

HBBR Basic

Language Reference

Version 2.000.306

Keywords

Keywords are case insensitive.

Following words are reserved as HBBR Basic keywords:

ACCESS ADDRESS ADDRESSOF AND APPEND AS ASM ASSERT AT
BINARY BOOLEAN BREAK BYREF BYTE BYVAL
CALL CASE CBOOL CBYTE CINT CLOSE CONST CSBYTE CSHORT CSNG CSTR CUINT CUSHORT
DEBUG DECLARE DIM DO DOUBLE
ELSE ELSEIF END ENDIF EXIT EXPLICIT
FALSE FOR FUNCTION
GET GOTO
IF INPUT INTEGER IRQ
LINE LOCK LOOP
MOD
NEXT NOT
OFF ON OPEN OPTION OR OUTPUT
PRINT PRIVATE PUBLIC PUT
RANDOM READ REGISTER REM RESET
SBYTE SEEK SELECT SHARED SHORT SINGLE STEP STRICT STRUCTURE STRING SUB
THEN TO TRUE
UINTEGER UNTIL USHORT
WIDTH WHILE WRITE
XOR

Identifiers

Identifier is a symbolic name of variable, constant, **Sub**, **Function** or **Irq**.

Identifier rules:

Copyright © 2006-2008 Hobby-Robotics, LLC. All rights reserved.

First character has to be either:

- lower case letter ['a' to 'z']
- upper case letter ['A' to 'Z'] or
- underscore ['_']

Following characters can be 0 or more of the:

- lower case letter ['a' to 'z']
- upper case letter ['A' to 'Z'],
- underscore ['_'] or
- digit ['0' to '9']

Literal Values

Literal Values (Literals) represent explicit numerical or string values in the program. Depending on the type they represent: logical values, numbers or character strings.

Boolean literals

True

False

Integral literals

Decimal notation - base 10

Syntax:

[-] [0123456789]

[-]	Negative number
[0123456789]	1 to 10 characters from the specified set, has to be within range for the type used.

Hex notation - base 16

Syntax:

& [h | H] [0123456789aAbBcCdDeEfF]

&[h H]	Hexadecimal literal prefix &h or &H
[0123456789aAbBcCdDeEfF]	1 to 8 characters from the specified set, has to be within range for the type used. A or a - represents 10 B or b - represents 11 C or c - represents 12 D or d - represents 13 E or e - represents 14 F or f - represents 15

Octal notation - base 8

Syntax:

&[o|O] [01234567]

&[o O]	Octal literal prefix &o or &O
[01234567]	1 to 11 characters from the specified set, has to be within range for the type used.

Binary notation - base 2

Syntax:

&[b|B] [01]

&[b B]	Binary literal prefix &b or &B
[01]	1 to 32 characters from the specified set, has to be within range for the type used.

Floating point literals

Decimal notation - base 10

Syntax:

[-] [0123456789] . [0123456789]

[-]	Negative number (sign)
[0123456789]	1 to 10 characters from the specified set, has to be within range for the type used.
.	
[0123456789]	1 to 6 characters from the specified set, fractional part.

Scientific notation

Syntax;

[-] [0123456789] . [0123456789] [eE] [+ -] [0123456789]

[-]	Negative number
------------	-----------------

[0123456789]	1 to 10 characters from the specified set, significand (mantissa) part
.	
[0123456789]	1 to 6 characters from the specified set, fractional part of significand (mantissa)
[eE]	
[+-]	Exponent sign
[0123456789]	1 to 10 characters from the specified set, exponent part

String literals

String literal are input as sequence of characters enclosed in double quotes which are not part of the **String** literal. String literals can contain spaces.

Syntax:

```
"[ ascii_character ]"
```

Data Types

Simple Data Types

Type	Range	Storage Size
Boolean	True, False	2 Bytes in Array
Byte	0 to 255	1 Byte in Array
SByte	-128 to 127	1 Byte in Array
Short	-32768 to 32767	2 Bytes in Array
UShort	0 to 65535	2 Bytes in Array
Integer	-2147483648 to 2147483647	4 Bytes in Array
UInteger	0 to 4294967295	4 Bytes in Array
Single	-3.4028235E+38 to -1.401298E-45 0 1.401298E-45 to 3.4028235E+38	4 Bytes in Array
String		1 Byte per character, aligned to 4 byte boundary

Boolean Type

Boolean type represents logical values **True** and **False**, internally represented as 1 **True** and 0 **False**.

Can be used to declare variable, structure field or constant.

Can be passed as argument by reference and value.

SByte Type

SByte type represents signed 8 bit values from -128 to 127.

SByte type is accessed as 1 byte value

In ARM32 code due to alignment requirements it occupies 4 byte in memory except for the arrays where it occupies 1 byte.

Can be used to declare variable, structure field or constant.

Can be passed as argument by reference and value.

Byte Type

Byte type can represent unsigned 8 bit values from 0 to 255.

Byte type is accessed as 1 byte value

In ARM32 code due to alignment requirements it occupies 4 byte in memory except for the arrays where it occupies 1 byte.

Can be variable and constant declared globally or locally.

Can be passed as argument by reference and value.

Short Type

Short type represents signed 16 bit values from to -32768 to 32767.

Short type is accessed as 2 byte value

In ARM32 code due to alignment requirements it occupies 4 byte in memory except for the arrays where it occupies 2 bytes.

Can be variable and constant declared globally or locally.
Can be passed as argument by reference and value.

UShort Type

UShort type represents unsigned 16 bit values from to 0 to 65535.

UShort type is accessed as 2 byte value

In ARM32 code due to alignment requirements it occupies 4 byte in memory except for the arrays where it occupies 2 bytes.

Can be used to declare variable, structure field or constant.
Can be passed as argument by reference and value.

Integer Type

Integer type represents signed 32 bit values from to -2147483648 to 2147483647.

Integer type is accessed as 4 byte value

Can be used to declare variable, structure field or constant.
Can be passed as argument by reference and value.

UInteger Type

UInteger type represents signed 32 bit values from to 0 to 4294967295.

UInteger type is accessed as 4 byte value

Can be used to declare variable, structure field or constant.
Can be passed as argument by reference and value.

Single Type

Single type represents floating point values from $-3.4028235E+38$ to $-1.401298E-45$, 0 and from $1.401298E-45$ to $3.4028235E+38$.

Single type is accessed as 4 byte value

Can be used to declare variable, structure field or constant.

Can be passed as argument by reference and value.

String Type

String is a collection of characters that can be manipulated as one. Can be variable and constant, as argument Strings can only be passed by reference only.

String type is accessed as sequence of 1 byte values.

In ARM32 code due to alignment requirements string length is incremented in 4 byte steps.

Can be used to declare variable or constant.

Can be passed as argument by reference

Variables and Constants

Variables and Constants store values of either simple type: **Boolean**, **SByte**, **Byte**, **Short**, **UShort**, **Integer**, **UInteger**, **Single** or **String** type.

Variables can also store complex types **Structure** and **Array**.

Both variables and constants have to be declared before they can be used. Declarations can be a global or local. Global declaration which is made outside of any **Sub**, **Function** or **Irq** is visible globally and can be used anywhere in the program. Local declaration which is made inside **Sub**, **Function** or **Irq** is only visible there.

Constant values are allocated in read-only memory (Flash) and can not be changed during program execution.

Global variables are allocated in RAM memory at specific location (address) which does not change during program execution.

Local variables are allocated on the stack in RAM memory and do not have fixed location.

For details on how to declare variable see **Dim** statement, for constant see **Const** statement.

Arrays

Array type

Array stores values of the simple type: **Boolean**, **Sbyte**, **Byte**, **Short**, **Ushort**, **Integer**, **UInteger** and **Single** arranged as a matrix with specified number of dimensions.

Array of **String** type is not supported in this version of HBBR Basic

Array of **Structure** type is not supported in this version of HBBR Basic

Array dimensions

Array index data type is **Integer**. Index range for each dimension is defined when array is declared.

All indexes are checked at runtime when array element is accessed. If index is not within declared range runtime error &h1001 "Invalid index" is issued.

Array can be passed by reference only as an argument to **Sub** and **Function**

Array can be declared globally or locally.

Global **Array** can be initialized when declared with **Dim** keyword using initialization expression.

Array declaration

Syntax:

```
Dim array_variable ( index_begin To index_end) As Type Name
```

```
Dim array_variable ( index_end) As Type Name
```

```
Dim array_variable ( index_begin To index_end) As Type Name
```

```
Dim array_variable ( index_end) As Type Name
```

Copyright © 2006-2008 Hobby-Robotics, LLC. All rights reserved.

Array initialization

Array values can be initialized with initialization expression at the time of **Array** variable declaration. Initialization expression has to match number of array dimension as well as a number of values in the array. Values are for each dimension grouped with **{ }** and must be valid literal values of the same type as array. If the array has more than 1 dimension then groups of values for the have to be nested.

Syntax:

```
Dim array_variable ( di_begin To di_end) As Type Name = { vbegin, ... v_end}  
Dim array_variable ( di_end) As Type Name = { vbegin, ... v_end}
```

Limitations:

Maximum number of array dimensions is 3.

Array can not be constant

Array can not be returned from **Function**

Local **Array** can not be initialized when declared with **Dim** keyword

Structures

Structure type

Structure type is a user defined collection of named data fields having values of the Simple Type: **Boolean**, **SByte**, **Byte**, **Short**, **UShort**, **Integer**, **UInteger** and **Single**.

Structure type has to be defined before it can be used in variable declarations.

Structure type is defined using following syntax:

Syntax:

```
Structure Type Name
    field_1 As Simple Type
    ...
    field_n As Simple Type
End Structure
```

Once the **Structure** type has been defined it can be used to declare variables using following syntax:

Syntax:

```
Dim structure_variable As Type Name
```

Structure can be passed by reference as an argument to **Sub** and **Function**

Structure can be declared globally or locally

Structure can be returned from **Function**

Global **Structure** can be initialized when declared with **Dim** keyword and initialization expression.

Two structure variables of the same type can be assigned using assignment operator

Structure initialization

Structure field values can be initialized with initialization expression at the time of **Structure** variable declaration. Initialization expression has to match number of fields in the **Structure**. Values are for each field must be valid literal value of the same type as the field.

Syntax:

```
Dim structure_variable As Type Name = { field1, ..., fieldn}
```

Limitations:

Structure types can not be nested inside another **Structure** (field can not be of type **Structure**)

Structure can not be constant

Local **Structure** can not be initialized

Maximum size of the **Structure** is limited to 256 Bytes

Expressions and Operators

Expression is any valid combination of operations performed on variables, constants and literals through the use of supported operators and functions. Elements of expression can be grouped with parenthesis to disambiguate types or override operator precedence. All of the standard BASIC operators are supported. Expression value has type determined by the values, operators and conversion rules.

Logical operators

Comparison operators

Comparison operators are used to compare expressions of the simple types as well as strings. Result of the comparison operator is **Boolean** value **True** or **False**. Strings are compared case sensitive based on alphabetical order.

Equal =

Syntax:

```
left_expression = right_expression
```

True if expression values are equal, **False** otherwise .

Not Equal <>

Syntax:

```
left_expression <> right_expression
```

True if expression values are not equal, **False** otherwise .

Less Than <

Syntax:

left_expression < right_expression

True if left expression value is less then the right expression value , **False** otherwise .

Less Then or equal <=

Syntax:

left_expression <= right_expression

True if left expression value is less then or equal to the right expression value, **False** otherwise .

Greater Then >

Syntax:

left_expression > right_expression

True if left expression value is greater then the right expression value , **False** otherwise .

Greater Then or Equal >=

Syntax:

left_expression >= right_expression

True if left expression value is greater then or equal to the right expression value, **False** otherwise .

Bitwise operators

Bitwise operators operate on individual bits of Integral Type values. Bitwise operators can not be used on **Single** or **String**.

And

And truth table

Bit 1	Bit 2	Result
0	0	0
0	1	0
1	0	0
1	1	1

Or

Or truth table

Bit 1	Bit 2	Result
0	0	0
0	1	1
1	0	1
1	1	1

Exclusive Or - Xor

Xor truth table

Bit 1	Bit 2	Result
0	0	0
0	1	1
1	0	1
1	1	0

Unary Not

Not truth table

Bit	Result
0	1
1	0

Shift operators

Shift left and right operators are defined for values of Integral Type only: **SByte**, **Byte**, **Short**, **UShort**, **UInteger** and **Integer**. Bits are shifted either left or right, new bit position are filled with 0. Maximum number of shift positions depends on the size of the value, if `shift_amount` is greater then the maximum it is treated as if it was bitwise **And**'ed with the range bit mask specific to the size of the value.

Shift right >>

Syntax:

```
value >> shift_amount
```

Shift left <<

Syntax:

```
value << shift_amount
```

Type	Size in Bytes	Maximum shift	Shift Range Mask
Sbyte, Byte	1	7	&h03
Short, UShort	2	15	&h0F
Integer, UInteger	4	31	&h1F

Mathematical operators

Mathematical operations are defined for both integral and floating point types.

Integer operators

If both values are of integral type result will be integral with the exception of the \ divide which will produce floating point value. To get integral result use Integer Divide \

Plus +

Result is sum of two expressions

Syntax:

```
left_expression + right_expression
```

Minus -

Result is subtraction of two expressions.

Syntax:

```
left_expression - right_expression
```

Multiply *

Result is multiplication of two expression.

Syntax:

```
left_expression * right_expression
```

Divide /

Result is division of two expressions.

Syntax:

```
left_expression / right_expression
```

Integer Divide \

Result is integer division of two expressions.

Integer operations are not checked for overflow.

Syntax:

left_expression \ right_expression

Floating point

Floating operation require Runtime with built-in Math support. If code is compiled with the Runtime without Math support Linker will report errors while trying to locate runtime floating point support functions.

Plus +

Result is sum of two expressions.

Syntax:

left_expression + right_expression

Minus -

Result is subtraction of two expressions.

Syntax:

left_expression - right_expression

Multiply *

Result is multiplication of two expression.

Syntax:

*left_expression * right_expression*

Divide /

Result is division of two expressions.

Syntax:

```
left_expression / right_expression
```

1. Division by 0.0 will result in runtime error &h2001 "Division by zero"

String operators

Concatenate & and +

Two string operators are defined, both will concatenate strings their behavior is equivalent.

Syntax:

```
left_expression & right_expression  
left_expression + right_expression
```

Combined assignment operators

Combined assignment and mathematical, string or shift operators are a short form of the following general assignment expression:

```
symbol = symbol operator expression
```

which can be expressed in the short form:

```
symbol operator = expression
```

In each case the short form is equivalent to the general form of the assignment and follows the same rules.

Plus and assign +=

Syntax:

```
symbol += expression
```

Minus and assign -=

Syntax:

```
symbol -= expression
```

Multiply and assign *=

Syntax:

```
symbol *= expression
```

Divide and assign /=

Syntax:

```
symbol /= expression
```

Integer Divide and assign \=

Syntax:

```
symbol \= expression
```

Concatenate and assign &= or +=

Syntax:

```
symbol &= expression  
symbol += expression
```

Shift right and assign >>=

Syntax:

```
symbol >>= expression
```

Shift left and assign <<=

Syntax:

```
symbol <<= expression
```

Special operators

AddressOf

Address of the symbol `AddressOf`

Syntax:

```
AddressOf name
```

Returns address of the symbol be variable, constant, **Sub**, **Function** or **Irq**.

Conversion operators

Syntax:

CBool

```
CBool (expression) - convert expression value to Boolean type
```

CByte

CByte(expression) - convert expression value to **SByte** type

CByte

CByte(expression) - convert expression value to **Byte** type

CShort

CShort(expression) - convert expression value to **Short** type

CUShort

CUShort(expression) - convert expression value to **UShort** type

CInt

CInt(expression) - convert expression value to **Integer** type

CUInt

CUInt(expression) - convert expression value to **UInteger** type

CShgn

CShgn(expression) - convert expression value to **Single** type

CStr

CStr(expression) - convert expression value to **String** type

Conversion operators are used for explicit conversion of the expressions to the specific type.

Comments

Comments can be added with the **Rem** keyword or ' ' character.

Any text that follows up to the end of line is a comment.

Example:

```
Rem Comment after rem keyword
```

```
'Comment after tick
```

Statements in alphabetical order

Call statement

Syntax:

```
Call name ( argument_list )
```

name	Identifier, Sub
argument_list	List of arguments, separated by coma “,”

Description:

Procedure call using name and passing list of arguments enclosed in the parenthesis. Arguments can be variables, constants or literal values.

If necessary actual argument value will be converted to match type of the corresponding argument in the in **Sub** declaration.

Close statement

Syntax:

Close

Close *#file_num*

Close *#file_num* [, *#file_num*]

<i>#file_num</i>	File number can be one of: Integer literal, Integer constant or Integer variable.
------------------	---

Description:

Close file or files identified by single file number or a list of file numbers *#file_num* [, *#file_num*]. File being closed must have been opened previously with **Open** statement. If no *#file_num* is specified then all currently open files will be closed. Once the file associated with the specific number has been closed then this file number can be reused for processing another file.

Close has no effect on files representing built-in devices like Console.

Errors:

&h3001 "File operation failed"

&h3002 "Bad file mode or access"

&h0004 "Not supported by the runtime"

Const statement

Syntax:

```
Const name As type = value
```

```
Const name As String = string_literal
```

name	Identifier
name	String identifier
type	Simple type: Boolean , Sbyte , Byte , Short , UShort , Integer , UInteger , Single or String
value	Literal value used to initialize constant
<i>string_literal</i>	Literal String value used to initialize String type constant

Description:

Declares constant value of the specified type. Type can a simple type (**Boolean**, **SByte**, **Byte**, **Short**, **UShort**, **Integer**, **UInteger**, **Single**) or **String**. If constant is declared outside of any **Sub**, **Function** or **Irq** it can be used anywhere in the program (global constant). If declared inside of **Sub**, **Function** or **Irq** it can be only used inside that **Sub**, **Function** or **Irq** (local constant). All constants global and local are allocated in the Flash (read only) memory.

Debug statement

Syntax:

Debug.Break

Debug.Print *name*

name	Identifier
------	------------

Description:

Debug statement is used to support debugging during development. Compiler will generate code for **Debug** statements only in debug build when compiler debug option is turned on. When compiler debug option is turned off **Debug** statements have no effect.

Debug.Break will insert breakpoint instruction into the generated code. Breakpoint instructions are handled by the runtime. Runtime with built in debug monitor allow debugging using HBBR IDE debugger.

Debug.Print statement sends value of the symbol converted to the **String** representation to the debug channel which is handled by the runtime. Runtime with built in debug monitor will send this value to the configured console device.

Declare statement

Syntax:

Declare Sub *name*(*argument_list*)

Declare Function *name*(*argument_list*) **As** *type*

Declare Register *name* **As Integer** @ *address*

Declare Dim *name*(*dl_start* **To** *dl_end* [, *dn_start* **To** *dn_end*]) **As** *type*

name	Identifier
<i>argument_list</i>	List of arguments, separated by coma “,”
type	Type, any of the: Boolean , Sbyte , Byte , Short , Ushort , Integer , UInteger , Single or String
address	Literal integer value representing address in memory

Description:

Declares reference to and external symbol which can be procedure **Sub**, **Function** or memory mapped hardware **Register**.

Declare Sub and **Declare Function** arguments declaration follow the same convention as the internal **Sub** or **Function** with the addition of the **ByRef** *name* **As** *address* which allows passing address of a variable or **Sub**, **Function** and **Irq**.

Declare Register at the specified memory address defines a symbol that can be used in the same way as s variable of the same type. Proper alignment for the specified address and data type has to be followed for a target platform.

Declare Dim declare external array of the specified type. See Dim statement for details.

Dim statement

Syntax:

Simple type variable declaration

```
Dim name As simple_type
```

```
Dim name As simple_type = value
```

String variable declaration

```
Dim name As String * size
```

```
Dim name As String = string_literal
```

Array declaration

```
Dim name( dimension_list ) As type
```

```
Dim name( d1_start To d1_end ) As type
```

```
Dim name( d1_start To d1_end, d2_start To d2_end ) As type
```

```
Dim name( d1_start To d1_end, d2_start To d2_end, d3_start To d3_end ) As type
```

Description:

name	Identifier
simple_type	Simple type (Boolean , SByte , Byte , Short , UShort , Integer , UInteger , Single)
value	Literal value used to initialize variable
size	Literal integer value used to determine length of the String variable

<i>string_literal</i>	Literal String value used to initialize String variable, literal length determines length of the String variable
<i>dimension_list</i>	List of array dimension ranges separated by coma ”,”
<i>d1_start d2_start d3_start</i>	Literal integer determines minimum index value
<i>d1_end d2_end d3_end</i>	Literal integer determines dimension maximum index value

Declares variable of the specified type. Type can a simple type (**Boolean, SByte, Byte, Short, UShort, Integer, UInteger, Single**) or **String**. If variable is declared outside of any **Sub,Function** or **Irq** it can be used anywhere in the program (global variable). If declared inside of **Sub, Function** or **Irq** it can be only used inside that **Sub, Function** or **Irq** (local variable). All variables global and local are allocated in the RAM (read write) memory. Global variables have fixed locations. Local variables are allocated on the stack and do not have fixed locations.

Do Loop statement

Do Loop is a flow control statement. Statements in a block between **Do** and **Loop** are executed based on the **Boolean Condition**.

```
Do [ Until | While ] Condition
```

```
...
```

```
[ Exit Do ]
```

```
...
```

```
Loop
```

Or

```
Do
```

```
...
```

```
[ Exit Do ]
```

```
...
```

```
Loop [ Until | While ] Condition
```

Infinite loop:

```
Do
```

```
...
```

```
[Exit Do]
```

```
...
```

```
Loop
```

<i>Condition</i>	Expression evaluating to Boolean
...	0 or more valid statements
[Exit Do]	Exit Do statement, early exit, optional statement

Description:

Flow control statement, will repeat block of code between **Do** and **Loop** statements depending on the specified *Condition* (conditional loop). *Condition* can be specified after **Do** or **Loop** using **While** or **Until** keyword.

When **While** is used *Condition* has to be **True** for loop to continue. When **Until** is used *Condition* has to be **False** for loop to continue. *Condition* is optional, if not present the loop will repeat continuously (infinite loop). **Do Loop** can be terminated (exited) using **Exit Do** placed anywhere in the loop block.

Depending on the placement of the *Condition*, two forms of conditional loop are possible:

1 – Pre-condition form where condition is checked before executing any loop statements in the current loop iteration. If condition is not met in the first iteration then no loop statement will be executed.

2 – Post-condition form where condition is checked after 1 loop iteration. If condition is not met in the first iteration loop statements will be executed exactly once.

For Next statement

For Next is a flow control statement. Statements in a block between **For** and **Next** are executed based on the counter variable and *End_expression*.

Syntax:

```
For Counter = Start_expression To End_expression
```

```
...
```

```
[ Exit For ]
```

```
...
```

```
Next Counter
```

```
For Counter = Start_expression To End_expression Step Step_expression
```

```
...
```

```
[ Exit For ]
```

```
...
```

```
Next Counter
```

<i>counter</i>	Variable used as loop <i>counter</i> , can be any variable global or local.
<i>Start_expression</i>	Expression determining initial (start) value of the <i>counter</i> variable. Will be converted to the <i>counter</i> data type before comparison.
<i>End_expression</i>	Expression determining end value of the <i>counter</i> variable. Will be converted to the <i>counter</i> data type before comparison.

<i>Step_expression</i>	Expression determining increment (step) in the value of the <i>counter</i> variable in each iteration. Optional, if Step is not specified then it defaults to 1
...	0 or more valid statements
[Exit For]	0 or 1 Exit For statement, optional exit statement

Description:

Flow control control loop statement, repeats statements between **For** and **Next** statements depending on the value of the loop *Counter* variable. Before first iteration loop *Counter* is set to *Start_expression*, in each subsequent iterations current value of the *Counter* is calculated by adding previous *Counter* value and *Step_expression*. If **Step** is not specified then *Step_expression* is equal to 1. Value of the *Counter* is compared with the *End_expression* to determine whether to execute loops statements. Comparison depends on the sign of the *Step_expression*.

If *Step_expression* is positive then loop continues to execute while *Counter* is less then or equal to *End_expression*.

If *Step_expression* is negative hen loop continues to execute while *Counter* is greater then or equal to *End_expression*.

Function statement

Syntax:

```
Function name ( argument_list ) As type
```

```
...
```

```
name = expression
```

```
[ Exit Function ]
```

```
...
```

```
End Function
```

Description:

Defines internal **Function** returning computed value of the specified type. **Function** can be used (called) in expression anywhere in the program using *name*.

When **Function** is called, statements between **Function** and **End Function** are executed and then computed **Function** value is returned.

Function body has to have at least one assignment to a **Function** *name* before exit from the **Function**.

If the function exits before assignment to *name* is made then the return value is undetermined.

Exit Function statement can be placed anywhere in the body of the **Function** to exit it early.

name	Identifier
argument_list	List of arguments, separated by coma ”,”
type	Type return value, can be any of the Simple Types: Boolean, SByte, Byte, Short, UShort, Integer, UInteger, Single as well as String Structure
expression	Return value expression

[**Exit Function**]

0 or 1 **Exit Function** statement, early exit, optional statement

Get statement

Syntax:

```
Get #file_num,, variable
```

```
Get #file_num, record_num, variable
```

<i>#file_num</i>	File number can be one of: Integer literal, Integer constant or Integer variable.
<i>record_num</i>	Optional record number expression.
<i>variable</i>	Variable name

Description:

Get reads binary data value from the file specified by file number and stores in the variable. Data type is determined by the variable type. File has to be opened in **BINARY** or **RANDOM** mode. Optional record number specifies starting position in the file depending on the mode. In **BINARY** mode record number is the byte offset within the file, in **RANDOM** mode it is the record number. Byte or record position in file starts at 1, second position is 2, third one is 3 etc.

Errors:

&h3001 "File operation failed"

&h3002 "Bad file mode or access"

Goto statement

Goto is the flow control statement.

Syntax:

Label:

```
...  
Goto label  
...
```

Description:

Flow control statement, transfers program execution to point where label is defined. **Label** must be defined inside **Function Subroutine** or **Irq**. Goto can jump to label in the current scope or in the scope above if statements are nested.

If Then statement

If ... **Then** is the flow control statement.

Syntax:

```
If condition Then  
    ...  
End If
```

```
If condition Then  
    ...  
Else  
    ...  
End If
```

```
If condition Then  
    ...  
ElseIf condition1 Then  
    ...  
ElseIf conditionN Then  
    ...  
End If
```

```
If condition Then  
    ...  
ElseIf condition1 Then  
    ...  
ElseIf conditionN Then  
    ...  
Else  
    ...  
End If
```

condition	Expression evaluating to Boolean value
condition1	Expression evaluating to Boolean value
conditionN	Expression evaluating to Boolean value
...	0 or more valid statements

Description:

Flow control statement, executes block of statements based on the *condition[1...N]*. Has following forms.

If Then form

If *condition* evaluates to **Boolean** value **True** statements between **If** and **End If** statements are executed.

If *condition* evaluates to **Boolean** value **False** then none of statements are executed.

If Then Else form

If *condition* evaluates to **Boolean** value **True** then statements between **If** and **Else** statements are executed.

If *condition* evaluates to **Boolean** value **False** then statements between **Else** and **End If** statements are executed.

If Then ElseIf Then form

If *condition* evaluates to **Boolean** value **True** then statements between **If** and **ElseIf** statements are executed.

If *condition* evaluates to **Boolean** value **False** then *condition1* following **ElseIf** statement is evaluated.

If *condition1* evaluates to **Boolean** value **True** then statements between **ElseIf** and next **ElseIf** statements are executed.

If *condition1* evaluates to **Boolean** value **False** then *condition2* following **ElseIf** statement is evaluated.

The above process is repeated if the current condition evaluates to **False** up to *conditionN*.

If none of the conditions evaluates to **True** then none of the statements is executed.

If Then ElseIf Then Else form

The execution flow is the same as **If Then ElseIf Then** except for the last step.

If none of the conditions evaluates to **True** then then statements between **Else** and **End If** statements are executed.

Input statement

Syntax:

Input *#file_num, variable*

Input *#file_num, variable [, variable]*

Description:

<i>#file_num</i>	File number can be one of: Integer literal, Integer constant or Integer variable.
<i>variable</i>	Variable name
<i>[, variable]</i>	0 or more coma ',' delimited variable names

Description:

Input reads text representation of the data value or list of values from file specified by file number and stores it in the specified variable or coma delimited variable list. File must be opened in **INPUT** or **APPEND** mode.

Errors:

&h3001 "File operation failed"

&h3002 "Bad file mode or access"

Line Input statement

Syntax:

Line Input *#file_num, string_variable*

<i>#file_num</i>	File number can be one of: Integer literal, Integer constant or Integer variable.
<i>string_variable</i>	Identifier of the String type variable.

Description:

Line Input reads line of text from the file specified by file number and stores it in the **String** variable. File must be opened in **INPUT** or **APPEND** mode. Line has to be terminated by carriage return and lined feed characters and can not be longer then the size of the runtime **String** buffer.

Errors:

&h3001 "File operation failed"

&h3002 "Bad file mode or access"

Irq statement

Syntax:

```
Irq name()  
    ...  
    [ Exit Irq ]  
    ...  
End Irq
```

<i>name</i>	
...	0 or more valid statements

Description:

Defines procedure that can be called to handle interrupt request. **Irq** takes no arguments and returns no value. Compiler generates code that preserves the state of the CPU on entry into the **Irq** and then restores it on exit.

Irq has to properly interact with the hardware interrupt controller which is specific to the target CPU. For the exact requirements consult hardware manual for the target CPU.

Exit Irq statement can be placed anywhere in the body of the **Irq** to exit it.

Open statement

Syntax:

```
Open file_path For [ APPEND | BINARY | INPUT | OUTPUT | RANDOM ] [ Access [ READ  
| WRITE ] ] [ SHARED | [ LOCK [ READ | WRITE ] ] ] As #file_num [ Len = expression ]
```

Simple form:

```
Open file_path For [ APPEND | BINARY | INPUT | OUTPUT | RANDOM ] As #file_num
```

Description:

<i>file_path</i>	File path
For	File mode: APPEND file opened for text append (Read Write) INPUT file opened for text input (Read) OUTPUT file opened for text output (Write) BINARY file opened for binary reading or writing (Read and/or Write) RANDOM file opened for record based reading or writing, depends on Access (Read and/or Write)
Access	File access type (optional): READ read only WRITE write only READ WRITE both read and write

Shared or Lock	File locking: SHARED LOCK READ LOCK WRITE LOCK READ WRITE
<i>#file_num</i>	File number can be one of: Integer literal, Integer constant or Integer variable.
Len = expression	Expression

Open file specified by *file_path* in one of the modes: **APPEND**, **BINARY**, **INPUT**, **OUTPUT** or **RANDOM**. Optionally allowed file operations type can be specified with **Access READ**, **WRITE**, or both **READ WRITE** keywords. If not specified then access type depends on file mode. Optionally file can be opened as **SHARED** allows subsequent opening or file can be locked for read and/or write operations **LOCK READ WRITE**.

Opened file is associated with the file number *#file_num* used in subsequent file operations.

Optionally for file opened in **RANDOM** mode record length can be specified with the **Len** = expression.

Predefined file numbers:

#0	Console file, redirected to the communication device as specified by the runtime configuration options. Default is serial device 0 (UART0).
----	---

Print statement

Syntax:

```
Print #file_num , expression
```

```
Print #file_num , expression [; expression]
```

Description:

<i>#file_num</i>	File number can be one of: Integer literal, Integer constant or Integer variable.
<i>expression</i>	Expression
[; <i>expression</i>]	0 or more semicolon ';' delimited expressions

Print value of expression or semicolon ';' delimited list of expressions to the file specified by file number. File must be opened in **OUTPUT** or **APPEND** mode. Each value is converted to the **String** representation before being printed. After printing expression values **Print** terminates with new line character or characters which can be configured through runtime options.

Errors:

&h3001 "File operation failed"

&h3002 "Bad file mode or access"

Put statement

Syntax:

```
Put #file_num , , expression
```

```
Put #file_num , record_num , expression
```

<i>#file_num</i>	File number can be one of: Integer literal, Integer constant or Integer variable.
<i>record_num</i>	Optional record number expression.
<i>expression</i>	Value

Description:

Put writes binary data representing value of the expression to the file specified by file number. Data type is determined by the expression type. File has to be opened in **BINARY** or **RANDOM** mode. Optional record number specifies starting position in the file depending on the mode. In **BINARY** mode record number is the byte offset within the file, in **RANDOM** mode it is the record number. Byte or record position in file starts at 1, second position is 2, third one is 3 etc.

Errors:

&h3001 "File operation failed"

&h3002 "Bad file mode or access"

Seek statement

Syntax:

```
Seek #file_num , expression
```

<i>#file_num</i>	File number can be one of: Integer literal, Integer constant or Integer variable.
<i>expression</i>	File position expression.

Description:

Seek sets current file position to the value of expression.

Errors:

&h3001 "File operation failed"

&h3002 "Bad file mode or access"

Select Case statement

Syntax:

```
Select Case expression  
  Case   expression_list1  
  ...  
  Case   expression_listN  
  ...  
End Select
```

```
Select Case expression  
  Case   expression_list1  
  ...  
  Case   expression_listN  
  ...  
  Case Else  
  ...  
End Select
```

Description:

<i>expression</i>	Expression used for selecting Case
<i>expression_list1</i>	Coma delimited list of expressions for the first Case
<i>expression_listN</i>	Coma delimited list of expressions for the Nth Case
...	0 or more valid statements

Flow control statement, executes statements selected from predefined cases.

Selection is based on the current value of the *expression*. At runtime *expression* is evaluated once and its value is used to find matching **Case** using *expression_list* for each case.

If match is found in the *expression_list* then statements between this and next **Case** and are executed.

If no matching **Case** is found behavior depends whether **Case Else** is included in the list of cases.

If it is not included then **Select Case** is skipped, no statements are executed.

If it is included then statements between **Case Else** end **End Select** are executed.

Case Else must be the last **Case**.

Sub statement

Syntax:

```
Sub name ( argument_list )  
    ...  
    [ Exit Sub ]  
    ...  
End Sub
```

name	Identifier
argument_list	List of arguments, separated by coma ”,”
[Exit Sub]	0 or 1 Exit Sub statement, early exit, optional statement
...	

Description:

Defines internal **Sub** procedure that can be called from anywhere in the program using name.

When **Sub** is called, statements between **Sub** and **End Sub** are executed.

Exit Sub statement can be placed anywhere in the body of the **Sub** to exit it early.

Width statement

Syntax:

Width *#file_num* , *expression*

<i>#file_num</i>	File number can be one of: Integer literal, Integer constant or Integer variable.
<i>expression</i>	Expression in the range [0 to 255]

Description:

Width sets line width for the file specified by file number to the value of expression. File must be opened in **OUTPUT** or **APPEND** mode.

Errors:

&h3001 "File operation failed"

&h3002 "Bad file mode or access"

Write statement

Syntax:

Write *#file_num* , *expression*

Write *#file_num* , *expression* [, *expression*]

<i>#file_num</i>	File number can be one of: Integer literal, Integer constant or Integer variable.
<i>expression</i>	Expression
[, <i>expression</i>]	0 or more coma ',' delimited expressions

Description:

Write value of expression or coma ',' delimited list of expressions to the file specified by file number. File must be opened in **OUTPUT** or **APPEND** mode. Each value is converted to the **String** representation before being written. Expression values written to the file are delimited with coma while string values are also enclosed in double quote characters. After writing all of the expression values **Write** terminates with new line character or characters which can be configured through runtime options.

Errors:

&h3001 "File operation failed"

&h3002 "Bad file mode or access"

Asm Statement

Syntax:

```
Asm  
    ...  
    [ Assembler instruction ]  
    ...  
End Asm
```

Description:

Defines block of assembler instructions. All text between **Asm** and **End Asm** is processed by the built-in assembler.

In-line Assembler

In-line assembler is supported in HBBRB version 1.300 or higher.

All text enclosed between **Asm** and **End Asm** is processed by the in-line assembler in the last stage of compilation.

In-line assembler has visibility into absolute symbols defined by the runtime module, user code symbols are not visible to the in-line assembler.

Visible symbols include:

- All hardware registers declared in the runtime module (MCU specific).
- Linker memory symbols:

- `__rom_start`
- `__rom_text_start`
- `__ram_word_size`
- `__rom_word_size`
- `__ram_byte_size`
- `__rom_rodata_start`
- `__rom_data_start`
- `__rom_bas_text_start`
- `__rom_bas_data_end`
- `__rom_data_end`
- `__rom_bas_rodata_start`
- `__rom_bas_data_start`
- `__rom_max`
- `__rom_byte_size`
- `__ram_start`
- `__ram_data_start`
- `__ram_bss_start`
- `__ram_bss_end`
- `__ram_bas_data_start`
- `__ram_bss_bas_end`

`__ram_bss_bas_start`
`__ram_heap`
`__ram_stack`
`__ram_max`

Address of the symbols can be loaded into a register using LDR pseudo-instruction:

Syntax:

```
LDR Rn,=symbol_name
```

In-line assembler will emit 2 or 3 instructions required to load the address into the specified register.

Preprocessing

Preprocessing Directives

Include directive

Includes specified file at the point where it is located in the original file.

Syntax:

```
#Include file_path
```

Conditional compilation directives

Const directive

Defines new compile constant with the value based on evaluating the expression.

Syntax:

```
#Const name = expression
```

If Then Elseif Else End If directives

Selects source files to be included in the compilation based on the value of the compile constants.

Syntax:

```
#If const_expression Then
```

```

...
#ElseIf  const_expression
...
[#Else]
...
#End If

```

Predefined #Const values

Name	Type	Value	Notes
HBBRVERSION	String	2.000.306	Version of the compiler
DATE	String		Date of compilation
TIME	String		Time of compilation
FILE	String		Name of the file being compiled
MODVERSION	String		Version of the runtime module – extracted from the MOD file
CPUID	Integer	Numeric representation of the target CPU Id	Specific to the runtime (MOD file)
DEBUG	Boolean	True or False	True when compiling with debug turned on, false otherwise.
LPC2000	Boolean	True or undefined	True when compiling for any of the NXP LPC2000 series microcontroller
LPC210X	Boolean	True or undefined	True when compiling for any of the NXP LPC210X
LPC2104	Boolean	True or undefined	True when compiling for NXP LPC2104
LPC2105	Boolean	True or undefined	True when compiling for NXP LPC2105
LPC2106	Boolean	True or undefined	True when compiling for NXP LPC2106
LPC213X	Boolean	True or undefined	True when compiling for any of the NXP LPC213X
LPC2131	Boolean	True or undefined	True when compiling for NXP LPC2131

LPC2132	Boolean	True or undefined	True when compiling for NXP LPC2132
LPC2134	Boolean	True or undefined	True when compiling for NXP LPC2134
LPC2136	Boolean	True or undefined	True when compiling for NXP LPC2136
LPC2138	Boolean	True or undefined	True when compiling for NXP LPC2138
LPC214X	Boolean	True or undefined	True when compiling for any of the NXP LPC214X
LPC2141	Boolean	True or undefined	True when compiling for NXP LPC2141
LPC2142	Boolean	True or undefined	True when compiling for NXP LPC2142
LPC2144	Boolean	True or undefined	True when compiling for NXP LPC2144
LPC2146	Boolean	True or undefined	True when compiling for NXP LPC2146
LPC2148	Boolean	True or undefined	True when compiling for NXP LPC2148
LPC229X	Boolean	True or undefined	True when compiling for any of the NXP LPC229X
LPC2292	Boolean	True or undefined	True when compiling for NXP LPC2292
LPC2294	Boolean	True or undefined	True when compiling for NXP LPC2294
AT91SAM7	Boolean	True or undefined	True when compiling for any of the Atmel AT91SAM7 series microcontroller
AT91SAM7S	Boolean	True or undefined	True when compiling for any of the Atmel AT91SAM7S
AT91SAM7S64	Boolean	True or undefined	True when compiling for Atmel AT91SAM7S64
AT91SAM7S128	Boolean	True or undefined	True when compiling for Atmel AT91SAM7S128
AT91SAM7S256	Boolean	True or undefined	True when compiling for Atmel AT91SAM7S256
AT91SAM7X128	Boolean	True or undefined	True when compiling for Atmel AT91SAM7X128
AT91SAM7X256	Boolean	True or undefined	True when compiling for Atmel AT91SAM7X256