

it won't work. The way around this is to rebuild the CIN using a C compiler for the platform that LabVIEW is running on. At press time of this book, the C compilers compatible with LabVIEW are:

Windows 3.1	Watcom C
Windows 95/NT	Microsoft Visual C++, Microsoft Win32 SDK C/C++ compiler
MacOS	THINK C (ver. 5 or above), Symantec C++ (ver. 8 or above), Metrowerks CodeWarrior, MPW from Apple
Solaris	Sun ANSI C compiler
HP-UX	HP-UX C/ANSI C compiler

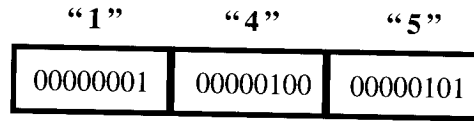
There are many complex issues involved in LabVIEW communicating with external code. Since many of these are highly dependent on the processor, operating system, and compiler you're using, we won't attempt to go any further into discussing CINs or Call Library functions. If you'd like more details, contact National Instruments and request their application notes on this subject, or look at the *Code Interface Reference Manual* included in the LabVIEW manual set.

Fitting Square Pegs into Round Holes: Advanced Conversions and Type- casting

Remember *polymorphism*, discussed early in this book? It's one of LabVIEW's best features, allowing you to mix data types in most functions without even thinking about it (any compiler for a traditional programming language would scream at you if you tried something like adding a constant directly to an array—but not LabVIEW). LabVIEW normally takes care of doing conversions internally when it encounters an input of a different but compatible data type than it expected at a function.

If you're going to develop an application that incorporates instrument control, interapplication communication, or networking, chances are you'll be using mostly string data. Often you'll need to convert your numeric data, such as an array of floating-point numbers, to a string. We talked about this a little in Chapter 9. It's important to distinguish now between two kinds of data strings: *ASCII strings* and *binary strings*.

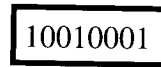
ASCII strings use a separate character to represent each digit in a number. Thus, the number 145, converted to an ASCII string, consists of the characters "1," "4," and "5." For multiple numbers (as in arrays), a delimiter such as the <space> character is also used. This kind of string representation for numbers is very common in GPIB instrument control, for example.



3 bytes

Binary strings are a bit more subtle—for one thing, you can't tell what data they represent just by reading them as ASCII characters. The actual bit pattern (binary representation) of a number is used to represent it. Thus, the number 145 with I8 representation is just a single byte in a binary string (which, incidentally, corresponds on my computer's ASCII set to the “ë” character—not that most humans could tell that means 145). Binary strings are common in applications where a minimal overhead is desired, because generally it is much faster to convert data to binary strings and they take up less memory.

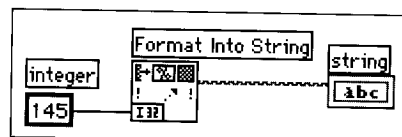
“ë”



1 byte

Ultimately, all data in computers is stored as binary numbers. So how does LabVIEW know if the data is a string, Boolean, double-precision floating-point number, or an array of integers? All data in LabVIEW has two components: the *data* itself, and the data's *type descriptor*. The data type descriptor is an array of I16 integers that constitute code that identifies the representation of the data (integer, string, double-precision, Boolean, etc.). This type descriptor contains information about the length (in bytes) of the data, plus additional information about the type and structure of the data. For a list of data type codes, see Appendix A in the *LabVIEW User's Manual*.

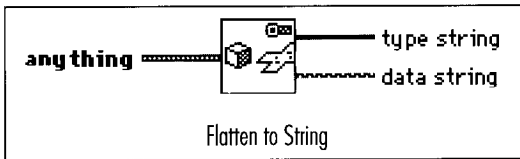
In a typical conversion function, such as changing a number to a decimal string, the type descriptor is changed and the data is modified in some way. The conversion functions you use most of the time convert numbers to ASCII strings, such as in this example.




 Note

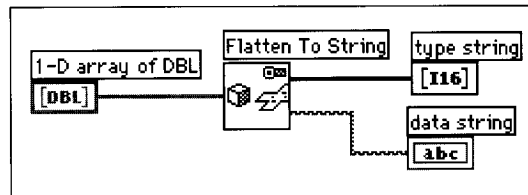
The topics in the rest of this section can be very confusing to beginners. If you do not have any need to use binary string conversions in your application, you can safely skip the rest of this section.

In some cases, you may want to convert data to a *binary string*. Binary strings take up less memory, are sometimes faster to work with, and may be required by your system (such as a TCP/IP command, or an instrument command). LabVIEW provides a way to convert data to binary strings using the **Flatten To String** function. You can access this function from the **Data Manipulation** subpalette under the **Advanced** palette.

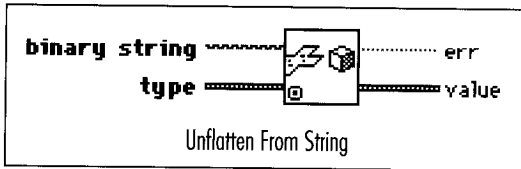


The input to **Flatten To String** can be literally any LabVIEW data type, including complex types such as clusters. The function returns two outputs: the binary **data string** representing the data, and **type string** (which is not a string, but actually an array of **I16**, yet called a string because of nerdy programmer's terminology). The type string gives you the information in the data type descriptor.

A flattened binary string contains not only the data in compact form, but four bytes of *header* information as well. The header information included at the beginning of the binary string contains information about the length, type, structure, etc. of the data. For example, the following diagram shows an array of DBL numeric types flattened to a string. The **type string**, which is always an array of **I16**, contains this header information.

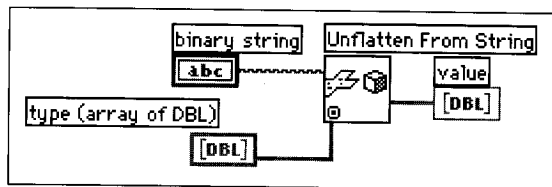


The key to using these confusing binary flattened strings is that most of the time you don't need manipulate or view the strings directly—you just pass these strings to a file, network, instrument, etc., for little overhead. Later, to read the data, you will need to unflatten the string.



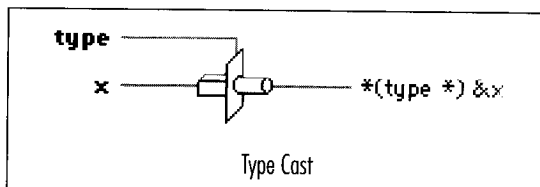
To read back binary strings, use the inverse function, **Unflatten from String**. The **type** input is a dummy LabVIEW data type of the same kind that the **binary string** represents. The **value** output will contain the data in the **binary string**, but will now be of the same data type as **type**. **err** is TRUE if the conversion was unsuccessful.

Here's the example of how we convert back our flattened binary string to an DBL array.



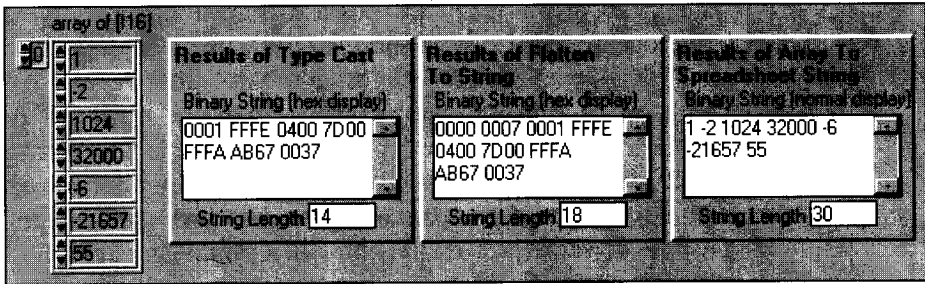
We wire the binary string input, as well as a dummy (empty) 1D array of DBL to specify the data type, and we get back our original array of DBL. Notice how we flattened and unflattened without ever needing to “peek” or “manipulate” the binary strings directly.

For fast, efficient conversions let's take a final look at a very powerful LabVIEW function: **Type Cast**.

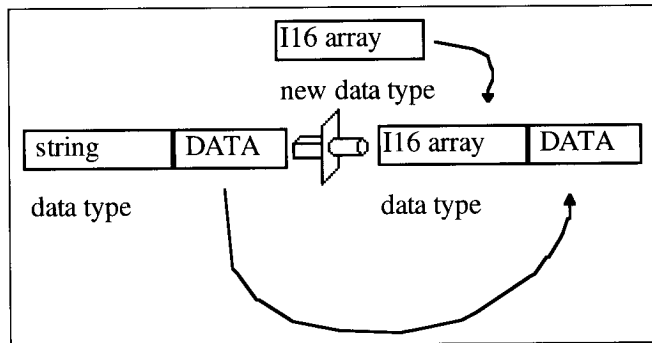
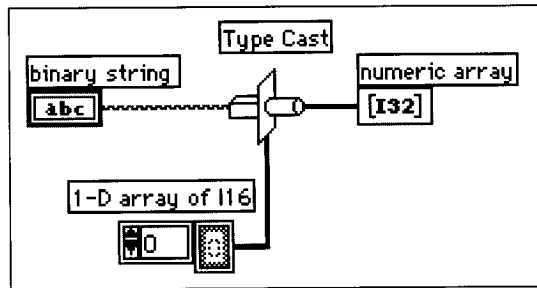


This function allows you to change the data type descriptor, without changing the data at all. There is no conversion of the data itself in any way, just of its type descriptor. You can take almost any type of data (strings, Booleans, numbers, clusters, and arrays), and call it anything else. One advantage of using this function is that it saves memory since it doesn't create another copy of the data in memory, like the other conversion functions do.

The following figure shows the comparative results of converting an array to a string through flattening, typecasting, and numerical formatting.



A common use of *typecasting* is shown in the following figure, where some instrument returns a series of readings as a binary string, and they need to be manipulated as an array of numbers. We are assuming that we know ahead of time that the binary string is representing an array of **I16** integers.



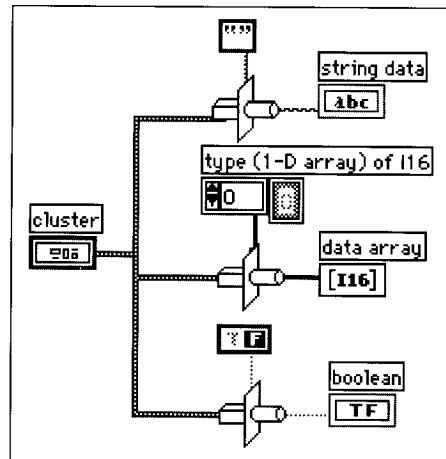
Type casting from scalars or arrays of scalars to a string works like the **Flatten To String** function, but it has the distinct feature that it doesn't add or remove any header information, unlike the four bytes of header strings created by the Flatten and Unflatten functions. The **type** input on the **Type Cast** function is strictly a "dummy" variable used to define the type—any actual data in this variable is ignored.

You need to be very careful with this function, however, because you have to understand exactly how the data is represented so your type casting yields a meaningful result. As mentioned previously, binary strings often contain header information (such as the output from the **Flatten to String** function). **Type Cast** does not add or remove any header information in binary strings. If you don't remove headers yourself, they will be interpreted as data and you'll get garbage as a result.

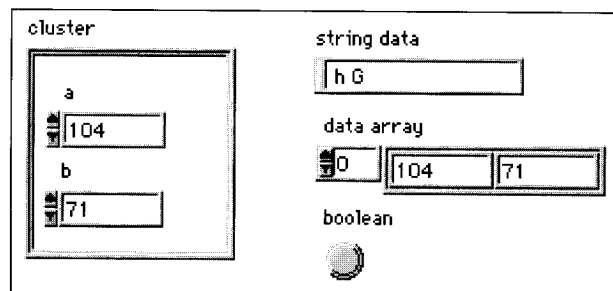


Make sure you understand how data to be type cast is represented. Type casting does not do any kind of conversion or error-checking on the data. Byte ordering (MSB first or last) is platform-dependent, so understand what order your data is in.

You might try experimenting with the **Type Cast** function to see what kind of results you get. To show you how bizarre you can get with typecasting, we took a cluster containing two numeric controls (type **I16**) and cast the data to a string, a numerical array, and a Boolean.



The front panel shows you what results we obtained.



The string returned two characters corresponding to the ASCII values of 104 and 71. The array simply reorganized the numbers into array elements instead of cluster elements. And the Boolean? How would a Boolean variable interpret cluster data? Since we didn't know, we tried this, and found out that the Boolean is TRUE if the numerical value is negative; otherwise it's FALSE. In the case of a cluster, it apparently ORs the interpretation of the cluster elements, so if any element had a negative number, the Boolean interprets it as TRUE. Pretty strange, eh?

Wrap It Up!

In this chapter, we mined LabVIEW's gemstones: local variables, global variables, and attribute nodes. We also looked at some of the advanced functions: occurrences, dialogs, sound, calling external code, binary string conversion, and typecasting. The power and versatility offered by these structures and functions allow you to go a level deeper into programming applications in LabVIEW.

Local variables allow you to create a block diagram "copy" of a front panel object, which you can read from or write to in different places. *Global variables* are similar to locals, but they store their data independently of any particular VI, allowing you to share variables between separate VIs without wiring them together. Locals are very useful for controlling parallel loops or updating a front panel object from multiple locations in the diagram. Globals are a powerful structure, but care is required in their use because they are prone to cause problems.

Attribute nodes give you immense control over the appearance and behavior of indicators and controls, allowing you to change their attributes programmatically. Every control and indicator has a set of base attributes (such as color, visibility, etc.). Many objects, such as graphs, have numerous attributes that you can set or read accordingly.

With LabVIEW, you can use external C routines by using a *CIN (code interface node)*. A CIN lets you call C routines compiled with external compilers. Additionally, you can directly call DLLs (Dynamic Link Libraries) in Windows.

The functions **Flatten to String**, **Unflatten from String**, and **Type Cast** are powerful conversion utilities for working with different data types. These functions allow you to convert LabVIEW data types to and from binary strings. Binary strings are often needed in many applications.