RITTLE

Reference Manual

Abstract

Description of the programming language Rittle

Contents

Rittle is a new imperative multi-purpose programming language and integrated programming environment. It is algorithmically structured language with syntax and features built as a mixture from several other prominent predecessors such as C, Python, BASIC, Pascal, and others. It also contains some new features not found elsewhere.

The concept is to have only a few but powerful program words and functions.

*By design the Rittle programming environment is an interactive shell composed from Source Editor (**RSE**), Source Compiler (**RSC**), Integrated Debugger (**RID**), and a Virtual Machine (**RVM**), which executes the produced p-code after compilation.*

1 BASICS

1.1 Source Files

Rittle source code is a script consisting of commands, functions, parameters, constants, and expressions in text form. The typical file extension of a Rittle source file is ***.RIT***.

A program can be written in any text editor, and then fed through the RSC it comes out as a compiled pseudo-code in format that RVM understands and executes.

Only a very limited set of non-printable characters are allowed in the source. These are the '*Space*' character (ASCII code 32), '*Horizontal Tab*' (ASCII code 9), '*New Line*' (ASCII code 10), '*Vertical Tab*' (ASCII code 11), and '*Carriage Return*' (ASCII code 13). RSC will exit with an error if any other non-printable character is found in the source during compilation. These characters are called *whitespace*, and are ignored during the compilation.

It is important to remember that once compiled, the file is in binary format and becomes unreadable by normal human standards, so always keep the sources so you can edit your code and generate new compiled output when needed!

The typical file extension of a binary file with compiled Rittle code which is executable by the RVM, is ***.RIX***.

1.2 Comments in the Source

Comments in Rittle are only presented in the input source script, and are stripped by RSC in the output compiled p-code.

There are two types of comments – to the end of the current text line, and multi-line ones spanning from beginning marker to end marker over one or more text lines.

The first type occupies only the text within the current text line. It starts with an apostrophe character **'** and ends along with the text line on which it is.

```
' this is a comment to the end of the current line only
```

The multi-line comments start with a special marker which is an apostrophe character followed immediately by an underscore character. After this point the comment can spread over as many lines as necessary without any limitations. It only ends by reaching a combination opposite to the opening marker – an underscore character, followed by an apostrophe. Since the comment will end at the first found such sequence, nesting of multi-line comments is not allowed.

```
'_ this is a comment that can last as many lines as needed

    This line continues the same comment

    … and this one as well, until the end marker is reached _'
```

1.3 Numeric Constants

Numbers in Rittle can be represented in several different ways:

1.3.1 Binary Numbers

Binary numbers always start with a '***0b***' or '***0B***' sequence, followed by one or more binary digits. Binary digits are only the digits 0 and 1.

So, 0b111011010100 is a valid binary number, but 0b111011210100 is not, since it contains a character which is not a valid binary digit.

The largest binary number currently supported in Rittle can have 64 binary digits.

Additionally it is worth mentioning that binary numbers are always unsigned.

1.3.2 Hexadecimal Numbers

Similar to the binary numbers, hexadecimal numbers in Rittle always start with a '***0x***' or '***0X***' sequence, followed by one or more hexadecimal digits. Hexadecimal digits are the digits from 0 to 9, as well as the letters 'A', 'B', C', D', 'E', and 'F', as well as their lowercase counterparts 'a', 'b', 'c', 'd', 'e', 'f'.

The largest hexadecimal number currently supported in Rittle can have 16 hexadecimal digits.

Hexadecimal numbers are also always unsigned.

1.3.3 Decimal Integer Numbers

These are the most commonly used number as they are represented in the standard decimal format in which people operate in the everyday life.

The decimal integer numbers can be signed (preceded by a '**–**' or '**+**' character), and contain the digits from 0 to 9.

In unsigned form (i.e. if not preceded by a minus character '-') the largest decimal integer number currently supported in Rittle's 64-bit arithmetic is huge: 18446744073709551615

In signed form the numbers can range from -9223372036854775808 to +9223372036854775807

This range of number is sufficient in almost every aspect in the everyday integer arithmetic calculations.

1.3.4 Decimal Real Numbers

These numbers represent the full range of real numbers with decimal point and optional exponent. Their binary representation in the system memory can vary

between the implementations, but in the minimum case they are at least 80 bits long, and able to cover at least the range 3.65×10^{-4951} to 1.18×10^{4932}

Typically the accuracy after the decimal point can go as deep as 17 digits or more.

The format is again an optional minus or plus character, followed by one or more decimal numbers, followed by an optional decimal point and more decimal numbers, after which an optional exponent character '**e**' or '**E**' should be followed again by a standard signed or unsigned decimal number and optional decimal point part.

The Exponent part defines the number in standard scientific exponential format.

In summary the whole real number can be expressed as:

[**-**] **integer** [**.fraction**] [**E** [**-**] **exponent** [**.fraction**]]

Examples of several real numbers in Rittle: *-0.3 6.779 -2E11 9.03E-2.84*

1.4 Symbolic Constants (TEXT)

Symbolic constant in Rittle is defined as an undefined length sequence of 8-bit ASCII symbols enclosed between **"** double quote characters. This data type is called ***TEXT***.

"*This is a text constant***"**

In some cases the text constant may need to include a double quote character, or other characters outside of the standard printable set. Rittle allows specifying any character code from within the text constant without introducing ambiguity over whether it is a closing double quote (in the particular case) or not. These special sequences are called escape codes, and always start with an underscore character followed by one or more other characters to form the needed code.

The pre-defined escape codes in Rittle are listed below:

_a ASCII code 0x07 (alert, beep)

_b ASCII code 0x08 (backspace)

_t ASCII code 0x09 (horizontal tab)

_n ASCII code 0x0a (new line)

_v ASCII code 0x0b (vertical tab)

_f ASCII code 0x0c (form feed)

_r ASCII code 0x0d (carriage return)

_e ASCII code 0x1b (escape)

_" double quote character

__ underscore character

In addition to the pre-defined escape codes, any character code may be included by entering its ASCII code directly as an escape code:

_NNN character with decimal code NNN

_xNN character with hexadecimal code NN

In some example:

"*This text constant includes _"special_" escape codes and a new line_**n***"

In specific cases if a text constant is too long to fit within the current source line, it can be composed as a sequence of more than one text constants that are added together during compilation. In order to achieve this the text constant must be followed immediately by its continuation part preceded by an underscore character.

<u>Important:</u> Nothing except comments is allowed between a text constant and its continuation. Failure of this rule will result in incorrectly compiled output code.

"*This is a text constant*"
Nothing else is allowed between these two lines!
_" *that gets continued here*"

At system level text constants are compiled as part of the program code and represented in the memory as zero-terminated strings. Obviously due to this fact, character with code 0 cannot reside within a text constant.

1.5 Statements

Statement is a piece of code that is telling that something needs to be done. That could be to call some function, to assign value to something, etc.

The important thing to know at this stage is, that every statement in Rittle must be completed with a '**;**' character. A statement ends where the logic meaning of something ends, and something else starts from there.

1.6 Identifiers

Identifiers in Rittle are all words that identify something to use, or something to do – variables, function names, reserved keywords. They all follow the same naming rules for identifiers.

An identifier is a single continuous word with no spaces, must start with a Latin letter or the underscore character, and can contain only Latin letters, digits, and underscore characters.

So, in examples:

A12 record Next_ID _name field7

…are all valid identifiers.

Some invalid identifiers:

9pp *(doesn't start with a letter)* **Error!** *(contains an invalid character)*

…are invalid identifiers.

All identifiers in Rittle are case-sensitive.

That means for example '*data*, '*Data*, and '*datA*, are three different identifiers from language perspective.

1.6.1 Variables

Variable is an identifier which represents some data. It can be read or written, located at a specific location in the memory, or consist of a number of elements of the same type. In the last case the variable is called “array”.

There are several data types in Rittle:

uint8	- 8 bits, unsigned	range 0 … 255
sint8	- 8 bits, signed	range -128 … +127
uint16	- 16 bits, unsigned	range 0 … 65535
sint16	- 16 bits, signed	range -32,768 … +32,767
uint32	- 32 bits, unsigned	range 0 … 4,294,967,295
sint32	- 32 bits, signed	range -2,147,483,6478 … +2,147,483,647
uint64	- 64 bits, unsigned	range 0 … 18,446,744,073,709,551,615
sint64	- 64 bits, signed	range -9,223,372,036,854,775,808 … +9,223,372,036,854,775,807
real	- 80+ bits (depending on the system), floating point	guaranteed minimum range 3.65×10^{-4951} … 1.18×10^{4932}
text	- unlimited sequence of ASCII characters except NUL (ASCII code 0)	
func	- special type used to pass function names as parameters to functions	

The **uint8** type can be also declared in a more human-friendly form as **byte**.
Similarly to that, the **sint8** type can be declared as **small**.
And another alias is available for **sint64** – it can be declared simply as **int**.

Declaring variables is one of the most complex and important aspect of any language, and here that is no exception. The declaration statement is composed from several sections and the overall format is this:

var [*role*] *type name* [**maxlen** *length*] [**array[***size* [**:***size* …]**]**] [**at** *address*] [**,** *name* …] [**=** *value***,** …]**;**

The parameters "*address*" and "*size*" can be only numeric constants or variable names, while "*value*" can be a composite expression.

"*role*" is optional specific term which will be explained later with the functions.

"*type*" is one of the valid data types.

"*name*" is a unique and valid identifier name.

"*maxlen*" is valid only when the type is text. It defines a maximum length for the variable. All operations that produce a longer result will lead to the variable being automatically truncated to the specified maximum length.

The var-statement must be completed with a '**;**' character, just like any other statement in Rittle;

Let's consider the declaration in more details.

The first word **var** informs that one or more variables will be declared in this statement. This is the keyword in Rittle that makes declarations possible.

After **var** follows an optional section, which tells the compiler where that variable is located in the physical memory. Generally this refers to specific hardware registers or areas of memory which are fixed by the hardware. Unless there is a very good reason why a variable should be directed to a fixed address, there is no need to use **at** otherwise.

```
var byte reg at 0x82a0;
```

This example declares a variable with name "*reg*", which is of type **uint8**, and is located at fixed memory address *0x82a0*.

Address specifier works with multiple variables by declaring all of them pointing to the same address. It also works with arrays by specifying the initial address of the array.

Declaring arrays is done by using the word **array**:

```
var real a array [10];
```

This declares an array of 10 elements of type **real** as a variable with name "*a*";

Multi-dimensional arrays are declared in the same way by simply adding the sizes and separating them with a '**:**' character:

var real *a* **array [***10:10:10***] at** *0x400000*;

The statement above declares a three-dimensional array of real numbers (1000 elements in total) as variable with name "*a*", and the array resides in a fixed memory location starting from address 0x400000;

Variables can be also declared in multiples within the same var-statement, if all of them are of the same type. Their names are separated by a '**,**' character:

var uint8 *second***,** *minute***,** *hour***,** *day***,** *month*;

The last section in a var-statement allows variables to be initialised during the declaration. Initialisation values follow after a '**=**' character. In a simple form:

var small *x* **=** *-1*;

This statement declares a variable "*x*" and assigns it a value of -1.

Initialisation of arrays during their declaration is not possible. It is a good practice to have arrays individually declared each in its own "*var*" statement.

1.6.2 Functions

Functions are pieces of code that is called by name, and performs some operation. Optionally functions may need to receive some data from the outside world, and return some other data back after finishing the operation.

Declaring a function in Rittle is done by using two special words: "*func*" and "*endfunc*":

func *name*;
……. something to do …….
……. something to do …….
……. something to do …….
endfunc;

"*name*" is the desired function name. It must be a unique (within the current namespace) and valid identifier.

Nesting of functions is allowed. In such case the nested function names can be the same if declared in different parent functions:

```
func foo1;

    func moo;
      ……. something to do …….
      ……. something to do …….
    endfunc;

  ……. something to do …….
  ……. something to do …….

endfunc;


func foo2;

    func moo;
      ……. something to do …….
      ……. something to do …….
    endfunc;

  ……. something to do …….
  ……. something to do …….

endfunc;
```

In this example both the functions "*foo1*" and "*foo2*" contain a nested function "*moo*", which may be something completely different in the two cases.

Functions use specially declared local variables to receive data from the caller, or to return data back to the caller. These variables define what type of data is expected, and what role that data plays in the function.

There are three types of role for a local variable: "*input*", "*output*", and "*refer*".

Input role is when the variable takes input from the caller. Data is copied into the local variable, and can be used with the body of the function only. The local variable gets destroyed after the function ends.

IMPORTANT! All interface variables "*input*", "*output*", and "*refer*" must be declared before calls to other functions.

Output role is when the variable carries something back to the caller. When the function finishes, the caller gets back information from all output variables.

Refer role is more specific. It is bidirectional (serves as input and output at the same time). No actual local variable is created, but instead, the function gains access directly to the variable supplied by the caller. That external variable is just known under the specified local name.

Variables which are passed to functions as refer-type need to be preceded by a '**@**' character. This instructs the compiler to send to the function a reference to the variable, instead of its data. The same rule is in place for functions passed to func-type variables.

So the bottom line of the roles is: a function may have none or a number of input parameters; may return nothing or several pieces of data to the caller; or may directly read and modify data which belongs to the caller.

In an example:

```
func foo;

   var input integer a, b, c;
   var output integer n, m, p;
   var refer byte k;


endfunc;
```

This function receives two bytes, returns two bytes, and refers to another byte which belongs to the caller.

Calling functions in Rittle is similar to variable initialisation – a function is called with one or more, or none parameters, and the output is assigned to one or more, or none variables:

```
var integer a,b,c = foo(x,y,@z);
```

In this case variable "a" will be assigned with the first output variable from function, variable "b" with the second, etc.

In this example the variable "z" is referred to when calling the function.

The number of receiving variables must match or be greater than the number of output variables in the function.

One important feature of this model, is that Rittle allows a single function to behave differently, based on input conditions. For example a function may have only one fixed input parameter, based on which it may take more input parameters and return different data.

The initialisation statement allows freely mixed constants, variables, and functions:

var integer a,b,c = foo(x,y,z), A, 25;

The last thing to point out is that Rittle does not require function parameters to be enclosed in brackets. It is up to the user to decide when to use brackets for more clarity in the source. From Rittle's perspective the two statements

foo(1,2,3); and ***foo 1,2,3;*** … are exactly the same.

In some cases such as during variable initialisation however, brackets may play an important role to tell the compiler what is a parameter to a function, and what is not.

1.6.3 Reserved Words

Rittle allocates a certain number of words such as "do", "var", "*small*", etc. These reserved words build the language and cannot be used as variable or function names.

1.7 Operations

Operations are generally divided in two main groups: operations with numbers, and operations with text.

Operations are performed by taking into account their level of precedence. Same level operations are executed sequentially in the order they appear in the expression.

Comparisons work with any data type. When performed on text arguments, the operation is performed by comparing the ASCII codes in each argument sequentially.

Some operations have dual function depending on the type of the arguments they work with. The operator "****" for example will perform integer division when working with integer numbers. The same operator however can be used to round a real number:

```
x = 2.75 \ 1          ' this is equivalent to x = 3
```

Arithmetic and logic operations

Operation	Argument type	Description
+	any	Performs addition with numbers Texts are concatenated
-	numeric	Subtraction
*****	numeric	Multiplication
/	numeric	Division
****	numeric	Integer division with integer numbers With real numbers the result is rounded to the nearest integer number
****	numeric	Modulo operation with integer numbers With real numbers the result is only the fraction after the decimal point
^	numeric	Exponentiation
and	integer only	Bitwise AND
or	integer only	Bitwise OR
xor	integer only	Bitwise Exclusive OR
not	numeric	The function is 1 if the argument is 0, and 0 if the argument is not 0
~	numeric	Bitwise negation with integer numbers With real numbers only the sign of the number is inverted
<<	integer only	Bit shift left
>>	integer only	Bit shift right

++	numeric	Increment by 1
--	numeric	Decrement by 1

The "**++**" and "**--**" operations deserve a few more words. These two work with numeric variables only and immediately precede or follow in the source the variable name to perform increment by 1 or decrement by 1, respectively.

When preceding the variable (as in "**++a**") the operation is performed before the value from the variable is taken. Trailing the variable (as in "**a++**") means the variable value modified after it was taken in the statement.

Comparison operations

Operation	Description
==	Returns 1 if the two arguments are equal
<>	Returns 1 if the two arguments are different
<	Returns 1 if argument 1 is smaller than argument 2
<=	Returns 1 if argument 1 is smaller or equal than argument 2
>=	Returns 1 if argument 1 is greater or equal than argument 2
>	Returns 1 if argument 1 is greater than argument 2

By their level of precedence all operations are grouped in this way:

Operation	Level
++ --	highest
~ not	
^	
*** / \ **	
<< >>	
+ -	
== <> < <= => >	
and or xor	lowest

The order of execution the operations can be changed if necessary, by enclosing in **(** brackets **)** the needed sections in an expression.

2 DATA TYPE "TEXT"

2.1 General Information

The "***text***" data type is an important feature of Rittle. It allows easy work with symbolic data represented as a sequence of unlimited length consisting of any 8-bit ASCII code with the exception of code 0. The sequence is terminated by a character with code 0.

There are only a few but powerful functions to operate with text in Rittle.

An important detail to be mentioned: RVM performs all the necessary operations to allocate the needed memory and transfer the data when working with text data, so from user's perspective text data has no limit in length. The developer can specify the **maximum allowed length** for text data, though. RVM will take care to ensure that the length of the variable during execution never goes beyond a specified value.

Arrays of text data type are allowed and work in the same way as numeric arrays.

2.2 Operations

As basic operations can be considered those such as conjunction, measuring, and code extraction.

Two texts can be conjoined by using the "**+**" operation:

```
var text s1,s2,s3;

……

s1=s2+s3;    ' s1 becomes a text which contains the conjoined s2 and s3 result
```

The function measuring the length of a text returns the number of characters in it (except the closing 0, which is invisible by the user).

Provided is also a function for searching a text within another text, starting from a specified character index. Another function cuts and returns part of a text.

The most interesting functionality however, and also the most useful one, is the text formatting.

2.3 Text Formatting

Text formatting in Rittle is performed by the function "*format*":

format *fmt* [**,** *parameter1***,** *parameter2***,** …]

The function takes undefined number of parameters (minimum one – the format specifications), and returns a single text which can be assigned to a variable, output, etc.

The function name "*format*" can be replaced with its alias in Rittle – the character '**$**'.

The format specification text is the only mandatory parameter. It is a text, which also may contain instructions to take more parameters. The formed output text is taken from the input "*fmt*" with all references to parameters processed and replaced with their results.

Format instructions in the input text always start with a '**|**' character, and have the following general format:

| *type* [*modifiers* [*length* [.*precision*]] [*modifiers*]]

It is important to note that no spaces are allowed within a format instruction (except for specifying fill characters which will be discussed later).

"***type***" is a single character which specifies the data type of the parameter:

'**d**' or '**D**' Decimal number
Only decimal numbers can have the "*.precision*" part of the instruction.

'**x**' or '**X**' Hexadecimal number

The case register of the type determines the case register of the hexadecimal letters.

'**b**' or '**B**' Binary number

'**t**' or '**T**' Text

Modifiers are characters which instruct how the output result should be formatted. Some modifiers can only work with specific data types, or require explicitly defined length.

Supported modifiers:

'**+**'	Forced sign	(works only with decimal numbers)
'**-**'	Space for '-' sign	(works only with decimal numbers)
'**<**'	Left-aligned result	(require specified field length)
'**>**'	Right-aligned result	(require specified field length)
'**^**'	Centred result	(require specified field length)
'*****'	Fill with characters	(require specified field length) (followed by a mandatory character which specifies the fill character)

The modifier '*****' can be found twice within an instruction. The first occurrence specifies the fill character before the result. The second occurrence specifies the fill character after the result. Both fill characters are considered as space by default, unless changed with the modifier.

The "*length*" part defines the **minimum** total size of the output field. If the parameter is numeric and the suggested length is insufficient to accommodate it, then the output field automatically beyond the defined length value. Text data however does not expand beyond specified length, but is trimmed to the specification.

Without defined length the output will be as long as it needs to be.

Decimal numbers may also need to contain digits after a decimal point. To specify the number of required digits in the output, the part "**.***precision*" is used.

Both "*length*" and "**.***precision*" (whenever used) must be integer decimal numbers greater than 0.

If "*length*" is missing, but "*precision*" is present in the format specification, the number is output with the needed number of digits before the decimal point, and only the specified number of digits after it.

Examples:

cout $"Current time: |d*02>:|d*02>:|d*02>", h, m, s; *' print time hh:mm:ss*

var text s= $"|t^80", "TITLE LINE"; *' the text "TITLE LINE" centred within an*
' 80-character field

var text s= $"Amount: |>10.2, New: |>-10.2", total, new *' create formatted*
' text with numbers

3 PROGRAM CONTROL STRUCTURES

3.1 Conditional Branch

Conditional statements use the traditional structure in many programming languages "*if*" … "*else*" … "*endif*".

In Rittle this structure is expanded by allowing multiple "*else*" statements with their own conditional expressions.

The full format is (unlimited number of "*else*" statements is allowed):

if *condition1*;

……. here if condition1 is true …….

else *condition2*;

……. here if condition2 is true …….

else *condition3*;

……. here if condition3 is true …….

……

else *conditionX*;

……. here if conditionX is true …….

else;

……. here otherwise …….

endif;

Note the semi-colon '**;**' characters completing every statement. Without a semi-colon in the proper place, a statement may come to a completely different meaning during execution, or generate an error during compilation.

All "*else*" branches are optional. The opening "*if*" and the closing "*endif*" however must be present in every structure.

The final "*else*" may have no condition but still has the statement closing semi-colon. It catches all cases uncovered by any other branch in the structure.

3.2 Loop

Unlike other programming languages, Rittle offers only a single loop structure. It is however flexible enough to cover all needs.

The format it:

do [*condition1*];

……. body of the loop structure …….

until [*condition2*];

The loop can have none, one, or two conditions. Note again the closing semi-colons after the conditions. These semi-colons are required even if there is no conditional expression.

Condition1 gets checked before the body of the loop. If true, the loop continues with the iteration. If false the loop ends and continues with the code after "*until*".

Condition2 gets checked after the body of the loop. If false the loop closes and returns back to the "*do*" (and to check again *condition1*). If condition2 is true, the loop ends and the execution continues after it.

If both conditions are missing, the loop becomes an infinite loop, and can be stopped only with an "***exitdo***" command.

An example of a loop performing 100 iterations:

```
var byte a=0;
do a<100;
   ……. Something to do here …….
   a=a+1;
until;
```

3.3 Exit and Redo

There are two types of exit in Rittle – exit from function and exit from loop.

"**exitfunc**" allows an early exit from function. It can be anywhere in the function's body and in as many places as needed.

"**end**" similar to "*exitfunc*" but used only in the main program body outside of any function. Forces the program to finish execution.

The command "end" cannot be used within a function and will generate an error during compilation. Only the main program can end itself.

"**exitdo**" offers an early exit from a do…until loop structure.

"**redo**" in a do…until loop structure, forces an early return back to the opening "*do*" without having reached the corresponding "*until*".

3.4 Labels

Rittle allows placing labels in the source text. A label in Rittle is a single valid identifier starting with a '**!**' character. It is a marked place in the program, where the execution can be taken by using the name of that marked place.

Jumping to a label is done by placing its name as a normal word. Any time a word which is a known label, is reached, the execution unconditionally jumps to the spot marked with that label.

Here is a small example of a loop created by using a jump label:

```
var byte a=0;
!loop
   ……. Something to do here …….
   a=a+1;
if a<10; loop; endif;
```

Note that there is no semi-colon character after the label. This is because the label itself does not make a valid statement, but rather marks the beginning of one. Jumping to a label however, is a valid statement in its own right because it instructs a certain action to be taken. Hence it does have a closing semi-colon.

4 REFERENCE OF THE BUILT-IN FUNCTIONS

4.1 Mathematical Functions

4.1.1 sin

Format:

real= sin (real_x);

Calculate and return sin(x). The parameter should be in radians.

4.1.2 cos

Format:

real= cos (real_x);

Calculate and return cos(x). The parameter should be in radians.

4.1.3 tan

Format:

real= tan (real_x);

Calculate and return tan(x). The parameter should be in radians.

4.1.4 asin

Format:

real= asin (real_x);

Calculate and return arcsin(x). The parameter should be in radians.

4.1.5 acos

Format:

real= acos (real_x);

Calculate and return arccos(x). The parameter should be in radians.

4.1.6 atan

Format:

real= atan (real_x);

Calculate and return arctan(x). The parameter should be in radians.

4.1.7 hsin

Format:

real= hsin (real_x);

Hyperbolic sine function.

4.1.8 htan

Format:

real= htan (real_x);

Hyperbolic tangent function.

4.1.9 trim

Format:

integer= trim (real_x);

Return the integer part of x.

4.1.10 abs

Format:

real= abs (real_x);

Return the absolute value of x.

4.1.11 sign

Format:

integer= sign (real_x);

Return 1, if x>0
Return 0, if x=0
Return -1, if x<0

4.1.12 deg

Format:

real= deg (real_x);

Convert x from radians to degrees.

4.1.13 rad

Format:

real= rad (real_x);

Convert x from degrees to radians.

4.1.14 log

Format:

real= log (real_x);

Decimal logarithm of x.

4.1.15 ln

Format:

real= ln (real_x);

Natural logarithm (base ‘e’) of x.

4.1.16 exp

Format:

real= exp (real_x);

Natural exponent e^x.

4.1.17 E

Format:

real= E;

Return the value of ‘e’.

4.1.18 PI

Format:

real= PI;

Return the value of π.

4.1.19 random

Format:

real= random;

Return random number between 0 and 1. The number may be 0, but is never 1.

4.2 Text Functions

4.2.1 val

Format:

real= val (text_arg);

Return the numerical value of the argument.

Assuming that the argument is a decimal number represented in text form.

4.2.2 format

Format:

text= format (text_fmts [, any, any, …]);

Formatting function.
See Chapter "*Text Formatting*".

4.2.3 sim

Format:

real= sim (text_arg1, text_arg2);

Return the level of similarity of two texts, as number between 0 and 1.

4.2.4 search

Format:

integer= search (text_what, text_where, integer_index);

Search for the first occurrence of text parameter "what" in text parameter "*where*", starting from position "*index*".
The valid range for "*index*" starts from 0.
Will return index of the beginning of the found occurrence, or -1, if none if found.

4.2.5 insert

Format:

insert (text_what, text_where, integer_index);

Insert the text parameter "what" into text parameter "*where*", starting from position "*index*".
The valid range for "*index*" starts from 0.

4.2.6 char

Format:

text= char (integer_ascii);

Return a single-character text with the character representation of ASCII code x.

4.2.7 code

Format:

integer= code (text_arg);

Return the ASCII code of the first character from the text argument.

4.2.8 cut

Format:

text_cut [, text_remainder]= cut (text, integer_begin, integer_count);

Cut and return substring from the text parameter s, starting from index "*begin*" and "*count*" characters long. Optionally also returns the remaining portion of the text argument without the cut part.

4.3 Files and Communications

Currently supported devices:

dsx**:** Data storage device.
The file name may be preceded by path e.g. ***ds1:/dir/dir/file***

con: System console ***(com0:115200,8N1)***. Always open and initialised.

comx**:** Serial communication through UART port.

spix**:** Serial communication through SPI port.

iicx**:** Serial communication through I^2C port.

canx**:** Serial communication through CAN bus port.

4.3.1 init

Format:

integer= init (text_devspecs)

Initialise device and return size of the available data after initialisation, or -1 in case of an error.
In case of data storage this function will destroy all data currently on the device.
In case of serial communication devices, it will reset the communication buffers.

4.3.2 delete

Format:

integer= delete (text_filespecs)

Delete specified file from data storage device. Return 1 if successful, or 0 otherwise.

4.3.3 open

Format:

integer_handler= open (text_filespecs);

Open file or communication device. The "*filespecs*" parameter contain all the necessary information for the operation in format:
For data storage devices: "***device:filename***"
For communication devices: "***device:parameters***"
File handler number is returned in case of success, or -1 in case of a failure.

4.3.4 close

Format:

close (integer_handler);

Close file/device with provided handler.

4.3.5 isopen

Format:

integer= isopen (integer_handler);

Return device number, if the provided handler is open, or -1 otherwise. A device number is greater or equal to 0 and can indicate the type of the open device.

4.3.6 pos

Format:

integer= pos (integer_handler);

Return the current data pointer position within an open file with provided handler, otherwise return -1.

4.3.7 eod

Format:

integer= eod (integer_handler);

Return 1, the data pointer has reached the end of data stream with the provided handler, or return 0 otherwise.

4.3.8 ioerr

Format:

integer= ioerr (integer_handler);

Return error code for the specified handler, or 0 if there is no error. The error is cleared after that.

4.3.9 seek

Format:

integer= seek (integer_handler, integer_position);

Move the data pointer to the specified position and return 1 if successful, and 0 otherwise.

4.3.10 write

Format:

integer= write (integer_handler, any [, any, any, …]);

Write data into open file with provided handler. Return the number of written bytes, or -1 in case of an error.

4.3.11 read

Format:

variable [, variable, variable, …]= read (integer_handler);

Read data from open file with the provided handler, and store it into the designated variables.

4.3.12 ffind

Format:

text= ffind (path, integer)

Return the name of the **n**th file in the specified path skipping given number of files before it. Skipping indexes start from 0.
If there are not enough files in the path, or the skipping index is a negative number, the function will return a blank text.
If the function is attempted on a device which is not a data storage device, it will return a blank text.

4.3.13 mkdir

Format:

integer= mkdir (path)

Make a new directory and return 1 if successful, or 0 otherwise.

This function can be executed on data storage devices only. It will return 0 if an attempt is made to use it in other devices.

4.3.14 rmdir

Format:

integer= rmdir (path)

Remove directory and return 1 if successful, or 0 otherwise. The directory must be empty in order to be removed.
This function can be executed on data storage devices only. It will return 0 if an attempt is made to use it in other devices.

4.3.15 chdir

Format:

integer= chdir (path)

Change the current directory directory and return 1 if successful, or 0 otherwise.
This function can be executed on data storage devices only. It will return 0 if an attempt is made to use it in other devices.

4.4 Others

4.4.1 include

Format:

include file, file, ...;

Include specified Rittle source files during compilation.

Not an actual function but instruction to RSC to include source code from external text files into the compilation process. Does not produce own executing code.

The inclusion happens at the exact spot where "***include***" is placed. If more than one file is specified, all are included and compiled sequentially.

4.4.2 size

Format:

integer= size (any);

Return the size of the argument in bytes. If the argument is text, then will return the length of the text.

4.4.3 isval

Format:

integer= isval (any);

Return 1, if the argument is a number, otherwise return 0.
If the argument is supplied as text, then an assessment is made whether the text contains a representation of a number.

4.4.4 cout

Format:

cout (any [, any, any, …]);

Output of one or more parameters to the console. The parameters can be variables, constants, or expressions. The type of the parameters determines automatically the form in which data is presented – texts are output directly, numbers are represented in their decimal text form.

4.4.5 cin

Format:

variable[, variable, variable, …]= cin;

One or more variables are read from the console. The natural output from ***cin*** is text, if inputting numbers is required, then the ***val()*** function should be used additionally in the form *"**val(cin)**"*. Reading is terminated by ASCII code 13 (CR).

4.4.6 cchr

Format:

variable[, variable, variable, …]= cchr;

Return the next single character from the console input buffer, or return blank string if the buffer is empty.

5 EXTREME PROGRAMMING WITH RITTLE

Extreme programming is techniques to squeeze out the maximum possible from a programming language. Due to its very loose syntax rules, Rittle is quite welcoming towards extreme programming. These techniques can save code and in some cases speed up the execution a little, but in general they also make the code difficult for understanding and maintenance. A careful balance between extreme techniques and standard syntax structures is the best general approach. Here is a short summary of some of the possible extreme programming techniques in Rittle.

5.1 Group Assignments

Rittle has a very specific feature allowing multiple assignments to be made within a single statement.

In some examples:

var small *x, y, z* **=** *-1*;

This will declare "*x*", "*y*", and "*z*", and will assign value -1 to all of them.

var small *x, y, z* **=** *-1, -2, -3*;

This will declare "*x*", "*y*", and "*z*", and will assign value -1 to "*x*", value -2 to "*y*", and value -3 to "*z*".

There is also the possibility to skip the initialisation of some variables, while skip assigning values to the others:

var small *x, y, z* **=** *-1, , -3*;

This statement will still initialise "*x*" and "*z*", but will do nothing about "*y*".

The same is valid in statements outside of declaration, as well as other variables, functions, and expressions. Even reusing variables within the same statement is allowed:

x, y, z = foo(1,2), (a+1), (y-2);

In the most general case any number of variables can be sequentially initialised with any number of values as long as the number of values is not greater than the number of variables to be initialised. If the number of values is smaller than the number of variables, all remaining variables will be initialised with the last value from the list of initialising values.

Since the assignment is performed sequentially, it is possible to use group assignment to perform various programming "tricks":

x, y = y, x; *' swap the values of x and y without using a third variable*

a, b, c = b, c, a; *' rotate values in three variables*

5.2 Multiple Comparisons

Rittle allows more complex comparison expressions consisting of more than two parameters:

if 1< = x <= 10 > y;

is completely valid and translates as "if 1<=x and x<=10 and 10>y".

5.3 Increments and Decrements

Using both pre-modifier and post-modifier on the same variable at the same time is completely valid and is allowed.

For example, the statement

b++=++a++;

is valid, and is equivalent to the sequence: "*b=a+2, a=a+1*".

Even pre-increment (not post-increment, though) of numeric constants is allowed when they are used in expressions:

++b++=++1+(++a--)+(++1);

5.4 Morphing Functions

Morphing functions exhibit different behaviour depending on some input condition.

Within a function a few different blocks of declarations with "*input*", "*output*", and "*refer*" type variables, can exist, and the execution to branch to a particular part in the function, and that based on an external condition.

In an example:

```
func foo;
   var input byte form;        ‘ this parameter will define the behaviour

   if form==0;
      var input byte a, b, c;  ‘ the function takes three byte values
      var output real z;       ‘ and returns one real
     ……
     ……
```

```
   else form==1;
      var input text s1, s2;    ‘ the function takes two texts
      var output real t1, t2;   ‘ and returns two texts
      ……
      ……

   else;
      var input int x, y;          ‘ the function takes two integers
      var refer real p array[10];    ‘ and refers to array
      ……
      ……

   endif;
endfunc;
```

5.5 Nested Functions

Rittle allows functions to be nested within other functions.

In the example below both functions ***myIntFunc1*** and ***myIntFunc2*** exist only within the scope of ***myFunc***.

```
func myFunc;

   func myIntFunc1;
```

```
    ……..
    ……..
endfunc;

func myIntFunc2;
    ……..
    ……..
endfunc;

……..
……..
endfunc;
```

5.6 Function Variables

Function variables are special type variables that refer to actual functions. With function variables a function can be passed as argument to another function, array of function pointers can be created (call tables), or a function name can be aliased for other purposes.

Physically the function variables are integer variables that contain the memory address of the referred function. When a function variable is written to, an integer value is stored. That value is assumed to be a start address of a function. When read, a function variable simply calls the referred function.

5.6.1 Declaring Function Variables

Function variables are declared in the normal way for declaring variables. They are of type "***func***".

```
var func myvar = @foo;
```

This example declares a function variable "***myvar***" and assigns it the address of the function ***foo()***. Note the character '**@**' in the assignment. It tells the RSC that the word is about the address of the function foo().

Without the leading '@' character, the function will be simply executed during the initialisation phase, and its result (if any) will be assigned to myvar thus resulting it pointing to a completely different address.

5.6.2 Using Function Variables

In the previous example a function variable named "***myvar***" was declared, and initialised to point to the function ***foo()***. Anywhere in the program from that point further, the name "myvar" will cause the function foo() to be called.

If at certain point myvar needs to point to another function, it can be re-initialised:

myvar = @newfoo;

This will change "***myvar***" to point to the function ***newfoo()*** instead.

6 LICENSE CONDITIONS

25 September 2017

© Konstantin Dimitrov
knivd@me.com