DR. BOB DAVIDOV

Использование памяти виртуальным прибором LabView

Цель работы: получить представление о потреблении памяти объектами LabView.

Задача работы: научиться минимизировать память занимаемую LabView прибором.

Приборы и принадлежности: Персональный компьютер, LabView.

ВВЕДЕНИЕ

Зачастую, надежность работы системы управления с LabView интерфейсом зависит от распределения и использования памяти LabView. Особенно это может проявляться при использовании нескольких каналов передачи данных и относительной сложности VI интерфейса когда в реальном времени необходимо поддерживаться работу нескольких процессов, например, таких как

- высокоскоростной обмен данными по нескольким каналам (например, USB и PCI) с синхронизацией потоков;
- обработка и накопление больших массивов данных в стековую FIFO память;
- графическое отображение данных, включая построение фигур Лиссажу.

Понимание механизмов использования памяти в LabView помогает в решении задач повышения надежности системы управления с LabView модулями.

ОБЩИЕ СВЕДЕНИЯ

ИСПОЛЬЗОВАНИЕ ПАМЯТИ ВИРТУАЛЬНЫМ ПРИБОРОМ LABVIEW

В LabView решаются задачи выделения памяти и её высвобождения по окончании работы.

Управление потоками данных в LabView устраняет множество проблем управления памятью. В LabView переменные не выделяются и им не присваиваются значения как в текстовых языках программирования. Вместо этого создается блок-схема с соединениями, отображающими передачу данных.

Функции, которые генерируют данные занимаются и выделением области памяти для хранения данных. Когда данные больше не используется, связанная с ними память высвобождается. При добавлении новых данных в массив или строку, автоматически выделяется достаточно памяти для управления новыми данными.

Это автоматическое управление памятью является одной из главных особенностей LabView. Однако, поскольку оно происходит автоматически, у пользователя остается не много возможностей для контроля процесса управления памятью. Если программа работает с большими массивами данных процессы выделения памяти и копирования данных могут занимать значительное время, поэтому важно иметь представление о том, как происходит выделение памяти. Понимание этих принципов помогает минимизировать

используемую память, а это, в свою очередь, может увеличить скорость выполнения виртуального прибора (**VI**) LabView.

Виртуальная память

Операционные системы (ОС) используют виртуальную память для того, чтобы приложения могли получать доступ к большей памяти, чем доступный объем физической памяти. ОС разделяет физическую память на блоки, называемые страницами. Когда приложение или процесс, назначает адрес блока памяти, адрес напрямую не относится к физической памяти, а относится к памяти на странице. ОС может поменять страницы между физической оперативной памятью и жестким диском.

Если приложению или процессу необходим доступ к определенному блоку или странице, которой нет в физической памяти, операционная система может переместить неиспользуемую страницу физической памяти на жесткий диск и заменить ее нужной страницей. ОС отслеживает страницы в памяти и транслирует виртуальные адреса страниц в реальные адреса физической памяти, когда приложению или процессу необходим доступ к этой памяти.

На следующем рисунке показано, как два процесса могут подкачивать страницы из физической памяти. Для этого примера, процесс A и процесс B работают одновременно.

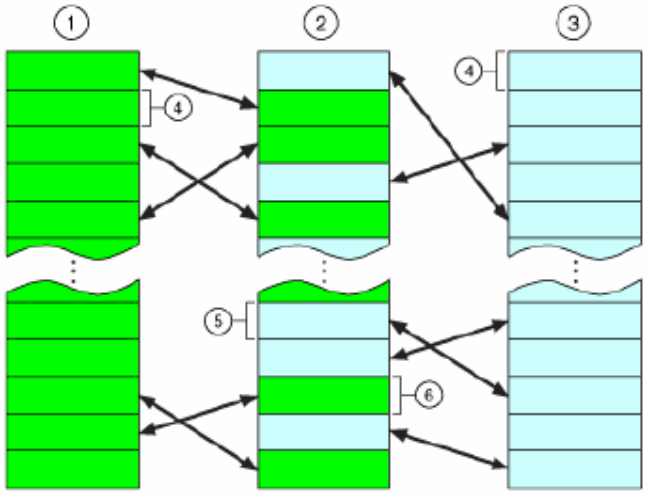


Рисунок 1. Обмен страницами памяти. 1 - Процесс A; 2 - Физическая RAM; 3 - Процесс B; 4 - Страница виртуальной памяти; 5 - Страница памяти процесса B; 6 - Страница памяти процесса A.

Поскольку количество страниц используемых приложением или процессом зависит от свободного места на диске, а не от объема доступной физической памяти, приложение может использовать больше памяти, чем размер доступной физической памяти. Размер адреса памяти используемой приложением ограничивает объем виртуальной памяти, к которой можно получить доступ. В 32-битном приложении LabView используются 32-битные адреса, и поэтому можно получить доступ не более чем к 4 Гб виртуальной памяти.

Если система имеет меньше, чем 4 ГБ ОЗУ, 32-разрядные приложения могут назначать адреса только до 3 Гб виртуальной памяти. ОС всегда оставляет определенное количество виртуальной памяти для своего ядра, или основного компонента ОС. По умолчанию операционная система оставляет 2 Гб для ядра и выделяет оставшиеся 2 Гб для пользователей.

Использование LabView виртуальной памяти в Windows

Объем виртуальной памяти к которой LabView может получить доступ зависит от версии используемой Windows.

- (Windows Vista x64 Edition) LabView может получить доступ к виртуальной памяти до 4 Гб по умолчанию. Поскольку LabView является 32-битным приложением, нельзя обеспечить его доступ к более чем 4 Гб виртуальной памяти.
- (Windows Vista) Чтобы обеспечить доступу LabView к виртуальной памяти до 3 Гб, можно использовать команду Bcdedit / set increaseuserva 3072 в окне командной строки, чтобы добавить запись в (BCD).
  Примечание: Чтобы открыть окно командной строки с правами администратора, перейдите в окно, в меню Пуск Windows, щелкните правой кнопкой мыши по имени программы и выберите Запуск от имени администратора в контекстном меню.
- (Windows XP/2000) Для того чтобы получить доступ LabView к виртуальной памяти с 2 Гб до 3 Гб на 32-битных ОС, следует модифицировать файл boot.ini в следующей последовательности [2].
  1. Щелкните правой кнопкой мыши на значке Мой компьютер и нажмите кнопку Properties (Свойства).
  2. В диалоговом окне Properties перейдите на вкладку Advanced (Дополнительно)
  3. .На вкладке Advanced в группе Startup and Recovery (Загрузка и восстановление) нажмите кнопку Settings (Настройка).
  4. В диалоговом окне Startup and Recovery нажмите кнопку Edit (Редактировать). В окне Microsoft ® Notepad появится boot.ini.
  5. Создайте резервную копию файла boot.ini .
     Примечание: Boot.ini файлы могут изменяться от компьютера к компьютеру.
  6. Выберите следующую строку в boot.ini файле:

     multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /fastdetect
  7. Нажмите Ctrl+C , чтобы скопировать строку и нажмите Ctrl+V , чтобы вставить ее непосредственно под оригинальной строкой.
     Примечание: текстовая строка может отличаться от текстовой строки в этом примере, так что не забудьте скопировать текстовую строку из своего файла boot.ini , а не текстовую строку этого примера.Добавьте в скопированную строку " / 3 GB " , как показано в следующем примере:

     multi(0)disk(0)rdisk(0)partition(2)\WINDOWS="Microsoft Windows XP Professional 3GB" /3GB /fastdetect
     Примечание: не перезаписывайте уже существующие линии
  8. Сохраните и закройте boot.ini файл.
  9. Нажмите OK, чтобы закрыть все диалоговые окна.
  10. Перезагрузите компьютер.
  11. Во время запуска , выберите опцию 3 Гб . Если опцию 3 Гб не выбрана, система по умолчанию установит 2 Гб общей памяти.

  Обратите внимание, что только опытный пользователь должен выполнять вышеуказанные процедуры.

Управление памятью компонентов VP

VP имеет следующие четыре основных компонента:

- Передняя панель
- Блок-схема
- Код (блок-схема скомпилированная в машинные коды)

- Данные (значения управления и индикаторов, данные по умолчанию, константы и т.д.)

Когда загружается VP, его передняя панель, код и данные попадают в память, также в память загружаются код и данные его подустройств - SubVIs. При определенных обстоятельствах, в память могут быть загружены и передние панели (front panels) некоторых SubVI. Например, когда SubVI использует узел Property Nodes, поскольку он манипулирует состоянием передней панели.

Примечание: Узел Property Nodes (например, 

) вызывается щелчком правой кнопкой мыши по объекту передней панели или его терминалу на блок-схеме, и далее, как показано на рисунке ниже.

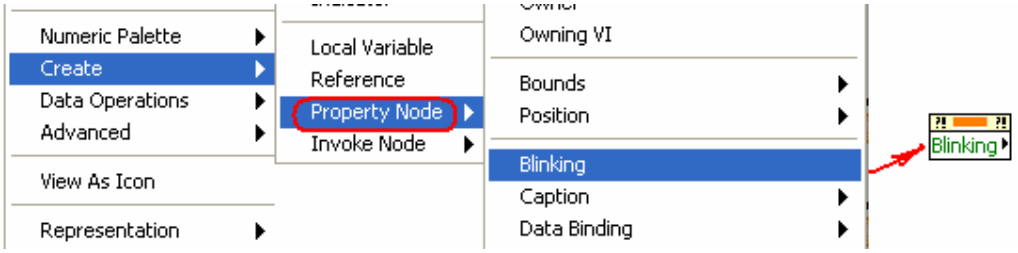


Рис 1. Пример создания узла Property Nodes объекта передней панели.

Стоит отметить, что при преобразовании фрагментов VP в SubVI общая память почти не увеличивается, однако память кода верхнего уровня VP сокращается. В некоторых случаях, при разбиении VP на SubVI можно заметить уменьшение общей используемой памяти.

Для редактирования массивных VP требуется больше времени. Разбиение VP на SubVI может повысить эффективность редактирования. Кроме того, VP с более крупной иерархической организацией, как правило, легче читается и обслуживается.

Примечание: Переднюю панель или блок-схему которая гораздо больше дисплея можно сделать более доступной разбив ее на SubVI,

Необходимо отметить, что при перекомпиляции VP в память загружается и его блок-схема.

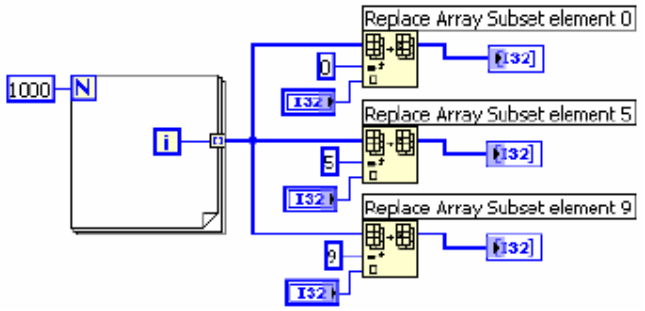
Программирование потока данных и буферизация данных

В программировании потока данных переменные, как правило, не используются. Модели потока данных обычно представляют собой узлы со входами для приема данных и узлы с выходами для передачи данных которые формируются самими узлами. Вариантом этой модели являются приложения, которые используют очень большие объемы памяти при низкой производительности. В таких реализациях каждая функция создает копию данных для каждого узла которому данные передаются. Компилятор LabView улучшает эту реализацию путем определения можно ли повторно использовать память и надо ли делать копии для каждого отдельного терминала. Можно использовать окно “**Show Buffer Allocations**” для того, чтобы определить, где LabView создает копии данных.

При обычной компиляции, в следующей блок-схеме используются два блока памяти: один для входных данных и один для выходных.

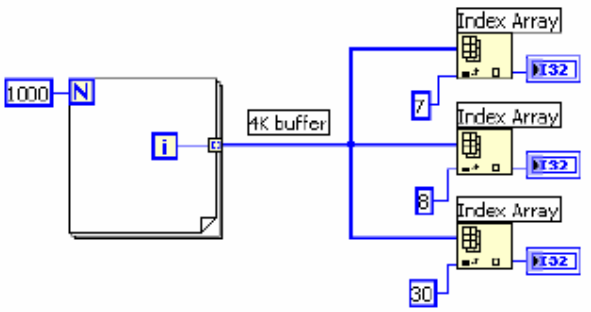
Входной и выходной массивы содержат одинаковое число элементов, и одинаковый тип данных. Использование компилятором входного массива в виде буфера данных позволяет вместо создания дополнительного буфера для выходных данных повторно использовать входной буфер. Такой подход экономит память и приводит к повышению производительности.

Однако компилятор не может использовать входные массивы в качестве буферов памяти во всех случаях. Это показано на примерах следующих блок-схем.



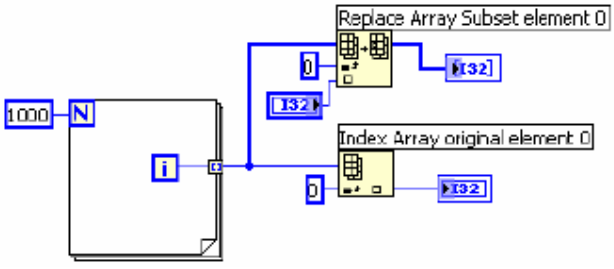
В показанной выше схеме сигнал источника данных идет в нескольких направлениях. Функции Replace Array Subset (замена подмножества массива) преобразует входной массив в выходной. Только для двух функций компилятор создает новые буферы данных и копирует в них вычисленные массивы, третья функция использует для результата входной массив. Эта блок-схема занимает около 12 Кб памяти: 4 Кб для исходного массива (1000 слов по 4 байта (Integer 32)) и 4 Кб x 2 для двух дополнительных буферов данных.

Теперь рассмотрим следующую блок-схему.



Как и прежде, вход направляется к трем функциям. Однако, в этом примере используются функции Index Array (индексный элемент массива) которые не изменяют входной массив а выдают на выход заданный элемент входного массива. LabView не создает копию данных которые передаются в несколько мест если они читаются без изменения. Эта блок-схема использует около 4 Кб памяти.

Наконец, рассмотрим следующую блок-схему.



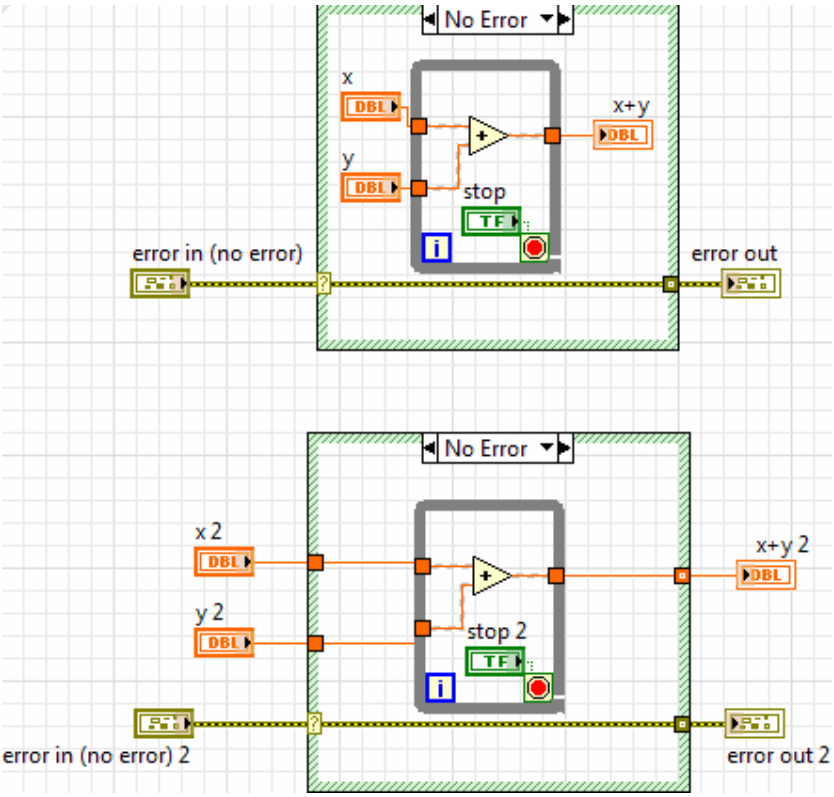
Здесь вход поступает к двум функциям, одна из которых изменяет данные. Между функциями нет связи и можно предположить, что, по крайней мере, одна копия должна быть сделана для того, чтобы подмножество могло заменить элементы массива. Однако, компилятор задает такой порядок, что сначала выполняется функция, которая выбирает заданный элемент данных, а только затем выполняется функция, которая изменяет данные. Таким образом замена элементов массива подмножеством выполняется без создания дублирующего массива. Если порядок выполнения узлов важен, это необходимо указать явно с помощью последовательности в которой выход одного узла идет на вход другого.

На практике, анализ блок-схемы компилятором не является совершенным. В некоторых случаях, компилятор не может определить оптимальный способ использования памяти блок-схемы.

Условные индикаторы и буферы данных

Повторное использование буферов данных зависит от варианта построения блок-диаграммы. Применение условного индикатора в SubVI не позволяет LabView использовать буфер данных. Условным индикатором называется индикатор внутри Case структуры или For цикла. При размещении индикатора в блоке выполняемом по условию LabView копирует все данные в индикатор. Когда индикатор размещен за пределами Case структуры и For цикла, LabView изменяет данные внутри цикла или структуры а результаты передает на индикатор не создавая копии данных.

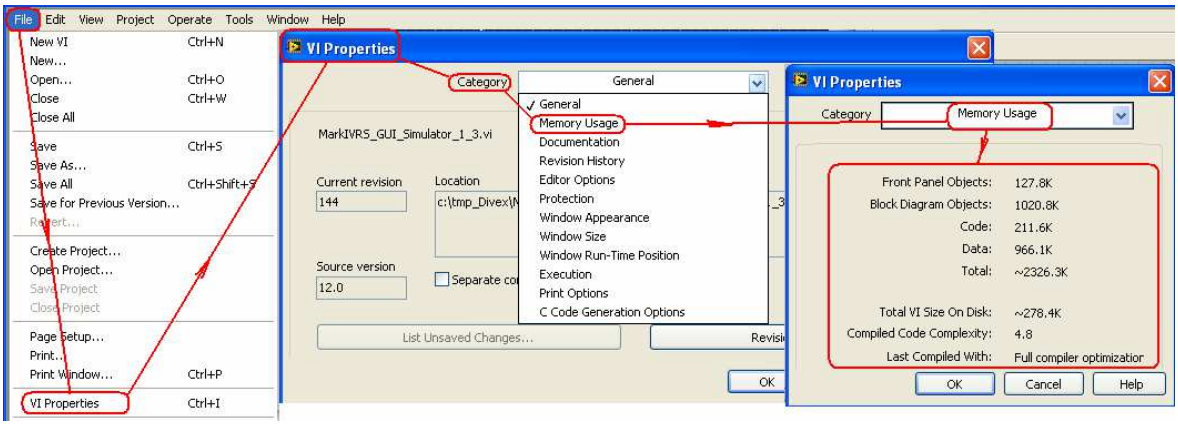
В следующем примере верхняя структура менее эффективна поскольку содержит три условных DBL объекта.



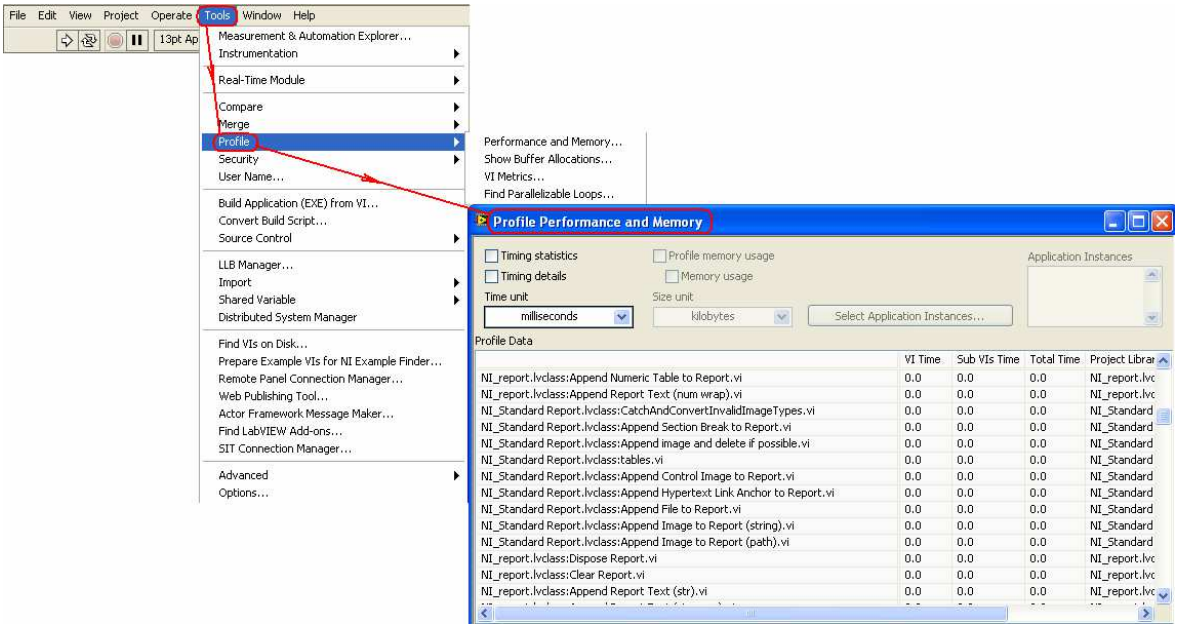
Мониторинг используемой памяти

Существует несколько методов определения размера используемой памяти.

Чтобы увидеть сколько памяти используется текущим VP, выберите File > VI Properties > и Memory usage из верхнего выпадающего меню. Обратите внимание, что в ответе нет информации о том сколько памяти используют SubVI.



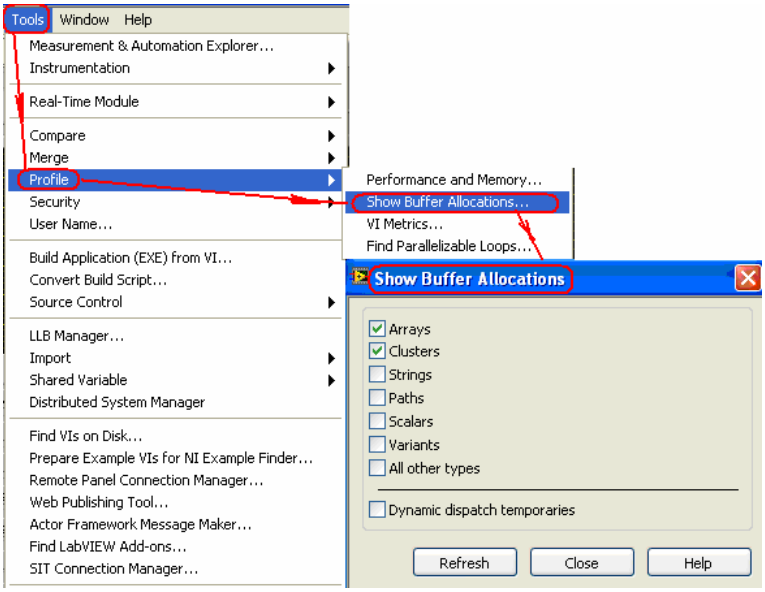
Можно использовать окно Tools > Profile > Performance and Memory для наблюдения за памятью используемой всеми VP находящимися в памяти.



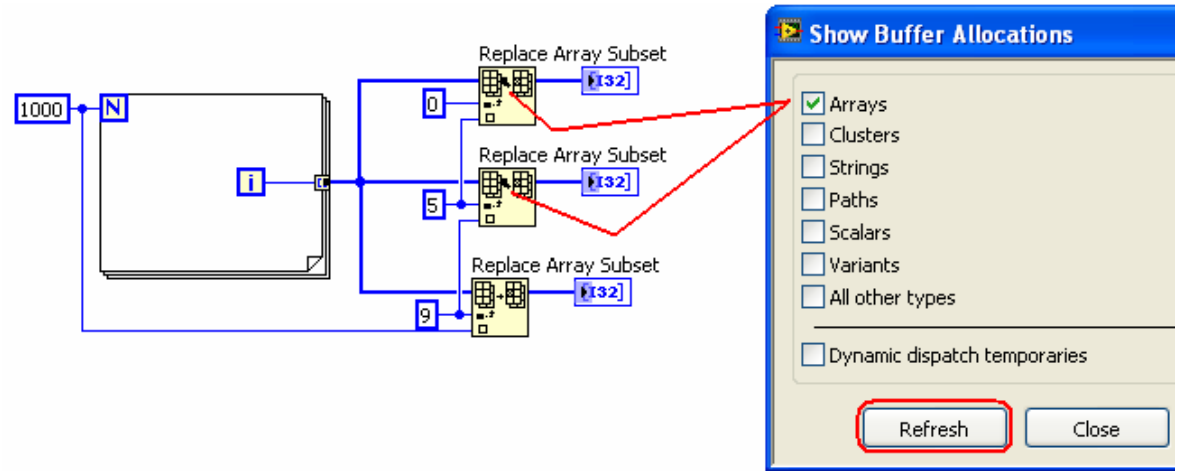
В окне ведется статистика по минимальному, максимальному и среднему количеству байт и блоков, используемых каждым VP в процессе выполнения.

Примечание. Перед наблюдением за используемой памятью VP, не забывайте сохранять VP. Иначе созданные временные копии объектов и данных могут увеличить память VP. Сохранение VP очищает порожденные Undo копии, и тем самым увеличивает точность информации о использовании памяти.

Можно использовать окно Tools > Profile > Performance and Memory, чтобы определить SubVI по производительности, а затем использовать Tools > Profile > Show Buffer Allocations, чтобы увидеть конкретные области на блок-схеме которым LabView выделяет память установите флажок рядом с соответствующим типом данных и нажмите кнопку Refresh (обновить):



Обратите внимание, что после Refresh на блок-схеме появляются черные квадратные метки указывающие, где на блок-схеме LabView создает буфер для хранения данных. Например, только для двух блоков из трех созданы буферы на следующей схеме.



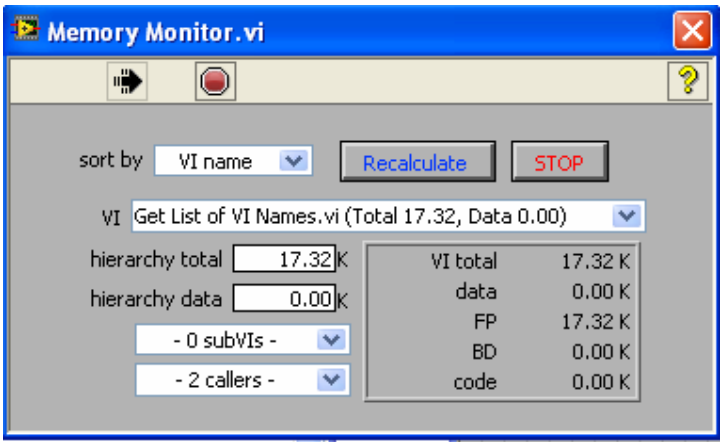
Примечание. Окно Show Buffer Allocations доступно только в LabView Full and Professional Development Systems.

Зная где LabView создает буферы, можно отредактировать VP с целью уменьшения памяти необходимой для работы VP. Все буферы устранить нельзя, поскольку часть из них используется для запуска VP.

Чтобы увидеть общее количество памяти, используемое приложением выберите Help > About LabView:

Сообщение показывает размер памяти виртуальных приборов, а также размер памяти используемой приложением. Можно проверить эту сумму до и после выполнения приборов, чтобы получить примерное представление о том, сколько памяти используют виртуальные приборы.

Также можно определить используемую память с помощью Memory Monitor VI в labview\examples\memmon.llb. Этот VP использует функции сервера VP, чтобы определить размер памяти всех загруженных VP. Вот его интерфейс.

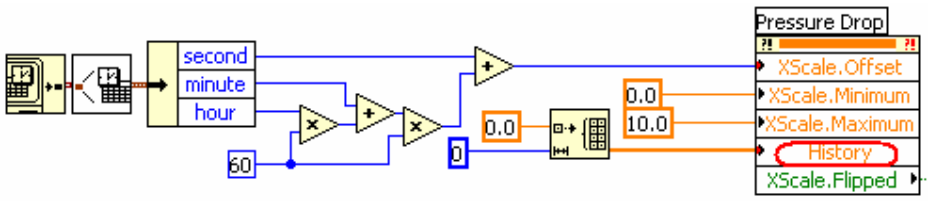


Правила по улучшению использования памяти

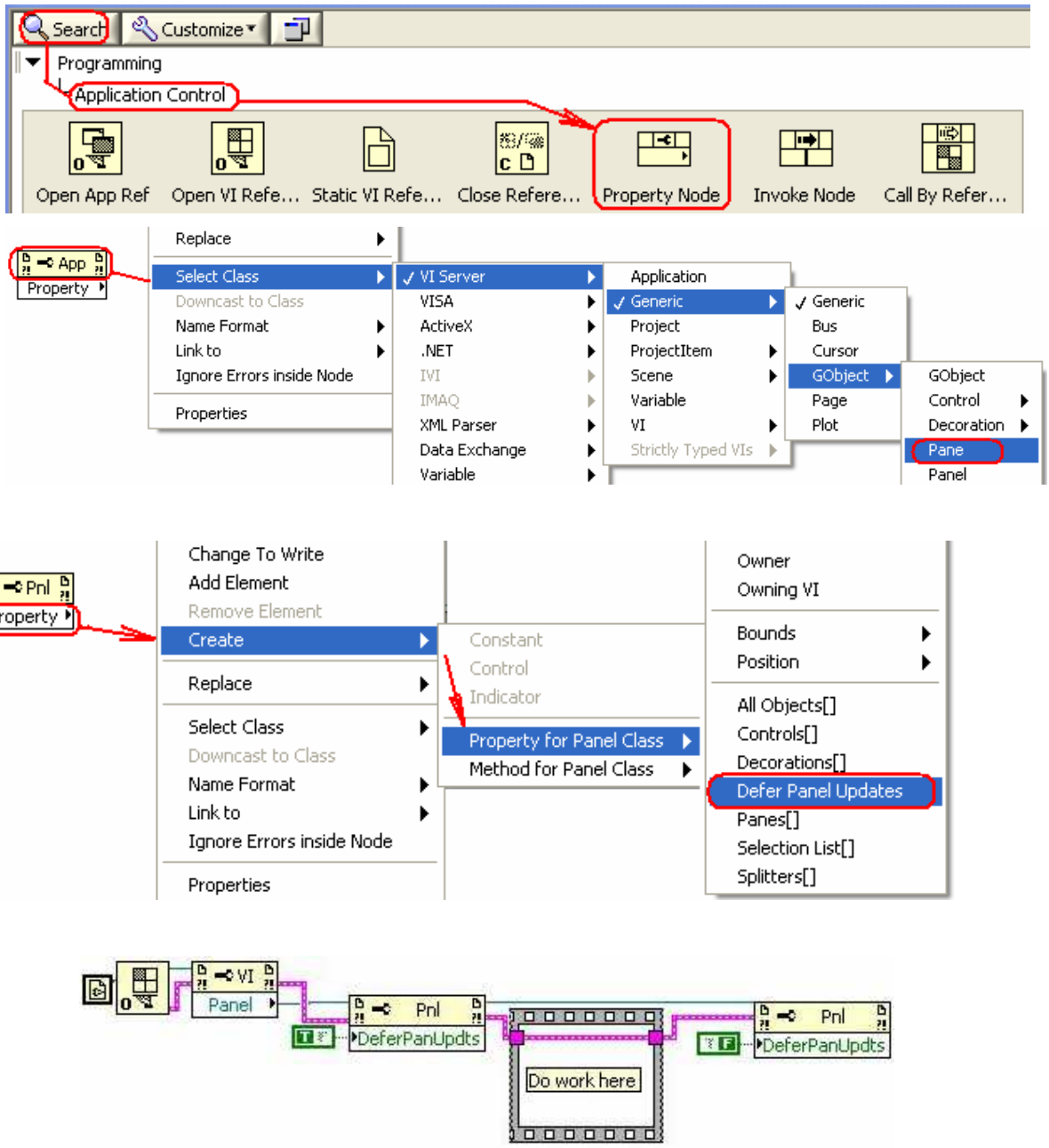
Выше показано, что компилятор пытается повторно использовать память. Правила использования памяти компилятором сложны. Следующие замечания помогают на практике создать VP с эффективным использованием памяти:

- Разбиение VP на SubVI как правило, не вредит памяти. Во многих случаях, использование памяти улучшается, потому что работающая система может использовать память данных SubVI, когда SubVI не выполняется.
- Слишком большое количество копий скалярных величин увеличивает память.
- Не злоупотребляйте глобальными и локальными переменными при работе с массивами и строками. Чтение глобальной или локальной переменной в LabView вызывает создание копии данных.
- Не выводите большие массивы и строки на открытую переднюю панель, если в этом нет необходимости. Органы управления и индикаторы открытой передней панели сохраняют копию данных, которые они отображают.

  Совет. Не используйте графопостроители без необходимости, помним, что они сохраняют копии отображаемых данных. Увеличение памяти продолжается, до тех пор пока буфер графика не заполнится. LabView автоматически не очищает историю графика при перезагрузке VP. Для очистки графика следует записать в него пустые массивы с соответствующим атрибутом, например, как показано ниже.



- Используйте свойство Defer Panel Updates. Когда это свойство TRUE, значения индикатора передней панели не изменяются. Установка управлением индикатора и пример использования показаны на следующих рисунках.



Примечание LabView обычно не открывает переднюю панель SubVI перед его вызовом.

- Если передняя панель SubVI не должна отображаться, не оставляйте неиспользуемые Property Nodes на SubVI поскольку они удерживают переднюю панель SubVI в памяти, а это может привести к ненужному перерасходу памяти.
- При проектировании блок-схемы, следите за местами, где размер входных данных отличается от размеров данных выхода. Например, увеличение размера массива или строки с помощью функций Build Array или Concatenate Strings приводит к появлению копии данных.
- Используйте соразмерные типы данных для массивов и следите за точками передачи данных SubVI и функциям. При изменении типа данных, система создает копию данных.

- Не используйте сложные, иерархические типы данных, такие как массивы кластеров или кластеры, содержащие большие массивы или строки.
- Не используйте очевидные и дублирующие объекты на передней панели, если в этом нет необходимости. Такие объекты могут использовать дополнительную память.

Особенности памяти передней панели

Когда передняя панель открыта, элементы ввода данных и индикаторы имеют свои собственные копии данных, которые они отображают.

Следующая блок-схема показывает функцию увеличения на единицу вводимых с передней панели данных и индикатор.

При запуске VP, данные передней панели управления передаются блок-схеме. Функция приращения повторно использует входной буфер. Затем индикатор делает копию данных для отображения. В результате используются три копии буфера.

Это сделано для того, чтобы индикаторы могли надежно отображать предыдущие значения пока не получат новые.

Для элементов ввода данных и индикаторов SubVI исполнительная система делает копию данных в следующих случаях:

- Передняя панель располагается в памяти. Это может произойти по одной из следующих причин:
  - Передняя панель открыта.
  - VP был изменен, но не сохранен (все компоненты VP остаются в памяти до сохранения VP).
- Панель используется для печати данных (File > Print, File > Print Window).
- Блок-схема использует Property Nodes (см. Рис 1).
- VP использует локальные переменные (см. Рис 2).
- Панель использует логирование данных (меню >Operate>Data Logging).
- Управление использует проверку приостановленных данных (см. Suspend when called в окне SubVI Node Setup ниже).

Примечание: Локальная переменная (например, Numeric 2) создается щелчком правой кнопкой мыши по объекту передней панели или его терминалу на блок-схеме, и далее, как показано на рисунке ниже.

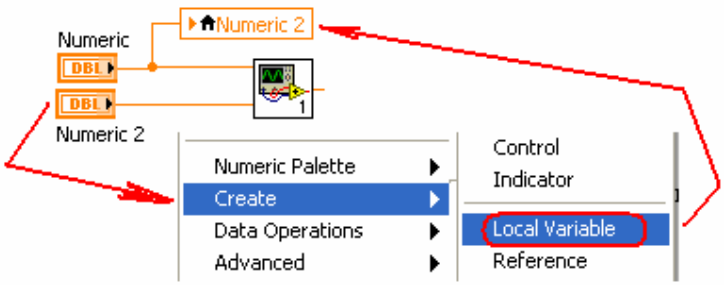


Рис 2. Пример создания узла локальной переменной объекта передней панели.

Открытие передней панели SubVI и приостановку его выполнения (например, с целью изменения параметров SubVI) можно задать в окне Setup открываемом нажатием правой клавишей мыши на значке SubVI (см. **Рис 3**).

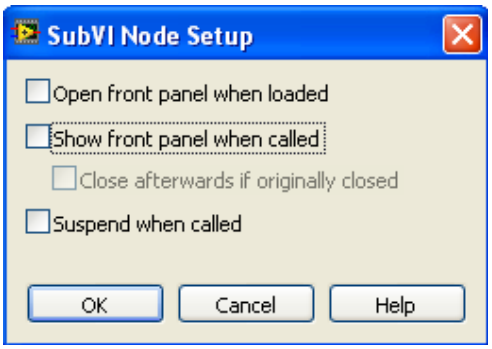


Рис 3. Функции управления SubVI.

Чтобы настроить SubVI на отображение передней панели при запуске и ее удаление из памяти после выполнения SubVI необходимо установить “Show front panel when called” и “Close afterwards if originally closed” в окне Setup (см. **Рис 3**)..

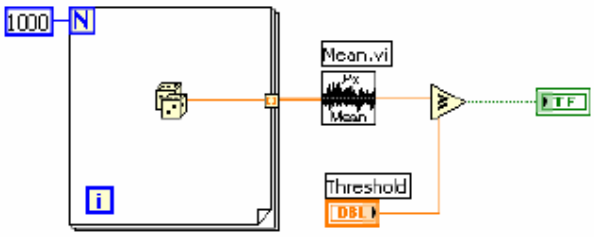
Узел Property Node позволяет считывать данные объектов ввода / вывода закрытой панели SubVI и управлять многочисленными свойствами объектов поэтому выполняющая система сохраняет панели SubVI в памяти, если SubVI использует Property Nodes.

SubVI может повторно использовать память данных

В большинстве случаев память не увеличивается, если часть блок-схемы переводится в SubVI поскольку SubVI используют буферы данных также как и схемы верхнего уровня. Случаи использования SubVI дополнительной памяти перечислены выше в разделе “Особенности памяти передней панели**”.**

Понимание ситуаций, когда память не высвобождается

Рассмотрим следующую блок-схему.



После вычисления среднего значения массив данных больше не нужен. Но, поскольку определение того, когда данные больше не нужны может стать очень сложной задачей в больших блок-схемах, буферы данных конкретного VP не высвобождаются во время его исполнения.

Рассмотрим ту же VP в качестве SubVI. Массив данных создается и используется только в SubVI. На Macintosh, если SubVI не выполняется и системе не хватает памяти, операционная среда может высвободить память данных SubVI. Это тот случай, когда использование SubVI вместо схемы верхнего уровня может сэкономить память.

В Windows и Linux, буферы данных обычно не высвобождаются когда VP закрывается и удаляется из памяти.

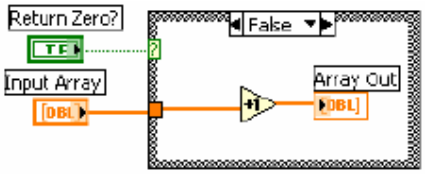
Можно использовать функцию Request Deallocation 

для высвобождения неиспользуемой памяти. Высвобождение памяти происходит только после выполнения VP. Функцию целесообразно применять когда VP создает большое количество данных которые не используются повторно.

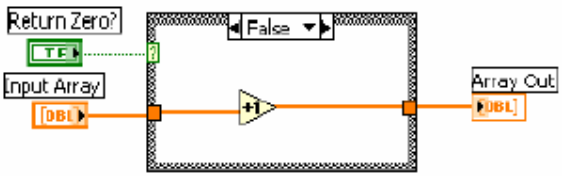
Когда выходы могут использовать входные буферы

Если выход имеет тот же размер и тип данных как и вход, и данные входа не требуется в другом месте, то выход может повторно использовать буфер ввода. Как упоминалось ранее, в некоторых случаях, даже если вход используется в другом месте его буфер может использоваться для выходного буфера.

Можно использовать окно “Меню > Tools > Profile > Show Buffer Allocations”, чтобы увидеть, использует ли выходной буфер входной буфер. В следующей блок-схеме, размещение индикаторов внутри страницы Case структуры вынуждает LabView создавать копию данных для индикатора, при этом выходные буферы не используют входной массив.



Если переместить индикатор за пределы Case структуры (см. следующий рисунок), выходной буфер начнет использовать данные входного массива (функция приращения изменяет входной массив и передает его в выходной массив). В этом случае дополнительный буфер не создается.



Использование подобных типов данных

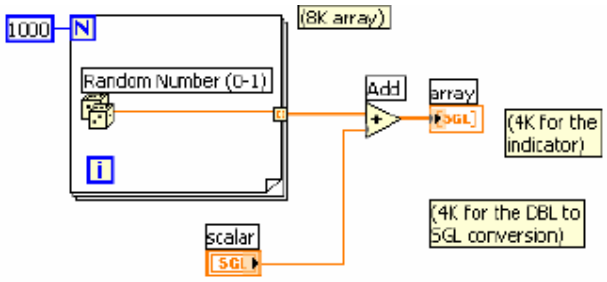
Если вход имеет другой тип данных выход не может повторно использовать этот вход. Например, при добавлении 32-разрядного числа к 16-разрядному целому последнее преобразуется в 32-разрядное число. В этой операции выход может использовать только входной 32-разрядный буфер если он не используется повторно в другом месте. Но в общем случае, компилятор создает новый буфер для преобразованных данных.

Чтобы свести к минимуму использование памяти, используйте подобные типы данных везде где это возможно. Это уменьшает количество копий данных. Использование подобного типа данных позволяет компилятору более гибко определять когда буферы данных можно использовать повторно.

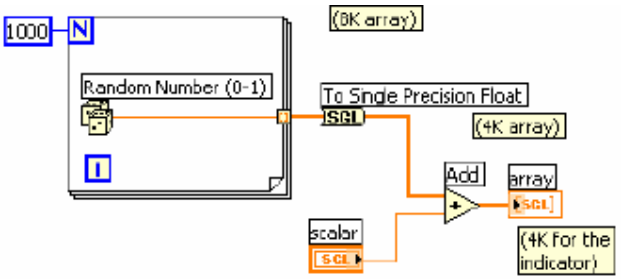
В некоторых приложениях, следует оценить возможность использования меньших типов данных, например, использование чисел одинарной точности из четырех байт, вместо чисел двойной точности из восьми байт. Однако, старайтесь избегать ненужных преобразований.

Как создавать данные требуемого типа

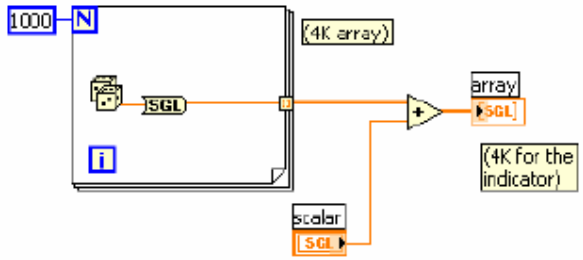
В следующем примере создается массив из 1000 случайных чисел к которому добавляется скалярная величина. Преобразование типа в функции Add используется потому, что случайные числа имеют двойную точность (double, 64 bits), в то время как скаляр имеет одинарную точность (single, 32 bits). Скаляр увеличивается до двойной точности. Полученные данные затем передаются на индикатор. Эта блок-схема использует 16 Кб памяти.



Следующая блок-схема показывает неправильные попытки минимизации памяти путем преобразования массив случайных чисел двойной точности в массив случайных чисел одинарной точности. Схема по-прежнему использует тот же объем памяти – 16Кб, как и в предыдущем примере.



Лучшее решение (8Кб используемой памяти) показано на следующей блок-схеме. Здесь случайные числа преобразуются к одинарной точности до создания массива. Это позволяет избежать преобразования больших массивов данных из одного типа в другой.



Избегайте частого изменения длины данных

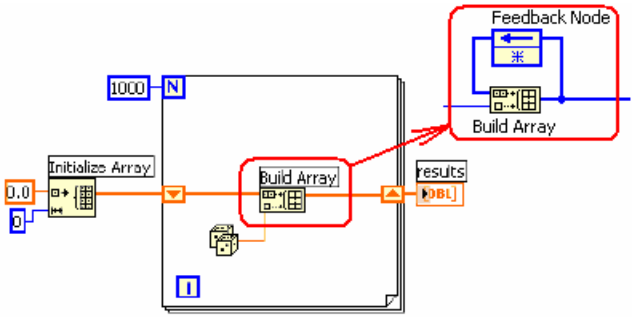
Если размер выхода отличается от размера входа, выход не использует повторно буфер входных данных. Это касается таких функций, как Build Array, Concatenate Strings, и Array Subset которые увеличивают или уменьшают размер массива или строки. При работе с массивами и строками, избегайте частого использования этих функций, потому что с ними программа использует больше данных и памяти и выполняется медленнее, из-за постоянного копирования данных.

Пример 1: Создание массивов

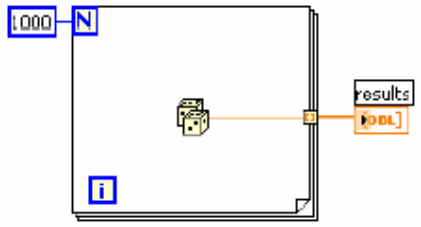
Рассмотрим следующую блок-схему которая создает массив данных в цикле, постоянно вызывая Build Array для присоединения нового элемента. На каждой итерации VP

изменяет размер буфера и добавляет новый элемент. В результате скорость выполнения замедляется, особенно если цикл выполняется многократно.

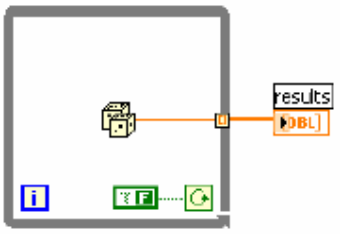
Внимание. При работе с элементами массивов, кластеров, переменных и сигналов для оптимизации памяти и повышения эффективности целесообразно использовать структуру Search > Programming > Structures > In Place Element  которая при помощи компилятора оптимизирует блоки находящиеся внутри ее зоны.



Добавление значения в массив на каждой итерации можно обеспечить автоматической индексацией цикла. Такой прием повышает производительность. С циклом For, VP назначает размер массива (в зависимости от значения N) и размер буфера только один раз.



В цикле While, автоматическая индексация не столь эффективна, потому что не известен конечный размер массива. Однако, цикл While автоматического индексирования изменяет размер с большими приращениями и делает это соответственно не на каждой итерации. Когда цикл закончен, выходной массив уменьшается до нужного размера. Производительность циклов While и For автоматической индексации практически идентична.

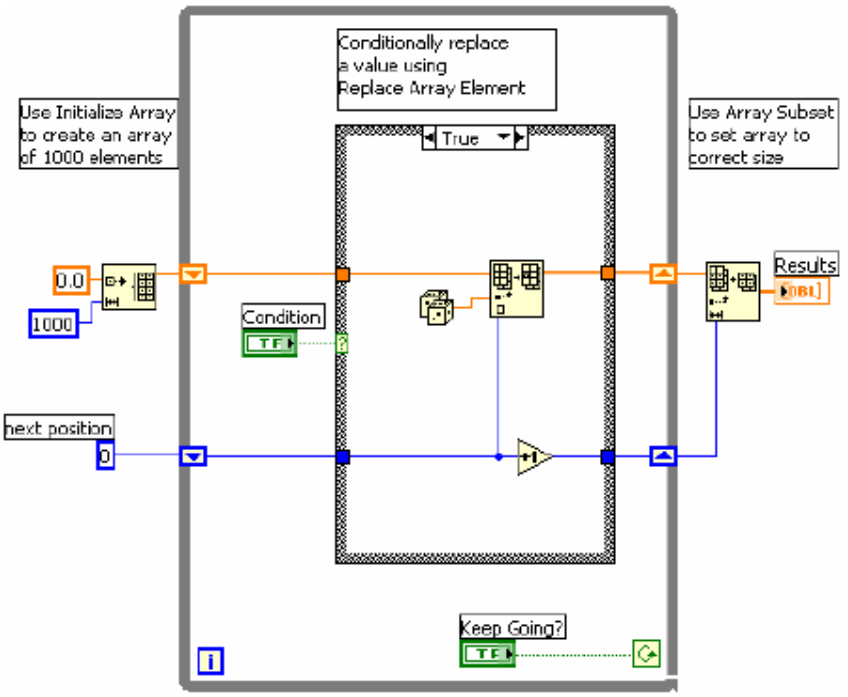


Автоматическая индексация добавляет значение в результирующий массив на каждой итерации цикла. Задав верхний предел размера массива можно использовать функцию Replace Array Subset для добавления значений в массив.

Когда заканчивается заполнение массива его можно изменить до нужного размера. Массив создается только один раз, и Replace Array Subset может использовать буфер ввода для выходного буфера. Производительность этого варианта близка к выполнению циклов при помощи автоматической индексации. При использовании этого метода, будьте

внимательны при выборе начального размера массива поскольку функция Replace Array Subset не меняет размер массива.

Пример использования Replace Array Subset показан на следующей блок-схеме. Здесь сначала задается увеличенный размер массива - 1000 элементов (Initialize Array), а после остановки циклической записи размер массива уменьшается до величины равной количеству циклов. Уменьшение размера выполняется за пределами цикла в блоке Array Subset.

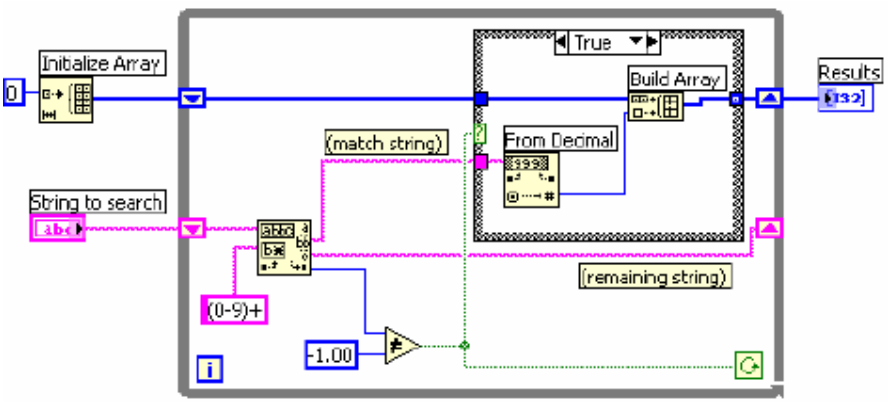


Пример 2: Поиск в строке

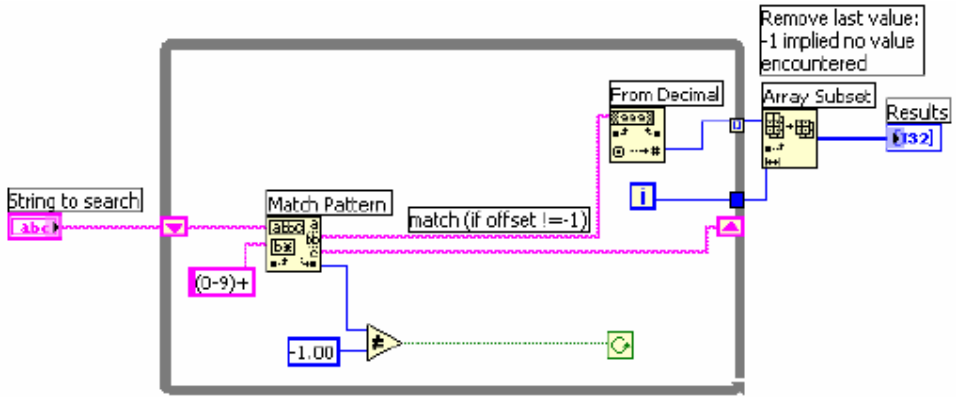
Для поиска строки по шаблону можно использовать функцию Match Pattern. При неправильном использовании функции можно потерять производительность даже если буферы строковых данных не создаются.

Предположим, что необходимо найти целые числа в строке, для этого можно использовать [0-9] + как регулярное выражение входа в эту функцию. Чтобы создать массив из всех целых чисел строки, используется цикл и вызывается блок Match Regular Expression до тех пор пока блок не выдаст -1.

Следующая блок-схема является одним из вариантов сканирования всех вхождений числа в строку. Она создает пустой массив, а затем ищет в оставшейся строке цифровой шаблон на каждой итерации цикла. Если шаблон найден Build Array добавляет найденное число в результирующий массив. Когда в исходной строке не остаётся значений, блок Match Regular Expression возвращает -1 и блок-схема завершает выполнение.



Проблемой этой блок-схемы является то, что она использует сборку массива в цикле для объединения нового значения с предыдущими. Вместо того, чтобы накапливать значения можно использовать автоматическое индексирование. Обратите внимание, что накопление заканчивается с последней итерацией цикла, когда Match Regular Expression не может найти соответствия. Решение заключается в использовании функции Array Subset для удаления нежелательной дополнительной величины массива. Это показано на следующей блок-схеме.

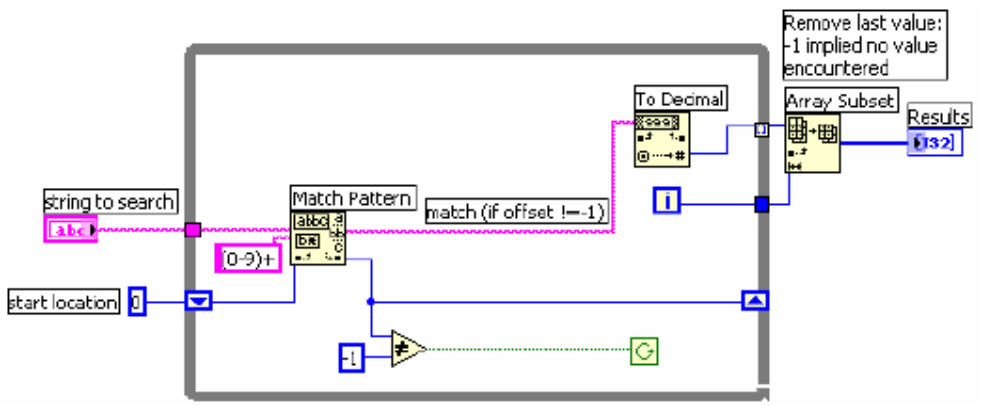


Другая проблема этой блок-схемы - создание ненужной копии остатка строки в каждом цикле. Match Pattern имеет вход который можно использовать, для указания начала поиска. Если запоминать смещение предыдущей итерации, его можно использовать как начало следующей итерации. Для этого (см. следующий рис.)

- подключите константу “ноль”, находящуюся за пределами цикла, ко входу блока Match Pattern;
- из списка свойств входа в цикл выберите Replace with Shift Register: 

(в результате получите вход и выход);
- подключите к Shift register ( ) выход Offset блока Match Pattern.

Теперь на каждой новой итерации поиск начнется с позиции найденного элемента.

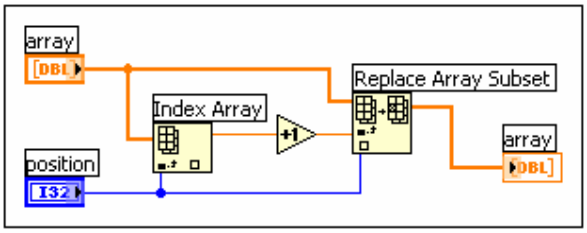


Разработка эффективных структур данных

Ранее было отмечено, что иерархические структуры данных, такие как кластеры или массивы кластеров, содержащие большие массивы чисел или строки, не могут обрабатываться эффективно. Этот раздел объясняет, почему это так, и дает рекомендации по выбору более эффективных типов данных.

Проблема со сложными структурами данных состоит в том, что трудно обеспечить доступ к элементам и их изменение в структуре данных без создания копии элементов. Если эти элементы большие то для дополнительного копирования используется больше и памяти и времени.

Однако можно манипулировать скалярными типами данных и очень эффективно. Кроме того, можно эффективно управлять небольшими строками и массивами в которых элемент является скалярной величиной. В случае массива скаляров, следующий пример показывает как увеличить значение элемента в массиве.

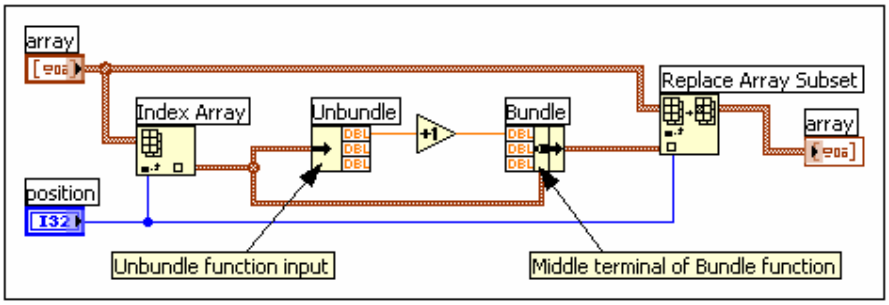


Внимание. При работе с массивами, кластерами, сигналами (waveform) и переменными используйте структуру Search > Programming > Structures > In Place Element  которая улучшает использования памяти в VP.

Это очень эффективный приём, так как нет необходимости создавать дополнительные копии общего массива. Кроме того, поскольку элемент производимый функцией Index Array является скалярным, он может управляться эффективно.

То же самое касается массива кластеров, предполагая, что кластер содержит только скаляры. В следующей блок-схеме, манипуляция с элементами становится немного сложнее, потому что используются Unbundle и Bundle функции. Однако, поскольку кластер, вероятно, невелик (скаляр использует очень мало памяти), нет никаких значительных издержек, связанных с доступом элементов к кластеру и заменой элементов в исходном кластере.

Следующая блок-схема показывает эффективную модель разделения (unbundling), преобразования и объединения (rebundling) данных. Канал от источника данных должен иметь только два направления – ко входу функции Unbundle, и к среднему терминалу функции Bundle. LabView понимает эту модель и способен генерировать более эффективные код.

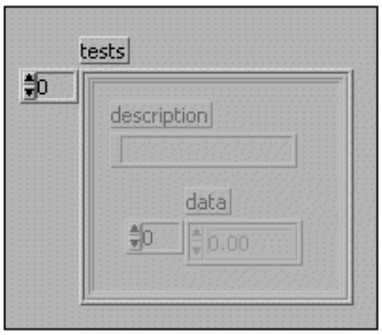


Если имеется массив кластеров, где каждый кластер содержит большие суб-массивы или строки, индексация и изменение значений элементов в кластере может занять значительно больше памяти и времени выполнения.

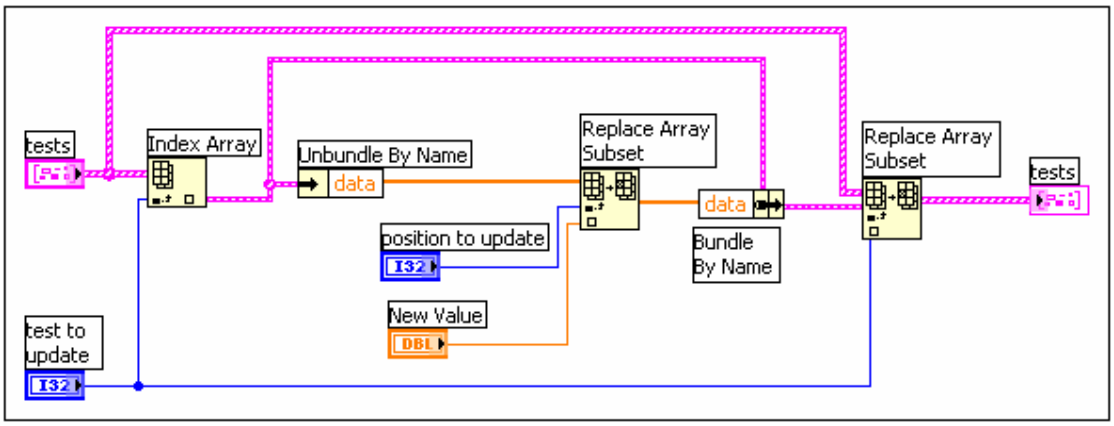
Решение этой проблемы состоит в том, чтобы найти альтернативные представления данных. Следующие три примера содержат предложения по улучшению структуры данных в каждом конкретном случае.

Пример 1: Как избежать сложных типов данных

Рассмотрим пример, в котором необходимо записать результаты нескольких тестов в строку, включающую описание испытания и массив результатов. Пример типа данных для хранения этой информации показан на следующей передней панели.

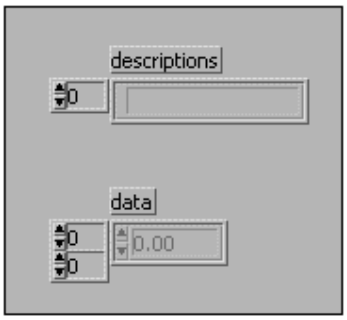


Чтобы изменить элемент массива, необходимо указать на элемент общего массива – номер тестовой записи. Далее для этого кластера необходимо выполнить unbundle чтобы использовать элемент массива. Затем следует заменить элемент массива и сохранить полученный массив в кластере. И, наконец, сохранить полученные кластеры в исходном массиве. Пример этого показан на следующей блок-схеме.

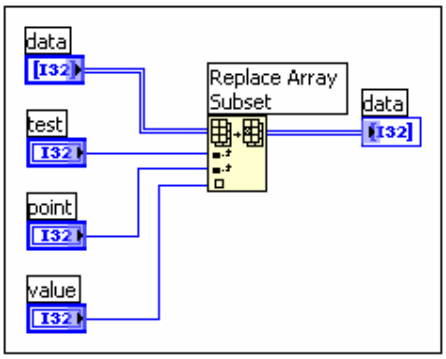


Каждый уровень разделения/индексации (unbundling/indexing) может привести к копированию создаваемых данных, а это требует времени и занимает память. Решение проблемы состоит в создании как можно более плоской структуры данных. Например, в этом случае структура данные разбивается на два массива. Первый массив является

массивом строк. Второй массив – двумерный (2D) массив, где каждая строка является результатом данного теста. Этот вариант показан на следующей передней панели.



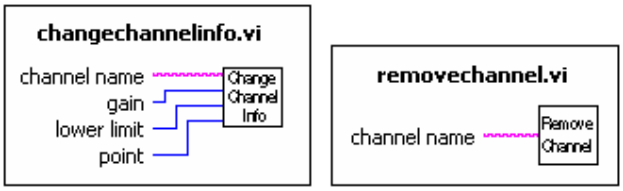
В такой структуре данных можно заменить элемент массива напрямую, используя функцию Replace Array Subset, как показано на следующей блок-схеме.



Пример 2: Глобальная таблица разнотипных данных

Вот еще приложение в котором требуется сохранить информацию в табличном виде. В нем хотелось бы обеспечить глобальный доступ к данным. Эта таблица может содержать настройки инструмента, в том числе усиление, нижний и верхний пределы напряжения и имя канала.

Чтобы сделать данные доступными в приложении, рассмотрим возможность создания множества VP для доступа к табличным данным, например, как в следующем VP: Change Channel Info и Remove Channel Info.



В следующих разделах представлены три реализации этих VP.

Очевидная реализация

Имеется несколько структур данных позволяющих работать с базовой таблицей. Во-первых, можно использовать глобальную переменную, содержащую массив кластеров, где каждый кластер содержит коэффициент усиления, нижний предел, верхний предел и имя канала.

Как описано в предыдущем разделе, этой структурой данных трудно эффективно управлять, потому, что обычно следует выполнить несколько уровней индексации и разделения доступа к данным. Кроме того, функцию Search 1D Array можно использовать только для поиска конкретного кластера в массиве кластеров, но нельзя использовать его для поиска элементов кластера.

Альтернативная реализация 1

Как и в предыдущем случае, данные сохраняются в двух отдельных массивах. Один из них содержит имена каналов. Другой содержит данные канала. Индекс имени канала в массив имен используется для поиска соответствующих данных канала в другом массиве.

Заметьте, что поскольку массив строк отделён от данных, можно использовать функцию Search 1D Array.

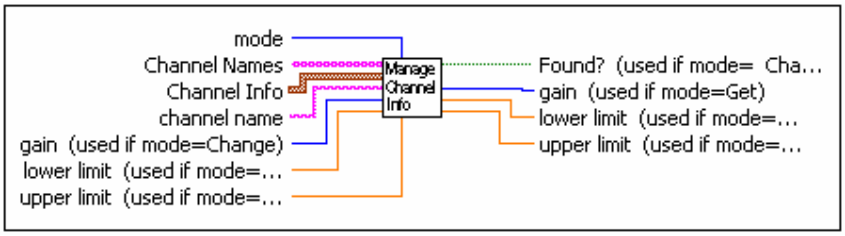
На практике, если массив из 1000 каналов создаётся с помощью Change Channel Info VI, это выполняется примерно в два раза быстрее, чем в предыдущей версии. Но эта версия содержит и недостатки.

При чтении глобальной переменной создается копия данных глобальной переменной. Таким образом, при обращении к элементу каждый раз генерируется полная копия данных массива. Следующий метод показывает как избежать этих нежелательных расходов и повысить эффективность.

Альтернативная реализация 2

Существует альтернативный способ сохранения глобальных данных с использованием неинициализированного регистра сдвига. Если начальное значение не подключено к регистру сдвига он сохраняет свое значение между вызовами.

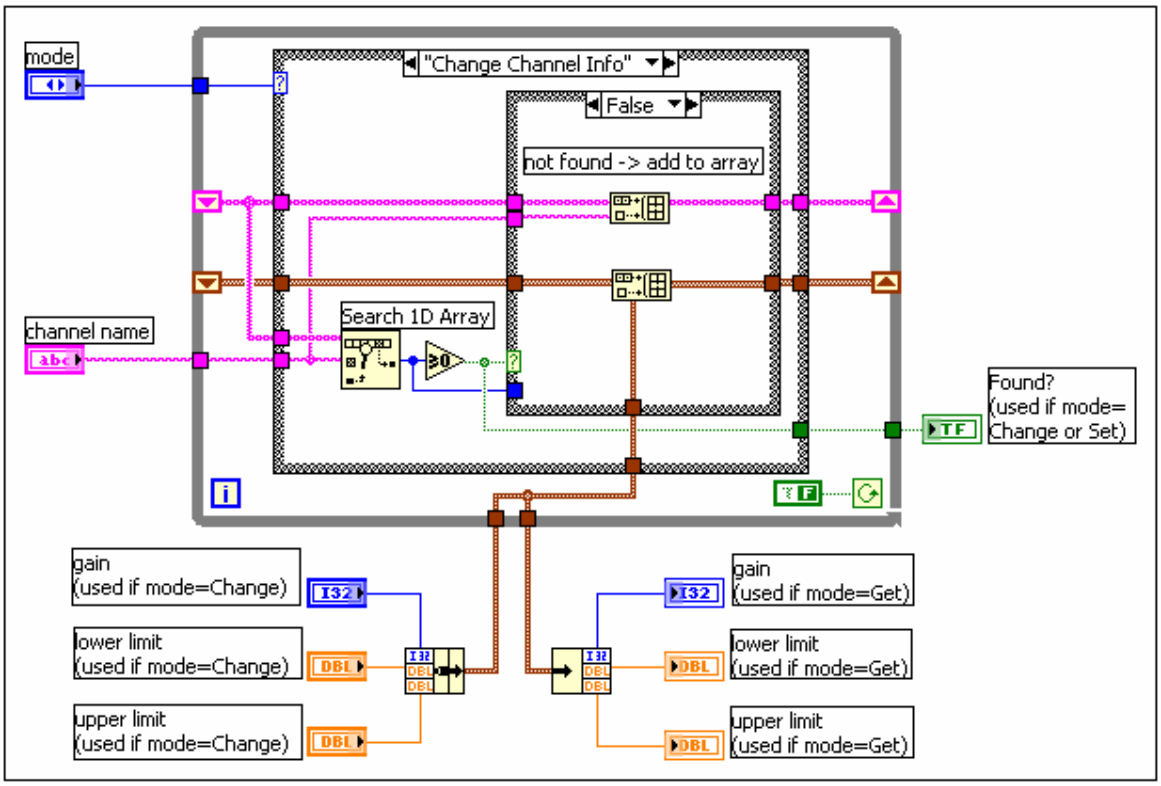
Компилятор LabView эффективно обрабатывает доступ к регистрам сдвига. Чтение значения регистра сдвига не обязательно создает копию данных. В самом деле, можно индексировать массив хранящийся в регистре сдвига и даже изменять и обновлять его значения без создания дополнительных копий общих массивов.



Можно создать один SubVI который в зависимости от входного режима определяет, требуется ли читать, изменить или удалить канал, или обнулить данные по всем каналам.

SubVI содержит While цикл с двумя регистрами сдвига – данными каналов, и названиями каналов. Ни один из этих регистров сдвига не инициализируется. Внутри While цикла необходимо разместить Case структуру подключенную ко вводу режима. В зависимости от значения режима, можно прочитать и изменить данные в регистре сдвига.

Ниже приведена схема SubVI с интерфейсом, в которой реализованы все три режима. Показан только код Change Channel Info.

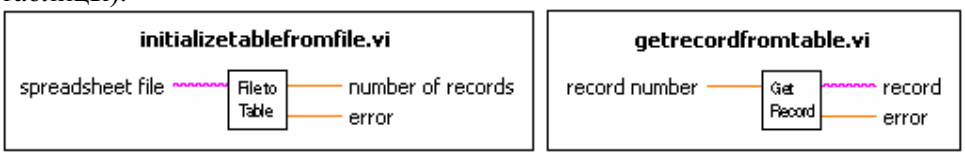


Для 1000 элементов эта схема работает в два раза быстрее чем предыдущая и в четыре раза быстрее чем начальная реализация.

Пример 3: Глобальная таблица записей

В предыдущих примерах таблицы содержали смешанные типы данных, таблицы могли часто изменяться. Однако во многих случаях информационная таблица мало меняется после создания. Таблица может быть считана, например, из файла электронной таблицы и затем, в основном, использоваться для поиска информации.

В этом случае, реализация может состоять из следующих двух функций Initialize Table From File (инициализация таблицы из файла) и Get Record From Table (получение записи таблицы).



Один из способов применения таблицы заключается в использовании двумерного массива строк. Обратите внимание, что компилятор сохраняет каждую строку в массиве строк в отдельном блоке памяти. Большое количество строк (например, более 5000 строк) может нагрузить менеджер памяти, что в свою очередь может привести к заметной потере производительности.

Альтернативный метод для хранения большой таблицы - читать таблицу в виде одной строки. Затем построить отдельный массив, содержащий смещение каждой записи в строке. Это изменяет организацию, и вместо потенциальных тысяч относительно небольших блоков памяти используется один большой блок памяти (строка) и отдельные меньшие блоки памяти (массив смещений).

Этот метод может оказаться более сложным в реализации, но может стать и более производительным для больших таблиц.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Задание 1. Минимизация памяти при работе с массивами в LabView.

1. Постройте и разберите диаграммы раздела “Как создавать данные требуемого типа”.
2. Постройте блок-схемы и разберите все варианты использования памяти, приведенные выше в Примере 1: “Создание массивов”.
3. Постройте и разберите все блок-схемы, приведенные в Примере 2: “ Поиск в строке”

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Перечислите варианты определения размера памяти, занимаемой VP - Виртуальным Прибором и его объектами в LabView.
2. Покажите на примерах, как можно минимизировать память, занимаемую прибором LabView.
3. Приведите примеры повышения быстродействия вычисления объектов LabView.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. National Instruments. VI memory usage.
2. Enabling 3GB switch on Windows XP or Windows Vista http://dwf.blogs.com/beyond_the_paper/2009/04/enabling-3gb-switch-on-windows-vista.html
3. Dr. Bob Davidov. Компьютерные технологии управления в технических системах http://portalnp.ru/author/bobdavidov