LabVIEW™ Data Storage

Introduction

This Application Note describes the formats in which you can save data. This information is most useful to advanced users, such as those using shared libraries (DLLs) or code interface nodes (CINs) and those using the file I/O functions for reading and writing binary data to files. This application note describes the following concepts:

- How LabVIEW stores data in memory
- How LabVIEW converts binary data for file storage on disk
- Relationship of type descriptors to data storage

How LabVIEW Stores Data in Memory

This section describes how LabVIEW stores data in memory for controls, indicators, wires, and other objects.

Boolean Data

LabVIEW stores Boolean data as 8-bit values. If the value is zero, the Boolean value is FALSE. Any nonzero value represents TRUE.

Numeric Data

For more information, refer to the *Numeric Data Type* topic in the *LabVIEW Help*.

Byte Integer

Byte integer numbers have 8-bit format, signed and unsigned.

Word Integer

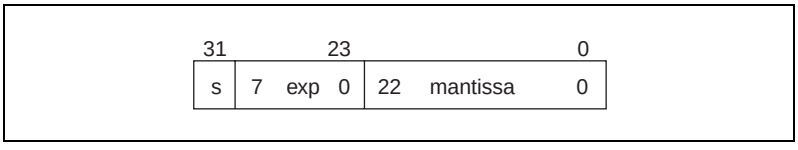
Word integer numbers have 16-bit format, signed and unsigned.

Long Integer

Long integer numbers have 32-bit format, signed and unsigned.

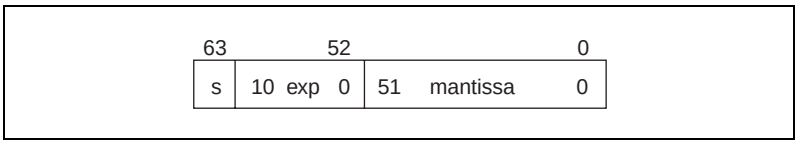
Single

Single-precision floating-point numbers have 32-bit IEEE single-precision format.



Double

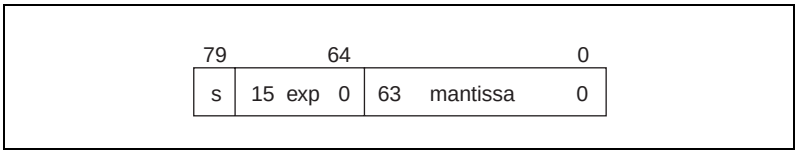
Double-precision floating-point numbers have 64-bit IEEE double-precision format.



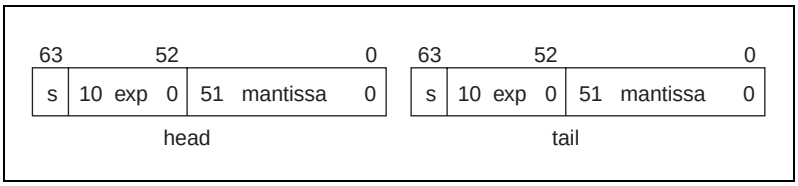
Extended

In memory, the size and precision of extended-precision numbers vary depending on the platform, as described in the following sections:

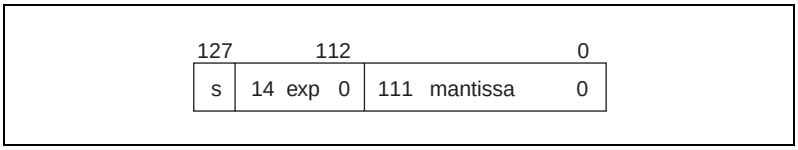
- **Windows and Linux** – Extended-precision floating-point numbers have 80-bit IEEE extended-precision format.



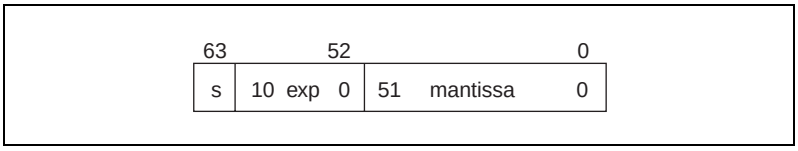
- **Power Macintosh** – Extended-precision floating-point numbers are represented as two double-precision floating-point numbers added together, called the Apple double-double format.



- **Sun** – Extended-precision floating-point numbers have 128-bit IEEE extended-precision format.



- **HP-UX** – Extended-precision floating-point numbers are represented as double-precision floating-point numbers.

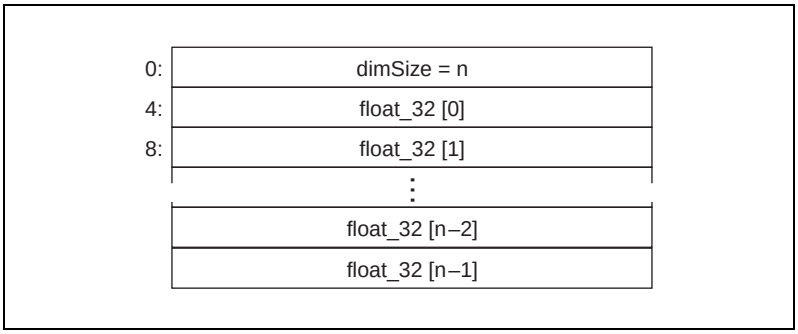


Note For floating point and complex numbers, `s` is the sign bit (0 for positive, 1 for negative), `exp` is the biased exponent (base 2), and `mantissa` is a number in the [0,1] range.

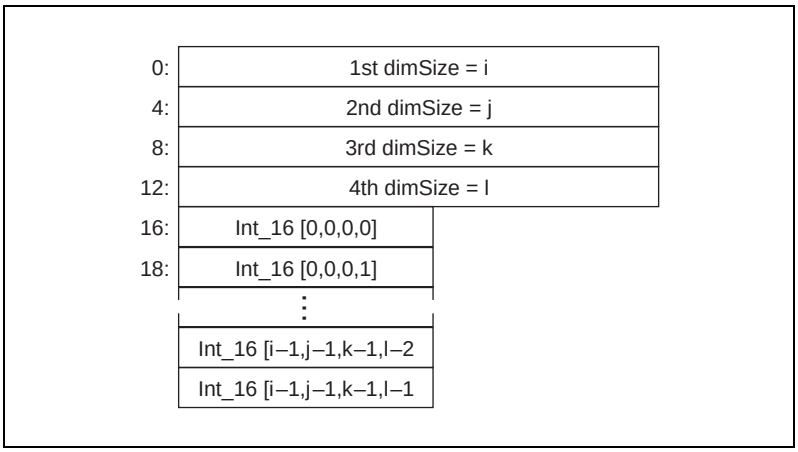
Arrays

LabVIEW stores arrays as handles, or pointers to pointers, containing the size of each dimension of the array in long integers, followed by the data. If your handle is 0, the array is empty. Because of alignment constraints of certain platforms, the dimension size may be followed by a few bytes of padding so that the first element of the data is correctly aligned. If you write DLLs or CINs, refer to the *Alignment Considerations* section of Chapter 2, *CIN Parameter Passing*, of the *Using External Code* manual for more information. This document is available only in PDF format on your LabVIEW CD.

The following illustration shows a 1D array of single-precision floating-point numbers. The decimal numbers to the left represent the byte offsets of locations in memory where the array begins.

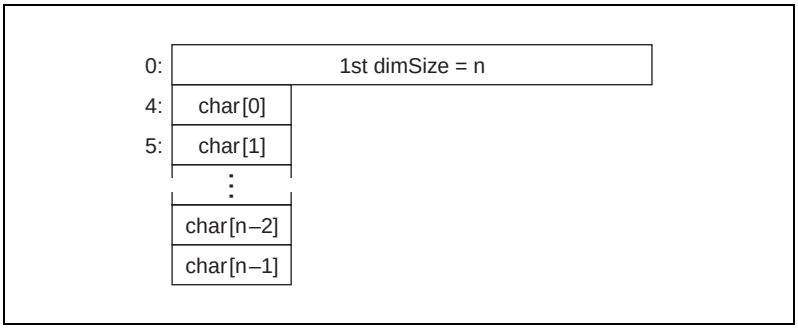


The following illustration shows a 4D array of word integers.



Strings

LabVIEW stores strings as 1D arrays of byte integers (8-bit characters), as shown in the following illustration. If your handle is 0, the array is empty.

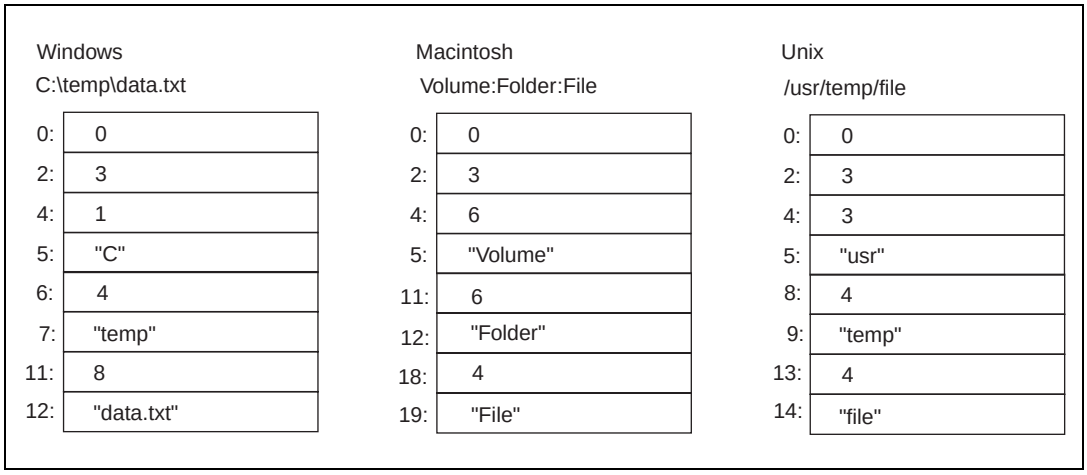


Paths

LabVIEW stores paths as handles, or pointers to pointers, containing the path type and number of path components in word integers, followed by the path components. The path type is 0 for an absolute path, 1 for a relative path, and 3 for a Universal Naming Convention (UNC) path. A UNC path occurs on Windows only and has `\\<machine name>\<share name>` rather than a drive letter as its first component. Any other value of path type

indicates an invalid path. Each path component is a Pascal string (P-string) in which the first byte is the length, in bytes, of the P-string, not including the length byte.

The following illustrations show how LabVIEW stores representative paths for each platform.

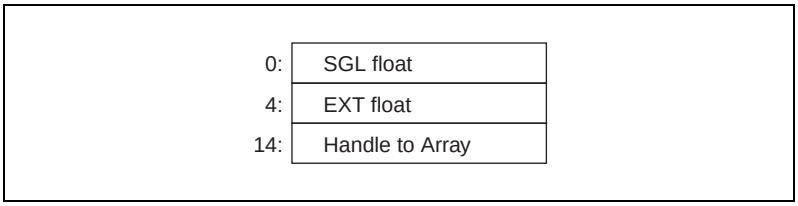


Clusters

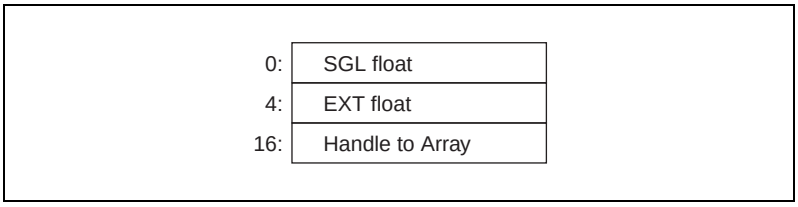
LabVIEW stores cluster elements of varying data types according to the cluster order. To set cluster order, pop up on the cluster border and select **Cluster Order**. LabVIEW stores scalar data directly in the cluster. LabVIEW stores arrays, strings, and paths indirectly. The LabVIEW cluster stores a handle that points to the location in memory where the data is stored. Because of alignment constraints of certain platforms, the dimension size may be followed by a few bytes of padding so that the first element of the data is correctly aligned. If you write DLLs or CINs, refer to the *Alignment Considerations* section of Chapter 2, *CIN Parameter Passing*, of the *Using External Code* manual for more information. This document is available only in PDF format on your LabVIEW CD.

The following illustrations show a cluster that contains a single-precision floating-point number, an extended-precision floating-point number, and a handle to a 1D array of unsigned word integers, respectively.

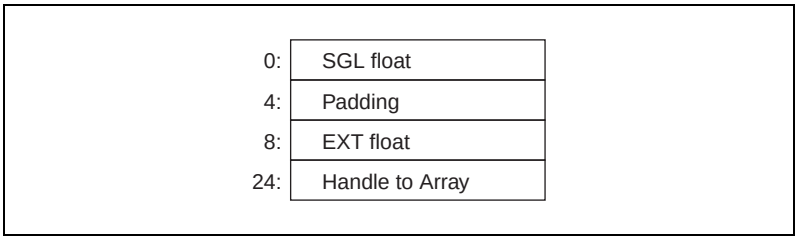
Windows and Linux



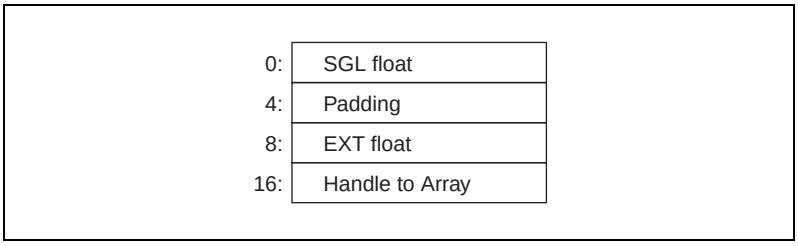
Macintosh



Sun

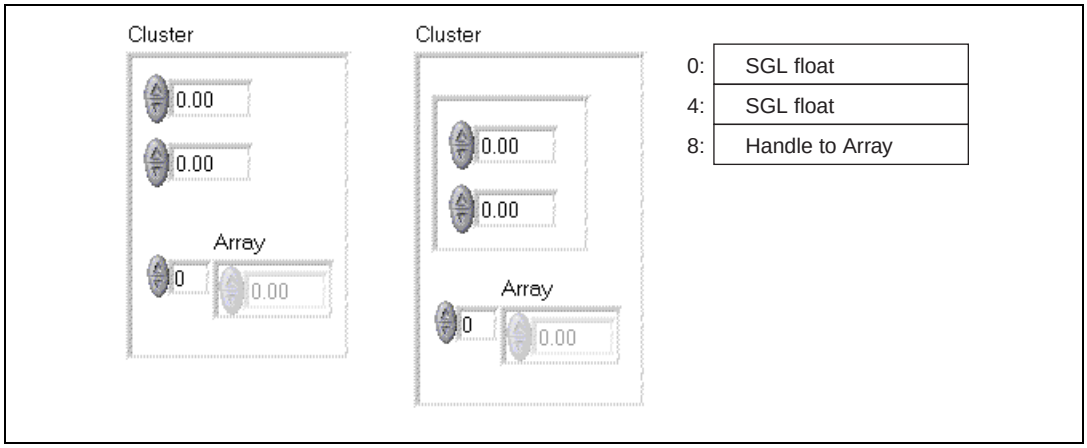


HP-UX



LabVIEW stores embedded clusters directly – meaning that the data is stored as if the data were not embedded in the subcluster. LabVIEW stores only arrays, strings, and paths indirectly.

The following illustration shows two different clusters that store their data the same way.



Flattened Data

LabVIEW converts data from the format in memory to a form more suitable for writing to or reading from a file. This more suitable format is called *flattened* data.

Because LabVIEW stores strings, arrays, and paths in handles (pointers to pointers in separate regions of memory), clusters that contain these strings and arrays are noncontiguous. In general, LabVIEW stores data in tree form. For example, LabVIEW stores a cluster as a double-precision floating-point number and a string as an 8-byte floating-point number, followed by a 4-byte handle to the string. LabVIEW does not store the string data adjacent to the extended-precision floating-point number in memory. Therefore, to write the cluster data to disk, LabVIEW must get the data from two different places. Of course, with a cluster that contains many strings, arrays, and/or paths, LabVIEW stores the data in many different places.

When you save data to a file, LabVIEW *flattens* the data into a single string before saving it. This way, even the data from an arbitrarily complex cluster is made contiguous instead of stored in several pieces. When LabVIEW loads data from a file, it must perform the reverse operation. It must read a single string and *unflatten* it into its internal, possibly noncontiguous, form.

LabVIEW normalizes the flattened data to a standard form so VIs that run on any platform can use the data. LabVIEW stores flattened numeric data in *big endian* form (most-significant byte first), and it stores flattened extended precision floating-point numbers as 16-byte quantities using the Sun extended-precision format described earlier in this application note.

Note When writing data to a file for use by an application not created using LabVIEW or when reading data from a file produced by an application not created using LabVIEW, you can transform your data into *little endian* (least-significant byte first) or *big endian* form after flattening or before unflattening. Windows applications typically expect numeric data to be in *little endian* form.

Use the Flatten to String and Unflatten from String functions, described in the *LabVIEW Help*, to flatten and unflatten data just as LabVIEW does internally when LabVIEW saves and loads data. These functions are in the **Functions»Advanced»Data Manipulation** subpalette.

The flattened form of a piece of data does not encode the type of the data. LabVIEW stores this information in a type descriptor. Refer to *Type Descriptors*, for more information. The Unflatten From String function requires you to wire a data type as an input so the function can decode the string properly.

Use the variant data type to work with data independently of data type instead of flattening the data when you write to memory and unflattening the data when you read from memory. Use the Variant functions, located on the **Functions»Advanced»Data Manipulation»Variant** palette, to create and manipulate variant data. Refer to the Handling Variant Data section of Chapter 5, Building the Block Diagram, of the *LabVIEW User Manual* for more information about using the variant data type.

Booleans and Numerics

The flattened form of any numeric and Boolean type stores the data only in *big endian* format. For example, a long integer with value –19 is flattened to `FFFF FFED`. A double-precision floating-point number with a value equal to `1/4` is flattened to `3FD0 0000 0000 0000`. A Boolean TRUE is any nonzero value. A Boolean FALSE is `00`.

The flattened form for extended-precision numbers is the Sun 128-bit extended-precision floating-point format. When you save extended-precision numbers to disk, LabVIEW stores them in this format.

Strings and Paths

Because strings and paths have variable sizes, a flattened long integer that records their length in bytes precedes the flattened form. For example, a string type with value ABC is flattened to 0000 0003 4142 43. For strings, the flattened format is similar to the format the string takes in memory.

However, paths do not have a length value preceding them when LabVIEW stores them in memory, so this value comes from the actual size of the data in memory and prefixes the value when LabVIEW flattens the data. This length is preceded by four characters: PTH0.

For example, a path with value C:\File is flattened to
5054 4830 0000 000B 0000 0002 0163 0466 696C 65.

5054 4830 indicates PTH0. 0000 000B indicates 11 bytes total. 0000 is the type. 0002 is the number of components. 0163 indicates the letter C as a Pascal string. 0466 696C 65 indicates the word File as a Pascal string.

Arrays

Flattened long integers that record the size, in elements, of each of the dimensions of an array, precede the data for a flattened array. The slowest varying dimension is first, followed successively by the faster varying dimensions, just as the dimension sizes are stored in memory. The flattened data follows immediately after these dimension sizes in the same order in which LabVIEW stores them in memory. The following example shows a 2D array of six 8-bit integers.

{ {1, 2, 3}, {4, 5, 6} } is flattened to 0000 0002 0000 0003 0102 0304 0506.

The following example shows a flattened 1D array of Boolean variables.

{T, F, T, T} is flattened to 0000 0004 0100 0101. The preferred value for TRUE is 01.

Clusters

A flattened cluster is the concatenation, in cluster order, of the flattened data of its elements. For example, a flattened cluster of a word integer of value 4 (decimal) and a long integer of value 12 is 0004 0000 000C.

A flattened cluster of a string ABC and a word integer of value 4 is 0000 0003 4142 4300 04.

A flattened cluster of a word integer of value 7, a cluster of a word integer of value 8, and a word integer of value 9 is 0007 0008 0009.

Type Descriptors

Each wire and terminal in the block diagram is associated with a data type. LabVIEW keeps track of this type with a structure in memory called a *type descriptor.* This type descriptor is a sequence of word integers that can describe any data type in LabVIEW. Numeric values are written in hexadecimal format, unless otherwise noted.

The generic format of a type descriptor is

<length> <type code>

Some type descriptors have additional information following the type code. Arrays and clusters are structured or aggregate data types because they include other types. For example, the cluster type contains additional information about the type of each of its elements.

The first word (16 bits) in any type descriptor is the length, in bytes, of that type descriptor, including the length word. The second word (16 bits) is the type code. LabVIEW reserves the high-order byte of the type code (the *xx* in the

following table) for internal use. When comparing two type descriptors for equality, you should ignore this byte. Two type descriptors are equal even if the high-order bytes of the type codes are not.

The type code encodes the actual type information, such as single-precision or extended-precision floating-point number, as listed in Tables 1 and 2. These type code values might change in future versions of LabVIEW.

Data Types

Tables 1 and 2 list numeric and non-numeric data types, type codes, and type descriptors.

Table 1. Scalar Numeric Data Types

Data Type	Type Code (numbers in hexadecimal)	Type Descriptor (numbers in hexadecimal)
Byte Integer	01	0004 *xx*01
Word Integer	02	0004 *xx*02
Long Integer	03	0004 *xx*03
Unsigned Byte Integer	05	0004 *xx*05
Unsigned Word Integer	06	0004 *xx*06
Unsigned Long Integer	07	0004 *xx*07
Single-Precision Floating-Point Number	09	0004 *xx*09
Double-Precision Floating-Point Number	0A	0004 *xx*0A
Extended-Precision Floating-Point Number	0B	0004 *xx*0B
Single-Precision Complex Floating-Point Number	0C	0004 *xx*0C
Double-Precision Complex Floating-Point Number	0D	0004 *xx*0D
Extended-Precision Complex Floating-Point Number	0E	0004 *xx*0E
Enumerated Byte Integer	15	<nn> *xx*15 <k> <k pstrs>
Enumerated Word Integer	16	<nn> *xx*16 <k> <k pstrs>
Enumerated Long Integer	17	<nn> *xx*17 <k> <k pstrs>
Single-Precision Physical Quantity	19	<nn> *xx*19 <k> <k base-exp>
Double-Precision Physical Quantity	1A	<nn> *xx*1A <k> <k base-exp>
Extended-Precision Physical Quantity	1B	<nn> *xx*1B <k> <k base-exp>
Single-Precision Complex Physical Quantity	1C	<nn> *xx*1C <k> <k base-exp>
Double-Precision Complex Physical Quantity	1D	<nn> *xx*1D <k> <k base-exp>
Extended-Precision Complex Physical Quantity	1E	<nn> *xx*1E <k> <k base-exp>
n=length; x=reserved; k=number; k pstrs=number of Pascal strings; k base-exp=number of base-exponent pairs. Refer to the *Physical Quantity* section of this Application Note for more information.		

Table 2. Non-Numeric Data Types

Data Type	Type Code (numbers in hexadecimal)	Type Descriptor (numbers in hexadecimal)
Boolean	21	0004 *xx*21
String	30	0008 *xx*30 <dim>
Path	32	0008 *xx*32 <dim>
Pict	33	0008 *xx*33 <dim>
Array	40	<nn> xx40 <k> <k dims> <element type descriptor>
Cluster	50	<nn> xx50 <k> <k element type descriptors>
x=reserved; dims=dimensions; k=number; k dims=number of dimensions. Refer to the following sections of this Application Note for more information.		

The minimum value in the size field of a type descriptor is 4, as shown in Table 1. However, any type descriptor can have a name (a Pascal string) appended, in which case the size field is larger by the length of the name rounded up to a multiple of 2.

Enumerated Byte Integer

In the following example of an enumerated byte integer for the items `am`, `fm`, and `fm stereo`, each group of characters represents a 16-bit word. The space enclosed in quotation marks (" ") represents an ASCII space.

```
0016 0015 0003 02a m02 fm 09f m" " st er eo
```

`0016` indicates 22 bytes total. `0015` indicates an enumerated byte integer. `0003` indicates there are three items.

Physical Quantity

In the following example of a double-precision physical quantity with units `m/s`, each group represents a 16-bit word.

```
000E 001A 0002 0002 FFFF 0003 0001
```

`000E` indicates 14 bytes total. `001A` indicates this is a double-precision physical quantity. `0002` indicates two base-exponent pairs. `0002` denotes the seconds base index. `FFFF (-1)` is the exponent of seconds. `0003` denotes the meters base index. `0001` is the exponent of meters.

Note LabVIEW stores all physical quantities internally in terms of base units, regardless of the units used to display them.

Table 3 shows the nine bases that are represented by indexes 0 through 8 for radians through candela.

Table 3. Base Units

Quantity Name	Unit	Abbreviation	Base Value
plane angle	radian	rad	0
solid angle	steradian	Sr	1
time	second	s	2
length	meter	m	3
mass	kilogram	kg	4
electric current	ampere	A	5
thermodynamic temperature	kelvin	K	6
amount of substance	mole	mol	7
luminous intensity	candela	Cd	8

String, Path, and Pict Data Types

The string, path, and pict data types have a 32-bit length, similar to the array dimension size. Although the only value currently encoded is `FFFFFFFF (-1)`, which indicates variable sized. Currently, all strings, paths, and picts are variable sized. The actual length is stored with the data.

Array and Cluster Data Types

Notice the array and cluster data types each have their own type code. They also contain additional information about the data types of their elements and the dimensionality for arrays or number of elements for clusters.

Array

The type code for an array is 40. A word that contains the number of dimensions of the array immediately follows the type code. Then, for each dimension, a long integer contains the size in elements of that dimension. Finally, after each of the dimension sizes, the type descriptor for the element appears. The element type can be any type except an array. Currently all sizes are `FFFFFFFF (–1)`, which means the array dimension size is variable. LabVIEW stores the actual dimension size, which is always greater than or equal to zero, with the data. The following example is a type descriptor for a 1D array of double-precision floating-point numbers:

```
000E 0040 0001 FFFF FFFF 0004 000A
```

`000E` is the length of the entire type descriptor, including the element type descriptor. The array is variable sized, so the dimension size is `FFFFFFFF`. Notice the element type descriptor (`0004 000A`) appears exactly as it does for a scalar of the same type.

The following example is a type descriptor for a 2D array of Boolean values:

```
0012 0040 0002 FFFF FFFF FFFF FFFF 0004 0021
```

Cluster

The type code for a cluster is 50. A word that contains the number of items in the cluster immediately follows the type code. After this word is the type descriptor for each element in *cluster order.* For example, consider a cluster of two integers – a signed-word integer and an unsigned long integer:

```
000E 0050 0002 0004 0002 0004 0007
```

`000E` is the length of the type descriptor including the element type descriptors.

Since array and cluster type descriptors contain other type descriptors, they may become deeply nested. For example, the following is a type descriptor for a multiplot graph. The numeric types can vary.

```
0028 0040 0001 FFFF FFFF...1D array of
    001E 0050 0001...1 component cluster of
        0018 0040 0001 FFFF FFFF...1D array of
            000E 0050 0002...2 component cluster of
                0004 000A...double-precision floating-point number
                    0004 0003...long integer
```

342012A-01 Jun00