of file containing error
lineNumber	will contain line number of error
lineText	will be filled with the text of the offending line
errorPosition	will contain the location of the error.

Note that the handles lineText and filename should be created by the caller before calling GetIndParserErrorInfo: GetIndParserErrorInfo will resize and fill these handles as appropriate, but they remain the property of the caller (ie., the caller is responsible for their disposal).

index should be between zero and the result of CountErrors (see above) minus one.

IsIndErrorNodeError

```
pascal Boolean IsIndErrorNodeError( ErrorLogRef errors, short index );
```

This function returns true if an entry in the error log has an attached node reference (for example, the node in the tree at which the error occurred). This additional node reference can be obtained by calling GetIndNodeErrorInfo (below).

index should be between zero and the result of CountErrors (see above) minus one.

GetIndNodeErrorInfo

```
pascal void GetIndNodeErrorInfo( ErrorLogRef errors, short index,
    NodeRef* returnedNode );
```

This function returns the node reference associated with an error log entry. The returned node reference may be used just as any other node reference.

index should be between zero and the result of CountErrors (see above) minus one.

LogError

```
pascal void LogError( ErrorLogRef whereTo, OSErr macErrorCode, short
    privateErrorCode, enum EErrorSeverity severity );
```

The LogError function allows an error message to be written into an error log.

While this function is provided primarily for authors of feeder and flush functions (for data decomposition and repacking, respectively), it may also be of use in other areas.

General Functions

InitMonte

```
pascal void InitMonte( void );
```

This function must be called before using any other Monte API call. Failure to do so will cause severe memory corruption.

TerminateMonte

```
pascal void TerminateMonte( void );
```

This function disposes of the Monte library's internal data structures. Calling TerminateMonte is only necessary if the calling process is not also terminating.

Translator Management Functions

AddTranslators

```
pascal void AddTranslators( const FSSpec* where, ErrorLogRef error );
```

The AddTranslators function adds translators in the file specified in the parameter 'where' to the internal transaltor archive. Once added to the archive, translators may be accessed and used by other Monte API functions.

DisposeAllTranslators

```
pascal void DisposeAllTranslators( );
```

Call this function to remove all translators from the internal translator archive. After calling this function, no translators will be available for use by the Monte library unless a subsequent call to AddTranslators (above) is made.

DisposeNamedTranslators

```
pascal void DisposeNamedTranslator( Handle world, Handle name, Error*
    error );
```

Use this function to remove a particular translator from the translator archive.

DisposeTranslatorsInFile

```
pascal void DisposeTranslatorsInFile( const FSSpec* theFile, Error*
    error );
```

This function is used to remove from the archive all translators which originated in the file specified in 'theFile'. This function is useful if the caller detects (or implicitly knows) that the file has been moved, deleted, or otherwise modified.

DisposeTranslatorsInWorld

```
pascal void DisposeTranslatorsInWorld( Handle world, Error* error );
```

This function is used to remove from the archive all translators which belong to the translator world specified in the world with the name specified in 'world' (as a sequence of Unicode/ISO 10646 characters).

GetWorldCount

```
pascal short GetWorldCount( Error* error );
```

Returns the number of translator worlds in the translator archive.

GetIndWorld

```
pascal void GetIndWorld( short index, Handle retunedWorld,
    Error* error);
```

Given an index into the list of worlds in the translator archive, this function returns the name of the world at that index. This returned world may be passed to any other translator management function which requires a world specification.

'index' should be between zero and the value returned by GetWorldCount minus one.

GetTranslator

```
pascal void GetTranslator( Handle world, Handle name, enum
    ETranslatorType typeToFind, TranslatorRef* returnedTranslatorRef,
    Error* error );
```

This function is used to retrieve a translator from a given translator world within the translator archive. The translator is specified by name. If a translator is found, its TranslatorRef is returned in the 'returnedTranslatorRef' parameter. Note that it is the responsibility of the caller to release this translator reference when it is no longer required.

'typeToFind' is used to reject certain translators, as shown below:

typeToFind	*meaning*
ePrivateLoadedTranslator	Translator must be loaded and not be an 'export' translator.
eExportLoadedTranslator	Translator must be loaded and must be a public ('export') translator.
ePrivateUnloadedTranslator	Translator must not be loaded and must not be an 'export' translator
eExportUnloadedTranslator	Translator must not be loaded and must be an 'export' translator.
eAnyTranslator	Any translator of the correct name, whether in memory or not, or export or private.
eAnyLoadedTranslator	Any translator, so long as it is already in memory.
eAnyUnloadedTranslator	Any translator, so long as it is not already in memory.
eAnyExportTranslator	Translator must be an 'export' translator.
eAnyPrivateTranslator	Translator must not be an 'export' translator.

note: GetTranslator will attempt to release the existing translator object pointed to by 'returnedTranslatorRef' if the value of 'returnedTranslatorRef' is not 0L when this function is called. Thus, if this is the first time calling GetTranslator, the 'returnedTranslatorRef' TranslatorRef **must** be initialised to 0L.

GetFirstTranslator

```
pascal void GetFirstTranslator( Handle world, enum ETranslatorType
    typeToFind, TranslatorRef* returnedTranslatorRef, Error* error );
```

Returns the first translator in the world whose name is specified in 'world' and is of the type specified by 'typeToFind' (see GetTranslator for meaning of 'typeToFind' values).

note: GetFirstTranslator will attempt to release the existing translator object pointed to by 'returnedTranslatorRef' if its value is not 0L when this function is called. Thus, if this is the first time calling GetFirstTranslator, the 'returnedTranslatorRef' TranslatorRef **must** be initialised to 0L.

GetNextTranslator

```
pascal void GetNextTranslator( TranslatorRef thisTranslator, enum
    ETranslatorType typeToFind, TranslatorRef* returnedTranslatorRef,
    Error* error );
```

Returns the next translator in the archive after the translator specified by 'thisTranslator' whose type specified by 'typeToFind' (see GetTranslator for meaning of 'typeToFind' values).

ReleaseTranslator

```
pascal void ReleaseTranslator( TranslatorRef* theTranslator, Error*
    error);
```

Call ReleaseTranslator when a translator is no longer required. Doing so will allow Monte to perform internal garbage-collection and save memory.

note: Never pass an invalid TranslatorRef to this function. Doing so may seriously corrupt Monte's internal data structures.

GetTranslatorName

```
pascal void GetTranslatorName( TranslatorRef theTranslator, Handle
    returnedName, Error* error );
```

Returns the name of the translator specified in 'theTranslator'. The 'returnedName' handle must be a valid memory handle (do *not* pass in NULL), which Monte will resize and fill as appropriate.

IsTranslatorLoaded

```
pascal void IsTranslatorLoaded( TranslatorRef theTranslator, Boolean*
    isLoaded, Error* error );
```

Returns true if the translator specified has been loaded into memory. For efficiency reasons, translators are not loaded and parsed until they are actually required to decompose data.

Very few applications will need to call this function.

GetTranslatorLocation

```
pascal void GetTranslatorLocation( TranslatorRef theTranslator, FSSpec*
    returnedLocation, Error* error );
```

Returns the location of the file in which the specified translator resides.

GetTranslatorType

```
pascal void GetTranslatorType( TranslatorRef theTranslator, enum
    ETranslatorType* returnedType, Error* error);
```

Returns the type of the specified translator - loaded or unloaded, export or private. The returned type may be inspected by AND-masking with the ePrivateOrExportTranslatorState or eLoadedOrUnloadedTranslatorState masks, as illustrated in the code sample below:

```
GetTranslatorType( myTranslator, &type, &error );
    switch ( type&ePrivateOrExportTranslatorState )
```

```
        {
        case  ePrivateTranslator:
            //  private  (non-'export'  translator)

        case  eExportTranslator:
            //  export  translator
        }
    switch  (  type  &  eLoadedOrUnloadedTranslatorState  )
        {
        case  eLoadedTranslator:
            //  translator  is  in  memory

        case  eUnloadedTranslator:
            //  translator  is  not  in  memory.
        }
```

Most applications will never need to call the GetTranslatorType function.

GetFirstTranslatorVariant

```
pascal  void  GetFirstTranslatorVariant(  TranslatorRef  headTranslator,
    TranslatorRef*  returnedVariantTranslator,  Error*  error  );
```

This function returns the first variant of the translator specified in 'headTranslator'. The first variant of that translator will be returned in 'returnedVariantTranslator'. It is the responsibility of the caller to release this returned reference when it is no longer required.

note: GetFirstTranslatorVariant will attempt to release the existing translator object pointed to by 'returnedVariantTranslator' if the value of 'returnedVariantTranslator' is not 0L when this function is called (see ReleaseTranslator). TranslatorRef objects **must** be initialised to 0L before attempting to use them.

If the translator specified in 'headTranslator' has no variants, then the returned variant translator reference will be null.

GetVariantHeadTranslator

```
pascal  void  GetVariantHeadTranslator(  TranslatorRef  variantTranslator,
    TranslatorRef*  returnedHeadTranslator,  Error*  error  );
```

This function is used to find the base translator of a given variant translator. The base, or variant head, translator will be returned in 'returnedHeadTranslator'. It is the responsibility of the caller to release this returned reference when it is no longer required.

note: GetVariantHeadTranslator will attempt to release the existing translator object pointed to by 'returnedHeadTranslator' if the value of 'returnedHeadTranslator' is not 0L when this function is called (see ReleaseTranslator). For this TranslatorRef objects **must** be initialised to 0L before attempting to use them.

Data Extraction & Creation Functions

DecomposeDataBlock

```
pascal void DecomposeDataBlock( TranslatorRef translator, void *
    dataBlock, unsigned long dataLength, TextEncoding dataTextEncoding,
    NodeRef* returnedTreeRoot, ErrorLogRef errors );
```

This function is used to decompose a block of data into a decomposed data tree.

The parameter 'translator' should be a valid TranslatorRef (see GetTranslator) referring to a data translator which describes the data to be decomposed.

'dataTextEncoding' is a MacOS Text Encoding Manager constant which indicates how 8-bit text is encoded in the packed data block. For example, when decomposing Japanese text, this constant would be kTextEncodingMacJapanese; for English and most Western European languages, it would be kTextEncodingMacRoman.

The root data node of the decomposed tree is returned in the NodeRef pointed to by 'returnedTreeRoot'. Note that even if errors occur, the returned NodeRef may be valid, and hence will need disposal.

The parameter 'dataBlock' should point to the beginning of the block to be decomposed, and 'dataLength' should contain the length of the packed data block in full.

Errors in decomposition will be logged into the error log 'errors'. To supress logging of errors, the constant kNoErrorLogging may be passed in place of a valid ErrorLogRef.

CreateEmptyDataTree

```
pascal void CreateEmptyDataTree( TranslatorRef translator, NodeRef*
    returnedTreeRoot, ErrorLogRef errors );
```

The CreateEmptyDataTree function is used to create a default-value data tree which corresponds to the translator 'translator'. A reference to the returned tree's root node is passed back in the NodeRef pointed to by the parameter 'returnedTreeRoot'.

The data tree created by this function will usually have all of its data nodes set to their default values, as listed below:

node type	*default value*
non-constant numeric	zero
non-constant strings	empty string

constant numeric	constant value
constant string	constant value
list (“list” instruction)	empty list (no items)
fixed-length list (“repeat” instruction)	*n* items, each of which is an empty data tree corresponding to the list’s item translator.
substructures (“sub” instruction)	an empty data tree corresponding to the substructure’s translator

The translator ‘translator’ may also be a variant translator: in this case, the created data tree will have some data nodes set to whichever values are necessary for the created data structure to conform with the variant translator’s rules.

Data-tree Navigation Functions

GetNextNode

```
pascal void GetNextNode( NodeRef thisNode, NodeRef *returnedNextRef,
    Error* error );
```

This function returns the next node at same level of the data tree as ‘thisNode’. GetNextNode will skip empty nodes (those with no data, such as alignment or padding fields). To iterate over empty nodes as well as data nodes, use GetNextNodeByType.

GetChildNode

```
pascal void GetChildNode( NodeRef thisNode, NodeRef *returnedChildRef,
    Error* error );
```

This function returns the first child node of the container node ‘thisNode’. GetChildNode will skip empty nodes (those with no data, such as alignment or padding fields). To iterate over empty nodes as well as data nodes, use GetChildNodeByType.

GetParentNode

```
pascal void GetParentNode( NodeRef thisNode, NodeRef
    *returnedParentRef, Error* error );
```

This function returns parent node of the node ‘thisNode’. By defintion, the node in ‘returnedParentRef’ will be a container node of some type (unless of course, GetParentNode is called for a node with no parent, such as the root node of the data tree).

GetNextNodeByType

```
pascal void GetNextNodeByType(NodeRef thisNode, enum ENodeType
    includeThese, NodeRef *returnedNextRef, Error* error);
```

Use this function to iterate over all nodes at a given level of the data tree. The enumeration 'includeThese' represents the type of nodes to be included. Possible values are:

'includeThese' value	meaning
eAllNodes	Any node
eEmptyNode	Any node without editable data
eContainerNode	Any type of container node.
eNonEmptyNode	Any node with some kind of data (also includes container nodes).

GetChildNodeByType

```
pascal void GetChildNodeByType(NodeRef thisNode, enum ENodeType
    includeThese, NodeRef *returnedChildRef, Error* error);
```

Returns the first child of the node 'thisNode' whose data type is of the class 'includeThese'. Values of 'includeThese' are as for GetNextNodeByType above.

Data-tree Inspection

GetNodeType

```
pascal void GetNodeType( NodeRef thisNode, enum ENodeType*
    returnedType, Error* error );
```

Returns the type of the data-tree node 'thisNode'. 'returnedType' is a combination of one or more of the following flags:

'returnedType'	*node type*
eDataNode	Text or numeric data node.
eSubcontainerNode	Node represents a subcontainer (for example, the result of the 'sub' instruction).
eListItemNode	Node represents a list item. Bear in mind that list item nodes are also container nodes.
eConstValueNode	Node represents a constant value.
eAutomaticNode	Node's value is automatically generated (eg. result of 'sizeof', 'offset' or 'location' instructions).

eListNode	Node represents a list. This node will have zero or more child nodes of type ‘eListItemNode’.

GetNodeLabel

```
pascal void GetNodeLabel( NodeRef thisNode, Handle returnedLabel,
    Error* error );
```

This function returns the label attached to the data-tree node ‘thisNode’. The label, which is a text-string, normally comes from the data translator instruction which generated the node, except for list element nodes, whose label is the name of the translator describing the list element.

Not all nodes will have a label: those that do not will simply empty the handle ‘returnedLabel’.

Note that the handle ‘returnedLabel’ and its contents remain the property of the caller of this function, and thus the caller must eventually dispose of this handle.

GetNodeDescription

```
pascal void GetNodeDescription( NodeRef thisNode, Handle
    returnedDescription, Error* error );
```

This function returns the descriptive text attached to the data-tree node ‘thisNode’. This description comes from the data translator instruction which generated the node. The descriptive text is intended for user-interface purposes, and should be used in preference to a node’s label in situations where a description of the node is required (for example, the caption for an editable-text field for editing the node’s value).

Not all nodes have descriptive text attached to them. In such situations, the ‘returnedDescription’ handle is emptied. On dectecting this, a user-interface module may fall-back to using the node’s label text, if present.

Note that the handle ‘returnedDescription’ and its contents remain the property of the caller of this function, and thus the caller must eventually dispose of this handle.

GetNodeChildCount

```
pascal void GetNodeChildCount( NodeRef thisNode, unsigned long
    *returnedChildCount, Error* error );
```

This function returns the number of nodes contained as children of the node ‘thisNode’. GetNodeChildCount will return the number of child nodes in ‘returnedChildCount’. Zero is returned for nodes which are not containers (or for containers which just don’t have any child-nodes).

GetNodeDataType

```
pascal void GetNodeDataType( NodeRef thisNode, enum ENodeDataType*
    returnedType, Error* error );
```

GetNodeDataType returns the kind of data which is contained in the node 'thisNode'. This is not to be confused with the type of the node itself (see GetNodeType). Possible return values, in 'returnedType', are:

'returnedType' value	*meaning*
eNoData	Node contains no data. Note that this doesn't necessarily make the node an empty node: container nodes, for example, will return 'eNoData'.
eTextData	Node's data is text.
eNumericData	Node's data is numeric.
eReadOnlyNumericData	Nodes's data is numeric, but may not be modified.
eReadOnlyTextData	Node's data is textual, but may not be modified.

GetNodeNumericData

```
pascal void GetNodeNumericData( NodeRef thisNode, Numeric*
    returnedValue, Error* error );
```

This function returns the numeric data of a data node, providing of course that the node contains numeric data. 'returnedValue' is a 64-bit value representing the node's value. For signed data nodes (see IsNodeSigned) with negative values, 'returnedValue' will be sign-extended to 64 bits, regardless of the data node's actual width.

IsNodeSigned

```
pascal void IsNodeSigned( NodeRef thisNode, Boolean* isSigned, Error*
    error );
```

This function reports whether or not the numeric node 'thisNode' contains signed data. The result 'isSigned' is true if the data is a 2's-complement signed value, and false if the value of the node is a simple unsigned magnitude.

GetNodeTextualData

```
pascal void GetNodeTextualData( NodeRef thisNode, Handle returnedValue,
    Error* error );
```

This function returns the value of the data node 'thisNode' in a textual form. If 'thisNode' is a numeric node, the node's value is converted to a text string according to the node's data presentation.

GetNodeTextCharacterCount

```
pascal void GetNodeTextCharacterCount(NodeRef thisNode, unsigned long
    *returnedLength, Error* error);
```

This function returns the length of a node's text in Unicode/ISO10646 characters. Note that this character count may be smaller that the number of characters required to encode the node's text using the data tree's text encoding scheme. The number of *encoded* characters required to represent the node's text can be obtained by calling GetNodeTextEncodedCharacterCount (qv).

Nodes containing no textual data will return the error kErrNodeIsNotTextNode.

GetNodeTextEncodedCharacterCount

```
pascal void GetNodeTextEncodedCharacterCount(NodeRef thisNode, unsigned
    long *returnedLength, Error* error);
```

This function returns the number of characters required to represent a node's text when encoded using the data tree's text encoding scheme.

If used with nodes which do not represent textual data, this function will return the error kErrNodeIsNotTextNode.

IsNodeTextConstrained

```
pascal void IsNodeTextConstrained( NodeRef thisNode, Boolean
    *isConstrained, Error* error);
```

This function is used to report whether or not the textual data represented by a node ('thisNode') must be less than a certain length.

If the node's text is limited, IsNodeTextConstrained returns a "true" value in the Boolean variable pointed to by 'isConstrained'; otherwise, the value at 'isConstrained' will be set to zero.

Note that all Pascal-style strings are by definition constrained to a maximum length (the maximum value which can be contained in their "length" field). Thus, nodes representing Pascal-style strings will return a value of "true" in 'isConstrained'.

If used with nodes which do not represent textual data, this function will return the error kErrNodeIsNotTextNode.

GetNodeTextMaxEncodedCharacterCount

```
pascal void GetNodeTextMaxEncodedCharacterCount(NodeRef thisNode,
    unsigned long *returnedLength, Error* error);
```

For nodes whose text is constrained in length, this function will return the maximum number of encoded words allowed for the node's text. The long pointed to by 'returnedLength' will contain this maximum character count.

Note that this function will return a correct upper character count even for textual data nodes which are not explicitly constrained in length. For these nodes, the value 2^{31}-1 is returned, which is the largest possible length for a string in Monte.

If used with nodes which do not represent textual data, this function will return the error kErrNodeIsNotTextNode.

GetFirstProperty

```
pascal void GetFirstProperty( NodeRef thisContainerNode, PropertyRef
    *returnedProperty, Error* error );
```

The GetFirstProperty function returns a property reference (the PropertyRef pointed to by 'returnedProperty') for the first property of the container node 'thisContainerNode'. If the container has no properties attached to it, a null property reference is returned.

This function may only be called with container nodes; calling with a non-container NodeRef will result in the kNodeIsNotContainer error being returned. Bear in mind that list items are also containers, and thus will posses properties.

GetNextProperty

```
pascal void GetNextProperty(PropertyRef thisProperty, PropertyRef
    *returnedNextProperty, Error* error );
```

When passed a property reference belonging to a container node (such as that returned by GetFirstProperty, above), this function returns a reference to the next property of the container. If there are no more properties attached to the container, a null property reference is returned, along with the warning kWarningAtEndOfPropertyList.

GetPropertyByName

```
pascal void GetPropertyByName( NodeRef thisContainerNode, Handle name,
    PropertyRef *returnedProperty, Error* error );
```

This function is used to find a property attached to a container by its name. The name of the required property is passed in the handle 'name'. If the container does not posses the required property then a null property reference is returned.

Note that the handle 'name' and its contents remain the property of the caller of this function, and thus the caller must eventually dispose of this handle.

GetPropertyName

```
pascal void GetPropertyName( PropertyRef thisProperty, Handle
    returnedName, Error* error );
```

Given a property reference, this function returns the name of that property in the handle 'returnedName'.

Note that the handle 'returnedName' and its contents remain the property of the caller of this function, and thus the caller must eventually dispose of this handle.

GetPropertyValue

```
pascal void GetPropertyValue( PropertyRef thisProperty, Handle
    returnedValue, Error* error );
```

Given a property reference, this function returns the value of that property in the handle 'returnedValue'.

All property values are textual: numeric values will be converted to text according to their originating data node's data presentation.

Note that the handle 'returnedValue' and its contents remain the property of the caller of this function, and thus the caller must eventually dispose of this handle.

IsPropertyModifiable

```
pascal void IsPropertyModifiable( PropertyRef thisProperty, Boolean*
    isModifiable, Error* error );
```

This function determines whether or not a property may be directly modified. Modifiable properties are those which correspond directly to a data node within the container. This data node containing the property's value may be found by calling GetPropertyValueSource (below).

GetPropertyValueSource

```
pascal void GetPropertyValueSource( PropertyRef thisProperty, NodeRef*
    returnedNodeRef, Error* error );
```

The GetPropertyValueSource function returns a reference to the data node which contains the value of the property 'thisProperty'. This function can only provide this information for modifiable property nodes (see IsPropertyModifiable, above).

If the property 'thisProperty' is not a modifiable property, the error kErrPropertyIsNotModifiable is returned, and the node reference at 'returnedNodeRef' is set to a null node reference.

GetDataTreeTextEncoding

pascal void GetDataTreeTextEncoding(NodeRef rootNode, TextEncoding *returnedEncoding, Error* error);

This function returns the text-encoding scheme which the specified data-tree uses to encode text with a character width of 8 bits or smaller. Unless subsequently changed (see SetDataTreeTextEncoding, below), this function will return the same text encoding value as originally passed to DecomposeDataBlock.

This function can return one of the following errors:

error	*cause*
kErrNotRootNode	The supplied node reference does not refer to the root of a data tree: only the root node of a data tree contains the text encoding information for the data tree.
kErrBadNodeReference	The supplied node reference does not refer to a valid node.

CalculatePackedDataSize

```
pascal void CalculatePackedDataSize( NodeRef dataTreeRoot, unsigned
    long* calculatedSize, Error* error );
```

This function is used to determine the packed size of a decomposed data tree. 'dataTreeRoot' is a reference to the root node of the tree whose size is to be calculated. The calculated size, in bytes, is returned in the longword pointed to by 'calculatedSize'.

Note that it is *not* necessary to call this function before PackDataBlockToHandle (*qv*), as the PackDataBlockToHandle function performs the size calculation anyway.

Data Modification Functions

SetNodeNumericData

```
pascal void SetNodeNumericData( NodeRef thisNode, Numeric newValue,
    Error* error );
```

This function is used to set the value of a data node to the value in the Numeric variable 'newValue'. The node 'thisNode' must be a writable numeric data node

This function can return one of the following error codes:

error	*cause*
kErrNodeIsNotWritableNumeric	The supplied node reference does not refer to a writable numeric data node.
kErrBadNodeReference	The supplied node reference does not refer to a valid node.

SetNodeTextualData

```
pascal void SetNodeTextualData( NodeRef thisNode, Handle newValue,
    Error* error );
```

This function is used to set the value of a data node to the value in the handle 'newValue'. The node 'thisNode' must be a writable data node, but may be textual or numeric. For numeric nodes, the text in 'newValue' is converted a number according to the rules below:

newValue text	*value is…*
`$`xxxxx… (x=0…9, a…f or A…F)	Hexadecimal
`\`xxxxx… (x=0…7)	Octal
`%`xxxxx… (x=0 or 1)	Binary
xxxxx… (x=0…9)	Decimal

Negative numeric values can be indicated by a leading '-' character (eg. "`-$20`", "`-32`", "`-\40`", "`-%10000`").

Note that the handle newValue remains the property of the caller, and thus the caller is responsible for its eventual disposal.

This function can return any of the following error codes:

error	*cause*
kErrValueTooLarge	The supplied numeric value is too large to be represented by the data node.
kErrNodeCannotEncodeSuppliedText	The supplied text cannot be encoded in the data tree's current destination text encoding.
kErrTextIsTooLong	The supplied text, when encoded, exceeds the textual node's maximum text length (remember that the maximum length is always expressed in *encoded* words).
kErrCodePointOutOfRange	The supplied text contained out-of-range encoded characters. This occurs when the supplied text is itself encodable in the destination encoding, but the encoded *values* are too big to fit in the destination words (eg. trying to squeeze 8-bit characters into a string of 7-bit characters).

kErrNodeIsNotWritable	The supplied NodeRef referred to a read-only node.
kErrBadNodeReference	The supplied NodeRef doesn't refer to a valid node.

SetPropertyValue

```
pascal void SetPropertyValue( PropertyRef thisProperty, Handle
    newValue, Error* error );
```

The SetPropertyValue function changes the value of the modifiable property 'thisProperty'. Actually, it is the data node which provides the property's value that is changed (see GetPropertyValueSource).

For numeric-value properties, the text in 'newValue' is converted to a number according to the rules below:

newValue text	*value is…*
$xxxxx… (x=0…9, a…f or A…F)	Hexadecimal
\xxxxx… (x=0…7)	Octal
%xxxxx… (x=0 or 1)	Binary
xxxxx… (x=0…9)	Decimal

Negative numeric values can be indicated by a leading '-' character (eg. "`-$20`", "`-32`", "`-\40`", "`-%10000`").

Note that the handle newValue remains the property of the caller, and thus the caller is responsible for its eventual disposal.

This function can return any of the following error codes:

error	*cause*
kErrValueTooLarge	The supplied numeric value is too large to be represented by the data node.
kErrNodeCannotEncodeSuppliedText	The supplied text cannot be encoded in the data tree's current destination text encoding.
kErrTextIsTooLong	The supplied text, when encoded, exceeds the textual node's maximum text length (remember that the maximum length is always expressed in *encoded* words).

kErrCodePointOutOfRange	The supplied text contained out-of-range encoded characters. This occurs when the supplied text is itself encodable in the destination encoding, but the encoded *values* are too big to fit in the destination words (eg. trying to squeeze 8-bit characters into a string of 7-bit characters).
kErrPropertyIsNotModifiable	The supplied PropertyRef refers to a read-only property.
kErrBadPropertyReference	The supplied PropertyRef doesn't refer to a valid property.

SetDataTreeTextEncoding

```
pascal void SetDataTreeTextEncoding( NodeRef rootNode, TextEncoding
    newEncoding, ErrorLogRef errors );
```

This function is used to change the text-encoding scheme to be used when re-packing the data tree whose root node is referred to by 'rootNode'. The new, desired, text encoding is specified in 'newEncoding' – the values for 'newEncoding' correspond directly to those used by the MacOS Text Encoding Converter library.

Any items whose text is found to be unrepresentable under the new text encoding will be logged in the error long 'errors'. To extract the node references of the nodes with unencodable text, use the IsIndErrorNodeError and GetIndNodeErrorInfo functions described previously.

SetDataTreeTextEncoding only reports unencodable text nodes; it does not modify the text of these nodes. It is the responsibility of the caller to remedy any unencodable text before attempting to re-pack a data block.

Data Packing and Disposal Functions

PackDataBlockToHandle

```
pascal void PackDataBlockToHandle( NodeRef dataTreeRoot, Handle
    destination, ErrorLogRef errors );
```

To convert a decomposed data tree back into a packed data block, use the PackDataBlockToHandle function. ‘dataTreeRoot’ is the root node of the tree to be packed.

‘destination’ is an already-allocated handle to a memory block in which the packed data structure will be placed. It is not necessary for the caller to calculate the size of the packed data, or to set the handle size, before calling this function, as PackDataBlockToHandle will resize the supplied handle to fit the resulting data block.

Note that the handle ‘destination’ remains the property of the caller. It is the caller’s responsibility to dispose of this handle when it is no longer required.

Any errors encountered during packing are logged to the error log ‘errors’. This log may contain the following errors:

error code	*cause*
kErrOutputBufferTooSmall	The packed data overran the buffer. This error indicates an internal failure.
kErrNotRootNode	Supplied node reference is not the root of the data tree.
kErrBadNodeReference	Supplied node reference does not point to a valid node.

DisposeDataTree

```
pascal void DisposeDataTree( NodeRef theTree, Error* error );
```

To dispose of a decomposed data tree, use the DisposeDataTree function. ‘theTree’ is the root node of the data tree to be disposed of.

Appendix A. Keyword Quick Reference

Keywords in the monte translator language are listed here in alphabetical order.

Parameters are shown in *italics*, required parameters are shown in ***boldface***. Some instructions have more than one form, in which case all possible forms are shown.

align

`align:`***bitMultiple***`:`*bitOffset*

Represents untranslated data, used to align subsequent elements – differs from the `fill` instruction in that the alignment field's size may vary depending on the sizes of preceding fields.

The *bitOffset* field, if present indicates which bit of the *bitMultiple*-sized word the input should be aligned to. For example, to align to the odd-byte of a 16-bit word, use `align:16:8`. If omitted, *bitOffset* defaults to zero.

bin

`bin` ***numeric-instruction***

Must precede a numeric instruction (`sint`, `uint`, etc.). Indicates that the numeric data should be presented as binary (base-2). Monte presents binary numbers with a leading "`%`"symbol.

cstr

`cstr:`*charSize*

`cstr:`*charSize*`[`*count*`]`

Represents a string of characters, terminated by a zero character value, analogous to those used by the ANSI C library. If *count* is sepecified, then the string will always occupy that many characters in the packed data (with unused characters being set to zero on output). *The terminator character is included in this count!* For example, a C-style string which must be 256 characters or less (including terminator) will be specified as `cstr[256]`. If unspecified, *charSize* defaults to 8 bits.

dec

`dec` ***numeric-instruction***

Must precede a numeric instruction (`sint`, `uint`, etc.). Indicates that the numeric data should be presented as decimal. This is the default presentation for numeric instructions, and so is rarely used explicitly.

export

… see **translator**.

fill

`fill:`*`size`*

An unimportant filler field – ignored on input, and zeroed on output. Typically used to represent the "reserved" fields in data structures.

hex

`hex` *`numeric-instruction`*

Must precede a numeric instruction (`sint`, `uint`, etc.). Indicates that the numeric data should be presented as hexidecimal (base-16). Monte presents hexadecimal numbers with a leading "`$`"symbol.

insert

`insert(` *`translator-name`* `)`

The named translator is used to decompose data. When that translator has finished, decomposition continues from the instruction following the "insert" instruction. Note that *translator-name* may refer to the head of a translator variant family, in which case the eligible variants of *translator-name* will be given a chance to decompose the data.

insert is typically used when defining inherited data structures – especially those which use muliple base structures.

Data elements produced by the translator are placed at the same level of the data tree as the `insert` instruction (compare with action of `sub`, below).

list

`list(` *`translator-name`* `)`

The named translator is used repeatedly to create a number of elements in a list. The number of elements can be controlled by a previous `listcount` (see below) instruction which refers to this `list` instruction. If no `listcount` instruction referring to this list exists, then list items are read until the end of the input data is reached (ie., a free-form list). However, if an instruction following the list instruction specifies a constant integer value, then that value, if encountered, will terminate the list before the end of data is reached.

Note that translator-name may be the head of a variant family, in which case it is possible for each item in the list to be described by a different variant of that head translator (ie., a heterogenous list).

listcount

`listcount:`*`size`*`(` *`ListLabel`* `)`

The number of items in a following list (see list, above). If omitted, *size* defaults to 16.

location

`location:`*`size`*`(` ***Label*** `)`

Contains the location (in **bytes** from the beginning of the packed data block) of the element with the name *Label*. The element specified by *Label* must exist at the same tree-level as the `location` command (ie., *Label* cannot reside in a sub-translator included in the current translator). If omitted, *size* defaults to 32. In other respects, `location` can be treated as a specialised `uint` instruction.

monte

`monte` ***version-number***

The `monte` keyword must be present in a translator file, and it must be the first instruction in the file. This keyword identifies the file as a Monte data translator, and provides information about which version of the language was used to write the translator.

The *version-number* parameter is the version of Monte required to use this translator. For translators described in this document, version-number should be `1.0`

oct

`oct` ***numeric-instruction***

Must precede a numeric instruction (`sint`, `uint`, etc.). Indicates that the numeric data should be presented as octal (base-8). Monte presents octal numbers with a leading "\"symbol.

offset

`offset:`*`size`*`(` ***Label*** `)`

Contains the offset (in **bytes**!) to the first byte representing the element specified by *Label*, calculated from the beginning of the offset data field. The element specified by *Label* must exist at the same tree-level as the `offset` command (ie., *Label* cannot reside in a sub-translator included in the current translator). If omitted, *size* defaults to 32. In other respects, `offset` can be treated as a specialised `uint` instruction.

packed

`packed` ***numeric-instruction***

Must precede a numeric instruction (`sint`, `uint`, etc.). Indicates that the numeric data should be presented as an array of 8-bit characters. Monte presents packed character integers as one or more characters surrounded by single quotes ("`'`").

On the MacOS platform, packed-character integer values are presented (and modified) using characters from the Mac Roman text encoding only.

pstr

`pstr:`*lSize*`:`*charSize*

`pstr:`*lSize*`:`*charSize*`[`***count***`]`

Represents an explicitly-sized string of characters, analogous to a Pascal string. The length of the string is stored before the characters in an unsigned word which is *lSize* bits wide. Each character is then *charSize* bits wide.

The *count* parameter controls how many characters will be allocated in the data-file for the string. If the *count* parameter is defined, then that number of characters will always be read from or written to the output: extra characters will be written as zero-value code words.

Note that the count parameter *excludes the length word.* For example, a name which must be 31 eight-bit characters or less has the definiton `pstr:8:8[31]`, and will occupy 32 bytes of the packed data (one for the length, and always 31 for the string). Monte keeps track of this size limit and will warn if the decomposed version of the string is made longer than this limit through editing. If not specified, *charSize* and *lSize* both default to 8.

repeat

`repeat[` ***repeat-count*** `](` ***translator-name*** `)`

The named translator is used repeatedly to create a number of elements in a list. The number of elements is specified in the *repeat-count* field. The resulting list is of fixed length: it cannot have items removed from, or added to, it.

As for `list` (see above) *translator-name* may refer to the head of a variant family, to allow heterogenous lists to be specified.

sint

`sint:`*size*

signed numeric value, may be any bit width from 1 to 64 bits. *size* defaults to 16

sizeof

`sizeof:`*size*

`sizeof:`*size*`(` ***Label*** `)`

Contains the packed-size (in *bytes*!) of the sub-structure with the name *Label*. If the element name *Label* is present, then the structure or element specified by *Label* must exist at the same tree-level as the `sizeof` command (ie., *Label* cannot reside in a sub-translator included in the current translator). If *Label* is omitted, then the `sizeof` command refers to the size of the entire container which contains the `sizeof` command.
size defaults to 32 if omitted. In other respects, `sizeof` can be treated as a specialised `uint` instruction.

skip

`skip`

`skip:`*`size`*

Represents data to be preserved, but not processed: data in a `skip` field is remembered at input time, and is written back verbatim on output.

If *size* is omitted, then the skip command will read in all data up to the end of the packed data block, or up to the end of the current container, if that container's size can be determined, for example if the container is the subject of a previous `sizeof` (*qv*) instruction.

string

`string:`*charsize*

`string:`*charsize*`[`*count*`]`

Represents a string of characters. If omitted, *charSize* defaults to 8 bits. If the *count* parameter is specified, then the string will occupy *count* number of code words; shorter strings will be padded with zero-value code words. If the *count* parameter is not specified, the string data is assumed to run all the way to the end of the data structure, unless the next instruction after `string` in the translator is a constant-value integer, in which case that integer's value, if found, will terminate the string.

sub

`sub(` *translator-name* `)`

Like `insert` above, the named translator is used to decompose packed data. However, unlike `insert`, a container is created at the current level of the decomposed data tree, and the results from this translator are placed in that container.

translator

`translator` *translator-name*

`translator` *translator-name* `:` *base-name*

`translator` *translator-name* `:` *base-name* `(` {*label*} `==` *value* `)`

`export translator` *translator-name*

`export translator` *translator-name* `:` *base-name*

`export translator` *translator-name* `:` *base-name* `(` {*label*} `==` *value* `)`

Marks the beginning of a new data translator. *translator-name* is the name by which the new translator is to be known: this name may be referred to by `list`, `sub`, `insert` or `repeat` instructions, as well as by further translator definitions.

If *base-name* is specified, then the new translator is taken to be a variant of the translator called *base-name*. If no rule ("{***label***} == ***value***") is specified, then the new translator will be the default variant of the translator base-name. Note that only one default variant of any given base translator is allowed: attempting to specify a default translator for a base translator which already has one will result in an error.

If the "{***label***} == ***value***" rule is included, then the translator *translator-name* will be a variant of the translator base-name: a variant which will only be used in situations where the item with the label *label* (this item will be defined in *base-name* or one of its antecedents) has the value *value*.

Translator definitions preceded with the keyword `export` will be included in the set of "exportable" translators. Exportable translators are those which the end-user of a Monte-based tool will deal with (eg., DITL, Layout, Control, Button, Pane), rather than the translators which are required to define these (eg., DITLItem, QDBoundingBox, PPobRecord).

Note that the export keyword is simply a flag attached to a translator, for the benefit of users of Monte: it does not constitute a separate namespace, and Monte does not distinguish between export and private translators when looking up translators internally.

uint

`uint:`*size*

Represents an unsigned numeric value of any bit width from 1 to 64 bits. If not explicitly specified, *size* defaults to 16.

world

`world` *world-specification*

Only allowed in file header (ie., before the first translator definition), the `world` directive is used to group translators into namespaces. Thus, translators specifically for traditional Macintosh resources can be placed in the "`mac/rsrc`" world, those for Mac OS X structures could be placed in the "`mac/osx`" world, and so on.

The world specification is similar in structure to a UNIX file path: a series of worlds are specified, each separated by a forward-slash character. This allows users of Monte to ask for translators in a particular world, such as in "`mac/rsrc/MacApp/MyApp2.0`", or to specifiy a more general world, such as "`mac`", which will cause Monte to search in subordinate translator worlds for the requested translator.

World names are case sensitive: "`Macintosh`" and "`MacIntosh`" are two separate worlds.

Appendix B. Formal Grammar of Translator Language

This appendix provides a formal description of the Monte data translator language.

The translator format is described using Extended Backus-Naur notation. Productions are numbered for reference only.

B.1. General Format

B.1.1. Encoding

Translator Files must use the Unicode/ISO 10646 text encoding scheme, either as UTF-8 or UTF-16 (in either its Big- or Little-Endian forms). Monte will detect the correct format by inspecting the beginning of the file.

Monte prefers UTF-16 with character byte-pairs stored as More-Significant, Less-Significant (for example, the text '`abc`' is represented as `0061 0062 0063`, and not `6100 6200 6300`), although support for the other representation is provided.

B.1.1.1. Encoding Detection

Translator file streams are expected to be encoded in Unicode/ISO 10646. UTF-8 or UTF-16 are currently supported; no support for UCS-32 is planned. No support is provided for legacy Macintosh encodings.

To differentiate between different encodings (UTF-8 versus UTF-16), Monte relies on the presence of a Unicode byte-order mark. This is the character U+FEFF (Zero Width Non-Breaking Space). Using this information, it is possible to determine the encoding of the stream:

file starts with...	*encoding scheme*
`ef bb ef ...`	UTF-8
`fe ff ...`	UTF-16, big-endian
`ff fe ...`	UTF-16, little-endian

Translator streams which do not meet these criteria are rejected as being invalid.

B.1.2. Packaging

B.1.2.1. MacOS HFS / HFS+ File Systems

Translator Files must be placed in the data fork of a file. Where allowed by the file-system, the file should have the file-type code '`mtpl`'. The file-creator field can be any value, perhaps representing the tool used to create the file.

For cross-platform interchange, the file-extension '`.mtpl`' (preferred) or '`.mtp`' (where forced by file-system constraints) should be used to identify Monte translator files.

B.1.2.1. Non-MacOS HFS/HFS+ File Systems

On non-MacOS platforms, the file should have the file-type 'mtpl'. The creator-type can be anything.

For cross-platform interchange, the file-extension '`.mtpl`' (preferred) or '`.mtp`' (where forced by file-system constraints) should be used to identify Monte translator files.

B.1.2.2. Non HFS File Systems

On file-systems which do-not support the file-type/file-creator model, the filename extension '`.mtpl`' (preferred) or '`.mtp`' (where forced by file-system constraints) should be used to identify Monte translator files.

B.2. Header

```
1  TranslatorFile ::= NewLine? Header Translator*
```

Every translator file must contain a valid Header section. The header section may be preceded by any number of spaces, newlines or comments. Following this, there may be any number[2] of Translator definitions (§B.3).

```
2  Header ::= 'monte' SPACE MonteVersion NewLine Directive*
```

The header for a translator must be the first code line in the translator file. Comments are allowed before the first line of the header.

B.2.1. Version Declarations

```
3  MonteVersion ::= '1.0'
```

The MonteVersion declaration declares which version of the translator format this is. Currently, only '1.0' is allowed here.

Note that this is the version of the translator format and not necessarily the version of Monte.

B.2.2. Directives

```
4  Directive ::= WorldDirective | IncludeDirective
```

Following the header is a set of one or more directives. These are instructions intended for the translator parser itself.

[2] zero is a number too.

B.2.2.1. world

```
5  WorldDirective ::= 'world' SPACE WorldSpec
```

The world directive is used to categorise translators into logical groups. For instance, translators for non-MacOS types generally are not used with those for MacOS types. Appendix C lists the currently recommended groupings.

```
6  WorldSpec ::= (LETTER | DIGIT)+ ( SLASH (LETTER | DIGIT)+ )*
                 SLASH?
```

A world is defined using a simple path, as in the examples: "mac" "mac/rsrc" "mac/nib" "qt", etc. Although a trailing slash is allowed, there is no semantic difference between, for example, "mac/rsrc/" and "mac/rsrc".

B.2.2.2. include

```
7  IncludeDirective ::= 'include' SPACE FileName
```

The include directive is used to pull in definitions from other translator files. Unlike the C/C++ #include directive, included files for Monte are read *after* the entire header has been read, but still before the first translator definition. included files are read in the order of the include directives which specified them.

B.3. Comments

B.3.1. Comment Position

```
8  NewLine ::= (SPACE? Comment? SPACE? LINEFEED)+
```

Comments are allowed only where a new-line can occur. This is different from C/C++ which allows comments anywhere spaces can occur.

B.3.1.1. Comment Types

```
9  Comment ::= SingleLineComment | MultiLineComment
```

Two kinds of comment are allowed in Monte: a single-line comment and a multi-line comment.

```
10 SingleLineComment ::= '//' CHARACTER*
11 MultiLineComment ::= '/*' ([^*]|*[^/])* '*/'
```

A side-effect of the rule that comments are only allowed where new-lines can occur is that multi-line comments in Monte translators must end on a line on their own – no instruction data can follow on the same line as the end of a multi-line comment. For example, the following is invalid:

```
translator "wxyz"/* Multi-line comment here!
                This is the wxyz translator
end of comment*/ sint:16       //instruction is invalid here!
```

…the correct arrangement is:

```
translator "wxyz"/* Multi-line comment here!
                This is the wxyz translator
end of comment*/
   sint:16        //instruction now valid, as it's on its own line
```

B.4. Translator Definitions

```
12  Translator ::= ('export' SPACE)? 'translator' SPACE
                        TranslatorName (SPACE? DerivativeSpec)? NewLine
                        ( Instruction NewLine )*
```

B.4.1. Exported and Private Translators

If the definition of the translator is preceded by the keyword 'export', then the translator is externally visible. Translators wihout the 'export' keyword will not be immediately visible to users of Monte (although it will still be possible to select them if required). This mechanism is required because of the potentially large number of translators which can be used to describe a structure – having two categories of translator (export and non-export) avoids cluttering the user-visible translator set with all of these specialised translators. Both categories of translator (exported and private) live in the same namespace.

B.4.2. Translator Names

```
13  TranslatorName ::= QuotedString
```

Translator names are quoted strings rather than Element names in order to allow legacy data type names (from MacOS resource types) which may contain symbols (such as spaces) not permitted in Element names.

B.4.3. Specifying Derivation

```
14  DerivativeSpec ::= ':' SPACE? TranslatorName
                              ( SPACE '(' SPACE? Rules SPACE? ')' )?
```

A translator is specified as being a derivative of another by use of a single colon followed by the name of the translator from which it the translator is derived. Currently, only one parent type is allowed (single inheritance only).

Rules, if required, are specified after the parent type, and enclosed in parentheses. For example:

```
translator "Icon" : "DITLItem" ( {type}==32 )
```

B.4.4. Rules

```
15  Rules ::= EqualityRule
```

Currently, only one rule per derivative is allowed. Also, only equality rules are implemented in Monte.

B.4.4.1. Equality Rules

```
16  EqualityRule ::= ElementValueRef SPACE? '==' SPACE? Literal
```

Equality rules are specified as a reference to the value of an element, a double equals sign (as in C/C++) and a literal value which that element value must match.

B.4.4.2. Element Value References

```
17  ElementValueRef ::= '{' ElementName '}'
```

An reference to the value of an element (as opposed to the element itself) is indicated by enclosing the element's name in braces (curly brackets). Value references are only resolved when data is being decomposed.

B.5. Instructions

```
18  Instruction ::=     UIntInstruction | SIntInstruction |
                        SkipInstruction |
                        FillInstruction | AlignInstruction |
                        ListCountInstruction |
                        ZeroListCountInstruction
                        OffsetInstruction | LocationInstruction |
                        SizeofInstruction | PStringInstruction |
                        CStringInstruction | StringInstruction |
                        ListInstruction | RepeatInstruction |
                        InsertInstruction | SubInstruction |
                        PropertyInstruction
```

B.5.1. Data Instructions

```
19  UIntInstruction ::= (Presentation SPACE)? 'uint' ISizeSpec?
                            (SPACE ElementNameDef)?
                            (SPACE IConstValueSpec)?
20  SIntInstruction ::= (Presentation SPACE)? 'sint' ISizeSpec?
                            (SPACE ElementNameDef)?
                            (SPACE IConstValueSpec)?
```

…For `uint` and `sint`, a size specifier is optional: these commands default to 16 bits. If a presentation keyword is omitted, the data presentation defaults decimal.

B.5.2. Skip, Fill, Align

```
22  SkipInstruction ::= 'skip' ISizeSpec? (SPACE ElementNameDef )?
23  FillInstruction ::= 'fill' ISizeSpec (SPACE ElementNameDef )?
24  AlignInstruction ::= 'align' AlignSizeSpec
                                (SPACE ElementNameDef)?
```

…unlike `sint`, `uint` or `char`, the `fill` and `align` instructions **must** take a size specifier.However, `skip` is not required to take an explicit size specifer: if the size of a `skip` command is omitted, Monte will skip to the end of the current container (or end of the input data if the end of the current container cannot accurately be determined).

B.5.3. Element References

```
25  ListCountInstruction ::= Presentation? 'listcount' (ISizeSpec)?
                        '(' SPACE? ElementName SPACE? ')'
                        (SPACE ElementNameDef)?
25b ZeroListCountInstruction ::= Presentation? 'zerolistcount'
                        (ISizeSpec)? '(' SPACE? ElementName SPACE? ')'
                        (SPACE ElementNameDef)?
26  OffsetInstruction ::= Presentation? 'offset' ISizeSpec?
                        '(' SPACE? ElementName SPACE? ')'
                        (SPACE ElementNameDef)?
27  LocationInstruction ::= Presentation? 'location' ISizeSpec?
                        '(' SPACE? ElementName SPACE? ')'
                        (SPACE ElementNameDef)?
28  SizeofInstruction ::= Presentation? 'sizeof' ISizeSpec?
                        ( '(' SPACE? ElementName SPACE? ')' )?
                        (SPACE ElementNameDef)?
```

…the `listcount`, `offset` and `location` instructions all take as an argument the name of another element surrounded by parentheses. Note that this argument is optional for `sizeof` (in which case the target of the command is the container containing the sizeof command).

These productions are illustrated in the examples below:

```
listcount:16( items ) itemCount // "1" would mean 1 item.
zerolistcount:16( items ) itemCount // "0" would mean 1 item.
offset( noteList ) notesOffset       // default size is 32
sizeof:16( note ) noteSize
location( note ) thisNote            // default size is 32
sizeof:32 containerSize     // size of this and its siblings
```

B.5.4. Strings

```
29  PStringInstruction  ::=  'pstr'  (ISizeSpec  CSizeSpec)?
                              ('['  SPACE?  PositiveInteger  SPACE?  ']')
                              (SPACE  ElementNameDef)?
                              (SPACE  SConstValueSpec)?
30  CStringInstruction  ::=  'cstr'  CSizeSpec?
                              ('['  SPACE?  PositiveInteger  SPACE?  ']')
                              (SPACE  ElementNameDef)?
                              (SPACE  SConstValueSpec)?
31  StringInstruction  ::=  'string'  (CSizeSpec)?
                              ('['  SPACE?  PositiveInteger  SPACE?  ']')?
                              (SPACE  ElementNameDef)?
                              (SPACE  SConstValueSpec)?
```

The size of pstr and cstr data can be fixed using an optional character count, specified inside square brackets. This informs Monte to always read that number of characters for the string, regardless of the length of the string. For example:

```
translator  "ftag"
   uint:8  ftagVersion  =  2;
   uint:8   fileMajorVersion
   uint:8   fileMinorVersion
   uint:8   developmentStage
   uint:8   release
   pstr:8:8[31]  filename    // will always occupy 32 bytes
                             // (one for length, 31 for string data)
   fill:8
   uint  candidate/"Release  candidate  number"
```

The `string` instruction also takes an optional character count, inside square brackets, but the meaning differs slightly. If this count is present, it represents the fixed number of characters in the string; if absent, the string is of indeterminate length, and continues to the end of the input data stream (or to the end of the current container if the size of this can be accurately determined):

```
translator  "HighScoreRecord"
   string:8[3]  initials    // string is always 3 characters long
   fill:8
   uint:32    score
```

```
translator  "TEXT"
    string:8  theText
        // theText continues to end of data stream


translator  "STR "
    sizeof:8        // refers to size of whole data block
    string:8  theText
```

B.5.5. Lists

```
32  ListInstruction  ::=  'list' '(' SPACE? TranslatorName SPACE? ')'
                          (SPACE  ElementNameDef)?
33  RepeatInstruction  ::=  'repeat' '[' SPACE? PositiveInteger SPACE?
                          ']' SPACE? '(' TranslatorName SPACE? ')'
                          (SPACE?  ElementNameDef)?
```

The `repeat` instruction takes a repetition count as an argument: the structure described by TranslatorName will be repeated that number of times. Examples of list and repeat:

```
repeat[16]("PaletteEntry")  // 16 entries described by
PaletteEntry
list("DialogItem")  theItems/"Dialog  item  list"
        // n items described by DialogItem
```

B.5.6. Substructures

```
34  InsertInstruction  ::=  'insert' '(' SPACE? TranslatorName SPACE?
    ')'
35  SubInstruction  ::=  'sub' '(' TranslatorName SPACE? ')'
                          (SPACE?  ElementNameDef)?
```

Note that `insert` instructions do not take an element name specifier. This is simply because the insert instruction doesn't represent anything which can be named: it simply causes the data read by the named translator to be "dumped" into the current data container.

Examples:

```
insert("QDRect")        // Quickdraw rectangle macro
sub( PPobject ) theObject    // place a PPobject in a container
                             // called "theObject"
```

B.5.7. Properties

```
36  PropertyInstruction ::= 'prop' SPACE PropertyName SPACE? '='
                                SPACE? (ElementName | QuotedString )
```

Properties may be assigned using an element name (for modifiable properties) or using a quoted string (for non-modifiable properties).

Property names and element names live in the same namespace: thus it is not possible to use the same name to describe a property and an element.

B.5.9. Presentation keywords

```
37  Presentation ::= 'oct' | 'bin' | 'dec' | 'hex' | 'packed'
```

Monte allows data elements to be presented as octal (`oct`), binary (`bin`), decimal (`dec`), hexadecimal (`hex`) or as a packed-character array (`packed`). Where no presentation is specified, decimal is assumed.

B.5.10 Data Size fields

```
38  CSizeSpec ::= ISizeSpec
39  AlignSizeSpec ::= ISizeSpec ISizeSpec?
40  ISizeSpec ::= ':' PositiveInteger
```

Data sizes are specified using a colon followed by a number. Note that no spaces are permitted in size specifications.

B.5.11. Constant Value Specifications

```
41  IConstValueSpec ::= '=' SPACE? Integer | PackedCharArray
42  SConstValueSpec ::= '=' SPACE? QuotedString
```

For those elements which are allowed to take a constant value, this is specified using a single equals sign and a literal value (appropriate to the element's type). For instance:

```
translator Control : DITLItem ( {type}==7 )
        sint:8 = 2      // size of following data ...
```

For named elements, the constant value specifier comes after the name:

```
        sint:8 label=21     // comment
```

For named elements with descriptions, the constant value specifier comes after the description:

```
        sint:8 label/"signed byte"=21        // comment
```

B.6. Literals

Literals are used for rules (§B.4.4), for property definitions, and for specifying constant values for elements (§B.5.1.2).

```
43   Literal ::= Integer | QuotedString | PackedCharArray
44   Integer ::= (MINUS)? PositiveInteger | HexInteger | OctInteger
                        | BinInteger
45   PositiveInteger ::= DIGIT DIGIT*
45a  HexInteger ::= '$' HEXDIGIT HEXDIGIT*
45b  OctInteger ::= '\' OCTDIGIT OCTDIGIT*
45c  BinInteger ::= '%' BINDIGIT BINDIGIT*
46   QuotedString ::= '"' (CHARACTER - ["\] | '""' | '\' |
                        '\' HEXDIGIT HEXDIGIT HEXDIGIT HEXDIGIT)* '"'
47   PackedCharArray ::= ''' (CHARACTER - ['\] | '''' | '\\' |
                        '\' HEXDIGIT HEXDIGIT)+ '''
```

Quoted string literals are delimited by double quote (", U+0022) characters. Within a string the character '"' can be obtained by doubling-up the quote character, as in:

```
pstr:8:8 = " Just to say ""Hi!"""
```

Carriage returns, tabs and other non-printing characters are not permitted directly within strings, but may be specified using a backslash (\, U+005C) followed by a four-digit hexadecimal code representing the character's Unicode code point:

```
pstr:8:8 return = "\000d"
```

Should a backslash itself be required, it can be obtained by doubling-up the backslash character:

```
pstr:8:8 hint = "Carriage return = \\000d"
```

Packed character arrays can be specified in a similar manner to strings, but using single quotes as delimiters. For example:

```
packed uint:32 resType = 'PICT'
packed uint:32 token = '''hi'''      // value is 'hi'
packed sint:16 crlf = '\0D\0a':8
```

Unlike strings, packed character arrays are limited to being composed of 8 bit wide characters. Also, the text encoding used for packed character arrays on the Macintosh platform is *always* MacOS Roman (West European).

Where a character in a packed character is specified in hexadecimal (eg. "`\000D`" above), the character codes are *not* from the Unicode/ISO10646 character set, but rather from the basic character set of the platform (Mac Roman for the Macintosh platform).

By convention, Monte places the last character in a packed character array sequence into the lowest-order word of the data element in question (thus, `crlf` above would have the hexadecimal value `0d0a`).

note In a change from normal behaviour, packed character arrays are converted to integers by assuming that the characters within them are encoded in the platform's most basic encoding. For MacOS, this means that character values are taken to be in the MacOS Roman (West European) encoding.

note In nearly all cases, `string:8[4] = "abcd"` is a better choice for textual data than `uint:32 = 'abcd'`, as the former allows editing of the value to include non-Roman characters (`uint` values can only be presented).

Packed character arrays are provided for the convenience of translator writers who have to deal with the class and resource IDs which are commonplace in Mac Classic and Carbon object formats. Packed character arrays should *never* be used for textual data.

B.7. Names

B.7.1. Labels

```
48  PropertyName  ::=  ElementName
49  ElementNameDef  ::=  ElementName
                              (SPACE?  SLASH  SPACE?  ElementDescription)?
49a ElementDescription  ::=  QuotedString
50  ElementName  ::=  (LETTER)(LETTER  |  DIGIT  |  [_.]  |
                                      COMBININGCHAR  |  EXTENDERCHAR  )*
```

Names of elements must begin with a letter character. After this initial character there may be one or more digits, letters, combining or extending characters. The underscore and full-stop (period) characters are also allowed in names, but only as a second or subsequent character. Quoted strings are not permissable as element names.

Where an element name is being defined (rather than just used), an additional Element Description may be specified by following the element name with a slash character and the descriptive text (a quoted string).

Name resolution is **case sensitive**: for example `DITL` and `DitL` are not the same symbol.

B.7.2. File Names

```
51  FileName  ::=  QuotedString
```

Filename validation is not performed by the Monte translator parser. The use of relative paths is forbidden in filename specifiers, but this prohibition is enforced by Monte using platform-dependent checks rather than by a stricter filename specification.

B.8. Character Types

B.8.1. Whitespace

```
52  SPACE  ::=  (#x0020  |#x0009)+
```

Whitespace is defined as any sequence or any number of TAB (U+0009) or space (U+0020) characters. Note that newlines are not permitted as spaces.

```
53  LINEFEED  ::=  (#x000D  |#x000A  )+
```

Monte accepts both CR and LF, or any combination thereof, as a line-feed.

B.8.2. General Character

```
54  CHARACTER  ::=  [#x20-#xD7FF]  |[#xE000-#xFFFD]
```

Monte defines a character as any Unicode code, excluding legacy control characters (U+0000 to U+001F), surrogate pairs (U+D800 to U+DFFF) and the reserved code points U+FFFE and U+FFFF.

B.8.3. Letters

```
55  LETTER[3]  ::= BASECHAR | (BASECHAR COMBININGCHAR) | IDEOGRAPH
```

A letter is defined as any of the Unicode base characters or ideographics. Thus, Monte translators may contain labels or names in non-Roman scripts.

```
56  BASECHAR ::= [#x0041-#x005A] |□[#x0061-#x007A] |□[#x00C0-#x00D6]
        |□[#x00D8-#x00F6] |□[#x00F8-#x00FF] |□[#x0100-#x0131]
        |□[#x0134-#x013E] |□[#x0141-#x0148] |□[#x014A-#x017E]
        |□[#x0180-#x01C3] |□[#x01CD-#x01F0] |□[#x01F4-#x01F5]
        |□[#x01FA-#x0217] |□[#x0250-#x02A8] |□[#x02BB-#x02C1] |□#x0386
        |□[#x0388-#x038A] |□#x038C |□[#x038E-#x03A1] |□[#x03A3-#x03CE]
        |□[#x03D0-#x03D6] |□#x03DA |□#x03DC |□#x03DE |□#x03E0
        |□[#x03E2-#x03F3] |□[#x0401-#x040C] |□[#x040E-#x044F]
        |□[#x0451-#x045C] |□[#x045E-#x0481] |□[#x0490-#x04C4]
        |□[#x04C7-#x04C8] |□[#x04CB-#x04CC] |□[#x04D0-#x04EB]
        |□[#x04EE-#x04F5] |□[#x04F8-#x04F9] |□[#x0531-#x0556] |□#x0559
        |□[#x0561-#x0586] |□[#x05D0-#x05EA] |□[#x05F0-#x05F2]
        |□[#x0621-#x063A] |□[#x0641-#x064A] |□[#x0671-#x06B7]
        |□[#x06BA-#x06BE] |□[#x06C0-#x06CE] |□[#x06D0-#x06D3] |□#x06D5
        |□[#x06E5-#x06E6] |□[#x0905-#x0939] |□#x093D |□[#x0958-#x0961]
        |□[#x0985-#x098C] |□[#x098F-#x0990] |□[#x0993-#x09A8]
        |□[#x09AA-#x09B0] |□#x09B2 |□[#x09B6-#x09B9] |□[#x09DC-#x09DD]
        |□[#x09DF-#x09E1] |□[#x09F0-#x09F1] |□[#x0A05-#x0A0A]
        |□[#x0A0F-#x0A10] |□[#x0A13-#x0A28] |□[#x0A2A-#x0A30]
        |□[#x0A32-#x0A33] |□[#x0A35-#x0A36] |□[#x0A38-#x0A39]
        |□[#x0A59-#x0A5C] |□#x0A5E |□[#x0A72-#x0A74] |□[#x0A85-#x0A8B]
        |□#x0A8D |□[#x0A8F-#x0A91] |□[#x0A93-#x0AA8] |□[#x0AAA-#x0AB0]
        |□[#x0AB2-#x0AB3] |□[#x0AB5-#x0AB9] |□#x0ABD |□#x0AE0
        |□[#x0B05-#x0B0C] |□[#x0B0F-#x0B10] |□[#x0B13-#x0B28]
        |□[#x0B2A-#x0B30] |□[#x0B32-#x0B33] |□[#x0B36-#x0B39] |□#x0B3D
        |□[#x0B5C-#x0B5D] |□[#x0B5F-#x0B61] |□[#x0B85-#x0B8A]
        |□[#x0B8E-#x0B90] |□[#x0B92-#x0B95] |□[#x0B99-#x0B9A] |□#x0B9C
        |□[#x0B9E-#x0B9F] |□[#x0BA3-#x0BA4] |□[#x0BA8-#x0BAA]
        |□[#x0BAE-#x0BB5] |□[#x0BB7-#x0BB9] |□[#x0C05-#x0C0C]
        |□[#x0C0E-#x0C10] |□[#x0C12-#x0C28] |□[#x0C2A-#x0C33]
        |□[#x0C35-#x0C39] |□[#x0C60-#x0C61] |□[#x0C85-#x0C8C]
        |□[#x0C8E-#x0C90] |□[#x0C92-#x0CA8] |□[#x0CAA-#x0CB3]
        |□[#x0CB5-#x0CB9] |□#x0CDE |□[#x0CE0-#x0CE1] |□[#x0D05-#x0D0C]
        |□[#x0D0E-#x0D10] |□[#x0D12-#x0D28] |□[#x0D2A-#x0D39]
        |□[#x0D60-#x0D61] |□[#x0E01-#x0E2E] |□#x0E30 |□[#x0E32-#x0E33]
        |□[#x0E40-#x0E45] |□[#x0E81-#x0E82] |□#x0E84 |□[#x0E87-#x0E88]
        |□#x0E8A |□#x0E8D |□[#x0E94-#x0E97] |□[#x0E99-#x0E9F]
        |□[#x0EA1-#x0EA3] |□#x0EA5 |□#x0EA7 |□[#x0EAA-#x0EAB]
        |□[#x0EAD-#x0EAE] |□#x0EB0 |□[#x0EB2-#x0EB3] |□#x0EBD
```

[3] These definitions are modified versions of those in Appendix B of the XML specification.

|□[#x0EC0-#x0EC4] |□[#x0F40-#x0F47] |□[#x0F49-#x0F69]
|□[#x10A0-#x10C5] |□[#x10D0-#x10F6] |□#x1100 |□[#x1102-#x1103]
|□[#x1105-#x1107] |□#x1109 |□[#x110B-#x110C] |□[#x110E-#x1112]
|□#x113C |□#x113E |□#x1140 |□#x114C |□#x114E |□#x1150
|□[#x1154-#x1155] |□#x1159 |□[#x115F-#x1161] |□#x1163 |□#x1165
|□#x1167 |□#x1169 |□[#x116D-#x116E] |□[#x1172-#x1173] |□#x1175
|□#x119E |□#x11A8 |□#x11AB |□[#x11AE-#x11AF] |□[#x11B7-#x11B8]
|□#x11BA |□[#x11BC-#x11C2] |□#x11EB |□#x11F0 |□#x11F9
|□[#x1E00-#x1E9B] |□[#x1EA0-#x1EF9] |□[#x1F00-#x1F15]
|□[#x1F18-#x1F1D] |□[#x1F20-#x1F45] |□[#x1F48-#x1F4D]
|□[#x1F50-#x1F57] |□#x1F59 |□#x1F5B |□#x1F5D |□[#x1F5F-#x1F7D]
|□[#x1F80-#x1FB4] |□[#x1FB6-#x1FBC] |□#x1FBE |□[#x1FC2-#x1FC4]
|□[#x1FC6-#x1FCC] |□[#x1FD0-#x1FD3] |□[#x1FD6-#x1FDB]
|□[#x1FE0-#x1FEC] |□[#x1FF2-#x1FF4] |□[#x1FF6-#x1FFC] |□#x2126
|□[#x212A-#x212B] |□#x212E |□[#x2180-#x2182] |□[#x3041-#x3094]
|□[#x30A1-#x30FA] |□[#x3105-#x312C] |□[#xAC00-#xD7A3]

57 IDEOGRAPH::= [#x4E00-#x9FA5] |□#x3007 |□[#x3021-#x3029]

58 COMBININGCHAR ::= [#x0300-#x0345] |□[#x0360-#x0361] |□[#x0483-
#x0486] |□[#x0591-#x05A1] |□[#x05A3-#x05B9] |□[#x05BB-#x05BD]
|□#x05BF |□[#x05C1-#x05C2] |□#x05C4 |□[#x064B-#x0652] |□#x0670
|□[#x06D6-#x06DC] |□[#x06DD-#x06DF] |□[#x06E0-#x06E4]
|□[#x06E7-#x06E8] |□[#x06EA-#x06ED] |□[#x0901-#x0903] |□#x093C
|□[#x093E-#x094C] |□#x094D |□[#x0951-#x0954] |□[#x0962-#x0963]
|□[#x0981-#x0983] |□#x09BC |□#x09BE |□#x09BF |□[#x09C0-#x09C4]
|□[#x09C7-#x09C8] |□[#x09CB-#x09CD] |□#x09D7 |□[#x09E2-#x09E3]
|□#x0A02 |□#x0A3C |□#x0A3E |□#x0A3F |□[#x0A40-#x0A42]
|□[#x0A47-#x0A48] |□[#x0A4B-#x0A4D] |□[#x0A70-#x0A71]
|□[#x0A81-#x0A83] |□#x0ABC |□[#x0ABE-#x0AC5] |□[#x0AC7-#x0AC9]
|□[#x0ACB-#x0ACD] |□[#x0B01-#x0B03] |□#x0B3C |□[#x0B3E-#x0B43]
|□[#x0B47-#x0B48] |□[#x0B4B-#x0B4D] |□[#x0B56-#x0B57]
|□[#x0B82-#x0B83] |□[#x0BBE-#x0BC2] |□[#x0BC6-#x0BC8]
|□[#x0BCA-#x0BCD] |□#x0BD7 |□[#x0C01-#x0C03] |□[#x0C3E-#x0C44]
|□[#x0C46-#x0C48] |□[#x0C4A-#x0C4D] |□[#x0C55-#x0C56]
|□[#x0C82-#x0C83] |□[#x0CBE-#x0CC4] |□[#x0CC6-#x0CC8]
|□[#x0CCA-#x0CCD] |□[#x0CD5-#x0CD6] |□[#x0D02-#x0D03]
|□[#x0D3E-#x0D43] |□[#x0D46-#x0D48] |□[#x0D4A-#x0D4D] |□#x0D57
|□#x0E31 |□[#x0E34-#x0E3A] |□[#x0E47-#x0E4E] |□#x0EB1
|□[#x0EB4-#x0EB9] |□[#x0EBB-#x0EBC] |□[#x0EC8-#x0ECD]
|□[#x0F18-#x0F19] |□#x0F35 |□#x0F37 |□#x0F39 |□#x0F3E |□#x0F3F
|□[#x0F71-#x0F84] |□[#x0F86-#x0F8B] |□[#x0F90-#x0F95] |□#x0F97
|□[#x0F99-#x0FAD] |□[#x0FB1-#x0FB7] |□#x0FB9 |□[#x20D0-#x20DC]
|□#x20E1 |□[#x302A-#x302F] |□#x3099 |□#x309A

```
59  EXTENDERCHAR ::= #x00B7 |□#x02D0 |□#x02D1 |□#x0387 |□#x0640 |□#x0E46
          |□#x0EC6 |□#x3005 |□[#x3031-#x3035] |□[#x309D-#x309E]
          |□[#x30FC-#x30FE]
```

B.8.3. Digits

```
60  DIGIT ::= [#x0030-#x0039]
```

Monte supports decimal digits from the Arabic-Indic (ie., “European”) set only.

```
61  MINUS ::= #x002d
```

The minus character is defined as character U+002d (‘-’)

```
62  SLASH ::= #x002f
```

The forward-slash character as used in world defintions (§B.2.2.1) and Element Description specifications (§B.7.1) is U+0002f (‘/’)

Digits used for hexadecimal, octal or binary numbers are as follows:

```
63  HEXDIGIT ::= DIGIT | [A-F] | [a-f]
64  OCTDIGIT ::= [#x0030-#x0037]
65  BINDIGIT ::= #x0030 | #x0031
```

Appendix C. Standard Names

This appendix list the recommended names for properties and translator world definitions. These names are *not* keywords or reserved words in the Monte language, they are simply suggestions for translator writers.

C.1. Properties

C.1.1. Item Bounds

Item boundaries should always be expressed *in pixels*, according to the MacOS Quickdraw co-ordinate convention (figure C.1).

Name	*Data type*	*Description*
`Bounds.x1`	numeric	Left edge of item bounding box (see figure C.1)
`Bounds.y1`	numeric	Top edge of item bounding box (see figure C.1)
`Bounds.x2`	numeric	Right-hand edge of item bounding box (see figure C.1)
`Bounds.y2`	numeric	Bottom edge of item (see figure C.1)
`Bounds.w`	numeric	Width of item (see figure C.1)
`Bounds.h`	numeric	Height item (see figure C.1)

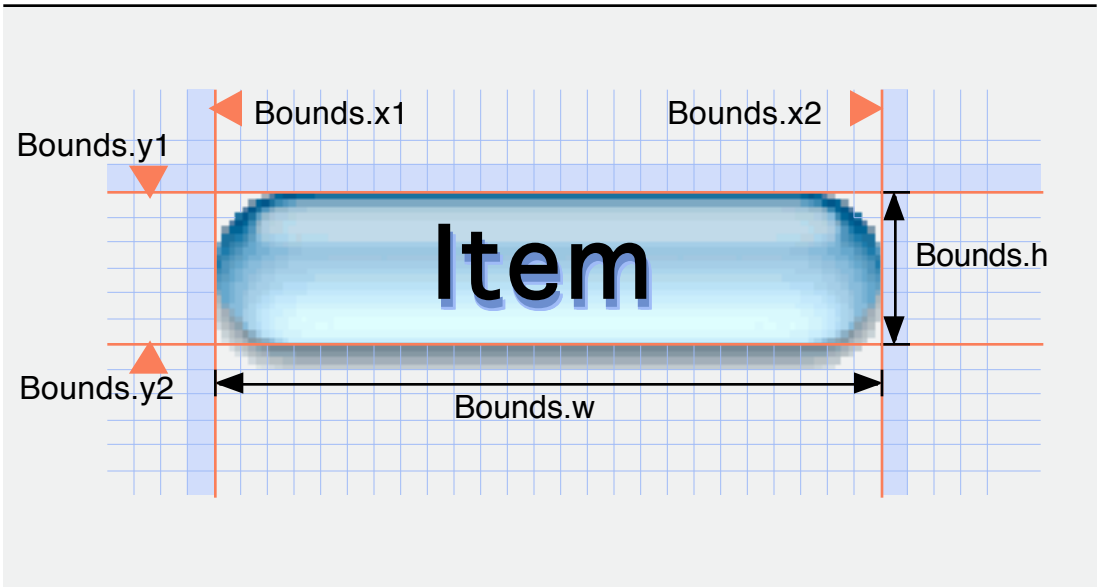

figure C.1. Positional properties and their meanings

Not all translators will define all of these properties, *nor do they have to* – sometimes it will be necessary for applications to look for their preferred property (perhaps "`Bounds.x2`"), and if it is not found, to generate it from a less desirable one ("`Bounds.w`").

C.1.2. Item Indices

The following properties are automatically attached to items in lists by Monte.

Name	*Data type*	*Description*
`Index0`	numeric	Index number of a list item, first item is “0”
`Index1`	numeric	Index number of a list item, first item is “1”

C.1.3. Item Text

Name	*Data type*	*Description*
`LocaleText`	text	User-visible text, potentially localisable. This property should directly point to a textual data element. Note that just because a LocaleText property exists for a structure, this does not mean that the particular item should unconditionally be localised.

C.1.3. Identification

Name	*Data type*	*Description*
`Name`	text	Name of structure

C.1.4. Cross-referencing

Name	*Data type*	*Description*
`ResRef`	text	Used to indicate a reference to, or dependency on, another Classic/Carbon MacOS resource. Should take the form: '*(four character code)*' *(numericID)* (Note the single-quote characters) This property is only meaningful to tools which manipulate Classic/Carbon MacOS resources, and thus should only be defined in MacOS resource translators.

C.2. Translator Groups

Translator authors may organise their translators into any groups they see fit using the `world` directive (§B.2.2.1). However, the groups defined here must be used only for the purposes shown, in order to facilitate easier organisation of translators.

C.2.1. MacOS support

world	*Description*
`mac`	translators for the MacOS platform

C.2.1.1. MacOS Resource Groups

world	*Description*
`mac/rsrc`	translators for MacOS resources
`mac/rsrc/installer`	translators for resources used by Apple's Installer/Updater application.
`mac/rsrc/classic`	translators for Classic-specific resources
`mac/rsrc/carbon`	translators for MacOS Carbon-specific resources
`mac/rsrc/macapp`	translators for custom resources used by Apple Computer's MacApp programming framework
`mac/rsrc/powerplant`	translators for the 'PPob' resource type used by the Metrowerks Powerplant framework

C.2.2. MacOS X Support

world	*Description*
`mac/osx`	translators for data formats used on the MacOS X platform

Currently, no sub-groupings have been defined for MacOS X translators, although this will undoubtedly change.

C.2.2. Other Platforms

Third-party translator authors are free to use any appropriate unassigned world name for their translators which do not fit into the above groupings.