Application Note 168

LabVIEW™ Performance and Memory Management

Introduction

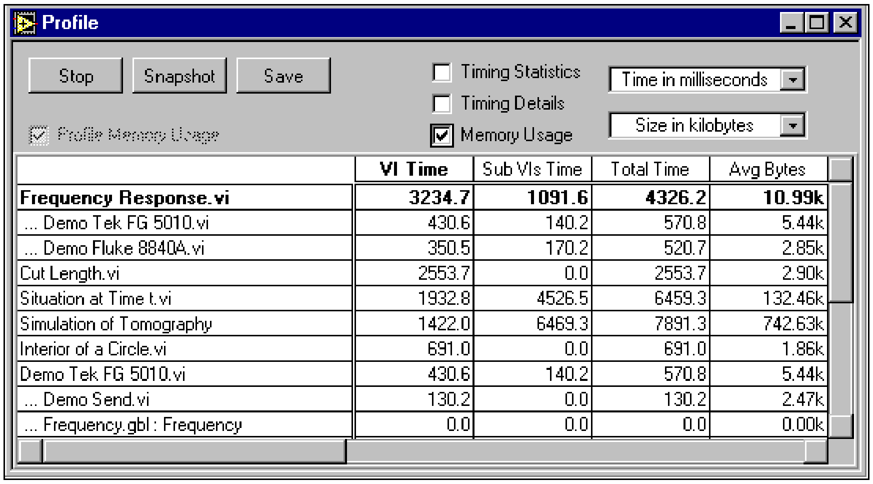
This application note describes the Performance Profiler, a feature that shows you data about execution time of your VIs and monitors single-threaded, multithreaded, and multiprocessor applications. This note also describes factors affecting run-time speed and memory usage.

VI Performance Profiling

The VI Performance Profiler is a powerful tool for determining where your application is spending its time and how it is using memory. The **Profile** window has an interactive tabular display of time and memory usage for each VI in your system. Each row of the table contains information for a specific VI. The time spent by each VI is divided into several categories and summarized. The **Profile** window calculates the minimum, maximum, and average time spent per run of a VI.

You can use the interactive tabular display to view all or part of this information, sort it by different categories, and look at the performance data of subVIs when called from a specific VI.

Select **Tools»Advanced»Profile VIs** to access the **Profile** window. The following illustration shows an example of the window already in use.



☑ Profile Memory Usage

□ Timing Statistics

□ Timing Details

☑ Memory Usage

Time in milliseconds

Size in kilobytes

	VI Time	Sub VIs Time	Total Time	Avg Bytes
Frequency Response.vi	**3234.7**	**1091.6**	**4326.2**	**10.99k**
... Demo Tek FG 5010.vi	430.6	140.2	570.8	5.44k
... Demo Fluke 8840A.vi	350.5	170.2	520.7	2.85k
Cut Length.vi	2553.7	0.0	2553.7	2.90k
Situation at Time t.vi	1932.8	4526.5	6459.3	132.46k
Simulation of Tomography	1422.0	6469.3	7891.3	742.63k
Interior of a Circle.vi	691.0	0.0	691.0	1.86k
Demo Tek FG 5010.vi	430.6	140.2	570.8	5.44k
... Demo Send.vi	130.2	0.0	130.2	2.47k
... Frequency.gbl : Frequency	0.0	0.0	0.0	0.00k

The collection of memory usage information is optional because the collection process can add a significant amount of overhead to the running time of your VIs. You must choose whether to collect this data before starting the Profiler by checking the **Profile Memory Usage** checkbox appropriately. This checkbox cannot be changed once a profiling session is in progress.

The following buttons are available on the **Profile** window:

- **Start** – Enables the collection of performance data. It is best to start a profiling session while your application is not running to ensure that you measure only complete runs of VIs, and not partial runs.
- **Snapshot** – Views the data currently available. This gathers data from all the VIs in memory and displays it in the tabular display.
- **Save** – Saves the currently displayed data to disk as a tab-delimited text spreadsheet file. This data can then be viewed in a spreadsheet program or by VIs.

Viewing the Results

You can choose to display only parts of the information in the table. Some basic data is always visible, but you can choose to display the statistics, details, and (if enabled) memory usage by checking or unchecking the appropriate checkboxes in the **Profile** window.

Performance information also is displayed for Global VIs. However, this information sometimes requires a slightly different interpretation, as described in the category-specific sections below.

You can view performance data for subVIs by double-clicking the name of the subVI in the tabular display. When you do this, new rows appear directly below the name of the VI and contain performance data for each of its subVIs. When you double-click the name of a Global VI, new rows appear for each of the individual controls on its front panel.

You can sort the rows of data in the tabular display by clicking in the desired column header. The current sort column is indicated by a bold header title.

Timings of VIs do not necessarily correspond to the amount of elapsed time that it takes for a VI to complete. This is because a multithreaded execution system can interleave the execution of two or more VIs. Also, there is a certain amount of overhead not attributed to any VI, such as the amount of time taken by a user to respond to a dialog box, or time spent in a Wait function on a block diagram, or time spent to check for mouse clicks.

The basic information that is always visible in the first three columns of the performance data consists of the following items:

- **VI Time** – Total time spent actually executing the code of the VI and displaying its data, as well as time spent by the user interacting with its front panel controls (if any). This summary is divided into sub-categories in the **Timing Details** described in the following section. For Global VIs, this time is the total amount of time spent copying data to or from all of its controls. Double-click the name of the Global VI to view timing information for individual controls.
- **SubVIs Time** – Total time spent by all subVIs of the VI. This is the sum of the VI time (described above) for all callees of the VI, as well as their callees, etc.
- **Total Time** – Sum of the VI Time and SubVIs Time, calculating the total amount of time.

Timing Information

When the **Timing Statistics** checkbox is checked, the following columns are visible in the tabular display:

- **# Runs** – Number of times that the VI completed a run. For Global VIs, this time is the total number of times any of its controls were accessed
- **Average** – Average amount of time spent by the VI per run. This is simply the VI time divided by the number of runs.

- **Shortest** – Minimum amount of time the VI spent in a run
- **Longest** – Maximum amount of time the VI spent in a run

When the **Timing Details** checkbox is checked, you can view a breakdown of several timing categories that sum up the time spent by the VI. For VIs that have a great deal of user interface, these categories can help you determine what operations take the most time.

- **Diagram** – Time spent only executing the code generated for the block diagram of the VI
- **Display** – Time spent updating front panel controls of the VI with new values from the block diagram
- **Draw** – Time spent drawing the front panel of the VI. Draw time includes the following:
  - The time it takes simply to draw a front panel when its window just has been opened, or when it is revealed after being obscured by another window
  - The time it takes to draw controls that are overlapping or transparent. These controls must invalidate their area of the screen when they receive new data from the block diagram so everything in that area can redraw in the correct order. Other controls can draw immediately on the front panel when they receive new data from the block diagram. More overhead is involved in invalidating and redrawing, most (but not all) of which shows up in the Draw timings.
- **Tracking** – Time spent tracking the mouse while the user was interacting with the front panel of the VI. This can be significant for some kinds of operations, such as zooming in or out of a graph, selecting items from a pop-up menu, or selecting and typing text in a control.
- **Locals** – Time spent copying data to or from local variables on the block diagram. Experience with users shows this time can sometimes be significant, especially when it involves large, complex data.

Memory Information

When the **Memory Usage** checkbox is checked (remember, this is only available if the **Profile Memory Usage** checkbox was selected before you began the profiling session), you can view information about how your VIs are using memory. These values are a measure of the memory used by the data space for the VI and do not include the support data structures necessary for all VIs. The data space for the VI contains not just the data explicitly being used by front panel controls, but also temporary buffers the compiler implicitly created.

The memory sizes are measured at the conclusion of the run of a VI and might not reflect its exact, total usage. For instance, if a VI creates large arrays during its run but reduces their size before the VI finishes, the sizes displayed do not reflect the intermediate larger sizes.

This section displays two sets of data – data related to the number of bytes used, and data related to the number of blocks used. A block is a contiguous segment of memory used to store a single piece of data. For example, an array of integers might be multiple bytes in length, but it occupies only one block. The execution system uses independent blocks of memory for arrays, strings, paths, and pictures (from the Picture Control Toolkit). Large numbers of blocks in the memory heap of your application can cause an overall degradation of performance (not just execution).

The categories of Memory Usage are as follows:

- **Average Bytes** – Average number of bytes used by the data space of the VI per run
- **Min Bytes** – Minimum number of bytes used by the data space of the VI for an individual run
- **Max Bytes** – Maximum number of bytes used by the data space of the VI for an individual run
- **Average Blocks** – Average number of blocks used by the data space of the VI per run
- **Min Blocks** – Minimum number of blocks used by the data space of the VI for an individual run
- **Max Blocks** – Maximum number of blocks used by the data space of the VI for an individual run

VI Execution Speed

Although the compiler produces code that generally executes very quickly, when working on time-critical applications you want to do all you can to obtain the best performance out of your VIs. This section discusses factors that affect execution speed and suggests some programming techniques to help you obtain the best performance possible.

Examine the following items to determine the causes of slow performance:

- Input/Output (files, GPIB, data acquisition, networking)
- Screen Display (large controls, overlapping controls, too many displays)
- Memory Management (inefficient usage of arrays and strings, inefficient data structures)

Other factors, such as execution overhead and subVI call overhead, can have an effect, but these are usually minimal.

Input/Output

Input/Output (I/O) calls generally incur a large amount of overhead. They often take much more time than a computational operation takes. For example, a simple serial port read operation might have an associated overhead of several milliseconds. This amount of overhead occurs in any application that uses serial ports. The reason for this overhead is that an I/O call involves transferring information through several layers of an operating system.

The best way to address too much overhead is to minimize the number of I/O calls you make. Performance improves if you can structure your application so that you transfer a large amount of data with each call, instead of making multiple I/O calls using smaller amounts of data.

For example, if you are creating a data acquisition (NI-DAQ) VI, you have two options for reading data. You can use a single-point data transfer function such as the AI Sample Channel VI, or you can use a multipoint data transfer function such as the AI Acquire Waveform VI. If you must acquire 100 points, use the AI Sample Channel VI in a loop with a Wait function to establish the timing. You also can use the AI Acquire Waveform VI with an input indicating you want 100 points.

You can produce much higher and more accurate data sampling rates by using the AI Acquire Waveform VI, because it uses hardware timers to manage the data sampling. In addition, overhead for the AI Acquire Waveform VI is roughly the same as the overhead for a single call to the AI Sample Channel VI, even though it is transferring much more data.

Screen Display

Frequently updating controls on a front panel can be one of the most time-expensive operations in an application. This is especially true if you use some of the more complicated displays, such as graphs and charts.

This overhead is minimized to a certain extent because most of the controls are intelligent. They do not redraw when they receive new data if the new data is the same as the old data. Graphs and charts are exceptions to this rule.

If redraw rate becomes a problem, the best solutions are to reduce the number of controls you use and keep the displays as simple as possible. In the case of graphs and charts, you can turn off autoscaling, scale markers, and grids to speed up displays.

If you have controls overlapped with other objects, their display rate is significantly slower. The reason for this is that if a control is partially obscured, more work must be done to redraw that area of the screen. Unless you have the **Smooth Updates** preference on, you might see more flicker when controls are overlapped.

As with other kinds of I/O, there is a certain amount of fixed overhead in the display of a control. You can pass multiple points to an indicator at one time using certain controls, such as charts. You can minimize the number of chart updates you make by passing more data to the chart each time. You can see much higher data display rates if you collect your chart data into arrays to display multiple points at a time, instead of displaying each point as it comes in.

When you design subVIs whose front panels are closed during execution, do not be concerned about display overhead. If the front panel is closed, you do not have the drawing overhead for controls, so graphs are no more expensive than arrays.

In multithreaded systems, controls and indicators have a synchronous display pop-up item that controls whether or not updates can be deferred. In single-threaded execution, this item has no effect. However, if you turn this item on or off within VIs in the single-threaded version, those changes affect the way updates behave if you load those VIs into a multithreaded system.

By default, this item is off, which means that after the execution system passes data to front panel controls and indicators, it can immediately continue execution. At some point thereafter, the user interface system notices that the control or indicator needs to be updated, and it redraws to show the new data. If the execution system attempts to update the control multiple times in rapid succession, you might not see some of the intervening updates.

In most applications, this significantly speeds up execution without affecting what the user sees. For example, you can update a Boolean value hundreds of times in a second, which is more updates than the human eye can discern. Asynchronous displays permit the execution system to spend more time executing VIs, with updates automatically reduced to a slower rate by the user interface thread.

If you want synchronous displays, you can turn the synchronous display on.

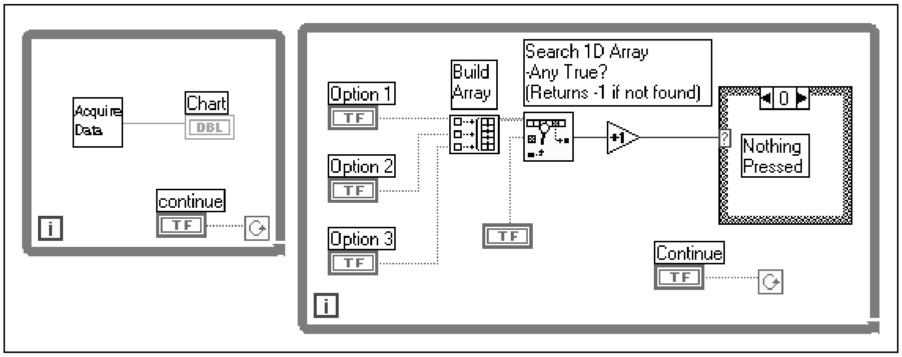
Note Turn on synchronous display only when it is necessary to display every data value. Using synchronous display results in a large performance penalty on multithreaded systems.

Other Issues

Parallel Block Diagrams

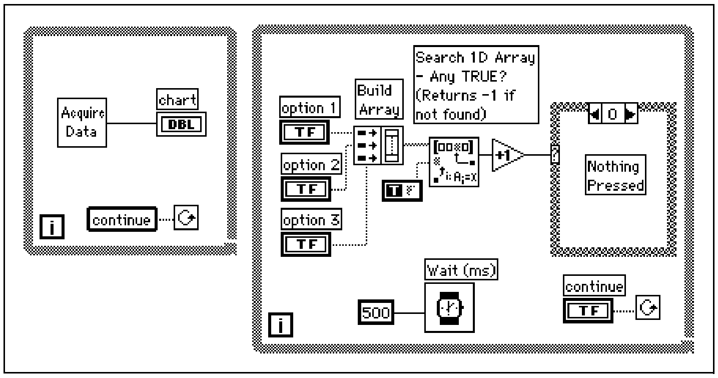
When you have multiple block diagrams running in parallel, the execution system switches between them periodically. If some of these loops are less important than others, use the Wait function to ensure the less important loops use less time.

For example, consider the following block diagram.



There are two loops in parallel. One of the loops is acquiring data and needs to execute as frequently as possible. The other loop is monitoring user input. The loops receive equal time because of the way this program is structured. The loop monitoring the user’s action has a chance to run several times a second.

In practice, it is usually acceptable if the loop monitoring the button executes only once every half second, or even less often. By calling the Wait (ms) function in the user interface loop, you allot significantly more time to the other loop.



SubVI Overhead

When you call a subVI, there is a certain amount of overhead associated with the call. This overhead is fairly small (on the order of tens of microseconds), especially in comparison to I/O overhead and display overhead, which can range from milliseconds to tens of milliseconds.

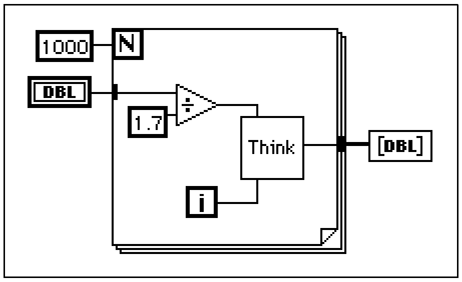
However, this overhead can add up in some cases. For example, if you call a subVI 10,000 times in a loop, this overhead might take up a significant amount of time. In this case, you might want to consider whether the loop can be embedded in the subVI.

Another option is to turn certain subVIs into subroutines (using the VI Setup **Priority** item). When a VI is marked as a subroutine, the execution system minimizes the overhead to call a subVI. There are a few trade-offs, however. Subroutines cannot display front panel data (LabVIEW does not copy data from or to the front panel controls of subroutines), they cannot contain timing or dialog box functions, and they do not multitask with other VIs. Subroutines are short, frequently executed tasks and are generally most appropriate when used with VIs that do not require user interaction.

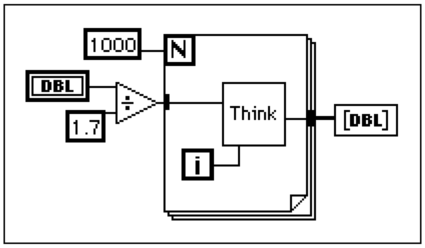
Unnecessary Computation in Loops

Avoid putting a calculation in a loop if the calculation produces the same value for every iteration. Instead, move the calculation out of the loop and pass the result into the loop.

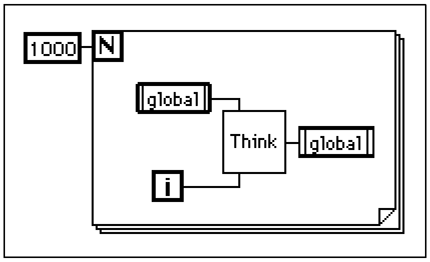
For example, examine the following block diagram.



The result of the division is the same every time through the loop; therefore you can increase performance by moving the division out of the loop, as shown in the following illustration.

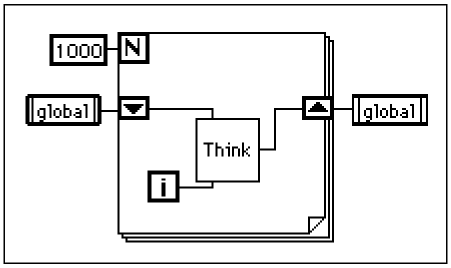


Now, refer to the block diagram in the following illustration.

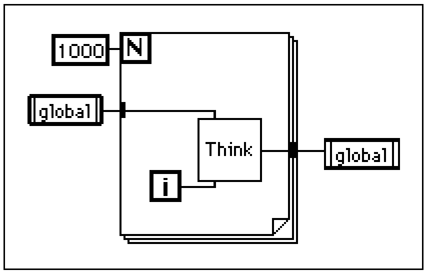


If you know the value of the global variable is not going to be changed by another concurrent block diagram or VI during this loop, this block diagram wastes time by reading from the global variable and writing to the global every time through the loop.

If you do not require the global variable to be read from or written to by another block diagram during this loop, you might use the following block diagram instead.



Notice that the shift registers must pass the new value from the subVI to the next iteration of the loop. The following block diagram shows a common mistake some beginning users make. Since there is no shift register, the results from the subVI never pass back to the subVI as its new input value.



VI Memory Usage

LabVIEW handles many of the details that you must handle in a text-based programming language. One of the main challenges of a text-based language is memory usage. In a text-based language, you, the programmer, have to take care of allocating memory before you use it and deallocating it when you finish. You also must be careful not to write past the end of the memory you allocated in the first place. Failure to allocate memory or to allocate enough memory is one of the biggest mistakes programmers make in text-based languages. Inadequate memory allocation is also a difficult problem to debug.

The dataflow paradigm for LabVIEW removes much of the difficulty of managing memory. In LabVIEW, you do not allocate variables, nor assign values to and from them. Instead, you create a block diagram with connections representing the transition of data.

Functions that generate data take care of allocating the storage for that data. When data is no longer being used, the associated memory is deallocated. When you add new information to an array or a string, enough memory is automatically allocated to manage the new information.

This automatic memory handling is one of the chief benefits of LabVIEW. However, because it is automatic, you have less control of when it happens. If your program works with large sets of data, it is important to have some understanding of when memory allocation takes place. An understanding of the principles involved can result in programs with significantly smaller memory requirements. Also, an understanding of how to minimize memory usage can also help to increase VI execution speeds, because memory allocation and copying data can take a considerable amount of time.

Virtual Memory

If you have a machine with a limited amount of memory, you might want to consider using virtual memory to increase the amount of memory available for applications. Virtual memory is a capability of your operating system by which it uses available disk space for RAM storage. If you allocate a large amount of virtual memory, applications perceive this as memory generally available for storage.

For applications, it does not matter if your memory is real RAM or virtual memory. The operating system hides the fact that the memory is virtual. The main difference is speed. With virtual memory, you occasionally might notice more sluggish performance, when memory is swapped to and from the disk by the operating system. Virtual memory can help run larger applications but is probably not appropriate for applications that have critical time constraints.

Macintosh Memory

When you launch an application on the Macintosh, the system allocates a single block of memory for it, from which all memory is allocated. When you load VIs, components of those are loaded into that memory. Similarly, when you run a VI, all the memory it manipulates is allocated out of that single block of memory.

You configure the amount of memory the system allocates at launch time using the **File»Get Info** command from the Finder. Remember, if your application runs out of memory, it cannot increase the size of this memory pool. Therefore, set up this parameter to be as large as is practical. If you have a 16 MB machine, consider the applications you want to run in parallel. If you do not plan to run any other applications, set the memory preference to be as large as possible.

VI Component Memory Management

A VI has the following four major components:

- Front panel
- Block diagram
- Code (diagram compiled to machine code)
- Data (control and indicator values, default data, diagram constant data, and so on)

When a VI loads, the front panel, the code (if it matches the platform), and the data for the VI are loaded into memory. If the VI needs to be recompiled because of a change in platform or a change in the interface of a subVI, the block diagram is loaded into memory as well.

The VI also loads the code and data space of its subVIs into memory. Under certain circumstances, the front panel of some subVIs might be loaded into memory as well. For example, this can occur if the subVI uses attribute nodes, because attribute nodes manipulate state information for front panel controls. More information on subVIs and their panels is discussed later in this application note.

An important point in the organization of VI components is that you generally do not use much more memory when you convert a section of your VI into a subVI. If you create a single, huge VI with no subVIs, you end up with the front panel, code, and data for that top-level VI in memory. However, if you break the VI into subVIs, the code for the top-level VI is smaller, and the code and data of the subVIs reside in memory. In some cases, you might see lower run-time memory usage. This idea is explained later in this application note in the Monitoring Memory Usage section.

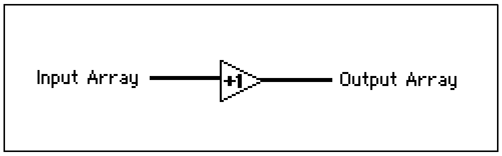
You also might find that massive VIs take longer to edit. You can avoid this problem if you break your VI into subVIs, because the editor can handle smaller VIs more efficiently. Also, a more hierarchical VI organization is generally easier to maintain and read.

Note If the front panel or block diagram of a given VI is much larger than a screen, you might want to break it into subVIs to make it more accessible.

Dataflow Programming and Data Buffers

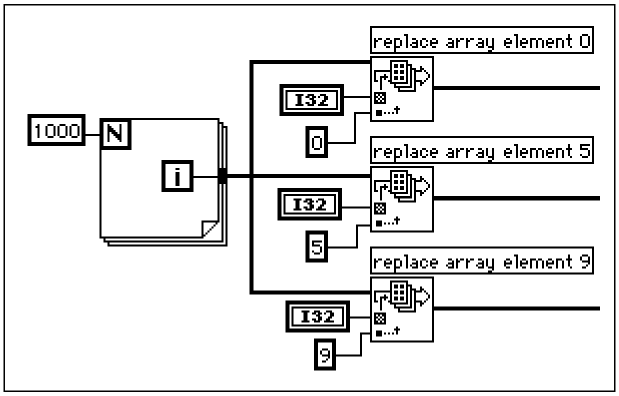
In dataflow programming, you generally do not use variables. Dataflow models usually describe nodes as consuming data inputs and producing data outputs. A literal implementation of this model produces applications that can use very large amounts of memory and have sluggish performance. Every function produces a copy of data for every destination to which an output is passed. The LabVIEW compiler improves on this implementation by attempting to determine when memory can be reused and by looking at the destinations of an output to determine whether it is necessary to make copies for each individual terminal.

For example, in a more traditional approach to the compiler, the following block diagram uses two blocks of data memory, one for the input and one for the output.



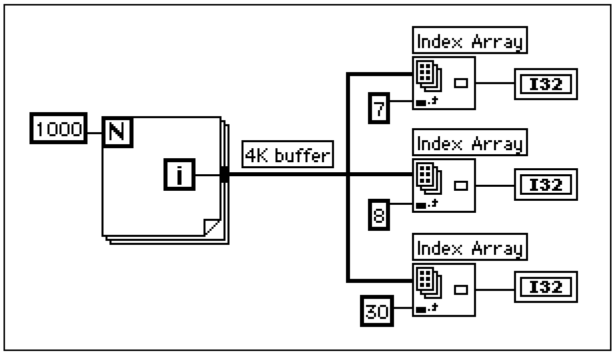
The input array and the output array contain the same number of elements, and the data type for both arrays is the same. Think of the incoming array as a buffer of data. Instead of creating a new buffer for the output, the compiler reuses the input buffer. This saves memory and also results in faster execution, because no memory allocation needs to take place at run time.

However, the compiler cannot reuse memory buffers in all cases, as shown in the following illustration.



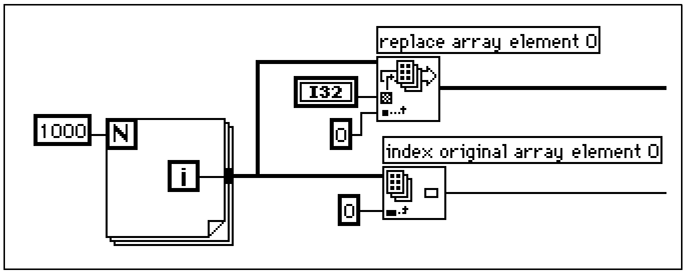
A signal passes a single source of data to multiple destinations. The Replace Array Element functions modify the input array to produce the output array. In this case, the compiler creates new data buffers for two of the functions and copies the array data into the buffers. Thus, one of the functions reuses the input array, and the others do not. This block diagram uses about 12 KB (4 KB for the original array and 4 KB for each of the extra two data buffers).

Now, examine the following block diagram.



As before, the input branches to three functions. However, in this case the Index Array function does not modify the input array. If you pass data to multiple locations, all of which read the data without modifying it, LabVIEW does not make a copy of the data. This block diagram uses about 4 KB of memory.

Finally, consider the following block diagram.



In this case, the input branches to two functions, one of which modifies the data. There is no dependency between the two functions. Therefore, you can predict that at least one copy needs to be made so the Replace Array Element function can safely modify the data. In this case, however, the compiler schedules the execution of the functions in such a way that the function that reads the data executes first, and the function that modifies the data executes last. This way, the Replace Array Element function reuses the incoming array buffer without generating a duplicate array. If the ordering of the nodes is important, make the ordering explicit by using either a sequence or an output of one node for the input of another.

In practice, the analysis of block diagrams by the compiler is not perfect. In some cases, the compiler might not be able to determine the optimal method for reusing block diagram memory.

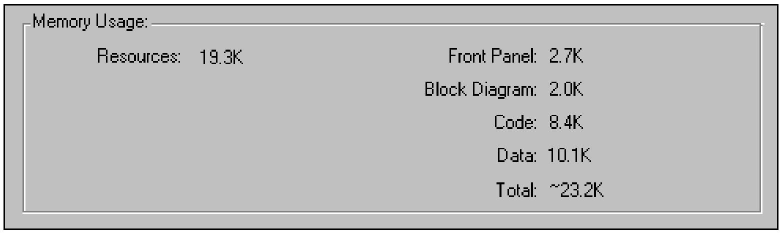
Monitoring Memory Usage

There are a couple of methods for determining memory usage.

Select **Help»About LabVIEW** to view statistics that summarize the total amount of memory used by your application. This includes memory for VIs as well as memory the application uses. You can check this amount before and after execution of a set of VIs to obtain a rough idea of how much memory the VIs are using.

You can obtain a view of the dynamic usage of memory by your VIs with the Performance Profiler. It keeps statistics on the minimum, maximum, and average number of bytes and blocks used by each VI per run. Refer to the VI Performance Profiling section at the beginning of this application note for more details.

As shown in the following illustration, you can use **Windows»Show VI Info** to see a breakdown of the memory usage for a VI. The left column summarizes disk usage, and the right column summarizes how much RAM currently is being used for various components of the VI. Notice these statistics do not include memory usage of subVIs.



Memory Usage:

Resources: 19.3K

Front Panel: 2.7K
Block Diagram: 2.0K
Code: 8.4K
Data: 10.1K
Total: ~23.2K

A fourth method for determining memory usage is to use the Memory Monitor VI (located in `memmon.llb` in the `examples` directory). This VI uses the VI Server functions to determine memory usage for all VIs in memory.

Rules for Better Memory Usage

The main point of the previous discussion is that the compiler attempts to reuse memory intelligently. The rules for when the compiler can and cannot reuse memory are fairly complex and are discussed at the end of this application note. In practice, the following rules can help you to create VIs that use memory efficiently:

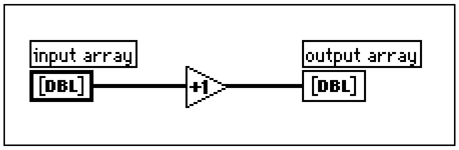
- Breaking a VI into subVIs generally does not hurt your memory usage. In many cases, memory usage improves, because the execution system can reclaim subVI data memory when the subVI is not executing.
- Do not worry too much about copies of scalar values; it takes a lot of scalars to negatively effect memory usage.
- Do not overuse global and local variables when working with arrays or strings. Reading a global or local variable causes a copy of the data of the variable to be generated.
- Do not display large arrays and strings on open front panels unless it is necessary. Indicators on open front panels retain a copy of the data they display.
- If the front panel of a subVI is not going to be displayed, do not leave unused attribute nodes on the subVI. Attribute nodes cause the front panel of a subVI to remain in memory, which can cause unnecessary memory usage.
- When designing block diagrams, watch for places where the size of an input is different from the size of an output. For example, if you see places where you are frequently increasing the size of an array or string using the Build Array or Concatenate Strings functions, you are generating copies of data.

- Use consistent data types for arrays and watch for coercion dots when passing data to subVIs and functions. When you change data types, the execution system makes a copy of the data.
- Do not use complicated, hierarchical data types such as clusters or arrays of clusters containing large arrays or strings. You might end up using more memory. Refer to the Developing Efficient Data Structures section later in this application note for more details and suggestions for tactics in designing your data types.

Memory Issues in Front Panels

When a front panel is open, controls and indicators keep their own, private copy of the data they display.

The following illustration shows the increment function, with the addition of front panel controls and indicators.



When you run the VI, the data of the front panel control is passed to the block diagram. The increment function reuses the input buffer. The indicator then makes a copy of the data for display purposes. Thus, there are three copies of the buffer.

This data protection of the front panel control prevents the case in which you enter data into a control, run the associated VI, and see the data change in the control as it is passed in-place to subsequent nodes. Likewise, data is protected in the case of indicators so that they can reliably display the previous contents until they receive new data.

With subVIs, you can use controls and indicators as inputs and outputs. The execution system makes a copy of the control and indicator data of the subVI in the following conditions:

- The front panel is in memory. This can occur for any of the following reasons:
  - The front panel is open.
  - The VI has been changed but not saved (all components of the VI remain in memory until the VI is saved).
  - The panel uses data printing.
  - The block diagram uses attribute nodes.
- The VI uses local variables.
- The panel uses data logging.
- A control uses suspend data range checking.

For an attribute node to be able to read the chart history in subVIs with closed panels, the control or indicator needs to display the data passed to it. Because there are numerous other attributes like this, the execution system keeps subVI panels in memory if the subVI uses attribute nodes.

If a front panel uses front panel datalogging or data printing, controls and indicators maintain copies of their data. In addition, panels are kept in memory for data printing so the panel can be printed.

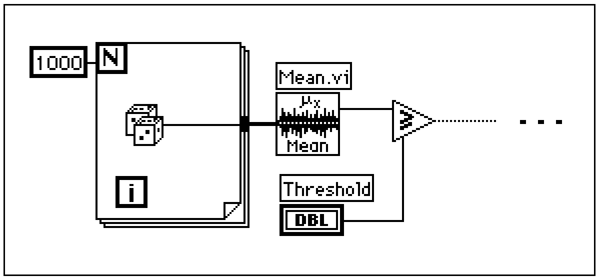
Remember, if you set a subVI to display its front panel when called using VI Setup or SubVI Setup, the panel is loaded into memory when the subVI is called. If you set the **Close if Originally Closed** item, the panel is removed from memory when the subVI finishes execution.

SubVIs Can Reuse Data Memory

In general, a subVI can use data buffers from its caller as easily as if the block diagrams of the subVI were duplicated at the top level. In most cases, you do not use more memory if you convert a section of your block diagram into a subVI. For VIs with special display requirements, as described in the previous section, there might be some additional memory usage for front panels and controls.

Understanding When Memory Is Deallocated

Consider the following block diagram.



After the Mean VI executes, the array of data is no longer needed. Because determining when data is no longer needed can become very complicated in larger block diagrams, the execution does not deallocate the data buffers of a particular VI during its execution.

On the Macintosh, if the execution system is low on memory, it deallocates data buffers used by any VI not currently executing. The execution system does not deallocate memory used by front panel controls, indicators, global variables, or uninitialized shift registers.

Now consider the same VI as a subVI of a larger VI. The array of data is created and used only in the subVI. On the Macintosh, if the subVI is not executing and the system is low on memory, it might deallocate the data in the subVI. This is a case in which using subVIs can save on memory usage.

On Windows and Unix platforms, data buffers are not normally deallocated unless a VI is closed and removed from memory. Memory is allocated from the operating system as needed, and virtual memory works well on these platforms. Due to fragmentation, the application might appear to use more memory than it really does. As memory is allocated and freed, the application tries to consolidate memory usage so it can return unused blocks to the operating system.

On all platforms, you can optionally set the “Deallocate memory as soon as possible” preference. When this item is on, subVIs immediately deallocate memory as they complete execution. This might help out with memory usage, but it will slow down performance significantly.

Determining When Outputs Can Reuse Input Buffers

If an output is the same size and data type as an input, and the input is not required elsewhere, the output can reuse the input buffer. As mentioned previously, in some cases even when an input is used elsewhere, the compiler and the execution system can order code execution in such a way that it can reuse the input for an output buffer. However, the rules for this are complex. Do not depend upon them.

Consistent Data Types

If an input has a different data type from an output, the output cannot reuse that input. For example, if you add a 32-bit integer to a 16-bit integer, you see a coercion dot that indicates the 16-bit integer is being converted to a 32-bit integer. The 32-bit integer input can be usable for the output buffer, assuming it meets all the other requirements (for example, the 32-bit integer is not being reused somewhere else).

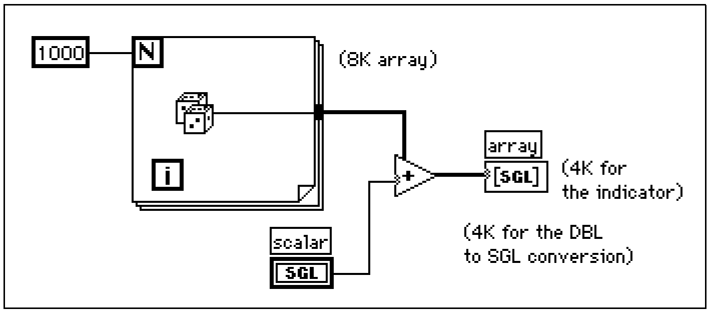
In addition, coercion dots for subVIs and many functions imply a conversion of data types. In general, the compiler creates a new buffer for the converted data.

To minimize memory usage, use consistent data types wherever possible. Doing this produces fewer copies of data because of promotion of data in size. Using consistent data types also makes the compiler more flexible in determining when data buffers can be reused.

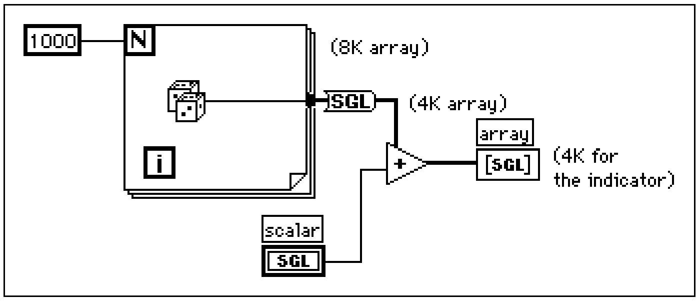
In some applications, you might consider using smaller data types. For example, you might consider using four-byte, single-precision numbers instead of eight-byte, double-precision numbers. However, carefully consider which data types are expected by subVIs you can call, because you want to avoid unnecessary conversions.

How to Generate Data of the Right Type

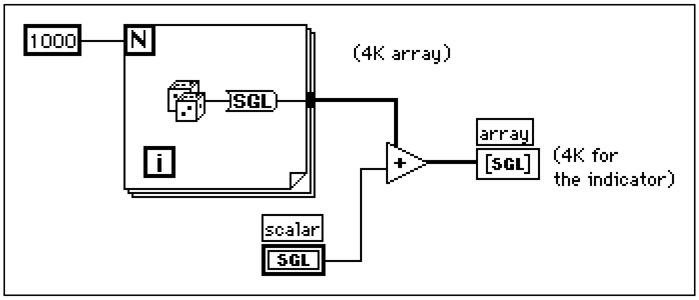
Refer to the following example in which an array of 1,000 random values is created and added to a scalar. The coercion dot at the Add function occurs because the random data is double-precision, while the scalar is single-precision. The scalar is promoted to double-precision before the addition. The resulting data is then passed to the indicator. This block diagram uses 16 KB of memory.



The following illustration shows an incorrect attempt to correct this problem by converting the array of double-precision random numbers to an array of single-precision random numbers. It uses the same amount of memory as the previous example.



The best solution, shown in the following illustration, is to convert the random number to single-precision as it is created, before you create an array. Doing this avoids the conversion of a large data buffer from one data type to another.

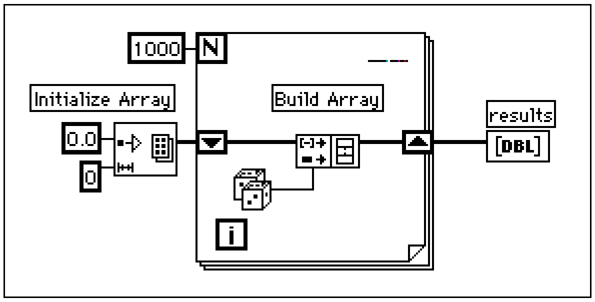


Avoid Constantly Resizing Data

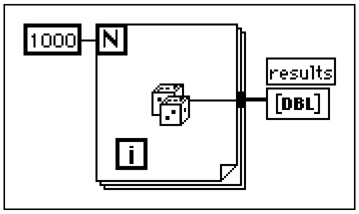
If the size of an output is different from the size of an input, the output does not reuse the input data buffer. This is the case for functions such as Build Array, String Concatenate, and Array Subset, which increase or decrease the size of an array or string. When working with arrays and strings, avoid constantly using these functions, because your program uses more data memory and executes more slowly because it is constantly copying data.

Example 1 – Building Arrays

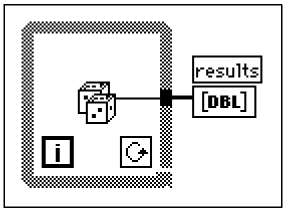
Consider the following block diagram, which is used to create an array of data. This block diagram creates an array in a loop by constantly calling Build Array to concatenate a new element. The input array is reused by Build Array. The VI continually resizes the buffer in each iteration to make room for the new array and appends the new element. The resulting execution speed is slow, especially if the loop is executed many times.



If you want to add a value to the array with every iteration of the loop, you can see the best performance by using auto-indexing on the edge of a loop. With For Loops, the VI can predetermine the size of the array (based on the value wired to N), and resize the buffer only once.



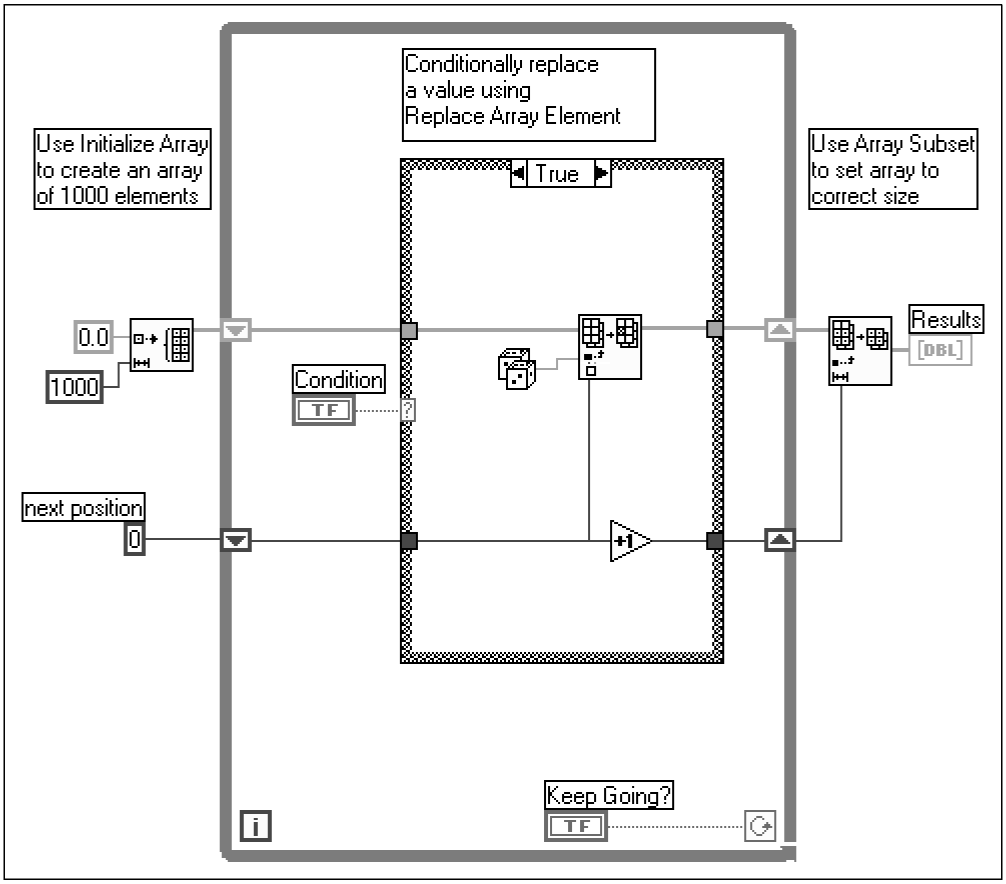
With While Loops, auto-indexing is not quite as efficient, because the end size of the array is not known. However, While Loop auto-indexing avoids resizing the output array with every iteration by increasing the output array size in large increments. When the loop is finished, the output array is resized to the correct size. The performance of While Loop auto-indexing is nearly identical to For Loop auto-indexing.



Auto-indexing assumes you are going to add a value to the resulting array with each iteration of the loop. If you must conditionally add values to an array but can determine an upper limit on the array size, you might consider preallocating the array and using Replace Array Element to fill the array.

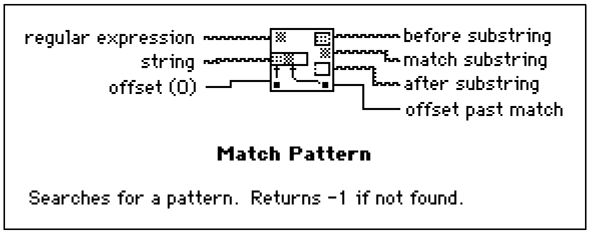
When you finish filling the array values, you can resize the array to the correct size. The array is created only once, and Replace Array Element can reuse the input buffer for the output buffer. The performance of this is very similar to the performance of loops using auto-indexing. If you use this technique, be careful the array in which you are replacing values is large enough to hold the resulting data, because Replace Array Element does not resize arrays for you.

An example of this process is shown in the following illustration.


Conditionally replace a value using Replace Array Element
Use Initialize Array to create an array of 1000 elements
True
Use Array Subset to set array to correct size
0.0
1000
Condition
TF
Results
DBL
next position
0
+1
Keep Going?
TF
i

Example 2 – Searching through Strings

You can use the Match Pattern function to search a string for a pattern. Depending on how you use it, you might slow down performance by unnecessarily creating string data buffers. The following illustration shows the Help window for Match Pattern.

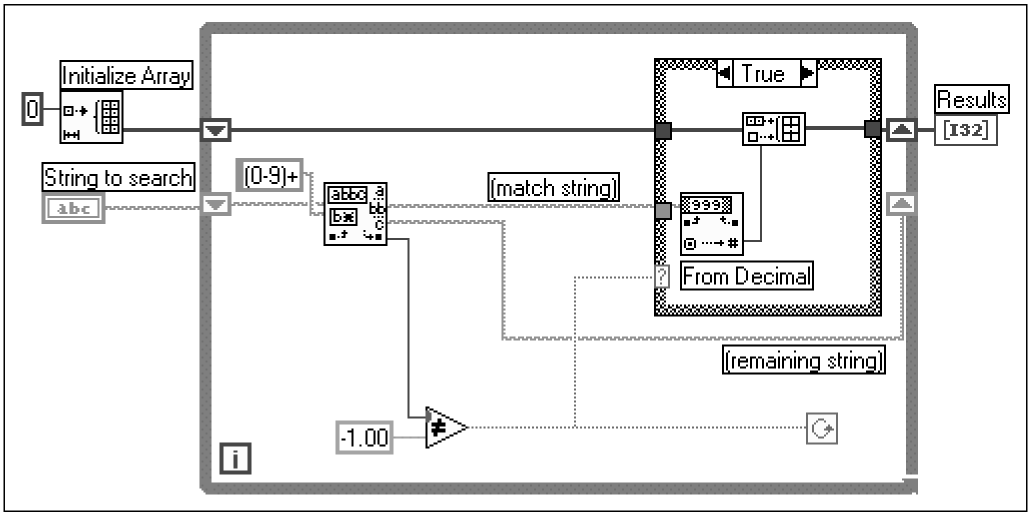


Match Pattern

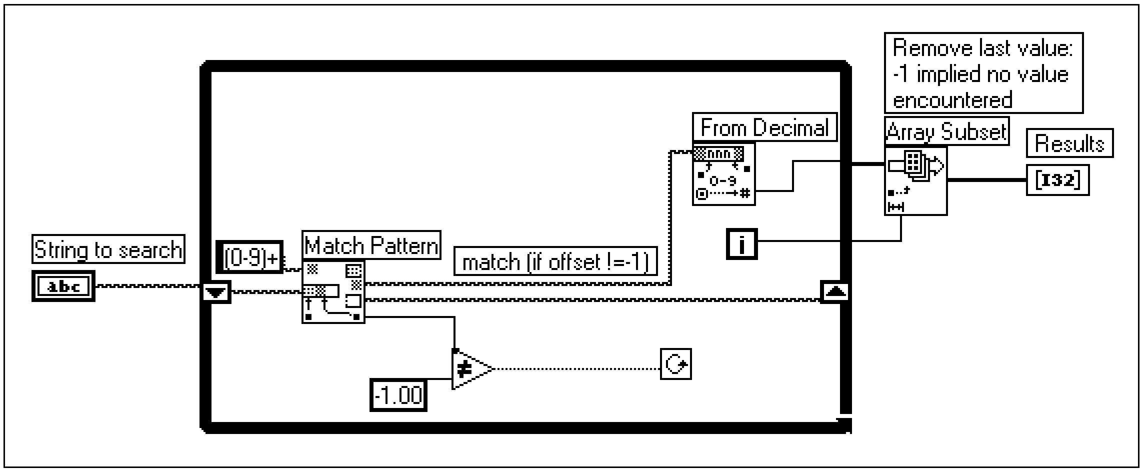
Searches for a pattern. Returns -1 if not found.

Assuming you want to match an integer in a string, you can use [0–9]+ as the regular expression input to this function. To create an array of all integers in a string, use a loop and call Match Pattern repeatedly until the offset value returned is –1.

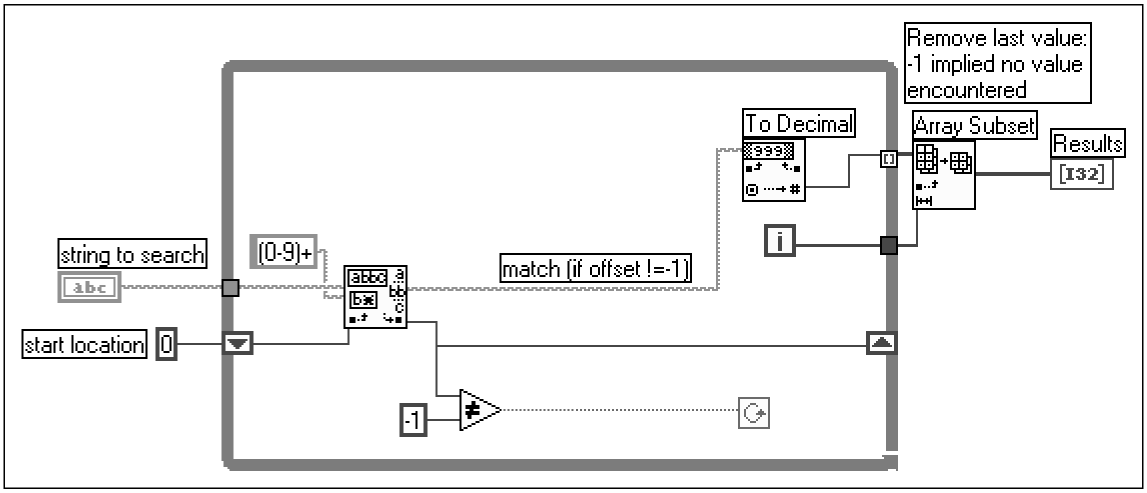
The following block diagram is one method for scanning for all occurrences of integers in a string. It creates an empty array and then searches the remaining string for the numeric pattern with each iteration of the loop. If the pattern is found (offset is not –1), this block diagram uses Build Array to add the number to a resulting array of numbers. When there are no values left in the string, Match Pattern returns –1 and the block diagram completes execution.



One problem with this block diagram is that it uses Build Array in the loop to concatenate the new value to the previous value. Instead, you can use auto-indexing to accumulate values on the edge of the loop. Notice you end up seeing an extra unwanted value in the array from the last iteration of the loop where Match Pattern fails to find a match. A solution is to use Array Subset to remove the extra unwanted value. This is shown in the following illustration.



The other problem with this block diagram is that you create an unnecessary copy of the remaining string every time through the loop. Match Pattern has an input you can use to indicate where to start searching. If you remember the offset from the previous iteration, you can use this number to indicate where to start searching on the next iteration. This technique is shown in the following illustration.

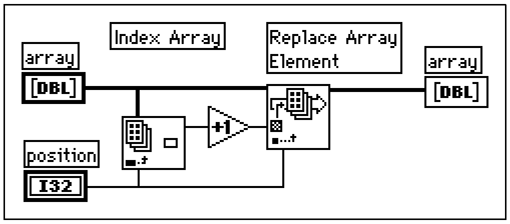


Developing Efficient Data Structures

One of the points made in the previous section is that hierarchical data structures, such as clusters or arrays of clusters containing large arrays or strings, cannot be manipulated efficiently. This section explains why this is so and describes strategies for choosing more efficient data types.

The problem with complicated data structures is that it is difficult to access and change elements within a data structure without causing copies of the elements you are accessing to be generated. If these elements are large, as in the case where the element itself is an array or string, these extra copies use more memory and the time it takes to copy the memory.

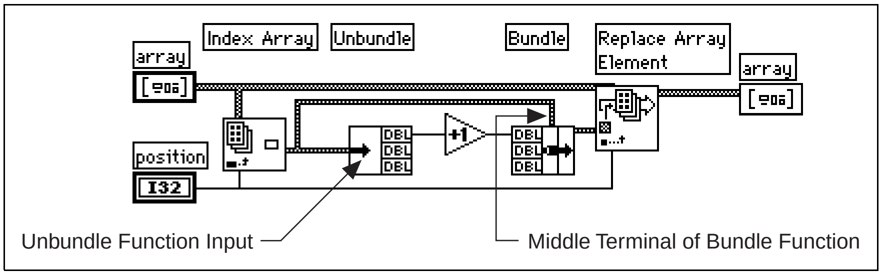
You can generally manipulate scalar data types very efficiently. Likewise, you can efficiently manipulate small strings and arrays where the element is a scalar. In the case of an array of scalars, the following code shows what you do to increment a value in an array.



This is quite efficient because it is not necessary to generate extra copies of the overall array. Also, the element produced by the Index Array function is a scalar, which can be created and manipulated efficiently.

The same is true of an array of clusters, assuming the cluster contains only scalars. In the following block diagram, manipulation of elements becomes a little more complicated, because you must use Unbundle and Bundle. However, because the cluster is probably small (scalars use very little memory), there is no significant overhead involved in accessing the cluster elements and replacing the elements back into the original cluster.

The following illustration shows the efficient pattern for unbundling, operating, and rebundling. The wire from the data source should have only two destinations – the Unbundle function input, and the middle terminal on the Bundle function. LabVIEW recognizes this pattern and is able to generate better-performing code.



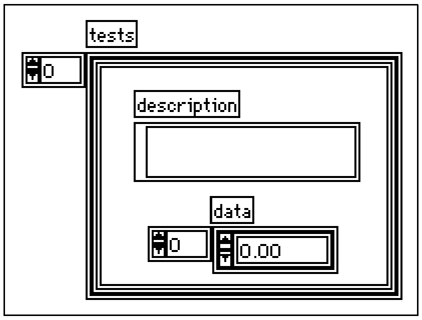
If you have an array of clusters where each cluster contains large sub-arrays or strings, indexing and changing the values of elements in the cluster can be more expensive in terms of memory and speed.

When you index an element in the overall array, a copy of that element is made. Thus, a copy of the cluster and its corresponding large subarray or string is made. Because strings and arrays are of variable size, the copy process can involve memory allocation calls to make a string or subarray of the appropriate size, in addition to the overhead actually copying the data of a string or subarray. This might not be significant if you only plan to do it a few times. However, if your application centers around accessing this data structure frequently, the memory and execution overhead might add up quickly.

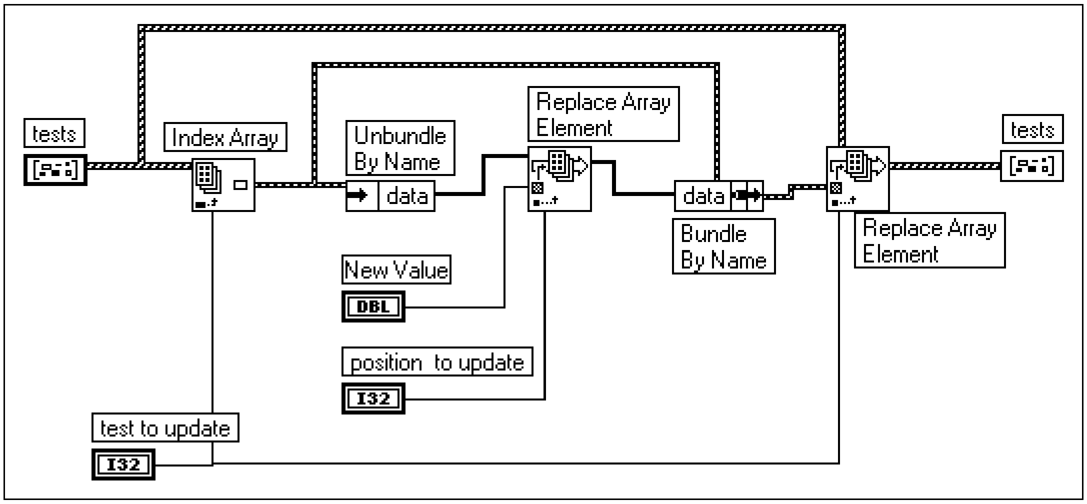
The solution is to look at alternative representations for your data. The following three case studies present three different applications, along with suggestions for the best data structures in each case.

Case Study 1 – Avoiding Complicated Data Types

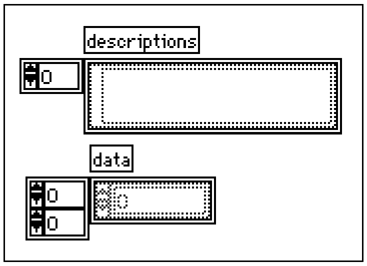
Consider an application in which you want to record the results of several tests. In the results, you want a string describing the test and an array of the test results. One data type you might consider using to store this information is shown in the following illustration.



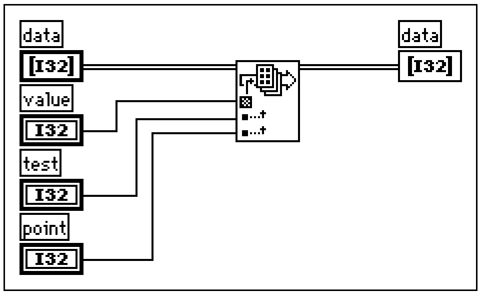
To change an element in the array, you must index an element of the overall array. Now, for that cluster you must unbundle the elements to reach the array. You then replace an element of the array and store the resulting array in the cluster. Finally, you store the resulting cluster into the original array. An example of this is shown in the following illustration.



Each level of unbundling/indexing might result in a copy of that data being generated. Notice a copy is not necessarily generated. Copying data is costly in terms of both time and memory. The solution is to try to make the data structures as flat as possible. For example, in this case study break the data structure into two arrays. The first array is the array of strings. The second array is a 2D array, where each row is the results of a given test. This result is shown in the following illustration.



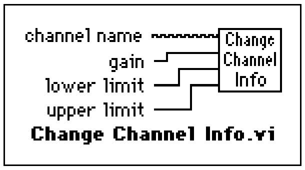
Given this data structure, you can replace an array element directly using the Replace Array Element function, as shown in the following illustration.

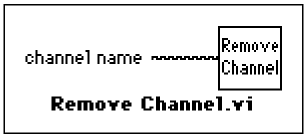


Case Study 2 – Global Table of Mixed Data Types

Here is another application in which you want to maintain a table of information. In this application, you want the data to be globally accessible. This table might contain settings for an instrument, including gain, lower and upper voltage limits, and a name used to refer to the channel.

To make the data accessible throughout your application, you might consider creating a set of subVIs to access the data in the table, such as the following subVIs, the Change Channel Info VI and the Remove Channel Info VI.





The following sections present three different implementations for these VIs.

Obvious Implementation

With this set of functions, there are several data structures to consider for the underlying table. First, you might use a global variable containing an array of clusters, where each cluster contains the gain, lower limit, upper limit, and the channel name.

As described in the previous section, this data structure is difficult to manipulate efficiently, because generally you must go through several levels of indexing and unbundling to access your data. Also, because the data structure is a conglomeration of several pieces of information, you cannot use the Search 1D Array function to search for a channel. You can use Search 1D Array to search for a specific cluster in an array of clusters, but you cannot use it to search for elements that match on a single cluster element.

Alternative Implementation 1

As with the previous example, choose to keep the data in two separate arrays. One contains the channel names. The other contains the channel data. The index of a given channel name in the array of names is used to find the corresponding channel data in the other array.

Notice that because the array of strings is separate from the data, you can use the Search 1D Array function to search for a channel.

In practice, if you are creating an array of 1,000 channels using the Change Channel Info VI, this implementation is roughly twice as fast as the previous version. This change is not very significant because there is other overhead affecting performance.

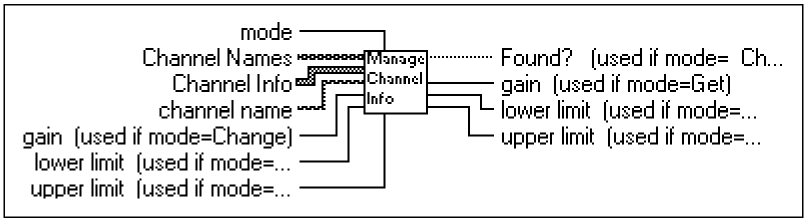
A note in the Memory Usage section of this application note cautions against overusing local and global variables. When you read from a global variable, a copy of the data of the global variable is generated. Thus, a complete copy of the data of the array is being generated each time you access an element. The next method shows an even more efficient method that avoids this overhead.

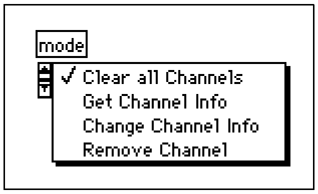
Alternative Implementation 2

There is an alternative method for storing global data, and that is to use an uninitialized shift register. Essentially, if you do not wire an initial value, a shift register remembers its value from call to call.

The LabVIEW compiler handles access to shift registers efficiently. Reading the value of a shift register does not necessarily generate a copy of the data. In fact, you can index an array stored in a shift register and even change and update its value without generating extra copies of the overall array. The problem with a shift register is only the VI that contains the shift register can access the shift register data. On the other hand, the shift register has the advantage of modularity.

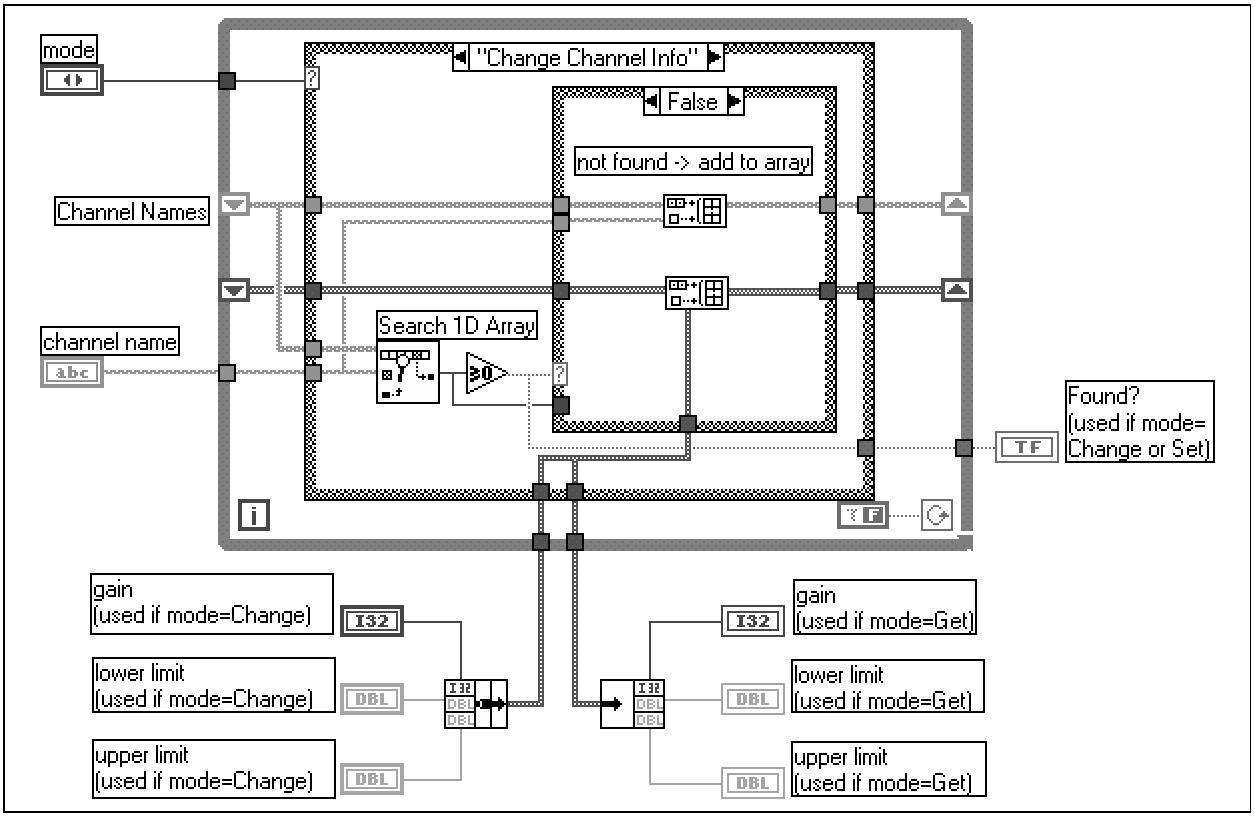
You can make a single subVI with a mode input that specifies whether you want to read, change, or remove a channel, or whether you want to zero out the data for all channels, as shown in the following illustration.


mode
Channel Names
Channel Info
channel name
gain (used if mode=Change)
lower limit (used if mode=...
upper limit (used if mode=...
Manage Channel Info
Found? (used if mode= Ch...
gain (used if mode=Get)
lower limit (used if mode=...
upper limit (used if mode=...



The subVI contains a While Loop with two shift registers – one for the channel data, and one for the channel names. Neither of these shift registers is initialized. Then, inside the While Loop you place a Case structure connected to the mode input. Depending on the value of the mode, you might read and possibly change the data in the shift register.

Following is an outline of a subVI with an interface that handles these three different modes. Only the Change Channel Info code is shown.

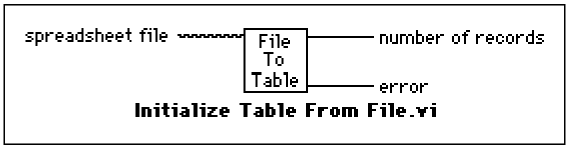


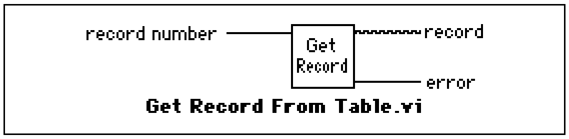
For 1,000 elements, this implementation is twice as fast as the previous implementation, and four times faster than the original implementation.

Case Study 3 – A Static Global Table of Strings

The previous example looked at an application in which the table contained mixed data types, and the table might change frequently. In many applications, you have a table of information that is fairly static once created. The table might be read from a spreadsheet file. Once read into memory, you mainly use it to look up information.

In this case, your implementation might consist of the following two functions, Initialize Table From File and Get Record From Table.





One way to implement the table is to use a two-dimensional array of strings. Notice the compiler stores each string in an array of strings in a separate block of memory. If there are a large number of strings (for example, more than 5,000 strings), you might put a load on the memory manager. This load can cause a noticeable loss in performance as the number of individual objects increases.

An alternative method for storing a large table is to read the table in as a single string. Then build a separate array containing the offsets of each record in the string. This changes the organization so that instead of having potentially thousands of relatively small blocks of memory, you instead have one large block of memory (the string) and a separate smaller block of memory (the array of offsets).

This method might be more complicated to implement, but it can be much faster for large tables.