# 3 Arrays Module (arrays.hhf)

The HLA Arrays module provides a set of datatypes, macros, and procedures that simplify array access in assembly language (especially multidimensional array access). In addition to supporting standard HLA arrays with static size declarations, the HLA arrays module also supports dynamic arrays that let you specify the array size at run-time.

**Note**: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

## 3.1 The Arrays Module

To use the array macros in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "arrays.hhf" )
or
#include( "stdlib.hhf" )
```

## 3.2 Array Data Types

The *array* namespace defines the following useful data types:

**`#macro array.dArray( type, dimensions );`**

The first feature in the array package to consider is the support for dynamic arrays. HLA provides a macro/data type that lets you tell HLA that you want to specify the array size under program control. This macro/data type is *array.dArray* (*dArray* stands for *dynamic array*). You use this macro invocation in place of a standard data type identifier in an HLA variable declaration.

The first macro parameter is the desired data type; this would typically be an HLA primitive data type like *int32* or *char*, although any data type identifier is legal.

The second parameter is the number of dimensions for this array data type Generally this value is two or greater (since creating dynamic single dimensional arrays using only malloc is trivial). Because of the way array indicies are computed by HLA, it is not possible to specify the number of dimensions dynamically.

Note: since *array.dArray* is not a data type identifier (it's a macro), you cannot directly create a dynamic array of dynamic arrays. I.e., the following is not legal:

```
static
  DAofDAs: array.dArray( array.dArray( uns32, 2), 2 );

However, you can achieve exactly the same thing by using the following code:

type
  DAs: array.dArray( uns32, 2 );

static
  DAofDAs: array.dArray( DAs, 2 );
```

The TYPE declaration creates a type identifier that is a dynamic array. The STATIC variable declaration uses this type identifier in the *array.dArray* invocation to create a dynamic array of dynamic arrays.

# 3.3 Array Allocation and Deallocation

**#macro array.daAlloc( dynamicArrayName, <<list of dimension bounds>> );**

The *array.dArray* macro allocates storage for a dynamic array variable. It does not, however, allocate storage for the dynamic array itself; that happens at run-time. You must use the array.daAlloc macro to actually allocate storage for your array while the program is running. The first parameter must be the name of the dynamic array variable you've declared via the *array.dArray* macro. The remaining parameters are the number of elements for each dimension of the array. This list of dimension bounds must contain the same number of values as specified by the second parameter in the *array.dArray* declaration. The dimension list can be constants or memory locations (note, specifically, that registers are not currently allowed here; this may be fixed in a future version).

The following code demonstrates how to declare a dynamic array and allocate storage for it at run-time:

```
program main;
static
  i:uns32;
  j:uns32;
  k:uns32;
  MyArray: array.dArray( uns32, 3 );

begin main;

  stdout.put( "Enter size of first dimension: " );
  stdin.get( i );
  stdout.put( "Enter size of second dimension: " );
  stdin.get( j );
  stdout.put( "Enter size of third dimension: " );
  stdin.get( k );

  // Allocate storage for the array:

  array.daAlloc( MyArray, i, j, k );

  << Code that manipulates the 3-D dynamic array >>

end main;
```

**#macro array.daFree( dynamicArrayName );**

Use the *array.daFree* macro to free up storage you've allocated via the *array.daAlloc* call. This returns the array data storage to the system so it can be reused later. Warning: do not continue to access the array's data after calling *array.daFree*. The system may be using the storage for other purposes after you release the storage back to the system with *array.daFree*.

Note: You should only call *array.daFree* for arrays you've allocated via *array.daAlloc*.

Example:

```
// Allocate storage for the array:

array.daAlloc( MyArray, i, j, k );

<< Code that manipulates the 3-D dynamic array >>

array.daFree( MyArray );
```

## 3.4 Array Predicates

**#macro array.IsItVar( objectName )**

This is a macro that evaluates to a compile-time expression yielding true if the object is a variable identifier. Variable identifiers are those IDs you declare in a VAR, STATIC, READONLY, or STORAGE declaration section, or IDs you declare as parameters to a procedure. This macro returns false for all other parameters.

**#macro array.IsItDynamic( arrayName )**

This is a macro that expands to a compile-time expression yielding true or false depending upon whether the parameter was declared with the *array.dArray* data type. If so, this function returns true; else it returns false. Note that a return value of false does not necessarily indicate that the specified parameter is a static array. *Anything* except a dynamic array object returns false. For example, if you pass the name of a scalar variable, an undefined variable, or something that is not a variable, this macro evaluates false. Note that you can use the HLA @type function to test to see if an object is a static array; however, @type will not return *hla.ptArray* for dynamic array objects since *array.dArray* objects are actually records. Hence the *array.IsItDynamic* function to handle this chore.

## 3.5 Array Element Access

**#macro array.index( reg32, arrayName, <<list of indicies>> );**

This macro computes a row-major order index into a multidimensional array. The array can be a static or dynamic array. The list of indicies is a comma separate list of constants, 32-bit memory locations, or 32-bit registers. You should not, however, specify the register appearing as the first parameter in the list of indicies.

If the VAL constant *array.BoundsChk* is true, this macro will emit code that checks the bounds of the array indicies to ensure that they are valid. The code will raise an *ex.ArrayBounds* exception if any index is out of bounds. You may disable the code generation for the bounds checking by setting the *array.BoundsChk* VAL object to false using a statement like the following:

```
?array.BoundsChk := false;
```

You can turn the bounds checking on and off in segments of your code by using statements like the above that set *array.BoundsChk* to true or false.

This macro leaves pointer into the array sitting in the specified 32-bit register.

Example:

```
static
  arrayS: uns32[ 2,3,4 ];
  arrayD: array.dArray( uns32, 3 );

      .
      .
      .
  // copy arrayS[i, j, k] to arrayD[m,n,p]:

  array.index( ebx, arrayS, i, j, k );
  mov( [ ebx ], eax );// EAX := arrayS[I,j,k];
  array.index( ebx, arrayD, m, n, p );
  mov( eax, [ebx] );// EAX := arrayD[m,n,p];
```

**iterator array.element( arrayName );**

This iterator returns each successive element of the specified array. It returns the elements in row major order (that is, the last dimension increments the fastest and the first dimension increments the slowest when returning elements of a multidimensional array). This iterator returns byte objects in the AL register; it returns word objects in the AX register; it returns dword objects in the EAX register; it returns 64-bit (non-real) objects in the EDX:EAX register pair. This routine returns all floating point (real) objects on the top of the FPU stack.

Note that *array.element* is actually a macro, not an iterator. The macro, however, simply provides overloading to call one of seven different iterators depending on the size and type of the operand. However, this macro implementation is transparent to you. You would use this macro exactly like any other iterator.

Note that *array.element* works with both statically declared arrays and dynamic arrays you've declared with *array.dArray* and you've allocated via *array.daAlloc*.

Examples:

```
static
  arrayS: uns32[ 2,3,4 ];
  arrayD: array.dArray( uns32, 3 );

      .
      .
      .

  foreach array.element( arrayS ) do

      stdout.put( "Current arrayS element = ", eax, nl );

  endfor;
      .
      .
      .
  foreach array.element( arrayD ) do

      stdout.put( "Current arrayD element = ", eax, nl );

  endfor;
      .
      .
      .
```

## 3.6 Array Operations

**#macro array.cpy( srcArray, destArray );**

This macro copies a source array to a destination array. Both arrays must be the same size and shape (shape means that they have the same number of dimensions and the bounds on all the dimensions correspond between the source and destination arrays). Both static and dynamic array variables are acceptable for either parameter.

Example:

```
static
  arrayS: uns32[ 2,3,4 ];
  arrayD: array.dArray( uns32, 3 );

      .
      .
      .

  // note: for the following to be legal at run-time,
  // the arrayD dynamic array must have storage allocated
  // for it with a statement like
  //       "array.daAlloc( arrayD, 2, 3, 4 );"

  array.cpy( arrayS, arrayD );
```

```
#macro array.reduce( srcArray, destArray );

#keyword array.beforeRow;

#keyword array.reduction;

#keyword array.afterRow;

#terminator array.endreduce;
```

The *array.reduce* macro emits code to do a "row-reduction" on an array. A row reduction is a function that compresses all the rows (that is, the elements selected by running through all the legal values of the last dimension) to a single element. Effectively, this macro reduces an array of arity n to an array of arity n-1 by eliminating the last dimension.

Reduction is not accomplished by simply throwing away the data in the last dimension (although it's possible to do this). Instead, you've got to supply some code that the array.reduce macro will use to compress each row in the array.

A very common reduction function, for example, is addition. Reduction by addition produces a new array that contains the sums of the rows in the previous array. For example, consider the following matrix:

```
1  2  3  4
6  5  4  1
5  9  8  0
```

This is a 3x4 array. Reducing it produces a one dimensional array with three elements containing the value 10, 16, 22 (the sums of each of the above rows).

The best way to understand how the array.reduce macro works is to manual implement addition reduction manually. To reduce the 3x4 array above to a single array with three elements, you could use the following code:

```
// (a) Any initialization required before loops
//     (this example requires no such initialization.)

for( mov( 0, i ); i < 3; inc( i )) do

    mov( 0, eax );// (b) Initialize sum for each row.

    for( mov( 0, j); j < 4; inc( j )) do

        // (c) Sum up each element in this row into EAX:

        index( ebx, array3x4, i, j );
        add( [ebx], eax );

    endfor;

    // (d) At the end of each row, store the sum away
    // into the destination array.

    mov( i, ebx );
    mov( eax, array3[ ebx*4 ]);

endfor;
```

The *array.reduce* macro is an example of an HLA *context-free macro construct*. This means that the call to *array.reduce* consists of multiple parts, just like the REPEAT..UNTIL and SWITCH..CASE..ENDSWITCH control structures. Specifically, an array.reduce invocation consists of the following sequence of macro invocations:

```
array.reduce( srcArray, destArray );

      << Initialization statements needed before
```

```
            loops,(a) in the code above >>

        array.beforeRow;

            << Initialization before each row, (b) in the
               code above.  Note that edi contains the row
               number times the size of an element and esi contains
               an index into the array to the current element. >>

        array.reduction;

            << Code that compresses the data for each
               row, to be executed for each element
               in the row.  Corresponds to (c) in the
               the code above.  Note that ecx contains
               the index into the current row. >>

        array.afterRow;

            << Code to process the compressed data at
               the end of each row.  Corresponds to (d)
               in the code above. >>

    array.endreduce;
```

A conversion of the previous code to use the array.reduce macro set looks like the following:

```
    array.reduce( array3x4, array3 )

           // No pre-reduction initialization...

        array.beforeRow

            mov( 0, eax );// Initialize the sum for each row.

        array.reduction

            add( array3x4[esi], eax );

        array.afterRow

            mov( i, edx );
            mov( eax, array3[ edx*4 ]);

    array.endreduce;
```

Note that the array.reduce macro set makes extensive use of the 80x86 register set. The EAX and EDX registers are the only free registers you can use (without restoring) within the macro. Of course, *array.reduce* will preserve all the regsiters is uses, but within the macro itself it assumes it can use all registers except EAX and EDX for its own purposes.

**#macro array.transpose( srcArray, destArray, optionalDimension);**

The *array.transpose* macro copies the source array to the destination array transposing the elements of the last dimension with the dimension specified as the last parameter. For the purposes of this macro, the array dimensions of an n-dimensional array are numbered as follows:

```
    SomeArray[ n-1, n-2, ..., 3, 2, 1, 0 ];
```

Therefore, *array.transpose* will transpose dimension zero with some other dimension (1..n-1) in the source array when copying the data to the destination array. By default (if you don't supply the optional, third parameter), *array.transpose* will transpose dimensions zero and one when copying the source array to the destination array.

The source and destination arrays must have at least two dimensions. They can be static or dynamic arrays. Note that *array.transpose* emits special, efficient, code when transposing dimensions zero and one.

The source and destination arrays must have compatible shapes. The shapes are compatible if the arrays have the same number of dimensions and all the dimensions have the same bounds except dimension zero and the transpose dimension (which must be swapped). For example, the following two arrays are transpose-compatible when transposing dimensions zero and two:

```
static
  s: uns32[ 2, 2, 3];
  d: uns32[ 3, 2, 2];
```

Generally, one uses *array.transpose* to transpose a two-dimensional matrix. However, the transposition operation is defined for any number of dimensions. To understand how *array.transpose* works, it is instructive to look at the code you'd write to manually transpose the data in an array. Consider the transposition of the data in the *s* and *d* arrays above:

```
for( mov(0, i); i<2; inc(i)) do

  for( mov(0,j); j<2; inc(j)) do

     for( mov(0,k); k<3; inc(k)) do

        index( edx, s, i, j, k );
        mov( [edx], eax );
        index( edx, d, k, j, i );
        mov( eax, [edx] );

     endfor;

  endfor;

endfor;
```

Note that when storing away the value into the destination array, the *i* and *k* indicies were swapped. The following example demonstrates the use of *array.transpose*:

```
static
  s: uns32[2,3] := [1,2,3,4,5,6];
  d: uns32[3,2];

     .
     .
     .
  array.transpose( s, d );
     .
     .
     .

note: The code above copies s, as

  1 2 3
  4 5 6

to d, as

  1 4
  2 5
```

```
3 6
```

# 3.7 Lookup Tables

The *array.lookupTable* macro lets you easily construct a standard lookup table. This macro invocation must appear within a STATIC or a READONLY declaration section in your program. A lookup table declaration takes the following form:

```
readonly
  tableName:
     array.lookupTable
     (
        element_data_type,
        default_table_value,
        value: list_of_indexes,
        value: list_of_indexes,
            .
            .
            .
        value: list_of_indexes,
        value: list_of_indexes
     );
```

where:

*element_data_type* is the data type for each element of the array, for example, *byte*.

*default_table_value* is a value to use for "holes" in the table for which you do not supply an explicit index/value.

*value* is some value that you want to use to initialize a sequence of one or more table entries with.

*list_of_indexes* is a list of values that specify indexes into the lookup table. Each entry in a specific list is separated from the other entries with a space (not commas!). Note that each index value you specify must be unique across all lists of indexes in this table (that is, you cannot put two values into the array element specified by a single index). The *array.lookupTable* macro will report an error if you specify a non-unique index value in one of the lists.

Here is a concrete example:

```
static
  tableName:
     array.lookupTable
     (
        int32,
        -1,
        0: 1 2 3 4,
        1: 5 6 7 8,
        2: 9 10 11 12,
        3: 13 14 15,
        4: 20 22 24,
        5: 16 21 25,
        6: 23 19 18
     );
```

This declaration creates a table with 25 dword entries, that will hold the values 1..25. The table will be initialized as follows:

```
tableName:int32[25] :=
[
  0, 0, 0, 0, // Elements 0..3
```

```
    1, 1, 1, 1, // Elements 4..7
    2, 2, 2, 2, // Elements 8..11
    3, 3, 3,    // Elements 12..14
    5,          // Element 15
    -1,         // Element 16 (no index 17 specified above)
    6, 6,       // Elements 17 & 18
    4,          // Element 19
    5,          // Element 20
    4,          // Element 21
    6,          // Element 22
    4,          // Element 23
    5           // Element 24
];
```

The number of elements appearing in the lookup table will be the difference between the largest index value you supply in all the lists (25 in this example) and the smallest value (1 in this example) plus one. This particular lookup table has 25 entries because (25-1+1) = 25.

Note that each line in the example above specifies the value to store into each of the table entries in the list that immediately follows. This is probably backwards to what your intution would suggest. But the nice thing about this arrangement is that it lets you specify a single value to be placed into several different array indices. If there are any gaps in the array indexes you specify (as the value 17 is missing above), then the array.lookupTable macro will fill in those entries with the default value specified as the second parameter.

In order to access this lookup table at run-time, you must know the minimum index into the array so you can subtract this from the calculated index you use to access the table. The array.lookupTable macro generates four constants for you to help you do this (and other things):

*tableName*_maxValue
*tableName*_minValue
*tableName*_maxIndex
*tablename*_maxIndex

The *tableName*_minValue and *tableName*_maxValue constants specify the minimum and maximum index values for the table. In the current example, these constants would be 1 and 25, respectively. The tableName_minIndex and tableName_maxIndex values are the product of the array element's size with the _minValue and _maxValue constants. In the table above, the element size is four, so tableName_minIndex will be four and tableName_maxIndex will be 100. Whenever you access an element of the tableName array (in this example), you'll want to subtract the tableName_minIndex value from your computed index in order to adjust for non-zero starting indexes, e.g.,

```
mov( someIndex, ebx );
mov( tableName[ebx*4 - tableName_minIndex], eax );
```