RITTLE

Reference Manual

Abstract

Description of the programming language Rittle

Contents

1 NEWS AND UPDATES

Join the Rittle group

Rittle for PIC32MZ

Rittle for Windows *(limited demo with only core functionality)*

March 2020

Rittle for PIC32MZ:

Added COM5.

September 2019

Rittle core:

Closing **;** *characters in source lines are now optional, not mandatory.*

July 2019

Rittle core:

Improved experience and better functionality in RIDE.

Virtual cursor in displays.

Added sector() command.

Rittle for PIC32MZ (in addition to the core functionality):

Introduced driver support for touch panels.

Support for XPT2046/TSC2046 resistive touch panels.

June 2019

Rittle core:

Introduced standard interface driver model.

Added graphic user interface primitives such as pixel(), rect(), and others.

Functions for drawing complex vectored shapes (turtle graphics) and filling areas.

Support for user-defined fixed and variable width fonts with character size up to 255x255 pixels.

Font smoothing at higher scaling factors.

Platform-dependent function codes finalised in the region 0xC0 … 0xEF. Binary executables with platform-dependent functions from versions before J6.3, will be incompatible without new compilation.

Rittle for PIC32MZ (in addition to the core functionality):

Introduced support for small standardised SPI displays with resolutions from 32x32 pixels to 480x480 pixels, and 24-bit colour.

Display rotation and mirroring – with appropriate custom fonts, support for left-to-right and right-to-left scripting.

Support for 4-bit HD44780-compatible alpha-numeric LCD modules.

Improved power consumption profile.

May 2019

First public demo release and activation of the website http://rittle.org as well as the public forum.

Alpha stage release for PIC32MZ and PC.

April 2017

Earliest RSC and RVM. Version G4.1-alpha.

February 2017

Development started.

2 INTRODUCTION

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, BASIC, Python, and others. It also contains some new features not found elsewhere.

Download this documentation as PDF

The concept behind Rittle, is to build a fully integrated framework for small systems and microcontroller, where development can be done in a simpler, more streamlined way, similar to what could be found in the computers of the late 20th century. Rittle however, is approaching the problem from a modern perspective with the inclusion of language structures and features used in the mainstream languages of recent time.

By design the Rittle programming environment is an interactive shell composed from a Source Compiler (**RSC**), a lightweight Virtual Machine (**RVM**), which executes the produced p-code after compilation, and a basic console-based integrated development environment and text editor, called **RIDE** (Rittle Integrated Development Environment).

3 BASICS

3.1 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 ***.RXE***.

3.2 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 **_'**

A very specific case is when the developer wants to leave some comments embedded in the output compiled code.

In such cases there should be an exclamation mark immediately following the apostrophe: **'!** or **'!_**

3.3 Numeric Constants

Numbers in Rittle can be represented in several different ways:

3.3.1 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.

3.3.2 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.

3.3.3 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.

The integer decimal numbers currently supported in Rittle’s 64-bit arithmetic cover from -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

This range of numbers is enough for almost every aspect in the everyday integer arithmetic calculations.

3.3.4 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 is standardised by IEEE-754 as 64 bits long, and able to cover at least the range from 10^{-308} to 10^{308}

Typically, the accuracy after the decimal point can go as deep as 16 digits.

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*

3.4 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.

Important: 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.

3.5 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.

If a source line contains only one statement, then the closing '**;**' character is optional and can be omitted.

3.6 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.

3.6.1 Variables and Data Types

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 several elements of the same type. In the last case the variable is called "array".

There are several data types in Rittle:

byte	- 8 bits, integer	***range -128 … +128***
small	- 16 bits, integer	***range -32,768 … +32,767***
big	- 64 bits, integer	***range** -9,223,372,036,854,775,808 … +9,223,372,036,854,775,807*

real	- 64 bits, floating point ***range*** $\boldsymbol{10^{-308} \ldots 10^{308}}$
text	- unlimited sequence of 8-bit ASCII characters ending with NUL (ASCII code 0)
func	- special type used to pass function names as parameters to functions
any	- morphing into any supported data type

The **small** type can be declared also as **sint16**. Similarly, the **int** type can be declared as **sint32**. And **big** can be declared as **sint64**.
And finally, the **byte** type can be declared with the more technical **sint8**.

In a typical scenario the types "byte" and "small" have practical meaning only when applied to variables that are directly linked to hardware (i.e. MCU registers for instance). Using them elsewhere in a program offers no considerable improvement in speed or memory consumption, but could be a potential source of problems coming from erroneous use of numbers outside of their limited range.

Respectively the type "big" should be used when there is an actual need to work with exceptionally big integer numbers. Therefore, for the majority of application scenarios, especially with the popular 32-bit system architectures, the type "**int**" is the recommended type for work with integer numbers.

The type "**any**" is most flexible. It can represent data in any other type depending on the context.

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

var [*role*] *type name* [**[***size* [**:***size* …]**]**] [**maxlen [***length***]**] [**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. Note that the length parameter also needs to be enclosed in square brackets. 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 **byte** 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 indicating the number of elements in the array:

var real *a* **[***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* **[***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;

It is important to remember that <u>indexes for arrays always start from 0</u>. Thus, an array with 10 elements will have valid indexes [0] through [9]. The same applies to multi-dimensional arrays – the index for any dimension always starts from 0.

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 small** *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 int** *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.

3.6.2 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 the 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 also conform to the requirements of a 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 in the function …….
        ……. something to do in the function …….
     endfunc;

   ……. something else to do …….

endfunc;


func foo2;

     func moo;
        ……. something to do in the function …….
        ……. something to do in the function …….
     endfunc;

   ……. something else 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.

When referring to arrays the variable name must be followed by “**[]**”. As an example: referring to a single variable “**x**” would be “**@x**”, whereas referring to an array with name “**x**” would be “**@x[]**”.

It is an important detail to point out that referencing variables always points to the variable itself, but not to a single element in it, in case of arrays. Hence the forms **@var** (for single variables) and **@var[]** (for arrays), are valid, but the form ~~**@var[index]**~~, is not.

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 int a, b, c;
   var output int n, m, p;
   var refer small k;

         ……. something to do in the function …….
         ……. something to do in the function …….

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 int 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 int 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.

3.6.3 Reserved Words

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

3.7 Operations

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

Operations are performed by considering 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 extra words. These two work with numeric variables only, and immediately precede or follow in the source the variable name, on which will perform increment by 1 or decrement by 1, respectively.

When preceding the variable (as in "**++a**") the operation is performed <u>before</u> the value from the variable is taken.

Trailing the variable (as in "**a++**") means the variable value modified <u>after</u> 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.

4 DATA TYPE "TEXT"

4.1 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.

4.2 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 the most useful one, is the text formatting.

4.3 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:

print $”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

5 UNITS

5.1 Defining Data Structures

Rittle enables the creation of more complex data types and data structures called “*units*”. They can be created by using the words “***unit***” and “***endunit***” to enclose one or more variables in a structure that from that moment on, behaves as a single variable. In an example:

unit person;

    var text name;

    var byte age;

endunit;

This definition creates a data structure called “person” with two fields – for name and for age. Then the structure can be used or assigned to other variables in a normal manner just like other data types:

var person sender, recipient;

will create two record of type “person” and names “sender” and “recipient”.

The individual fields within a definition are accessed through their local names following the name of the definition and a ‘**.**’ character:

sender.name

sender.age

A structure may be formed of lower level structures. For example:

```
unit mail;
    person sender;
    person recipient;
endunit;
```

This definition combines both persons from above into a single record "***mail***". Accessing the data fields in this case will follow the levels of the structure. For example: ***mail.sender.name*** will refer to the text field "***name***" in sub-definition "***sender***".

5.2 Structural Arrays

Units can be combined in arrays just like the other single data types. A unit structure can be declared as a single variable array just like any normal variable:

```
unit something[10];
    var int member1;
    var text member2 maxlen [30];
endunit;
```

The example above creates a 10-element array of units with common name "*something*". The access to individual elements in a unit has a standard format:

unit_name . member *[* ***. member*** *…]* ***[index*** *[****: index*** *…]* ***]***

In an example with the unit from above:

something.member1[3] will refer to the variable "member1" in the fourth unit structure (arrays always start from index 0).

When a unit is used as data type for another variable, which is array, then the number of dimensions are getting increased accordingly. For example:

var something newer[15];

This will crate a unit structure “newer” built from unit structures “something”. But in the previous example “something” is actually an array of 10 elements, so the new variable “newer” will inherit the original dimension, and add its own on top of it. As result the new variable will have two dimensions: [10] inherited from “something” and [15] as declared in “newer”. Therefore accessing elements will require two indexes:

newer.member1[3:7]

In case internal member variables in a unit also have their own dimensions, those are added as junior level indexes when accessing the member variables:

Let’s consider this unit:

```
unit pack;
    var int ival[12];
endunit;
```

var pack p[20];

The newly defined variable is built from 20 units of type “pack”, each one of which containing a single member array “ival[12]”.

Accessing “p” would look like this: ***p.ival [****p_index* ***:*** *ival_index****]***

Generally speaking indexes are always added by seniority of declaration – units first, then member units (if any), top-level indexes, and finally individual indexes.

Working with units containing too many indexes may become very confusing, so as a rule of thumb it is always good to optimise and keep the number of needed dimensions as low as possible. Internally the RVM always convers all multi-dimensional arrays to a single-dimension flat array, so from developer’s perspective maintaining a close proximity to how data is actually being stored in memory can be only beneficial for the reliability of the program.

6 CONSTANT DATA ARRAYS

Constant data arrays are data pieces embedded directly into the program code so they can be always available to the program. They also occupy almost no memory in the RAM thus leaving more for actual data.

Data constants are defined in a way somewhat similar to variables but much simpler:

data *type name* **=** *value***,** …**;**

The only required information for the definition is the data type and the name of the data constant. The compilator does the rest by automatically sizing the array according to the number of following values.

The initialisation part after the '**=**' sign, is mandatory.

Once defined, a constant data array is accessible in the same way as a one-dimensional array from variables of the defined type. And again, like all arrays in Rittle, indexes always start from 0.

Constant data arrays are read-only. Writing is not permitted. An attempt to do so will generate a program execution error.

Example:

data text days = "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday";

This example will create a constant data array of text type with name "***days***", and it will contain the names of all days in the week. Accessing element [0] for instance, will return "Monday", element [5] will return "Saturday", etc.

Another example:

data byte month_days = 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31;

This constant data array defines 12 elements with the number of days for every month in a year. Accessing "month_days[4]" will return 31 (the number of days in May), etc.

By using the type “any” constant data arrays can be made of mixed type. They can be read out in dedicated variables of the specified type, or in a variable of type “*any*”, and then processed further with the help of the function **type()**:

```
data any people = “Annie”, 24, “Michael”, 46, “William”, 51, “Catherine”, 39;
var any temp;
var int index=0;
while index < count(@people[]);
        temp = people[index++];
        if type(temp)==30;          ‘ the name of the person
                print “Name: “,temp,”_r_n”;
        else;                       ‘ the age of the person
                print “Age: “,temp,”_r_n”;
        endif;
until;
```

It is important to point out that constant data arrays can only be built from single data type constants but not from units.

7 PROGRAM CONTROL STRUCTURES

7.1 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.

7.2 Loop

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

The format it:

while [*condition1*]; *' Repeating the loop while this condition is true*

……. body of the loop structure …….

```
until [condition2];        ' Repeating the loop until this condition become true
```

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 "*while*" (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 "***exitloop***" command.

An example of a loop performing 100 iterations:

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

The popular *"for()"* programming structure in other languages can be easily also expressed in Rittle. An example below is for a loop with 1000 iterations:

```
var small a=1000; while(a--);
   ……. Something to do here …….
until;
```

7.3 Exit and Repeat

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.

“**exitloop**” offers an early exit from a while…until loop structure.

“**repeat**” in a while…until loop structure, forces an early return to the opening “*while*” without having reached the corresponding “*until*”.

7.4 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 small 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.

8 PARALLEL PROCESSES

Rittle offers multitasking at ground level. The RVM takes care of background work while from programmer's perspective the functionality is contained within only two reserved words – "**pproc**" and "**ppterm**".

In an example:

```
func task1;
   ……. Something to do here …….
endfunc;

pproc task1;          ' run task1 as a parallel process
   ……. Main body continues here while task1 now works in parallel …….
```

As it can be seen, parallel processes are normal functions (which can still be executed as such). The word "**pproc**" creates a separate time-sharing process in which the function with name following "pproc" runs. Once the corresponding "**endfunc**" is reached, the parallel process terminates just like a normal function exit. A function running as parallel process can still execute any other function (including itself), as well as contain nested functions. It can also start other parallel processes.

A parallel process can terminate itself in the standard ways to end a function. It is also possible the process to be terminated from the main program body (only) by using the word "**ppterm**" followed by the name of the process' entry function as specified in "**pproc**". An attempt to terminate a process from within another process will cause the program to exit with an error message.

NOTE: The main program body has priority over all other parallel processes. Once the main body finishes execution, all other parallel processes are also automatically terminated.

9 REFERENCE OF THE BUILT-IN FUNCTIONS

Full list of the built-in core functions in Rittle. Note that these are only the hardware-independent functions, while any others specific for a particular platform are described separately.

9.1 Mathematical Functions

9.1.1 sin

Format:

real= sin (real_x);

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

9.1.2 cos

Format:

real= cos (real_x);

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

9.1.3 tan

Format:

real= tan (real_x);

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

9.1.4 asin

Format:

real= asin (real_x);

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

9.1.5 acos

Format:

real= acos (real_x);

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

9.1.6 atan

Format:

real= atan (real_x);

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

9.1.7 hsin

Format:

real= hsin (real_x);

Hyperbolic sine function.

9.1.8 htan

Format:

real= htan (real_x);

Hyperbolic tangent function.

9.1.9 trim

Format:

integer= trim (real_x);

Return the integer part of x.

9.1.10 abs

Format:

real= abs (real_x);

Return the absolute value of x.

9.1.11 sign

Format:

integer= sign (real_x);

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

9.1.12 deg

Format:

real= deg (real_x);

Convert x from radians to degrees.

9.1.13 rad

Format:

real= rad (real_x);

Convert x from degrees to radians.

9.1.14 log

Format:

real= log (real_x);

Decimal logarithm of x.

9.1.15 ln

Format:

real= ln (real_x);

Natural logarithm (base ‘e’) of x.

9.1.16 exp

Format:

real= exp (real_x);

Natural exponent e^x.

9.1.17 E

Format:

real= E;

Return the value of ‘e’.

9.1.18 PI

Format:

real= PI;

Return the value of π.

9.1.19 random

Format:

real= random;

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

9.2 Text Functions

9.2.1 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.

9.2.2 format

Format:

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

Formatting function.
See Chapter “***Text Formatting***”.

9.2.3 sim

Format:

real= sim (text_arg1, text_arg2);

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

9.2.4 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.

9.2.5 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.

9.2.6 char

Format:

text= char (integer_ascii);

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

9.2.7 code

Format:

integer= code (text_arg, integer_index);

Return the ASCII code of the character pointed by the index, from the text argument.

9.2.8 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.

9.3 Files and File Storage Devices

Currently supported file devices are:

IFS:	Internal file storage. This volume uses the part of the internal non-volatile memory in the system. The **IFS:** drive is always present and Rittle mounts it as default.
RAM:	Internal RAM storage. Uses part of the system RAM to organise a drive which is available for sharing information between programs, or to store temporary data. The information in the **RAM:** drive is destroyed on reset or power loss, and the system normally starts without it. In case it is requires, it can be initialised at any time to make it available.
SD*x*:	***(x=1,…)*** External (SD card) storage using an externally connected SD card.

File names may be preceded by path e.g. ***sd1:/dir/dir/file***

It is important to know that in Rittle all file functions work with the currently mounted drive. The only exception to this is ***copy()*** which can work with two drives simultaneously.

9.3.1 init

Format:

integer= init (text_devspecs);

Initialise device and return size of the available data after initialisation, or a negative number in case of an error.
In case of data storage this function will destroy all data currently on the device.

9.3.2 mount

Format:

integer= mount (text_devspecs);

Make specified storage drive current for operations. Return 0 in case of success, or a negative number in case of an error.

9.3.3 where

Format:

text= where;

Return the current drive and directory set for file operations. Will return blank string in case of an error.

9.3.4 delete

Format:

integer= delete (text_filespecs);

Delete specified file from data storage device. Return 0 if successful, or a negative value result, otherwise.

9.3.5 rename

Format:

integer= rename (text_filecurr, text_filenew);

Change the name of a file or move it to new directory. Return 0 if successful, or a negative value result, otherwise.

9.3.6 copy

Format:

integer= copy (text_file, text_path);

Copy file from the current file storage device to a new place specified in the 'path' parameter. The parameter may include a drive and directory. File name must be included in the path regardless of whether it is the same or different. Return 0 if successful, or a negative value result, otherwise.

9.3.7 open

Format:

integer_handler= open (text_filespecs);

Open file for read and write operations. If the file already exists, it is open for appending data and the file pointer is positioned at the end of the existing data. If the file does not exist, it is automatically created and the file pointer is positioned at the beginning of the file.
The "*filespecs*" parameter specifies path and name: "***device:/dir/…/filename***"
File handler number is returned in case of success, or a negative value in case of a failure.

9.3.8 close

Format:

integer= close (integer_handler);

Close file with provided handler. A successful closing will return result 0, otherwise a negative value result will indicate an error.

9.3.9 isopen

Format:

integer= isopen (integer_handler);

Return 1, if the provided file handler is open, or 0 otherwise.

9.3.10 fpos

Format:

integer= fpos (integer_handler);

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

9.3.11 eof

Format:

integer= eof (integer_handler);

Return 1, the data pointer has reached the end of data stream with the provided handler, or return 0 otherwise. Will return -1 is the file is not open.

9.3.12 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.

9.3.13 seek

Format:

integer= seek (integer_handler, integer_position);

Move the data pointer to the specified position (starting from 0) and return 0 if successful, or a negative value result, otherwise.

9.3.14 fsize

Format:

integer= fsize (integer_handler);

Return file size in bytes, if the provided file handler is open, or negative result value in case of an error.

9.3.15 write

Format:

integer= write (integer_handler, ref_var, num_bytes);

Write data to open file with provided handler. Return the number of actually written bytes, or a negative number in case of an error.

The data is provided by referencing a variable. It is very important for the developer to ensure that the data size given in the parameter “*num_bytes*” is not greater than the actual size of the referenced variable or buffer.

9.3.16 read

Format:

integer= read (integer_handler, ref_var, num_bytes);

Read data from open file with provided handler. Return the number of actually read bytes, or a negative number in case of an error.

The incoming data is stored in a buffer provided by referencing a variable. It is very important for the developer to ensure that the data size given in the parameter “*num_bytes*” is not greater than the actual size of the referenced variable or buffer.

9.3.17 ffirst

Format:

text= ffirst (text_pattern);

Return the name of the first file that is matching the specified pattern. If there are no matching files, a blank string is returned.

9.3.18 fnext

Format:

text= fnext;

Return the name of the next file that is matching the specified pattern given in the last executed ***ffirst()***. If there are no matching files, a blank string is returned.

The ***ffirst/fnext*** pair of functions are allocated and maintained individually for each parallel process in order to avoid cross-process confusions.
If *fnext()* is executed without a preceding *ffirst()*, the behaviour is undefined.

9.3.19 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.

9.3.20 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.

9.3.21 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.

9.4 Multitasking

9.4.1 pproc

Format:

pproc (label);

Start new parallel process.
For additional details, see paragraph "***Multitasking***".

9.4.2 pterm

Format:

pterm (label);

Terminate parallel process.
For additional details, see paragraph "***Multitasking***".

9.5 Others

9.5.1 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.

9.5.2 run

Format:

run (text_filename);

Run and external executable file. The file can be either a normal compiled .RXE file, or a system commands batch .SYS file. The current program is terminated and removed from the memory, and the new file is loaded and started instead.

9.5.3 platform

Format:

text, text= platform;

Return the platform identifier in the first receiver variable, and the software version in the second one.

9.5.4 freemem

Format:

integer= freemem;

Return the size in bytes of the data memory currently available. The amount of free memory shows the total amount of free memory blocks. These blocks may not be forming a contiguous area and in systems with high level of memory fragmentation the actually available free memory in the form of a single block may be smaller.

9.5.5 uptime

Format:

Integer= uptime;

Return the number of microseconds passed since the beginning of execution of the current code in RVM.

9.5.6 wait

Format:

wait (integer_usec);

Perform a delay for specified number of microseconds.
If the parameter is a positive number, the delay is non-blocking, i.e. only the current process is paused while all other processes continue their execution. If the parameter is a negative number, then the function executes a blocking delay and all parallel processes pause until the delay is complete.
The accuracy of *"wait()"* depends on a number of factors, but generally, the blocking delays are much more accurate.

9.5.7 tick

Format:

tick (integer_usec, @func_ref);

Periodically execute a function at specified interval in microseconds. The exact timing of the call is not guaranteed more than the fact the function will be executed after *at least* the specified number of microseconds have passed. Value 0 in the period will disable further calls to the function.

All parallel processes have independent tick counters, however, only one tick function is possible per process.

Example:

tick 1000000, @myfunc;

The function myfunc() will get executed every 1 second.

An important requirement for tick-handling functions are not to have any input or output parameters.

9.5.8 type

Format:

integer= type (any);

Return a number indicating the data type of the argument. This function is particularly useful in conjunction with variables of type "***any***" to determine the type of data stored in the variable.

The function can be used on everything in Rittle - constants, variables, data arrays.

Data type codes returned by the function are:

0	invalid
16	any
18	byte (sint8)
20	small (sint16)
22	int (sint32)
24	big (sint64)
28	real
30	text
31	func

9.5.9 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.
If the argument is reference to a variable, then the function will return the size of the variable in memory. In case the variable is text, its length will be returned, and if the variable is an array, the sum of all its elements will be returned.

9.5.10 clear

Format:

clear (@var);

Clear the referenced variable. If the variable is an array, the entire array will be cleared.

Examples:

clear @a;
clear @b[];

9.5.11 isword

Format:

integer= isword (text);

Will return 1 if the supplied text parameter is a valid Rittle word, otherwise will return 0. This function can be used to determine whether words supporting some specific functionality are present in a system.

9.5.12 count

Format:

integer= count (any);

Return the number of elements in an array. If the array is multidimensional, the function will return the actual number of elements in all dimensions combined. When used with single variables it will always return 1.
It is important to notice that since this function refers to a variable itself, but not to the value of that variable, the correct calling syntax is by using a reference:

Example:

var byte a [10:10];
print count(@a[]);

9.5.13 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.

9.5.14 bit

Format:

integer= bit (integer);

Will return an integer number equivalent to $2^{(\text{parameter})}$. The parameter itself is in integer number in the range from 0 to 63.

9.5.15 userbrk

Format:

userbrk (integer);

Enable (when the parameter is not 0) or disable (when the parameter is 0) the **Ctrl-C** break by the user during execution. By default the user break functionality is always enabled and can be disabled only for the currently executed program.

9.6 User Interface and Graphics

9.6.1 print

Format:

print (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.

9.6.2 CRLF

Format:

text= CRLF;

Return a two-character string composed of the ASCII codes CR (0x0d) and LF (0x0a). An equivalent to "***_r_n***".

9.6.3 conch

Format:

variable[, variable, variable, …]= conch;

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

The function does not wait for characters to arrive from the console.

9.6.4 conrd

Format:

variable[, variable, variable, …]= conrd;

One or more variables are read from the console. The natural output from ***conrd*** is text, if inputting numbers is required, then the ***val()*** function should be used additionally in the form *"**val(conrd)**"*.

The total length of the line read at once could be up to 255 characters long.

Reading is terminated by receiving an ASCII code 10 (LF). The terminating code is not passed on to the output.

Note that "*conrd()*" is a **blocking function**. This means that whenever it needs to be executed, ALL parallel processes stop until the function produces its result.

9.6.5 cls

Format:

cls;

Clear the attached screen device and fill the screen with the currently selected background colour.

The background colour is the one set by the gp*attr()* function. Its default value is 0 and in the vast majority of displays that corresponds to black colour.

If the current background colour is set as transparent, the function will use colour 0 for clearing the screen.

On the system console outputs 250 blank lines.

9.6.6 gpattr

Format:

gpattr (int_Fcol, int_Bcol, int_scale);

Specify text attributes for the current font for *gprint()*. The first parameter defines the main font colour, the second parameters specifies the background colour, and the third parameter is the font scale.

When any of the colours is defined as negative number, it becomes transparent.

The colours are provided in 24-bit values that correspond to RGB:888 format. In many displays however, the colour is actually an 18-bit RGB value in bit format ***6--:6--:6--***, where the ‘**–** ’ bits don’t matter and are typically set in 0. Therefore, in an example, colour 0xAD237A will be the same as colour 0xAC2078, because the RGB bitmask of the first is 10101101**:**00100011**:**01111010, and that converted to 18-bit format takes the form 10101100**:**00100000**:**01111000.

The function has no effect in the console or on text-only displays.

9.6.7 gprint

Format:

gprint (int_posX, int_posY, any [, any, any, …]);

Performs as the normal *print()* function, but on the attached graphics display device only. It takes the first two parameters as display coordinates from where the output will start.

Nothing is sent to the console.

The function also ignores the functionality of the special ASCII codes below code 32, and prints their defined graphical images instead. Therefore, the text constants such as "_r" or "_n", are drawn as characters.

gprint() uses the currently installed font with the currently set attributes for colour and scaling.

9.6.8 pixel

Format:

pixel (int_x, int_y, int_colour);

Draw single pixel with specified colour. If the coordinates are outside of the screen boundaries, nothing is drawn.

The function has no effect in the console or on text-only displays.

9.6.9 fill

Format:

fill (int_x, int_y, int_colour);

Implements "flood fill" with specified colour in an enclosed area. The parameters X and Y can be any point within the area, and the colour of the pixel at position (x,y) defines the background colour that will be replaced by the fill colour. The area is filled using the colour specified in the last parameter of the function.

NOTE:

The *fill()* function requires reading from the display memory. Some cheap displays on the market don't have the relevant data pins exposed and available for use. The function will be unable to operate normally with those displays.

9.6.10 line

Format:

line (int_x1, int_y1, int_x2, int_y2, int_colour);

Draw line from screen coordinates (x1,y1) to screen coordinates (x2,y2), with specified colour. If any part of the line is outside of the screen boundaries, only the visible remainder is drawn.

The function has no effect in the console or on text-only displays.

9.6.11 circle

Format:

circle (int_x, int_y, int_radius, int_colour);

Draw circle with centre at coordinates (x,y) and specified radius. If any part of the circle is outside of the screen boundaries, only the visible bit is drawn.

If the colour is specified as negative number, only the outline of the circle will be drawn, otherwise if the colour is a positive number, the circle will be solid.

The function has no effect in the console or on text-only displays.

9.6.12 ellipse

Format:

ellipse (int_x, int_y, int_xradius, int_yradius, real_tilt, int_colour);

Draw ellipse with centre at coordinates (x,y) and x-radius and y-radius. If any part of the ellipse is outside of the screen boundaries, only the visible bit is drawn.

In cases when the X-radius is equal to the Y-radius, the ellipse becomes a circle.

If the colour is specified as negative number, only the outline of the ellipse will be drawn, otherwise if the colour is a positive number, the ellipse will be solid.

The tilt parameter defines an angular tilt for the ellipse's coordinate system in relation to the native screen coordinates. The parameter is a real number given in radians. When this parameter is 0.0, i.e. there is no tilt, then the coordinate system of the ellipse match exactly the coordinate system of the screen.

Although not limited, the actually useful range for the tilt parameter is between -3.14 and +3.14 radians (negative Pi to Pi), since all values outside will replicate the same results.

The function has no effect in the console or on text-only displays.

9.6.13 triangle

Format:

triangle (int_x1, int_y1, int x2, int_y2, int_x3, int_y3, int_colour);

Draw triangle specified by the coordinates of its three defining points. If any part of the triangle falls outside of the screen boundaries, only the visible remainder is drawn.

If the colour is specified as negative number, only the outline of the triangle will be drawn, otherwise if the colour is a positive number, the triangle will be solid.

The function has no effect in the console or on text-only displays.

9.6.14 rect

Format:

rect (int_x1, int_y1, int_x2, int_y2, int_colour);

Draw rectangle whose top-left corner is specified with the coordinates (x1,y1) and bottom-right corner at coordinates (x2,y2). If any part of the rectangle falls outside of the screen boundaries, only the visible remainder is drawn.

If the colour is specified as negative number, only the outline of the rectangle will be drawn, otherwise if the colour is a positive number, the rectangle will be solid.

The function has no effect in the console or on text-only displays.

9.6.15 gput

Format:

gput (int_x1, int_y1, int_x2, int_y2, ref_sint32_var);

Move defined rectangular area from a referred variable on to the screen within specified coordinates range. The variable is must be a one-dimensional or multi-dimensional array with total number of elements equal or greater than the needed minimum which is calculated as:

*needed_bytes = (x2-x1+1) * (y2-y1+1) * 3*

The formula above assumes the following conditions are met: *(x2 ≥ x1), (y2 ≥ y1)*

The function automatically reorders internally the input parameter in order to meet the conditions above for proper calculation.

If the size of the referred variable does not have enough number of elements, the function will terminate its work once the entire array has been processed.

The last multiplication in the formula above is needed because the colour of every pixel on the screen is stored in three consecutive bytes for red, green, and blue, respectively. Since the "sint32" type (also defined as "*int*") is 32-bit, its size is enough for a single pixel 24-bit colour.

If the colour value for a pixel is negative number it is ignored and the colour of the relevant pixel on the screen remains unchanged.

Example:

```
gput (25, 15, 49, 34, @figure[] );        ' put 25x20 pixels bitmap figure[] at
                                          ' screen coordinates (25,15)
```

9.6.16 gget

Format:

gget (int_x1, int_y1, int_x2, int_y2, ref_sint32_var);

Get defined rectangular area from the screen into a referred variable. The variable is must be a one-dimensional or multi-dimensional array with total number of elements equal or greater than the needed minimum which is calculated as:

*needed_bytes = (x2-x1+1) * (y2-y1+1) * 3*

The formula above assumes the following conditions are met: *(x2 ≥ x1), (y2 ≥ y1)*

The function automatically reorders internally the input parameter in order to meet the conditions above for proper calculation.

If the size of the referred variable does not have enough number of elements, the function will terminate its work once the entire array has been processed.

The last multiplication in the formula above is needed because the colour of every pixel on the screen is stored in three consecutive bytes for red, green, and blue, respectively. Since the "sint32" type (also defined as "*int*") is 32-bit, its size is enough for a single pixel colour. Although it also creates a 25% redundancy in the occupied bytes in memory, using the sint32 type offers backward compatibility with the ***gput()*** function, and allows further manipulation of the pixel colours and conversion into transparent without need for additional buffers.

After execution, the referred variable contains a copy of the screen area with the colour of every pixel stored in a separate 32-bit array element.

NOTE:

The *gget()* function requires reading from the display memory. Some cheap displays on the market don't have the relevant data pins exposed and available for use. The function will be unable to operate normally with those displays.

9.6.17 font

Format:

font (ref_sint8_var);

Set a new font to be used by all text output functions on the externally attached display device.

The parameter points to a *sint8* ("*byte*") type array which contains the binary definitions for the font.

Every font in Rittle has the following standardised structure (all fields are byte):

startL, startH 16-bit number that specifies the character code of the first character in the font. It must be greater than 0 since code 0 is reserved for end of string indication and cannot be displayed.

In standard ASCII fonts, currently supported by Rittle, the character codes are always 8-bit, so the high byte in this field should be kept 0.

countL, countH	16-bit number that tells how many character definitions are included in this font. Rittle currently supports only 8-bit character codes, so although more characters can be defined in the font, the actually usable range is up to 255 characters and the high byte should be kept 0.
width	Pixel width of a single character. This parameter specifies the number of columns in the characters. In fonts where this field is 0, every character definition starts with an additional byte that defines how many columns are present in that character only. In fonts with fixed width where the "*width*" parameter is greater than 0, the leading width-specifying byte in every definition is missing since the width is already know for all characters.
height	Pixel height of a single character. Although they could be different in width, the height of all characters in the font is the same. The actual number of bytes for every character definition depend on the width and height of the font combined: for fonts with height 8 lines or less, every byte represents one column, for fonts with height 16 lines or less, every column takes two bytes, and so on. Then the width specifies how many columns are needed for the entire character. Bit counting starts from bit 0 which represents the top pixel, bit 1 is the one below, and so on. For fonts with more than 8 lines the counting continues in the same fashion on the following byte. If the number of lines in the font are not divisible by 8, the remaining bits in the last byte of every column remain unused, not displayed, and should be kept 0.

blankL	Number of blank columns to add on the left side of every character.
blankR	Number of blank columns to add on the right side of every character.
blankU	Number of blank rows to add on top of every character.
blankD	Number of blank rows to add under every character.
name ... 0x00	Text name of the font. The text must finish with a byte 0. If there is no name, then this field contains only a single byte 0.

The actual character definitions start after this font header.

Every character is defined as a bit mask within one or more bytes. The characters have consecutive codes, starting from the code specified in the "*start*" field in the header, and going on for the number of "*count*" characters. There are no separators between two neighbouring definitions. Rittle works out the numbers from the values supplied in the font header.

<u>Font definition examples:</u>

1. The definition of character 'A' in a font with fixed width 5 (the "width" field in the font header has value 5 and that defines that all characters in the font will be within 5 columns) and height 7:

   0x7C, 0x12, 0x11, 0x12, 0x7C

   This sequence of bytes looks like this:

	7c	12	11	12	7c
0			#		
1		#		#	
2	#				#
3	#				#
4	#	#	#	#	#
5	#				#
6	#				#
7	*not used*				

In addition to the definition, in the font header there are fixed definitions for certain number of blank pixels to be added to all sides of every character in the font.

2. Definition of character ‘W’ in a font with variable width (the “width” field in the font header has value 0) and height 14 pixels.

   Since the width is variable, every character in the font has its own width, and that is specified in one additional byte at the beginning of every character definition.
   The particular character in this example has 12 columns, and every column is described by two bytes. Since by definition the font has maximum height of 14 pixels, the last two bits in the second byte will remain unused and they are not displayed on the screen.

   The definition will look like this:

   12, 0x01,0x00, 0xFE,0x01, 0x00,0x07, 0x00,0x20, 0x20,0x18, 0xE0,0x07, 0x20,0x1e, 0x00,0x20, 0x00,0x18, 0xFE,0x07, 0x01,0x00

	01	fe	00	00	00	20	e0	20	00	00	fe	01
0	#											#
1		#									#	
2		#									#	
3		#									#	
4		#									#	
5		#				#	#	#			#	
6		#					#				#	
7		#					#				#	

	00	01	07	1e	20	18	07	1e	20	18	07	00
8		#	#				#				#	
9			#	#			#	#			#	
10			#	#			#	#			#	
11				#		#		#		#		
12				#		#		#		#		
13					#				#			
14	not used											
15												

All character definitions start immediately after the font header, and follow up in sequential order, so in the example above, after the character ‘W’ will follow character 'X', then character ‘Y’, and so on.

Example of the font() function use:

```
data byte myfont =
        32, 0,                  ‘ the font starts from code 32 (space character)
        1, 0,                   ‘ there is only one character in the font
        8,                      ‘ the width is fixed at 8 columns
        8,                      ‘ the height will be 8 lines
        0, 1, 0, 1,             ‘ blank pixels on left/right/up/down
        0,                      ‘ name of the font (no name)
        0, 0, 0, 0, 0, 0, 0, 0;‘ definition of the first character in the font

font ( @myfont[] );             ‘ now “myfont” becomes the active font
```

9.6.18 shape

Format:

shape (int_x, int_y, text_def);

Draw vectored shape described in the text parameter, starting from coordinates (x,y). This method is also known as “turtle graphics”. The text definition consists of a number of micro-commands telling “the turtle” in what direction to do or set the colour for drawing.

Spaces in the definition string are not needed and are ignored during translation, however numbers cannot contain spaces within.

“***C*** *colour*”	This command sets the drawing colour. The colour can be specified either as decimal or hexadecimal number (starting with an ‘**x**’ character). In case of hexadecimal, the leading ‘0’ character is only optional.
“***F***”	Flood fill an area starting from the pixel ate the current coordinates. The area is then filled with the last colour

	set by the ‘C’ command. The colour parameter is given following the same rules as for the ‘C’ command.
“***N***”	Set transparent drawing colour. This is used when the drawing pen needs to be lifted in order to go somewhere else without drawing on the way there.
“***M*** *[stepsX]* ***,*** *[stepsY]*”	Move in specified direction for number of horizontal and vertical steps. The steps are provided as a decimal number that can be positive or negative. There is two parameters to this micro-command, and they are separated by a comma. The first parameter defines horizontal displacement, and the second one defines vertical displacement. If any of the parameters is not needed, it can be omitted, but the presence of the comma is important to ensure the place of the parameter.

Example:

```
shape ( 60, 30,  "Cxff0000 M10,20 M20,10 M-20,10 M-10,20 Cx00ff00 M-10,-20
                          M-20,-10 M20,-10 M10,-20 NM,15 Cxff M,30 N M-15,-15
                          Cxff M30," );

                     ‘ This definition will draw a four-rayed star with green left
                     ‘ part and red right part, and a blue cross inside, starting from
                     ‘ the top point at coordinates (60,30)
```

10GLOSSARY OF THE CORE FUNCTIONS

This is the standard set of functions that form the core set of the Rittle language. They don’t depend on the hardware configuration of the system and work with the console only.

! ' '_ '! '!_ - -- $ * / \ \\ ^ ~ + ++
< << <= <> = == > >= >> abs acos and
any asin at atan big bit byte char chdir
clear close code conch conrd copy cos count
CRLF cut data deg delete E else end endfunc
endif endunit eof exitfunc exitloop exp ffirst
fnext format fpos freemem fsize func hsin
htan include if init input insert int ioerr
isopen isval isword ln log maxlen mkdir mount
not open or output PI platform pproc print
pterm rad random read real redim refer renvar
repeat rmdir run search seek sign sim sin
sint16 sint32 sint64 sint8 size small tan text
tick trim type unit until uptime userbrk val
var wait where while write xor

11 GLOSSARY OF THE GUI FUNCTIONS

These functions only work when a driver relevant to their functionality is installed. Otherwise, they are valid and part of the always available Rittle's hardware-independent core, but provide no output on the system console.

circle cls ellipse fill font gget gpattr gprint
gput line pixel rect shape triangle

12 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.

12.1 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*

12.2 Variables as Data Types

A very important feature in Rittle is that every variable immediately also becomes a data type. Thus, the developer could define own data types for a clearer source code.

Example:

var int size;
var size code_length;

It can be seen that the second variable "***code_length***" is of type "***size***" which is actually a previously defined variable.

12.3 Renaming and Reusing Variables

Rittle provides the option to the developer to rename already defined variables and use them again instead of defining new ones. This does not generate any additional code but instead the RSC only renames variables to for better source clarity. The word to do that is "***renvar***":

renvar oldname, newname;

Units can be renamed as well. Note that renaming does not require specifying [] for arrays.

12.4 Dynamic Arrays

Another useful feature of Rittle is the possibility to dynamically change the size of an array variable, or even convert a single variable into an array.

In order for that to be done, the variable needs to exist already (has been declared earlier with a "var" statement). Rittle provides a word "***redim***" which is used to change the dimensions of already existing variables. This feature significantly simplifies work with various lists and other structures with unknown length as the developer can expand or reduce the structure as per the needs on the go.

Example:

```
var int something[10];    ' the array is initially declared to have 10 elements
redim something[20];      ' the same array now has 20 elements
```

The word "***redim***" retains all existing data in the array, however if the new size is smaller than the old one, the excess elements at the end of the array will be lost. A special consideration must be given to re-dimensioning of multi-dimensional arrays. Although it is technically possible, the results might differ from the expectations, and the data elements might end up shifted away from their original indexes, so ideally re-dimensioning should be done on one-dimensional arrays only.

12.5 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".

12.6 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);
```

12.7 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 small form;          ‘ this value defines the behaviour

   if form==0;
      var input small 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 [10];          ‘ and refers to array
     ……………………………………………
     ……………………………………………

   endif;
endfunc;
```

12.8 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;
```

12.9 Function Variables

Function variables are special type variables that refer to actual functions. With function variables a function can be passed as an 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.

12.9.1 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.

12.9.2 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.

13 RIDE

The “Rittle Integrated Development Environment” (RIDE) is part of the Rittle system and has the task to help facilitate writing programs, perform compilation, generate executable files, as well as do basic debugging of the written code.

RIDE is activated by a console command “***ride***” with optionally a file name following the command.

Entering RIDE goes straight to its line text editor. The output looks like this:

```
IFS:/_ride
  1:
```

Now the text editor is positioned on line 1, and ready to accept text. Let’s start writing in Rittle. It is important to know that every line can be edited by using the left and right arrow keys, the keys <Del> and <BckSpc>. Text can be inserted or removed, all done until an <Enter> key is pressed.

Another very important detail is, if the first character in a line is a ‘**.**’ Then RIDE will interpret it is a command for the environment, and will try to execute the following characters as one or more RIDE commands.

RIDE’s text editor is not a typical editor but a line-based one. This means that what is shown on the screen is not necessarily how the text looks. Individual lines can be displayed, edited, and then other lines displayed below them in a non-sequential order. Using line editor brings several benefits. First, it does not depend on the hardware in any way. The line editor will look the same way on a small 2-line LCD as it will look on a large terminal screen. Display height and width don’t matter. Another benefit is, once mastered, using a line editor does actually make writing code quicker. For example, in a source of several hundred or several thousand lines of code, a developer might find themselves frequently scrolling up and down over large chunks of code in order to check and refer to different parts of code. In RIDE this is achieved by typing short one-character commands or sequences. In a short ‘dot-command’ line the developer could for example jump to a line and change something. Compile and execute, or trace code. RIDE also supports repeating the same command defined number of times. This is very useful when searching and replacing, as well as during debug.

Full help about the RIDE commands can be invoked by executing a command **.H**

```
IFS:/_ride
  1: .h
```

```
A single dot is enough for full command line with multiple commands
._   exit (text remains in the memory)
.Z   clear screen (250 new lines)                .H   this information
.U   undo the last edited line                   .?   other useful information
.L   [number of lines] [,] [starting from line number]  list (recent or from)
.D   <number of lines> [,] [starting from line number]  delete lines
.I   <number of lines> [,] [starting from line number]  insert lines
.C   <number of lines> [,] [starting from line number]  copy lines at current
.M   <number of lines> [,] [starting from line number]  move lines to current
.[J] [line] or <N> or <P>    jump to line or to next line or to previous line
.F   [text to EOL]      define 'Find' string to find or perform find function
.R   [text to EOL]      define 'Replace' string or perform find and replace
.O   <NEW> or <file.RIT> start new blank file or open a file with given name
.S   [file.RIT]  save file (optionally can save with a new file name)
.*   [number of times]   repeat the following command line number of times

Rittle Compilation and Debug:
..   first enter step mode then execute single instruction in current process
.=   first enter step mode then execute single instruction in all processes
.#   <file.RXE>  create executable file for RVM
.V   <variable id> [, <linear index>]   inspect variable
.\   inspect RVM stack for current process      .[   compile in mem [opts $%]
.B   place/remove breakpoint                    .>   run or continue after BP
  1:
```

After executing the help command, the editor returns back to the same line waiting for text.

13.1 Writing Programs in the Text Editor

Let's write a line:

```
  1: var byte a=0;
  2:
```

After writing the line and pressing the <Enter> key, the text editor has accepted the line, stored it in the memory, and ready for a new one. Let's continue further:

```
1: var byte a=0;
2: while a<10;
3:    print a++,CRLF;
4: until;
5:
```

At this stage we may decide to make a change somewhere in the source, for example to change the number in the 'while' condition. For that, first we need to go back to the line, but even before that we may want to see the program again:

```
5: .l

1: var byte a=0;
2: while a<10;
3:    print a++,CRLF;
4: until;
5:
```

Now we jump to the needed line:

```
5: .2

2: while a<10;
```

After jumping to line 2, the cursor positioned on its first character and now we can edit the line.

Note that while over a line with text, any RIDE dot-commands that start from the very beginning of the line, are executed without affecting the source code. Thus while on line 2, if we jump back to the bottom of the source code by command **.0**

```
2: .0while a<10;
```

After <Enter> the rest of the line gets erased from view for clarity.

```
  2: .0

  5:
```

13.2 Compiling and Executing Programs

We have now written a simple Rittle program to print the numbers from 0 to 9. The program can be compiled and executed, stored in a file, or stored as a binary executable file.

Compiling the program is done by dot-command **.[**

```
  1: var byte a=0;
  2: while a<10;
  3:    print a++,CRLF;
  4: until;
  5: .[
lines compiled: 4
code length: 48 bytes
  5:
```

Now the program ready for execution with a **.>** command. In fact the entire compilation and execution process could be expressed in a single line. We might also like to see the compiled binary RVM code by inserting '**%**' key before the '**[**' command (refer back to the help):

```
  5: .%[>
lines compiled: 4
code length: 48 bytes

000000   0f 4b b8 44              .reset     K╕D
000004   00                       .nop
000005   2a 12 01 00 00 00        .var       .sint8   V1
00000b   12 00                    .sint8     0
00000d   29 01 00                 .set       V1
000010   28 01 00                 .get       V1
000013   12 0a                    .sint8     10
000015   32                       <
```

```
000016   0a 2e 00 00 00          .ifnot    (addr 0x0000002e)
00001b   28 01 00                .get      V1
00001e   28 01 00                .get      V1
000021   49                      .++
000022   29 01 00                .set      V1
000025   38                      CRLF
000026   3f                      print
000027   12 00                   .sint8    0
000029   0a 10 00 00 00          .ifnot    (addr 0x00000010)
00002e   3f                      print
00002f   05                      .exit

0
1
2
3
4
5
6
7
8
9

>> ok


  5:
```

The program compiled, listed the binary RVM code, and then executed. After it finished, a message ‘**>> ok**’ was displayed and the editor is ready for more text or new commands.

13.3 Debugging Programs

RIDE offers basic debugging to trace how the executed code preforms. One or more RVM instructions from the current or all processes, can be executed in steps. Note that debugging is done on the compiled RVM binary code, not on the input Rittle source code.

The RVM code is sort of machine code that executes in the RVM. It is optimised to level that allows realisation of most of the RVM directly in hardware. The entire model is built around a stack machine, and all instructions take operands from the stack and return data back these.

In order to start debugging, a compilation is required in order to prepare the code and to reset the RVM for a new execution.

Using with the small program from above:

```
  1: var byte a=0;
  2: while a<10;
  3:    print a++,CRLF;
  4: until;
  5: .[
lines compiled: 4
code length: 48 bytes
  5: ..

000005   2a 12 01 00 00 00         .var       .sint8   V1

  5:
```

We have entered debugging and seeing the instruction due for execution.

Continuing further with the **..** command for step execution:

```
  5: ..

00000b   12 00                     .sint8     0

  5: ..

]  0: 0
```

After executing this instruction, we see a value has been added to the data stack. The data stack always puts a new value on top of the already existing ones, and the value that is taken is always the one that is on the top.

Step-debugging further:

```
00000d   29 01 00                  .set       V1

  5: ..

000010   29 01 00                  .get       V1

  5: ..

]  0: 0

000013   12 0a                     .sint8     10

  5: ..

]  0: 10
] -1: 0

000015   32                        <

  5: ..

]  0: 1

000016   0a 2e 00 00 00            .ifnot     (addr 0x0000002e)

  5:

00001b   28 01 00                  .get       V1

  5: ..

]  0: 0

00001e   28 01 00                  .get       V1

  5: ..

]  0: 0
] -1: 0

000021   49                        .++

  5: ..

]  0: 1
] -1: 0

000022   29 01 00                  .set       V1
```

```
  5: ..
]  0: 0
000025   38                       CRLF
  5: ..
]  0:

] -1: 0
000026   3f                       print
  5: ..
0

000027   12 00                    .sint8     0
  5: ..
]  0: 0
000029   0a 10 00 00 00           .ifnot     (addr 0x00000010)
  5: ..
000010   28 01 00                 .get       V1
  5:
```

We now see the first printed character in the marked line (it is not marked in the actual system output), and the loop has completed its first iteration returned back to the start for a next one. We can also see that there is a blank line in the stack output between elements 0 and 1. That blank line is actually a valid data element, generated by the CRLF instruction.

Adding more dots will add more instructions traced in a single step.

... will execute two instructions, **....** will execute three instructions, and so on.

Debugging can be done on executing batches by using the **.*** command to specify the number of instructions. For example:

.*50.

will execute 50 instructions before it stops for further interaction with the user.

13.4 Saving, Loading, and Generating Executable Files

Source files can be saved or opened by using the **.S** and **.O** commands, respectively. Let’s save the small test program:

```
  5: .s test1.rit
saving to test1.rit
>>> 50 bytes written

  5:
```

Starting a completely new source can be initiated by using the open command with parameter ‘**NEW**’:

```
  5: .o new
  1:
```

Now all the previous text in the editor is deleted and it has returned back to its initial state, ready for new text.

Let’s open the test program again:

```
  1: .o test1.rit
opening test1.rit
>>> 50 bytes read

5: .l

  1: var byte a=0;
  2: while a<10;
  3:    print a++,CRLF;
  4: until;
  5:
```

After a program has been written and the developer is happy with its performance, it can be compiled and stored as a binary executable for further distribution. Binary executables do not contain source code, but only the machine code as compiled for the RVM.

Generating a binary executable file is done by using the **.#** command. Before saving as binary executable, though, the program needs to be compiled, so combining the two commands in a single line will perform the needed operations at once:

```
  5: .[# test1.rxe
lines compiled: 4
code length: 48 bytes


saving to test1.rxe
>>> 48 bytes written

  5:
```

Now on the current file drive we should have saved both the source file **test1.rit** and the executable binary **test1.rxe**

Although not enforced by RIDE, in order to ensure further compatibility in future, it is recommended that Rittle source files are always stored with file extension **.RIT** and the executables are stored with file extension **.RXE**

Exiting RIDE is done by executing **._** command. The source file that is currently in the editor remains in memory, so re-entering RIDE later, will allow further work from that point.

14 OPERATING ENVIRONMENT

The operating environment in Rittle is where the system interacts with the user for work with files and storage devices. This is the place where the user finds themselves after initial system boot. The environment is a simplified version of a disk operating system. It allows executing Rittle commands in direct mode straight from the console, executing shell commands, entering RIDE, as well as working with file storage devices and running files from them.

On initial start a text like this will be seen in the console:

```
Rittle v:J5.1-alpha [PC],  (C) KnivD

IFS:/_
```

Rittle always boots up in the device called '**IFS**' (Internal File Storage). The presence of IFS is required for the normal operation of a Rittle system. Other file devices may also be available, depending on the particular system configuration.

Several commands, along with '**ride**', are available to the user at this point.

One important detail to note, is that the environment commands, wherever they accept parameters, the parameters should be enclosed in double quotes (**"**) just like normal Rittle text.

For convenience, however, the environment commands also allow execution with parameters that are not enclosed. Execution of the same commands from within a Rittle program, will require the parameters properly enclosed as text.

For example, the command **chdir** can be executed as **chdir path** or **chdir "path"** within the environment, but must be executed as **chdir "path"** in a Rittle program.

14.1 The DIR Command

This is probably the most user environment command. It lists the files and directories on the current or other specified device.

```
IFS:/_dir

directory IFS:/

 ----A 2019/04/24 18:38         15    config.sys
 ----A 2019/04/11 11:29        734    hello.rxe
 ----A 2019/05/05 09:28         50    test1.rit
 ----A 2019/05/05 09:37         48    test1.rxe

   4 file(s),          847 bytes total
   0 dir(s),       8330795 bytes free

IFS:/_
```

This is a typical output of a DIR command. It lists all files and directories along with their date, size, and attributes. It also shows the amount of free space remaining on the drive.

DIR can accept parameters to specify particular storage device, a specific path to directory or file, or a standard file mask by using the '*****' and '**?**' characters.

For example:

DIR *.rxe

will list the executable files in the current directory.

DIR ifs:/mydir1/prog*.rit

will list files that start with 'prog' and have file extension '.rit', from directory ifs:/mydir1.

14.2 The MOUNT and INIT Commands

In case there are more than one storage device in the system, changing between them is done by executing command 'mount'. For example, in some systems, a storage device located in the RAM might exist for easier exchange of operational data between various programs. Changing from the current drive to that one could be done as in the example below:

```
IFS:/_mount ram:
RAM:/_
```

Now the current drive has become RAM:

This, however, is only possible if the specified device is already initialised with a file system on it. In the particular example of a RAM drive, the information on it disappears every time when the system restarts, so it is likely that there won't be any file system at the time of first attempted access.

The system will generate an error when the user attempts to access a device that doesn't exist or has no file system.

The output in such case when the RAM drive is not initialised will actually be like this:

```
IFS:/_mount ram:
>>> drive error 13: no valid file system on the device

_
```

Note that after this error the system is '*nowhere*'. There is no active drive listed in the prompt. In order to gain access to files again, the user will need to mount some valid device, or initialise one. Rittle operates normally even being 'nowhere', hardware-independent commands can be executed from the console, RIDE is also accessible, but no file operations are possible.

We can return back to IFS like this:

```
_mount ifs:
IFS:/_
```

But in the example above we actually wanted to access the RAM drive, so instead of remounting IFS, we will use the INIT command to create a file system on the RAM drive and prepare it for normal work:

```
_init ram:
>>> initialised size 1028096 bytes

RAM:/_
```

The RAM drive is now initialised and successfully mounted. In case there is already a file system in a drive that the user is trying to initialise, the command will display a warning and will expect a manual confirmation before performing the operation.

Rittle currently supports the following storage devices:

ifs:	Internal File Storage, must be always present in every Rittle system
ram:	Optional data drive based in RAM
sd1:	Optional external SD card on ports 3 (CS#), 4 (SCLK), 5 (MISO), 6 (MOSI)
sd2:	Optional external SD card on ports 2 (CS#), 4 (SCLK), 5 (MISO), 6 (MOSI)

When initialising a new file system, Rittle uses FAT or FAT32, automatically selected depending on the size of the storage drive.

In the command line, the '***mount***' command can omitted. Thus, for example, typing just '***sd1:***' in the console will have the same effect as '***mount sd1:***'. This is valid for all file devices.

14.3 Running Executable Files

The command **RUN** is used to execute binary RXE files. For example:

run hello.rxe

will load and execute the file 'hello.rxe'.

The RUN command can be used also for execution of .SYS text files, similar to CONFIG.SYS. In such case every line of the text file is executed individually as if it was entered in the console.

14.4 Listing Text Files

The command **LIST** will dump the contents of a text file on to the console screen. For example:

list myprog.rit

will load and list the text file ‘myprog.rit’. No execution will be performed. Listing process can be terminated with the break key Ctrl-C.

14.5 Other commands for Work with File System

Several other commands are also available in the environment to work with the file system:

delete *filename*	- remove file
rename *filename***,** *newname*	- change the name of a file
copy *filename***,** *newpath*	- copy file from one drive/directory to another
mkdir *dirname*	- create a new directory
rmdir *dirname*	- remove a directory
chdir *dirname*	- make a directory current

14.6 System Configuration in CONFIG.SYS

On initial start, Rittle looks for a text file called ‘config.sys’, located in the root directory of the ‘ifs:’ drive. If file **ifs:\config.sys** exists, it is read out and interpreted as configuration commands for Rittle.

All environment commands (including ‘RUN’) as well as the full Rittle functionality are available for inclusion into the configuration file. It is in essence a script of individual Rittle lines that always get executed first on system start. Note that it is not a program since every line is compiled and executed independently, therefore loops and variables will only work within the same configuration line.

Executing the ‘config.sys’ file allows devices automatically mounted on initial start, files deleted or renamed, and importantly - one or more files can be made to execute automatically on system boot by using the ‘run’ command in the config.sys file. If there are more than one lines with ‘run’ command, they will be executed sequentially, i.e. the second program will start after the first one has finished, a third one will start after the second, and so on.

15 RITTLE FOR THE PIC32MZ MICROCONTROLLER

The PIC32MZ EF series are powerful microcontrollers with plenty of resources, and come in a convenient 64-pin package (other packages are also offered). It is very suitable for the realisation of a Rittle system.

Download the Current Version

Disclaimer: Rittle is in continuous development. There could be incomplete or missing functionality in the current version, or still unresolved bugs. The author will appreciate highly any feedback or suggestions for solving existing problems or adding more functionality.

15.1 Pinout

Rittle on the 64-pin PIC32MZ2048EFH064 (TQFP64 and QFN64) microcontroller uses the following pinout:

Pin	Service	Function	Alt Functions
1:		DO/DI/AI	
2:		DO/DI/AI	SD Card SEL2#
3:		DO/DI/AI	SD Card SEL1#
4:		DO/DI/AI	System SCLK
5:		DO/DI/AI	System MISO
6:		DO/DI/AI	System MOSI
7:	GND		
8:	V_{dd} +3.3V		
9:	[5V] MCLR#		
10:		DO/DI/AI	
11:		DO/DI/AI	COM4 Rx
12:		DO/DI/AI	
13:		DO/DI/AI	COM4 Tx
14:		DO/DI/AI	COM5 Rx / PWM (group 2)
15:		DO/DI/AI	PWM (group 1) / Vref-
16:		DO/DI/AI	COM5 Tx / Vref+
17:		DO/DI/AI	Console Rx
18:		DO/DI/AI	Console Tx
19:	AV_{dd} (filtered +3.3V)		
20:	AGND (filtered GND)		

21:		DO/DI/AI	PWM (group 1)		DB0
22:		DO/DI/AI	SPI2 MISO	Display MISO	DB1
23:		DO/DI/AI	SPI2 MOSI	Display MOSI	DB2
24:		DO/DI/AI		Display DC	DB3
25:	GND				
26:	V_{dd} +3.3V				
27:		DO/DI/AI		Display RST#	DB4
28:		DO/DI/AI	SPI2 Slave CS#	Display CS#	DB5
29:		DO/DI/AI	SPI2 SCLK	Display SCLK	DB6
30:		DO/DI/AI	PWM (group 3)		DB7
31:	24MHz Input Clock				
32:	Osc Control SLEEP				
33:	[5V] USB VBUS Input				
34:	V_{usb} +3.3V				
35:	GND				
36:	USB Console D-				
37:	USB Console D+				
38:		DO/DI	PWM (group 2)		
39:	V_{dd} +3.3V				
40:	GND				
41:		**[5V]** DO/DI	COM3 Rx / IIC2 SDA	TSDA	
42:		**[5V]** DO/DI	COM3 Tx / IIC2 SCL	TSCL	
43:		**[5V]** DO/DI	IIC1 SDA		
44:		**[5V]** DO/DI	IIC1 SCL		
45:		**[5V]** DO/DI	PWM (group 3)		
46:		**[5V]** DO/DI	PWM (group 3) / Wake#	TIRQ#	
47:		**[5V]** DO/DI		TCS#	
48:		**[5V]** DO/DI	32.768kHz Input Clock		
49:		**[5V]** DO/DI	SPI1 SCLK		
50:		**[5V]** DO/DI	SPI1 MISO		
51:		**[5V]** DO/DI	SPI1 MOSI		
52:		**[5V]** DO/DI	COM1 Rx		
53:		**[5V]** DO/DI	COM1 Tx		
54:	V_{dd} +3.3V				
55:	GND				
56:		**[5V]** DO/DI	COM2 Rx		
57:		**[5V]** DO/DI	COM2 Tx		
58:		**[5V]** DO/DI	SPI1 Slave CS#	DB8 RESET#	
59:	GND				
60:	V_{dd} +3.3V				
61:		**[5V]** DO/DI		DB8 DC (Display)	
62:		**[5V]** DO/DI		DB8 RD#	
63:		**[5V]** DO/DI		DB8 WR#	
64:		DO/DI/AI		DB8 CS#	

Externally generated 24 MHz clock is required on pin 31 for normal work of the system. Pin 32 is an output reserved for oscillator sleep control. When Rittle enters sleep mode, the output becomes in active high state which can be used to disable the external oscillator.

On initial start, Rittle initialises the USB console and its serial duplicate on pins 17 and 18. The serial console pins, however, can be recycled and used for other purposes, if initialised in the user software. Rittle will not attempt to re-initialise them again until the next boot.

The same pins – 17 and 18, along with the MCLR#, can be used to upload firmware into the PIC32 microcontroller by using the standard Microchip programmer tools such ICD3 or PickIt3. In such case pin 17 serves as clock (PGEC), and pin 18 serves as data (PGED).

Rittle allocates three pins for System SPI bus on pins 4, 5, and 6, needed to communicate with standard supported external hardware such as SD cards, serial displays, etc. In addition to these three pins, others are used for the relevant CS# line on the external device. Within the same group, Rittle allocates pins 5 and 6 for System I^2C, if needed.

The System SPI pins are not initialised during the boot procedure, and can be used by software for other purposes. Once there is a need for System SPI, however, Rittle will take over and initialise these pins, so they should be used with caution, and only if the software does not call functions that require System SPI.

The same is valid for the SD card select pins – they are not initialised during initial boot, and are free for use for other needs, however, if the system attempts to contact file device SD1: or SD2:, the relevant select pin will be automatically assigned and used as SD card select line.

On initial boot, Rittle checks whether there is an external feed of 32.768 kHz clock on pin 48, and if so, assigns the built-in real-time clock to use that clock. If there is no external clock on the pin, an internal (lower accuracy) LPRC oscillator will be feeding clock into the real-time clock, and pin 48 is then available for other use.

All other pins, not mentioned above, remain uninitialised and are available to the user.

NOTE: Rittle for PIC32MZ enables internal pull-up resistors on the following pins by default:

1, **2**, **3**, **28**, **41**, **42**, **43**, **44**, **46**, **47**, **62**, **63**, and **64**.

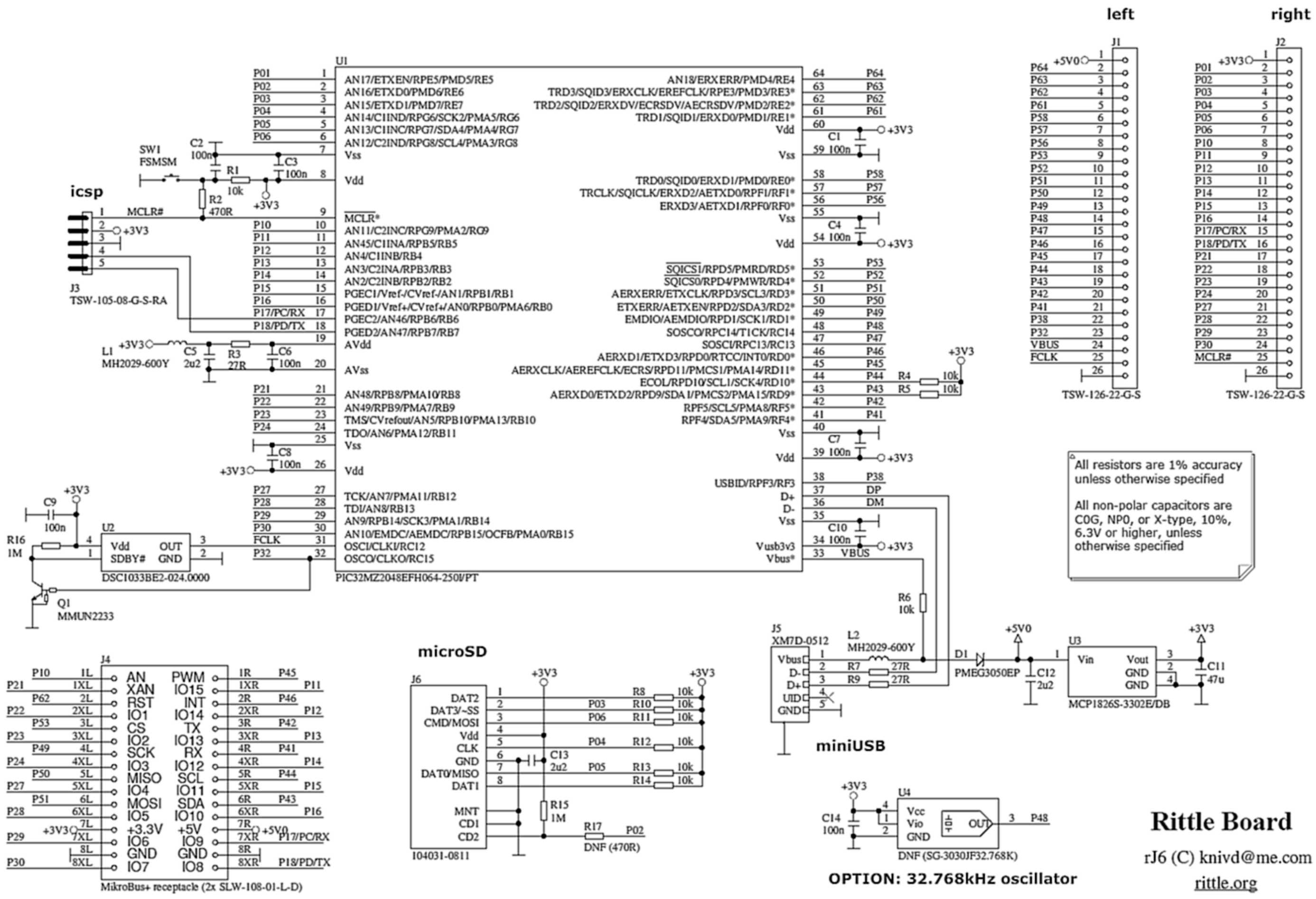
If any of these pins are required for functions where the internal pull-up will be undesirable, the relevant port should be re-initialised by the user program.

15.2 The RITTLE Board

An open source development board around the PIC32MZ2048EFH064-250I/PT microcontroller. All of the PIC32's I/O ports are exposed to the outside world, with only one permanently reserved for input clock, and one more for an optional secondary input clock for the RTCC. The schematics and all other files are available online.

The serial console is always available as Virtual COM on the USB port.

It is also doubled on the PIC32's pins 17 (Rx input) and 18 (Tx output). Default protocol is 115200, 8N1.



left
right
icsp
microSD
miniUSB
OPTION: 32.768kHz oscillator
PIC32MZ2048EFH064-250I/PT
All resistors are 1% accuracy unless otherwise specified
All non-polar capacitors are C0G, NP0, or X-type, 10%, 6.3V or higher, unless otherwise specified
Rittle Board
rJ6 (C) knivd@me.com
rittle.org

Download the Schematic

Download the Full Pack

Order RITTLE Board PCB for DIY Assembly

15.3 Work with I/O ports

The functions for control of the I/O ports include setting up port function and data direction.

15.3.1 port

Format:

port (text_function, integer_portN [, integer_portN ...]);

The 'port' function is used to initialise an I/O port for specific functionality.

The function parameter is provided as text, and the following one or more parameters are port numbers that need to be initialised all with the same function.

The port numbers are specified as per the numbering system in the RITTLE board – i.e. equal pin numbers. Therefore, PIC32's pin 5 corresponds to Rittle port 5, PIC32's pin 48 is Rittle port 48, and so on. Note that some port numbers are not available as they are being used for system purposes – power, etc.

Furthermore, depending on the system configuration, Rittle reserves some of these ports for other required functionality. Ports **17** and **18** are used by the serial system console, ports **3, 4, 5,** and **6**, are shared with the external SD card, and other ports may be shared with other functions.

The function is built from two sub-parts – switch and function. The switch is optional precedes the function.

Valid switches are:

"**+**"	Enable the built-in pull-up resistor on the port
"**-**"	Enable the built-in pull-down resistor on the port
"**#**"	The port is initialised as Open-Drain

Supported functions are:

"**DIN**"	The port is initialised as digital input. Reading from the port will return a value 0 or 1 only, depending on the logic level applied on the port.

”**DOUT**”	The port is initialised as digital output. Writing 0 to it will set the output low, otherwise writing a non-0 value will set the port high.
”**AIN**”	The port is initialised as analogue input. Reading from the port will return a real number value between 0 and 1, proportional to the voltage applied on the port in respect to the configured negative and positive reference for the ADC. See command ‘**Vref**’ for further details about ADC referencing. Not all ports can be initialised as AIN. Valid AIN port numbers (before applying the already reserved functionality for other needs) are: **1, 2, 3, 4, 5, 6, 10, 11, 12, 13, 14, 15, 16, 17, 18, 21, 22, 23, 24, 27, 28, 29, 30,** and **64**.
”**PWM:*freq***”	The port is initialised as digital output that generates pulse-modulated signal. This function requires its own parameter ‘frequency’ (given in Hertz) to specify the carrying frequency for the pulse width modulation. Carrier frequencies may vary from 1Hz up to 64MHz, however the practically usable values are between 10Hz and 10MHz since with increasing the frequency, the accuracy of the duty reduces. Not all ports can be initialised as PWM. Valid PWM port numbers (before applying the already reserved functionality for other needs) are: **14, 15, 21, 30, 38, 45, and 46** In addition to that, PWM ports share the carrier frequency in groups. Initialising carrier frequency for one port will apply the same frequency for the other ports in the same group, regardless of whether they have been initialised with a different value prior to that. There are three groups of ports, therefore at any moment of time up to three separate PWM carrier frequencies are possible: Group 1 – ports 15 and 21 Group 2 – ports 14 and 38 Group 3 – ports 30, 45, and 46

Examples:

port "-DIN", 12, 38, 50, 57;

port "+#DOUT", 64;

port "AIN", 21, 28, 29;

port "-PWM:50000", 45, 46;

15.3.2 Vref

Format:

Vref (text_function);

Specifies referencing model for the ADC. The parameter is given in text format. Valid options are:

"**Vdd/Vss**"	Reference taken from the positive and negative rails of the power supply, where Vref+ is 3.3V, and Vref- is 0V (GND). The accuracy of the ADC in this case depends on the accuracy of the power supply.
"**Vref+/Vss**"	Vref- is 0V (GND), and Vref+ is taken from the dedicated pin on PIC32 – port 16.
"**Vdd/Vref-**"	Vref+ is taken from the positive rail of the power supply, and Vref- is taken from the dedicated pin on PIC32 – port 15.
"**Vref+/Vref-**"	Both Vref- and Vref+ are taken from the dedicated pins on PIC32 – ports 15 and 16, respectively.

By default, Rittle uses ADC referencing to the power rails.

15.3.3 Din

Format:

integer= Din (integer_portN);

Read digital input and returns 0 or 1.

15.3.4 Dout

Format:

Dout (integer_bitVal, integer_portN [,integer_portN, …]);

Output of digital value 0 or 1 on specified one or more ports.

15.3.5 Dtog

Format:

Dtog (integer_portN [,integer_portN, …]);

Toggle the digital value of specified one or more ports. Toggling changes the output of the port to its negating state – if the port is currently high, it will become low, and vice versa.

15.3.6 Ain

Format:

real= Ain (integer_portN);

Read analogue input port and return a real number value between 0 and 1, proportional to the voltage on the port in respect to the set Vref+/Vref- values.

Example:

Let's assume Rittle uses the default ADC referencing model, where Vref+ is taken from the positive power supply 3.3V, and Vref- is taken from the ground power rail 0V, and there is 1.88V (in respect to GND) applied on the AIN port 22.

Executing *Ain(22)* will return value *0.569697*, which is proportional to the voltage over the entire range: 1.88/3.3 = 0.569697.

15.3.7 setPWM

Format:

setPWM (real_duty, integer_portN);

Set PWM duty cycle between 0 (0%) and 1 (100%) for the specified port number.

15.4 Work with Communication Interfaces

Rittle supports several serial interfaces, controlled by a small number of functions.

15.4.1 enable

Format:

enable (text_interface, text_parameters** [, **@func_callback()**] **);

Enable specified communication interface. Enabling the interface automatically configures the needed I/O ports.

There are several interfaces currently supported in Rittle:

"**SPI1**"	- SPI interface on ports 49 (SCLK), 50 (MISO), 51 (MOSI).
	Additionally, when initialised in slave mode, port 58 is automatically assigned as digital input, and monitored during execution. The call back function gets executed when a low level on the port has been detected. The user's function must monitor itself for the port going back to high, before exit.
"**SPI2**"	- SPI interface on ports 29 (SCLK), 22 (MISO), 23 (MOSI).
	Additionally, when initialised in slave mode, port 28 is automatically assigned as digital input, and monitored during execution. The call back function gets executed when a low level on the port has been detected. The user's function must monitor itself for the port going back to high, before exit.
"**IIC1**"	- I^2C interface on ports 43 (SDA) and 44 (SCL).
"**IIC2**"	- I^2C interface on ports 41 (SDA) and 42 (SCL).
	The same port as COM3, so only one of them can be used at a time.
"**COM0**"	- System console. Serial UART interface on pins 17 (Rx) and 18 (Tx)

"**COM1**"	- Serial UART interface on pins 52 (Rx) and 53 (Tx)
"**COM2**"	- Serial UART interface on pins 56 (Rx) and 57 (Tx)
"**COM3**"	- Serial UART interface on pins 41 (Rx) and 42 (Tx) The same port as IIC2, so only one of them can be used at a time.
"**COM4**"	- Serial UART interface on pins 11 (Rx) and 13 (Tx)
"**COM5**"	- Serial UART interface on pins 14 (Rx) and 16 (Tx)

The second parameter in the 'enable' function defines specific options for the selected interface. It is also given in text form like the first one with interface name.

An optional third parameter specifies a call back function that is executed when a data word is received. The requirements toward the call back function are very strict – it must have no input or output parameters.

Call back functions are always allowed for UARTs but only allowed in slave roles for the other interfaces.

For SPI interfaces the format is:

"role (**M** or **S**) [**, baudrate**] [**,** SPI mode (**0** or **1** or **2** or **3**), data word length (**8** or **16** or **32**)]]" [**, @rx_callback()**]

The role could be either **M**(aster) or **S**(lave).

Baudrate values are only valid in master mode and can be between 1 and 100,000,000 bits per second.

If not specified, the default SPI mode is 0, and the default data word length is 8 bits. New data word length can be defined only if SPI mode is specified too.

Examples for SPI interface:

enable "SPI1", "M, 12000000";	*' SPI master working at 12MHz in mode 0*
enable "SPI2", "S, 3, 16", @rx_spi();	*' SPI master in mode 3 with 16-bit data ' word and a call back function executed ' on every received data word*

For I^{2}C interfaces the parameters have the following format:

"role (**M** or **S**) [**, baudrate**]" [**, @rx_callback()**]

The role could be either **M**(aster) or **S**(lave), followed by baudrate (only valid in Master only) value between 1 and 5,000,000 bits per second.

Example for enabling the I^{2}C interface:

enable "IIC1", "M, 400000";

The UART serial interfaces implement the RS232 protocol and have this format:

"**baudrate** [**,** data bits (**8** or **9**) parity (**N** or **E** or **O**) stop bits (**1** or **2**)]" [**, @rx_callback()**]

Baudrate value can be between 1 and 25,000,000 bits per second.

If not specified, the default protocol parameters are 8 data bits, no parity, 1 stop bit. Other options for parity include **E**(ven) or **O**(dd).

Examples for enabling and configuration of UART interface:

enable "UART1", "115200, 8N1", @rx1; *' will execute the callback function*
' rx1() on every received data byte

enable "UART4", "9600"; ' default protocol 8N1 and without call back function

15.4.2 disable

Format:

enable (text_interface);

Disable specified communication interface. The I/O pins used for the interface are reset back to their default state.

15.4.3 trmt

Format:

trmt (text_interface, any_data [, any_data, …]);

Transmit data to the specified interface.

The interfaces is specified in text form – “UART1”, “SPI2”, etc.

Specifically for the I^{2}C case, the interface also includes an optional action, immediately following the interface identifier.

Actions can be

“**:S**”	- The transmission is preceded by an I^{2}C start condition.
“**:P**”	- The transmission is followed by an I^{2}C stop condition.
“**:R**”	- The transmission is preceded by an I^{2}C repeated start condition.

Data elements can be one or more of integer, real, or text type. Integer information sends a single data word, real is transmitted in its native form (8 bytes), and text is transmitted as a zero-terminated string of bytes.

Examples:

trmt “UART2”, 10, 20, 30, 40;	*‘ send bytes 10, 20, 30, 40, to UART2*
trmt “SPI1”, str1;	*‘ str1 is a text variable transmitted over SPI1*
trmt “IIC1:S”, 0x44;	*‘ a start condition followed by byte 0x44 is ‘ transmitted over I^{2}C. The bus remains open for ‘ following bytes*
trmt “IIC1:P”, 0x80, 0xfe;	*‘ a stop condition is generated after sending the ‘ supplied sequence of bytes*

15.4.4 recv

Format:

integer= recv (text_interface);

Receive a single data word from the specified interface.

The interfaces is specified in text form – “UART1”, “SPI2”, etc.

Specifically for the I^{2}C case, the interface also includes an optional action, immediately following the interface identifier.

Actions can be

“**:S**”   - The transmission is preceded by an I^{2}C start condition.

“**:P**”   - The transmission is followed by an I^{2}C stop condition.

“**:R**”   - The transmission is preceded by an I^{2}C repeated start condition.

Example:

var byte key = recv(“UART1”);

15.5 Real-Time Clock and Calendar

The Real-Time Clock and Calendar (RTCC) is a built-in hardware block in the PIC32MZ microcontroller. It has the function to keep track of the actual time and date, provided an external power supply is always supplying the microcontroller with power.

15.5.1 srctime

Format:

srctime (text_src, int_cal);

Enable the RTCC’s clocking source and specifies calibration value for the crystal drift. Two choices for the first parameter are possible:

“**LPRC**” selects the PIC32’s internal low power clock.

“**SOSC**” selects an external secondary 32.768 kHz clock connected to port 48.

The calibration value is typically 0, but can be changed in the range -512 … +511 adjusted clocks per minute.

On very first initialisation, Rittle tries to enable the external oscillator first, and falls back to the internal LPRC in case there is no clock coming on port 48.

15.5.2 settime

Format:

settime (text_dt);

Set time and date for the RTCC. The parameter is supplied as fixed length 16-character text in the format "***yymmddwwhhmmss00***".

The input text is composed by eight pairs of characters, each expressing a decimal value. If the value is smaller than 10, a leading '0' must be added to it.

The pairs are as follow:

year (00-99), ***month*** (01-12), ***day*** (01-31), ***weekday*** (Sunday=00 - Saturday=06), ***hour*** (00-23), ***minute*** (00-59), ***second*** (00-59), ***00***

Note that the weekday must be always supplied, as well as the trailing pair '***00***' which is reserved for future use but must be included in the parameter.

15.5.3 gettime

Format:

text= gettime;

Get time and date from the RTCC and return a fixed length 16-character text in the same format as in the ***settime()*** function. In case there are invalid characters in the text (such as in case the RTCC has not been initialised with a proper time, or is not functioning normally), the invalid characters are marked as '***#***' within the returned text.

15.6 System Control

Functions that control the system performance and power consumption.

15.6.1 clock

Format:

clock (int_freq);

Set CPU frequency clock or put the system into sleep mode. The frequency parameter is given in Megahertz.

Valid options are ***4***, ***16***, ***64***, ***128***, ***192***, and ***256*** MHz.

On initial boot, Rittle starts at its default clock frequency of 192 MHz.

NOTES:

1. The USB console will not work at 4 MHz clock. In such case only the serial console available on pins 17 and 18, is available.
2. The "Turbo" mode 256 MHz may not work on all PIC32MZ chips, or may render the system unstable. Additional measures to supply the processor with higher current or heat dissipation may have to be taken as well.

Current consumption with different clocks as measured in the RITTLE Board:

4 MHz	16 MHz	64 MHz	128 MHz	192 MHz	256 MHz
45.6 mA	50.8 mA	70.6 mA	104.4 mA	139.1 mA	171.5 mA

15.6.2 sleep

Format:

sleep (int_ms);

Put the system into sleep mode with minimum power consumption for a specified period of time, given in milliseconds. The typical accuracy of the sleep period is in around 32ms periods, so any values smaller than the granularity period, will cause an immediate wake up. Execution of all parallel processes is suspended.

NOTE: The current revision "B2" of the PIC32MZ EF microcontroller has an internal hardware problem with the sleep mode. That prevents it from achieving optimum sleep current, but instead, the minimum current remains as high as 9mA. Due to this and until Microchip manage to rectify the problem, the sleep() function will not be able to produce its desired result and it is recommended not to be used.

Value 0 for the parameter, will force the system into indefinite sleep with only remaining option to wake up from an external source.

Upon entering sleep mode, Rittle automatically configures pin 46 as input and enables its internal pull-up resistor. A falling edge on pin 46 will wake up the system and restore the last active frequency of the CPU clock, even if the specified time for sleep has not expired. Execution will continue with the instruction immediately following clock(). Pin 46 will remain configured as input but the pull-up resistor will be restored back to its initial disabled state. The user program will need to re-configure the port again if it is used for other purposes.

15.7 Specific Peripheral Modules

Rittle includes functions to support a selected range of popular peripherals.

15.7.1 LCD4

Format:

LCD4 (text_cmd [, any_param, …]);

Enable support of popular LCD modules with 4-bit HD44780-compatible interface. Since this interface is a de-facto standard among this type of display, the function should be able to cover the vast majority of such modules, currently on the market.

This type of displays are text-only and don't support colours. They also have fixed character table in ROM and no possibility for different screen orientations. However, their simple interface, good readability, and durable design make them a very popular choice in industrial applications where only limited user interface is required.

```
--------------\                /----------------------\
          P56 |--- RS ---------| RS (D/C in some)     |
          P57 |--- EN ---------| E                    |
PIC32MZ   P58 |--- D4 ---------| DB4                  |
          P61 |--- D5 ---------| DB5                  |
          P62 |--- D6 ---------| DB6                  |
          P63 |--- D7 ---------| DB7                  |
--------------/                |          LCD Module  |
                         GND|--| DB0                  |
                         GND|--| DB1                  |
                         GND|--| DB2                  |
                         GND|--| DB3                  |
                         GND|--| R/W                  |
                               \----------------------/
```

All display sizes from ***8x1*** (8 characters per line, 1 line) to ***40x4*** (40 characters per line, 4 lines) are supported.

Rittle uses six data lines for the interface as shown in the connection diagram. LCD's data lines DB0-DB3 and signal R/W are not used and should be permanently connected to ground.

COMMANDS:

1. ***Initialisation***

   Before any work with the LCD4 module, the interface needs to be initialised.
   LCD4 "initL", int_columns, int_rows
   Initialise the LCD4 interface with specified number of columns (8 – 40) and rows (1 – 4) for the display.
   The "***initL***" command initialises the display as "locked" – all characters printed at cursor position outside of the screen boundaries, will be ignored.

   LCD4 "initS", int_columns, int_rows
   Initialise the LCD4 interface with specified number of columns (8 – 40) and rows (1 – 4) for the display.
   The "***initS***" command initialises the display as "scrolling" – upon reaching the bottom of the screen, any character printed beyond the screen boundaries will cause a "scroll up" for all information on the screen.

   Screen initialisation can be performed as many times as needed, and if necessary the screen can be dynamically re-initialised from "locked" to "scrolling" and vice versa.

2. ***LCD4 "clear"***

   Clear the screen and set cursor position at the top left corner at coordinates 0, 0;

3. ***LCD4 "scroll:U"***
   LCD4 "scroll:D"
   LCD4 "scroll:L"
   LCD4 "scroll:R"

Perform screen scroll – up, down, left, or right, respectively.

4. ***LCD4 “goto”, int_x, int_y***

   Set screen coordinate (x, y) as the next location for output. All screen coordinates start from 0.

5. ***LCD4 “print”, any_data, …***

   Print information on the screen. One or more parameters of all ordinal types, can follow.

   Depending on the type of initialisation, printing will perform scroll up of the display, or ignore characters printed outside of the screen boundaries.

6. ***LCD4 “char”, int_code, big_bitmask***

   Define specific character. HD44780-compatible controllers allow up to eight custom-defined characters with codes from 0 through 7.

   The bitmask parameter consists of five bytes (all characters occupy 5x8 matrix) that define the character’s five columns from left to right.

15.7.2 display

Format:

display (text_type, int_resH, int_resV);

Enable support, initialise, and attach specified display driver. If a SPI display module is used, it occupies the same pins as SPI2 as shown in the pinout, plus two additional pins for display RESET and D/C signals.

Connections for SPI display

```
                                             SPI LCD Module
--------------\                 /------------------------\
          P22 |--- MISO -------| MISO (SDO in some)      |
          P23 |--- MOSI -------| MOSI (SDA in some)      |
PIC32MZ   P24 |--- D/C# -------| DC   (A0 in some)       |
          P27 |--- RESET# -----| RES                     |
          P28 |--- CS# --------| CS                      |
          P29 |--- SCLK -------| SCLK (SCL in some)     |
--------------/                \------------------------/
```

Connections for 8-bit display

```
                                  8-bit bus LCD Module
--------------\                /------------------------\
          P21 |--- DB0 --------| D0                     |
          P22 |--- DB1 --------| D1                     |
PIC32MZ   P23 |--- DB2 --------| D2                     |
          P24 |--- DB3 --------| D3                     |
          P27 |--- DB4 --------| D4                     |
          P28 |--- DB5 --------| D5                     |
          P29 |--- DB6 --------| D6                     |
          P30 |--- DB7 --------| D7                     |
              |                |                        |
          P58 |--- RESET# -----| RESET (optional)       |
          P61 |--- D/C# -------| D/C                    |
          P62 |--- RD# --------| RD                     |
          P63 |--- WR# --------| WR                     |
          P64 |--- CS# --------| CS                     |
--------------/                \------------------------/
```

Some displays may not require one of more of the control signals. In such case those signals can be left floating, however Rittle will still be driving them in a way as if they are being used by the display.

An exception to this are the HD44780-based text display modules as they use a different type of interface. Connections to those displays are described in the paragraph for LCD4.

The first text parameter for the function specifies the display controller model.

Currently supported displays are:

"NULL"	Null device. All currently attached display driver functions are detached and released with console remaining the single output destination. The parameters for display resolution have no meaning, and orientation is always the native for the console.

*“**HD44780**”*	NOTE: The parameter “HD44780” can be aliased as “LCD4”. Alphanumeric LCD with HD44780-compatible controller on 4-bit bus as described for the LCD4() function. These displays support only their native orientation, and their minimal addressable component is character, not pixel. Supported resolutions (in characters) are from 8 to 40 for resH, and from 1 to 8 for resV.
*“**LRSPI***”*	Covers the majority of popular display modules using controllers such as ST7735, ST7789, ILI9163, ILI9341, ILI9481, ILI9488, and many others from other manufacturers. These displays have resolutions up to 320x480 pixels, or smaller. They only support hardware scrolling in portrait mode, and for full support, in landscape mode the image scroll is handled in software, so presence of the MISO (or SDO) pin is required, otherwise all graphics will work, but the system console will not be properly supported.
*“**MLRSPI***”*	This driver is the same as *“**LRSPI***”* but supports displays with internally mirrored graphic matrix.
*“**SSD196***”*	Support for displays with SSD1961 / SSD1962 / SSD1963 controller and connected on the 8-bit bus. These are usually built around SSD1963 with lot of internal graphic memory. They are also able to perform fast scroll in all directions, so console performance in landscape mode will not suffer as in the case with the SPI displays.
*“**MSSD196***”*	Same as the “**SSD196***” driver but with mirrored graphic matrix.

The following two parameters are numeric and define respectively the **horizontal** (resH) and **vertical** (resV) resolution of the display in number of the smallest addressable component. Normally that is “pixel”, but for text-only displays, it is “character”.

Horizontal and vertical resolution parameters must match the specifications of the particular display panel, otherwise it won’t work properly.

Orientation of the panel is calculated automatically on the basis of the given parameters for display resolution.

When the horizontal resolution is greater than the vertical resolution, the display is in LANDSCAPE mode, otherwise, if the vertical resolution is greater than the horizontal, the display is in PORTRAIT mode.

If any of the two following parameters for resolution is given as a negative number, then the image of the display is mirrored, reverse landscape or reverse portrait.

In an isolated case when the horizontal resolution is exactly the same as the vertical resolution, the orientation is determined by the sign of both parameters as per the following table:

*Positive **resH**, Positive **resV***	LANDSCAPE mode
*Positive **resH**, Negative **resV***	Reversed LANDSCAPE mode
*Negative **resH**, Positive **resV***	PORTRAIT mode
*Negative **resH**, Negative **resV***	Reversed PORTRAIT mode

Text-only and a small number of graphic displays do not support different orientation modes. Those displays will work only in their native mode of orientation regardless the sign of the “*resH*” and “*resV*” parameters.

There is another accepted form for the display() function.

display (“CONSOLE”, int_fontFcol, int_fontScale);

This one is not to install a display driver, but instead, delivers instructions to the currently installed driver. The first parameter specifies 24-bit colour for the text font in the system console (the background is automatically initialised as black colour), and the second parameter specifies the scale for the system font.

If both parameters are greater than zero, system console output is enabled for the currently attached display device.

If any of the parameters is zero or negative number, the system console output for the attached display will be disabled.

This has no effect on the system console attached to the USB or serial port.

15.7.3 touch

Format:

touch (text_type/cmd [, …]);

Enable support, initialise, and attach specified touch screen driver. If one with SPI is used, it occupies the same pins as the system SPI as shown in the pinout, plus two additional pins for TCS# and TIRQ# signals.

```
                                               SPI touch panel
--------------\                 /------------------------\
          P4 |--- SYS_SCLK ---| SCLK (T_CLK in some)    |
          P5 |--- SYS_MISO ---| MISO (T_DO in some)     |
PIC32MZ   P6 |--- SYS_MOSI ---| MOSI (T_DIN in some)    |
         P46 |--- IRQ# -------| IRQ  (T_IRQ in some)    |
         P47 |--- CS# --------| CS   (T_CS in some)     |
--------------/                 \------------------------/
```

The first parameter in the function specifies the model of the touch panel or a command to be executed by the touch panel. Commands should be sent only after the touch panel has been initialised.

Currently supported are:

The first text parameter for the function specifies the display controller model.

Currently supported display controllers are:

"***NULL***"	Null device. Any currently active touch panel is disconnected from the system and the CS# and TIRQ# pins are released.

“***RSPI*46***”	Resistive panels served by controller such as XPT2046, TSC2046, ADS7846, ADS7843, or similar. These controllers use SPI interface and should be connected to the system SPI bus as shown above.
“***RRSPI*46***”	Same as the above but with swapped X and Y axes on the touch panel. This is usually needed when the panel is used in landscape mode.

Once a touch panel has been initialised, the program can send commands to it by using the same function touch().

“***CALIBRATE***” Interactive calibration of the panel. This works only if a display driver is also initialised and active.

The calibration shows sequentially four small white squares in every corner of the panel, and expects the user to touch in the square. Raw values are then taken from the touch controller and processed to produce two calibration values that are returned by the function.

The specific format of the command is:

int cx,cy = touch (“CALIBRATE”);

“***SETCAL***” Set calibration values for touch panel. Once calibrated, the panel can be immediately restored to its calibrated state every time whenever that is needed (usually after system reset).

The command takes two parameters and stores them in internal variables for further processing when the touch panel is used.

The specific format of the command is:

touch (“SETCAL”, int_cx, int_cy);

"***POINTS***" Return the number of points currently being touched on the panel. Resistive panels can only have one point while capacitive panels can support one or more, sometimes as many as 10.

The specific format of the command is:

```
int p = touch ("POINTS");        ' return the number of
                                 ' currently touched points
```

"***READ***" This is the main command that can read information from the touch panel, used in conjunction with "POINTS".

Every time a "READ" command is executed it returns one currently touched point from the panel.

If the panel is calibrated, the returned coordinates are in pixels, otherwise they are raw values from the touch controller.

The specific format of the command is:

int x, y, p = touch ("READ");

Three integer values are returned – coordinate on the X-axis, coordinate on the Y-axis, and pressure. The pressure value is arbitrary and depends on the actual type of the touch panel.

If there are no more active touch points to be read, the "READ" command will return values -1 for the X, Y, and pressure variables. At that point execution of another "POINTS" command will re-initialise the internal data stack with active points, if there are any.

16 GLOSSARY OF THE PIC32MZEF ADDITIONS

These functions are specific for the Rittle implementation for the PIC32MZ microcontroller. Their availability and format are not guaranteed in other system configurations.

Ain clock Din disable display Dout Dtog enable gettime LCD4 port recv setPWM settime sleep srctime touch trmt Vref

17 LICENSE CONDITIONS

Full or partial source codes can be requested for free following the link below.

Request the current source codes

© Konstantin Dimitrov
knivd@me.com