What's New In SX/B 2.0

Additions:

LOCAL VARIABLES - STACK
TASK SWITCHING - TASKS, TASK...ENDTASK, PAUSETICKS
BIT MANIPULATION - GETBIT, PUTBIT
IEEE-754 32-bit constants "#"
ANDN (&~) Operator
ABS Operator
SGN Operator
I2CSPEED Directive
STR command – Converts value to ASCII decimal digits
HEXSTR command – Converts value to ASCII hex digits
VAL command – Converts ASCII decimal digits to value
HEXVAL command – Converts ASCII hex digits to value
CREAD, CREADINC, CDATA – Compact (12 bit) Data
'{$IFFREQ cond value} compiler directive
'{$WARNING xxx} compiler directive
'{$ERROR xxx} compiler directive
'{$IFUSED subname}
'{$IFNUSED subname}
'{$USES subname}
__DEFAULT constant
WCON directive (Word Constant)

Enhancements:

Large arrays on SX28
SWAP var1, var2 ' Swaps values in variables
XOR, OR, AND for bit variables
Variable modifiers "@", SPAN, ALIGN, BANK
GET sourceVar, destVar1 TO destVar2
PUT destVar, sourceVar1 TO sourceVar2
INTERRUPT EXT_RISEx | EXT_FALLx ` x=prescaler 1,2,4,8,16,32,64,128,256
    RTCC increments on external signal, interrupts when RTCC overflows
IF var THEN ' acts like "IF var <> 0 THEN"
I2CSEND & I2CRECV now allow the slave to perform "clock stretching"
SUB and FUNC allow your to specify parameter as BYTE or WORD
    MySub SUB 2,3,WORD,BYTE
    MyFunc FUNC 2,2,3,WORD,WORD,BYTE
Conditional Compiler Directives can now be nested (32 level deep)

Changes:

Using array names
DEVICE – TURBO, OPTIONX, STACKX added automatically for SX20/SX28

Fixes:

Error "PROGRAM" not used if LOAD precedes PROGRAM
Wordvar / 1 gave invalid result
SX48 PWM command generated error __TRISx not defined
SEROUT WordVar as array Index – should error but sends invalid data
Using the tilde with an array bit myBit = ~myArray(1).3
Using "\" within an ASM...ENDASM block
INTERRUPT now saves __PARAM5/__PARAMCNT
I2CSTOP on SX48 could change direction of pins on same port

Local Variables

Local variables are variables that ONLY exist within the subroutine that declares them. In SX/B 1.51 you had to declare any variables used in subroutines in the main program. And if you used the same variable in two subroutines (one that calls the other) you could have problems. But if you just create different variables for each subroutine, you can quickly run out of variable space.

Local variables solve this problem. You can declare variables WITHIN the subroutine. They are created in a special array call a "Stack". This stack holds the variables values and the space is re-used.

Here is an example of the problems caused by NOT using local variables:

```
temp1 VAR BYTE
temp2 VAR BYTE

SUB SendOne
  temp1 = __PARAM1
  SEROUT SPin, Baud, temp1
ENDSUB

SUB SendStars
  temp1 = __PARAM1 ' count
  FOR temp2 = 1 to temp1
    SendOne "*"
  NEXT
ENDSUB
```

At first glance this program fragment may look fine. But there is a problem. The SendStars subroutine is using temp1 to hold the count, but after the first time SendOne is called, temp1 will hold the value 42 (the ascii value of "*"). So the FOR...NEXT loop in SendStars will always run 42 times, no matter what value you pass it.

Now here is the same program using local variables:

```
SUB SendOne
  l_temp1 VAR BYTE

  l_temp1 = __PARAM1
  SEROUT SPin, Baud, l_temp1
ENDSUB

SUB SendStars
  l_temp1 VAR BYTE
  l_temp2 VAR BYTE

  l_temp1 = __PARAM1 ' count
  FOR l_temp2 = 1 to l_temp1
    SendOne "*"
  NEXT
ENDSUB
```

There are a few things to note with the new version of the program. First the variables are prefixed with "l_" (that is a lowercase L not a digit one). This is a personal preference of mine because local variables cannot have the same name as a global variable (although two subroutines CAN use the same local variable name).

Second the variables are actually array elements. This is because the stack is itself an array. Most commands have been enhanced to allow an array element where a byte variable is required. One exception is in array indexing. An array index variable cannot be an array element. In other words if you create a local variable "l_temp1 VAR BYTE" you cannot use l_temp1 as an array index as in myArray(l_temp1). The solution is to create a global variable, and save and restore it's value into a local variable. As in:

```
index VAR BYTE

SUB RecvCommand
  l_data      VAR BYTE (10)
  l_holdIndex VAR BYTE


  l_holdIndex = index
  ' Get data array
  FOR index = 0 TO 9
    SERIN SPin, Baud, l_data(index)
  NEXT
  ' Process data array

  index = l_holdIndex
ENDSUB
```

This method will work even if two subroutines use the same “index” variable.

Local variable type allowed are: BIT, BYTE, BYTE(xx), and WORD

Task Switching

Task switching is the ability to schedule a subroutine to be run periodically without the main program having to explicitly run it. As you might have guessed the tasks are scheduled from inside an interrupt routine. But they are not really called from the interrupt routine. Here is what happens:

A) The interrupt is triggered and the interrupt code executes the “TASKS RUN” command.

B) If a task is scheduled to be run, the interrupt return address saved, and is replaced with the task’s entry address. The rest of the interrupt code is executed.

C) Assuming a task WAS scheduled, when the RETURNINT is executed control goes to the task (instead of back to the main code that was executing when the interrupt occurred).

D) When the task code is complete, execution is routed back to the main code where the interrupt originally occurred.

Some will say “Why not just perform the task inside the interrupt routine ?”. Well let’s say you have an interrupt that happens every 1 millisecond. And you have a task that takes several milliseconds. If you try to perform that task inside the interrupt routine you will miss interrupts. This is the beauty of tasks: Interrupts continue to be executed even while the task code is running.

Here are the commands associated with task switching:

TASKS - This command controls how tasks are run.
TASK...ENDTASK - This works just like SUB...ENDSUB
PAUSETICKS - This delays for x tasks ticks

Two important concepts with task scheduling is the "task tick", and the "task slot". A task tick is the smallest increment of time that a task can be scheduled to run. A task slot is an array of what tasks to run, and how often to run them. The SX28 can have up to 5 task slots (can have 5 tasks in the schedule), the SX48 can have up to 8 task slots. Different task routines can be assigned to any slot at any time. So you can have more than 5 (or 8) tasks, but only 5 (or 8) can be scheduled at any one time.

Let's look at an example program that flashes an LED using tasks:

```
DEVICE SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ   4_000_000

LED    PIN RA.0 OUTPUT

SUB INTERRUPT 1000
  TASKS RUN, 10
  RETURNINT
ENDSUB

FlashLED TASK

PROGRAM Start

Start:
  TASKS SET, 0, FlashLED, 50
  TASKS ENABLE
END

TASK FlashLED
  TOGGLE LED
ENDTASK
```

As you can see we setup an interrupt to occur 1000 times a second (every 1 millisecond). Then the have the line "TASKS RUN, 10" this means that tasks will get a chance to run every 10 interrupts (or every 10 milliseconds). So our task tick is 10 milliseconds.

Further down after the "Start:" label we see the line "TASKS SET, 0, FlashLED, 50". The "TASKS SET" means that we are going to set a task slot. The zero means that we are going to set task slot #0 (task slots are numbered like arrays, starting at 0). Of course the "FlashLED" is the name of the task we want to run. And the "50" is how often to run the task. This parameter means to run the task every 50 task ticks. Since our task tick is 10 milliseconds, this task will be run every 10 * 50 or 500 milliseconds (2 times a second).

The next line "TASKS ENABLE" enables the task scheduler. You can enable or disable the task schedule so that tasks can be syncronized. You can also START, STOP, or SUSPEND an individual task slot.

That's it. The main program just has an "END" command next. This just keeps the SX in an infinite loop. But the LED will flash right on schedule.

The PAUSETICKS command will pause the main program for x number of task ticks. So if we use "PAUSETICKS 100" our main program will wait for 10 * 100 = 1000 milliseconds (1 second). Remember our task tick is 10 milliseconds.

Here are all the options for the TASKS command:

TASKS SET, slot, taskname{, interval_ticks{, delay_ticks}}
      Sets up a task slot. If interval_ticks is zero or not given, the task will run one time only. If delay_ticks is one or not given the task will run on the next task tick.

TASKS DISABLE
      Stops the task scheduler. No tasks will run.

TASKS ENABLE
      Starts the task scheduler.

TASKS STOP, slot
      Permanently suspends the task in "slot" from being scheduled.

TASKS START, slot
      Resumes scheduling of the task in "slot".

TASKS SUSPEND, slot, ticks
      Suspends the task in "slot" for "ticks" task ticks.

It should be noted that the same task routine can be assigned to more than one task slot. When any task routine is run, the __PARAM1 variable will automatically be set to the task slot that called the task routine. This allows one task routine to perform differently depending on what task slot called the routine.

Another advantage of tasks is you “know” they are called at the end of the interrupt routine, so when a task starts you can be assured that another interrupt will not be called for awhile. So let’s say you want to send a couple bytes of serial data every second. If your interrupt rate is low enough and your serial baud rate is fast enough, then you can rest assured that an interrupt will not occur while you are sending the serial data. For example let’s say you have the interrupt rate at 10 milliseconds, and you want to send 2 characters at 9600 baud. Well each character at 9600 baud will take 10/9600 (there are 10 bits sent) so that is 1/960 or 1.04 milliseconds per character. Sending two characters should only take 2.08 milliseconds. As long as your interrupt routine doesn’t take more than 7.92 milliseconds you should be fine. Remember that the “TASKS RUN” command does not really RUN your task. It only schedules it to run after ALL of the interrupt code.

You would use code something like:

```
SUB INTERRUPT 100   ' Interrupt rate is 10 milliseconds
  ' Other interrupt code
  TASKS RUN, 10     ' Task tick is 100 milliseconds
  ' Other interrupt code
ENDSUB

SendData  TASK

PROGRAM Start

Start:
  TASK SET 0, SendData, 10, 10 ' Task rate is 10 ticks (once per second)
  DO
  LOOP
END

TASK SendData
  SEROUT SoutPin, N9600, Data1
  SEROUT SoutPin, N9600, Data2
ENDTASK
```

An example of where you would use a task interval of zero (only runs once) would be if you wanted an LED to light for a certain time, then stay off. You would turn the LED on, setup a task that turns if off to run once after “x” ticks.

TASK VARIABLES

To keep track of the task schedule there are various variables created in the array space. The space used is 16 bytes plus 3 bytes per task. So if you have a maximum of 4 tasks "TASKS RUN 10, 4", 16+12 bytes of array space will be required.

Here is what the variables would look like if we created them in SX/B. Of course you DON'T declare them, but this is just to show what it would look like. This example assumes 4 tasks.

```
__TASKSRUNAT      VAR BYTE(4) ' 4=number of tasks
__TASKSINTERVAL   VAR BYTE(4) ' 4=number of tasks
__TASKSSTATUS     VAR BYTE(4) ' BITS: 0-4=tasksID; 5=RunOnce; 6=Stopped

__TASKSKERNEL     VAR BYTE(15)
__kMisr_W         VAR __TASKSKERNEL(0) ' Kernel save area
__kMisr_M         VAR __TASKSKERNEL(1)
__kMisr_ST        VAR __TASKSKERNEL(2)
__kMisr_FSR       VAR __TASKSKERNEL(3)
__kMisr_PCL       VAR __TASKSKERNEL(4) ' Main code return address
__kMisr_PCH       VAR __TASKSKERNEL(5)
__kMisr_Timer     VAR __TASKSKERNEL(6)
__kMisr_Count_LSB VAR __TASKSKERNEL(7)
__kMisr_Count_MSB VAR __TASKSKERNEL(8)
__kFlags          VAR __TASKSKERNEL(9)
__kMisrEnabled    VAR __kFlags.5
__kMisrRunning    VAR __kFlags.6
__kMisrFinished   VAR __kFlags.7
__kPARAM1         VAR __TASKSKERNEL(10)
__kPARAM2         VAR __TASKSKERNEL(11)
__kPARAM3         VAR __TASKSKERNEL(12)
__kPARAM4         VAR __TASKSKERNEL(13)
__kPARAM5         VAR __TASKSKERNEL(14)
__kTaskPending    VAR __TASKSKERNEL(15)
```

The program code to handle tasks is about 200 instructions. So there is quite a bit of overhead. For this reason TASKS is not suited to operations that need to be executed in very often (for example high speed serial data). Because the time spend in the TASKS overhead would far outweigh the cycles available to perform the task.

PUTBIT and GETBIT

```
PUTBIT destvar, bit_pos, value(0, 1, 2)
      destvar is the variable to change
      bit_pos is the bit to be changed
      value is 0 to clear bit; 1 to set bit; 2 to invert bit

GETBIT sourcevar, bit_pos, destvar
      sourcevar is the variable to detect
      bit_pos is the bit to be detected
      destvar receives the value of the bit (0 or 1)
```

IEEE-754 32-Bit Constants

The compiler now supports IEEE-754 32-bit floating point constants. These are four byte values that are used by various math coprocessors. Note that the compiler does NOT support any type of math using these values. This feature is only included to make it easier to develop your own floating point routines. Or to ease the use of a floating point coprocessor. These values can be assigned to an array of 4 elements. Like so:

```
pi     VAR BYTE (4)
minus1 VAR BYTE (4)

pi() = #3.1415927
minus1() = #-1.0
```

Note that for negative numbers, the minus sign comes AFTER the "#" symbol.

Large arrays on SX28

The compiler now supports large arrays (array with more than 16 elements) on the SX28. Note that if you use assembly language with arrays, these large arrays require different code. See the assembly listings of the SX/B code to see the differences.

SWAP var1, var2

SWAP may now be used with two parameters. This will swap the values of the two parameters. This is useful in many sorting routines.

XOR, OR, AND for bit variables

Bit variables can now use the XOR, OR, and AND binary operators. For example:

```
bit1 = bit2 XOR bit3
```

Variable modifiers "@", SPAN, ALIGN, BANK

There are several new modifiers when declaring variables.

The “@” is used to assign a variable to a specific address. This address may be a constant (like $10) or another variable or array element.

The compiler will treat variables as the type they are declared regardless of WHERE they are declared. For example the global memory area (up to $0F), and the default bank ($10 to $1F) are used for the normal variables. RAM locations above $1F are for arrays. However if you use "temp VAR BYTE @ $30", the variable temp will be considered a normal byte variable even though it is in the array RAM area.

If you need to declare a variable in the default bank that you want to be used as an array element simply alias it with the __RAM() array. For example "temp VAR __RAM($10)". temp will now be treated as an array element, even though it is located in the default bank.

All these options are added to allow you to create banks of "normal" variables within an array. For example let's say you have a subroutine that requires 3 WORD variables. You could do this:

```
moreVars  VAR BYTE (16)

SUB DoAlot
  tempW1 VAR WORD @ moreVars(0)
  tempW2 VAR WORD @ moreVars(2)
  tempW3 VAR WORD @ moreVars(4)

  BANK @moreVars
  tempW1 = 1000
  tempW2 = 2000
  tempW3 = tempW1 + tempW2
  BANK
ENDSUB
```

NOTE: That you must use an “@” before the array name that you want to use with BANK. This is different that in previous versions of SX/B.

Note that after the "BANK @moreVars" command, you will not be able to access variables in addresses $10 to $1F (unless you make aliases of them using __RAM(). Also note that "BANK" by itself resets to the default bank.

SPAN is used when declaring arrays on the SX28. The SPAN modifier allows the array to span over two memory banks. This is useful if you have a fragmented memory map and an array cannot fit within any single bank. Note that SPAN arrays generate more code to access them.

BANK is used when declaring arrays on the SX48. The BANK modifier creates an array that is entirely contained within one memory bank. Since a memory bank is 16 bytes long, any array with the BANK modifier cannot be larger than 16 elements.

ALIGN is used to declare an array that starts at the beginning of a memory bank. An array with the ALIGN modifier will always begin at an address of $x0 (where x can be any of the valid memory banks).

Note that SPAN and BANK are mutually exclusive. If you use one, then you cannot use the other on the same variable declaration.

GET sourceVar, destVar1 TO destVar2

Let's say you have some global variables defined as:

```
temp1 VAR BYTE
temp2 VAR BYTE
temp3 VAR BYTE
temp4 VAR BYTE
```

And you want to get values from an array into these variables. You can now use:

```
GET myArray, temp1 TO temp4
```

This is the same as:

```
GET myArray, temp1, temp2, temp3, temp4
```

PUT destVar, sourceVar1 TO sourceVar2

Let's say you have some global variables defined as:

```
temp1 VAR BYTE
temp2 VAR BYTE
temp3 VAR BYTE
temp4 VAR BYTE
```

And you want to put values into an array from these variables. You can now use:

```
PUT myArray, temp1 TO temp4
```

This is the same as:

```
PUT myArray, temp1, temp2, temp3, temp4
```

CREAD, CREADINC, CDATA

Each location of program memory on the SX can actually store 12 bits. DATA and WDATA only use 8 of the 12 bits. Programs that need 12 bit values (0 to 4095) can use the CDATA (compact data) command and only use half the program memory as WDATA.

CREAD is used to read the CDATA. The variable that holds the value MUST be a WORD.

```
temp  VAR BYTE
tempW VAR WORD

FOR temp = 0 TO 11
  CREAD MyData + temp, tempW
  ' Use tempW as needed
NEXT

MyData:
  CDATA 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095
```

Assumed Use of array names

In SX/B 1.xx array names are constants that hold the address where the array values are stored. In SX/B 2.0 array names are variables, and you must prefix the array name with an "@" if you want the address of the array.

```
SX/B 1.xx   arrayName = Address of array
SX/B 2.xx   arrayName = Value stored in array
```

If you use just the array name as a parameter in a subroutine or function call, SX/B 1.xx will assume you want the ADDRESS of the array. In SX/B 2.0 the compiler assumes you want to pass the VALUES stored in the array.

In SX/B 2.0 you must use @arrayName if you want the address of the array.

I2CSPEED Directive

The I2CSPEED directive will adjust the speed of the I2C commands(I2CSTART, I2CSTOP, I2CSEND and I2CRECV). Using "I2CSPEED 4" will make the commands execute 4 times faster. Using "I2CSPEED 0.5" will make the commands execute 1/2 as fast. This feature was added because some devices are capable of faster speeds, and some devices require slower speeds. NOTE: This is a directive and not a command. It does not generate any code by itself, and functions completely top-down.

STR Command

The STR command will convert values to ASCII characters. The syntax is "STR array, var{,option}". The array holds the ASCII characters (5 elements for a WORD variable and 3 elements for a BYTE variable). The var is variable to be converted, and the options are 0=Leading Zeros; 1(default)=Leading Spaces; 2=Non-Ascii. Here are some examples:

```
tempStr VAR BYTE(5)
tempW   VAR WORD

tempW = 1234
STR tempStr, tempW ' tempStr will be " 1234" (note leading space)
STR tempStr, tempW, 0 ' tempStr will be "01234" (note leading zero)
STR tempStr, tempW, 2 ' tempStr(0)=0; tempStr(1)=1; tempStr(2)=2; tempStr(3)=3;    tempStr(4) = 4
```

VAL Command

The VAL command will convert ASCII characters to values. The syntax is "VAL array, var{,digits}". The array holds the ASCII characters (5 elements for a WORD variable and 3 elements for a BYTE variable). The var is variable to hold the value, and the optional digits is how many digits to convert (default is 5 for WORD variables; 3 for BYTE variables). Note that SPACES are assumed to be "0". Here are some examples:

```
tempStr VAR BYTE(5)
tempW   VAR WORD
temp    VAR BYTE

PUT tempStr, "12345"
VAL tempStr, tempW ' tempW will be 12,345
VAL tempStr, temp  ' temp will be 123
```

HEXSTR Command

The HEXSTR command will convert values to ASCII hex characters. The syntax is "HEXSTR array, var". The array holds the ASCII characters (4 elements for a WORD variable and 2 elements for a BYTE variable). The var is variable to be converted. Here are some examples:

```
temp     VAR BYTE
tempW    VAR WORD
TempStr4 VAR BYTE(4)
TempStr2 VAR BYTE(2)

Temp = 123
HEXSTR tempStr2, temp ' tempStr2 will be "7B"
tempW = 1234
HEXSTR tempStr4, tempW ' tempStr4 will be "04D2"
```

HEXVAL Command

The HEXVAL command will convert ASCII hex characters to values. The syntax is "HEXVAL array, var". The array holds the ASCII characters (4 elements for a WORD variable and 2 elements for a BYTE variable). The var is variable to hold the value. Here are some examples:

```
tempStr VAR BYTE(4)
tempW   VAR WORD
temp    VAR BYTE

PUT tempStr, "04D2"
VAL tempStr, tempW ' tempW will be 1234
PUT tempStr, "7B"
VAL tempStr, temp  ' temp will be 123
```

ABS Operator

The ABS operator will return the absolute value of a (implied) signed value.

```
tempW1 VAR WORD
tempW2 VAR WORD

tempW1 = 1000
tempW1 = tempW1 - 1500
tempW2 = ABS tempW1
' tempW1 = 65036 (-500)
' tempW2 = 500
```

SGN Operator

The SGN operator will return either 1, 0 or -1 according to the sign of the given value. Note that -1 is 255 for BYTE variables and 65535 for WORD variables.

WCON

WCON allows you to declare a constant as a WORD value. Even if the value is less than 256, the value will be treated as a 16-bit value when passing to a SUB, FUNC or when used in a DATA statement.

Enhanced SUB and FUNC declarations

You may now specify the size of parameters passed to SUB and FUNC routines. When using the "BYTE" and "WORD" modifiers, you MUST specify all of the parameters before it. For example:

```
MySub   SUB 2,2,WORD ' Accepts a single WORD parameter

MySub   SUB 2,WORD ' This is NOT legal

MyFunc  FUNC 2,2,3,WORD,WORD,BYTE ' Returns a WORD, accepts a WORD or a WORD,BYTE

MyFunc  FUNC 2,2,WORD ' This is NOT legal
```

What happens when you specify a WORD parameter, the compiler will expand any byte value to a word before the subroutine or function is called.

```
MySub 10
      __PARAM1 = 10; __PARAM2 = 0

MySub byteVar
      __PARAM1 = byteVar; __PARAM2 = 0

MySub 2561
      __PARAM1 = 1; __PARAM2 = 10

MySub wordVar
      __PARAM1 = wordVar_LSB; __PARAM2 = wordVar_MSB

wordVar = MyFunc 10
            __PARAM1 = 10; __PARAM2 = 0; __PARAMCNT = 2

wordVar = MyFunc byteVar
            __PARAM1 = byteVar; __PARAM2 = 0; __PARAMCNT = 2

wordVar = MyFunc 2561
            __PARAM1 = 1; __PARAM2 = 10; __PARAMCNT = 2

wordVar = MyFunc wordVar
            __PARAM1 = wordVar_LSB; __PARAM2 = wordVar_MSB; __PARAMCNT = 2

wordVar = MyFunc 10, 11
            __PARAM1 = 10; __PARAM2 = 0; __PARAM3 = 11; __PARAMCNT = 3

wordVar = MyFunc byteVar, byteVar2
            __PARAM1 = byteVar; __PARAM2 = 0; __PARAM3 = byteVar2; __PARAMCNT = 3

wordVar = MyFunc byteVar, byteVar2, byteVar3
            NOT LEGAL BECAUSE byteVar is promoted to WORD
```

New Compiler Directives

'{$IFFREQ is a compiler directive you can use to create different code depending on the clock speed of the SX. Here are some examples:

```
'{$IFFREQ < 4_000_000}
'{$ENDIF}

'{$IFFREQ >= 4_000_000 <= 20_000_000}
'{$ENDIF}

'{$IFFREQ > 20_000_000}
'{$ENDIF}
```

The middle example will only compile between the IFFREQ and ENDIF if the frequency is between 4MHz and 20MHz.

'{$WARNING and '{$ERROR are used to generate compile time warnings or errors. Let's say you have a routine that will only work correctly if the clock speed is 20MHz or above. You could do something like this:

```
'{$IFFREQ < 20_000_000}
'{$ERROR Clock speed is too low}
'{$ENDIF}
```

Note that a warning does NOT prevent the program from compiling, where an error DOES prevent the program from compiling.

Remember these are "compile time" errors and warnings. So no actual SX code is generated for them. Don't try to do something like this:

```
IF A > 100 THEN
'{$ERROR A is too high}
ENDIF
```

If you do, you will get the error every time you compile the program.

'{$IFUSED subname} is a compiler directive that you can use to create different code depending on if a subroutine, function, or task has been used in the previous code. Most often this would be used to generate an empty subroutine if that subroutine was not used. For example:

```
'{$IFUSED Delay}
SUB Delay
l_tempW VAR BYTE (2)
  IF __PARAMCNT = 1 THEN
    l_tempW = __PARAM1
  ELSE
    l_tempW = __WPARAM12
  ENDIF
  PAUSE l_tempW
ENDSUB
'{$ELSE}
SUB Delay
ENDSUB
'{$ENDIF}
```

'{$IFNUSED subname} is the same but compiles if the subroutine is NOT used.

'{$USES subname} marks the subname as being used. The compiler cannot detect if a subname is used inside assembly code. If you call a subroutine in assembly code, use this directive to mark the subroutine as used.

Errors and Warnings

Errors:

1 INVALID VARIABLE NAME
The variable name is a reserved word.

2 DUPLICATE VARIABLE NAME
The variable name has already been used. The name could be a SX/B predefined variable (like __PARAM1).

3 VARIABLE EXCEED AVAILABLE RAM
SX/B has separate RAM areas for array and non-array variables. The array area is much larger. In many cases you can create a 1 byte array instead of a byte variable, or a 2 byte array instead of a word variable.

4 CONSTANT EXPECTED
Some parameters must be a constant. A variable cannot be used.

5 BYTE PARAMETER EXPECTED
The parameter must be a byte.

6 INVALID UNARY OPERATOR
Only "-", "~" are valid unary operators.

7 INVALID REGISTER OPERATION
Registers cannot be used in math assignments.
Example: PLP_C = temp + 5

8 INVALID PARAMETER
The parameter is not the correct type.

9 SYNTAX ERROR
The command's required syntax has not been followed.
Example: FOR temp = 1 TOO 5

10 INVALID NUMBER OF PARAMETERS
The command requires more or less parameters than has been given.
Example: FOR temp = 1

11 BYTE VARIABLE EXPECTED
A byte variable is required.

12 NOT A "FOR" CONTROL VARIABLE
Usually caused by a "NEXT" without a preceeding "FOR"

13 BIT VARIABLE EXPECTED
A bit variable is required.

14 BAUDRATE IS TOO LOW
The allowed baud rate is determined by the SX clock frequency. If possible operate the SX at a slower clock rate to achieve the required baud rate.

15 BAUDRATE IS TOO HIGH
The allowed baud rate is determined by the SX clock frequency. If possible operate the SX at a faster clock rate to achieve the required baud rate.

16 UNKNOWN COMMAND
Command not recognized or variable name misspelled.

17 COMMA EXPECTED
Most parameters must be separated by a comma.

18 EXPECTED A VALUE BETWEEN 0 AND 7
Bit values must be between 0 and 7 for byte variables.

19 BIT IS NOT A HARDWARE PIN
For example the first parameter of the RCTIME command must be a hardware pin.

20 BIT CONSTANT EXPECTED

21 INTERRUPT MUST BE USED BEFORE ''PROGRAM''
The INTERRUPT routine must come before PROGRAM and any SUB, FUNC, or TASK declarations.

22 FOR WITHOUT NEXT
A “FOR” loop was started, but no “NEXT” was found to end the loop.

23 NEXT WITHOUT FOR
A “NEXT” command was encountered, but not a matching “FOR” command.

24 UNKNOWN VARIABLE NAME
Variable name misspelled.

25 TOO MANY SUBS DEFINED
Only 127 subroutines can be defined (SUB or FUNC)

26 ELSE OR ENDIF WITHOUT IF
“ELSE” or “ENDIF” was encountered without a preceding “IF”.

27 LOOP WITHOUT DO
“LOOP” was encountered without a preceding “DO”

28 EXIT NOT IN FOR-NEXT OR DO-LOOP
The EXIT command can only be used inside a FOR-NEXT or a DO-LOOP structure.

29 FREQUENCY DIFFERENT FROM DEVICE SETTING
If you use the internal SX clock, the FREQ parameter must match the device parameter.

30 NOT ALLOWED ON THIS DEVICE
You have tried to use a SX48 feature on the SX28.

31 NO "PROGRAM" COMMAND USED
You must use the “PROGRAM” directive in your program.

32 TOO MANY DEFINES
Only 512 compile directive defines are allowed.

33 INVALID LOCAL VARIABLE

34 NOT IN A SUB OR FUNC
“ENDSUB” or “ENDFUNC” encountered without a preceding “SUB” or “FUNC”.

35 SUB OR FUNC CANNOT BE NESTED
A subroutine cannot be defined inside another subroutine.

36 STACK MUST BE USED BEFORE VARIABLES ARE DEFINED
The “STACK” directive must be used before other variables are defined.

37 NOT VALID INSIDE SUB
Command is not valid inside a subroutine. “PROGRAM” for example.

38 COULD NOT READ SOURCE FILE
Could not read file specified by “LOAD” or “INCLUDE”.

39 CANNOT CREATE LOCAL VARIABLE, STACK NOT DECLARED
You must declare the STACK size before creating local variables.

40 DIRECTIVE ERROR: *message*
Error generated by directive ‘{$ERROR

41 NO FREQ SPECIFIED
Issued by the PROGRAM directive if no FREQ directive has been used.

Warnings:

1 NOT RECOMMENDED WITH INTERNAL CLOCK
SERIN and SEROUT are not recommended using the internal clock.

2 INTERRUPT RATE WILL BE
The interrupt rate is not exactly what was specifed.

3 ENDFUNC USED WITHOUT RETURN
Can be ignored if __PARAMx was loaded manually in function.

4 DIRECTIVE WARNING: *message*
Warning generated by directive ‘{$WARNING

5 NOT COMPATIBLE WITH NOPRESERVE
Attempted to use TASKS RUN in interrupt routine using NOPRESERVE