ScriptBasic for the DXM Controller

Instruction Manual

Original Instructions
191745 Rev G
15 May 2019

BANNER®
more sensors, more solutions

Contents

Program the DXM Controller using action rules, read/write maps, or by using a ScriptBasic program. This short tutorial demonstrates how to work with ScriptBasic files using the DXM Configuration Tool.

Use any text editor to create a ScriptBasic program (*.sb). Use the DXM Configuration Tool software to upload the file to the device. Debug ScriptBasic programs using print statements to the console. All console messages are sent to the USB port.

The DXM Controller is a Modbus device that transfers information using data registers. ScriptBasic can read/write on-board registers (local registers) as well as remote device registers using Modbus RTU protocol. After data is in ScriptBasic as variables, any typical programming operation can be performed on the data. The custom commands for DXM ScriptBasic are the register commands, GETREG, SETREG, MULTIGET and MULTISET. Use these commands as basic building blocks for gathering data in ScriptBasic. See the language syntax and program examples in the DXM ScriptBasic manual for more information.

Creating a ScriptBasic File

Use any text editor tool to create a script basic file. A good text editor that numbers lines and color codes the language syntax makes code easier to read and shows typing or copy/paste mistakes in the code. ScriptBasic is close to Visual Basic for syntax highlighting. After a ScriptBasic program is created, save the file using the extension ***.sb***. The filename needs to be *filename****.sb*** for the DXM Controller to recognize the file as a ScriptBasic file.

```
193
194  'Get GPS satellite info, PRN, Elevation, Azmuth, Signal Strength.
195  FUNCTION GetGPS_SatelliteInfo
196    LOCAL x, y, NumOfSatellites, SatInfo
197    NumOfSatellites = GETREG(GPS_NumSatellites_reg, ModSID, HoldingReg)
198
199    IF NumOfSatellites > 12 THEN
200      NumOfSatellites = 0
201    END IF
202    PRINT "\r\nSatellite Info: Number of Satellites being tracked :",NumOfSatellites," \r\n"
203    WrErr = SETREG (SatDataStart, NumOfSatellites, LocalReg, HoldingReg)
204
205    FOR x = 1 to NumOfSatellites
206      RdErr = MULTIGET(Sat_PRN_reg[x,1], 8, ModSID, HoldingReg)
207      SatInfo[x,1] = MBREGIN(0)
208      SatInfo[x,2] = MBREGIN(2)
209      SatInfo[x,3] = MBREGIN(4)
210      SatInfo[x,4] = MBREGIN(6)
211    NEXT x
212    FOR x = 1 to NumOfSatellites
213      PRINT "\t PRN:",SatInfo[x,1],"\t Elev:",SatInfo[x,2],"  \t Azim:",SatInfo[x,3],"\t SignalStr:",SatInfo[x,4],"\r\n"
214    NEXT x
```

Figure 1 A text editor with syntax highlighting

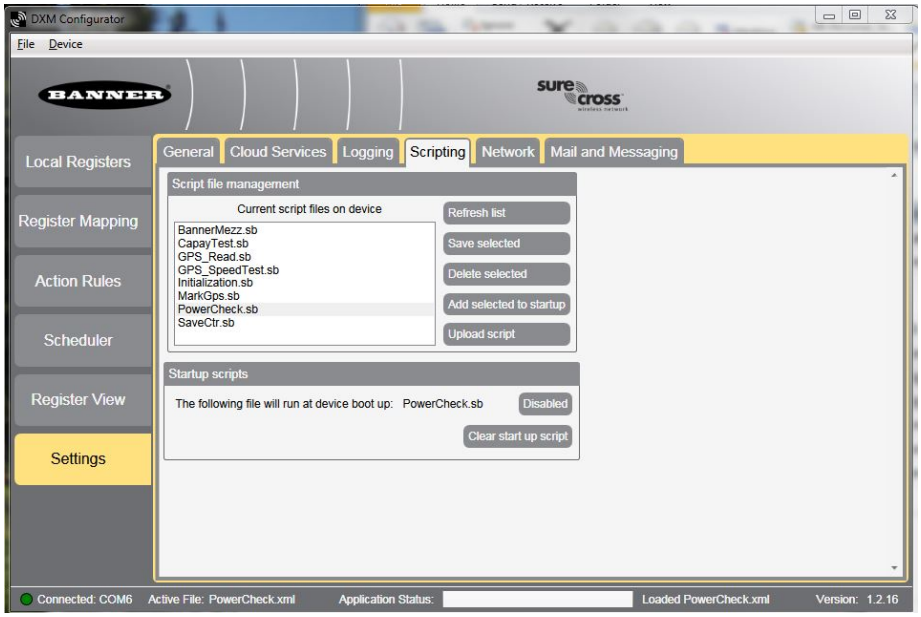
Uploading ScriptBasic Programs to the DXM Controller

The DXM Controller runs one ScriptBasic program after it boots up.

Use the DXM Configuration Tool to upload the ScriptBasic file (*sb) to the DXM Controller before you upload the XML configuration file. Define which ScriptBasic program to run, save the XML file, then load the XML configuration file onto the DXM Controller.

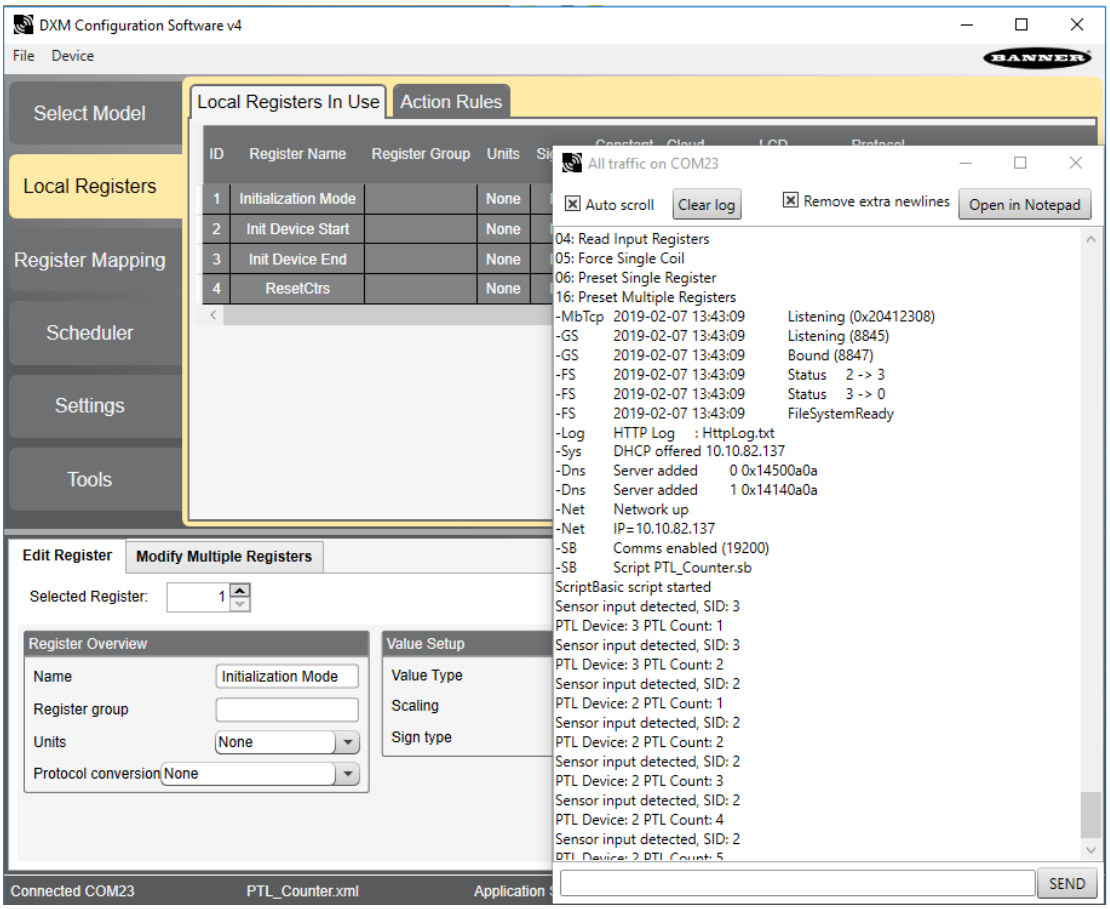
To upload the ScriptBasic program using the DXM Configuration Tool:

1. Launch the DXM Configuration Tool software.
2. Under the Device menu select the Connection Settings. Select the COMM port or enter the IP address if using TCP/IP
3. Go to **File** > **Load** and load a saved XML configuration file to add a ScriptBasic program.
4. Under the **Settings** > **Scripting** tab, click the **Upload Script** button to upload the ScriptBasic program (*.sb).
5. Click on the file name and then click on the **Add Selected to Startup** button. This stores the startup script name in the XML configuration file.
6. Save the XML configuration file and upload the XML file to the device.
7. After the device is restarted, the program outputs to the USB (default console port).



Debugging a ScriptBasic Program

The DXM Configuration Tool has a built-in terminal program that will show the console output; this includes error messages and PRINT statements from ScriptBasic. This is the primary method for debugging ScriptBasic programs. Under the ***Device*** menu select ***View Traffic on COMxx***; a pop-up box displays the console data traffic.



There are a few simple methods to help debug the ScriptBasic program.

- In the ScriptBasic program, write the ScriptBasic variables to local registers, then look at the local registers on the LCD display or look at them using the DXM Configuration Tool (REGISTER VIEW). To view a local register on the LCD display, set the LCD permissions flag to ‘read’ within the DXM Configuration Tool.
- Use PRINT statements in the ScriptBasic program to see variable values. All console output is sent to the USB port.
- Don’t forget to use the LEDs on the display, ScriptBasic can turn on or off all four LEDs to help know what’s going on in a script.

ScriptBasic Language

The following summary provides an overview of the ScriptBasic language implementation for the DXM Controller. ScriptBasic supports string variables, string operations, and some file operations. Custom commands have been created in the language to incorporate Modbus register access.

Examples are shown in all upper case. ScriptBasic is not case sensitive; keywords or variable names may be upper or lower case. For example, myvar and MYVAR are the same variable.

API Commands

ScriptBasic can access certain system variables through the common API interface that are typically accessed by a host system. The command syntax for accessing the real time counter is:

```
TimeVariable = API(102)
```

The available system variables are shown in the table below.

API command	Return Value	Description
API(6)	-	Invoke register push
API(8)	-	Clear HttpLog
API(16)	DXM IP Address	Get IP Address
API(17)	Subnet Mask	Get SubNet
API(18)	Gateway IP Address	Get default gateway
API(28)	Modbus Slave ID	Get Modbus Slave ID
API(102)	TimeStamp	Get RTC value (real time clock)
API(103)	Firmware Version	Get firmware version
API(112)	MAC	Get MAC address
API(113)	Model number	Get Model number
API(114)	Serial number	Get Serial number
API(200)	-	Reboot
API(212)	-	Cellular FOTA - Message to the cellular modem to look for updated firmware. Forced to look in FTP.SENSONIX.NET for delta file of new version. Running this command stops all operations on the controller and resets the controller after 5 minutes.

Assignment

Begin an assignment line with the variable you wish to assign. For example:

```
OurAverage = (YourVariable +MyVariable)/2
A = B +C
phi = sin (theAngle)
```

Comment Line

Remarks or comments are any line starting with REM or a single quote ('). Everything to the end of the line is ignored. Do not add remarks to the end of a valid line of code.

```
'Comment line in ScriptBasic
```

DO UNTIL Statement

This command loops the code between the lines DO UNTIL and LOOP until the expression following the keywords on the loop starting line becomes true.

```
DO UNTIL expression
...
commands to repeat
...
LOOP
```

The expression is evaluated when the loop starts and each time the loop restarts. The code between the DO UNTIL and LOOP lines is skipped when the expression evaluates to TRUE, even during the first loop.

This command is practically equivalent to the construct

```
WHILE NOT expression
...
commands to repeat
...
WEND
```

You can and should use the construct that creates more readable code.

DO WHILE Statement

This command implements a looping construct that loops the code between the lines DO WHILE and LOOP until the expression following the keywords on the loop starting line becomes false.

This command is the same as the command WHILE, but with a different syntax to be compatible with different BASIC implementations.

```
do while
...
loop
```

You can use the construct that creates more readable code.

END Statement

End of the program. Stops program execution.

Use END to signal the end of a program. Although you can use STOP and END interchangeably, the BASIC convention is to use END to note the end of the program and STOP to stop the program execution based on a condition specified inside the code.

File Operations

File operations are possible using the FILEOUT and FILEIN functions. Specifying messages using the UART are directed to the RS232 port of the microcontroller. E-mail or SMS messages can be created using the FILEOUT operation. Use TCP Server, TCP Client or UDP for Ethernet communication. For creating/appending files on the micro SD card, the target content written to one of five default file names on the micro SD card. Reading the SD card is currently not supported.

FILEOUT function to Write Output

```
Result = FILEOUT(FileIndex, Length, Flags, ContentToWrite)
   FileIndex is the target, indicated by values 1-14
          1 = UART
```

```
            2 = E-mail message (FILEOUT only)
            3 = SMS (FILEOUT only)
            4 = TCP Server
            5 = TCP Client
            6 = UPD
            10 = File 1, SbFile1.dat
            11 = File 2, SbFile2.dat
            12 = File 3, SbFile3.dat
            13 = File 4, SbFile4.dat
            14 = File 5, SbFile5.dat
      Length is the character length of the content to write
            0 = auto detect length
            xx = character length
      Flags define the file operation
            0 = Append file
            1 = Overwrite file, if it exists
      ContentToWrite is the content to write to the UART or file
```

Result = 0 = successful, non-zero indicates a problem with the parameters supplied.

FILEOUT Examples:

Writing data to the RS232 UART; FILEOUT(FileIndex, Length, Flags, ContentToWrite)
Result = FILEOUT(1,0,0,"This string is sent out the RS232 port")

Sending an E-mail; FILEOUT(FileIndex, OptionalFilename, 0, E-mailAdrs, Message)
Result = FILEOUT(2, SbFile10.dat, 0, MyEmail@host.com, "Sending logfile")

Sending an SMS; FILEOUT(FileIndex, 0, 0, Phone#, Message)
Result = FILEOUT(3,0,0,"1112223333", "This is my text message")

Writing data to the RS232 UART
Result = FILEOUT(1,0,0,"This string is sent out the RS232 port")

Writing data to Ethernet; FILEOUT(FileIndex, Length, 0, ContentToWrite)
Result = FILEOUT(4,0,0,"This string is sent out Ethernet")

Append data to SbFile1.dat; FILEOUT(FileIndex, Length, Flags, ContentToWrite)
Result = FILEOUT(10,0,0,"This is the data put into the file")

FILEIN to Read Data

To read content use the FILEIN function

```
ReadData = FILEIN(FileIndex, MaxLen)
   FileIndex is the target, indicated by values 1-14
         1 = UART
         4 = TCP Server
         5 = TCP Client
         6 = UPD
   MaxLen is the maximum character length to read
```

ReadData is the string read data

FILEIN Examples

Read data from the UART

```
Datain = FILEIN(1,0)
```

Read data from Ethernet (TCP server)

```
TCP_Data = FILEIN(4, 50)
```

Ethernet Data Packets with ScriptBasic

The DXM Controller can be programmed with ScriptBasic to send data as a client or server on the Ethernet port. The data is an ASCII string that is contained in an Ethernet TCP packet. The server port, client port and IP address are configured in the XML file using the DXM configuration tool. Default ports are 8845 for the server port, 8847 for the client port. (8844 is the DXM Push port)

Floating Point Conversion

Modbus protocol defines a holding register as 16-bits; however, there is a widely used de facto standard for reading and writing data wider than 16 bits. The most common are IEEE 754 floating point, and 32-bit integer. The data simply consists of two consecutive registers treated as a single value. Although the convention of register pairs is widely recognized, agreement on whether the high order or low order register should come first is not standardized.

On the DXM Controller the read maps automatically define two consecutive Modbus read registers into floating point when written to local Modbus registers 1001 and greater. In a Script Basic program Modbus registers are read one at a time and must be joined together and then converted using the function LTOF before storing the value into local Modbus registers 1001 and greater.

Example Script Basic fragment reading an external Modbus Slave floating point register.

```
'Read two consecutive registers that make up a floating point value.
   UpperFloat = GETREG(101,SID, MBtype)
   LowerFloatReg = GETREG(102,SID, MBtype)
'Put the registers back together in one 32-bit value.
   LongRegValue = (UpperFloat * 0x10000) + LowerFloatReg
'The current definition of LongRegValue is considered an integer value; define it
as a floating point
   FloatValue = LTOF(LongRegValue)
```

LTOF – Long integer To Floating point: This function takes a 32-bit integer in IEEE 754 format (sign, exponent, and mantissa) and defines it as a floating point variable.

FTOL – Floating point To Long integer: This function takes a 32-bit floating point value in integer format (31:0) and defines it as an integer.

FOR NEXT Statement

FOR NEXT statements implement a FOR loop. The variable *var* is assigned the value of the start expression exp_start. After each execution of the loop body, *var* is incremented or decremented by the value exp_step until *var* reaches the stop value exp_stop.

```
FOR var= exp_start TO exp_stop [ STEP exp_step]
...
commands to repeat
...
NEXT var
```

The STEP part of the command is optional. If this part is missing, the default increment value is 1. The loop body is not executed and the variable retains its original value when:

- the expression exp_start is larger than the expression exp_stop and exp_step is positive

- the expression exp_start is smaller than the expression exp_stop and exp_step is negative

When the loop is executed at least once, the variable is assigned the values one after the other. After the loop exits, the loop variable holds the last value assigned to the variable.

IF THEN ELSE Statement

There are two different ways to use this command: single line IF and multi-line IF. (IF/THEN and other keywords are shown in upper case in these examples only for clarity.)

A single line IF has the form

```
IF condition THEN command
```

There is no way to specify an ELSE part for the command in the single line version. If you need the ELSE command, use the multi-line IF.

The multi-line IF should not contain a command directly after THEN. Use the following format:

```
IF condition THEN
commands
ELSE
commands
END IF
```

Because the ELSE command is optional, the IF/THEN command can also have the format:

```
IF condition THEN
commands
END IF
```

The condition is any valid comparison or expression. Examples include:

```
if a > b then print "greater"
if a <> b then print "not equal"
if GETREG(3,1,0) then print "is enabled"
```

Conditional operators are:

=	Equal
<>	Not equal
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal

Labels and GOTO Statement

The statement GOTO is the most famous statement of all BASIC languages. Many program theorists say that you should never use GOTO. Even so, the statement GOTO is part of most programming languages and Script Basic is no exception.

Using the statement GOTO, you can alter the execution order of statements. GOTO statements use labels to identify program lines. The form of a GOTO statement is

```
GOTO label
```

Labels are local within functions and subroutines. You cannot jump into a subroutine or jump out of it. Labels begin at the start of a line, are the only thing on the line, and end with a colon.

```
GOTO mylabel
...
commands
...
mylabel:
...
```

Use of GOTO is usually discouraged and is against structural programming. Before using the GOTO statement (except ON ERROR GOTO) think of a better solution that performs the same task without using GOTO.

Typical use of the GOTO statement includes jumping out of an error condition to the error handling code or jumping out of a loop on a specific condition.

Logic Operators

The following is a summary of logic operators recognized in SB:

AND	Logical AND - bitwise for values, or logical in "if" statements
NOT	Logical NOT - bitwise for values, or logical in "if" statements
OR	Logical OR - bitwise for values, or logical in "if" statements
XOR	Logical Exclusive OR - bitwise for values

Use parenthesis to establish precedence as necessary.

Math Functions and Operators

The following is a summary of math operators and functions recognized in SB:

+	Addition
-	Subtraction
*	Multiplication
/	Division
\	Integer Division
%	Modulus
^	a^b produces *a* raised to the power of *b*
ABS(n)	Returns absolute value of 'n'
ACOS(n)	Calculates arc cosine of 'n'
ASIN(n)	Calculates arc sine of 'n'
COS(n)	Calculates cosine of 'n'
FIX(n)	Returns integral part of argument, rounding toward zero, i.e., int(-3.3) = -3
FRAC(n)	Returns the fractional part of the argument

INT(n)	Returns integral part of argument, rounding down, i.e., int(-3.3) = -4
LOG(n)	Produces the natural logarithm of n
LOG10(n)	Products the logarithm of n
PI	Produces the constant pi
POW(n)	Produces 10 raised to the power of n
RND	Provides a random number (use RANDOMIZE(n) to seed random number generator)
SIN(n)	Calculates sine of 'n'
SQR(n)	Calculates square root of 'n'

Note: 'n' is in radians. Calculate radians from degrees using this equation: r = ((Degrees ÷ 180) × PI)

Numbers

ScriptBasic supports to types of numbers, integer numbers and real numbers.

Integer numbers can be used to represent integral values, while real numbers can be used to represent numbs that have fractional part or are too large to store as integer. Integer numbers are stored in a memory location of size equivalent to a *long* of the programming language C. Real numbers are stored internally as C double.

Number constants can be used in the basic program in the usual format. Integer numbers are represented in either decimal or hexadecimal format. Decimal numbers only contain digits. Hexadecimal numbers start with the characters *0x* or *0X* and are followed by hexadecimal digits. The format that many basic implementations follow using the *&H* characters to start a hex number is also allowed. When a number contains a # character inside, like 2#110110 then the number preceding the # is the RADIX of the number and the characters following the # is the number in the given radix. The following numbers are valid integer constants in ScriptBasic:

- 123
- 0xFF , hex 0xff equals 255 decimal
- 0x255, equals 597 decimal
- 16#0123AB is hexadecimal 0123AB or 1,193,131 decimal
- &H52 equals 82 decimal

Real number constants can only be decimal and may contain a fractional or exponential part.

- 3.14
- 4.67E+4

Print Statement

You may use a Print statement to display output on the console port. An example of a print statement might be:

```
print "Register #5 is ", GETREG(5, 1, 0), "\n\r"
```

Random numbers

The RND() command returns a random number.

The RANDOMIZE() command seeds the random number generator. If the command is presented without argument, the random number generator is seeded with the actual time. If an argument is provided, the random number generator is seeded with the provided argument.

Register Access

You may read any local register or remote register using the GETREG function and write them using the SETREG function.

```
MyData = GETREG (22, SID, MBType)
MyData = MyData * 2.5
WrErr = SETREG (24, MyData, SID, MBType)
```

SID is the Modbus Slave ID of the device to read or write the register data. A Modbus slave ID of 199 refers to the local on-board registers. A Slave ID between 0 and 198 refers to an external Modbus slave device on the RS-485 device bus.

DXM Modbus Slave IDs	
0-198	External Modbus slave devices
199	Internal Local Registers
200	I/O board registers
201	Display board registers

The MBType parameter defines the Modbus Register Type for the command to access. Codes 4 and 5 are not used for write operations.

0 = Holding Register (all DXM local registers are holding registers)
3 = coil
4 = input
5 = input register
6 = single coil
7 = single register

This example uses the GETREG function to read Modbus register 22 from slave ID ‘SID’ into the variable MyData, multiplies it by 2.5, then using the SETREG function writes the value back to Modbus slave ID ‘SID’ register 24.

A GETREG function returns the data when successful or one of the error codes listed below if a failure occurs. A SETREG function when successful returns a 0; failure returns a one of the error codes listed below.

Only GETREG or SETREG functions to remote devices are subject to error conditions. GETREG or SETREG functions to internal local registers cannot create an error and pass 32-bits values instead of the Modbus convention of 16-bit registers.

The list of possible error codes are defined below.

Register Access Errors

Return Value	Register Access Errors
70001	API not available
80001	Script timeout
80003	Queue full

80004	API parameter issue
80005	Not authorized
80006	General failure
80007	Out of memory
80008	Unsupported function
80009	Syntax error
80010	Bad command
80011	General process error
80012	Result process error
80021	Transport failure
80022	Transport timeout

Multiple Register Access

Use the following functions to efficiently read or write multiple sequential registers. The result of these functions is a single Modbus transaction.

The input/output arrays are capped at 40 registers per transaction, so the 'ArrayIndex' values shown below can range from 0-39. The 'ArrayIndex' of 0 is always used as the first register to read or write.

Write Multiple Registers: The values must first be loaded into an array with the function MBREGOUT and then sent using the function MULTISET. Results for both functions will be zero if successful. Error codes for MULTISET are defined in the **Register Access Errors** table. Error codes for MBREGOUT are 1 for invalid ArrayIndex specified.

```
Result = MBREGOUT (ArrayIndex, Value)
Result = MULTISET (StartRegister, RegisterCount, SID, MBtye)
```

Example: Write remote Slave ID 5 registers 10 and 11 with values 50 and 51 (load values into array index 0 and 1)

```
Result = MBREGOUT (0, 50)
Result = MBREGOUT (1, 51)
Result = MULTISET (10, 2, 5, 0)
```

Read Multiple Registers: Commands MBREGIN and MULTIGET are used. The result codes for both are the same as the multiple write commands. In the case of the MULTIGET result code, it will be the first register value read (array index 0).

```
FirstRegisterValue     = MULTIGET (StartRegister, RegisterCount, SID, MBtye)
RegisterValue          = MBREGIN (ArrayIndex )
```

Example: to read back the values from the write registers example above:

```
RegisterValue   = MULTIGET (10, 2, 5, 0)
Register10Value        = MBREGIN (0)
Register11Value        = MBREGIN (1)
```

Strings

The simplest form of a string is a string constant in a source file. Enclose the string in double quotes.

```
PRINT "Hello World\n"
```

At the end of the string there is a special character denoted by two characters. A backslash followed by certain characters have special meaning. The special characters that ScriptBasic handles are:

- \t is converted to a tab character.
- \n is converted to a new line character
- \r is converted to a carriage return character
- \" is converted to a double quote character. This is a way to include a double quote into a string.
- \0-9 is converted to ASCII code

All other characters remain the same after a backslash.

String Functions & Operators

String concatenate operator, **&**

```
StringTotal = "string A" & "string B"
```

ASC – returns the ASCII code of the first character in the argument string.

```
ASC("soup")  = returns 115, ASCII value for 's'
```

CHR – returns the character for the ASCII number in the argument.

```
CHR(35)  = returns #
```

INSTR – This function can be used to search for a sub-string within a string. The first argument is the string to search. The second argument is the string to find within the first argument. The third argument is optional and is the starting index of where to begin the search. If no third argument is present the search will be begin with the first character. The function returns the position of where the sub-string can be found in the first string. Not found will return a value of undef.

```
INSTR("abcdefghijk", "fg") will result in 6 as the return value.
```

INSTRREV – This function is the same as INSTR but will begin the search from the end of the string.

```
INSTRREV("abcdefghijklmnopqrstuvwxyz", "xyz") will return a value of 24.
```

LCASE – returns the argument string in all lower case characters

```
LCASE("My String") – returns "my string"
```

UCASE – returns the argument string in all upper case characters

```
UCASE("My String") – returns "MY STRING"
```

LEFT – Creates the left of a string. The first argument is the string, the second argument is the number of characters to be put in the result, starting from the left.

```
LEFT("abcdefghi", 3)  returns a string of "abc"
```

RIGHT – Creates the right of a string. The first argument is the string, the second argument is the number of characters to be put in the result, starting from the right.

```
RIGHT("abcdefghi", 3)
```
returns a string of "ghi"

MID – Creates a sub-string from a string. The first argument is the string, the second argument is the starting position and the third argument is the number of characters.

```
MID("abcdefghijklmnopqrstuvwxyz", 10,3)
```
returns the string "jkl"

LEN – This function interprets its argument as a string and returns the length of the string.

```
LEN("my string is short")
```

returns a value of 18

LTRIM - Removes the space(s) from the left of the string

```
LTRIM("          123456")
```

returns "123456" as the string.

RTRIM – Remove the space(s) from the right of the string

```
RTRIM("123456          ")
```

returns "123456" as the string.

TRIM – Removes the space(s) from the left and right of the string.

```
TRIM("       abcdef       ")
```

returns "abcdef" as the string.

REPLACE – This function replace one or more occurrences of a sub-string in a string. The first argument is the base string, the second is the search string, and the third is the replacement string. The fourth and fifth arguments are optional. The fourth argument is the number of instances to replace and the fifth argument may specify a starting position.

```
REPLACE("abc abc abc abc", "b", "x", 2, 4)
```

replaces two instances of 'b' with 'x' starting at the 4th index. The resulting string is "abc axc axc abc"

SPACE – This function returns a string of N spaces.

```
SPACE(10)
```

returns a string of 10 spaces.

SPLIT – Takes a string and splits it into variables using a second string as a delimiter.

```
SPLIT "abcdef, ghi, jkl, mno" BY "," TO cmd, func, arg1, arg2
```

result is four varaiables, cmd = "abcdef", func – "ghi", arg1= "jkl", arg2 = "mno"

SPLITA – Similar to SPLIT but the result is stored in an array.

```
SPLITA "CMD0001 1234 2345 456" BY " " TO MyArray
```

result is MyArray [0]= CMD0001, MyArray [1] = "1234", MyArray [2] = "2345", MyArray[3] = "456"

STRREVERSE – This function takes a string input and reverses the order of the entire string.

```
STRREVERSE("abcdefghi")
```

The return string is "ihgfedcba"

Pattern Matching

LIKE – The operator LIKE compares a string to a pattern. If the pattern matches the string, the result of the operator is true (-1) otherwise the result is false (0) The pattern may contain normal characters, wild card characters and joker characters. The normal characters match themselves. The wild card characters match one or more characters from the set they are for. The joker characters match one character from the set they stand for.

```
PRINT "0123456789abcdefghi" LIKE "*abc*", "\n"
PRINT "0123456789abcdefghi" LIKE "0123*", "\n"
PRINT "0123456789abcdefghi" LIKE "abc" , "\n"
```

The three code lines will print out:

-1
-1
0

The wild card character * matches a list of characters of any code. The joker character ? matches a single character of any code. The wild card character is the most general wild card because it matches one or more of any character. There are other wild card characters. The character # matches one or more digits, $ matches one or more alphanumeric characters and the @ matches one or more alpha characters.

```
* -  all characters
# - 0123456789
$ - 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
@ - abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

A space in the pattern matches one or more spaces, but the space is not a regular wild card character.

We can match a string to a pattern, but that is little use, unless we can tell what substring the joker or wildcard characters matched. The function joker is used to indicate matches for the joker or wildcard characters. The joker function uses an integer argument starting from 1 and the result is the substring that the last pattern matching operator found to match the joker or wild card character. The JOKER index of 1 will reference the first special character in the pattern field, the next special character in the pattern field will use index 2.

For example:

```
Z = "12*24" LIKE "#*#"
PRINT  joker(1),"  ", joker(3)
```

The output will be 1 24. To specifically look for a wild card character use the ~ (tilde character)

```
Z = "12*24" LIKE "#~*#"
PRINT joker(1),"  " joker(3)
```

The output will be 12 24.

Advanced Pattern Matching

The rules for wild card characters and the joker character can be set to alter the set of characters that a joker or wild card character matches. There are 13 characters that can play joker or wild card character roles in pattern matching

```
*  #     $     @     ?     &     %     !     +     /     |     <     >
```

When the program starts only the first five characters have special meaning the others are normal characters. To change the roles of a character the program has to execute a SET JOKER or SET WILD command.

```
SET JOKER "&" TO "012345"
SET WILD "+" TO "abcdefgh"
```

The first character should be the joker or wild card character the second string should contain all the characters that the joker or wild card character matches.

The command SET JOKER alters the behavior of the character to be a joker character matching a single character in the compare string. The SET WILD command alters the behavior of the character to be a wild card character matching one or more characters in the compare string.

If a character is currently a joker or wild card character use the SET NO JOKER or SET NO WILD to set the character to a normal character.

```
SET NO JOKER "&"
SET NO WILD "*"
```

Time Commands

The time commands within ScriptBasic are used to create timing based programming. Two commands exist to show different resolutions of time since boot. The NOW command is in second increments, the TICKS command is in 10ms increments, both show the time since the last boot time of the DXM controller. The SLEEP command is a delay mechanism that pauses program execution for a period of seconds. Local registers defined as Timers (or counters) can be created from any local registers to provide a method to manage time, without the worry of roll overs.

Use the DXM Configuration Tool to define any local register to be a timer register that will increment every 100ms or 1 second and can be reset with the SETREG function.

NOW Command

The NOW command provides the number of seconds since bootup. Saving the value of NOW in a variable and then calculating the difference to current time allows measurement of real time. The NOW command can only provide the seconds since the last boot time; it cannot assign a new value to NOW. The NOW count is created from a 1ms counter divided by 1000. The 1 ms counter rolls over at (2^32) or 4,294,967,296. So the NOW value rolls over to zero at 4294967 or about 50 days. The user must manage the roll over to correctly deal with timing functions within ScriptBasic.

Get the current time since boot

```
TimeStamp = NOW
```

TICKS Command

TICKS Command is similar to the NOW command except it provides the number of 10 ms counts since the last boot-up. The TICKS command can only provide the number of 10 ms counts since the last boot time; it cannot assign a new value to TICKS. The TICKS count is created from a 1 ms counter divided by 10. The 1 ms counter rolls over at (2^32) or 4,294,967,296. So the TICKS value rolls over to zero at 429496729 or about 50 days.

Get the current time since boot

```
CountTime10ms = TICKS
```

SLEEP Command

The function SLEEP suspends program execution for some number of seconds. Examples:

```
SLEEP (5)
SLEEP (0.25)
```

In the first example the program execution is paused for 5 seconds. The second example pauses for a quarter of a second.

Timer Configuration

Any local register can be defined as counter register that just counts in 0.1 seconds or 1 second increments. The register can be reset to zero at any time which makes it ideal for creating a timer mechanism to manage time periods. See the DXM Configuration tool for defining local registers as counters.

Example:

With local register 10 defined as a timer the ScriptBasic program waits until 15 minutes has elapsed then set local register 15 to one for a couple of seconds then set it to zero.

```
' Basic timer usage.
' local register 10 must be defined as a counter register
'
Timer_reg= 10
Output_reg      = 15
' MaxCount is 15 minutes, 900 seconds or 9000 .1 second increments
MaxCount  = 9000
LocalReg        = 199
HoldingReg      = 0
'
'Reset timer
FUNCTION ResetTimer(TReg)
   SETREG(TReg, 0, LocalReg, HoldingReg)
END FUNCTION
'
FUNCTION FlashLight
   SETREG (Output_reg, 1, LocalReg, HoldingReg)
   SLEEP(2)
   SETREG (Output_reg, 0, LocalReg, HoldingReg)
END FUNCTION
'
' Initialize count
ResetTimer(Timer_reg)
SETREG(Output_reg, 0, LocalReg, HoldingReg)

WHILE (1)
   IF GETREG(Timer_reg, LocalReg, HoldingReg) >= MaxCount THEN
          '  Timer has reached the limit, reset it to start over...
         ResetTimer(Timer_reg)
         FlashLight
   END IF

   'The rest of the main control loop....
   ' hopefully doing something useful...
WEND
```

User Functions

Use the FUNCTION command to define a function. A function is a piece of code called by the BASIC program from the main part or from a function or subroutine.

```
FUNCTION fun(a,b,c)
...
fun = returnvalue
...
END FUNCTION
```

The end of the function is defined by the line containing the keywords END FUNCTION. This function would be called by the line x=fun(a,b,c) and the result would be placed in 'x'.

The SUB command should be used to define a subroutine. A subroutine is a piece of code called by the BASIC program from the main part or from a function or subroutine.

```
SUB sub(a,b,c)
...
END SUB
```

The end of the subroutine is defined by the line containing the keywords END SUB.

Note that functions and subroutines are not really different in ScriptBasic. ScriptBasic allows you to return a value from a subroutine and to call a function using the command CALL. It is just a convention to have separate SUB and FUNCTION declarations. You would call this subroutine with the line CALL sub.

Variables

Variables are core entities of ScriptBasic and store string, real, or integer values. Variable names start with alpha characters, underscore, dollar sign or colon and from the second character they may contain digit characters in addition to all these characters. The last character of a name should not be a colon.

WHILE Statement

Implements the 'while' loop as it is usually done in most BASIC implementations. The loop starts with the command WHILE and finishes with the line containing the keyword WEND. The keyword WHILE is followed by an expression and the loop is executes until the expression is no longer true.

```
while expression
...
commands to repeat
...
wend
```

The expression is evaluated when the loop starts and each time the loop restarts. The code between the WHILE and WEND lines is skipped when the expression evaluates to FALSE, even during the first loop.

If a condition requires exiting the loop from within the loop, use the GOTO command.

Variables and Arrays

Variables may be any name that starts with a letter and, after the first letter, may also contain digits, underscore, and dollar sign (for old style denotation of string variables). Variables are automatically typed as integer, double (floating point), or string according to the context in which they are used and may dynamically change type during program execution.

Arrays are created automatically as soon as you use subscripts. Uninitialized elements of the arrays return "undef". Array subscripts use square brackets (not parenthesis that are easily confused with function calls). For example:

```
a[1] = 45.9
a[2] = 99.8
a[3] = "something else"
a[4,12] = 10
```

These are all valid elements of the same array. Normally the index values must be integer. However, SB also supports associative arrays. Associative arrays use "curly" brackets. For example:

```
animal{"cat"} = "Garfield"
animal{"dog"} = "Snoopy"
animal{"tiger"} = "Tony"
```

UBOUND – Use to determine the upper occupied index of an array.

LBOUND – Use to determine the lowest occupied index of an array

ScriptBasic Examples

A simple program structure to follow is to declare variables first, then functions and then the main program body. Add comments and indentation to make the code easier to understand.

```
'Comments
Global variables
Functions
   Local variables
   commands, assignments, loops, etc.
Main program
commands, assignments, loops, etc.
```

Example of a function called MEDIANFILTER. Passes a variable list 'alist' and returns the middle value once it is sorted.

```
'Median filter, sorts a list of numbers then returns the middle value of the list
 FUNCTION MEDIANFILTER(alist)
   LOCAL NotSorted, Midpoint
   LOCAL x, k
   NotSorted = 1
   WHILE NotSorted <> 0
          NotSorted = 0
          FOR x = 1 to (ubound(alist)-1)
                IF (alist[x] > alist[x+1]) THEN
                      k = alist[x]
                      alist[x] = alist[x+1]
                      alist[x+1] = k
                      NotSorted = 1
                END IF
          NEXT x
   WEND
   Midpoint     = (ubound(alist)/2)
   MEDIANFILTER =  alist[Midpoint]
 END FUNCTION
```

Example of a function using GETREG to read local register values.

```
'Combine two 16-bit register reads into one 32-bit value. Check for negative
values
 FUNCTION COMBINEINT(upper_reg, lower_reg)
   LOCAL a, b, c, NegFlag, LocalReg, MBtype
   LocalReg     = 199
   MBtype       = 0
   NegFlag      = 0
   a = GETREG(upper_reg,LocalReg,MBtype)
   IF (a AND 0x8000) THEN
          'PRINT "Negative \r\n"
          a = (a XOR 0xFFFF)
          NegFlag = 1
   END IF
   b = GETREG(lower_reg,LocalReg,MBtype)
   IF (NegFlag = 1) THEN
          b = (b XOR 0xFFFF)
   END IF
   c = (a * 0x10000) + b
   IF (NegFlag = 1) THEN
          c = c + 1
   END IF
   CombineInt = c
   'PRINT c, "\r\n"
```

```
END FUNCTION
```

Example of a state machine function that uses the NOW command to define when to run operations.

```
 FUNCTION StateMachine
'State machine definitions for the periodic reading of temp/humidity
' TH_State      = 0  current state of the state machine
' TH_Idle= 0    initial state
' TH_Wait = 1     wait time between samples
' TH_Sample            = 2     get samples from remote sensor
' TH_Error             = 3     error state - unknown condition

  LOCAL StartState, WrErr, MBtype
  WrErr                = 0
  MBtype       = 0
  StartState   = TH_State
  WrErr = SETREG (SM_reg, TH_State, LocalRegSID, MBtype)

  IF TH_State = TH_Idle THEN
        StartTime     = NOW
        TH_State      = TH_Wait
  ELSEIF TH_State = TH_Wait THEN
        IF NOW >= (StartTime + WaitTime) THEN
              TH_State = TH_Sample
        ELSE
              TH_State = TH_Wait
        END IF
  ELSEIF TH_State = TH_Sample THEN
        GetTempHumidityData
        TH_State = TH_Idle
  ELSE
        TH_State = TH_Error
  END IF
  IF StartState <> TH_State THEN
        PRINT "\r\n Time ",NOW,"  SM Started-> ",THState[StartState],"  End->
",THState[TH_State]," \r\n"
  END IF

END FUNCTION
```

Example of an entire program with a state machine to read the temp/humidity sensor and turn on/off LEDs to indicate rising/falling temperatures.

```
'Local Register definitions
Humidity_reg                  =     1
TempC_reg             =     2
SM_reg                        =     5

'Modbus Registers on the Temp Humidity sensor
SensorHumidity_reg            =     1
SensorTempC_reg       =     2
SensorTempF_reg       =     3

'Display LED's
ScriptRunnning_LED1_reg       =     1102
TempGoingUp_LED2_reg  =     1103
TempGoingDown_LED3_reg =    1104
CommsError_LED4_reg           =     1105

 'Global Variables internal to ScriptBasic
TempC                 = 0
Humidity              = 0
```

```
LastValueTempC         = 0
LastValueHumidity            = 0

StartTime              = 0
WaitTime               = 15

WrErr                        = 0

'Set Modbus type to holding registers
MBtype                       = 0


 'State machine definitions for the periodic reading of temp/humidity
 TH_State              = 0
 TH_Idle               = 0
 TH_Wait               = 1
 TH_Sample                    = 2
 TH_Error              = 3

 'Make an array of state names to make it easier to read
 THState[0]     = "Temp/Humd Idle"
 THState[1]     = "Temp/Humd Wait"
 THState[2]     = "Temp/Humd Sample"
 THState[3]     = "Temp/Humd Error"

 'Define Modbus Slave ID's for different devices
 LocalRegSID                  = 199
 TH_SID                = 1
 IoBoardSID                   = 200
 DisplaySID                   = 201

 'Function to read the T/H sensor
 FUNCTION GetTempHumidityData
   LastValueTempC      = TempC
   LastValueHumidity   = Humidity
   Humidity            =GETREG(SensorHumidity_reg, TH_SID, MBtype)
   TempC         = GETREG(SensorTempC_reg, TH_SID, MBtype)
   IF Humidity > 65535 or TempC > 65535 THEN
          PRINT "Read Error - humidity / temp reading...", Humidity,"  ",TempC,
"\n\r"
   END IF
WrErr = SETREG (Humidity_reg, Humidity, LocalRegSID, MBtype)
WrErr = SETREG (TempC_reg, TempC, LocalRegSID , MBtype)

 END FUNCTION

  FUNCTION StateMachine
 'State machine definitions for the periodic reading of temp/humidity
' TH_State      = 0  current state of the state machine
' TH_Idle       = 0  initial state
' TH_Wait       = 1  wait time between samples
' TH_Sample     = 2  get samples from remote sensor
' TH_Error      = 3 error state - unknown condition

   LOCAL StartState
   StartState = TH_State
   WrErr = SETREG (SM_reg, TH_State, LocalRegSID, MBtype)

   IF TH_State = TH_Idle THEN
          StartTime = NOW
          TH_State = TH_Wait
```

```
  ELSEIF TH_State = TH_Wait THEN
         IF NOW >= (StartTime + WaitTime) THEN
                TH_State = TH_Sample
         ELSE
                TH_State = TH_Wait
         END IF
  ELSEIF TH_State = TH_Sample THEN
         GetTempHumidityData
         TH_State = TH_Idle
  ELSE
         TH_State = TH_Error
  END IF
  IF StartState <> TH_State THEN
         PRINT "\r\n Time ",NOW,"  SM Started-> ",THState[StartState],"  End->
",THState[TH_State]," \r\n"
  END IF

 END FUNCTION

 FUNCTION LED_driver
  IF LastValueTempC < TempC THEN
         WrErr = SETREG (TempGoingUp_LED2_reg,1,DisplaySID, MBtype)
  ELSE
         WrErr = SETREG (TempGoingUp_LED2_reg,0,DisplaySID, MBtype)
  END IF
  IF LastValueTempC > TempC THEN
         WrErr = SETREG (TempGoingDown_LED3_reg,1,DisplaySID, MBtype)
  ELSE
         WrErr = SETREG (TempGoingDown_LED3_reg,0,DisplaySID, MBtype)
  END IF
  IF (Humidity > 65535 ) OR (TempC > 65535) THEN
         WrErr = SETREG (CommsError_LED4_reg,1,DisplaySID, MBtype)
  ELSE
         WrErr = SETREG (CommsError_LED4_reg,0,DisplaySID, MBtype)
  END IF

  IF GETREG(ScriptRunnning_LED1_reg, DisplaySID, MBtype) THEN
         WrErr = SETREG (ScriptRunnning_LED1_reg,0,DisplaySID, MBtype)
  ELSE
         WrErr = SETREG (ScriptRunnning_LED1_reg,1,DisplaySID, MBtype)
  END IF

 END FUNCTION

'Main program loop
 BEGIN:
  PRINT "Script Starting\r\n"
  ITERATE:
         'PRINT "\r\n Time = ",NOW," \r\n"
         StateMachine
         LED_driver
         Sleep(1)
  GOTO ITERATE
 END
```

Configuring DXM I/O Using ScriptBasic

The universal inputs on a DXM Controller are configurable inputs that are useful and flexible, but more flexibility can create complexity that is difficult to manage. ScriptBasic can help manage the options into a

single start-up script that can be used to clone multiple DXM Controllers or just help collect all the parameter changes for an application.

The attached ScriptBasic file contains the basic building blocks to create a program that will set up parameters on the DXM I/O base board. This won't be a perfect script for any particular application but should provide the basic understanding of DXM configuration using ScriptBasic.

The first 50 or so lines of the program are defining variables with a few added comments. The next 90 lines are functions that group certain processes together. The functions are:

- **SetFactoryDefaults**. This function writes a couple of Modbus registers in the I/O board that force the I/O board to restore the factory default settings for all the I/O. This erases any parameter settings that may have been done. Most likely this function will be not be called by the program during normal operation, but it is handy if you need it.
- **InitArrays**. This function puts the Modbus register addresses in arrays for each universal input. For example, all universal inputs have a parameter called **Input Type**. The Modbus address to access the Input Type for universal input 1 is stored in the variable *InputType_reg[1]*. The array index '[1]' is the universal input number 1 through 8. The Modbus register address for the input type on universal input 5 will be stored in *InputType_reg[5]*.
- **GetIOreg(regnum)**. This function does a Modbus read to the I/O board to get one piece of data. This function has the ability to retry a read command five times before it gives up. Normally reading or writing registers within the controller does not fail, but because I have the SetFactoryDefaults function in the script, the I/O board resets and causes read failures until the board finishes booting.
- **ReadAllParams**. This function reads the parameters, saves the values in arrays, and prints them out to the console.

The main body of the program starts with the label 'BEGIN:'. The program starts and finishes with turning on an LED on the display to show when the script is running. First, the **InitArrays** and **ReadAllParams** functions are called to initialize the process. Then Modbus register writes are performed to set the actual parameters to adjust. This example is simple -- only four items are written -- but the list could be hundreds of parameters long.

```
' Initialization script for I/O board
' Resets the I/O board to factory defaults then prints out the parameter settings.
'
'I/O board Modbus slave ID = 200
IO_SID = 200
DISPLAY_SID = 201
'
' Define all the I/O board Modbus registers as Holding registers.
MBType = 0
'
'Display LED to indicate the script running.
LED1_reg = 1102
'
' Initialization register on the I/O board for restoring factory defaults.
IO_Reset_reg = 4151
IO_FactoryRestore_reg = 4152
'
' Universal input type codes. Each universal input has a factory default of '8'
Type_NPN = 0
Type_PNP = 1
Type_Ma = 2
Type_Vdc = 3
Type_Thermistor = 4
Type_Pot = 5
Type_Bridge = 7
```

```
Type_NPN_raw = 8
'
' The I/O board charging circuit default operation is battery backup; reg 6071 = 1
' Change to enable the charging circuit to a solar system by setting reg 6071 = 0
' Default = ChargingDefintion_reg = BatteryBackupEnabled
ChargingDefintion_reg = 6071
SolarChargerEnabled = 0
BatteryBackupEnabled = 1
'
' Enable a universal input as a synchronous counter. (defaults = disabled)
Disable_Rising = 0
Enable_Rising = 1
Disable_Falling = 0
Enable_Falling = 1
'
'Enable full scale, default 0, Modbus registers stores uAmps or mVolts
Enable_FullScale = 1
Disable_FullScale = 0
'
'Set temperature units to Celsius or Fahrenheit, default = Celsius
UnitsCelsius = 0
UnitsFahrenheit = 1
'
'WrError will save SETREG status information
WrError = 0
'*****************************************************************************
' Various functions created to group procedures together.
'
FUNCTION SetFactoryDefaults
'Restore I/O factory defaults on the I/O board
   Wr_Error = SETREG (IO_FactoryRestore_reg, 1, IO_SID, MBType)
   Wr_Error = SETREG (IO_Reset_reg, 1, IO_SID, MBType)
SLEEP(10)
END FUNCTION
'
FUNCTION InitArrays
' Initialize arrays for the data values.
   FOR x = 1 to 8
          EnableFullScale_data[x] = 0
          TempC_F_data[x] = 0
          InputType_data[x] = 0
          Threshold_data[x] = 0
          Hysteresis_data[x] = 0
          EnableRising_data[x] = 0
          EnableFalling_data[x] = 0
   NEXT x
' Define all the Modbus registers on the I/O board for the universal inputs 1
   EnableFullScale_reg[1] = 3303
   TempC_F_reg[1] = 3304
   InputType_reg[1] = 3306
   Threshold_reg[1] = 3308
   Hysteresis_reg[1] = 3309
   EnableRising_reg[1] = 4908
   EnableFalling_reg[1] = 4909
'
' Create the rest of the universal input register addresses, U2- U8, each input is
offset by 20 from the previous input.
   FOR x = 2 to 8
          EnableFullScale_reg[x] = EnableFullScale_reg[x-1] + 20
          TempC_F_reg[x] = TempC_F_reg[x-1] + 20
          InputType_reg[x] = InputType_reg[x-1] + 20
```

```
          Threshold_reg[x] = Threshold_reg[x-1] + 20
          Hysteresis_reg[x] = Hysteresis_reg[x-1] + 20
          EnableRising_reg[x] = EnableRising_reg[x-1] + 20
          EnableFalling_reg[x] = EnableFalling_reg[x-1] + 20
   NEXT x
' Print all the register numbers, remove the comments if you want the program to
print the register addresses.
   'FOR x = 1 to 8
          ' PRINT " Universal Input ",x," register numbers.\n\r"
          ' PRINT " EnableFullScale_reg = ",EnableFullScale_reg[x],"\n\r"
          ' PRINT " TempC_F_reg = ",TempC_F_reg[x],"\n\r"
          ' PRINT " InputType_reg = ",InputType_reg[x],"\n\r"
          ' PRINT " Threshold_reg = ",Threshold_reg[x],"\n\r"
          ' PRINT " Hysteresis_reg = ",Hysteresis_reg[x],"\n\r"
          ' PRINT " EnableRising_reg = ",EnableRising_reg[x],"\n\r"
          ' PRINT " EnableFalling_reg = ",EnableFalling_reg[x],"\n\r\n\r"
   'NEXT x
'
END FUNCTION
'
FUNCTION GetIOreg(regnum)
' Simple function to retry if an error occurs, waiting for the I/O board to reset.
   LOCAL x, retry, maxcnt, data
   retry = 1
   maxcnt = 5
   data = GETREG(regnum, IO_SID, MBType)
   WHILE (data > 65535) AND (retry <= maxcnt)
          data = GETREG(regnum, IO_SID, MBType)
          retry = retry + 1
          PRINT "-- read error -= retry ",retry,"\n\r"
   WEND
   GetIOreg = data
END FUNCTION
'
FUNCTION ReadAllParams
' Read all the current parameter values from the I/O board.
   LOCAL x
   FOR x = 1 to 8
          EnableFullScale_data[x] = GetIOreg(EnableFullScale_reg[x])
          TempC_F_data[x] = GetIOreg(TempC_F_reg[x])
          InputType_data[x] = GetIOreg(InputType_reg[x])
          Threshold_data[x] = GetIOreg(Threshold_reg[x])
          Hysteresis_data[x] = GetIOreg(Hysteresis_reg[x])
          EnableRising_data[x] = GetIOreg(EnableRising_reg[x])
          EnableFalling_data[x] = GetIOreg(EnableFalling_reg[x])
          ' Print out what was read from the I/O board.
          PRINT "Universal Input ",x," Parameters\n\r"
          PRINT " EnableFullScale_data = ",EnableFullScale_data[x],"\n\r"
          PRINT " TempC_F_data = ",TempC_F_data[x],"\n\r"
          PRINT " InputType_data = ",InputType_data[x],"\n\r"
          PRINT " Threshold_data = ",Threshold_data[x],"\n\r"
          PRINT " Hysteresis_data = ",Hysteresis_data[x],"\n\r"
          PRINT " EnableRising_data = ",EnableRising_data[x],"\n\r"
          PRINT " EnableFalling_data = ",EnableFalling_data[x],"\n\r\n\r"
   NEXT x
END FUNCTION
'
' The main body of the program.
BEGIN:
' Turn on display LED 1 to indicate we are starting script...
WrError = SETREG (LED1_reg, 1, DISPLAY_SID, MBType)
```

```
'Reset the I/O board back to factory defaults on the I/O, may want to comment out.
'SetFactoryDefaults
'Create arrays of register addresses and data
InitArrays
'Read the I/O board universal input parameters
ReadAllParams
'*******************************************************************************
****
*
'************ Use SETREG commands to set parameters on I/O board
*********************
'*******************************************************************************
****
*
'SETREG is the command for writing Modbus registers
'WrError = SETREG ('Register address', 'Register value', 'Modbus slave ID',
'Modbus
type')
'
'Set the charging algorithm to backup battery
WrError = SETREG (ChargingDefintion_reg, BatteryBackupEnabled, IO_SID, MBType)
'Set the input type to Potentiometer for universal input 1
WrError = SETREG (InputType_reg[1], Type_Pot, IO_SID, MBType)
'Set the input type to Current (mA) for universal input 3
WrError = SETREG (InputType_reg[3], Type_Ma, IO_SID, MBType)
'Set the temperature units on universal input 4 to Celsius
WrError = SETREG (TempC_F_reg[4], UnitsCelsius, IO_SID, MBType)
'
' Add more SETREG commands here...
'
' Turn off display LED 1 to indicate we are done...
WrError = SETREG (LED1_reg, 0,DISPLAY_SID,MBType)
END
```

Error Codes

Error Code	Description
0	SUCCESS
1	Not enough memory
2	Function cannot return a whole array
3	Division by zero or other calculation error
4	Argument to operator is undefined
5	The command or sub was called the wrong way
6	There are not enough arguments of the module function
7	The argument passed to a module function is not the needed type
8	The argument passed to a module function is out of the accepted range
9	The module experiences difficulties reading the file
10	The module experiences difficulties writing the file
11	The module experiences handling the file
12	There is a circular reference in memory
13	The module cannot be unloaded, because it was not loaded
14	Some modules were active and could not be unloaded
15	The module cannot be unloaded, because it is currently active
16	The requested module cannot be loaded
17	The requested function does not exist in the module
18	The module did not initialize correctly
19	The module was developed for a different version of ScriptBasic
20	File number is out of range, it should be between 1 and 512
21	The file number is already used
22	The file cannot be opened
23	The file is not opened
24	The lock type is invalid
25	The print command failed. The file may be locked by another process.
26	Directory cannot be created
27	The directory or file could not be deleted
28	Command is not implemented and no currently loaded extension module defined behavior for it
29	The character cannot be a joker or wild card character
30	The code tried to execute a resume while not being in error correction code
31	The directory name in open directory is invalid
32	Invalid option for directory open

Error Code	Description
33	The directory cannot be opened
34	The record length is invalid in the open statements (undefined, zero or negative)
35	The current directory cannot be retrieved for some reason
36	The directory name in chdir cannot be undef
37	Cannot change the current working directory to the desired directory
38	The command RETURN cannot be executed, because there is nowhere to return
39	The argument for the function address is invalid
40	The attribute value or symbol is invalid in the set file command
41	The user does not exist
42	The shown command is not supported on Win95 and Win98
43	Cannot change owner
44	The file name is invalid
45	Setting the create time of the file has failed
46	Setting the modify time of the file has failed
47	Setting the access time of the file has failed
48	The specified time format is invalid
49	The time is not valid, cannot be earlier than January 1, 1970. 00:00
50	Extension specific error: %s
51	The operation can be done on files only and not on sockets
52	The embedding application tried to start the code at an invalid location
53	Mandatory argument is missing
54	Subprocess did not finish within time limits
55	The module cannot be unloaded
56	The preprocessor said to abort program compilation or execution
57	The file is either corrupt or was written with a different version of sb
58	The compiled program contains no executable code
59	Code file cannot be saved
60	The interpreter file cannot be read
61	Bad syntax in include statement
62	Include file is not found
63	Too many includes, probably recursive include
64	Preprocessor \%s\" is not defined"
65	Preprocessor name is too long
66	Preprocessor \%s\" is not available"

Error Code	Description
67	Preprocessor \%s\" is invalid"
68	The file cannot be read
69	The file is empty or is not readable
70	The external preprocessor failed
71	The preprocessor executable is not configured
72	The preprocessor temporary directory is not configured
73	Symbol is too long
74	String is not terminated anywhere before end of file
75	String is not terminated anywhere before end of line
76	The number contains invalid radix. BASE#NNN numbers are available for 2<=BASE<=36 only
77	The BASE#NNN formatted number digit is out of range
78	Program counter points out of the code. The executed code is corrupt
79	Internal error or the cached code is corrupt
80	Internal error or the cached code is corrupt
81	Invalid op code is in the code table
82	The code exists, but it is not implemented
83	Internal error or the cached code is corrupt
84	Internal error or the cached code is corrupt
85	Internal error when releasing a variable it has no correct type
86	Internal error when releasing undefined variable, nonsense
87	Internal error when releasing variable reference found
88	Any non-classified internal error
89	Local variable referenced in global context or in a local context having no local variables
90	Internal error or the cached code is corrupt
91	Internal error or the cached code is corrupt
92	Internal error or the cached code is corrupt
93	Internal error or the cached code is corrupt
94	Internal error or the cached code is corrupt
95	Internal error or the cached code is corrupt
96	Internal error or the cached code is corrupt
97	The user function is undefined
98	Recursive function calls exceeded configuration limit
99	Program is running too long, probably infinite loop

Error Code	Description
100	Internal error in the syntax analyzer
101	A line label was defined more than once
102	A label \%s\ “was not defined during syntax analysis”
103	The user function was already defined
104	The user function \%s\" is used but is not defined"
105	There was no module started when an end module statement was found
106	A module was not closed at end of file
107	The name space grew too long
108	Variable and namespace does not fit into the buffer
109	Closing) is missing in expression
110	Opening (is missing in sub call
111	Closing] is missing following array indices
112	Closing } is missing following associative array indices
113	Built in function needs arguments
114	Built in function has too few arguments
115	Built in function has too many arguments
116	Erroneous nesting of constructs like IF/ELSIF/ELSE/ENDIF/REPEAT/LOOP/WHILE and so on
117	Nested construct remained unclosed when local end
118	Syntax error, no syntax definition matches the line
119	Fatal syntax error, no syntax definition matches the line
120	There are more than MAX_SAME_LABEL labels referencing the same line
121	Global variable was not declared
122	Compiler option \%s\" is not implemented"
123	Global variable \%s\" is redefined"
124	Variable \%s\" cannot be used as global and local in a sub"
125	The variable is declared as constant. Needs 'var' declaration
126	DupVar cannot duplicate array. This is an internal error
127	The format string contains $n where n is out of range
128	The provided buffer is too short
129	Invalid pattern contains no character after the ~ sign

Error Code	Description
130	The provided string array is too short
131	Operation failed
132	The provided buffer is too short
133	Port specific undefined error