PARALLAX SEMICONDUCTOR
www.parallaxsemiconductor.com
sales@parallaxsemiconductor.com
support@parallaxsemiconductor.com
phone: 916-632-4664 • fax:916-624-8003

Application note AN003

Implementing Abstract Data Structures with Spin Objects

Abstract: A number of programming techniques including parallel arrays, indexed arrays, and external objects are available to implement abstract data structures with named fields in Spin. These include multiple arrays, indexed single arrays, and external object arrays.

Introduction

Implementing user-defined data structures and multi-dimensional arrays in Spin is different than it is in C/C++, Java, Python, PHP and other high-level languages. With a few special considerations, Spin can do this for larger software projects requiring complex data structures that need to be accessible via named fields. For example, to create an array of the following records in C/C++:

Record Format

```
byte name[ 33 ]
byte age
byte height
byte weight
```

...make a simple type declaration such as:

```
struct typedef Person_typ
        {
        char name[ 32 ] ;
         byte age;
         byte height;
         byte weight;
                } Person;
```

...then instantiate an array *Persons[]* of these records with:

```
Person Persons[ NUM_RECORDS ];
```

This kind of programming pattern is very common. A “container” data structure is created then an array, list, file, etc. is built of the constituent data structures or records as a linear array.

Even to those not familiar with the C/C++ syntax above, it is clear that *Person* contains the fields of the desired record format. A simple dot or arrow operator in C/C++ can access these fields:

```
Persons[ 12 ].age = 25;
```

This assigns the value 25 to the *age* field in index 12 of the array.

Achieving this is the kind of flexibility in Spin, with respect to user-defined data structures and types requires a few programming patterns and transformations:

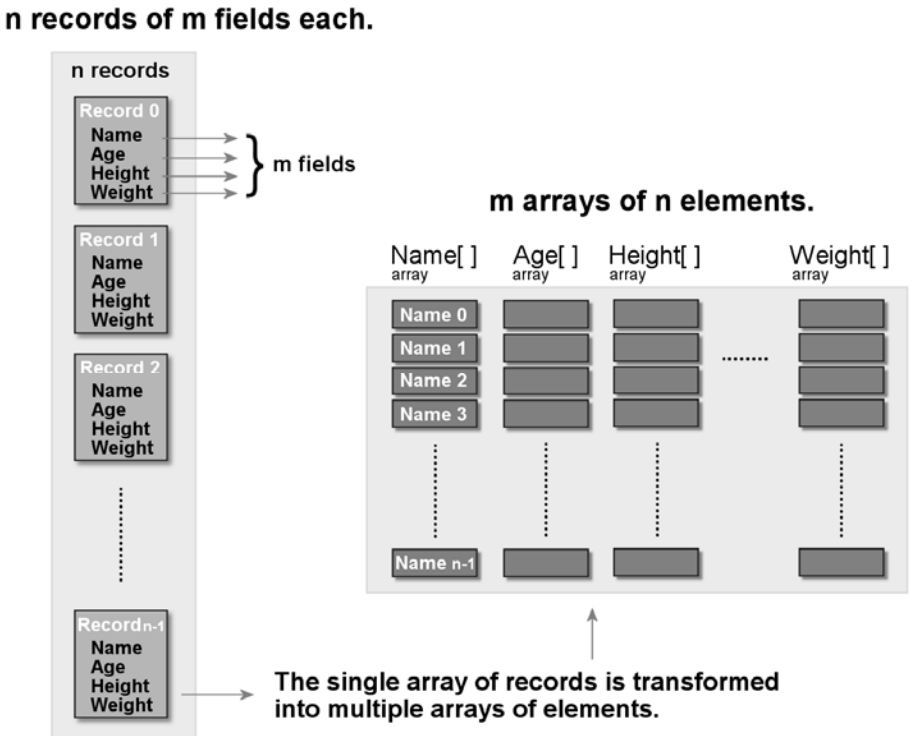
- **Multiple Arrays** — This technique uses a multiple arrays where each array represents one field of the data structure. In other words, an array of n records where each record has m fields is transformed into m arrays of n elements each.
- **Indexed Single Array** — This technique uses a single array to hold records linearly in place much as the computer would internally store them, if the language supported user-defined data structures. With this format, an indexing scheme is used to access each record based on the prior knowledge of the size of each field in the record.
- **External Object Array** — This is the most advanced technique and the most robust of the three approaches. Spin allows not only external objects that are drivers and functional in nature, but also can use objects as actual data structures/containers. Additionally, Spin allows not only single objects to be declared, but also arrays of the same object, and then using the dot operator to access methods in those objects. This technique develops into a very object-oriented approach: user-defined data structures implemented as objects.

The following sections illustrate the methodology and programming pattern of each approach with theory and a supporting demo program.

Multiple Arrays Approach

The idea behind the multiple arrays approach is to take a record format and instead of having a single array of ***n*** records with ***m*** fields each, transform the data into ***m*** arrays each with ***n*** elements. This is shown graphically in Figure 1 below.

Figure 1: Transforming an array of n records of m fields into m arrays of n elements



Referring to Figure 1, the transformation is straightforward. Begin with an abstract data structure with m fields each representing one element of the abstract data type or record. In this case, the idea is to create a data base that holds information about people, so the elements or fields of the record are:

```
' name   - string
' age    - integer
' height - integer
' weight - integer
```

Then the next step is to formalize what intrinsic Spin data type to use to represent each field. In this case, bytes work for all the data types:

```
{{
byte name - string[33] ' holds a string 32 characters long with null terminator
byte age               ' holds ages up to 255 years
height                 ' hold height up to 255 inches
weight                 ' holds weights up to 255 lbs
```

The next step is to create a number of arrays that each hold's one element of the entire record, and then access them in parallel:

```
CON

NUM_RECORDS = 16

VAR

byte name[33*NUM_RECORDS]
byte age[NUM_RECORDS]
byte height[NUM_RECORDS]
byte weight[NUM_RECORDS]
```

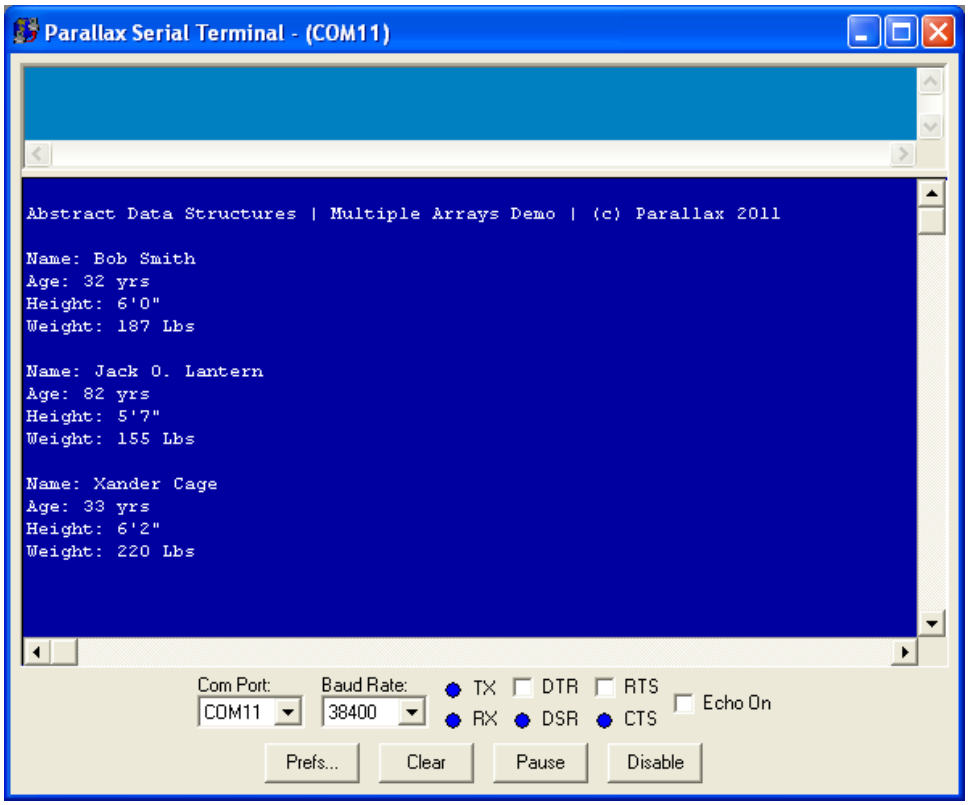
A list of byte variable declarations implements the record format as a set of parallel arrays.

Note that the declaration of **name** is a little different from the others. Without the support for multi-dimensional arrays in Spin, we either have to use an array of pointers or statically store the strings in place, each requiring 33 bytes of storage. The latter is the tactic used here. An array of pointers is more flexible, but slightly more complex to maintain and work with since the actual string storage for each string would have to be defined in a VAR or DAT section then pointer links made to each string.

Example 1: Demo of using multiple arrays to implement an array of the Person record format

Load the sample application top-level file ADS_MultipleArraysDemo_010.spin, from the AN003 Resources zip archive. It requires a Propeller development board with a serial connection to display output via the Parallax Serial Terminal[1] or any other serial terminal program. Be sure to set the serial terminal to 38.4 K baud, and modify the demo source lines that set the TX and RX lines of the serial terminal in the CON section of the code to the appropriate serial pins on your board. In addition, the source assumes a 5 MHz crystal , so change as needed. Figure 2 shows the output of the demo program.

Figure 2: The Multiple Arrays Demo running and outputting records to the Parallax Serial Terminal



When using the Parallax Serial Terminal (or any COM port terminal for that matter) download the program to the Propeller device first with F11, so it is stored in EEPROM. Next, switch on the terminal to take over the COM port, and then re-boot the Propeller programming board. When you are done with the demo, make sure to release the COM port (Enable/Disable) on the Parallax Serial Terminal.

The program starts by defining a number of arrays that make up the storage records, as shown below:

```
VAR
' -------------------------------------------------------------------------
' DECLARED VARIABLES, ARRAYS, ETC.
' -------------------------------------------------------------------------

' these arrays hold our record(s), we want to store an array of records that
' store a person's record based on the following abstract data structure"

' byte name[33]
' byte age
' byte height
' byte weight

byte gPersonName[ MAX_PERSONS*MAX_NAME_LENGTH ]    ' we must use a statically declared
                                                   ' array of fixed length strings
                                                   ' since SPIN doesn't support
multidimensional arrays
byte gPersonAge[ MAX_PERSONS ]                     ' age in years
byte gPersonHeight[ MAX_PERSONS ]                  ' height in inches
byte gPersonWeight[ MAX_PERSONS ]                  ' weight in pounds
```

Then the program inserts these records into the “database” using calls to a support method InsertRecord:

```
  ' person 0: Bob Smith, 32 yrs, 6', 187 lb
  ' person 1: Jack O. Lantern, 82 yrs, 5'7", 155 lb
  ' person 2: Xander Cage, 33 yrs, 6'2", 220 lb

  InsertRecord( 0, string("Bob Smith"), 32, 6*12, 187 )
  InsertRecord( 1, string("Jack O. Lantern"), 82, 5*12+7, 155 )
  InsertRecord( 2, string("Xander Cage"), 33, 6*12+2, 220 )
```

Using a support method is a clean strategy rather than doing the insertions manually since we can separate the user interface from the data structure implementation. This flexibility allows for use of different storage techniques without changing a lot of code. Only the final insertion, deletion, and access methods must change.

Finally, the records are “pretty” printed to the serial terminal with calls to PrintRecord which is another support method written to make printing records easy.

```
  ' now print the records out
  repeat index from 0 to 2
    PrintRecord( index )
```

Here are the two methods that show how to both write and read the parallel arrays respectively. The InsertRecord method writes the arrays in parallel:

```
PUB InsertRecord( pIndex, pStrNamePtr, pAge, pHeight, pWeight )
{{
DESCRIPTION: Inserts the sent record into the storage array.
PARMS:       pIndex      - index of record to use.
             pStrNamePtr - pointer to name string to insert.
             pAge        - age of person.
             pHeight     - height in inches of person.
             pWeight     - weight in pounds of person.
RETURNS: nothing.
}}

  ' copy name string
  bytemove ( @gPersonName[ pIndex*MAX_NAME_LENGTH ], ←
                                        pStrNamePtr, strsize( pStrNamePtr )+1 )

  gPersonAge   [ pIndex ] := pAge
  gPersonHeight[ pIndex ] := pHeight
  gPersonWeight[ pIndex ] := pWeight

' end PUB ---------------------------------------------------------------------
```

The only complexity in this case is the access to the string array, which is really a contiguous array of bytes that holds multiple name strings in place adjacent to each other. Thus, to store 16 strings of 1 character each, a lot of statically wasted space would result since each string is 33 bytes if it is needed or not.

Next, the method that reads the arrays in parallel and prints to the terminal:

```
PUB PrintRecord( pIndex ) | feet, inches

{{
DESCRIPTION: Prints the requested record to terminal.
PARMS: pIndex - index of record to pretty print to screen.
RETURNS: nothing.
}}

  ' convert height to feet and inches from inches
  feet   := gPersonHeight[ pIndex ] / 12
  inches := gPersonHeight[ pIndex ] // 12

  serial.tx( ASCII_CR )
  serial.txstring( string ("Name: "))
  serial.txstring( @gPersonName[ pIndex*MAX_NAME_LENGTH ] )
  serial.tx( ASCII_CR )

  serial.txstring( string ("Age: "))
  serial.dec( gPersonAge[ pIndex ] )
  serial.txstring( string (" yrs"))
  serial.tx( ASCII_CR )

  serial.txstring( string ("Height: "))
  serial.dec( feet )
  serial.tx( ASCII_SNGL_QUOTE)
  serial.dec( inches )
  serial.tx( ASCII_QUOTE )
  serial.tx( ASCII_CR )

  serial.txstring( string ("Weight: "))
  serial.dec( gPersonWeight[ pIndex ] )
  serial.txstring( string (" Lbs"))
  serial.tx( ASCII_CR )

' end PUB ---------------------------------------------------------------------
```

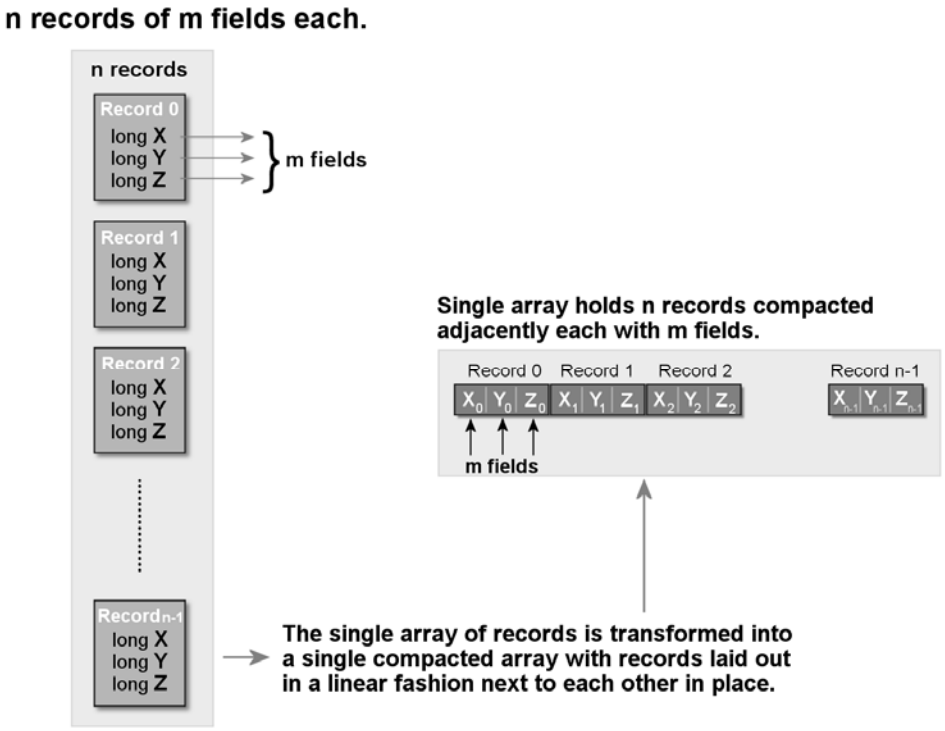
Of course, all this demo code is to show the technique in action. Your particular problem might be different, but the data structure transform technique can be used in many cases, so that the data can be referred to by an array name (for each field of the original data structure) along with an index.

Using multiple arrays is somewhat cumbersome syntactically, but from a memory allocation and efficiency point of view, it is very effective. There isn't any wasted memory, and access time is very fast. The only downside is instead of accessing a single record then drilling down to a field or element of that record with a dot operator, you must access an entire array of only the elements in question. Thus, the memory footprint and computation loads are the same as they would be with an array of user-defined typed records, but the syntax is always multiple array based.

Indexed Single Array Approach

The indexed single array approach compacts all the data into a single array. Thus, it transforms ***n*** records of ***m*** fields into a single array that has size ***n*sizeof(m fields)***.

Figure 3: Transforming *n* records of *m* fields into a single array of *n*sizeof(m fields)*



For example, implement a record that held points in 3D space (x,y,z) each with a single `long`. With the parallel arrays method in the above section, make the following declarations to hold 100 records (points):

```
long x[ 100 ]
long y[ 100 ]
long z[ 100 ]
```

However, using a single array, the elements compact into the array adjacent to each other with a declaration like this:

```
long xyz[ 100 * 3 ]
```

The multiplication by 3 creates the space for each (`x`,`y`,`z`) point since each takes 3 longs.

Using a single array has the single advantage that it is easier to move around, without worrying about copying multiple arrays and carrying them around. However, access to the individual records in the array becomes more complex and an indexed multiplication scheme is necessary—hence the name “index array method.” Moreover, using Spin to perform the computations uses up more cycles than the parallel/multiple array approach. Nonetheless, the gains of using a single array might outweigh the more complex access techniques needed.

As an example, let's re-visit the initial model problem of representing an array of records each holding a Person with the following fields:

```
{{...
byte name - string[33] ' holds a string 32 characters long with null terminator
byte age               ' holds ages up to 255 years
height                 ' hold height up to 255 inches
weight                 ' holds weights up to 255 lbs
```

The declaration of the single indexed array would look like this:

```
CON

MAX_PERSONS       = 16 ' 16 records to play with
BYTES_PER_PERSON  = 36 ' number of bytes each "person" record takes

PERSON_INDEX_NAME   = 0  ' offset of "name" field in a record
PERSON_INDEX_AGE    = 33 ' offset of "age" field in a record
PERSON_INDEX_HEIGHT = 34 ' offset of "height" field in a record
PERSON_INDEX_WEIGHT = 35 ' offset of "weight" field in a record

VAR

' record format:
'
' name:   33 bytes
' age:    1 byte
' height: 1 byte
' weight  1 byte
'
' each "record" is 36 bytes long
'
byte gPerson[ MAX_PERSONS*BYTES_PER_PERSON ]
```

The interesting thing to note is that we must pre-compute the size of each compacted record, so we can later use a multiplication to find the base index of the record in question. In this case, there are 36 bytes per record. Use the following syntax to access any record:

```
gPerson[ index * BYTES_PER_PERSON]
```

Assuming each record is in the format given in the template shown above, **name** (33 bytes), **age** (1 byte), **height** (1 byte), **weight** (1 byte), use the following code to index in each virtual field of the compacted record:

```
gPerson[ index* BYTES_PER_PERSON + "offset of field in record"]
```

The "offset of field in record" would be 0 for the **name** field, 33 for the **age**, 34 for the **height**, and finally 35 for the **weight** field, thus the following code would access the name and height for example using the constants declared in the code list above:

```
' access name
gPerson[ index* BYTES_PER_PERSON + PERSON_INDEX_NAME]

' access weight
gPerson[ index* BYTES_PER_PERSON + PERSON_INDEX_WEIGHT]
```

Astute Spin programmers might notice the opportunity for a slight syntactic simplification using Spin's built in "base + offset" array syntax like this:

```
' access name
gPerson[ index* BYTES_PER_PERSON ][ PERSON_INDEX_NAME ]

' access weight
gPerson[ index* BYTES_PER_PERSON ][ PERSON_INDEX_WEIGHT ]
```

Use whichever suits; they are both roughly the same. The latter form might be slightly faster since the Spin compiler doesn't need to generate code for an external addition, but uses the built-in code for base + index array mode.

Example 2: Demo of using a single indexed array to implement an array of the Person record format

Load the sample application top-level file ADS_IndexedArrayDemo_010.spin. It requires a Propeller development board with a serial connection to display its output in the Parallax Serial Terminal or any other serial terminal program. Be sure to set the serial terminal to 38.4 K baud, and modify the demo source lines that set the TX and RX lines of the serial terminal in the CON section of the code. Also, the source assumes a 5 MHz crystal as well, so change as needed.

Note that the demo's output is almost identical to the multiple arrays demo in Figure 2, so refer to that for what the serial terminal output should look like.

Since the demo code uses methods to insert and print the records, the majority of the program doesn't need to be changed. Only the declaration of the data structure itself (a single array) changes. Then the main startup code is identical that calls to insert the records and print them. Only the functional bodies of InsertRecord and PrintRecord change. This allows us to re-use a great deal of software if this design pattern is maintained in your programming practices. With that in mind, here are the two new methods that work with a single indexed array. First, the InsertRecord method:

```
PUB InsertRecord( pIndex, pStrNamePtr, pAge, pHeight, pWeight ) | recordOffset
{{
DESCRIPTION: Inserts the sent record into the storage array.
PARMS:       pIndex      - index of record to use.
             pStrNamePtr - pointer to name string to insert.
             pAge        - age of person.
             pHeight     - height in inches of person.
             pWeight     - weight in pounds of person.
RETURNS: nothing.
}}

  ' first compute the offset to access the record we want, a simple multiplication by
  ' the size of each record
  recordOffset := pIndex * BYTES_PER_PERSON

  ' now when we access each field of the compressed data, we use the base offset just
  ' computed along with the "field" indices defined as constants, this gives some feel
  ' of typed data structure

  ' copy name string
  bytemove ( @gPerson[ recordOffset + PERSON_INDEX_NAME], pStrNamePtr, ←
                                                          strsize( pStrNamePtr )+1 )

  ' and now other field, notice the simple syntax
  ' transform a single addition along with the computed
  ' base address/offset is all that is required to access each element properly
```

```
  gPerson[ recordOffset + PERSON_INDEX_AGE ]    := pAge
  gPerson[ recordOffset + PERSON_INDEX_HEIGHT]  := pHeight
  gPerson[ recordOffset + PERSON_INDEX_WEIGHT ] := pWeight

' end PUB ---------------------------------------------------------------------
```

Notice all the indexing arithmetic that is required now, but all on a single array. Next is the updated `PrintRecord` method:

```
PUB PrintRecord( pIndex ) | feet, inches, recordOffset
{{
DESCRIPTION: Prints the requested record to terminal.
PARMS: pIndex - index of record to pretty print to screen.
RETURNS: nothing.
}}

  ' first compute the offset to access the record we want,
  ' a simple multiplication by the size of each record
  recordOffset := pIndex * BYTES_PER_PERSON

  ' convert height to feet and inches from inches
  feet   := gPerson[ recordOffset + PERSON_INDEX_HEIGHT ] / 12
  inches := gPerson[ recordOffset + PERSON_INDEX_HEIGHT ] // 12

  serial.tx( ASCII_CR )
  serial.txstring( string ("Name: "))
  serial.txstring( @gPerson[ recordOffset + PERSON_INDEX_NAME ] )
  serial.tx( ASCII_CR )

  serial.txstring( string ("Age: "))
  serial.dec( gPerson[ recordOffset + PERSON_INDEX_AGE ] )
  serial.txstring( string (" yrs"))
  serial.tx( ASCII_CR )

  serial.txstring( string ("Height: "))
  serial.dec( feet )
  serial.tx( ASCII_SNGL_QUOTE)
  serial.dec( inches )
  serial.tx( ASCII_QUOTE )
  serial.tx( ASCII_CR )

  serial.txstring( string ("Weight: "))
  serial.dec( gPerson[ recordOffset + PERSON_INDEX_WEIGHT ] )
  serial.txstring( string (" Lbs"))
  serial.tx( ASCII_CR )

' end PUB ---------------------------------------------------------------------
```

The two previous approaches based on arrays both have their pros and cons. The multiple arrays approach executes faster and access is simpler. However, the single indexed array approach allows declaring a single source of data, which makes things like storing to a flat file on SD card or memory easier. The final approach presented here is the most complex of the three, but the most robust and flexible as well—using objects as data containers.

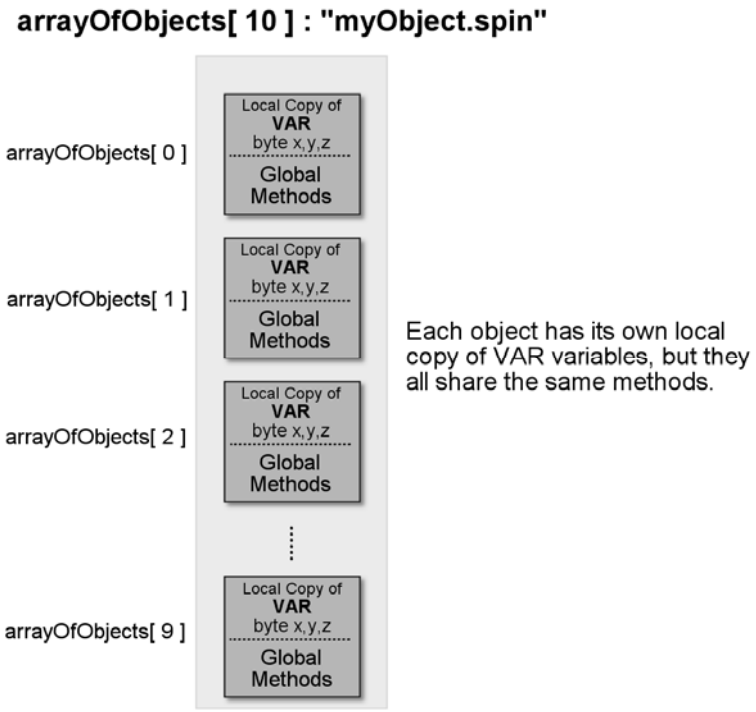
External Object Array Approach

Those new to Spin may not be aware that *arrays* of objects can be declared:

```
arrayOfObjects[ 10 ] : "myObject.spin"
```

This syntax generates ten copies of the object, as shown in Figure 4 below.

Figure 4: Declaring object arrays



As an example, assume the object file myObject.spin has the following code in it:

```
' myObject.spin contents

VAR

byte x,y,z

PUB DoNothing
```

And that's all. This is totally valid and the result will be 10 objects, each with its own "local" VAR memory of three 1–byte variables named x,y,z. Now, you might be tempted to think you can access the variables like this:

```
arrayOfObjects[ 4 ].x := 5 ' this won't work, but good try!
```

Unfortunately, this syntax will not work, but, it's close. There are two problems with this attempt. First, every single Spin object needs at least one PUB method even if it doesn't do anything (that's what PUB DoNothing is for)—this will be replaced by methods in a moment though.

The second and more important problem is that the access syntax of the 4th element won't work and is illegal. Spin can't access the global variables from an external object whether it's a singleton or array. However, Spin ***can*** access and call a child object's public methods and this is exactly how to gain read/write access to the global variables of the object instantiation. Thus, if methods are added to access the object's declared variables, these methods can be used to read and write the variables from the calling object.

With this in mind, adding some code to the myObject.spin file with accessor methods might look like this:

```
' myObject.spin

VAR
' the record and data structure is defined here
byte x,y,z

' read methods
PUB Read_x
  return x

PUB Read_y
  return y

PUB Read_z
  return z

' write methods
PUB Write_x( val )
  x := val

PUB Write_y( val )
  y := val

PUB Write_z( val )
  z := val
```

Now, with this new definition of myObject.spin a declaration of 10 of them is made:

```
arrayOfObjects[ 10 ] : "myObject.spin"
```

At this point the object methods can be used to indirectly access the variables of the object. For example, to read the value of **x**, the following code will work:

```
arrayOfObjects[ 4 ].Write_x( 5 )
```

This makes a method call with the parameter equal to the value to be written; other than that syntax awkwardness, this technique in essence allows arrays of objects to be declared where each instance in the array has its own copy of variables (in its **VAR** section). The only heavy lifting needed is to access the variables with "setter" and "getter" methods like **Write_x** above.

Moving onto reading, the slightly inconvenient method call format becomes even more natural for reading a value as shown below:

```
speed := arrayOfObjects[ 4 ].Read_x
```

The right hand side actually makes a method call to the 4th element in the **arrayOfObjects** array to **Read_x**, but it looks like a variable access with a dot operator which is nice. If we want to add a little more syntactic simplification then we can make the method names nearly identical to the actual variables—and using the axiom "of simple is good"—a single underscore "_" before and after each variable name denotes "read" and "write" respectively. Thus, using this convention the methods might look like this:

```
' myObject.spin

VAR
' the record and data structure is defined here
byte x,y,z

' read methods, prefix with _ to denote read
PUB _x
  return x

PUB _y
  return y

PUB _z
  return z

' write methods, suffix with _ to denote write
PUB x_( val )
  x := val

PUB y_( val )
  y := val

PUB z_( val )
  z := val
```

Now there's a little cleaner write syntax:

```
arrayOfObjects[ 4 ].x_( 5 )
```

But, the read syntax transforms completely:

```
speed := arrayOfObjects[ 4 ]._x
```

This is nearly identical to C/C++, Java, PHP, or any other high-level, object oriented language that supports objects with data and methods. Moreover, it has the additional benefit of the caller never having direct access to the actual variables themselves. The caller must use the accessor methods, which again is yet another bonus called "data hiding" — an object-oriented feature of classes. This way, if the actual storage of the data changes slightly, as long as the accessor method interfaces don't change, high-level applications will continue to function and won't break. Thus, data structures are de-coupled from methods, as they should be.

Converting the Person Database to Objects

Using the design pattern from the paragraphs above, transforming the Person demo to use object arrays is almost trivial. The first step is to create the object that will hold each record and methods. The file, named ADS_PersonObject_010.spin is shown below with extraneous white space and some comments removed.

```
' -------------------------------------------------------------------------
' CONSTANTS, DEFINES, MACROS, ETC.
' -------------------------------------------------------------------------

  ' string processing constants
  MAX_NAME_LENGTH   = 33 ' max length of a person's name, 32 plus a NULL

VAR
' -------------------------------------------------------------------------
' DECLARED VARIABLES, ARRAYS, ETC.
' -------------------------------------------------------------------------
' Each time the caller creates one of these "objects" another set of these variables
' are created that are local to this object. The idea is for the calling application
' to create an object array for the data structure/record and then use accessor
' methods to access the variables of each object array record
byte name[ MAX_NAME_LENGTH ] ' string as usual
byte age
byte height
byte weight

CON
' -------------------------------------------------------------------------
' GETTER METHODS FIRST, convention will be to prefix "read" methods with "_"
' -------------------------------------------------------------------------

PUB _name
  return ( @name )

PUB _age
  return ( age )

PUB _height
  return ( height )

PUB _weight
  return ( weight )

CON
' -------------------------------------------------------------------------
' SETTER METHODS NEXT, convention will be to suffix "write" methods with "_"
' -------------------------------------------------------------------------

PUB name_( pStrPtr )
  bytemove( @name, pStrPtr, strsize( pStrPtr ) + 1)

PUB age_( pAge )
  age := pAge

PUB height_( pHeight )
  height := pHeight

PUB weight_( pWeight )
  weight := pweight
```

The design pattern illustrated with the **x**,**y**,**z** point example previously is followed verbatim and the leading and trailing underscore syntax is used to represent read and write methods respectively. Now, with this object in hand, the main demo can import and use it to create an object array that represents the Person database. The following example illustrates a complete example of the object array pattern.

Example 3: Demo of object arrays to implement an array of the Person record format.

Load the sample application top-level file named ADS_ObjectArrayDemo_010.spin. It requires a Propeller programming board with a serial connection to display its output; use the Parallax Serial Terminal or any other serial terminal program. Be sure to set the serial terminal to 38.4 K baud, and modify the demo source lines that set the TX and RX lines of the serial terminal in the CON section of the code. Also, the source assumes a 5 MHz crystal as well, so change as needed. The demo's output is the same as the others, so refer to Figure 2.

Reviewing the source of the demo, the OBJ section declares the object array with the single line of code:

```
' create an array of "person" objects
  gPerson[ MAX_PERSONS ] : "ADS_PersonObject_010.spin"
```

Then the entire demo remains the same (again this is due to the fact that the insertion and printing methods are abstracted away). The only things that change are the body of the insertion and printing methods InsertRecord and PrintRecord which are both shown below.

```
PUB InsertRecord( pIndex, pStrNamePtr, pAge, pHeight, pWeight )
{{
DESCRIPTION: Inserts the sent record into the object storage array.
PARMS:       pIndex      - index of record to use.
             pStrNamePtr - pointer to name string to insert.
             pAge        - age of person.
             pHeight     - height in inches of person.
             pWeight     - weight in pounds of person.
RETURNS: nothing.
}}

  ' now we access the records as "method" calls to our object array, the syntax
  ' is a little rough, but surely better than all the indexing and contrived
  ' arrays of the previous examples, now we have a nice layer of abstraction

  ' now write the fields, syntax is a little tricky since to write to any
  ' field we have to make a call to the setter method, and then pass the value
  ' as a parameter, but a couple parens is really all we need syntactically that
  ' takes the place of ":=" if we could support operator overloading, but can't
  ' NOTE: all write methods have a trailing underscore "_" this is so we can
  ' keep the name similar to the data fields and remember to put an underscore
  ' before for getter, underscore after for setter

  gPerson[ pIndex ].name_( pStrNamePtr )
  gPerson[ pIndex ].age_( pAge )
  gPerson[ pIndex ].height_( pHeight )
  gPerson[ pIndex ].weight_( pWeight )

' end PUB ---------------------------------------------------------------------
```

The majority of the method is actually comments now. Due to the object oriented use of external objects, the code to write each field is trivial. Moreover, the code now has a modern "OO" feel to it. The use of dot operators and named fields are both supported with the slight syntactical aberration of making a method call with the data in parentheses to write each field. Other than that, we have achieved what we set out to do. Moving onto the printing method, it's even more elegant when reading object data.

```
PUB PrintRecord( pIndex ) | feet, inches
{{
DESCRIPTION: Prints the requested record to terminal.
PARMS: pIndex - index of record to pretty print to screen.
RETURNS: nothing.
}}

  ' this is where the object really shines, reading values is a snap
  ' syntactically
  ' NOTE: all write methods have a trailing underscore "_" this is so we can
  ' keep the name similar to the data fields and remember to put an underscore
  ' before for getter, underscore after for setter

  ' convert height to feet and inches from inches
  feet   := gPerson[ pIndex ]._height / 12
  inches := gPerson[ pIndex ]._height // 12

  serial.tx( ASCII_CR )
  serial.txstring( string ("Name: "))
  serial.txstring( gPerson[ pIndex ]._name )
  serial.tx( ASCII_CR )

  serial.txstring( string ("Age: "))
  serial.dec( gPerson[ pIndex ]._age )
  serial.txstring( string (" yrs"))
  serial.tx( ASCII_CR )

  serial.txstring( string ("Height: "))
  serial.dec( feet )
  serial.tx( ASCII_SNGL_QUOTE)
  serial.dec( inches )
  serial.tx( ASCII_QUOTE )
  serial.tx( ASCII_CR )

  serial.txstring( string ("Weight: "))
  serial.dec( gPerson[ pIndex ]._weight )
  serial.txstring( string (" Lbs"))
  serial.tx( ASCII_CR )

' end PUB ---------------------------------------------------------------------
```

Reviewing the method, you can see that reading data from the object is seamless. By using methods with nearly the same names as the data elements themselves, the object array performs nearly like a user-defined type.

Summary

This application note has reviewed three separate software patterns and transformations allowing complex and rich data structures to be implemented with Spin. The first two techniques multiple arrays and indexed arrays are fast and easy to implement, but still feel primitive. The final technique using object arrays leverages the very powerful concept of object arrays in a non-intuitive way where instead of having multiple drivers or code objects, the objects are used to store single records with a set of accessor methods. In fact, this mimics classes found in languages like C++, Java, and other modern object oriented languages.

Resources

Download a zip archive with the following example files from this application note's web page: www.parallaxsemiconductor.com/an003.

ADS_IndexedArrayDemo_010.spin
ADS_MultipleArraysDemo_010.spin
ADS_ObjectArrayDemo_010.spin
FullDuplexSerial_drv_014.spin

http://www.parallax.com/Portals/0/Downloads/sw/propeller/Parallax-Serial-Terminal.exe

References

1. The Parallax Serial Terminal is available alone and is also included with the Propeller Tool Software. Download both from www.parallaxsemiconductor.com/software.

Revision History

Version 1.0: original document.