Structures and Unions

[**private** | **public**] **structure** identifier
  variable-declaration
 {variable declaration}
end structure

- *Private* – An optional keyword which ensures that a structure is only available from within the module it is declared. Structures are private by default.
- *Public* – An optional keyword which ensures that a structure is available to other programs or modules.
- *Identifier* – A mandatory type name, which follows the standard identifier naming conventions
- *Variable-declaration* – One or more variable declarations. Supported types include boolean, bit, byte, word, longword, shortint, integer, longint, float, string, char and structures

A structure is a collection of one or more variable declaration fields. Each field can be a different data type. A structure is an extremely useful and powerful feature of the Swordfish language which enables you to assemble dissimilar elements under one single roof.

To better understand structures, the following example illustrates how to create a new structure called TTime,

```
structure TTime
   Hours as byte
   Minutes as byte
end structure
```

The declaration above informs the compiler that *TTime* contains two byte fields (Hours and Minutes). We can now create a variable of type *TTime*, in exactly the same way as you would any other compiler type, such as byte or float,

```
dim Time as TTime
```

Access to an individual field within the variable *Time* is achieved by using the dot (.) notation,

```
Time.Hours = 9
Time.Minutes = 59
```

A structure can also use another structure in one or more of its field declarations. For example,

```
structure TSample
   Time as TTime
   Value as word
end structure
dim Sample as TSample
```

We now have a type called *TSample*, who's field members include *Time* (of type *TTime*) and *Value* (of type *word*). Again, dot (.) notation is used to access individual field elements,

```
Sample.Time.Hours = 15
Sample.Time.Minutes = 22
Sample.Value = 1024
```

Structures can also be used with arrays. For example, using the previously declared *TSample* type, we could declare and access multiple TSample variables by declaring an array,

```
dim Samples(24) as TSample // array of samples, one every hour
```

To access each field for every array element, we just need to iterate through the samples array,

```
dim Index as byte
for index = 0 to bound(Samples)
   Samples(Index).Time.Hours = 0
   Samples(Index).Time.Minutes = 0
   Samples(Index).Value = 0
next
```

The above code is actually a very verbose way of initializing all fields to zero, but it does demonstrate how each field can be accessed. It should be noted that by using the inbuilt compiler command **clear**, the above can be achieved by using,

```
clear(Samples)
```

Unions

In the previous structure example, the total size of the structure is the sum of all members of the structure. For example, TTime has two member fields (hours and minutes) and each field is one byte in size. Therefore, the total size of the structure is two bytes. A union works differently in that member fields can share the same address space. For example,

```
structure TStatus
   Val as byte
   Enabled as Val.0
   Connected as Val.1
   Overrun as Val.2
end structure
```

The member fields *enabled*, *connected* and *overrun* are aliased to the byte variable *Val*. They don't have separate storage requirements - they are shared with Val. For example,

```
dim MyStatus as TStatus
MyStatus.Val = 0 // clear status
MyStatus.Connected = 1
```

In the above example, we can access the structure as a byte value or access individual bits. Importantly, the total structure size is only one byte. You can apply all the standard aliasing rules to structures. For example,

```
structure TIPAddr
   Val(4) as byte
   IP as Val(0).AsLongWord
end structure
dim IPAddr as TIPAddr
IPAddr.IP = $FFFFFFFF
IPAddr.Val(0) = $00
```

In this example, the IP address structure only uses 4 bytes of storage. In some cases, it may not be possible to create a union through aliasing alone. For example, the member field type may be another structure. In these situations, you can use the **union** keyword, like this:

```
structure TWord
   LSB as byte
```

```
   MSB as byte
end structure

structure TValue
   ByteVal as byte union
   WordVal as TWord union
   FloatVal as float union
end structure
```

In the above example, the size of the structure is equal to the size of the largest member field which is 4 bytes (the size of float). Another way to think of the union keyword is that it 'resets' the internal offset address of the member field to zero. For example,

```
Structure TValue
   FloatVal As Float     // offset = 0 (0 + 4 byte = 4)
   WordVal As Word       // offset = 4 (4 + 2 byte = 6)
   ByteVal As Byte       // offset = 6 (6 + 1 byte = 7)
End Structure            // total storage requirement = 7
```

The above structure declaration shows the starting offset address, with the total storage requirement for the structure. Now take a look at the same structure, but this time with the **union** keyword:

```
Structure TValue
   FloatVal As Float Union  // offset = 0 (0 + 4 byte = 4)
   WordVal As Word Union    // offset = 0 (0 + 2 byte = 2)
   ByteVal As Byte Union    // offset = 0 (0 + 1 byte = 1)
End Structure               // total storage requirement = 4
```