VArrayLib 1.2 Reference Manual

Document Revision 1.2
August 9, 2001

The heapsort implementation is based on code by Michael Schürig and Kyle Hammond.

The author's home page is
http://www.blugs.com/

TO DO LIST

Pascal support.
Fibonaccian search? (Knuth)
More assembly language!
`VALock`
Get rid of `VAMove`?

Contents

Preface

About VArrayLib

VArrayLib is a compact library of routines for manipulating one-dimensional arrays of same-size data elements. All arrays are referenced by a handle to a relocatable memory block, but the external API treats these references as opaque. It is especially useful for building more sophisticated software on top of it. It serves as the basis for the Stockpile Manager, an open-source clone of Apple's Collection Manager.

VArrayLib is distributed in source form (in spite of the name); the code base is small enough that it is probably unnecessary to compile it into a static library. Future versions may become available as a shared library for CFM-based runtimes.

VArrayLib Features

- Fast sorting (nonrecursive heapsort) and searching (binary search).
- Opaque, reference-counted array objects.
- Portions written in PowerPC assembly language for maximum speed.
- Runs on any Mac OS system. (This means you, System 6 users.)
- Uses no global data whatsoever.
- Never allocates temporary memory.

Array References

Currently a `VARef` is a handle to a relocatable block of memory in your application's heap. Sometimes you need to know this: should you obtain a pointer to part of this block and call a routine that moves memory, the pointer may become stale. To keep this from happening you need to call `HLock` and coerce the array reference to a handle. Future versions may have a `VALock` routine, possibly with an internal "lock count" to enhance reentrancy.

All references to array item numbers are one-based. This convention is used so that item zero can be returned as an error or 'not found' value.

Development Environment

VArrayLib uses a tiny subset of the Macintosh toolbox. Only Memory Manager routines — and not many of them — are used. Consequently, VArrayLib should work on even the most primitive machines and operating system versions. You can probably get away with using truly ancient Macintosh header files in your development environment.

If you define the symbol `__VA_DEBUG__` prior to inclusion of `VArray.h`, the VArrayLib binaries will include `DebugStr` calls for error conditions. Running an application based on the debug version on a machine without a debugger will result in a system error.

VArrayLib is in general not compatible with interrupt-level processing, nor is it usable from within an MP (multiprocessing) task. Many VArrayLib routines allocate memory. If you are sneaky, you might be able to *access* array data at interrupt time. As long as your comparison callback does not move memory, sorting and searching should work fine at interrupt time.

VArrayLib does not use QuickDraw routines or globals, so it is safe to use in a daemon or FBA (faceless background application).

Chapter 1
VArrayLib API

This chapter details the VArrayLib API which your application uses to create and manipulate variable arrays. A variable array is a memory block that contains a small header, followed by a variable number of data items. All data items are the same size.

Types and Constants

This section describes the types and constants in VArrayLib. They are extracted from the header file `VArray.h`.

Variable Array

A variable array begins with header defined in the structure below, followed by data items. *This information is provided for reference only.* You should use accessor functions. The struct is not part of the external API. Because the header file `VArray.h` is referenced by `VArray.c`, the `__VA_INTERNAL__` flag keeps `VArray.c` from redefining `VARef` as an opaque object.

```
#ifndef __VA_INTERNAL__
typedef struct OpaqueVAReference*               VARef;
#endif

typedef struct
{
      UInt32      nItems;
      UInt32      itemSize;
      UInt32      refCon;
      UInt32      ownerCount;
      char        data[1];
} VARec, **VARef;
```

Field Descriptions

`nItems`	The number of array entries.
`itemSize`	The size in bytes of each array entry.
`refCon`	A reference value for use by the application.
`ownerCount`	The number of references to the array. When the count falls below 1, the array is deallocated.
`data`	`data[0]` is the first byte of item data.

Comparison Results

Your data comparison callbacks for sorting and searching should return appropriate results from the enumerations below. The first set of constants makes sense for searching; the second set is useful for sorting, the third set is used for iterating through the array. The constant `vaEqual` is also used for sorting.

```
typedef SInt32                VAComparisonResult;
enum
{
      vaSearchDataIsGreater   = -1L,
      vaEqual                 = 0,
      vaSearchDataIsLess      = 1,
      vaNotEqual              = 1
};

enum
{
      vaFirstIsLess           = -1L,
      vaFirstIsGreater        = 1
};

typedef UInt32                VAIteratorResult;
enum
{
      vaIteratorKeepGoing,
      vaIteratorStop
};
```

Constant Descriptions

`vaSearchDataIsGreater`	The search data should come after the data in the array item.
`vaEqual`	The search data matches the data in the array item, or the two array items are the same.
`vaSearchDataIsLess`	The search data should come before the data in the array item.
`vaNotEqual`	The search data does not match the data in the array item. This return value is only used in a brute-force search.
`vaFirstIsLess`	The first array item should come before the second.
`vaFirstIsGreater`	The first array item should come after the second.
`vaIteratorKeepGoing`	`VAIterate` should continue to call the iterator callback on array entries.
`vaIteratorStop`	`VAIterate` should return.

Result Codes

VArrayLib defines one user-defined `OSErr` code.

```
enum
{
      vaNoMovementNecessaryErr      = 2600
};
```

Constant Description

`vaNoMovementNecessaryErr`
: Returned by `VAValidateMoveParameters`, this code indicates the parameters are OK but would result in no net change to the array (such as moving an entry to where it already is).

Callback Routines

In order to allow VArrayLib to search, sort, and iterate over array items, you provide routines that evaluate array data in an appropriate way.

```
typedef VAComparisonResult (*VASearchProcPtr)
      ( const Ptr inArrayDataPtr,
      void* inSearchData,
      UInt32 inRefCon );

typedef VAComparisonResult (*VABruteForceSearchProcPtr)
      ( const Ptr inArrayDataPtr,
      void* inSearchData,
      UInt32 inRefCon );

typedef VAComparisonResult (*VASortProcPtr)
      ( const Ptr inData1Ptr,
      const Ptr inData2Ptr,
      UInt32 inRefCon );

typedef VAIteratorResult (*VAIteratorPtr)
      ( Ptr inArrayDataPtr, void* ioUserData );
```

Type Descriptions

`VASearchProcPtr`
: A routine that compares search data with item data in a sorted array. This kind of routine returns one of the constants `vaSearchDataIsGreater`, `vaSearchDataIsLess`, or `vaEqual`.

`VABruteForceSearchProcPtr`
: A routine that compares search data with item data in an unsorted array. It returns one of the constants `vaEqual` or `vaNotEqual`.

`VASearchProcPtr`
: A routine that compares data between two array items. It returns one of the constants `vaFirstIsLess`, `vaFirstIsGreater`, or `vaEqual`.

`VAIteratorPtr`
: A routine that can do whatever it wants with the data in each array item. It returns one of the constants `vaIteratorKeepGoing` or `vaIteratorStop`.

VArrayLib Routines

This section describes all routines in the VArrayLib API.

Creating and Disposing of Arrays

Use the `VANew` function to create a variable array. Use the `VADispose` procedure to decrement the array's owner count and/or release memory associated with the array.

VANew

Creates a variable array.

```
VARef VANew( UInt32 inItemCount, UInt32 inItemSize,
             Boolean inZero, UInt32 inRefCon )
```

`inItemCount`	The number of data items the array is to hold initially.
`inItemSize`	The size, in bytes, of each data element.
`inZero`	If `true`, every byte of the data storage area is set to zero. If false, the memory block is left uninitialized.
`inRefCon`	A reference value that you can use any way you like.

`VANew` allocates a new variable array and returns a reference (handle) to it. The array's owner count is one. If there is not enough free memory to allocate the array, or if you pass zero for `inItemSize`, `VANew` returns `nil`. You can create an empty array (by passing zero for `inItemCount`) and fill it later.

VADispose

Decrements an array's owner count.

```
void VADispose( VARef inArray )
```

`inArray`	The array to be disposed of.

Assuming that you pass a non-`nil` array reference, `VADispose` decrements its owner count. If the count falls below one, `VADispose` releases all memory occupied by the array.

Array Information

Use these functions to get and set information on a variable array.

VACount

Determines how many items there are in an array.

```
UInt32 VACount( VARef inArray )
```

`inArray`	The array whose items are to be counted.

VACount returns the number of items in inArray.

VAItemSize

Determines the size of individual array items.

```
UInt32 VAItemSize( VARef inArray )
```

inArray	The array whose item size is to be retrieved.

VAItemSize returns the size, in bytes, of an individual array item. Since all items have to be the same size, this number applies to all items.

VARefCon

Returns an array's reference constant.

```
UInt32 VARefCon( VARef inArray )
```

inArray	The array whose reference constant is to be retrieved.

VARefCon returns the reference constant previously stored in the array by means of VANew or VASetRefCon. You can use this reference constant any way you like.

VASetRefCon

Sets an array's reference constant.

```
UInt32 VASetRefCon( UInt32 inRefCon, VARef inArray )
```

inRefCon	The array's new reference constant.
inArray	The array whose reference constant is to be set.

VASetRefCon sets the reference constant stored in the array. You can use this reference constant any way you like.

VAOwnerCount

Returns the number of references currently made to an array.

```
UInt32 VAOwnerCount( VARef inArray )
```

inArray	The array whose owners are to be counted.

VAOwnerCount returns the array's current owner count. When an array is created, it has an owner count of one. VAClone increments this count by one. VADispose decrements it by one. When the count falls to zero, the array is deallocated.

Adding and Removing Data Items

Use these functions to increase or decrease the number of data items in an array.

VAAdd

Adds a number of data entries at the specified array location.

```
void* VAAdd( UInt32 inCount, UInt32 inAddHere, Boolean inZero,
            VARef inArray )
```

inCount	The number of data entries to be added.
inAddHere	The (nonzero) position at which entries are to be added.
inZero	If true, every byte of the added entries is set to zero. If false, the entries are left uninitialized.
inArray	The array to which entries are to be added.

VAAdd adds the specified number of entries at the specified position, optionally zeros the new entries, and returns a pointer to the first byte of the first added entry. If either inCount or inAddHere are zero, or if there is not enough memory to resize the array to hold the added items, VAAdd returns nil.

VAAddToEnd

Adds a number of data entries to the end of the array.

```
void* VAAddToEnd( UInt32 inCount, Boolean inZero, VARef inArray )
```

inCount	The number of data entries to be added.
inZero	If true, every byte of the added entries is set to zero. If false, the entries are left uninitialized.
inArray	The array to which entries are to be added.

VAAddToEnd functions exactly as VAAdd, except it always appends the added items to the end of the array.

VAInsert

Adds a one entry at the specified location, and copies data into it.

```
OSErr VAInsert( UInt32 inAddHere, void* inData, VARef inArray )
```

`inAddHere`	The (nonzero) position at which an entry is added.
`inData`	The address from which data is copied.
`inArray`	The array to which an entry is added.

`VAInsert` adds an entry at the specified position and copies `VAItemSize()` bytes from `inData` to the new entry.

RESULT CODES

`inputOutOfBounds` (-190)	Out-of-bounds `inAddHere` parameter.
`memFullErr` (-108)	Could not resize array.
`noErr` (0)	No error.

VAInsertAtEnd

Adds a one entry to the end of the array, and copies data into it.

```
OSErr VAInsertAtEnd( void* inData, VARef inArray )
```

`inData`	The address from which data is copied.
`inArray`	The array to which an entry is added.

`VAInsertAtEnd` adds an entry the the end of the array and copies `VAItemSize()` bytes from `inData` to the new entry.

RESULT CODES

`memFullErr` (-108)	Could not resize array.
`noErr` (0)	No error.

VARemove

Removes data items from an array.

```
void VARemove( UInt32 inCount, UInt32 inStartHere,
                    VARef inArray )
```

`inCount`	The number of items to be removed.
`inStartHere`	The nonzero index of the first item to be removed.
`inArray`	The array from which entries are to be removed.

`VARemove` removes the specified number of entries at the specified position. It checks to make sure the requested count does not exceed the number of entries and adjusts this number if necessary so it is only removing entries that actually exist. If you pass zero for

inCount or inStartHere, or if there are no entries in the array, VARemove does nothing.

VARemoveAll

Removes all array entries.

```
void VARemoveAll( VARef inArray )
```

inArray	The array whose entries are to be removed.

VARemoveAll removes all entries from the array, but does not deallocate it. The array header still records the item size, and the refCon remains intact. You can subsequently add new entries with VAAdd.

Moving Data Items

Call VAMove to move entries to a specific array position and have other entries fall into place around them. Call VAMoveAfter to move entries to follow a specific entry. Call VAValidateMoveParameters to see if the parameters to VAMoveAfter are valid.

VAMove

Moves one or more items to a different location.

```
OSErr VAMove( UInt32 inCount, UInt32 inSource, UInt32 inDest,
            VARef inArray )
```

inCount	The number of entries to move.
inSource	The (nonzero) position of the first entry to move.
inDest	The (nonzero) position to which the first entry is moved.
inArray	The array in which entries are to be moved.

VAMove moves inCount entries starting with the one at position inSource, so that the first moved entry ends up at position inDest. Where necessary, it shifts other entries to fill the gap where the moved entries were. VAMove functions as though the entries were deleted and then added at the new location. If entries are moved to a higher position and there are not enough entries above to fill the gap, the elements are moved to the highest position possible. VAMove does not try to repair any parameters.

Internally, VAMove calculates the corresponding parameters to VAMoveAfter and calls that routine.

For example if you call VAMove(2, 2, 4, array), and the array contains {a,b,c,d,e} on entry, it contains {a,d,e,b,c} on exit; the second element (b) has moved to position four and taken a subsequent entry with it. If you call VAMove(2, 2, 5, array), and the array contains {a,b,c,d,e} on entry, it contains {a,d,e,b,c}

on exit; the second and third elements (`b and c`) can't move to positions five and six because the array is not that large: there is no position six.

RESULT CODES

`inputOutOfBounds` (-190)	Out-of-bounds parameter.
`noErr` (0)	No error.

VAMoveAfter

Moves one or more items to a position after another item. This is the preferred method for moving data items.

```
OSErr VAMoveAfter( UInt32 inCount, UInt32 inSource,
                   UInt32 inDest, VARef inArray )
```

`inCount`	The number of entries to move.
`inSource`	The (nonzero) position of the first entry to move.
`inDest`	The position after which the first entry is moved.
`inArray`	The array in which entries are to be moved.

`VAMoveAfter` moves `inCount` entries starting with the one at position `inSource`, so that the first moved entry ends up after the entry which is, on input, at position `inDest`. You can pass zero for `inDest`: entries will move to the front of the array. Where necessary, it shifts other entries to fill the gap where the moved entries were. If `inSource` is the same as `inDest` or `inCount` is zero, `VAMoveAfter` does nothing and returns `noErr`. If you try to move too many entries, move a sequence of entries into itself, move to a nonexistent position, or if `inSource` is zero, `VAMoveAfter` returns `inputOutOfBounds`. No attempt is made to repair parameters to keep them in bounds.

For example, if you call `VAMoveAfter ( 2, 2, 4, array )`, and the array contains `{a,b,c,d,e}` on entry, it contains `{a,d,b,c,e }` on exit; the second element (`b`) has moved after the element that started out in position four (`d`) even though the 'd' shifts down. If you call `VAMove( 2, 2, 6, array )`, and the array contains `{a,b,c,d,e}` on entry, nothing happens because there is no position six.

RESULT CODES

`inputOutOfBounds` (-190)	Out-of-bounds parameter.
`noErr` (0)	No error.

VAValidateMoveParameters

Checks the parameters to `VAMoveAfter`.

```
OSErr VAValidateMoveParameters( UInt32 inCount, UInt32 inSource,
                          UInt32 inDest, UInt32 inNArrayItems )
```

`inCount`	The number of entries to move.

inSource	The (nonzero) position of the first entry to move.
inDest	The position after which the first entry is moved.
inNArrayItems	The number of items in the array.

VAValidateMoveParameters checks the parameters to VAMoveAfter. This can be useful if you have a number of arrays to modify using the same parameters. You don't have to call this routine – VAMoveAfter (and, indirectly, VAMove) also do the same parameter checking. If VAValidateMoveParameters returns noErr, you can ignore the return value of VAMoveAfter.

RESULT CODES

inputOutOfBounds (-190)	Out-of-bounds parameter.
noErr (0)	No error.
vaNoMovementNecessaryErr (2600)	Call would not modify the array.

VASwap

Exchanges two entries.

```
OSErr VASwap( UInt32 inItem1, UInt32 inItem2, VARef inArray )
```

inItem1, inItem2	The array items to exchange.
inArray	the array whose items are exchanged.

VASwap exchanges two array entries by calling VASwapMemory. There is no memory overhead to calling this routine. If either item parameters are zero or out of bounds VASwap returns inputOutOfBounds. If they are the same entry, VASwap does nothing and returns noErr.

RESULT CODES

inputOutOfBounds (-190)	Out-of-bounds parameter.
noErr (0)	No error.

Sorting and Searching

Call VASort to sort array items. Call VASearch to find the first matching item in a sorted array. Call VABruteForceSearch to find an item in an unsorted array. You provide a comparison callback function in all cases.

VASort

Sorts the items in an array.

```
void VASort( VASortProcPtr inSortProc, UInt32 inRefCon,
                  VARef inArray )
```

inSortProc	The address of a data comparison callback function.
inRefCon	A reference constant that is passed along to your comparison callback when it is invoked.
inArray	The array that is to be sorted.

VASort uses a nonrecursive heapsort algorithm and a comparison callback you provide to quickly sort the items in the array. You do not need to lock the array before calling VASort; it calls HGetState and HLock on entry, and HSetState on exit.

If you pass a nil comparison procedure pointer, VASort does nothing.

VASearch

Finds the first matching item in a sorted array.

```
UInt32 VASearch( VASearchProcPtr inSearchProc, UInt32 inRefCon,
                  void* inSearchData, UInt32* outNext,
                  VARef inArray )
```

inSearchProc	The address of a data comparison callback function.
inRefCon	A reference constant that is passed along to your comparison callback when it is invoked.
inSearchData	The data to be searched for.
outNext	On output, the index of the next matching item if no exact match was found, or zero if there was no match and no next item. Pass nil to ignore next item.
inArray	The array that is to be searched.

VASearch uses a binary search algorithm and a comparison callback you provide to quickly find the index of the data in question. The array must be sorted beforehand or else the results will most likely not be correct. If the data is found, VASearch returns the index of the item that holds the data. If it is not found, it returns zero and, if you pass a non-nil value for outNext, passes back the index of the next "larger" item. If there is no "next" item (for example, the array contains {a,b,c,d} and you search for 'e') outNext points to zero on output.

In many cases, the array's data size will be greater than 32 bits, in which case you will actually pass a pointer to some type or struct instead of a UInt32 in the inSearchData parameter. The value you pass in inSearchData is exactly the same as the value passed to your comparison callback. VArrayLib doesn't care what kind of data is passed.

If you pass a nil comparison procedure pointer, VASearch returns zero.

WARNING

If your comparison procedure may allocate or move memory, you must lock the array handle (by calling `HLock`) before calling `VASearch`.

VABruteForceSearch

Finds the first matching item in an unsorted array.

```
UInt32 VABruteForceSearch( VABruteForceSearchProcPtr inProc,
                           UInt32 inRefCon, void* inSearchData,
                           VARef inArray )
```

`inProc`	The address of a data comparison callback function.
`inRefCon`	A reference constant that is passed along to your comparison callback when it is invoked.
`inSearchData`	The data to be searched for.
`inArray`	The array that is to be searched.

`VABruteForceSearch` uses a brute-force search algorithm and a comparison callback you provide to find the index of the data in question. It returns the first array item that the `inProc` callback indicates as matching the search data. If the data is found, `VABruteForceSearch` returns the index of the item that holds the data. If it is not found, `VABruteForceSearch` returns zero.

In many cases, the array's data size will be greater than 32 bits, in which case you will actually pass a pointer to some type or struct instead of a `UInt32` in the `inSearchData` parameter. The value you pass in `inSearchData` is exactly the same as the value passed to your comparison callback. VArrayLib doesn't care what kind of data is passed.

If you pass a `nil` comparison procedure pointer, `VABruteForceSearch` returns zero.

WARNING
If your comparison procedure may allocate or move memory, you must lock the array handle (by calling `HLock`) before calling `VABruteForceSearch`.

Internal Sorting and Searching

You can use these routines if you want VArrayLib to perform comparisons using its own internal routines. The routines can do 1, 2, and 4-byte signed and unsigned integer comparisons, and byte-by-byte comparisons for larger data sizes. You can sort using data structure fields. Pass the size of the field to compare, its offset in the entry, and whether it is signed. VArrayLib provides the va_`OffsetOf` macro if your headers do not define `offsetof`.

Example

```
typedef struct
{
      UInt32            fieldU32;
      SInt16            fieldS16;
      SInt16            padding;
```

```
} MyStruct;

// Sort an array by fieldU32
VAInternalSort( sizeof(UInt32), false,
                va_OffsetOf(MyStruct,fieldU32), array );
// Sort an array by field S16
VAInternalSort( sizeof(SInt16), true,
                va_OffsetOf(MyStruct, fieldS16), array );
```

VAInternalSort

Sorts the items in an array using internal comparison routines.

```
void VAInternalSort( Size inSize, Boolean inSigned,
                UInt32 inOffset, VARef inArray )
```

`inSize`	The size of the entry field to compare.
`inSigned`	`true` if the entry field is signed.
`inOffset`	The offset in bytes to the field within an entry.
`inArray`	The array to search.

`VAInternalSort` uses a nonrecursive heapsort algorithm and internal comparison routines to quickly sort the items in the array. You do not need to lock the array before calling `VASort`; it calls `HGetState` and `HLock` on entry, and `HSetState` on exit.

VAInternalSearch

Finds the first matching item in a sorted array using internal comparison routines.

```
UInt32 VAInternalSearch( Size inSize, Boolean inSigned,
                        UInt32 inOffset, void* inSearchData,
                        UInt32* outNext, VARef inArray )
```

`inSize`	The size of the entry field to compare.
`inSigned`	true if the entry field is signed.
`inOffset`	The offset in bytes to the field within an entry.
`inSearchData`	The data to be searched for.
`outNext`	On output, the index of the next matching item if no exact match was found, or zero if there was no match and no next item. Pass `nil` to ignore next item.
`inArray`	The array to be searched.

`VAInternalSearch` uses a binary search algorithm and internal comparison routines to quickly find the index of the data in question. The array must be sorted beforehand or else the results will most likely not be correct. If the data is found, `VAInternalSearch`

returns the index of the item that holds the data. If it is not found, it returns zero and, if you pass a non-`nil` value for `outNext`, passes back the index of the next "larger" item. If there is no "next" item (for example, the array contains `{a,b,c,d}` and you search for 'e') `outNext` points to zero on output.

VAInternalBruteForceSearch

Finds the first matching item field in an unsorted array using internal comparison routines.

```
UInt32 VAInternalBruteForceSearch( Size inSize, Boolean inSigned,
                                   UInt32 inOffset, void*
                                   inSearchData,
                                   VARef inArray )
```

`inSize`	The size of the entry field to compare.
`inSigned`	true if the entry field is signed.
`inOffset`	The offset in bytes to the field within an entry.
`inSearchData`	The data to be searched for.
`inArray`	The array to be searched.

`VAInternalBruteForceSearch` uses a brute-force search algorithm and internal comparison to find the index of the data in question. It returns the first array item that matching the criteria. If the data is found, `VAInternalBruteForceSearch` returns the index of the item that holds the data. If it is not found, `VAInternalBruteForceSearch` returns zero.

Getting and Setting Array Data

Use these functions when you need to get or change array item data. `VANthItemPtr` gets the address of an item's data. `VALastItemPtr` gets the address of the last item's data. `VAGetNthItem` copies item data into a buffer you provide.

VANthItemPtr

Gets the address of the first byte of data in an array entry.

```
void* VANthItemPtr( UInt32 inItem, VARef inArray )
```

`inItem`	The index of the item a pointer to whose data is to be retrieved.
`inArray`	The array that contains the item.

`VANthItemPtr` returns a pointer to the first byte of data contained in `inItem`. If `inItem` is greater than the actual number of array items, `VANthItemPtr` returns `nil`.

VALastItemPtr

Gets the address of the first byte of data in the last array entry.

```
void* VALastItemPtr( VARef inArray )
```

`inArray`	The array whose last item is sought.

`VALastItemPtr` returns a pointer to the first byte of data contained in the final array entry. If there are no items in the array, `VALastItemPtr` returns `nil`.

VAGetNthItem

Copies entry data into a buffer.

```
OSErr VAGetNthItem( UInt32 inItem, void* outData, VARef inArray )
```

`inItem`	The index of the item whose data is to be retrieved.
`outData`	The address to which data is copied.
`inArray`	The array that contains the item.

`VAGetNthItem` copies a number of bytes equal to the size of an array item into the buffer referenced by `outData`. Make sure your buffer is large enough to hold `VAItemSize()` bytes of data.

RESULT CODES

`inputOutOfBounds` (-190)	Index out of range.
`noErr` (0)	No error.

VASetNthItem

Copies data from a buffer to an array item.

```
OSErr VASetNthItem( UInt32 inItem, void* inData, VARef inArray )
```

`inItem`	The index of the item whose data is to be retrieved.
`inData`	The address from which data is copied.
`inArray`	The array that contains the item.

`VASetNthItem` copies a number of bytes equal to the size of an array item from the buffer referenced by `inData`. This routine only copies `VAItemSize()` bytes of data.

RESULT CODES

`inputOutOfBounds` (-190)	Index out of range.

noErr (0)	No error.

Copying

Call VACopy to make an exact copy of an array. Call VACopyPart to copy entries from the beginning of an array. Call VAClone to increment an array's owner count. Call VAConcatenate to merge two arrays.

VACopy

Makes a copy of an array.

```
VARef VACopy( VARef inSourceArray, VARef inTargetArray )
```

inSourceArray	The array that is to be copied.
inTargetArray	An array to which entries are copied, or nil to create a new array.

VACopy makes a copy of an existing array and returns it. If inTargetArray is non-nil, it is emptied of items and all entries are copied to it from inSourceArray. If inTargetArray is nil, a new array is allocated. inTargetArray's data size, count, refCon, and data are the same as inSourceArray. Only inTargetArray's owner count remains untouched, or initialized to one if a new array is created. If there is not enough memory to create a new array, VACopy returns nil.

VACopyPart

Makes a copy of part of an array.

```
VARef VACopy( UInt32 inItems, VARef inArray )
```

inItems	The number of data items copied from the beginning of the source array into the new one.
inArray	The array that is to be copied.

VACopyPart makes an copy of part of an existing array and returns it. The new array contains the first inItems data entries from the old array. The data size and refCon are the same as in the old one. If there is not enough memory to create the new array, VACopyPart returns nil.

VAClone

Increments an array's owner count.

```
VARef VAClone( VARef inArray )
```

inSourceArray	The array that is to be cloned.

VAClone increments the array's owner count by one and returns the array as a function result.

VAConcatenate

Merges two arrays.

```
OSErr VAConcatenate( VARef inSource, VARef inDest )
```

inSource	The array whose items are to be copied.
inDest	The array to which items are to be added.

VAConcatenate copies all entries in inSource to the end of inDest. Both arrays must have the same item size for VAConcatenate to work.

RESULT CODES

memFullErr (-108)	Could not resize destination array.
paramErr (-50)	Arrays have unequal entry sizes.
noErr (0)	No error.

Repetitive Tasks

Call VAIterate to invoke a callback for each array entry.

VAIterate

Invokes a callback for each data item.

```
void VAIterate( VAIteratorPtr inProc, void* ioUserData,
                VARef inArray )
```

inProc	The address of a callback routine that is to be invoked for each array item.
ioUserData	A reference value passed to your iterator procedure.
inArray	The array whose items are processed by the iterator.

VAIterate calls you iterator procedure once for each item in the array. It calls HGetState and HLock on entry, and HSetState on exit. Generally, using VAIterate is more convenient than writing your own for loop and calling VANthItemPtr every time through.

Utility

Call VAZeroMemory to set all bytes in a block of memory to zero. Call VASwapMemory to exchange the data in two buffers. The routines VANewNumberOnMovement, NewNumberOnAddition, and VANewNumberOnDeletion, simulate item renumbering.

VAZeroMemory

Sets all bytes in a block of memory to zero.

```
void VAZeroMemory( Ptr inPtr, UInt32 inBytes )
```

inPtr	The address of the first byte to be zeroed.
inBytes	The number of bytes to be zeroed.

VAZeroMemory sets inBytes bytes to zero, starting with the one pointed to by inPtr. You may call this routine for a block of memory you wish to initialize or reinitialize.

In the 68K version of VArrayLib this routine is written in C. In the PowerPC version it is written in assembly language. All versions try to store a maximum number of bytes (4 and 8, respectively) per iteration while respecting data alignment.

WARNING
You may not be saving time if you call NewPtr and VAZeroMemory instead of NewPtrClear. Inside Macintosh: Memory claims that NewPtrClear and NewHandleClear clear the memory byte-by-byte. However, performance seems to have been enhanced substantially perhaps as recently as Mac OS 8.6, possibly because BlockZero has been made available to pre-PCI Power Macs. Whatever the reason, currently NewPtrClear outperforms VAZeroMemory by a wide margin. Your best bet may be to use VAZeroMemory only for reinitializing memory that is already allocated. You may also wish to use BlockZero if it is available and fall back on VAZeroMemory if it is not.

Any information on these apparent speedups in the low-level Memory Manager would be appreciated (the Technotes say nothing, last I checked). As would suggestions for improving performance. A 68K assembly version sure would be nice. And I should get around to AltiVec, not that I have a G4 to test it on.

VASwapMemory

Exchanges the data in two buffers.

```
void VASwapMemory( Ptr inPtr1, Ptr inPtr2, UInt32 inBytes )
```

inPtr1	The address of the first buffer.
inPtr2	The address of the second buffer.
inBytes	The number of bytes to be swapped.

VASwapMemory exchanges inBytes bytes of data between inPtr1 and inPtr2. It assumes that the two memory blocks do not overlap. This routine does not allocate memory. It is used internally in the heapsort routines when dealing with data items over 32 bits in size.

In the 68K version of VArrayLib this routine is written in C. The PowerPC version is assembly language. All versions try to swap a maximum number of bytes (4 and 8, respectively) per iteration while respecting data alignment.

VANewNumberOnMovement

Calculates how an item is renumbered as a result of `VAMoveAfter`.

```
void VANewNumberOnMovement( UInt32* ioNumber, UInt32 inCount,
                            UInt32 inSource, UInt32 inDest )
```

`ioNumber`	On input, the number of the item before movement. On output, the number after movement.
`inCount`	The number of entries to move.
`inSource`	The (nonzero) position of the first entry to move.
`inDest`	The position after which the first entry is moved.

`VANewNumberOnMovement` calculates the value of `*ioNumber` as it would be after a call to `VAMoveAfter` with the `inCount`, `inSource`, and `inDest` parameters. All movement parameters are assumed to be valid. It is safe to pass zero in `ioNumber`.

VANewNumberOnAddition

Calculates how an item is renumbered as a result of `VAAdd`.

```
void VANewNumberOnAddition( UInt32* ioNumber, UInt32 inCount,
                            UInt32 inAddHere )
```

`ioNumber`	On input, the number of the item before movement. On output, the number after movement.
`inCount`	The number of data entries to be added.
`inAddHere`	The (nonzero) position at which entries are to be added.

`VANewNumberOnAddition` calculates the value of `*ioNumber` as it would be after a call to `VAAdd` with the `inCount` and `inAddHere` parameters. All addition parameters are assumed to be valid. It is safe to pass zero in `ioNumber`.

VANewNumberOnDeletion

Calculates how an item is renumbered as a result of `VARemove`.

```
void VANewNumberOnDeletion( UInt32* ioNumber, UInt32 inCount,
                            UInt32 inDeleteHere )
```

`ioNumber`	On input, the number of the item before movement. On output, the number after movement.
`inCount`	The number of items to be removed.
`inDeleteHere`	The nonzero index of the first item to be removed.

`VANewNumberOnDeletion` calculates the value of `*ioNumber` as it would be after a call to `VADelete` with the `inCount` and `inDeleteHere` parameters. All deletion parameters are assumed to be valid. It is safe to pass zero in `ioNumber`.