Variables

[**private** | **public**] **dim** identifier {, identifier} **as** type

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

A variable holds data on which a program operates. Unlike constants, variable values can change dynamically when the program is executing. A variable is like a box, which holds values. You have to tell the compiler in advance the type of variable that will fit into the box.

You can declare variables one at a time, like this

```
dim Index as byte
dim Range as byte
```

or you can declare them as a list,

```
dim
   Index as byte,
   Range as byte
```

In the examples opposite, the variables are of the same type (a byte). You could therefore use the following syntax

```
dim Index, Range as byte
```

As mentioned previously, the type defines what values can fit into a variable. It's important to note that data RAM on a PIC™ microcontroller is substantially less than the code memory used to store your program. In addition, program operations on large data types (for example, long words) will generate more underlying ASM code.

Type	Bit Size	Range
Boolean	1	True or False
Bit	1	1 or 0
Byte	8	0 to 255
Word	16	0 to 65535
LongWord	32	0 to 4294967295
ShortInt	8	-128 to 127
Integer	16	-32768 to 32767
LongInt	32	-2147483648 to 2147483647
Float	32	-1e37 to +1e38
Char	8	Single character
String	Variable	Multiple (up to 255) characters
Structure	Variable	Variable

The PIC™ 18 series is an 8 bit microcontroller, so it makes sense to keep your types limited to

unsigned bytes if at all possible. For example, you may want to store a numeric value which ranges from 0 to 200. In this case, a byte type would be ideal, as this can store numbers in the range 0 to 255 and only takes 8 bits of data RAM. Of course, the compiler can easily accommodate larger types, but choosing the right variable type is essential not only in terms of saving precious data RAM, but also in terms of the size and efficiency of the ASM code produced.

Unlike many other BASIC compilers, Swordfish does allow variables of different types to be used in the same expression. For example, an unsigned byte can be multiplied by an integer and assigned to a variable declared as floating point. However, this practice should be avoided if possible, as the code automatically generated by the compiler needs to convert one type into another in order to compute the correct result. This will result in a larger code footprint than would otherwise be generated if all of the variables used had been declared as the same type.

The types *bit, byte, word, longword, shortint, integer, longint* and *float* clearly outline the numeric ranges for any variables declared using them.

The following sections discuss in more detail *boolean, string* and *char.*

Boolean Types

The boolean data type enables you to represent something as **true** or **false**. It cannot hold a numeric value. The right hand side of an assignment expression must always evaluate to **true** or **false**, or set directly by using the compilers predefined boolean constants. For example,

```
dim OK as boolean
OK = true
```

Booleans are particularly useful when dealing with flow control statements, such as **if...then** or iteration statements, such as **while...wend** or **repeat...until**. A Boolean data type can significantly contribute to the readability of a program, making code sequences appear more logical and appropriate.

For example, the following code shows how you could set a *bit* flag, if the value of *index* falls within 10 and 20,

```
dim Index as byte
dim DataInRange as bit
if Index >= 10 and Index <= 20 then
   DataInRange = 1
else
   DataInRange = 0
endif
```

However, if we change the flag *DataInRange* to a boolean type, we could write the code like this,

```
dim Index as byte
dim DataInRange as boolean
DataInRange = Index >= 10 and Index <= 20
```

In the first example, testing *index* using **if...then** evaluates to **true** or **false**. In the second example, *DataInRange* is a boolean type, so we can dispense with the **if...then** statement altogether and assign the result directly to *DataInRange*.

In addition, because DataInRange is a boolean type, we don't have to explicitly test it when encountering any conditional expressions. Remember, a boolean can only be **true** or **false**, so we

simply write something like this,

```
if not DataInRange then
   // output an error
endif
```

In this example, if *DataInRange* is false, then the **if...then** statement will evaluate to true (the boolean operator **not** inverts the false into a true) and an error is output.

String and Char Types

A string variable can be described as a collection of character elements. For example, the string "Hello" consists of 5 individual characters, followed by a null terminator. The Swordfish compiler uses a null terminator (0) to denote the end of a string sequence. A string variable can be declared and initialized in the following way,

```
dim MyString as string
MyString = "Hello World"
```

By default, the compiler will allocate 24 bytes of RAM for each string declared. That is, you can assign a string sequence of up to 23 characters, plus one for the null terminator.

In the previous example, "Hello World" is 11 characters long. Assuming *MyString* will never get assigned a sequence larger than this, we can save some RAM storage by explicitly specifying the size of the string after the
string keyword, like this

```
// 11 characters + null terminator...
dim MyString as string(12)
```

> It is extremely important that string variables are declared with enough character elements to support the runtime operation of your program. Failure to do so will certainly result in problems when your code is executing. For example, concatenating (joining) two strings that contain 20 characters each will require a destination string that has reserved 41 elements (2 * 20, + 1 line terminator).

Swordfish enables you to specify string sizes of up to 256 bytes, which equates to 255 individual character elements. Unlike strings, a char type can only hold one single character. A char variable can be declared and initialized in the following way,

```
dim MyChar as char
MyChar = "A"
```

The compiler supports the "+" operator to concatenate (join) two strings.
For example,

```
dim StrA, StrB, StrResult as string
StrA = "Hello"
StrB = "World"
StrResult = StrA + " " + StrB
```

Will result in *StrResult* being set to "Hello World". The two relational operators = (equal) and <> (not equal) are also supported for string comparisons. For example,

```
if StrA = StrB then
   USART.Write("Strings are equal!")
```

```
endif
if StrA <> StrB then
   USART.Write("Strings are NOT equal!")
endif
```

You can also mix the concatenation operator with the supported relational operators, as shown in the following example,

```
include "USART.bas"
dim StrA, StrB as string
SetBaudrate(br19200)
StrA = "Hello"
StrB = "World"
if StrA + " " + StrB = "Hello World" then
   USART.Write("Strings are equal!", 13, 10)
endif
```

The compiler can also read or write to a single string element by indexing it in the following way,

```
StrResult(5) = "_"
```

This would result in "Hello World" being changed to "Hello_World". Note that the first character of a string variable is located at 0, the second character at 1 and so on.

> An alternative way to assign a single character to a string element or char variable is by using the # notation. For example, the underscore character ("_") can be represented by the ASCII number 95 decimal. We could therefore write StrResult = #95. This technique is particularly useful when dealing with non white space characters, such as carriage returns and line feeds.

A useful compiler constant is **null**, which can be used to set, or tested for, a string null terminator.

In the example overleaf, the length of a string is computed and output via the microcontroller's hardware USART.

```
include "USART.bas"
include "Convert.bas"

dim Str as string
dim Index as byte
SetBaudrate(br19200)
Str = "Hello World"
Index = 0
while Str(Index) <> null
   inc(Index)
wend
USART.Write("Length is ", DecToStr(Index), 13, 10)
```

It should be noted that the compiler constant **null** is logically equivalent to "" (an empty string)