

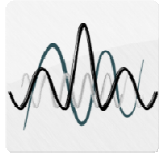
LANGUAGE  
REFERENCE



## [THE BLAZINGCORE SERIES]

WITH SUPERIOR NUMBER CRUNCHING ABILITIES AND PERIPHERAL HANDLING ON OUR CUSTOM EMBEDDED OS,  
RAPID PROTOTYPING IS NOW EASY... AND BLAZING FAST.

## Sonata — IDE —



Sonata is the accompaniment IDE for our 32bit Embedded OS, BlazingCore.

Sonata IDE is a **structured programming environment**, with support for global and local structures, subroutines and variables. Sonata features plenty of inbuilt library support for main core hardware configurations, and makes it easy for users to add-on high-level existing/new libraries for peripherals.

As of Version 1.0, Sonata IDE is the main programming environment of the BlazingCore OS for the 32bit PIC32MX695F512L Microprocessor by Microchip®.

The BlazingCore OS features built-in **libraries (OS Commands)** that **natively** handle control over I/O peripherals, external displays (e.g. OLED Displays/Graphic LCDs), CMOS Cameras (C3038), communication protocols (eg. UART/SPI/I2C/..), as well as external memory (DataFlash, FAT32/FAT16 MMC/SD-compatible Cards, Serial SRAM, SRAM).

### SYNTAX LANGUAGE

Sonata IDE provides the programming platform for the BCore Boards using Microsoft®'s Syntax Language.

At this point of writing, it currently supports the **VBm (Visual Basic Minor) language**, although **C#m language** support is underway.

“Minor” denotes that the language is streamlined to better suit the embedded platform.

### BCORE OS COMMANDS

The BCore OS Commands are native libraries that address the embedded side of the BCore Chip. They are separate from the VBm Syntax Language and are usually denoted by its syntax colour (dark blue), as opposed to the VBm syntax colour (blue). The OS Commands are used when dealing with the IO peripherals, system components, device communication protocols, and for performing bit, byte and word manipulations.

When custom control is required and not supported natively by the BCore OS nor amongst the many libraries available for common devices shipped with the IDE, you will find that modifying codes from the library similar to your device or software bit-banging is easily achieved.

# VBm LANGUAGE REFERENCE

## COMMENT/REMARK FORMAT

An apostrophe (') is used to denote a comment.

### Code:

```
'This is a remark
Dim S As String
S = " 'This is not a remark"
```

All characters to the right of the apostrophe are usually ignored by the compiler, unless the apostrophe is embedded inside a string literal.

## VARIABLE DECLARATION

```
Dim <Identifier> As <Data_Type>
```

All variables must be declared before they're used. Implicit declarations are not allowed.

For variables declared in module-level code:

```
[Public | Private | Dim] <variable> As <Data_Type>
```

Public variables are global and visible throughout the entire program. Private variables are visible only in the module in which they appear. The default is private – that is, if Dim is used for a module level variable, the variable is private.

For variables declared inside a subprogram:

```
Dim <variable> As <Data_Type>
```

Variables declared inside a subprogram are visible only inside the subprogram.

### Code:

```
Public Distance As Integer ' Module-level variable, global
Private Temperature As Single ' Module-level variable local to module
Sub Main()
    Dim PinNumber As Byte 'Variable is local to this subprogram
End Sub
Sub ReadPin()
    Dim PinNumber As Byte 'Variable is local to this subprogram
End Sub
```

## VARIABLE IDENTIFIERS

Identifiers must start with a letter, and all other characters must be letters, digits or underscores.

An identifier can be up to 255 characters long, and all characters are significant.

Identifiers are not case sensitive. (E.g. Identifiers xyz, XYZ and xYz are equivalent.)

Reserved words like keywords are not allowed to be used as an identifier.

## DATA TYPES

BCore currently supports these data types.

Data Type	Memory Size	Content
Boolean	8bits	True/False
Byte	8bits / 1 Byte	0 – 255
Integer	16bits / 2 Bytes	-32 768 to 32 767
Long	32bits / 4 Bytes	-65 536 to 65 535
Single		
String	64Bytes	Max: 64 Chars
Point/Points()	32bit; (16bit X, 16bit Y)	X & Y (Point.X, Point.Y)
Rectangle/Rectangles()		2 Points (Point1, Point2)

## CONSTANTS

For constants declared in module-level code:

```
[Public | Private] Const constant_name As <Data_Type> = <literal>
```

By default, a declaration is private.

## LABELS

Labels serve as targets for the 'goto' statements. Mark the desired statement with a label and colon like this:

```
label_identifier : statement
```

**Code:**

```
Dim Count As Integer
Public Sub Main()
    Count = 0
    AGAIN:
        Count = Count + 1
        If Count <> 5 Then
            Goto AGAIN
        End If
    End Sub
```

## EXPRESSIONS

An expression is a sequence of operators, operands, and punctuators that returns a value.

**Code:**

```
ABS(variable)
SETBIT(variable, position)
CLRBIT(variable, position)
GETBIT(variable, position)
HIGH(variable)
LOW(variable)
```

### ARRAYS

An array represents an indexed collection of elements of the same type (called the base type).

**Note:** Maximum dimension is 3.

Supports;

- Global and local declaration
- Passing by reference to routines/subprograms

### Code:

```
Dim ArrayName(array_length) As Integer '1 dimension
Dim ArrayName(array_length_1, array_length_2) As Integer '2 dimensions
Dim ArrayName(array_length_1, array_length_2, array_length_3) As Integer '3 dimensions
```

**NOTE:** While the maximum dimension for declaring an array is 3, please take note that the maximum length allowed for the 2<sup>nd</sup> and 3<sup>rd</sup> dimension *e.g.* (*array\_length\_2, array\_length\_3*) is 255. There is no limit for the 1<sup>st</sup> dimension of the array, but onboard SRAM memory pertaining to the respective chip you are using should be taken to mind. The array length should not exceed the SRAM Memory available.

## STRINGS

A string represents a sequence of characters and is known to be a specialised array of characters. Strings are always declared to be 64 characters long, which is the maximum number of characters it can hold.

```
[Public | Private | Dim] <variable> As String
```

## STRING METHODS

The following table lists the methods that are available with the use of a string, and its description.

Method	Description
[+]Add	Add: Plus operator enables strings to be concatenated.
Compare	Compares two strings. Returns True if strings match, False if it does not.
Len	Returns length of String
Mid	Returns a string containing a specified range of characters from a string.

### [+] ADD

Strings can be concatenated by adding them together using the plus operator. Additions of strings are also allowed between strings of different memory sources, as well as in conjunction with other string methods.

#### Code:

```
Dim StringA, StringB, StringC As String
Public Sub Main()
    StringA = "This Is "
    StringB = "A String!"
    StringC = StringA + StringB
    Debug.Print StringC
End Sub
```

#### Output :

```
This Is A String!
```

### .COMPARE

Compares two strings. Returns *True* if strings match, *False* if it does not.

```
Variable = String.Compare(string_A, string_B)
```

### .LEN

Returns the number of characters within a specified string.

```
String.Len(String)
```

#### Code:

```
Public Sub Main()
    Dim str As String
    Str = "0123456789"
    Dim length As Integer
    Length = String.Len(str)
    Debug.Print "Length of Str = ";cstr(Length)
End Sub
```

#### Output :

```
Length of Str = 10
```

## MID

Returns a string containing a specified range of characters from a string.

**Note:** Unspecified length will return characters from Index to end of string.

`MID(String, Start Index, Length)`

### Code:

```
Public Sub Main()  
    Dim MIDtest As String  
    Dim a As String  
    Dim b As String  
    Dim c As String  
    MIDtest = "First Middle Last"  
    a = Mid(MIDtest, 1, 5)  
    b = Mid(MIDtest, 7, 6)  
    c = Mid(MIDtest, 7)  
    Debug.Print "a = ";a  
    Debug.Print "b = ";b  
    Debug.Print "c = ";c  
End Sub
```

### Output :

```
a = First  
b = Middle  
c = Middle Last
```

## STATEMENTS

Operators describe and perform an operation between two or more values.

### Statement format

A statement begins at the beginning of a line of text and terminates at the end of a line of text.

### IF Statement

Conditionally executes a group of statements, depending on the value of an expression.

```
If <condition> Then
    [ statements ]
[ ElseIf <condition> Then
    [ statements ] ]
[ Else
    [ statements ] ]
End If
-or-
If condition Then [ statements ]
```

### Select Case Statement

Runs one of several groups of statements, depending on the value of an expression.

```
Select Case <test expression>
Case <expression>
    [ statements ]
[ Case <expression>
    [ statements ] ]
[ Case Else
    [ statements ] ]
End Select
```

## ITERATION STATEMENTS (LOOPS)

### **For Statement**

Repeats a group of statements a specified number of times.

```
For counter = start To end [ Step value ]  
    [ statements ]  
    [ Exit For ]  
    [ statements ]  
Next
```

### **Code:**

```
Dim i As Integer  
For i = 0 To 3  
    Debug.Print CSTR(i)  
Next
```

### **Output :**

```
0  
1  
2  
3
```

### **Do Statement**

Repeats a block of statements while a Boolean condition is True or until the condition becomes True.

```
Do { While } <condition>  
    [ statements ]  
    [ Exit Do ]  
    [ statements ]  
Loop  
-or-  
Do  
    [ statements ]  
    [ Exit Do ]  
    [ statements ]  
Loop { Until } <condition>
```

### **Code:**

```
Dim i As Integer  
i = 0  
Do  
    Debug.Print Cstr(i)  
    i = i + 1  
Loop Until (i = 3)
```

### **Output :**

```
0  
1  
2  
3
```

## OPERATORS

Operators describe and perform an operation between two or more values.

### ARITHMETIC OPERATORS

Arithmetic operators are used to perform mathematical computations. They have numerical operands and return numerical results.

The arithmetic operators are addition (+), subtraction (-), multiplication (\*), division (/), integer division (\), modulus (Mod), negation (-) and exponentiation (^).

Order of precedence for arithmetic operators follow the rules of precedence from basic math, which is, left to right.

Operator	Operation
^	Exponentiation
-	Negation
*, /	Multiplication and division
\	Integer Division
Mod	Modulus
+, -	Addition and subtraction

*Table above lists operators and operations from highest precedence to the lowest.*

### RELATIONAL OPERATORS

Use relational operators to test equality or inequality of expressions. All relational operators return TRUE or FALSE.

Operator	Operation
=	equal
<>	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

*All relational operators associate from left to right.*

### BITWISE OPERATORS

Use the bitwise operators to modify the individual bits of numerical operands.

Operator	Operation
and	bitwise AND; compares pairs of bits and generates a 1 result if both bits are 1, otherwise it returns 0
or	bitwise (inclusive) OR; compares pairs of bits and generates a 1 result if either or both bits are 1, otherwise it returns 0
xor	bitwise exclusive OR (XOR); compares pairs of bits and generates a 1 result if the bits are complementary, otherwise it returns 0
not	bitwise complement (unary); inverts each bit
<<	bitwise shift left; moves the bits to the left, it discards the far left bit and assigns 0 to the right most bit.
>>	bitwise shift right; moves the bits to the right, discards the far right bit and if unsigned assigns 0 to the left most bit, otherwise sign extends

### BOOLEAN OPERATORS

#### **Operator Operation**

AND	logical AND
OR	logical OR
XOR	logical exclusive OR (XOR)
NOT	logical negation

*Boolean operators associate from left to right. The negation operator 'not' associates from right to left.*

### OVERALL OPERATOR PRECEDENCE

(Highest)	[1] Not
	[2] * \ Mod And
	[3] + - Or Xor &
(Lowest)	[4] = > < <> <= >=

## SUB MAIN()

All projects must have a procedure “Main()” in the project. This is the starting point of the project.

**Note:** Only **ONE** Sub Main() is allowed in a single project.

### Code:

```
Public Sub Main()  
  
End Sub
```

## PROCEDURES

Procedures and functions, referred to collectively as routines, are self-contained statement blocks that can be called from different locations in a program.

A **function** is a routine that **returns a value** when it is executed. A **procedure** is a routine that **does not return a value**.

Once these routines have been defined, you may call them once or multiple times. A procedure is called upon to perform a certain task, while a function is called to compute and return a certain value.

## SUB PROCEDURES

```
[Private|Public] Sub procedure_name (arguments)  
[statements]  
End Sub
```

You may exit a procedure at any point in the routine by using an `Exit Sub` statement.

### Code:

```
Private Sub GetAnswer(ByRef b As Boolean)  
    If (b = TRUE) Then  
        Exit Sub  
    Else  
        'do something  
    End If  
End Sub
```

## FUNCTIONS

```
[Private|Public] Function function_name (arguments) As <Data_Type>  
[statements]  
[Return Variable/Value]  
End Function
```

The function returns a value. This is achieved by using the ‘Return’ command followed by the value to return within the Function. You may also exit a function by using an `Exit Function` statement.

### Code:

```
Public Function F(ByVal i As Integer) As Integer  
    If (i = 3) Then  
        Return 92  
        Exit Function  
    End If  
    Return i + 1  
End Function
```

## PASSING PARAMETERS TO ROUTINES/SUBPROGRAMS

Parameters can be passed to a subprogram by reference (`ByRef`) or by value (`ByVal`).

**Passing by reference** -- if you pass a parameter by reference, any changes to the parameter will propagate back to the caller. Pass by reference is the default.

**Passing by value** -- if you pass a parameter by value, no changes are allowed to propagate back to the caller. With a few exceptions, if an argument is passed by value, a copy is made of the argument, and the called subprogram operates on the copy. Changes made to the copy have no effect on the caller.

One exception is for string parameters passed by value. For efficiency reasons, a copy of the string is not made. Instead, the string is write-protected in the called subprogram, which means you can neither assign to it nor pass it by reference to another subprogram. You are allowed to pass it by value to another subprogram, however.

The other exception is for types `Unsigned Integer` and `Unsigned Long`, which are treated similarly – these parameters are write-protected in called subprograms.

**Actual vs. formal parameters** -- the type and number of the actual parameters must match that of the "formal" parameters (the formal parameters appear in the subprogram declaration). If there is a type mismatch, the compiler will declare an error. It will not do implicit type conversions.

### **Restrictions on passing mechanisms:**

- Scalar variables and array elements can be passed by value or by reference.
- Since arrays are global they cannot be passed into a function or subroutine.
- Numeric expressions and numeric literals can be passed by value but not by reference. The same applies to Boolean expressions and Boolean literals.

	<b>ByRef</b> (By Reference)	<b>ByVal</b> (By Value)
Boolean	✓	✓
Byte	✓	✓
Integer	✓	✓
Long	✓	✓
Single	✓	✓
String	✓	✓
Array	✓	
Structure	✓	
Point/Points()	✓	
Rectangle/Rectangles()	✓	

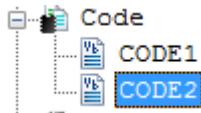
**STRUCTURES**

```
[Public | Private] Structure Structure_Name
    [Public | Dim] <variable> As <data_type>
End Structure
```

A structure declaration starts with the `Structure` statement and ends with the `End Structure` statement. The structure statement supplies the name of the structure, which is also the identifier of the data type the structure is defining. Other parts of the code can use this identifier to declare variables, parameters, and a function to return values of this structure's data type.

Structures are declared on its own, not contained within any procedures.

The declarations between the `Structure` and `End Structure` statements define the members of the structure.



Structures can be declared globally and accessed across different modules. The following code illustrates this.

A public structure is declared in Module “CODE2”.

**Code:**

```
'CODE2
Public Structure structUser

    Public Name As String
    Public Gender As Boolean '0 = Male; 1 = Female
    Public Age As Integer

End Structure
```

This structure is then declared and accessed through the main program in Module “CODE1”.

**Code:**

```
'CODE1
Public USER As CODE2.structUser

Public Sub Main()

    USER.Name = "Smith"
    USER.Gender = 0
    USER.Age = 28

End Sub
```

Notice that a variable declaration of a structure must be done before a structure can be used. In this case, the variable “USER” is declared of data type structure “*structUser*”.

## LATEST DOCUMENTATION

All of our documentations are constantly updated to provide accurate and/or new information that we feel would help you with developing with our products.

The latest documentation may be obtained from our website: <http://www.aiscube.com/main/downloads.html>

## HOW YOU CAN HELP

You can help us to improve our documentations by emailing to us or posting a thread in our forum, reporting any mistakes/typos or errata that you might spot while reading our documentation.

Email: [TechSupport@aiscube.com](mailto:TechSupport@aiscube.com)

Forum: <http://forum.aiscube.com/index.php>

## DISCLAIMER

All information in this documentation is provided 'as-is' without any warranty of any kind.

The products produced by AIS Cube are meant for rapid prototyping and experimental usage; they are not intended nor designed for implementation in environments that constitute high risk activities.

AIS Cube shall assume no responsibility or liability for any indirect, specific, incidental or consequential damages arising out of the use of this documentation or product.

COPYRIGHT© 2009 - 2011 AIS CUBE. ALL RIGHTS RESERVED.

ALL PRODUCT AND CORPORATE NAMES APPEARING IN THIS DOCUMENTATION MAY OR MAY NOT BE REGISTERED TRADEMARKS OR COPYRIGHTS OF THEIR RESPECTIVE COMPANIES. AND ARE ONLY USED FOR IDENTIFICATION OR EXPLANATION FOR THE OWNER'S BENEFIT. WITH NO INTENT TO INFRINGE.

SONATA IDE AND BLAZINGCORE(BCORE) ARE TRADEMARKS OF AIS CUBE IN SINGAPORE AND/OR OTHER COUNTRIES. ALL IMAGES DEPICTING THE BLAZINGCORE OR ANY PART OF IT IS COPYRIGHTED.

ALL OTHER TRADEMARKS OR REGISTERED TRADEMARKS ARE THE PROPERTY OF THEIR RESPECTIVE OWNERS.