

HBBR Basic Language

Version 2.000.306

Copyright

This document is Copyright © 2006-2008 by Hobby-Robotics, LLC

All trademarks within this guide belong to their legitimate owners.

Table of Contents

Introduction.....	4
HBBR BASIC Basics.....	6
HBBR Basic Project.....	8
Configuring the project.....	8
HBBR Basic Language.....	9
Data Types.....	10
Literals.....	13
Variables.....	14
Constants.....	15
Strings.....	16
Arrays.....	17
Structures.....	18
Registers.....	19
Expressions and Operators.....	20
Address of the symbol AddressOf.....	20
Conversion operators.....	20
Flow Control.....	22
If Then statement.....	22
Select Case statement.....	23
Do Loop statement.....	24
For To Next statement.....	25
Goto statement.....	26
Subroutine and Function.....	27
Subroutine.....	27
Function.....	28
Interrupts with Irq.....	29
File system support.....	30
Open statement.....	30
Close statement.....	31
Print and Write statements.....	31
Input and Line Input statements.....	32
Put and Get statements.....	32
Seek and Width statements.....	32
HBBRFS installable file system module	33
File system management API.....	33
Directory and file management API.....	33
File Input/Output API.....	33
Runtime.....	35
Errors.....	36
Preprocessing.....	37

Introduction

This is informal introduction of the HBBR Basic language, for detailed information please refer to the Reference section of this manual. HBBR Basic is a modern compiled structured dialect of a classical BASIC programming language. It is designed for programming advanced 32 bit microcontrollers with support from comprehensive runtime. It is also striving to be a subset of the most popular BASIC language implementation for PC computers.

Note:

More information on programming languages and BASIC can be found here:

[Wikipedia: Programming language](#)

[Wikipedia: BASIC](#)

[Wikipedia: Modern BASIC](#)

[Wikipedia: List of BASIC dialects](#)

[Wikipedia: Compiler](#)

[Wikipedia: Structured programming](#)

[Wikipedia: Integrated development environment](#)

Start page for the NXP (former Philips) LPC2000 series

[NXP: LPC2000](#)

Start page for the Atmel AT91SAM7 series

[Atmel: AT91SAM7](#)

Wikipedia microcontroller links

[Wikipedia: Microcontroller](#)

[Wikipedia: ARM7TDMI](#)

[Wikipedia: ARM architecture](#)

For up to date information on HBBR Basic go to

[HBBR: Basic Home Page](#)

This document is divided into two sections, in the first section “HBBR Basic Basics” you will find introduction to the concepts behind programming language and its environment. This section is meant for a beginner, if you are familiar with programming you can skip it and proceed directly to the “HBBR Basic Language Details” section.

HBBR BASIC Basics

If you are just starting with programming it will probably seem very confusing and daunting at first, so many new concepts and details. HBBR Basic is meant to make it easier. Let start by examining a real program which in a programming tradition will say hello to the world. For a step by step instructions on how to run HBBR Basic IDE and edit, compile and program your microcontroller board please use the “Hello World! Tutorial” included in the documentation. Here we will assume that you are already running HBBR Basic IDE, have the microcontroller board connected and discuss just discuss the program itself.

[Wikipedia: Computer Program](#)

Hello World! - step by step

Here is the very first program, just 3 lines that will show the message on the screen of your computer.

```
1 Sub Main()  
2     Print #0, "Hello World!"  
3 End Sub
```

It is easy to see that line 2 contains a “Hello World!” text but how does it get to your computer screen? It is send there with the “Print” command! First “Print” has to know what to print and where it should go. It already knows what to print the second argument string the “Hello World!”. The first “Print” argument “#0” stands for file number 0 . There are really no files here, because it is just a concept to treat all input and output devices the same way as files. In case of the microcontroller the most likely device that is hiding behind #0 is the serial device (UART0). So “Print” will actually send the message from the microcontroller and through the serial connection to the PC computer. To see this message on the computer screen there must be something that can receive it and show it. One option is to use a generic terminal program like Windows “HyperTerminal” or the HBBR Basic IDE built-in terminal which is configured to make this connection seamless.

“Print” takes the message and sends it character by character over the serial connection to the PC computer where it is received and shown on the screen.

Line 2 does the job of printing the message but there are still 2 more lines in this program, they must be doing something. These two lines are very important and every HBBR Basic program will have to have them because they define subroutine Main. What is a subroutine? It is a way to group programming statements into sub-program and then give them a name so it can be called when needed. In this case the line 1 defines “Sub” as having a name “Main” as taking no arguments “()”, line 3 “End Sub” denotes where the “Main” ends, there could be other Sub's following Main. “Sub” Main is special because this is the place where Basic program starts after it is uploaded to the microcontroller memory.

Message that to be printed can be put either in a string variable or constant.

```
1 Const msg$ As String="Hello World!"
2 Sub Main()
3     Print #0, msg$
4 End Sub
```

HBBR Basic Project

HBBR Basic is organized around concept of a project consisting of one or more source files “.bas” and one runtime module file “.mod”. One “.bas” file must be designated as the Main file and has to contain declaration of the Sub Main().

Before compiling the project should be configured through the options.

After compiling program can be either uploaded to the microcontoller or saved as a binary.

Configuring the project

Exact set of options depends on the specific runtime used but there are certain options that need to be configured for every project. Please refer to the MOD Reference for details.

XTAL - oscillator frequency

Found under **Configuration** -> **CPU_Family** -> **XTAL**

This is the reference frequency that must be correctly configured.

HBBR Basic Language

HBBR Basic is a structured dialect of the BASIC language. Following is the introduction to the fundamental concepts. Detailed reference can be found in the Language Reference.

[Wikipedia: Programming Language Syntax](#)

Data Types

Data type is a fundamental element of any programming language. Currently HBBR Basic supports following simple types as well strings and as arrays of simple types.

List of supported simple types:

- **Boolean**
- **SByte**
- **Byte**
- **Short**
- **UShort**
- **Integer**
- **UInteger**
- **Single**

[Wikipedia: Data type](#)

[Wikipedia: Primitive type](#)

Boolean type represents logical values **True** and **False**. It is used to perform logical operations (not bitwise!).

Example:

```
Dim boolVar As Boolean
Const boolConst As Boolean = True
Dim boolArray(1 To 2) As Boolean
Sub boolSub( ByVal boolArg As Boolean)
```

[Wikipedia: Boolean data type](#)

SByte type can be use to represent signed 8 bit value from -128 to 127

Example:

```
Dim sbyteVar As SByte
Const sbyteConst As SByte = &b11111111
Dim sbyteArray(1 To 2 ) As SByte
Sub sbyteSub( ByVal sbyteArg As SByte)
```

Byte type can be use to represent unsigned 8 bit value from 0 to 255.

Example:

```
Dim byteVar As Byte
Const byteConst As Byte = &b11111111
Dim byteArray(1 To 2 ) As Byte
Sub byteSub( ByVal byteArg As Byte)
```

[Wikipedia: Byte](#)

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

Example:

```
Dim shortVar As Short
Const shortConst As Short = &hFFFF
Dim shortArray(1 To 2 ) As Short
Sub shortSub( ByVal shortArg As Short)
```

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

Example:

```
Dim ushortVar As UShort
Const ushortConst As UShort = &hFFFF
Dim ushortArray(1 To 2 ) As UShort
Sub ushortSub( ByVal ushortArg As UShort)
```

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

Example:

```
Dim integerVar As Integer
Const integerConst As Integer = &hFFFFFFFF
Dim integerArray(1 To 2 ) As Integer
Sub integerSub( ByVal integerArg As Integer)
```

[Wikipedia: Integer datatype](#)

UInteger type represents unsigned 32 bit value from 0 to 4294967295.

Example:

```
Dim uintegerVar As UInteger
Const uintegerConst As UInteger = &hFFFFFFFF
Dim uintegerArray(1 To 2 ) As UInteger
Sub uintegerSub( ByVal uintegerArg As UInteger)
```

Single type represents floating point values from -3.402823E38 to 3.402823E38 .

Example:

```
Dim singleVar As Single
Const singleConst As Single = 1.0
Dim singleArray(1 To 2 ) As Single
Sub singleSub( ByVal singleArg As Single)
```

Floating operation require runtime MOD file 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.

[Wikipedia: Floating Point](#)

[Wikipedia: IEEE Floating Point Standard](#)

Literals

Literals represent values of a specific type in the source code.

Example:

```
' Integer literals
&b11111111111111111111111111111111
1
&hFFFFFFFF
-1

' Single (floating point literals)
0.0
1.234567
10E-10

'String literals
"this is a string literal"
```

[Wikipedia: Literal](#)

[Wikipedia: String literal](#)

[Wikipedia: Hexadecimal](#)

[Wikipedia: Octal](#)

[Wikipedia: Binary](#)

[Wikipedia: Computer numbering formats](#)

Variables

Variable hold values of a specified type. Variable value or content can be changed during program execution through assignment . Variables can be declared globally outside of any **Sub**, **Function** or **Irq** or locally inside **Sub**, **Function** or **Irq**. When variable is declared globally is is visible from any subroutine in the project.

All simple type and string variables can be initialized in the declaration

Example;

```
Dim b As Byte = &FF
```

```
Dim s$ As String = "string"
```

Variable initialization is the same as constant except for the **Dim** keyword. Variable value can be changed during program execution using assignment. All variables are located in the RAM memory.

Global variables have permanent locations and exists through entire duration of the program execution. Total size of global variables is only limited by the available RAM memory. Local variables are allocated on the stack and exists only procedure or function is called. Local stack size has a fixed size limit which means that the total size of all local variables must be smaller then the maximum allowed stack size.

[Wikipedia: Variable](#)

[Wikipedia: Variable in computer programming](#)

Constants

Constant values of simple types and strings can be declared globally outside of any **Sub**, **Function** or **Irq** or locally inside **Sub**, **Function** or **Irq**. When globally declared they are visible from any subroutine. **Const** values can not be changed during program execution, all constants are located in the Flash memory.

Example:

```
Const cBoolean As Boolean = True
Const cSByte As SByte = -1
Const cByte As Byte = 0
Const cShort As Short = &h1
Const cUShort As UShort = 65535
Const cInteger As Integer = -1000
Const cUInteger As UInteger = &HFFFFFFF
Const cSingle As Single = 10.33333
Const cString As String = "const string"
```

[Wikipedia: Constant](#)

Strings

HBBR Basic implementation supports both constant **Const** and non constant strings. Once initialized or declared each string occupies a fixed amount of memory which is big enough to hold entire string.

Runtime provides string manipulation subroutines see String Functions reference for details.

[Wikipedia: String datatype](#)

Arrays

Arrays are used to arrange values into a matrix. Array elements can be of **SByte**, **Byte**, **Short**, **UShort**, **Integer**, **UInteger** and **Single** types. Array can have multiple dimensions with specified index range. Maximum supported number of dimensions in the array is 3. Array element can be used in the same way as variable of the same type.

Arrays can be global or local and follow the same visibility rules as variables. Global arrays are limited in size by the available amount of memory. Local variables have size limited by the local stack limit.

Global arrays can be initialized when defined with **Dim** keyword.

Example:

```
' Array declaration
Dim arrayOfBytes1Dimension(1 To 32) As Byte
Dim arrayOfShorts2Dimensions(1 To 32, -1 To 10) As Short

' Array declaration with initialization
Dim arrayOfBytesInitialized(1 To 2) As Byte = {1, 2}
Dim arrayOfBytesInitialized(1) As Byte = {0, 1}

'Array element assignment
arrayOfBytes1Dimension(1) = 1
arrayOfBytes1Dimension(index) = 3
arrayOfShorts2Dimensions(1,-1) = 10
```

[Wikipedia: Array](#)

Structures

Structure can be used to group related fields (values) into a user defined type which can be used to declare global and local variables. Structure fields can be any combination of **SByte**, **Byte**, **Short**, **UShort**, **Integer**, **UInteger** and **Single** types.

Structures can be global or local, can be passed by reference as arguments and can be returned from **Function**. **Structure** can be assigned to another structure of the same type. Structures follow the same visibility rules as variables. Global structures are limited in size by the available amount of memory and compiler limit of 256 Bytes. Local structures have size limited by the local stack limit and current compiler limit of 256 Bytes. Global structures can be initialized when defined with **Dim** keyword, local structures can not be initialized.

Example:

```
Structure MyStructure
    byte_field    As Byte
    short_field   As Short
End Structure

Dim str As MyStructure = { 255, -1000}
```

Limitations:

String type fields are not supported in this release.

Total structure size is limited to 256 bytes.

Registers

Register type represents a hardware register present on a microcontroller at a specific address location. Typically ARM based microcontrollers have memory mapped 32 bit registers. Registers can be used in the same way as variables of the type **Integer**. However the key difference is that the compiler will always generate code to access the **Register** either to read or store the value. If needed current value of the **Register** can be assigned to the **Integer** variable for later use. All of the registers are declared in the runtime module specific to the microcontroller.

Example:

```
'Timer 0 IR Interrupt Register
Declare Register T0IR As Integer @ &hE0004000
' Assign HEX value
T0IR = &h0
'Assign binary value
T0IR = &b00000000000000000000000000000000
```

[Wikipedia: Hardware register](#)

Expressions and Operators

All computations are performed by evaluating expressions. 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.

Example:

```
1 + 1          ' integer addition, integer result
1 = 1          ' comparison, boolean result
varI \ 2       ' integer division
```

Note:

Additional information on programming expressions can be found here:

[Wikipedia Expression \(programming\)](#)

[Wikipedia: Operator](#)

[Wikipedia: Binary operator](#)

[Wikipedia: Unary operator](#)

Address of the symbol AddressOf

Returns address of the symbol either variable or subroutine (sub, function or irq) One use is to pass address of a callback function to runtime.

Example:

```
__hbbr_install_irq(I2C0_INT, AddressOf I2C0_Handler )
```

Conversion operators

Conversion operators are used for explicit conversion of the expression to the specified type.

CBool(*expression*) - cast expression to Boolean type

CByte(*expression*) - cast expression to SByte type

CByte(*expression*) - cast expression to Byte type
CShort(*expression*) - cast expression to Short type
CUShort(*expression*) - cast expression to UShort type
CInt(*expression*) - cast expression to Integer type
CUInt(*expression*) - cast expression to UInteger type
CSng(*expression*) - cast expression to Single type
CStr(*expression*) - cast expression to String type

Example:

```
varBool = CBool("True")
```

[Wikipedia: Type conversion](#)

Flow Control

Flow control statements alter program execution based on the conditions or values. HBBR Basic supports **If Then**, **Select Case**, loop constructs via **Do Loop** and **For Next** as well as **GoTo** statement.

[Wikipedia: Control flow](#)

If Then statement

If Then statement is used to alter program execution based on the condition. *Condition* is a expression which is first converted to the Boolean type and then used to make a decision.

```
If Condition Then  
...  
End If
```

This the simples form of the **If Then** statement, it checks whether the *Condition* evaluates to **Boolean** value **True** and if it does executes statements between **If** and **End If**. If the *Condition* evaluates to **False** no statements are executed.

```
If Condition Then  
...  
Else  
...  
End If
```

In the more complex form the **If Then** checks whether the *Condition* evaluates to **Boolean** value **True**, if it does it executes statements between **If** and **Else**. If the *Condition* evaluates to **Boolean** value **False** statements between **Else** and **End If** will be executed.

```
If Condition1 Then  
ElseIf Condition2 Then  
...
```

```
ElseIf ConditionN Then
...
Else
...
End If
```

Third form of the **If** with multiple conditions checks whether the *Condition1* evaluates to **Boolean** value **True** and if it does it executes statements between **If** and **Else** and then exists the **If**. If *Condition1* is **False** then *Condition2* of the first **Elseif** is evaluated, if that condition is **False** then following **Elseif** conditions are evaluated up to *ConditionN*, block of statements following the first **True** condition is executed and then **If** is exited. If no condition evaluates to **True** then statements between **Else** and **End If** will be executed. The **Else** block is optional. There is no limit on number of **Elseif** parts.

[Wikipedia: Structured If](#)

Select Case statement

```
Select Case Expression
    Case Expression1
    ...
    Case ExpressionN
    ...
End Select
```

This **Select Case** form allows to chose statements to be executed from the list of predefined cases. Selection is based on the current value of the *Expression* and *Expression1* to *ExpressionN*. At runtime *Expression* is evaluated once and its value is used to select block of statements from the list of cases. The selection is done sequentially starting from **Case Expression1** through **Case ExpressionN**. If comparison of current value of *Expression* and *ExpressionX* evaluates to **Boolean** value **True** then block of statements for this case is executed then **Select Case** is exited. If current *Expression* evaluates to **Boolean** value **False** then next **Case ExpressionX** is evaluated. If no comparison evaluates to **Boolean** value **True** then no statements are executed.

```

Select Case Expression
    Case    ExpressionList1
    ...
    Case    ExpressionListN
    ...
    Case Else
End Select

```

Second **Select Case** form introduces one change as compared to the form without the **Case Else**. If no comparison evaluates to **Boolean** value **True** then statements in a block between **Case Else** and **End Select** are executed.

[Wikipedia: Choice based on specific constant values](#)

Do Loop statement

Do Loop executes repeatedly block of statements based on a **Condition**. **Condition** can be any expression that evaluates to **Boolean** value **True** of **False**. The **Condition** is specified following **Until** or **While**. If **Until** is used then **Condition** must evaluate to **True** before loop will terminate. If **While** is used then **Condition** must evaluate to **False** before loop will terminate. If there is no **Condition** then statements will be executed forever, infinite loop. If condition follows **Do** keyword then it is evaluated before any statement in the **Do Loop** block is executed. If **Condition** follows **Loop** keyword then it is evaluated after **Do Loop** block statements have been executed once. The optional **Exit Do** statement allows to exit the **Do Loop** block immediately regardless of the condition, can be used to terminate infinite loops.

Do Loop form where **Condition** is evaluated first before any statements are executed.

```

Do [ Until | While ] Condition
...
[ Exit Do]
...
Loop

```

Do Loop form where **Condition** is evaluated after statements have been executed once.

```
Do
...
[ Exit Do]
...
Loop [ Until | While ] Condition
```

Infinite **Do Loop** form, statements are executed repeatedly.

```
Do
...
Loop
```

Infinite **Do Loop** form can be terminated with Exit Do.

```
Do
...
[ Exit Do]
...
Loop
```

[Wikipedia: Infinite loop](#)

[Wikipedia: Condition-controlled loops](#)

For To Next statement

For Next repeats block of statements based on the counter variable, counter variable is incremented after each execution by adding Step value. Counter variable is assigned Start_expression value on entry into the loop. Block of statements is executed if the counter variable value is less then or equal to the End_expression for positive Step values. For negative Step values the condition is more then or equal.

Loop with default Step 1

```
For Var = Start_expression To End_expression  
...  
Next Var
```

With Step specified

```
For Var = Start_expression To End_expression Step Expression  
...  
Next Var
```

[Wikipedia: Count-controlled loops](#)

[Wikipedia: For loop](#)

Goto statement

GoTo can be used to alter flow of execution by jumping to the specified *Label*.

Label has to be located in the same scope as the **GoTo** or in the scope above but in the same **Fun**, **Sub** or **Irq** or the compiler will issue an error.

```
...  
Goto Label  
...  
Label:  
...
```

[Wikipedia: GOTO](#)

Subroutine and Function

Subroutine **Sub** and **Function** group related statements to be called as a single block. Both **Sub** and **Function** accept arguments as inputs. Arguments can be used in the same way as variables of the same type.

Function returns a value of a specified type, there must be at least one assignment to the **Function** represented by its name in the body of the the **Function**. Function can be part of a an expression. Subroutine has to be called explicitly with **Call**.

Arguments are passed in two distinct ways:

ByRef – pass by reference

Reference to a variable or constant is passed to the subroutine. In the variable has been passed and as an argument then any change to it's value are retained when the subroutine or function finishes. Value of a constant passed by the reference can not be changed.

If is it not specified explicitly passing by reference is the default behavior

ByVal – **pass** by value

Value of the variable, constant or literal is passed to the subroutine. Value of the argument can be changed inside the subroutine regardless of what was passed as the argument. However, all of the changes are local to the function.

[wikipedia: Argument \(parameter\)](#)

Subroutine

There must be one subroutine called **Main** defined in the project. This is the entry point for the HBRR Basic program. **Main** does not take any arguments.

```
Sub Name ( Arguments )  
End Sub
```

[Wikipedia: Subroutine](#)

Function

Functions return values and can be used anywhere in the expression where a variable of the same type as the **Function** could be used. Assignment to a Function can be made multiple times but there must be at least one assignment before the function exists or the result will be unpredictable.

```
Function Name ( arguments ) As Type  
Name = value  
End Function
```

Interrupts with Irq

Irq is a special kind of Subroutine used to handle microcontroller interrupts. It takes no arguments and can not be called directly from the program. It is called after it is installed using runtime function to handle specific interrupt and this interrupt is generated. **Irq** can access any global variables however in a typical system interrupted handlers should execute very quickly so care has to be taken to limit the amount of time is spent in the **Irq**.

```
Irq Name ()  
End Irq
```

[Wikipedia: Interrupt](#)

File system support

HBBR Basic supports fundamental file operations through file statements. Following files statements are part of the language:

Open, Close, Print, Write, Input, Line Input, Put, Get, Seek, Width, Reset

File statements provide consistent interface to the file operations which can be implemented for both files and objects other than files like serial port, I2C interface or any communication channel. Typically the underlying functionality is implemented in the library module (MOL) separate from the HBBR Basic runtime module (MOD). However, HBBR Basic runtime module has built-in support for console, and serial ports (UART0, UART1) devices which are treated as permanently opened text files.

All files are accessed and processed using unique file number (#file_num) associated with the file. The association starts then the file is opened (**Open** statement) and ends when the file is closed (**Close** statement). File numbers can be reused once the file is closed. There is a number of predefined file numbers associated with the built-in devices for example Console device is #0.

Open statement

The simplest form of **Open** statement requires file name, open mode and associated file number.

Example:

```
Open "test" For Output As #fnum
```

In the above example file named “path” will be opened as a text file for output and the associated file number will be the value of the fnum variable or constant.

Full form of the Open statement has all details of the file mode, allowed access type, locking and record size.

File mode can be one of the:

APPEND, INPUT, OUTPUT – text file, line based processing

BINARY – binary file

RANDOM – binary file, fixed size record based processing

File access can be one of:

READ - read only access

WRITE - write only access

READ WRITE - both read and write access

File locking can be one of:

SHARED - no locking file can be opened again

LOCK READ - file can not be open for for reading

LOCK WRITE - file can not be open for writing

LOCK READ WRITE - file can not be open for both reading and writing

Close statement

Close statement is used to end all file operations for the file associated with the file number. After file is closed the file number can then reused.

Example:

```
Close #fnum1
```

Close statement has two more variations: if no file number is specified then all of the currently open files will be closed, list of file numbers can be specified on the same line in a coma delimited format.

Example:

```
Close  
Close #fnum1, #fnum2, #fnum3
```

Print and Write statements

Both **Print** and **Write** statements output text representation of the data to the specified file.

Despite being quite similar the is one key difference between them. **Write** separates all output values with coma ',' and encloses all strings in double quotes '"'. Coma delimited format and quoted strings make the **Write** output better suitable to be processed with **Input** statement.

Example:

```
Print #0, "Hello World!"
Write #fnum, var1; var2; var3
```

Input and Line Input statements

Both **Input** and **Line Input** statements read text representation of the data from the specified file.

Line Input reads single line of text into a String type variable. **Input** reads one or more values depending on how many variables are specified on the **Input** list. **Input** can read both white space separated output format produced by **Print** statement or comma delimited output format produced by **Write** statement (recommended).

Example:

```
Line Input #0, line_buffer$
Input #fnum, var1, var2, var3
```

Put and Get statements

Put and **Get** statements handle reading and writing binary data values to and from the file opened in either **BINARY** or **RANDOM** mode. Data written out with **Put** can be read with **Get**. Optionally it is possible to specify record number which is interpreted as byte position for file opened in **BINARY** mode or record number for file opened in **RANDOM** mode. If the record number is not specified current file position is advanced after each operation by the size of the data read or written.

Example:

```
Put #fnum,,var1
Get #fnum,,var1
```

Seek and Width statements

Seek statement sets current file position to the value specified. First position in the file is 1, second is 2 and so on.

Example:

```
Seek #fnum,1000
```

Width sets line width for file opened in one of the text modes INPUT, OUTPUT or APPEND.

Example:

```
Width #fnum, 16
```

HBBRFS installable file system module

HBBR Runtime can be extended with an installable file system module (HBBRFS) providing set of defined functions and subroutines for handling different file systems. Separate implementations are provided for SD/MMC card based FAT file system and Flash memory based file system, other file systems are being developed .

HBBRFS module provides uniform API divided into following categories:

File system management API

```
Function HBBRFSInit() As Integer
Function Mount(ByVal Device As Integer, ByRef Path As String) As Integer
Function Umount(ByVal Device As Integer, ByRef Path As String) As Integer
```

Directory and file management API

```
Sub ChDrive(ByRef Drive As String)
Sub ChDir(ByRef Path As String)
Sub Mkdir(ByRef Path As String)
Sub Rmdir(ByRef Path As String)
Function CurDir() As String
Sub Kill(ByRef Path As String)
Sub Rename(ByRef OldPath As String, ByRef NewPath As String)
Sub FileCopy(ByRef SrcPath As String, ByRef DestPath As String)
```

File Input/Output API

```
Function FreeFile() As Integer
Sub FileOpen( ByVal FileNumber As Integer, ByRef FileName As String, ByVal Mode As Uinteger)
Sub FileClose( ByVal FileNumber As Integer )
```

```
Sub FilePut( ByVal FileNumber As Integer, ByRef Value As Type)
Sub FileGet( ByVal FileNumber As Integer, ByRef Value As Type)
Function FileLen( ByRef PathName As String ) As UInteger
Sub FileSeek( ByVal FileNumber As Integer, ByVal Position As
UInteger)
Function EOF( ByVal FileNumber As Integer) As Boolean
Function Loc( ByVal FileNumber As Integer) As UInteger
Function LOF( ByVal FileNumber As Integer) As UInteger
```

For details please refer to the HBBR Basic Reference.

Runtime

Runtime module (MOD file) implements all of the support functionality necessary to initialize CPU, handle interrupts and exceptions, implement low level drivers for the selected peripherals as well as all of the support functions for the Basic language. To compile HBBR Basic program one of the runtime modules must be included in the project. HBBR Basic comes with runtime modules configured for different needs with different functionality. User has the flexibility to choose one that matches the need best.

Runtime Modules included in the distribution:

- Bare Metal - does not include full Math support, supports Single data format.
- Standard - full Math support
- Full - With support for HAL classes and full support for mathematical functions

For details refer to the Runtime reference.

[Wikipedia: Runtime](#)

Errors

Runtime runtime error codes and messages

- &h0000 - “OK “
- &h0001 - ”BASIC code not found”
- &h0002 - “Runtime function failed”
- &h0003 - “Bad argument value”
- &h0004 - “Not supported by the runtime”
- &h0005 - “Failed to allocate memory”
- &h0006 - “Failed to free memory”

Runtime BASIC error codes and messages

- &h1000 - “OK “
- &h1001 - "Invalid index"
- &h1002 - "Incorrect format"
- &h1003 - "Numerical conversion failed"

Runtime BASIC Math error codes and messages

- &h2000 - “OK “
- &h2001 - "Division by zero"
- &h2002 - "Overflow"

Runtime BASIC File error codes and messages

- &h3000 - “OK “
- &h3001 - "File operation failed"
- &h3002 - "Bad file mode or access"

[Wikipedia: Error message](#)

Preprocessing

[Wikipedia: Preprocessor](#)

HBBR Basic compiler supports compile time preprocessing directives that allow to customize compilation conditional compilation. Except for compiler defined #Const values all other directives are local to the file being compiled.

Following directives are defined globally by the compiler:

```
HBBRVERSION - compiler version
MODVERSION  - mod file version
CPUID       - CPU id
_FILE_      - name of the file being compiled
_DATE_      - compilation date
_TIME_      - compilation time
```

Preprocessing Directives

Include directive allows inclusion of a file into the file being compiled.

```
#Include file_path
```

Const directive defines new compile time constant visible in the file being compiled.

```
#Const Name = Expression
```

Example:

```
#Const DEBUG = True
```

If Then directive allows to the select parts of the source file to be processed by the compiler.

```
#If Const Expression Then
```

```
#ElseIf Const Expression
```

```
#Else
```

```
#End If
```

Conditional compilation directives

[Wikipedia: Conditional compilation](#)