

Pico script

The PICO script is a script based language which is tokenized and interpreted at run-time. The original script and source code is by Jan Verhoeven (<http://jansfreeware.com>). The original script language has been extended with additional functions. Most of the added functionality has to do with the support for the SAA7146A and the FLEXCOP generic driver. To learn more about this PICO script please read the original help file text contents which can be found further on in this document.

Differences as compared to original PICO script language

- Operations involving 'logic'/'bit' operators leave an integer result
- Logic operations give either a False (0) or True (-1) result or act on a condition (≤ 0 is interpreted as being 'true')
- And/or/xor/not are bitwise operators (originally they were 'logic' operators)
Note that the 'not' can only be used to negate a boolean flag if the value of the stack is a 'true' boolean (0/-1). You can use the 'bool' function to create a real boolean.
- There is a single stack ('logical' stack no longer exists) - all operations are now applied to numbers (and not to numbers or logic values separately)
- If a variable is preceded by a '~' then the address of the variable is returned. This can be used for the *readmem* and *writemem* functions. Note that only text variables can typically be used this way. The address returned for a text variable is where the first character starts (if any).

Added functions to original PICO script language

Generic functions added to script language	
bool	Convert to a real boolean. This might be required if, for instance, the 'not' function is used to negate a boolean value. This only works correctly on a true boolean.
	Stack: val - bool (-1 or 0)
delay	Delay for .. us (max. 100000000 == 100 seconds) It is advised to use exact multiples of 1000 if possible since this results in less system load. Any actual 'us' delay (or remainder) will result in a temporarily increase of system load due to the waiting process.
	Stack: us -
shl	Shift left.
	Stack: value shifts -
shr	Shift right.
	Stack: value shifts -
div	Divide (integer result).
	Stack: value1 value2 - trunc(value2/value1)
asinteger	Converts a number string to an integer value (equivalent to 'asnumber' except that this is for integer based numbers, including hexadecimal).
	Stack: text - integer
saystack	Displays the stack contents (only if something on the stack). Can be used for debugging purposes. The last value displayed is the top of the stack. The values are displayed as hexadecimal numbers (if possible).

Generic functions added to script language	
	Stack:
saystackd	Displays the stack contents (only if something on the stack). Can be used for debugging purposes. The last value displayed is the top of the stack. The values are displayed as decimal numbers.
	Stack:
loglevel	<p>Sets the logging level.</p> <p>When a 'log' message is about to be generated it is checked against the current logging level. If it is lower or equal to the logging level then it will be logged, otherwise it will be discarded. Setting the logging level (which is by default -1, meaning 'off') might create a new logging file if logging is made active. The file created can be set with 'logfile'. If the logging level is changed from a valid level to the 'off' level then the logging file is closed (and changing the logging level to an active level will re-create the file making it possible to clear the log programmatically).</p>
	Stack: loglevel -
logfile	<p>Defines the file for logging.</p> <p>By default the file 'picolog.msg' will be generated.</p> <p>Note that this setting is only taken into account when the 'loglevel' is being 'raised' from disabled (<0) to a normal logging level.</p>
	Stack: filename -
logappend	<p>Defines that log information is appended to the logging file or not. By default this is set to 'false', meaning that a new file is created (all old data is lost).</p>
	Stack: flag -
logmessage	<p>Add a message to the logging file (if enabled).</p> <p>Messages are only logged when the 'level' is lower or equal to the active logging level (as set with 'loglevel').</p> <p>The following bits in the level do the following:</p> <p>\$8x : adds the stack contents to the message</p> <p>\$4x : numbers as hexadecimal if possible (default is decimal)</p> <p>\$2x : does not add new line with timestamp</p>
	Stack: level message -
openrfile	<p>Open a file for reading.</p> <p>A 'false' (0) is returned if file could not be opened (e.g. file does not exist).</p> <p>Should be matched with a 'closefile' if the file was opened.</p> <p>Note: If you define a name make sure a double backslash is used to indicate a backslash, e.g. 'scripts\\test.pico' -> a backslash is typically used to include a control code in text.</p>
	Stack: name - handle
openwfile	<p>Open a file for writing (overwrites any existing file).</p> <p>A 'false' (0) is returned if the file could not be created.</p> <p>Should be matched with a 'closefile' if the file was opened.</p> <p>Note: If you define a name make sure a double backslash is used to indicate a</p>

Generic functions added to script language	
	backslash, e.g. 'scripts\\test.pico' -> a backslash is typically used to include a control code in text.
	Stack: name - handle
readfile	Read from file. The first byte on the stack (after 'bytesread') is the first byte read from the file.
	Stack: bytestoread handle - [... ..] bytesread
writefile	Write to a file. Flag is false if not all data could be written for some reason. The byte on top of the stack, after 'bytestowrite' is the first byte written. When 'bytestowrite' is <= 0 then no write operation is performed and a false flag is returned.
	Stack: [... ..] bytestowrite handle - flag
writefilestr	Write a string to a file. Flag is false if not all data could be written for some reason. The byte on top of the stack, after 'bytestowrite' is the first byte written. When the string is empty then no write operation is performed and a false flag is returned.
	Stack: text handle - flag
filepos	Set the position into a file (and returns the new position). Returns \$FFFFFFFF if an error is detected.
	Stack: position handle - newposition
closefile	Close a file.
	Stack: handle -
readmem	Read a byte from the indicated address.
	Stack: address - byte
writemem	Write a byte to the indicated address.
	Stack: byte address -

Driver related functions added to script language	
(all parameters are DWORD based unless stated otherwise – also see the manual of the driver)	
_saa7146a	Select SAA7146A driver. All driver related functions will be passed on to the SAA7146A driver. Note: Handles are related to the driver they access, so changing the selected driver has to be done with care.
	Stack: -
_?saa7146a	Returns the selection of the SAA7146A driver,
	Stack: - flag
_flexcop	Select FLEXCOP (B2C2) driver.

Driver related functions added to script language (all parameters are DWORD based unless stated otherwise – also see the manual of the driver)	
	<p>All driver related functions will be passed on to the FLEXCOP driver. Note: Handles are related to the driver they access, so changing the selected driver has to be done with care.</p>
	Stack: -
_?flexcop	<p>Returns the selection of the FLEXCOP driver.</p>
	Stack: - flag
_version	<p>Get version information of driver</p>
	<p>Stack: handle - name build minor major true Stack: handle – false</p> <p>Build: date which is defined as \$YYYYMMDD YYYY = Year MM = Month DD = Day</p>
_cards	<p>Return number of cards detected.</p>
	Stack: - cards
_handlecreate	<p>Create handle for active driver. Note: Although it is advised to release the created handle through ‘_handleclose’, this will also be done automatically when the script is terminated. When the script terminates all the handles acquired through this function are closed (this means that others using this handle can no longer use it too!).</p>
	Stack: card - handle
_handleclose	<p>Close handle.</p>
	Stack: handle -
_cardsubsys	<p>Return the SUBSYS (PCI) identifier of the indicated card. Returns 0 for any error.</p>
	Stack: card - subsys
_notifywait	<p>Wait for notification. Wait for a notification (interrupt). Only when an interrupt, which must have been ‘activated’, is generated this call will return. Thus, this call waits until this occurs. The only way to recover for this, besides an actual interrupt, is to manually trigger an event using ‘_notifygenerate’ (using a DIFFERENT script since this script is still waiting!). Note: the driver only supports a single ‘_notifywait’. If more than a single ‘_notifywait’ is issued only the first is ‘acknowledged’. The others will wait for the next generated event.</p>
	Stack: handle - flag
_notifygenerate	<p>Generate a manual notification.</p>

Driver related functions added to script language	
(all parameters are DWORD based unless stated otherwise – also see the manual of the driver)	
	<p>Note: A manual notification can only be used with a -different- handle then the one which is possibly waiting for a notification. This typically means that a second handle needs to be created. Secondly, the same script can't be used for this because the script is still waiting for the notification to occur.</p> <p>Stack: handle - flag</p>
_rd	<p>Read from chipset register.</p> <p>Stack: address handle - data true Stack: address handle - false</p>
_wr	<p>Write to chipset register.</p> <p>Stack: data address handle - flag</p>
_dmastatus	<p>Read DMA status.</p> <p>Stack: dmabuffer handle - fifooverflows size physicaladdress virtualaddress isr irqs true Stack: dmabuffer – false</p> <p>DmaBuffer: DMA buffer to get information for Fifooverflows: Global number of FIFO overflows. An overflow is indicated when a FIFO buffer is written to by the driver, without having the FIFO buffer read by an application. Size: Size of DMA buffer in bytes. PhysicalAddress: Physical address of DMA buffers VirtualAddress: Virtual address of DMA buffer. You need this address when FIFO buffers are to be used. FIFO buffers can only reference to memory allocated for DMA. Isr: Interrupt status register of last occurred interrupt (e.g. FLEXCOP register \$20C or SAA7146A register \$10C) Irqs: Interrupt occurs (total interrupts for driver).</p>
_dmaallocate	<p>Allocate DMA memory.</p> <p>Stack: dmasize handle - size physicaladdress virtualaddress dmabuffer true Stack: dmasize handle – false</p> <p>DmaSize: Size in bytes of contiguous memory to allocate. Size: Size in bytes of buffer. PhysicalAddress: Physical address of buffer. VirtualAddress: Virtual address of buffer. This address is to be used when FIFO buffers are allocated. These FIFO buffers are only allowed to use the memory allocated using this call. Note: Windows 98 allows using this address to access the allocated memory directly. However, this does not work with W2000. It is advised, for Windows 98 also, to use '_dmard' and '_dmawr' to access the allocated memory. DmaBuffer: Identification of buffer. This identification is used to de-allocate the memory or getting status information for it.</p>
_dmarelease	<p>Release DMA buffer.</p> <p>Stack: dmabuffer handle – flag DmaBuffer: Identification of buffer. This identification was returned when allocating</p>

Driver related functions added to script language (all parameters are DWORD based unless stated otherwise – also see the manual of the driver)	
	<p>the memory. To clear all DMA buffers, just start at dmabuffer 0 and keep calling this function while increasing dmabuffer. When it fails (at buffer 256 which does not exist) all DMA buffer is released.</p>
_dmard	<p>Read from DMA memory.</p>
	<p>Stack: size targetindex sourceindex address dmabuffer handle – flag</p> <p>Size: Number of bytes to transfer. TargetIndex: Index in bytes into target buffer to start copying to. SourceIndex: Index in bytes into source (DMA memory) to start copying from. Address: Pointer to target buffer. DmaBuffer: Identification of buffer. This is the identification returned when '_dmaallocate' is used.</p>
_dmawr	<p>Write to DMA memory.</p>
	<p>Stack: size targetindex sourceindex address dmabuffer handle – flag</p> <p>See '_dmard' except that Address is the pointer to the source buffer.</p>
_fifoallocate	<p>Allocate buffers from DMA allocated memory.</p>
	<p>Stack: length address1 address0 buffers handle - identifier true Stack: length address1 address0 buffers handle - false</p> <p>Length: Size in bytes of the memory to be copied. This is also the size of the FIFO buffers being allocated. Address0/1: Address of source data (data which is to be copied to the FIFO buffers). This address must be within the range of driver allocated memory (i.e. allocated with '_dmaallocate'). For the FLEXCOP driver Address0 refers to sub buffer 0, and Address1 to sub buffer 1. The SAA7146A only uses Address0. The selection of the sub buffer (for FLEXCOP) to use is done by the driver (based on which buffer is being written when an interrupt occurs). Note: For the FLEXCOP It is essential that the application writes or reads the sub buffer addresses at least once (eg. addresses \$000 and \$0010) since these values are preserved and used in the process. If the sub buffer addresses are never used then the sub buffer addresses are unknown (the driver does not read it specifically). Buffers: Number of FIFO buffers to allocate. These all use the same memory as source. When a FIFO buffer is written to, then the next FIFO in the list will be used (assuming buffers > 1). When the last FIFO buffer is written to, then the first FIFO buffer is used as next target.</p>
_fiforelease	<p>Release allocate buffers.</p>
	<p>Stack: buffers identifier handle - buffers identifier true Stack: buffers identifier handle - false</p> <p>Buffers: Number of FIFO buffers to release. The driver will make sure that only the correct amount of buffers to release are</p>

Driver related functions added to script language (all parameters are DWORD based unless stated otherwise – also see the manual of the driver)	
	<p>used. To release all FIFO buffers use 256 (or a higher value). Note: you should typically use the same number of buffers as have been allocated. Identifier: FIFO buffer identification (first buffer) to release. The driver will make sure that an invalid entry will be within the allowed range. This means that a negative value will effectively use the very first buffer. A very high value will use the last buffer. To release all FIFO buffers use -1.</p>
_fiford	<p>Read from buffer</p> <p>Stack: length address identifier handle - irqoverflows irqsfifooverflows order valid length true Stack: length address identifier handle - false</p> <p>Length: Size of target buffer. Must be at least the size of the allocated FIFO buffer. The whole FIFO buffer is always being copied. Address: Target buffer to where data from the FIFO buffer is copied to. Identifier: FIFO buffer identification to read data from. IrqOverflows: Overflows counted on FIFO's using the same interrupt source. Irqs: Total number of interrupts generated for the interrupt source as used by the FIFO. FifoOverflows: Overflows counted on this particular FIFO buffer. Order: Order number of the buffer being written. Every time a FIFO buffer is written to it gets a number that is increased for every write using the same interrupt source. Valid: Indicates if the FIFO contains valid data. If FALSE it indicates that the FIFO has not been written to since it has been read. Length: Number of actual bytes transferred. This is always the size of the allocated FIFO buffer.</p>
_irqwr	<p>Write IRQ handling.</p> <p>Stack: lastbuffer firstbuffer signalxor signalor signaland signalreg autodisable usefifo usesignalling useevent activate irq handle - flag</p> <p>LastBuffer: The last FIFO buffer to write data to when an interrupt is generated. Must be \geq fifoBufferFirst. FirstBuffer: The first FIFO buffer to write data to when an interrupt is generated. SignalXor: The XOR (invert) value applied to the ANDed and Ored contents as read from the register (signalRegister). SignalOr: The OR value applied to the ANDed contents as read from the register (signalRegister). SignalAnd: The AND value applied to the contents as read from the register (signalRegister). SignalReg: Register to read and write back when the signaling mechanism is active (signalActive = TRUE). AutoDisable: SAA7146A only! When TRUE the interrupt will be disabled once it has</p>

Driver related functions added to script language (all parameters are DWORD based unless stated otherwise – also see the manual of the driver)	
	<p>been activated. Some interrupts of the SAA7146 will issue a new interrupt when not properly cleared. This settings allows that the interrupt is only generated once. The driver disables the interrupt by writing to the interrupt enable register of the SAA7146.</p> <p>UseFifo: Indicates that FIFO buffering is to be used.</p> <p>UseSignalling: Indicates that the signaling mechanism is to be used. The signaling mechanism will write specific data to a FLEXCOP register.</p> <p>UseEvent: Indicates if a call which waits for an event is to be notified (_notifywait).</p> <p>Activate: Activates the interrupt or not. This is the global activation of the interrupt. When FALSE the other activations are not used.</p> <p>Irq: Interrupt number (0..11 for FLEXCOP, 0..31 for SAA7146A). See the FLEXCOP/SAA7146A interrupt status register for what each interrupt stands for.</p>
_irqrd	Read IRQ status.
	<p>Stack: irq handle - overflows allbuffercnt lastbuffer irqs true</p> <p>Stack: irq handle - false</p> <p>Overflows: Counter incremented when an overflow is detected. An overflow is detected when a FIFO buffer is written to without that it has been read.</p> <p>AllBufferCnt: Counter incremented each time when the first FIFO buffer is used.</p> <p>LastBuffer: The last written FIFO buffer.</p> <p>Irq: Number of interrupts detected of this type when interrupt is active.</p>

GoPico

The following text is mainly from the original Pico help file. It has been updated with some of the additional functions which has been added to the script language. The driver related functions are not added to the text. The changed text can be identified by the same background color as this text.

GoPico

GoPico is the demo application for the delphi TjanPico component that runs the Pico script language. Use TjanPico to add light-weight semi-compiled scripting to your delphi applications: include scripts, (local and global) variables (plain and structured), (nested) function definitions, function calls, classes (multiple inheritance), objects (with methods and properties), program flow (if, else, case, while, repeat), dozens of build-in functions (string, date, conversion, testing, interface etc.) and external (Delphi) functions.

Note: GoPico has been re-compiled with the added functions. However, the GoPico help file has not been updated with these changes.

License

TjanPico may be freely used in both freeware and commercial projects provided you do not claim that you are the author of TjanPico.

TjanPico may be freely changed providing you leave the LICENSE NOTE in the source code of the janPico.pas unit untouched.

Motivation

Motivation

Since 1977 I have been writing script languages. Just for fun. The first language was a Forth style language written for the 6502 processor in my PET Commodore.

Pico shares with Forth the stack based approach, allowing for very compact scripting.

With JScript and VBScript installed on every computer using Windows there seems to be no need for yet another scripting language. It is easy to use JScript or VBScript from within a Delphi program. But many times you want your scripting language to have very specific features that are just not part of the languages coming with Windows. That is where Pico comes in, or any other native Delphi scripting object. You can modify Pico to suit your own needs as a programmer, providing your users with an optimum programming interface.

Updates

Updates

GoPico is written by Jan Verhoeven with Delphi 5.

Email

ian1.verhoeven@wxs.nl

WebSite

<http://iansfreeware.com>

Getting started

Getting started

GoPico is a simple program for testing and demonstrating the **Pico** language.

You can enter a script in the syntax highlighted editor and either press the **F9** key or click the **Run** button.

A script can be saved with **File Save As** and subsequently with **File Save** or by pressing **Ctrl+S**. Please note that GoPico automatically adds the file extension **.pico** to any script that you save.

You can also load and directly run a script with **Run - Load and Run** or by pressing **Ctrl+F9** and selecting a script file.

Calling script functions

Calling script functions

Very often you want to run a function within a script from Delphi. This is possible with the **executeFunction** method.

executeFunction

```
procedure TjanPico.executeFunction(FuncName:string);
```

Before you use **executeFunction** you must have executed the script using:

```
function TjanPico.execute: TjanPicoObject;
```

This way you can use Pico in e.g. the THTMLViewer from Dave Baldwin (www.pbear.com).

The THTMLViewer component provides an **onScript** event with the script source as one of the arguments. You can run this script source in a TjanPico instance. Functions within the script can then be called in response to HTML Form Element events like, **onClick**, **onChange** etc.

Script basics

Script basics

Pico supports variables, flow control, user defined functions, external functions and a range of internal functions and operators.

Pico syntax

Pico uses the Reverse Polish Notation (RPN) that is also used in HP calculators: you enter values (operands) before the operator.

In most languages you would store the result of an addition as:

```
price=2*3
```

In Pico:

```
2 3 * !price
```

In most languages special characters are used to separate the parts of a statement. In Pico the space character is used. Pico also ignores line breaks. In fact when you assign the script all line breaks are replaced by a space character before

the script is parsed.

Pico is stack based. The result of an operation is pushed on the stack and every operation expects the operands on the stack. There is no such thing as function parameters and arguments. Before you call a function you must put any arguments on the stack and retrieve any result from the stack after the function returns.

In many languages you use parenthesis to change the order of evaluation. In Pico there are not parenthesis and evaluation is determined by the order of operands and operators. This needs getting used to but is very flexible.

Like many scripting languages, Pico has a single data type: the variant. You do not specify a data type. Values pushed on the stack or stored in a variable can be: integer, string, floating point, boolean, date etc.

Pico performance

Pico is semi-compiled and runs fast, specially when you have many loops in your script.

Semi-compiled

Pico is semi-compiled. The parser translates your script into tokens. Tokens are stored in a token tree. Statements between { and }, are stored as child tokens of the token before the block. The way the semi-compiled script is stored in memory, mimics the structure of the language.

Stacks

Pico is stack based and used to distinct stacks: the execution stack and the logical stack. The result of tests (== != > etc) is pushed on the logical stack. Program flow statements (if else while case) take their value from the logical stack. All other operations use the execution stack.

Results of tests now use the execution stack. The logical stack is now only used internally.

Security

Security

For security reason's the Pico language has no native statements to interface with the hard disk. As a programmer you can extend the language via the external directive `@externalfunction`, (see [Functions](#) and provide the user access to the hard disk via an external function.

Includes

Includes

Pico allows you to include other scripts into your script.

```
<<script>>
```

The `script` is the name of the script file without the `.pico` extension. E.g. `<<library>>` will try to include the `library.pico` file.

Includes can be nested. This means that you can have includes in included scripts. Every script is included only once. Pico keeps track of included scripts when it handles the script.

Comments

Comments

You can use c-style comments in your Pico scripts:

```
/* this is a comment */
```

Anything between `/*` and `*/` will be removed from the script before it is parsed.

Literals

Literals

Pico supports 3 kinds of literals: number, string, date and block.

String literals

String literals must be enclosed by either single or double quotes. The begin quote must match the end quote.

Examples:

```
"Jan Verhoeven"
```

```
'JanSoft'
```

```
"SELECT 'name' FROM Users WHERE UserId=4"
```

You can use the `\` escape character within a string:

- `\n` will be replaced by the new line character
- `\t` will be replaced by the tab character

Any other character after the `\` character will be included literally. To include the `\` character itself in a string you will have to use `\\`.

Example:

```
"This is a \"string\"\\nNext line"
```

Number literals

Numbers are entered without surrounding quotes. Use the dot as decimal separator.

Hex literals

Hexadecimal number literals must be prefixed with the dollar sign:

```
$FF
```

Date literals

Date literals are entered in the ISO-8601 format: YYYYMMDD preceded by the `#` character:

followed by four digits for the year, followed by 2 digits for the month, followed by 2 digits for the day.

Examples:

```
#20030921
```

```
#19531116
```

Variables

Variables

Pico variables can hold any variant value (numeric, string, boolean, date). Variables are scoped to the function within which they are defined. Variables defined outside a function are global and can be accessed from within a function. All variables must be defined before they are used.

Defining a variable

In Pico a variable is defined with:

```
var:variablename
```

This initializes the variable with the value **null**.

Storing a value in a variable

In Pico a value is stored in a variable with:

```
value !variablename
```

You will get an error message if the variable is not defined.

Retrieving the value of a variable

You push the value of a variable on the stack with:

```
?variablename
```

Retrieving the address of a variable

You can get the address of a variable on the stack with:

```
~variablename
```

Note that only text variables are typically to be used this way. The address fro a text variable points to the first character of the string.

Typically used in combination with the readmem and writemem functions to manipulate characters in a string directly.

The next demo illustrates this, were the 'X' is replaced by a '0'

```
var:Demo
```

```
"PICX" !Demo 79 ~Demo 3 + writemem
```

Incrementing a variable by one

Because incrementing a variable by one there is a special shortcut for this:

```
++variablename
```

Decrementing a variable by one

Because decrementing a variable by one there is a special shortcut for this:

```
--variablename
```

Adding to a variable

```
v1 +=variablename
```

Directly adds v1 to variablename

Subtracting from a variable

```
v1 -=variablename
```

Directly subtracts v1 to variablename

Multiplying a variable

```
v1 *=variablename
```

Directly multiplies variablename with v1.

Dividing a variable

`v1 /=variablename`

Directly divides variablename by v1.

Comparing a variable

`v1 ==variablename`

Tests if v1 equals the value of variablename.

`v1 !=variablename`

Tests if v1 does not equal the value of variablename.

Record Variables

Record Variables

Pico allows you to manipulate regular variables as records. similar to the css style format used with HTML.

Setting a record field value

`value !!variablename.fieldname`

Getting a record field value

`??variablename.fieldname`

Example:

`var:data`

`" " !data`

`"Jan" !!data.name`

`??data.name`

A record variable is a normal variable with a string value. Data is stored in the same way as in CSS specifiers used with HTML:

`name1:value1;name2:value2`

You can set and retrieve the complete record with:

`!variablename`

`?variablename`

Please note that you can only store string values in a record field. Use the [Conversion functions](#) when needed.

When you try to store a number in a record field, Pico will convert the number to a string. When you retrieve the value of a field, Pico will return a string. If you want to operate on the value as a number then you must use **asnumber** to convert it into a number.

System Variables

System Variables

Pico provides a list of system variables that you can access from anywhere in your script. You can set and get system variables without using the **var:variablename**.

Because system variables start with **system.** you are not allowed to create an object named **system**.

Setting a system variable value

`v1 !system.variablename`

If the variable does not exist it is added.

Getting a system variable value

`?system.variablename`

If the variable does not exist then 0 is returned.

Predefined system variables

When you execute a script the following system variables are always available:

?system.path

Returns the application path.

?system.version

Returns the version of Pico as a floating point value.

openfile

Open a file for reading.

The filename to be opened for reading is pushed on the stack before calling this function. The function returns the opened handle of the file, which is to be used with the other file operation functions.

A 'false' (0) is returned if file could not be opened (e.g. file does not exist), otherwise the file handle is returned, which is to be used with the other file functions.

Should be matched with a 'closefile' if the file was opened.

Note: If you define a name make sure a double backslash is used to indicate a backslash, e.g. 'scripts\\test.pico' -> a backslash is typically used to include a control code in text.

```
"afile.txt" openrfile
```

openwfile

Open a file for writing (overwrites any existing file).

The filename to be opened for writing is pushed on the stack before calling this function. The function returns the opened handle of the file, which is to be used with the other file operation functions.

A 'false' (0) is returned if the file could not be created, otherwise the file handle is returned, which is to be used with the other file functions.

Should be matched with a 'closefile' if the file was opened.

Note: If you define a name make sure a double backslash is used to indicate a backslash, e.g. 'scripts\\test.pico' -> a backslash is typically used to include a control code in text.

```
"afile.txt" openwfile
```

closefile

Closes a file as opened by openfile or openwfile.

The file handle to be close is pushed on the stack before calling this function.

```
Filehandle closefile
```

readfile

Read from file.

The number of bytes to read and the file handle are placed on the stack before calling the function. The result are the

bytes read, preceded by the number of bytes read. The first byte on the stack (after 'bytesread') is the first byte read from the file.

```
25 readhandle readfile
```

writefile

Write to a file.

The bytes to write, the number of bytes to write and the file handle are placed on the stack before calling the function. The resulting flag indicates the success or failure of the operation.

When the number of bytes to write is ≤ 0 then no write operation is performed and a false result is returned.

```
21 20 19 18 4 writehandle writefile
```

filepos

Set the position into a file (and returns the new position). The required position and the file handle are placed on the stack before calling the function. Returns \$FFFFFFF if an error is detected, otherwise the new position is returned.

```
128 readhandle filepos
```

readmem

Read the contents (a byte) of a memory location and leave it on the stack.

```
128 readmem
```

writemem

Write a byte to a memory location.

```
$80 128 writemem
```

List functions

List functions

List functions work on a list of strings. You can enter a list of strings as a literal using a new line `\n` escape character or you can build a list of strings using the `cr` constant.

first[]

Returns the first string of the list:

```
list first[]
```

last[]

Returns the last string of the list:

```
list last[]
```

count[]

Returns the number of strings in the list:

```
list count[]
```

index[]

Finds the index of a string in the list.

```
list value [index]
```

Returns -1 when the value is not found.

pick[]

Returns the **index** string of the list. Index runs from 0..count-1

```
list index pick[]
```

An error is raised when the index is out of bounds.

append[]

Appends a string to the list.

```
list newstring append[]
```

delete[]

Deletes string at index:

```
list index delete[]
```

An error is raised when the index is out of bounds.

name[]

Returns the name part of the **index** string of the list. Index runs from 0..count-1

```
list index name[]
```

Both the **name[]** and following **value[]** statement work on strings in the **name=value** format.

An error is raised when the index is out of bounds.

value[]

Returns the value part of the string in the list with name **name**.

```
list name value[]
```

Example:

```
var:list var:i var:c

"one\n two\n three\n four\n" !list
?list count[] !c

0 !=c
if {
  0 !i
  ?c ?i <
  while {
    ?list ?i pick[] trace
    ++i
    ?c ?i <
  }
}
```

Please note that there is a space behind every **\n**. This is not needed when the **\n** is add the end of a line.

Functions

Functions

In Pico script you can define functions:

```
function:functionname() {  
    ... any statements  
}
```

Functions can be defined anywhere in the script, before or after from where they are called.

Nested functions

You can define functions within a function. Functions are scoped. This means that when you call a function from within a function Pico first checks for the called function within the scoped function. If not found then the search continues one level up.

Calling a function

You call a function with:

```
functionname
```

Any arguments must be pushed on the stack before you call the function.

Calling external functions

You can call an external function with:

```
@externalfunctionname
```

Example:

```
@system.path
```

Classes and Objects

Classes and Objects

Pico allows you to create custom classes and instances of those classes.

Defining a class

You define a class with:

```
class:classname() {  
    ..any functions  
}
```

Classes can contain functions, variables and statements. Classes must be defined before objects based upon the class are defined. A Class may inherited from another class:

```
class:classname(parentClassName) {  
    ..any functions  
}
```

Class variables

Any variables defined within a class are created within the **object instance** of a class when the object is initialized.

Defining an object

You define an object with:

```
object:objectname(classname) {  
    ..any variables, statements and functions
```

```
}
```

Objects can contain everything a function can contain: variables, statements and functions.

You can also define an object without a class name:

```
object:objectname() {  
    ..any variables, statements and functions  
}
```

Example:

```
class:car() {  
    var:index  
    function:price() {  
        123.45  
    }  
  
    function:wheels() {  
        4 ?index +  
    }  
}
```

```
object:mazda(car) {  
  
    12 !index  
}
```

```
mazda.wheels
```

With `mazda.wheels` you are calling the `wheels` function of the `mazda` object. Pico checks if `mazda` has a function named `wheels`. If not Pico checks if `car` has a function (method) named `wheels`. If so it is executed otherwise an error is reported.

When you call `mazda.wheels` Pico first executes `mazda`. This means that any non-function statements of `mazda` are executed. Next the defined method `wheels` is executed.

Overriding class methods

Because of the way Pico handles object.method calls, you can override a class method by re-writing that method within the body of the object.

Object initialization

In Pico an object is initialized the first time it is used. Initialization has the following steps:

- First the class hierarchy of the object is initialized
- Next the object itself is initialized

Initialization means: **running** a class or object. This defines and initializes any variables and runs any initialization code.

Object properties

Object variables are the properties of the object.

You can set the value of an object property with:

```
v1 !objectname.propertyname
```

And get the value of an object property with:

```
?objectname.propertyname
```

When you set or get the value of an object property, Pico makes sure that the object is initialized.

Property access methods

Pico allows you to specify optional property access methods.

The optional property set method must be specified as:

```
function:set_propertyname() {  
}
```

The optional property get method must be specified as:

```
function:get_propertyname() {  
}
```

When you set or get the value of an object property, Pico checks if a property get or set method exists. If such a method exists it is executed. If there is not a property get or set method then Pico will directly access the object variable. Many times you may want to have direct read access to an object variable and use a property set method to do any range checking.

Class Inheritance

Class Inheritance

In Pico a class can (recursive) inherit from another class.

```
class:generic() {  
}
```

```
class:specific(generic) {  
}
```

You can also inherit a class from more than one class. This is called multiple inheritance.

```
class:thing() {  
}
```

```
class:color() {  
}
```

```
class:toy(thing,color) {  
}
```

```
object:doll(toy) {  
}
```

When you inherit a class from more than one other class, then provide a comma separated list of parent classes. This list may **not** contain any spaces.

Which method is executed?

When a object method is called, Pico searches in the class hierarchy for the method. If a class inherits from from than one class (multiple inheritance) then Pico searches the list of parent classes in the order they appear.

```
class:toy(thing,color) {  
}
```

If both the **thing** class and the **color** class would have a method named **creator** then the **creator** method of the **thing** class would be executed. So when inheriting a class from several other classes you must pay attention to the order in which you list the classes.

Object references

Object references

You can use variables to store a reference to an object:

```
class:cars() {  
  function:wheels() {  
    4  
  }  
}  
  
object:ford(cars) {  
  
}  
  
var:my  
ford !my  
my.wheels
```

By stating the plain name of an object, a reference to that object is pushed on the stack and can subsequently be stored in a variable.

Program Flow

Program Flow

In Pico you can alter the program flow. Without that Pico would not be very useful.

if

Executes a block of the top of stack is true:

```
if {  
  ..any statements  
}
```

if else

An **if** condition can also have an optional **else** clause:

```
if {  
  ..any statements  
}  
else {  
  ..any statements  
}
```

```
}
```

case

Enables the execution of one or more statements when a specified expression's value matches a label.

```
v1 case {  
  label1 { ..statements }  
  label2 { ..statements }  
  label3 { ..statements }  
  label4 { ..statements }  
}
```

The labels must be integer or string constants.

Note: Comparison internally is string based. This means that `label1` etc. are used as a string, so they must be exactly the same as when `v1` is converted into a string. This means no leading zeros, no hexadecimal etc.

Example:

```
3  
case {  
  1 { "low" }  
  2 { "medium" }  
  3 { "high" }  
  4 { "extreme" }  
}
```

Example (used incorrectly):

```
3  
case {  
  $1 { "low" }  
  $2 { "medium" }  
  $3 { "high" }  
  $4 { "extreme" }  
}
```

while

Pico has a single loop statement:

```
true while {  
  ..any statements  
}
```

Until expects a boolean value. So your last statement in the `{ }` block must be a logical test.

The starting `while` expects a true condition (<0), otherwise the `while { }` block is not executed.

Note: If the closing bracket of the while is preceded by another closing bracket, then a nop (`';`) is required in between them. A warning will be issued when this is neglected.

```
true while {  
  { ..any statements }  
  ; /* Required! */  
}
```

repeat

Executes a block of statements n times

```
n repeat {  
  ..any statements
```

```
}
```

Convenient when you know the number of desired repeats. If the number of repeats is ≤ 0 then nothing of the repeat between `{ }` will be executed.

Note: If the closing bracket of the repeat is preceded by another closing bracket, then a nop (`';`) is required in between them. A warning will be issued when this is neglected.

```
n repeat {  
    { ..any statements }  
    ; /* Required! */  
}
```

Operators

Operators

Pico has a whole range of mathematical operators. The operators pull there operands from the stack and push the result back on the stack.

+
`v1 v2 +`
Adds v2 to v1.

-
`v1 v2 -`
Subtracts v2 from v1.

`v1 v2 *`
Multiplies v1 by v2.

`v1 v2 \`
Divides v1 by v2.

^
`v1 v2 ^`
Raises v1 to the power v2.

mod
`v1 v2 mod`
Returns v1 modulo v2.

The mod operator returns the remainder obtained by dividing its operands. E.g. `12 5 mod` return 2.

div
`v1 v2 div`
Returns the integer result of `v1/v2`.

round
`v1 round`

Returns the value of v1 rounded to the nearest whole number.

max

v1 v2 max

Returns the greater of v1 and v2.

min

v1 v2 min

Returns the lesser of v1 and v2.

abs

v1 abs

Returns an absolute value.

neg

v1 neg

Returns the negated value. E.g 5 returns -5, -4.32 returns 4.32

sqr

v1 sqr

Returns the square of a number.

sqrt

v1 sqrt

Returns the square root of a number.

exp

v1 exp

returns the value of e raised to the power of v1.

ln

v1 ln

Returns the natural log of a real expression.

log

v1 log

Returns log base 10 of v1.

String operators

String operators

Pico supports a single string operator.

&

Concatenates 2 strings together.

s1 s2 &

Example:

"Jan " "Verhoeven" &

Results in Jan Verhoeven

Comparison operators

Comparison operators

The following comparison operators can be used in logical tests.

==

`v1 v2 ==`

Tests if v2 equals v1.

!=

Test if v2 is not equal to v1.

>

`v1 v2 >`

Tests if v2 is greater than v1.

>=

`v1 v2 >=`

Tests if v2 is greater than or equal to v1.

<

`v1 v2 <`

Tests if v2 is less than v1.

<=

`v1 v2 <=`

Tests if v2 is less than or equal to v1.

Logical operators

Logical operators

The following Logical operators can be used. Operators pull their operands from the stack and push their result back on the stack. **The numbers on the stack are first converted to integer values (rounded) before processing.**

and

`v1 v2 and`

Returns true when both operands are true, otherwise returns false.

Performs a binary AND function.

or

`v1 v2 or`

Returns true when at least one of both operands is true, otherwise returns false.

Performs a binary OR function.

xor

`v1 v2 xor`

Returns true when at one of both operands is true and the other one is false, otherwise returns false.

Performs a binary XOR function (each bit set in V2 will invert the associated bit in V1).

not

`not`

Inverts the top of stack value. True becomes false, and false becomes true.

Performs a binary NOT function (each bit is inverted).

bool

`bool`

Convert to a real boolean. This might be required if, for instance, the 'not' function is used to negate a boolean value, which only works correctly on a real boolean.

shl

`v1 v2 shl`

Performs a binary shift operation to the left. V1 is shifted V2 times to the left.

shr

`v1 v2 shr`

Performs a binary shift operation to the right. V1 is shifted V2 times to the right.

Stack Operators

Stack Operators

Because Pico is stack based it has some stack operators.

dup

Duplicates the top of stack value.

drop

Drops the top of stack value.

swap

Swaps the 2 top of stack values.

Trigonometry functions

Trigonometry functions

In Pico you can use the Trigonometry functions listed below. These functions take the top of stack as their argument and push the result on the stack.

sin

Returns the sine of the angle in radians.

cos

Calculates the cosine of an angle in radians.

tan

Returns the tangent of an angle in radians.

arctan

Calculates the arctangent of a given number.

Constants

Constants

In Pico you can use the following named constants.

pi

Returns 3.1415926535897932385.

true

Returns boolean true.

false

Returns boolean false.

maxint

Returns the highest value in the range of the Integer data type (2147483647).

Can be used in e.g. the string function **mid** to return all characters up to the end of a string:
`"jan verhoeven" maxint 5 mid`

cr

Returns a carriage return.

Date functions

Date functions

Pico supports the data functions listed below. See [Literals](#) for the way to enter literal dates.

date

Returns the current date.

now

Returns the current date and time.

year

Returns the year of a date:

`v1 year`

month

Returns the month of a date:

`v1 month`

day

Returns the day of a date:

`v1 day`

weekday

Returns the ISO day of the week where monday=1 and sunday=7

`v1 weekday`

weeknumber

Returns the week number of a date.

`v1 weeknumber`

easter

Returns the easter date of a given integer year.

`v1 easter`

formatdate

Formats a date to a string using a format specifier.

`v1 "format" formatdate`

Example:

`#19531116 "dd-mm-yyyy" formatdate`

Use **d** for day, **m** for month and **y** for year.

c

Displays the date using the format given by the ShortDateFormat global variable, followed by the time using the format given by the LongTimeFormat global variable. The time is not displayed if the fractional part of the DateTime value is zero.

d

Displays the day as a number without a leading zero (1-31).

dd

Displays the day as a number with a leading zero (01-31).

ddd

Displays the day as an abbreviation (Sun-Sat) using the strings given by the ShortDayNames global variable.

dddd

Displays the day as a full name (Sunday-Saturday) using the strings given by the LongDayNames global variable.

dddddd

Displays the date using the format given by the ShortDateFormat global variable.

ddddddd

Displays the date using the format given by the LongDateFormat global variable.

m

Displays the month as a number without a leading zero (1-12). If the m specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.

mm

Displays the month as a number with a leading zero (01-12). If the mm specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.

mmm

Displays the month as an abbreviation (Jan-Dec) using the strings given by the ShortMonthNames global variable.

mmmm

Displays the month as a full name (January-December) using the strings given by the LongMonthNames global variable.

yy

Displays the year as a two-digit number (00-99).

yyyy

Displays the year as a four-digit number (0000-9999).

h

Displays the hour without a leading zero (0-23).

hh

Displays the hour with a leading zero (00-23).

n

Displays the minute without a leading zero (0-59).

nn

Displays the minute with a leading zero (00-59).

s

Displays the second without a leading zero (0-59).

ss

Displays the second with a leading zero (00-59).

z

Displays the millisecond without a leading zero (0-999).

zzz

Displays the millisecond with a leading zero (000-999).

t

Displays the time using the format given by the ShortTimeFormat global variable.

tt

Displays the time using the format given by the LongTimeFormat global variable.

am/pm

Uses the 12-hour clock for the preceding h or hh specifier, and displays 'am' for any hour before noon, and 'pm' for any hour after noon. The am/pm specifier can use lower, upper, or mixed case, and the result is displayed accordingly.

a/p

Uses the 12-hour clock for the preceding h or hh specifier, and displays 'a' for any hour before noon, and 'p' for any hour after noon. The a/p specifier can use lower, upper, or mixed case, and the result is displayed accordingly.

ampm

Uses the 12-hour clock for the preceding h or hh specifier, and displays the contents of the TimeAMString global variable for any hour before noon, and the contents of the TimePMString global variable for any hour after noon.

/

Displays the date separator character given by the DateSeparator global variable.

:

Displays the time separator character given by the TimeSeparator global variable.

'xx'/"xx"

Characters enclosed in single or double quotes are displayed as-is, and do not affect formatting.

delay

Delay for .. us (max. 100000000 == 100 seconds)

It is advised to use exact multiples of 1000, if possible, since this results in less system load. Any actual 'us' delay (or remainder) will result in a temporarily increase of system load due to the waiting process. This function first handles any millisecond delay (without adding any system load) and then does the eventual remaining us delay (which does add to the system load).

```
1000 delay
```

String functions

String functions

Pico supports the following string functions.

length

```
v1 length
```

Returns the length of v1.

uppercase

```
v1 uppercase
```

Returns v1 in uppercase.

lowercase

```
v1 lowercase
```

Returns v1 in lowercase.

left

```
v1 count left
```

Returns the **count** left characters of v1.

right

```
v1 count right
```

Returns the **count** right characters of v1.

mid

```
v1 count from mid
```

Returns **count** characters of v1 starting at **from**.

posstr

```
v1 substring index posstr
```

Returns the position of **substring** within **v1**. Search is case sensitive and starts at **index**.

postext

```
v1 substring index postext
```

Returns the position of **substring** within **v1**. Search is case in-sensitive and starts at **index**.

replacestr

Case-sensitive replaces all occurrences of **FindString** in **SourceString** with **ReplaceString**.

```
sourcestring findstring replacestring replacestr
```

Example:

```
"jan verhoeven" "jan" "Jan" replacestr
```

replacetext

Case-insensitive replaces all occurrences of **FindString** in **SourceString** with **ReplaceString**.

```
sourcestring findstring replacestring replacetext
```

Example:

```
"jan verhoeven" "Jan" "Johannes" replacetext
```

split

Splits a source string in lines on a split string:

```
sourcestring splitstring split
```

Allows you to apply the [List functions](#) after the split.

join

Joins a series of lines into a string using the joinstring:

```
sourcestring joinstring join
```

Regular expressions

Regular expressions

Pico support regular expression testing and replacing.

regtest

```
SourceText RegularExpression regexpr.test
```

Return true if SourceText matches the RegularExpression.

regreplace

```
SourceText ReplaceText RegularExpression regexpr.replace
```

Replaces all matches in SourceText by ReplaceText

Syntax

Pico uses TRegExpr written by Andrey V. Sorokin. The help file for TRegExpr is included and you can directly jump to it via the following link.

See [Regular Expression Syntax](#)

Formatting functions

Formatting functions

Pico supports the following string formatting functions.

fix

```
v1 decimalcount fix
```

Returns a floating point value v1 converted to a string with **decimalcount** decimals.

Conversion functions

Conversion functions

Pico supports the conversion functions listed below.

asdate

Converts a ISO yyyyymmdd formatted date string to a date value. Raises an error if the string is not a valid date. Use **isdate** to test if the string is a valid date.

```
v1 asdate
```

asnumber

Converts a number string to a floating point value. Raises an error if the string is not a valid floating point number. Use `isfloat` to test if the string is a valid floating point number.

```
v1 asnumber
```

asinteger

Converts a number string to an integer value. Raises an error if the string is not an integer number. Hexadecimal numbers are also taken into account.

```
v1 asinteger
```

ashex

Converts an integer number to its hexadecimal representation.

```
v1 ashex
```

astext

Converts a value to a string.

```
v1 astext
```

degrees

Converts radians to degrees.

radians

Converts degrees to radians.

Debug functions

Debug functions

Pico supports the following debug functions.

trace

Triggers the onTrace event.

```
value trace
```

In GoPico the `trace` function will add `value` as a new line in the trace window.

You can use `trace` to debug your scripts and to send multiple values from a script. You can use `say` to give a message to the user, but `say` will interrupt the program flow and that is sometimes not desired.

loglevel

Sets the logging level.

When a 'log' message is about to be generated it is checked against the current logging level. If it is lower or equal to the logging level then it will be logged, otherwise it will be discarded. Setting the logging level (which is by default -1, meaning 'off') also might create a new logging file if logging is made active. The file created can be set with 'logfile'. If the logging level is changed from a valid level to the 'off' level then the logging file is closed (and changing the logging level to an active level will re-create the file making it possible to clear the log programmatically).

```
1 loglevel
```

logfile

Defines the file for logging.

By default the file 'picolog.msg' will be generated.

Note that this setting is only taken into account when the 'loglevel' is being 'raised' from disabled (<0) to a normal logging level.

```
"picolog.msg" logfile
```

logappend

Defines that log information is appended to the logging file or not. By default this is set to 'false', meaning that a new file is created (all old data is lost).

```
logappend
```

logmessage

Add a message to the logging file (if enabled).

Messages are only logged when the 'loglevel' is lower or equal to the active logging level (as set with 'loglevel').

The following bits in the level do the following:

\$8x : adds the stack contents to the message (top of stack is the last displayed value)

\$4x : numbers as hexadecimal if possible (default is decimal)

\$2x : does **not** add new line with timestamp before the message

Typically used if log messages are separately generated but need to be on the same line..

```
$C1 "message" logmessage
```

Interface functions

Interface functions

Pico supports the following interface functions:

say

```
message say
```

Displays **message**

ask

```
defaultvalue prompt ask
```

Ask the user for input, using **prompt** as the question and default value as the initialvalue (which be may "").

saystack

```
saystack
```

Displays the stack contents if there is anything on the stack. If the stack is empty nothing is displayed.

Test functions

Test functions

Pico supports the following test functions.

isinteger

Tests if a string is a valid integer. Returns a boolean result.

isfloat

Tests if a string is a valid floating point number. Returns a boolean result.

isdate

Tests if a string is a valid ISO date. Returns a boolean result. The ISO date must be a string in the `yyyymmdd` format.

Reflection functions

Reflection functions

Pico support the following reflection functions.

type

Returns the `<typestring>` of the top of stack object.

`type`

`<typestring> :: 'number' | 'text' | 'reference'`

classnames

Returns the list of classname(s) if any of the top of stack Pico object.

Example:

```
class:animals() {  
}
```

```
object:horse(animals) {  
}
```

```
horse classnames
```

Data types

Data types

Some programming language enforce you to declare the data type of a variable. This variable then can hold only values of the specified data type. Other languages allow you to use variant values that can be stored in any variable.

In Pico a variable is just a named store for a value and once you have defined a variable you can store a value of any data type that Pico supports in that variable.

In Pico any value is an object. The number `123` is an object and the string `"Pico"` is an object. Value objects have the following properties:

- `<symbol>`
- `<value>`
- `<type>`

`<symbol>` :: the text presentation of the value

`<value>` :: the value itself, e.g. `123` or `"Pico"` or a reference to a object

`<type>` :: the data type indicator of the value

Delphi programmers may say: why not use variants? Variants have the limitation that I do not have complete freedom of how I handle them. In Pico I want to have full control of how values are stored and handled.

In the current version of Pico there are 3 data types:

Text

You can enter a text value in your scripting by enclosing the text value with either single or double quotes.

Number

You enter a number value in decimal format in the usual way. In Pico you must use the dot as decimal separator (not the comma). You can enter integers in hexadecimal format by prefixing the number with the dollar sign, e.g. \$7F.

Reference

Reference values reference an object in the script. You enter a reference value by entering the name of an object.

Data type checking

In Pico all operators check if their operands are of the correct data type. An error is raised if a value is of an incorrect data type.