Swordfish Compiler

Structured BASIC for PIC® microcontrollers

Language Reference Guide

Provisional

Table of Contents

Introduction

Swordfish is a structured Basic language for 18 series PIC® microcontrollers from Microchip™ Corp. This document is a language reference only and describes the keywords, syntax and functions provided in the language. It is assumed that the reader is familiar with structured languages and the techniques employed in developing programs using a structured language.

The language can be further extended through the addition of libraries. A set of standard libraries is provided with the language and described in a separate manual.

Nomenclature

The following nomenclature is used throughout this manual:

[ ]	Optional item
|	Item choice
{}	Zero or more items

Keywords will be highlighted in **bold** where they are used in statements. Where example code is shown, this will reflect the default format of the Swordfish IDE. For example,

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

Contact Details

Mecanique
85 Marine Parade
Saltburn by the Sea
TS12 1BZ
United Kingdom

www.sfcompiler.co.uk
enquiries@sfcompiler.co.uk

Copyright

Identifiers and Reserved Words

The most common use of an identifier is to name constants, variables, aliases, structures, subroutines and functions. You should try and use descriptive names for identifiers, as this will make your program easier to follow and debug. A valid identifier is composed of the following elements:

[underscore] letter {letter | digit | underscore}

In other words: an optional underscore character followed by a single letter of the alphabet, followed by one or more letters, digits or underscores.

A reserved word has a special meaning for the compiler and cannot be used as an identifier. A list of reserved words is shown in Appendix 2.

Comments

Comments are not compiled and are used to document your program. To add a single line comment, use a quote character or double forward slash. For example,

```
// this is a comment...
’ this is a comment...
Value = 10 // this is also a comment
ValueB = 20 ’ and so is this.
```

Swordfish also supports block comments. You can use either left and right braces or you can use left parentheses plus asterisk followed by asterisk plus right parentheses.   For example,

```
(*
   this is a block
   comment
*)
ValueA = 10
{
   this is also a
   block comment
}
ValueB = 20
```

You cannot mix the two different style of block comment. For example,

```
(*
   this is NOT a
   valid block comment
}
```

However, you can nest the two different styles. For example,

```
(*
   this is a valid block
   { comment}
*)
```

It's usually good programming practice to stick with one type of block comment within your code. This way, you can make lots of notes using one block comment style and use the other to comment out large chunks of code when debugging.

Constants

[**private** | **public**] **const** identifier [**as** type] = expression

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

A constant declaration can be used in a program or module in place of a literal value. Constant data cannot be changed at runtime (that is, you cannot assign a new value to a constant when your program is executing). However, constants have the advantage of making your code much more readable and manageable.

You can declare constants one at a time, like this

```
const MaxSamples = 20
const SizeOfArray = 20
```

or as a list,

```
const
   MaxSamples = 20,
   SizeOfArray = 20
```

You can also use an expression on the right hand side of a constant declaration. For example,

```
const
   Hello = "Hello",
   HelloWorld = Hello + " World",
   ValueA = 12 * 0.4,
   ValueB = ValueA + 10
```

Swordfish will automatically assign the type of a constant, based on the expression itself. For example,

```
const MyStr = "Hello World" // a string constant
const ValueA = -100         // a signed constant
const ValueB = 100          // an unsigned constant
const ValueC = 0.4          // a floating point constant
```

Constants, unlike program variables, do not use RAM to store their values. If a constant is used in a program expression, code memory is used instead. When declaring numeric constants, or when using numeric literals in your program, you can use different number representations.

Representation	**Prefix**	**Example**	**Value**
Decimal	*none*	100	100 decimal
Binary	%	%100	4 decimal
Hexadecimal	$	$100	256 decimal

Swordfish will compute a constant declaration such as **const** Value = 5 / 1024 using integer arithmetic, resulting in *Value* being equal to zero. This is because both 5 and 1024 are ordinal values. If you want to force the type of *Value* to floating point, one or more of the literals in the expression should be made floating point, for example, **const** Value = 5.0 / 1024 or **const** Value = 5 / 1024.0

Array Constants

[**private** | **public**] **const** identifier(size) **as** type = (value {, value})

- *Private* – An optional keyword which ensures that a constant array is only available from within the module it is declared. Constant arrays are private by default.
- *Public* – An optional keyword which ensures that a constant array is available to other programs or modules.
- *Identifier* – A mandatory constant name, which follows the standard identifier naming conventions
- *Size* – A mandatory constant expression which defines the number of elements in the constant array
- *Type* – A mandatory data type. Supported types include boolean, bit, byte, word, longword, shortint, integer, longint, float, string and char.
- *Value* – One or more data values.

Array constants are extremely useful for accessing sets of data from within your program at runtime. Like single constant declarations, array data values cannot be changed at runtime.

Constant arrays can be declared one at a time, or in a list. You can even mix single constant declarations with array constants. For example,

```
const
   ArraySize = 5,
   ConstArray(ArraySize) as byte = (1, 2, 10, 20, 100),
   CR = 13,
   LF = 10
```

Constant arrays can be accessed in the same way as you would any other variable array. For example,

```
// import modules...
include "USART.bas"
include "Convert.bas"

const ConstArray(2) as byte = (100, 200)
dim Index as byte
```

```
// display value to terminal program, include
// CR and LF
USART.SetBaudrate(br19200)
for Index = 0 to bound(ConstArray)
   USART.Write(DecToStr(ConstArray(Index)), 13, 10)
next
```

You can also directly assign a constant array to a variable array, if the number of array elements is the same. For example,

```
const ConstArray(2) as byte = (100, 200) // two elements
dim VarArray(2) as byte                  // two elements
VarArray = ConstArray // assign values to variable array
```

You can even assign constant string arrays to variable string arrays. The only caveat is that the size of each string size, as well as the number of array elements, must be the same. This is easy to do by packing out the constant array strings with spaces. For example,

```
// largest string (Au revoir) is 9 characters, so
// pack each string to match…
const MenuEnglish(2) as string = ("Hello    ", "Goodbye  ")
const MenuFrench(2) as string = ("Bonjour  ", "Au revoir")
dim Menu(2) as string(10) // 9 + null terminator

// program start…
Menu = MenuEnglish // menu is now set for english
Menu = MenuFrench  // menu is now set for french
```

You only need to pack constant strings when performing direct assignment, like in the example above. If you just wish to access each array element individually, then no packing is required.

Constant arrays can only have a single dimension. Swordfish does not currently support multi-dimensional constant arrays. Constant arrays use program memory and not data RAM to store their values. This is unlikely to cause problems under normal circumstances, given the amount of code space available with current PIC® microcontrollers. However, code space is not unlimited and care should be exercised if using exceptionally large constant array declarations.

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.

Refer to Appendix 3 for a list of Variable types supported by the compiler and their storage requirements.

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* shown in Appendix 3 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).

Arrays

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

- *Private* – An optional keyword which ensures that an array is only available from within the module it is declared. Arrays are private by default.
- *Public* – An optional keyword which ensures that an array is available to other programs or modules.
- *Identifier* – A mandatory variable name, which follows the standard identifier naming conventions
- *Size* – A mandatory size which describes the number of elements in the array. Arrays can only have a single dimension. Swordfish does not currently support multi-dimensional arrays.
- *Type* – A mandatory data type. Supported types include boolean, bit, byte, word, longword, shortint, integer, longint, float, string, char and structures

Arrays are collections of values, with each element being of the same type. When you have many variables in your program, it is sometimes useful to use a more manageable array, rather than keep track of them manually.

You can declare arrays one at a time, like this

```
dim Array1(5) as byte
dim Array2(10) as byte
```

or as a list,

```
dim
   Array1(5) as byte,
   Array2(10) as byte
```

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

```
dim Array1(5), Array2(10) as byte
```

For bit or boolean arrays, accessing a single element using a variable index is very computationally expensive. For example BitArray(index) = 1. If at all possible, use a byte array instead.

To access a single array element in your program, use the array name followed by round brackets around the index of element you want. For example, if you want to access the second element of *Array1*, then use

```
Array1(1) = 10
```

Note that the first element of an array is located at 0, the second element at 1 and so on. Because Swordfish arrays begin with zero indexes, care must be taken when iterating through an array. For example,

```
for Index = 0 to 4
   Array1(Index) = 200
next
```

is correct, and will set all 5 array elements (0 to 4) of *Array1* to the value 200. A very useful compiler keyword is **bound**, which will automatically insert the correct upper bounds of your array when the program is compiled. The **bound** keyword is particularly useful when arrays are passed to functions or procedures, allowing you to write a routine that accepts arrays with different upper bounds. Using **bound**, we could rewrite the previous example like this,

```
for Index = 0 to bound(Array1)
   Array1(Index) = 200
next
```

Using **bound** prevents your program inadvertently indexing an array beyond its highest element, which would certainly lead to problems when you program is executing. For example,

```
dim Array(5) as byte
dim Index as byte
```

```
for Index = 0 to 5
   Array(Index) = 2
next
```

After compilation the variable *Index* would be stored in data RAM, directly after the *Array* declaration. When *Index* = 5, the memory location used by *Index* is overwritten with a 2 because we have set the incorrect upper limit in the **for…next** loop. Instead of terminating at 5, the **for…next** loop will never finish because the loop counter *Index* has been changed from 5 to a value of 2.

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
```

User Types

[**private**|**public**] **Type** identifier **= Type**

- *Private* – An optional keyword which ensures that an alias is only available from within the module it is declared. Variables are private by default.
- *Public* – An optional keyword which ensures that an alias is available to other programs or modules.
- *Identifier* – A mandatory and previously declared 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 and char.

You can create your own specific user type based on an existing data type or structure. This has two main purposes:

- To ensure a rigorous application of type checking. See type casting later in this reference.
- To enable overloaded operations to identify which overloaded routine to call when the data presented is of a similar data type. See overloading later in this reference)

Here is an example of a user type:

```
type MyType = byte
dim MyVar as MyType
```

Alias and Modifiers

[**private** | **public**] **dim** alias {, alias} **as** identifier{.modifier}

- *Private* – An optional keyword which ensures that an alias is only available from within the module it is declared. Variables are private by default.
- *Public* – An optional keyword which ensures that an alias is available to other programs or modules.
- *Alias* – A mandatory alias name, which follows the standard identifier naming conventions
- *Identifier* – A mandatory and previously declared variable name, which follows the standard identifier naming conventions
- *Modifier* – One or more optional modifiers which can be used to access different parts of the variable identifier

Unlike a variable, an alias is declared without a type and does not allocate any data RAM. As the name suggests, it shares its data RAM with a previously declared variable.

A simple alias declaration is shown below,

```
dim Value as byte
dim MyAlias as Value
MyAlias = 100
```

In this example, *Value* has been declared as a byte type. *MyAlias* has been declared as an alias to *Value*. When 100 is assigned to *MyAlias*, the number is stored at the RAM location reserved for *Value*. In other words, *Value* becomes equal to 100. Whilst a simple alias like this may be useful, substantial power and flexibility can be achieved when using an alias declaration with a modifier. For example, the program

```
dim LED as PORTB.7
while true
   high(LED)
   delayms(500)
   low(LED)
   delayms(500)
wend
```

will flash an LED connected to PORTB, pin 7. A list of additional modifiers is shown in Appendix 3.

Some modifiers use an array notation. For example, PORTB.7 can be written as PORTB.Bits(7). The Booleans() array modifier is very useful for changing a bit field into a boolean. For example, many PIC® microcontrollers have hardware USARTs to support serial communication. You can test to see if data is available in the receive buffer by testing PIR1.5. Using the booleans modifier, we can create a more descriptive alias which has the virtue of being easy to test when using **if…then**, **while…wend** and **repeat…until** statements,

```
dim DataIsAvailable as PIR1.Booleans(5) // RCIF flag

if DataIsAvailable then
   // process data here...
endif
```

Modifiers can also be used directly from within your program code. You don't need to explicitly declare an alias using the **dim** keyword. For example,

```
dim Value as word
Value.Byte1 = $FF
```

will set the high byte of the variable *Value* to 255 decimal. Alias and modifiers can also be used on arrays. This is particularly useful if you want to access a single element in a complex data structure. In the following example, *Value* is aliased to the first array elements *Minutes* field.

```
structure TTime
   Hours as byte
   Minutes as byte
end structure
dim Array(10) as TTime
dim Value as Array(0).Minutes

// these assignments are logically identical...
Value = 10
Array(0).Minutes = 10
```

Aliases and modifiers can also be used on previously declared aliases. For example,

```
dim Array(10) as word                // a word array
dim ElementAlias as Array(3)         // alias to element 4
dim ByteAlias as ElementAlias.Byte1 // element 4, high byte
dim BitAlias as ByteAlias.7          // element 4, high byte, bit 7
```

The modifiers discussed up until now have been used to access smaller parts of a larger variable. The Swordfish compiler also supports modifier promotion. That is, aliasing a smaller variable to a larger whole. For example, given the declarations

```
dim Array(8) as byte
dim Lower as Array(0).AsLongWord
dim Upper as Array(4).AsLongWord
```

we can now access the lower and upper 4 bytes of an array using single assignments, like this

```
Lower = $F0F0F0F0
Upper = $F0F0F0F0
```

Each element of *Array* will now be set to $F0. Modifier promotion can be very useful when interfacing to some of the PIC® microcontrollers hardware registers. For example, the File Select Register (FSR) on an 18 series PIC® is made up of two 8 bit registers, such as FSR0L and FSR0H. Using modifier promotion, we can assign a single 16 bit value, like this

```
dim FSR0 as FSR0L.AsWord
FSR0 = $0ABC
```

This works because FSR0H (the high byte) is located in the next RAM location after FSR0L (the low byte). A list of additional promotion modifiers is shown in Appendix 3.

Modifier promotion can also be used directly from within your program code for user declared variables. However, please note that the Swordfish compiler does not currently support modifier promotion of PIC® microcontroller register names from within a code block.

EEPROM Data

eeprom [(address)] = (item [**as** type] {, item [**as** type]})

- *Address* – An optional starting address for the EEPROM data. If a starting address is omitted, data is stored starting from address 0.
- *Item* – One or more data items.
- *Type* – An optional item type. Supported types include byte, word, longword, shortint, integer, longint, float, string and char.

The **eeprom** declaration enables you to store data on a microcontroller that supports on-chip EEPROM.

The simplest form of **eeprom** declaration does not have a starting address or type modifier. For example,

```
eeprom = (10, 20, 30)
eeprom = (40, 50, 60)
```

will store the values 10, 20, 30, 40, 50, 60 at consecutive EEPROM byte locations, starting at address 0. If you want to change the format of the data stored, you can specify a type. For example,

```
eeprom = (10 as word, 20 as longword)
```

will store the values 10, 00, 20, 00, 00, 00 at consecutive EEPROM byte locations, starting at address 0. You can also use string types in your data item list. For example,

```
eeprom = ("One", "Two")
```

will store the values "O", "n", "e", 0, "T", "w", "o", 0 at consecutive EEPROM byte locations, starting at address 0. Note that strings always end with a null terminator. Data item types can also be mixed in a single declaration. For example,

```
eeprom = ("One", 10, 3.142)
```

If you want your data items to start at particular EEPROM location, you can give an explicit starting address, for example

```
eeprom(100) = (10 as word, 20 as byte)
```

will store the values 10, 00, 20 at consecutive EEPROM byte locations, starting at address 100.

Unlike standard constant declarations, there is no direct language support for reading or writing to EEPROM using the Swordfish compiler. If you wish to read and write to the microcontroller's EEPROM when your program is executing, there are a number of routines provided in the compiler EEPROM library. For example,

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

eeprom = ("Value = ", 10)
dim NextAddress,Value as byte
dim Str as string

USART.SetBaudrate(br19200)
EE.Read(0,Str, Value)
USART.Write(Str, DecToStr(Value), 13, 10)
```

When creating large blocks of EEPROM data, it can sometimes become difficult to manage address locations, particularly if you use random access to EEPROM rather than reading data sequentially. For example,

```
eeprom(100) = ("MyString", 10, 20, 30)
```

In this example, it is difficult to identify the starting address of the byte data after the string. You could of course use a constant to identify the start of the byte data, like this,

```
const
   StringEE = 100,
   DataEE = 109
dim
   ValueA as byte

eeprom(StringEE) = ("MyString")
eeprom(DataEE) = (10, 20, 30)
EE.Read(DataEE, ValueA)
```

In this example, the first value of byte data can be accessed from your code by using the constant address value *DataEE*. Although this is just about manageable for small sets of EEPROM data, it becomes virtually unworkable for larger data sets because you need to manually calculate where each starting address will be. Worse still, if you change the size of a data element (for example, changing "MyString" to "My New String") then you will need to recalculate all starting addresses for items following the change. Fortunately, you can let swordfish do the work for you. For example,

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

eeprom(@StringEE) = ("String Data...")
eeprom(@DataEE) = (10, 20, 30)

dim
   ValueA, ValueB, ValueC as byte,
   ValueStr as string

SetBaudrate(br19200)
EE.Read(DataEE, ValueA, ValueB, ValueC)
EE.Read(StringEE, ValueStr)
USART.Write(ValueStr, " : ",
            DecToStr(ValueA), " ",
            DecToStr(ValueB), " ",
            DecToStr(ValueC), 13, 10)
```

Using the @ symbol before an EEPROM address identifier will tell the compiler to automatically create a constant declaration and initialize its value to the next free EEPROM address location. This means that if any items are modified or new items are inserted into you EEPROM table, the new starting address is computed automatically.

Conditional Statements

Conditional statements are used in a program to alter the operational flow. That is, to decide which statement or statements to execute, based on the evaluation of a single value or expression. Swordfish supports three types of conditional statements: **if…then**, **select…case** and **conditional jump**.

The If…Then Statement

```
if expression then
 statements
[elseif expression then]
  {statements}
[else]
 {statements}
endif
```

The **if…then** statement is used to make a program execute user code, but only when certain conditions are met. If an expression evaluates to true, then any statements following the evaluation are executed. If no expression evaluates to true, then any statements contained after an optional **else** are executed. For example,

```
if Value <= 10 then
   Message = "OK"
elseif Value > 10 and Value < 20 then
   Message = "Warning"
else
   Message = "Error"
endif
```

The Select…Case Statement

```
select expression
  case condition {, condition}
    {statements}
  …
  [else {statements}]
endselect
```

Although there is nothing technically wrong with using large **if…then** blocks, it can sometimes be difficult to read them. The **select…case** statement is an alternative to using a large or multiple **if…then…elseif** statement. For example,

```
select MenuChoice
   case "Q", "q" Message = "Quit"
   case "M", "m" Message = "Main Menu"
   case "A", "a" Message = "Option A"
   case "B", "b" Message = "Option B"
   case "C", "c" Message = "Option C"
else
   Message = "Error"
endselect
```

In this example, the select part is a single char type which is tested against each successive case condition. The commas used in the case conditions are equivalent to using an if statement's logical **or** operator. For example, you could write the first case as an **if…then** statement in the following way,

```
if MenuChoice = "Q" or MenuChoice = "q" then …
```

If one of the case conditions is met, then any statements following the condition are executed and the program jumps to any code immediately following **endselect**. If no case conditions are met, statements contained in the optional else clause are executed.

Case conditions can also include relational operators, or the **to** operator can be used for a range of values. For example,

```
select Value * 2
   case < 10, > 100
      Result = 1
   case 10 to 20, 50 to 100
      Result = 2
else
   Result = 0
endselect
```

In this example, *Value* is multiplied by two and then tested against each case condition. If the select expression is < 10 or > 100, then *Result* becomes equal to 1. If the select expression is in the range 10 to 20 or 50 to 100, then *Result* becomes equal to 2. If none of the select conditions are met, Result is set to 0 in the **select…case** else block.

Conditional Jump

if expression **goto** label

The conditional jump is a special construct that can be used with a standard **goto** statement. For example,

```
if Value <> 0 goto SkipCode
   high(LED)
   delayms(500)

SkipCode:
   low(LED)
```

Notice the difference in syntax when compared to a normal **if…then** statement. Firstly, no **endif** is required. Secondly, the **then** part of the statement is not present. You can of course use a **goto** inside a normal **if…then** statement, but the above form allows you to write the same thing more concisely.

The **goto** statement has a nasty reputation because of its ability to jump to almost anywhere. Some people view this lack of control as very bad. Using a **goto** can produce what is called spaghetti code. It gets this name because with a **goto** infested program, drawing a line between a **goto** and its destination label would look like a big plate of spaghetti. Used with care, a **goto** statement can be useful. However, given the highly structured nature of the compiler language, a **goto** statement should be used sparingly and is best avoided.

Repetitive Statements

Repetitive statements enable one or more statements to repeat. The Swordfish compiler has three repetitive statements**: while…wend**, **repeat…until** and **for…next**.

The While…Wend Loop

while expression
  {statements}
wend

A **while…wend** loop will execute one or more statements if the expression evaluates to true. When an expression becomes false, the while loop terminates.

The condition can be any boolean expression or a single boolean variable. The following example shows how a **while…wend** loop can be used to count from 0 to 9, outputting each value in turn,

```
Index = 0
while Index < 10
```

```
   USART.Write(DecToStr(Index), 13, 10)
   inc(Index)
wend
```

The **while…wend** loop is can be useful for delaying program execution until a certain event has occurred. For example, if we write

```
dim PinIsHigh as PORTB.Booleans(0)
while PinIsHigh
wend
```

then the code following **wend** is not executed until PORTB.0 becomes equal to 0.

The Repeat…Until Loop

repeat
  {statements}
until expression

A **repeat…until** loop will execute one or more statements if the expression evaluates to false. When an expression becomes true, the repeat loop terminates. The condition can be any boolean expression or a single boolean variable. Unlike a **while…wend** loop, any statements in a **repeat…until** loop *will be executed at least once*, since the conditional test is evaluated at the bottom of the loop. For example,

```
Index = 0
repeat
   USART.Write(DecToStr(Index), 13, 10)
   inc(Index)
until Index > 9
```

Note the conditional termination logic of a **repeat…until** loop expression is the opposite of a **while…wend** loop. That is, a **while…wend** will *loop while some condition is true* but a **repeat…until** will *loop while some condition is false*.

A **repeat…until** loop will always execute the enclosed statements at least once whereas **while…wend** may never execute the enclosed statements if the expression initially evaluates to false.

The For…Next Loop

for variable = expression **to** expression [**step** expression]
  {statements}
next

The **for…next** loop will execute one or more statements a predetermined number of times. Unlike **while…wend** or **repeat…until** loops, the **for…next** loop does not use a boolean expression for termination control but a *control variable*. For example,

```
for Index = 0 to 9
   USART.Write(DecToStr(Index), 13, 10)
next
```

In this example, the control variable is *Index*. The control variable must be an ordinal type, such as byte or word. It cannot be a non-ordinal, such as floating point. The control variable is initialized to zero when the loop begins and terminates when the control variable is greater than nine. In other words, the **for…next** iterates through its loop ten times (0 to 9). If no step value is given, the control variable is incremented by one each iteration.

You can change the default increment value of the control variable by specifying a step value. For example,

```
for Index = 0 to 9 step 3
   USART.Write(DecToStr(Index), 13, 10)
next
```

will output 0, 3, 6 and 9. If the start expression is larger than the end expression (that is, you want to count downwards) then a negative step value must always be specified. For example,

```
for Index = 9 to 0 step -3
   USART.Write(DecToStr(Index), 13, 10)
next
```

will output 9, 6, 3 and 0.

Note - When using a **for…next** loop, it is standard practice *never* to modify the control variable in any way.

Short Circuit Boolean Expressions

Statements such as **if…then**, **if…goto**, **while…wend** and **repeat…until** depend on the evaluation of a boolean expression to determine program flow or to control loop iterations. The Swordfish compiler uses *short circuit* evaluation, which can make these statements execute more quickly. For example,

```
if a < b and c = d and e > f then
   // execute statements
endif
```

The **if…then** statement block will only be executed if *all* of the three conditions are true. That is, if a is less than b **and** c is equal to d **and** e is greater than f. However, if a is NOT less than b, then there is no point testing any of the other conditions, because the final result will *always be false*.

In short, the Swordfish compiler will immediately stop evaluating any boolean expression, if a certain outcome becomes known in advance. The expression is said to *short circuit,* causing the program to skip over the rest of the evaluation code.

Break

Calling **break** from within a **while…wend**, **repeat…until** or **for…next** loop will force your program to jump immediately to the end of the *currently* executing loop. For example,

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

dim Index as word
SetBaudrate(br19200)
Index = 0
while Index < 1000
   if IsDataAvailable then
       break
   endif
   USART.Write(DecToStr(Index), 13, 10)
   delayms(100)
   inc(index)
wend
```

If the *IsDataAvailable* flag is false, the **while…wend** loop will iterate normally. However, if the hardware USART in this example receives data, the *IsDataAvailable* flag becomes true and the loop terminates.

Using **break** too often can result in multiple exit points in a block of code, which can make your program difficult to debug and harder to read. When possible, it is better programming practice to allow your looping construct to control all exit conditions. For example, the previous code sample code be written as,

```
while Index < 1000 and not IsDataAvailable
   USART.Write(DecToStr(Index), 13, 10)
   delayms(100)
   inc(index)
wend
```

If you find yourself using **break** inside a **for…next** loop, it may be an indication that a **while…wend** or **repeat…until** statement is a more appropriate construct to use.

Continue

Calling continue from inside a **while…wend**, **repeat…until** or **for…next** loop will force your program to begin the next iteration of the *currently* executing loop. For example,

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

dim Index as byte
SetBaudrate(br19200)
for Index = 0 to 5
   if Index = 3 then
      continue
   endif
   USART.Write(DecToStr(Index)), 13, 10)
next
```

will output 0, 1, 2, 4 and 5. This is because when the control variable *Index* reaches 3, the program immediately begins another iteration of the **for…next** loop, skipping the call to *Write()*.

Subroutines and Functions

Subroutines and functions enable you to divide a program into smaller parts. A subroutine or function is a named group of statements, constants, variables and other declarations that perform a particular purpose. A function is identical to a subroutine in every respect, with the one exception: it can return a single value to the calling program. Swordfish subroutines and functions are *non-reentrant*, that is, you cannot make recursive subroutine or functions calls.

Subroutine Declarations

[**private** | **public**] [**inline**] **sub** identifier ([param {, param}])
  {declarations}
  {statements}
end sub

- *Private* – An optional keyword which ensures that a subroutine is only available from within the module it is declared. Subroutine declarations are private by default.
- *Public* – An optional keyword which ensures that a subroutine is available to other programs or modules.
- *Identifier* – A mandatory subroutine name, which follows the standard identifier naming conventions
- *Param* – One or more optional formal parameters

A formal parameter has the following parts

[**byval** | **byref** | **byrefconst**] identifier **as** type [= constexp]

- *ByVal* – An optional keyword indicating that an argument is passed by value. By default, formal parameters arguments are passed by value.
- *ByRef* – An optional keyword indicating that an argument is passed by reference.
- *ByRefConst* – An optional keyword indicating that a code constant is passed by reference
- *Type* – A mandatory data type. Supported types include boolean, bit, byte, word, longword, shortint, integer, longint, float, string, char and structures
- *ConstExp* – An optional constant expression

Function Declarations

[**private** | **public**] [**inline**] **function** identifier ([param {, param}]) **as** type
  {declarations}
  {statements}
end function

- *Private* – An optional keyword which ensures that a subroutine is only available from within the module it is declared. Subroutine declarations are private by default.

- *Public* – An optional keyword which ensures that a subroutine is available to other programs or modules.
- *Identifier* – A mandatory subroutine name, which follows the standard identifier naming conventions
- *Param* – One or more optional formal parameters
- *Type* – A mandatory data type. Supported types include boolean, bit, byte, word, longword, shortint, integer, longint, float, string, char and structures. Array types are not supported.

A formal parameter has the following parts

[**byval** | **byref** | **byrefconst**] identifier **as** type [= constexp]

- *ByVal* – An optional keyword indicating that an argument is passed by value. By default, formal parameters arguments are passed by value.
- *ByRef* – An optional keyword indicating that an argument is passed by reference.
- *ByRefConst* – An optional keyword indicating that a code constant is passed by reference
- *Type* – A mandatory data type. Supported types include boolean, bit, byte, word, longword, shortint, integer, longint, string, float, char and structures
- *ConstExp* – An optional constant expression

Parameters

Subroutines and function headings that do not have any formal parameters are written in the following way,

```
include "USART.bas"
sub Print()
   USART.Write("Hello World", 13, 10)
end sub

// main code block
SetBaudrate(br19200)
Print
```

The subroutine declaration *Print()* outputs "Hello World" each time it is called. Note that although no formal parameters have been declared, start and end round brackets are still required. A more useful example would enable any string to be output. To do this, a formal parameter is added,

```
include "USART.bas"
sub Print(pStr as string)
   USART.Write(pStr, 13, 10)
end sub

// main code block
SetBaudrate(br19200)
Print("Hello World")
```

The *Print()* subroutine declaration will now output any string value passed to it.

You do not have to explicitly give the size of a formal parameter string when passing a string argument to a subroutine or function. For example, pStr **as** string(20). This is because the Swordfish compiler has a powerful mechanism for calculating at compile time the maximum RAM needed for any string passed by value.

In the previous examples, the string parameter argument was passed to the subroutine using the compilers default mechanism of *by value* (**byval)**. Passing by value means that a local copy of the variable is created, and the subroutine or function operates on a copy. If your subroutine or function statement block changes the parameter value, it doesn't change the value of the actual variable being passed. This is in contrast to passing a variable *by reference* (**byref)**. Passing by reference means that a subroutine or function receiving the variable can modify the contents of the variable being passed. This is sometimes referred to as a *variable parameter*. For example,

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

sub NoChange(pValue as byte)
   pValue = 10
end sub
sub ChangeValue(byref pValue as byte)
   pValue = 10
end sub

dim Value as byte
SetBaudrate(br19200)
Value = 0
NoChange(Value)
USART.Write("Value : ", DecToStr(Value), 13, 10)
ChangeValue(Value)
USART.Write("Value : ", DecToStr(Value), 13, 10)
```

The first subroutine *NoChange()* has a formal parameter that accepts arguments passed by value. The second subroutine *ChangeValue()* has a formal parameter that accepts arguments passed by reference. When the following lines are executed,

```
NoChange(Value)
USART.Write("Value : ", DecToStr(Value), 13, 10)
```

The value output will be 0, because *NoChange()* has received a copy of the contents of *Value*. When the following lines are executed,

```
ChangeValue(Value)
USART.Write("Value : ", DecToStr(Value), 13, 10)
```

The value output will now be 10, because *ChangeValue()* has received the actual RAM address of *Value.* Some declaration types, such as arrays, must always be passed by reference. For example,

```
sub PassArray(byref pArray() as byte)
end sub
```

Notice that *pArray* is followed by open and closing round brackets. This is to inform the compiler that an array is being passed. Without the brackets, the compiler would just interpret the parameter argument as a single byte type.

Unlike arrays, structures can be passed by value. However, if your structure has a large number of variables (or uses arrays and strings) it would be more computationally efficient to pass by reference, rather than the compiler having to copy large amounts of data, as would be required if passed by value.

It is important to remember that when a parameter argument is passed by reference, you *can only call a subroutine or function with a single variable type*. For example, given the declaration

```
sub MySub(byref pValue as word)
end sub
```

then an error *'cannot be passed by reference'* message is generated when any of the following calls are made,

```
MySub(10)
MySub(Index * Index)
```

Remember, passing by reference forces the compiler to pass the RAM address of a variable, allowing it to be changed from within a subroutine or function. Constants or expressions do not have RAM addresses associated with them, and so cannot be used if a parameter argument is expecting pass by reference. If your subroutine or function parameter declaration is likely to be passed as a constant or expression, then you must *always pass by value*.

When a parameter is passed by value, it is sometimes useful to initialize the argument with a constant. For example,

```
sub Print(pStr as string, pTerminator as string = #13 + #10)
   USART.Write(pStr, pTerminator)
end sub
```

The formal parameter *pTerminator* has a default value of #13#10, which corresponds to a carriage return, line feed pair. If the subroutine *Print()* is called without a *pTerminator* argument value,

```
Print("Hello World")
```

then *pTerminator* will default to #13#10 when *USART.Write()* is called. If you wish to explicitly override the formal parameter default, then call your subroutine with the required value, like this

```
Print("Hello World", null)
```

Here, *pTerminator* is set to the null terminator when *USART.Write()* is called. It should be noted that you can only assign constants if the formal parameter argument is passed by value. In addition, you can only assign constants to parameters that appear at the end of the formal parameter list. For example,

```
sub MySub(pA as byte, pB as byte = 10, pC as byte = 20)
end sub
```

is correct, but

```
sub MySub(pA as byte = 10, pB as byte, pC as byte)
end sub
```

will generate a compilation error.

There is a third parameter passing mechanism, which is primarily used for constant arrays. On a PIC® microcontroller, constant arrays are stored differently from data RAM which requires the use of **byrefconst**. This ensures that a ROM address is passed and not a RAM address. For example,

```
include "USART.bas"
const Names(3) as string = ("David", "Fred", "Peter")
sub DisplayNames(byrefconst pNames() as string)
   dim Index as byte
   for Index = 0 to bound(pNames)
      USART.Write(pNames(Index), 13, 10)
   next
end sub

SetBaudrate(br19200)
DisplayNames(Names)
```

In this example, *DisplayNames()* will output all the string values contained in the constant array *Names*.

Subroutine and Function Scope

Scope is a common term used to describe the visibility of a particular declaration within a program. The scope of parameter arguments, constants, structures and variables that are declared within a subroutine or function are *local*. That is, they do not exist outside of the subroutine or function block. For example,

```
sub MySub(pValue as byte)
   dim LocalIndex as byte
end sub
LocalIndex = 10
pValue = 20
```

Will generate two *'identifier not declared'* error messages, because *pValue* and *LocalIndex* can only be seen from inside *MySub().*

It's useful to understand how the compiler finds a local declaration. For example, if your subroutine or function references a variable called *Index*, it will first look to see if a local variable or parameter called *Index* has been declared. If it's not found, then it will then look in the current module or program to see if the variable has been declared. If it has still not been found, it will then search all include files referenced in the current module to see if any public variable called *Index* have been declared. If it still has not been found, an error is generated.

This means that only the include files that are specifically defined in the current module are within the scope of the module. Hence the same module could be included a number of times with a program. The compiler, however, will only include that module once.

There can be instances where a potential ambiguity arises, for example there may be 2 libraries which contain a Read Function. Use redirection to resolve the ambiguity by prefixing the item with the Module name separated by a period. For example,

```
USART.Write("My Message")
```

will provide a unique reference to the USART library's Write subroutine. This can apply to any variable, structure subroutine or function. Alternatively create an alias to the required function

```
Dim SerOut As USART.Write
```

Frame Recycling

A *frame* describes the area of RAM reserved for use by local variables and parameters. Variables and parameters that are declared local to a subroutine or function are recycled by the compiler, whenever possible. For example,

```
sub MySubA()
   dim Array(1000) as byte
   dim Index as byte
   for Index = 0 to bound(Array)
      Array(Index) = 0
   next
end sub
sub MySubB()
   dim Array(1000) as byte
   dim Index as byte
   for Index = 0 to bound(Array)
      Array(Index) = 0
   next
end sub
```

The subroutine *MySubA()* allocates just over one thousand RAM bytes for its frame, to support the *Array* and *Index* declarations. *MySubB()* does exactly the same. However, when you call both of the subroutines from your program,

```
MySubA
MySubB
```

the compiler will just allocate RAM for one frame only (a little over one thousand bytes). This is because the subroutine calls *are not dependent* on each other, which means *MySubB()* can overlay its frame over the one allocated for *MySubA()*. Of course, if *MySubB()* made call to *MySubA()*, then twice as much frame RAM is needed. This is to ensure that the variable and working register state of *MySubB()* is preserved, preventing *MySubA()* from overwriting it.

Frame recycling is a very powerful mechanism for minimizing RAM usage on microcontrollers with limited resources. If at all possible, declare working variables inside the scope of a subroutine or function, rather than at the program or module level, to fully exploit frame recycling.

Inline

When an inline subroutine or function is generated, the computational expense of a call and return is removed by inserting the subroutine or function statement block at the point where the original call was made. By default, the compiler will make all subroutines and functions inline, if they are called *only once* from your program. For example, the following *Print()* subroutine

```
sub Print()
   USART.Write("Hello World", 13, 10)
end sub
SetBaudrate(br19200)
Print
USART.Write("The End", 13, 10)
```

would be converted to inline, which is the same as writing,

```
SetBaudrate(br19200)
USART.Write("Hello World", 13, 10)
USART.Write("The End", 13, 10)
```

You can force the compiler to always inline a subroutine or function by prefixing the declaration with the **inline** keyword. For example,

```
inline sub Print()
   USART.Write("Hello World", 13, 10)
end sub
```

To prevent the compiler from making a subroutine or function inline, simply prefix the declaration with the noinline keyword. For example,

```
noinline sub Print()
   USART.Write("Hello World", 13, 10)
end sub
```

Take care when explicitly making a subroutine or function inline. Although inline routines remove the time overhead associated with making a call, there can be a significant cost in terms of code space used. Generally, you should only use inline for very small routines that need to execute quickly.

Function Return Types

A function is identical to a subroutine in every respect, with the one exception: it can return a single value to the calling program. You can use functions in expressions anywhere you would normally use a constant or variable of the same type. For example,

```
function Multiply(pValue as byte) as word
   Multiply = pValue * pValue
end function
dim Value as word
Value = 100
Value = Multiply(Value) + Multiply(Value * 2) - 50
```

Functions can return boolean, bit, byte, word, longword, shortint, integer, longint, float, string, char and structures. To assign a value to a function, you can use its name. For example,

```
function Multiply(pValue as byte) as word
   Multiply = pValue * pValue
end function
```

Alternatively, you can use an implicitly declared variable called *result*,

```
function Multiply(pValue as byte) as word
   result = pValue * pValue
end function
```

You can override the implicit result variable by declaring a variable of the same name,

```
function Multiply(pValue as byte) as word
   dim result as word
   result = pValue * pValue
   Multiply = result
end function
```

The function return type can be used on the left and right hand side of an expression. For example,

```
function Multiply(pValue as byte) as word
   result = pValue
   result = result * pValue
end function
```

You can also use modifiers with the function return type, like you would any other variable. For example,

```
function SetUpper(pValue as byte) as word
   result.Byte0 = 0
   result.Byte1 = pValue
end function
```

String return types are a special case. They can be declared in a number of ways. The first uses the same method as a formal parameter declaration. That is, no explicit size is given.

```
function StrCopy(pStr as string) as string
   result = pStr
end function
```

Here, the compiler will ensure *result* is allocated enough RAM to hold the return value, which depends on the value of *pStr*. It may be the case that you don't explicitly assign a string value to the function result. For example, when using assembler you will be manipulating the result string directly. The compiler therefore cannot calculate how much RAM to allocate, so you need to do one of two things. You can give the return string an explicit size

```
function MyFunc(pStr as string) as string * 32
end function
```

will allocate 32 bytes, including the null terminator, for the result. Your return string must therefore *never exceed* 31 characters. Alternatively use the **auto** keyword, to automatically track the size of a formal parameter,

```
function MyFunc(pStr as string) as string auto(pStr)
end function
```

as *pStr* grows in size during compilation, so will the size of the function result.

Forgetting to assign a value to a function return type is a very common error. You should always ensure that a function result is assigned a value before the function exits. If you don't, the function value will be left in an undetermined state leading to very erratic program behavior which may be difficult to debug.

Exit

Calling **exit** will immediately terminate a currently executing subroutine or function and return to the next code statement following the subroutine or function call. For example,

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

sub DisplayValue(pValue as byte)
   if pValue = 5 then
      exit
   endif
   USART.Write("Value = ", DecToStr(pValue), 13, 10)
end sub

dim Index as byte
SetBaudrate(br19200)

for Index = 0 to 10
   DisplayValue(Index)
next
```

In this example, the main program **for...next** loop will make repeated calls to *DisplayValue()* with an *Index* that ranges from 0 to 10. The **if...then** statement inside DisplayValue() will call **exit** if the value passed is equal to 5, giving an output of 0, 1, 2, 3, 4, 6, 7, 8, 9, and 10.

Using **exit** too often can result in multiple termination points in a subroutine or function, which can make your program difficult to debug and harder to read. When possible, it is better programming practice to allow conditional and looping constructs to control exit conditions. For example, the previous subroutine *DisplayValue()* could be written as,

```
sub DisplayValue(pValue as byte)
   if pValue <> 5 then
      USART.Write("Value = ", DecToStr(pValue), 13, 10)
   endif
end sub
```

If you must use **exit** from within a function, *it is essential that a return value is assigned before terminating*.

Subroutine and Function Aliasing

Subroutines and functions can be aliased, in much the same way as you would alias a variable. For example,

```
// standard libraries...
include "USART.bas"
include "LCD.bas"

// rename standard library functions...
dim HSerOut as USART.Write
dim LCDWrite as LCD.Write

// use the alias names
HSerOut("Hello World", 13, 10)
LCDWrite("Hello World")
```

In this example, the standard library routines for writing have been renamed to match the naming conventions used by some other PIC® microcontroller BASIC compilers.

Overloading

Overloading enables you to have multiple subroutines and functions in the same scope that share the same name. The compiler will select the most appropriate routine to call, based on its signature. A subroutine or function signature is constructed by using the number of formal parameters and also the type of each parameter. An overloaded routine must therefore have a unique combination of parameters, so that the compiler can identify which
routine to call during compilation. For example,

```
function Multiply(pValueA, pValueB as byte) as word
   Result = pValueA * pValueB
end function

function Multiply(pValueA, pValueB as byte) as float
   Result = pValueA * pValueB
end function
```

will generate an error because the overloaded function signatures are identical. That is, they both have two parameters each of type byte. It is important to note that the compiler does *not* use function return types as part of the signature, only parameters The previous problem can be corrected by overloading the function with a unique parameter signature, like this,

```
function Multiply(pValueA, pValueB as byte) as word
   Result = pValueA * pValueB
end function

function Multiply(pValueA, pValueB as float) as float
   Result = pValueA * pValueB
end function

dim Result as word
Result = Multiply(10,20)
```

In this example, the first overloaded function is called because the parameter arguments are of type byte. The compiler will try and invoke the routine whose parameters have the smallest range that will accommodate the arguments in the call. For example, if the call to *Multiply()* is made with the following arguments,

```
Result = Multiply(-10,20)
```

then the second function will be called, because the floating point parameter is the only one that can accommodate a value of -10.

If the type of the value to be returned is to be the only unique way of identifying an overloaded routine a **Sub** should be used and the result of the routine passed **byRef** in the parameters. For example:

```
sub MySub(byref pValue As Byte)
end sub

Sub MySub(byref pValue As Word)
end sub
```

If any parameters are assigned a constant in an overloaded routine, care should be taken to ensure you don't inadvertently create a situation where a routine cannot be called, for example,

```
sub MySub(pValueA as byte, pValueB as word = 0)
end sub

sub MySub(pValueA as byte)
end sub

MySub(10)
```

In this example, the compiler cannot determine if the first or second overloaded routine should be called, because the parameter arguments are ambiguous.

Compound Subroutines

[**private** | **public**] **compound sub** identifier ([subroutine {, subroutine}])

- *Private* – An optional keyword which ensures that a compound subroutine is only available from within the module it is declared. Subroutine declarations are private by default.
- *Public* – An optional keyword which ensures that a compound subroutine is available to other programs or modules.
- *Identifier* – A mandatory compound subroutine name, which follows the standard identifier naming conventions
- *Subroutine* – One or more previously declared subroutine identifiers. Please note that a compound subroutine can only call other subroutines, not functions.

A compound subroutine allows you to assign a single identifier that can be used to make multiple calls to a named subroutine, in one single statement. For example, rather than writing

```
WriteByte(10)
WriteByte(100)
WriteByte(5)
```

you can declare a compound subroutine,

```
compound sub Write(WriteByte)
```

and then call it from your main program,

```
Write(10,100,5)
```

Each time the compiler encounters the compound subroutine *Write()*, it takes each parameter argument in turn and generates a call to *WriteByte()*.
You can have more than one subroutine contained in the compound declaration parameter list. For example,

```
compound sub Write(SetAddress, WriteByte)
```

When declaring a compound subroutine with more than one subroutine parameter, only the last subroutine in the parameter list will be called multiple times. For example,

```
Write(100,100,20)
```

Would be the same as writing

```
SetAddress(100)
WriteByte(100)
WriteByte(20)
```

Because a compound subroutine will pass each argument in turn, it is essential that the subroutine to be called has been declared with exactly one parameter. Failure to do so will generate a compiler error message.

Compound subroutines are extremely powerful when used in conjunction with overloaded subroutines. For example,

```
// overloaded sub to output a string...
sub WriteItem(pValue as string)
end sub

// overloaded sub to output a byte...
sub WriteItem(pValue as byte)
end sub

// create compound subroutine...
compound sub Write(WriteItem)

// make the call...
Write(10,"Hello World", 13, 10)
```

In this example, the compound subroutine *Write()* is declared with an overloaded subroutine parameter called *WriteItem()*. When Write() is called from the main program, the compiler will make a call to the overloaded subroutine WriteItem(), based on the argument type. This allows you to create compound calls which accept arguments of different types and in any order.

Typecasting

The compiler uses relatively strong type checking, which means that an error is generated if you try and assign variables of different types to each other. This is good. It helps prevents program errors through the incorrect assignment of variable types. However, you can typecast a value to override this behavior. For example, you may want to assign a char to a byte or a bit to a boolean, like this

```
MyByte = byte(MyChar)
MyBoolean = boolean(STATUS.2)
```

Type assignment is a little more relaxed when assigning floating point to ordinal or vice versa, so typecasting is not needed. For example,

```
dim MyFloat as float
dim MyByte as byte
MyByte = MyFloat
MyFloat = MyByte
```

Typecasting can be particularly useful for controlling code generation when expressions are used. If an expression has more than two operands, temporary storage is automatically allocated by the compiler to store intermediate results. For example,

```
dim a, b, c as byte
dim result as word
result = a + b + c
```

Would translate to something like

```
TMP_WORD = a + b
result = TMP_WORD + c
```

Notice that the temporary variable is a word size, because adding two bytes may result in a number larger than 8 bits. To override this behavior, you can use a typecast,

```
result = byte(a + b) + c
```

The intermediate storage allocation for (a + b) will now only be a byte. Typecasting can also be used to reduce the size of a declared variable, for example

```
result = byte(WordVar) + b + c
```

This will generate a temporary storage value of 16 bits (rather than 32), because byte + byte will require a 16 bit result. Note the subtle difference between

```
// MyWord made a byte, added to b, tmp storage = 16 bits
result = byte(WordVar) + b + c

// result of (MyWord + b) is byte, tmp storage is 8 bits
result = byte(WordVar + b) + c
```

You do not normally have to worry about mixing types within Swordfish expressions, as the compiler will promote variables automatically to ensure the correct result is obtained. However, typecasting can be extremely useful for expressions that contain mixed types. For example,

```
dim SValue as integer
dim UValue as word
dim FValue as float
dim Result as word
Result = SValue + UValue + FValue
```

In this example, the compiler needs to ensure that the signed and unsigned ordinal addition (*SValue* + *UValue*) is computed correctly by internally converting and promoting the sub-expression before a signed addition is performed. Next, the intermediate result is converted to a floating point format, before it can be added to *FValue*. Finally, the floating point intermediate result is converted into an ordinal type and assigned to the variable *Result*. All these computations need to be done for the correct result to be calculated, given that the compiler knows nothing about the state of the mixed types used.

However, if you understand what your programming is doing, you can change the behavior of the compiler expression generator to produce better optimized code. For example, we could write the expression as,

```
Result = word(SValue) + UValue + word(FValue)
```

This would reduce the amount of code needed to support the computation by approximately 70%, when compared against the previous example. Of course, the correct result will only be obtained if the assumptions about the state of SValue and FValue when the expression is computed are correct.

Please remember, the compiler expression generator will produce code to obtain the correct results, given any mix of variable types. It does not make assumptions about your code. Changing this behavior may lead to an incorrect result being obtained. Worse, the incorrect result may be intermittent, which makes very difficult debugging sessions. Typecasting is very useful, but it must be used with extreme care.

Using Embedded Assembler

You can embed PIC® assembler code in subroutines and functions, as well as your main program code, by using an **asm... end asm** block. You can also declare local variables inside your subroutine or function and use them in you assembler code. For example,

```
function AddTen(pValue as byte) as byte
   dim LocalVar as byte
   LocalVar = 10
asm
   movf  LocalVar, W ; 10 into W
   addwf pValue, W   ; add parameter pValue to W
   movwf Result      ; move W into function result
end asm
end function
```

Please note that the compiler does not manage RAM banking in **asm...end asm** blocks. For further information on writing PIC® assembler and issues relating to bank switching, refer to the Microchip document *MPASM User's Guide*.

With Statement

with item {, item}
  {statements}
end with

A **with** statement is a shorthand for referencing the fields of a record or module. For example,

```
structure TDate
   Day as byte
   Month as byte
   Year as word
end structure
dim Date as TDate

with Date
   Day = 23
   Month = 3
   Year = 2007
end with
```

This is the same as writing

```
Date.Day = 23
Date.Month = 3
Date.Year = 2007
```

A **with** statement can have multiple items. For example,

```
with USART, Date
   Day = 23
   Month = 3
   Year = 2007
   Write(Day, Month, Year)
end with
```

Each reference name in a with statement is interpreted as a member of the specified structure or module. If there is another variable or procedure call of the same name that you want to access from the **with** statement, you need to prepend it with a qualifier. For example,

```
with USART, Date
   Day = 23
   Month = 3
   Year = 2007
   Write(Day, Month, Year)         // write to usart
   EE.Write(0, Day, Month, Year)   // write to eeprom
end with
```

Interrupts

interrupt Identifier([priority])
  {statements}
end interrupt

Priority is an optional numeric value that allows you to assign different priorities to multiple Interrupt Service Routines (ISR). The PIC® 18 series of microcontroller supports two priority levels, high and low. If your program has only one ISR, you should not assign any priority level. The compiler will assign the correct priority for you. An ISR declaration for a program with one interrupt would therefore look like this:

```
interrupt OnTimer()
   // code statements here…
end interrupt
```

A program that has two priority levels would look something like this,

```
const
   ipLow = 1,
   ipHigh = 2

interrupt OnTimer1(ipLow)
   // code statements here…
end interrupt

interrupt OnTimer3(ipHigh)
   // code statements here…
end interrupt
```

Interrupts perform basic context saving. If a single ISR is declared in your program, the STATUS, WREG and BSR are saved and restored in hardware shadow registers. If both high and low ISRs are declared within the same program, the high priority interrupt will save and restore STATUS, WREG and BSR in hardware shadow registers, but the low priority interrupt will save and restore STATUS, WREG and BSR in software.

The PIC® 18 series has only one level of hardware shadow registers for context saving. A lower priority interrupt has therefore to save and restore STATUS, WREG and BSR in software to preserve their values.

You can force the compiler to perform software context saving in software for both high and low interrupts, rather than using hardware shadow registers. This is achieved by using the ISR_SHADOW option. For example,

```
#option ISR_SHADOW = false
```

disables all hardware shadow register context saving.

The basic context saving of an interrupt means it is unsuitable for supporting high level language constructs. For example, what appears to be a simple statement may involve using many different system and compiler registers. If these are changed in your ISR, your main program will almost certainly fail. You should also never call another subroutine or function from an interrupt unless additional steps have been taken with respect to context saving.

You can modify an interrupt to handle more complex context saving. For example, you might want to buffer data to an array inside your interrupt using an indirect register that points to your data buffer. In order to do this, you must save and restore the register you intend to alter in your ISR:

```
interrupt OnDataRX()
   dim FSRSave as word        // temp register
   FSRSave = FSR0             // save FSR
   FSR0 = AddressOf(MyBuffer) // we can now change FSR
   asm
      ; buffer data here…
   end asm
FSR0 = FSRSave                // restore FSR
end interrupt
```

Another technique is to use a **save**…**restore** block, which is discussed later in this document.

A Swordfish interrupt has excellent performance characteristics *but assumes a user knows what they are doing*. Great care should be taken in implementing interrupts and studying the relevant PIC® datasheet is essential.

Enabling and Disabling Interrupts

Although the PIC® 18 series only allows a maximum of two interrupts to be used (one high and one low priority), the compiler allows you to declare more than one of each. You only commit the assignment of your ISR to an individual interrupt vector when you issue an enable command. For example:

```
enable(OnTimer)
```

The enable keyword explicitly assigns an ISR to a microcontroller high or low interrupt vector. In addition, the microcontroller GIEH or GIEL flags are set to true. Once enable is called, your program is committed to using this interrupt. Calling enable twice on two different ISRs of the same priority will generate a compiler error message. To set the interrupt enable flags to false, call:

```
disable(OnTimer)
```

Note that calling disable does not remove your ISR from the microcontroller interrupt vector.

```
// program constants...
const
   TimerReloadValue = 50,
   TimerValue = 65536 - _clock * 2500

// timer 1...
dim Timer1 as word(TMR1L)
dim Timer1IF as PIR1.0
dim Timer1IE as PIE1.0
dim Timer1On as T1CON.0

// additional program variables...
dim LED as PORTD.0
dim TimerCounter as byte

// timer interrupt - blink LED every 500ms or so...
interrupt OnTimer()
   Timer1 = Timer1 + word(TimerValue) // force integer arithmetic
   dec(TimerCounter)
   if TimerCounter = 0 then
      TimerCounter = TimerReloadValue
      toggle(LED)
   endif
   Timer1IF = 0
end interrupt

// configure and activate timer 1...
sub ActivateTimer()
   Timer1 = TimerValue
   T1CON = 0
   Timer1IF = 0
   Timer1IE = 1
   Timer1On = 1
   enable(OnTimer)
end sub

// program start...
low(LED)
ActivateTimer
// loop forever...
while true
wend
```

Events

[**private** | **public**] **event** identifier ()
  {declarations}
  {statements}
end event

- *Private* – An optional keyword which ensures that an event is only available from within the module it is declared. Event declarations are private by default.
- *Public* – An optional keyword which ensures that an event is available to other programs or modules.
- *Identifier* – A mandatory event name, which follows the standard identifier naming conventions

Events, unlike subroutines or functions, cannot be called directly from a user program or module. Instead, you call an event via a variable which holds the address of the event handler routine. For example,

```
// event handler type...
type TEvent = event()

// the event handler to call...
event EventHandler()
   high(PORTD.0)
end event

// declare event variable...
dim OnEvent as TEvent
OnEvent = EventHandler  // assign handler
OnEvent()               // call handler
```

Events are also different from subroutines and functions with respect to their frame allocation, which isn't recycled. This is because event pointers cannot be tracked at compile time, preventing the frame usage for an event from being calculated. Events are therefore allocated their own, unique frame space.

Whilst the non-recycle nature of event frame allocation may appear to be a disadvantage, it's actually extremely useful when used with interrupts. An interrupt cannot risk calling a subroutine or function without proper context saving (see **save**…**restore**, later in this document), because it may damage the frame stack. Events on the other hand have a unique frame space, which makes it safe to execute an event from within a subroutine. There are some caveats though. These are:

- A high and low priority interrupt should never call the same event handler.
- Although an event frame space is protected, making calls to other subroutines or functions could cause the program to become unstable, in much the same way as an interrupt would.

In short, events allow you to 'plug in' code into an ISR, without having to alter the interrupt logic. However, it should be noted that you should treat an interrupt triggered event handler with the same respect in terms of context saving and usage as you would the main interrupt routine itself.

The following is an example of an interrupt driven module, which fires an event handler when the hardware USART received some data:

```
module MyRX

// import USART library...
include "USART.bas"

// event handler type...
type TEvent = event()

// local and public variables...
dim FOnDataEvent as TEvent
public dim DataChar as USART.RCRegister.AsChar

// ISR routine...
interrupt OnRX()
   FOnDataEvent()
end interrupt

// initialize...
public sub Initialize(pOnDataEvent as TEvent)
   FOnDataEvent = pOnDataEvent
   USART.ClearOverrun
   USART.RCIEnable = true
   enable(OnRX)
end sub
```

The main program code can now use the generic functionality of the interrupt module to take some specific action, through the use of an event…

```
program MyRXProgram

// import module...
include "USART.bas"
include "MyRX.bas"

// event handler
event OnRX()
   if MyRX.DataChar = " " then
      high(PORTD.0)
   elseif MyRX.DataChar = "*" then
      high(PORTD.1)
   endif
end event

// main program
SetBaudrate(br19200)
MyRX.Initialize(OnRX)
while true
wend
```

In the above example, PORTD.0 goes high if a space character is received and PORTD.1 goes high if an asterisk is received.

Context Saving

save (Item {, Item})
  {statements}
restore

- *pItem* - A variable, subroutine or function to save. A constant 0 will context save the compilers system registers.

Extreme care needs to be taken when using interrupts and events with respect to context saving. This is because a certain number of system registers or frame variables may be allocated by the compiler when executing a routine or performing a mathematical calculation. The integrity of these variables *must be preserved at all times* when executing an interrupt or event.

For example, let's assume we have an interrupt which calls a simple function called *GetValue*(). The program looks like this:

```
// get value function...
function GetValue(pValue as byte) as longword
   result = pValue * pValue * pValue
end function

// some interrupt...
interrupt MyInt()
   dim Value as longword
   Value = GetValue(10)
end interrupt
```

It looks harmless enough doesn't it? But it isn't.

The *GetValue*() function uses seven frame variables. One for the parameter, two to store the intermediate results of the calculation and four for the result, which is a long word. When this function is called, RAM locations 0 to 6 will be altered. If the interrupt is triggered when your program is executing a routine which also uses one or more RAM locations between 0 and 6, then the outcome will be certain disaster. This is because when the interrupt finishes, it will return to the main program and these locations are now permanently damaged. Worst still, the calculation in *GetValue*() also uses a number of system registers to perform the long word multiplication. These will also be damaged.

To correct the problem, use a **save**…**restore** statement block to protect the system and frame registers, like this:

```
// some interrupt...
interrupt MyInt()
   dim Value as longword
   save(0,GetValue)
      Value = GetValue(10)
   restore
end interrupt
```

FSR0 and FSR1 are automatically saved when context saving the system registers. Next, we save the system register block followed by all of the *GetValue()* frame variables. In deciding which items you should context save, it is always worth

- Reading the microcontroller datasheet
- If calling a function or subroutine, examine the source code.
- Examine the generated ASM file. It can help determine what system registers or microcontroller registers are touched.

Compiler Directives

A compiler directive is a non executable statement which tells the compiler how to compile. That is, it won't generate any code that can be run on the target device. For example, some microcontrollers have certain hardware features that others don't have. A compiler directive can be used to tell the compiler to add or remove source code, based on that particular devices ability to support that hardware.

Unlike normal program code, the preprocessor works with directives on a line by line basis. You should therefore ensure that each directive is on a line of its own. Don't have directives and source code on the same line.

Directives can be nested in the same way as source code statements. For example,

```
#ifdef MyValue
   #if MyValue = 10
      const CodeConst = 10
      #else
      const CodeConst = 0
   #endif
#endif
```

#constant

#constant identifier = expression

Creates a constant identifier. A constant identifier is global and its value cannot be changed once set.

#variable

#variable identifier = expression

Creates a variable identifier. A variable identifier is global, but unlike a constant directive, its value and type can be changed. For example,

```
#variable MyValue = 10
#variable MyValue = “Hello”
```

#define

#define identifier [= expression]

Creates a define identifier. A define identifier is global, which can be taken out of scope using the #undefine directive. An optional expression can be assigned to a define directive. This can be used like a constant in any expression.

#undefine

#undefine identifier

Undefines a previously declared define directive.

#ifdef…#else…#endif

```
#ifdef identifier
  {code}
[#else
  {code}]
#endif
```

Include source statements if a define directive has been declared. If the expression evaluates to false (that is, the identifier has not been declared), the code statements following #else are included. The #else directive is optional.

#ifndef…#else…#endif

```
#ifndef identifier
  {code}
[#else
  {code}]
#endif
```

Include source statements if a define directive has not been declared. If the expression evaluates to false (that is, the identifier has been declared), the code statements following #else are included. The #else directive is optional.

#if…#elseif…#else…#endif

```
#if expression
  {code}
[#elseif expression
  {code}]
[#else
```

```
  {code}]
#endif
```

Include source statements if an expression evaluates to true. If the expression evaluates to false, each #elseif is then evaluated. If neither #if or #elseif evaluate to true, code statements following #else are included. Both #elseif and #else directives are optional.

#error

#error "Error string"

Generate a preprocessor error and halt compilation.

#warning

#warning "Error string"

Generate a preprocessor warning. Unlike **#error**, compilation will continue.

#option

#option identifier [= expression]

Creates an option identifier. An option identifier is global and identical to the #define directive in every way, except for two main differences. Firstly, an option identifier can be seen and used in the declaration block of a user program. For example,

```
#option BUFFER_SIZE = 64
const MyBufferSize = BUFFER_SIZE
```

In the above example, the program constant *MyBufferSize* becomes equal to 64, as this is the value assigned to the option identifier BUFFER_SIZE. It is important to note that a user program identifier will always take precedence over an option directive. For example,

```
#option BUFFER_SIZE = 64
const BUFFER_SIZE = 16
const MyBufferSize = BUFFER_SIZE
```

In this example, *MyBufferSize* becomes equal to 16. The other difference between option and define directives is that re-declaring an option twice, using the same identifie,r will not cause an error. For example,

```
#option BUFFER_SIZE = 64
#option BUFFER_SIZE = 16
```

In this example, BUFFER_SIZE will become equal to 64, which is the first option found by the preprocessor. The second declaration of BUFFER_SIZE is ignored.

This capability may appear a little redundant at first, but it's an extremely useful technique to enable users to configure a module from their main program code.

IsOption

IsOption(Identifier)

Used in conjunction with an **#If...#else...#endif** statement IsOption will return true if an **#option** for the referenced identifier has been defined. This is primarily of use for validating the parameters which have been defined in the **#option**. Typically, if the validation fails an error can be raised to the preprocessor and the error message reported to the IDE.

```
#if IsOption(USART_LS) and not (USART_LS in (true, false))
   #error USART_LS, "Invalid option. LS must be TRUE or FALSE."
#endif
#option USART_LS = false
```

This will check to see if the option USART_LS has been previously defined. If it has, it will check the values that have been assigned to the option are true or false. If any another value has been assigned to USART_LS it will generate an error.

The last line will then set a default value for the USART_LS option.

Preprocessor Expressions

The preprocessor allows you to use a rich set of operators within expressions. They include

Relational	<, <=, <>, >=, >, =
Logical	and, or, xor, not
Bitwise	<<, >>, and, or, xor, not
Math	+, -, /, *, mod

In addition to the standard operators show above, the preprocessor also allows the use of the *in* operator. For example,

```
#if _clock = 3 or _clock = 4 or _clock = 8 or _clock = 10
   #define LowSpeed
#endif
```

can be written more simply as

```
#if _clock in (3,4,8,10)
   #define LowSpeed
#endif
```

You can also use a range of values with the *in* operator. For example,

```
#if Value in (1 to 10, 20, 100 to 255)
   #define LegalRange
#else
   #error "Value out of range"
#endif
```

Because the in operator evaluates to true or false, you can apply other boolean operators. For example,

```
#if not (Value in (1 to 10))
   #error "Out of range"
#endif
```

In the above example, an error is generated if value is not in the range 1 to 10.

Predefined Directive Constants

The compiler supports a number of predefined directive constants. These are listed in Appendix 4.

Predefined Subroutines and Functions

Most of the time you will be using subroutines and functions that you have created, or you may use routines from a supplied library. However, the compiler provides a number of inbuilt subroutines and functions, many of which have been specially optimized for use with a PIC® microcontroller.

AddressOf

function addressof(**byref** *variable*) **as word**
function addressof(**byref** *sub* | *function*) **as word** | **longword**

There are times when you may want to access the address of a variable, subroutine or function. For example,

```
function CopyString(byref pStr as string) as string
   FSR1 = AddressOf(pStr)
   FSR0 = AddressOf(Result)
asm
   movf   POSTINC1, W
   bz $ + 6
   movwf  POSTINC0
   bra $ - 6
   clrf   INDF0
end asm
end function
```

You can also use the unary @ operator, rather than *AddressOf()*, to access an address value. For example, @pStr and @Result.

BitOf

function bitof(**byref** *variable*, *masked* **as boolean**= **true**) **as byte**

The **bitof** function returns the bit number of a bit variable. By default, **bitof** returns a masked value. To return a non-masked value, call with the *masked* parameter set to false. For example,

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

sub DisplayBitOf(byref pBit as bit)
   USART.Write("Mask  :", DecToStr(BitOf(pBit)), 13, 10)
   USART.Write("NoMask:", DecToStr(BitOf(pBit,false)), 13, 10)
end sub
USART.SetBaudrate(br19200)
Write(PORTD.7)
```

will output

```
Mask  : 128
NoMask: 7
```

Bound

function bound(**byref** Array()) **as word**

The **bound** function returns the highest addressable index for a given array. For example,

```
sub Init(byref pArray() as byte)
   dim Index as byte
   for Index = 0 to bound(pArray)
      pArray(Index) = $FF
   next
end sub

dim ArrayA(20) as byte
dim ArrayB(32) AS byte

Init(ArrayA)
Init(ArrayB)
```

Clear

sub clear(**byref** *variable*)

The clear subroutine fills a variable with zeros. For example,

```
// declare a structure...
structure TStruct
   a,b as byte
   Value as word
end structure

// create some variables...
dim Array(10) as byte
dim Struct as TStruct
dim Value as byte
```

```
// set to zero...
clear(Array)
clear(Struct)
clear(Value)
```

Dec

sub dec(**byref** *ordinal* [, expression **as word**])

The **dec** subroutine decrements an *ordinal* value either by one, or by an optional ordinal expression. For example,

```
dim Value as word

dec(Value)            // dec by one
dec(Value, 2)         // dec by two
dec(Value, Value / 2) // dec by half of value
```

DelayMS

sub delayms(*expression* **as word**)

The **delayms** subroutine suspends program execution for up to 65535 milliseconds (ms). For example,

```
delayms(100)       // delay 100 ms
delayms(Value)     // delay Value ms
delayms(Value * 2) // delay Value * 2 ms
```

DelayUS

sub delayus(*expression* **as word**)

The **delayus** subroutine suspends program execution for up to 65535 microseconds (µs). For example,

```
delayus(100)       // delay 100 µs
delayus(Value)     // delay Value µs
delayus(Value * 2) // delay Value * 2 µs
```

High

sub high(**byref** *portpin* **as bit**)

The **high** subroutine sets a port pin to a high state. The port pin is automatically set to output. For example,

```
// turn on LED...
dim LED as PORTD.7
high(LED)
```

Inc

sub inc(**byref** *ordinal* [, expression **as word**])

The **inc** subroutine increments an *ordinal* value either by one, or by an optional ordinal expression. For example,

```
dim Value as word
inc(Value)            // inc by one
inc(Value, 2)         // inc by two
inc(Value, Value / 2) // inc by half of value
```

Input

sub input(**byref** *portpin* **as bit**)

The **input** subroutine makes the specified port pin an input. For example,

```
input(PORTD.7)
```

Low

sub low(**byref** *portpin* **as bit**)

The **low** subroutine sets a port pin to a low state. The port pin is automatically set to output. For example,

```
// turn off LED...
dim LED as PORTD.7
low(LED)
```

Output

sub output(**byref** *portpin* **as bit**)

The **output** subroutine makes the specified port pin an output. For example,

```
output(PORTD.7)
```

Terminate

sub terminate**()**

The **terminate** subroutine stops the currently executing program and places it into a continuous loop. The device is also placed into low power mode. For example,

```
// if PORTD.0 goes low, terminate program...
if PORTD.0 = 0 then
   terminate
endif
```

Toggle

sub toggle(**byref** *portpin* **as bit**)

The **toggle** subroutine switches the input or output state of a port pin. That is, a high state becomes low and vice versa. The port pin is automatically set to output. For example,

```
dim LED as PORTD.7
low(LED)        // LED is off
delayms(1000)   // wait one second
toggle(LED)     // switch on
```

Creating and Using Programs

The simplest program that can be compiled using Swordfish is a blank page! The information outlined in this section outlines the key components you will require in moving from a blank page to generating some useful code that will be executed on a PIC® microcontroller.

Program Constructs

[**program** *identifier*]
[**device** = *devicename*]
[**clock** = *frequency*]
  {*modules*}
  {*declarations*}
  {*statements*}
[**end**]

- *Identifier* – A optional program name, which follows the standard identifier naming conventions
- *Device* – An optional PIC® device name
- *Clock* – An optional device crystal frequency
- *Modules* – An optional group of module files
- *Declarations* – An optional group of declarations. For example, constants, structures, variables, subroutine and function declarations
- *Statements* – An optional group of statements. For example, variable assignments, conditional statements and looping statements

If the **program** keyword is used, it must always be at the start of your program and be followed by a unique identifier. For example,

```
program DCMotorControl
```

A program identifier can be useful for documentation purposes. It also makes turning a program into a module a little easier, by substituting **program** with **module**. However, it is entirely optional and there is little to be lost by leaving it out. Like **program**, the **end** keyword is entirely optional and can be used to explicitly bring to an end your program block.

Device

The **device** keyword informs the compiler which PIC® microcontroller you are targeting. If you leave it out, the compiler will default to an 18F452. If you do explicitly assign a microcontroller to **device**, it *must always appear before any module includes or declarations*. If you don't, an error is generated. This is to ensure that any modules or declarations with compiler directives are processed correctly. For example,

```
device = 18F8720
include "EEPROM.bas"
```

In the above example, the EEPROM library uses a number of pre-processor directives to ensure the correct code is generated for devices that have more than 256 byte of onboard EEPROM.

You can access the device name as a string constant from within your main program code by prefixing with an underscore character. For example,

```
device = 18F4520
include "USART.bas"
SetBaudrate(br19200)
USART.Write(_device, 13, 10)
```

Clock

The **clock** keyword informs the compiler what crystal frequency is being used by the target device. This serves two primary purposes. Firstly, it enables to the compiler to produce the correct timings for some of its in-built commands, such as **delayms**. Secondly, it enables a program or module to generate different code for various frequencies through the use of compiler directives. If you don't explicitly set **clock**, it will default to 20MHz. Like the **device** keyword, **clock** *must always appear before any module includes or declarations*. For example,

```
device = 18F4520
clock = 10
include "USART.bas"
```

You can however place **clock** before **device**, like this

```
clock = 10
device = 18F4520
```

The crystal frequency of a device can be represented as a floating point number. For example,

```
clock = 3.2768
```

This can be extremely useful when used in conjunction with the pre-processor to generate different timing code. However, it is important to note that the built in routines will only currently generate the correct timings for a certain range of clock frequencies. These include: 3.58, 4 to 13, 14.32, 15 to 40 and 64Mhz. Any frequencies falling outside of this range will be matched to the nearest supported value.

You can access the clock frequency as a floating point constant from within your main program code by prefixing with an underscore character. For example,

```
device = 18F4520
include "USART.bas"
include "Convert.bas"

SetBaudrate(br19200)
USART.Write(DecToStr(_clock), 13, 10)
```

Program Includes, Declarations and Statements

A module include declaration enables you to import pre-tested subroutines and functions into your main program body. Modules may also contain other public declarations which your program can use, such as constants and variables. To import a module into your program, use the **include** keyword, followed by the modules name. For example,

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

Once imported, you can start using any of the module's public declarations in your main program. For example,

```
SetBaudrate(br19200)
Write(FloatToStr(sin(10)), 13, 10)
```

The subroutines *SetBaudrate()* and *Write()* are located in the USART library and the function *Sin()* is located in the math library.

Some modules may use the same naming convention for identifiers. For example, the USART library and LCD library both have an output subroutine called *Write()*. You can easily use both from within your program through redirection. This is achieved by prefixing the identifier by the modules name and by using the dot notation. For example,

```
include "USART.bas"
include "LCD.bas"

SetBaudrate(br19200)
USART.Write("USART Write", 13, 10) // redirect to USART module
LCD.Write("LCD Write")             // redirect to LCD module
```

If you did not use redirection, then all output would be sent via the USART module. This is because the USART library has been included *before* the LCD library. If you switched them around, then all output would be sent via the LCD module. Redirection is a really good way to document your programs, even if it is not actually needed. For example,

```
USART.SetBaudrate(br19200)
USART.Write("Hello World", 13, 10)
```

Module includes are usually followed by your program declarations such as constants, structures, variables, subroutines and functions. For example,

```
include "USART.bas"
const Factor = 10
dim Index, Value as byte
```

However, you can mix the order if you wish, like this,

```
dim Index, Value as byte
include "USART.bas"
const Factor = 10
dim FloatValue as float
```

Generally, the order in which you declare items is a matter of personal style. However, it is important to remember that even though Swordfish is a multi-pass compiler, it does not support forward referencing. Identifier names are resolved in the first compiler pass. This means you *must declare something before it is used*. For example,

```
dim Alias as Value.Byte0
dim Value as word
```

will generate a number of errors, because *Value* has been referenced before it has been declared. As a guide, it is usually good practice to import modules and declare items in the following order,

- modules
- constants
- structures
- variables
- aliases and modifiers
- subroutines and functions

The final part of a program block is the statements group. That is, the code you write to get the microcontroller to start doing something. The compiler does not require you to give an explicit entry point for your statement code, such as 'main'. You just start writing your code. For example,

```
include "LCD.bas"
include "Convert.bas"

dim Index as byte
delayms(200)
LCD.Cls
for Index = 0 to 10
   LCD.Cursor(1,1)
   LCD.Write(DecToStr(Index))
   delayms(500)
next
```

It should be noted that as soon as you start a statement group, you cannot then start importing modules or make further declarations. For example,

```
include "USART.bas"
SetBaudrate(br19200) // statement
dim Index as byte    // ERROR!
```

Creating Modules

module *identifier*
[**device** = *devicename*]
[**clock** = *frequency*]
  {*modules*}
  {*declarations*}
  {*statements*}
[**end**]

By keeping your most useful routines in a module, you can build programming libraries for other programs to reuse. Modules are also extremely useful for dividing large programs into small, more manageable pieces by storing different parts in separate modules.

- *Identifier* – A mandatory module name, which follows the standard identifier naming conventions
- *Device* – An optional PIC® device name
- *Clock* – An optional device crystal frequency
- *Modules* – An optional group of module files
- *Declarations* – An optional group of declarations. For example, constants, structures, variables, subroutine and function declarations
- *Statements* – An optional group of initialization statements. For example, variable assignments, conditional statements and looping statements

The **module** keyword is mandatory and is followed by unique identifier, which is used to support redirection from other modules or programs. The **end** keyword is entirely optional.

Device and Clock

You can explicitly give a device name and clock frequency inside a module in exactly the same way as you would a program. Although occasionally useful, *this practice is best avoided* as it ties the module to a particular device or clock speed and will also override any settings contained in the main program block.

Module Includes, Declarations and Statements

You can import module libraries in exactly the same way as you would in a normal program, using the **include** keyword. It doesn't matter if you new module and the program that uses it imports the same set of modules. The compiler will only load the necessary libraries once. For more information on importing modules, see the section entitled *Creating and Using Programs*.

The most fundamental difference between program and module declarations is the use of the **private** and **public** keywords. A declaration that is private is only available from within the module it is declared. A program that imports a module cannot access a private declaration. Private declarations are sometimes called *helper declarations*, because they help the module perform one or more specific tasks.

This is in contrast to a declaration that is made public, which is available to other modules and programs. A public declaration is therefore the *interface* to a module. All Swordfish declarations are private by default. If you want other modules or programs to access them, they must be explicitly declared as public. For example,

```
module MyUtils
include "USART.bas"
include "Convert.bas"

// private helper
function GetMax(byref pArray() as byte) as byte
   dim Index as byte
   result = 0
   for Index = 0 to bound(pArray)
      if pArray(Index) > result then
         result = pArray(Index)
      endif
   next
end function

// public interface
public sub DisplayMax(byref pArray() as byte)
   USART.Write("MAX = ", DecToStr(GetMax(pArray)), 13, 10)
end sub
```

The module is named *MyUtils*, which declares a private helper function called *GetMax()* and a public interface subroutine called *DisplayMax().* The DisplayMax() subroutine uses the private helper *GetMax()* to calculate the maximum array value, before outputting via the USART modules *Write()* subroutine. We can now use this module in a program, to output the maximum value of an array,

```
include "USART.bas"
include "MyUtils.bas"
dim Index, Array(5) as byte

SetBaudrate(br19200)
for Index = 0 to bound(Array)
   Array(Index) = USART.ReadByte
next
DisplayMax(Array)
```

However, if you try and make a call to the private helper function *GetMax()* from the program, an error is generated during compilation.

The final part of a module block is the statements group. This can be extremely useful for initializing the module before it is used, as module statements are always executed before the main program executes. For example,

```
module Stack

// private variables...
dim StackItems(100) as byte
dim StackPointer as byte

// push byte onto stack...
public sub Push(pValue as byte)
   if StackPointer <= bound(StackItems) then
      StackItems(StackPointer) = pValue
      inc(StackPointer)
   endif
end sub

// pop item off stack...
public function Pop() as byte
   if StackPointer > 0 then
```

```
      dec(StackPointer)
   endif
   result = StackItems(StackPointer)
end function

// initialise stack pointer
StackPointer = 0
```

In this example, the module manages an array of byte items that can be pushed and popped off a stack. Before the module can be used, it is essential that the stack pointer is initialized to zero. If not, the value of stack pointer may contain any value when the main program executes and the module would certainly fail.

Care must be exercised when using a module statement block, as any code included will be executed before the main program code begins executing. Don't fill it with unnecessary statements; just include code that is absolutely essential for the correct operation of a module.

Appendix 1 - Operators

Operator Precedence

Level	Operators
1	@, unary -, unary +, NOT
2	*, /, MOD
3	+, -
4	<<, >>
5	=, <, >, <=, >=, <>
6	AND
7	OR, XOR

Operators at level one have the highest precedence, with operators at level seven having the lowest precedence. Parentheses can be used to override the order of precedence rules, allowing parts of an expression to be evaluated before others.

Relational Operators

The operators used for comparison are called relational operators. Applying a relational operator will yield a result of true or false.

Operator	Meaning
=	Is equal to
<	Is less than
>	Is greater than
<=	Is less than or equal to
>=	Is greater than or equal to
<>	Is not equal to

The only relational operators supported for string types are equal and not equal. Because the internal binary representation of floating point numbers may not be exact, the equality operator should be avoided for floating point numbers.

Mathematical Operators

Operator	Meaning
*	Multiply
/	Divide
+	Addition
-	Subtraction
MOD	Modulus

In addition to the mathematical operators shown above, you can use the unary operator to change the sign. For example,

```
dim Value as shortint
Value = 10     // Value = 10
Value = -Value // Value = -10
```

Logical Operators

Operator	Meaning
NOT	Negate
AND	Logical AND
OR	Logical OR
XOR	Logical XOR

Logical operators can be used to build conditional expressions, for example, when using **if…then**, **while…wend** and **repeat…until** statements. Applying a boolean operator yields either **true** or **false**.

Bitwise Operators

The following operators perform bitwise manipulation on ordinal operands, which include byte, shortint, word, integer, longword and longint.

Operator	Meaning
NOT	A bitwise NOT or complement
AND	Bitwise AND
OR	Bitwise OR
XOR	Bitwise XOR
<<	Shift Left
>>	Shift Right

Appendix 2 - Reserved Words

A list of compiler reserved words is shown below.

absolute	**clear**	**end**	**low**	**shortint**
access	**clock**	**select**	**mod**	**step**
addressof	**compound**	**exit**	**module**	**string**
and	**config**	**false**	**next**	**structure**
as	**const**	**float**	**noinline**	**sub**
asm	**continue**	**for**	**not**	**system**
auto	**dec**	**function**	**null**	**terminate**
bit	**delayms**	**goto**	**or**	**then**
bitof	**delayus**	**high**	**output**	**to**
boolean	**device**	**if**	**port portpin**	**toggle**
bound	**dim**	**inc**	**private**	**true**
break	**disable**	**include**	**program**	**type**
byref	**eeprom**	**inline**	**public**	**until**
byrefconst	**else**	**input**	**repeat**	**wend**
byte	**elseif**	**integer**	**restore**	**while**
byval	**enable**	**interrupt**	**save**	**word**
case	**end**	**longint**	**select**	**xor**
char	**endif**	**longword**		

Appendix 3 - Types, Modifiers and Constants

Core Types

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

Variable Modifiers

Type	Modifiers
Byte, ShortInt	0..7, Bits(0..7), Booleans(0..7)
Word, Integer	0..15, Bits(0..15), Booleans(0..15) Byte0, Byte1, Bytes(0..1)
LongWord, LongInt, Float	0..31, Bits(0..31), Booleans(0..31) Byte0, Byte1, Byte2, Byte3, Bytes(0..3) Word0, Word1, Words(0..1)
	0..7, Bits(0..7), Booleans(0..7)

Promotional Modifiers

Variable Type	**Legal Promotional Modifiers**
Byte, Shortint, Char	AsWord, AsInteger, AsLongWord, AsLongtInt, AsFloat
Word, Integer	AsLongWord, AsLongtInt, AsFloat

Inbuilt Constants

True
False
Null

Appendix 4 - Predefined Directive Constants

_device	Target device name. For example, 18F452
_clock	Clock frequency
_core	Device core. Currently 16 only.
_maxram	The number of device RAM locations
_maxrom	The number of device ROM locations
_ports	The number of device ports
_adc	Number of device ADC channels
_adres	ADC resolution
_eeprom	The number of device EEPROM locations
_usart	The number of hardware USARTS
_usb	USB support
_mssp	The number of MSSP modules available
_ccp	The number of CCP module available
_eccp	The number of extended CCP modules available
_comparator	The number of comparator modules available
_psp	The number of onboard Parallel Slave Ports (PSP) available
_ethernet	Ethernet support
_can	CAN support
_flash_write	FLASH write capability