

SRP_FastArray

SRP Fast Arrays are an alternative replacement to BASIC+ dynamic arrays and can greatly enhance performance for large processes. SRP_FastArray provides the following services:

Method	Description	Added
Clear	Removes all values from an SRP Fast Array.	2.1.10
Count	Counts the number of fields, values, or subvalues in an SRP Fast Array.	
Create	Creates a new SRP Fast Array.	
Delete	Performs a DELETE on an SRP Fast Array.	
Extract	Performs an EXTRACT on an SRP Fast Array.	
GetVariable	Converts an SRP Fast Array back into a BASIC+ dynamic array.	
Insert	Performs an INSERT on an SRP Fast Array.	
InsertFromList	Inserts an SRP List into an SRP Fast Array.	
Match	Finds the first element to match a given string.	
Reduce	Creates a new fast array containing only those elements that match the given string.	
Release	Releases the handle to an SRP Fast Array.	
Replace	Performs a REPLACE on an SRP Fast Array.	
ReplaceWithList	Replaces an element in an SRP Fast Array with an SRP List.	

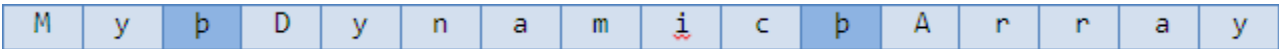
BASIC+ Dynamic Arrays

Easily one of the best features of the BASIC+ language is its dynamic array manipulation. It's easy, intuitive, and flexible: the trifecta of efficient programming. No need to predetermine the size of your array. No need to worry about going out of bounds. Just set a field, value, or subvalue and you're all set.

```
Array<2, 7> = MyValue
```

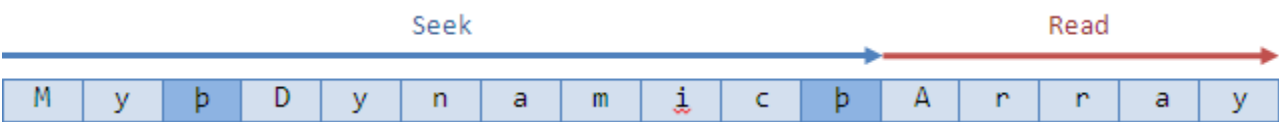
However, efficient programming comes at a cost. In this case, that cost is performance. To understand why, we have to remember that a dynamic array in BASIC+ is just a single string.

```
Array = "My":@FM:"Dynamic":@FM:"Array"
```



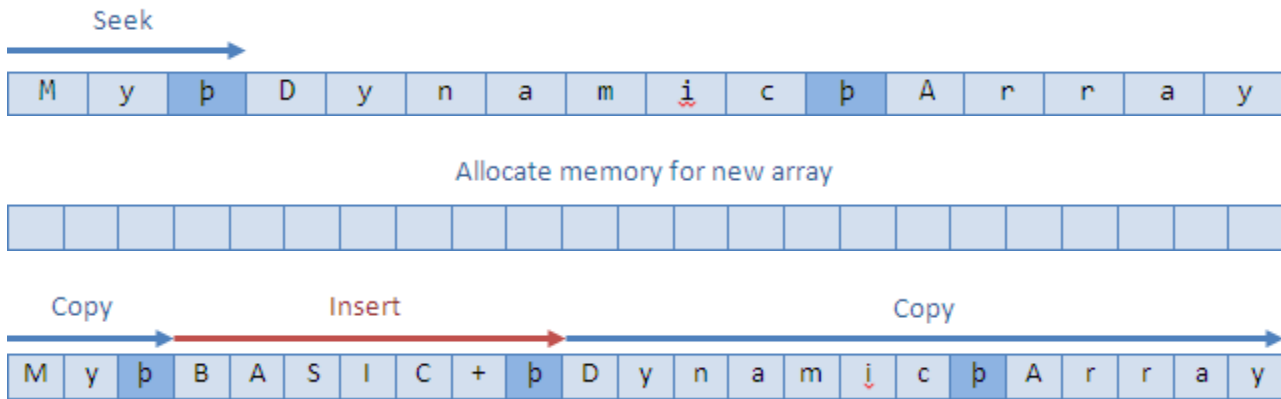
Whenever you extract a value from the array, BASIC+ scans the entire string counting delimiters until it finds what you are looking for:

```
Value = Array<3>
```



Inserts, Replaces, and Deletes involve three steps. First, it scans the array to find the point of insertion, deletion, or replacing. Next, a new block of memory is allocated. Finally, the new array is created by copying elements from the old array as well as copying in any new values.

```
Array = Insert(Array, 2, 0, 0, "BASIC+")
```



If you have a For loop that is inserting thousands of elements into a dynamic array, then BASIC+ is reallocating memory and rewriting your array thousands of times also. This kind of operation is expensive performance wise.

SRP Fast Arrays

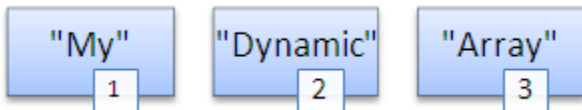
SRP Fast Arrays seek to offer an alternative to BASIC+ dynamic arrays that behave exactly the same way yet offer much more performance. This performance is achieved in two ways. First, it is written in optimized C++. Second, it never uses a single string to hold the array. You start by creating an SRP Fast Array. You can create an empty SRP Fast Array like this:

```
ArrayHandle = SRP_FastArray("Create")
```

Or you can create an SRP Fast Array initialized to a BASIC+ dynamic array, like this:

```
ArrayHandle = SRP_FastArray("Create", "My:@FM:"Dynamic":@FM:"Array")
```

Here's what your array looks like in memory.

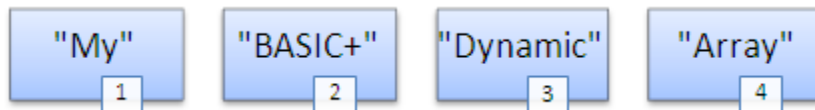


Notice how no delimiters are tracked and each element is stored in its own block of memory, although their order is maintained through a numeric index. Extracting an element is always fast because there is no need to scan a string.

```
Value = SRP_FastArray("Extract", ArrayHandle, 3, 0, 0)
```

SRP_FastArray simply grabs element three and returns it. Inserts, Deletes, and Replaces are all equally fast for similar reasons. SRP_FastArray never needs to recopy its elements. An insert simply places the new value as a new string right where it needs to go.

```
SRP_FastArray("Insert", ArrayHandle, 2, 0, 0, "BASIC+")
```



You can imagine how fast this is for dynamic arrays with thousands of elements being inserted, deleted, and replaced. Of course, you will eventually need your SRP Fast Array to become a BASIC+ dynamic array again, and this is how you do it:

```
Array = SRP_FastArray("GetVariable", ArrayHandle)
```

This function does allocated memory for the entire dynamic array and copies each element—with delimiters—into it. Obviously, if you call this method thousands of times, then you're no better off than when you were using a BASIC+ dynamic array. The idea is to manipulate your array using SRP_FastArray and then writing it out when you are done.

There is always one more step when using SRP Fast Arrays. You have to release the handle.

```
SRP_FastArray("Release", ArrayHandle)
```

Sure, it's an extra step, but the performance gains are usually worth it. If you forget to do this, SRP Utilities will clean up all unreleased handles when OpenInsight closes, but if you plan on your application running for hours at a time, you still want to avoid memory getting used up and never freed.

Array Sizes

One thing we often do is count the number of elements in an array. In BASIC+ we typically do this with the following code:

```
ArrayCount = Count(Array, @FM) + (Array NE " ")
```

It should be obvious that the Count method scans the entire string counting delimiters, and then we have to add 1 to the final count if the array is not empty. To count the number of fields in an SRP Fast Array, just do this:

```
ArrayCount = SRP_FastArray("Count", Handle, 0, 0)
```

Counting elements in an SRP Fast Array is instantaneous because field, value, and subvalue counts are all cached.

Multi-dimensional Arrays

So far, all the samples have been dealing with field-mark delimited arrays. However, SRP_FastArray supports three dimensional arrays as well. As previously stated, SRP_FastArray emulates the Insert, Delete, Replace, and Extract methods of BASIC+ to the fullest. In fact, the deeper and more complex your dynamic array, the more performance you will get from SRP_FastArray.

You can still place other delimiters into fields, values, and subvalues, but those other delimiters will just be treated like regular characters. This, of course, is no different than BASIC+.

When to Use SRP_FastArray

As with everything, use the right tool for the right job. There are two factors to consider: the size of the dynamic array and the number of operations you intend to do upon it. The smaller the dynamic array, the less amount of time BASIC+ spends scanning, allocating, and copying. Also, even if you have an array that takes up several megabytes in memory, if you are only doing two extracts, SRP_FastArray won't make that much of a difference.

Perhaps the best rule of thumb is this: if you have a process that takes a long time to run, look to see how much dynamic array manipulation it is doing. If there appears to be a great deal of it, then try out SRP_FastArray and see if you get an improvement. Benchmark the differences to see if the increase in code complexity (SRP_FastArray services are, after all, not as pretty as angle bracket operators) seems worth the performance gain, then huzzah! Some of us at SRP Computer Solutions have managed increase a routine's performance by a factor of ten!

When Not to Use SRP_FastArray

There is one situation in which SRP_FastArray does not seem to offer any significant performance increase: appending values. If your routine is only appending values at the end of the array and your dynamic array is only 1-dimensional, then BASIC+ is plenty fast for you. Essentially, BASIC+ dynamic arrays begin to suffer in performance when you perform random access, and appending values to a single-dimensioned array is not random access.

Sample Benchmark Code

Here is a routine you can copy into OpenInsight and run (after installing SRP Utilities 1.4 or greater, of course). It inserts and extracts elements into each type of array at random and times it. Notice that 1000 iterations or less offer pretty negligible gains, but 10,000 or more iterations really start to show the speed difference. Your mileage will vary depending upon your PC specifications.

```

Compile function Test_FastArray(VOID)

$Insert SRPFASTARRAY
// OI routines
Declare function GetTickCount
Declare subroutine Msg

Iterations = 10000                                ; // How many operations to perform
MaxPos = 100                                       ; // The maximum field, value, or subvalue position
InsertText = "SRP Computer Solutions, Inc."       ; // The text to insert

InitRnd Time():Date()

// Insert the text at random field, value, and subvalue positions
StartTime = GetTickCount()
TestOI = ""
For i = 1 to Iterations
    TestOI = Insert(TestOI, Rnd(MaxPos), Rnd(MaxPos), Rnd(MaxPos), InsertText)
Next i
InsertTimeOI = GetTickCount() - StartTime

// Insert the text at random field, value, and subvalue positions using SRP_FastArray
StartTime = GetTickCount()
Handle = SRP_FastArray("Create")
For i = 1 to Iterations
    SRP_FastArray("Insert", Handle, Rnd(MaxPos), Rnd(MaxPos), Rnd(MaxPos), InsertText)
Next i
TestSRP = SRP_FastArray("GetVariable", Handle)
InsertTimeSRP = GetTickCount() - StartTime

// Extract values at random field, value, and subvalue positions
StartTime = GetTickCount()
For i = 1 to Iterations
    A = Extract(TestOI, Rnd(MaxPos), Rnd(MaxPos), Rnd(MaxPos))
Next i
ExtractTimeOI = GetTickCount() - StartTime

// Extract values at random field, value, and subvalue positions using SRP_FastArray
StartTime = GetTickCount()
For i = 1 to Iterations
    A = SRP_FastArray("Extract", Handle, Rnd(MaxPos), Rnd(MaxPos), Rnd(MaxPos))
Next i
ExtractTimeSRP = GetTickCount() - StartTime

// Clean up the memory
SRP_FastArray("Release", Handle)

Msg(@Window, "OI Insert Time: ":InsertTimeOI:"ms|SRP Insert Time: ":InsertTimeSRP:"ms||OI Extract Time: ":
ExtractTimeOI:"ms|SRP Extract Time: ":ExtractTimeSRP:"ms")

Return 1

```

Note that the first benchmark will always be a little slower than subsequent benchmarks because upon the first run, the C++ code has to be loaded into memory.

Searching for Elements in an SRP Fast Array

In BASIC+, we use the `Locate` statement to search for elements in our dynamic arrays. If our array is already single-dimensional, then we just use `Locate` against it and we're happy. If we want to find elements in the second or third dimension, we have to extract the lower tier list first, and then we can search it.

```

Locate Target in Array<2, 10> using @SVM setting Pos then
end

```

`SRP_FastArray` does not have a `Locate` service, but it does have the [Match](#) and [Reduce](#) services. The `Match` service looks for an element that matches (in part or in total) a given string and gives you the position of the first match. The `Reduce` service finds all elements that match a string and return a reduced array containing only those elements.

```
Pos = SRP_FastArray("Match", Handle, Target, FieldPos, ValuePos, SubValuePos, "MatchAll")
```

Inserting SRP Lists

SRP Fast Arrays play nicely with [SRP Lists](#). In addition to using Insert and Replace on single values, you can also insert SRP whole SRP Lists or replace elements with whole SRP Lists. Since SRP Lists have no delimiter recognitions at all, they will behave as though they are delimited one level lower than the insert level. So, if you insert an SRP List into a field position, the SRP List will behave as though it is value-mark delimited.

```
SRP_FastArray_InsertFromList(ArrayHandle, 10, 0, 0, ListHandle)
```

If you insert an SRP List into a subvalue position, then the SRP List will behave as though it is also subvalue-mark delimited. The Replace method works on the same logic.

```
SRP_FastArray_ReplaceWithList(ArrayHandle, 10, 5, 7, ListHandle)
```

Points to Remember

Don't forget to release your SRP Fast Array handles. Always.

Note that one major difference between the BASIC+ Insert, Delete, and Replace routines and the SRP_FastArray equivalent services is that the BASIC+ routines always creates a new Array and returns it. SRP_FastArray does not do this since creating copies is the performance bottle neck SRP_FastArray is working to avoid.

Oh yeah, one more thing: **don't forget to release your SRP Fast Array handles.**