

# **Venom2 Help File**

---

*by Venom Control Systems Ltd*

Copyright © 2009-2021 Venom Control Systems Ltd

# Table of Contents

Foreword	0
<b>Venom2 Help</b>	<b>2</b>
<b>How to use this help file</b>	<b>4</b>
<b>Language Overview</b>	<b>6</b>
Description .....	6
Language Structure .....	9
New Features .....	13
<b>Language Keywords</b>	<b>16</b>
+ .....	17
- .....	17
* .....	18
/ .....	18
^ .....	19
" .....	19
= .....	21
<> .....	22
< .....	22
> .....	23
<= .....	23
>= .....	23
<< >> (Bit shift) .....	24
<<<: >>> (Embedded text) .....	24
\$ .....	26
% .....	26
' .....	27
~ .....	28
: Colon .....	28
, .....	32
? .....	32
! .....	33
@ .....	36
:= .....	38
() .....	38
[] .....	39

. [Dot] .....	40
; [Comment] .....	41
Abs .....	41
Acos .....	42
All .....	42
And .....	42
AndAlso .....	43
Any .....	43
Array .....	44
As .....	46
Asin .....	47
Assignment .....	47
Atan .....	47
AutoDestruct .....	47
Await .....	49
Base .....	50
Beep .....	51
Break .....	51
BS .....	51
Call .....	52
Case .....	54
Catch .....	54
Centre .....	54
Char .....	54
Chr .....	55
Class .....	55
Cls .....	62
Cos .....	62
CR .....	62
Delete .....	63
Derived .....	63
Div .....	64
Do .....	64
Else .....	64
End .....	65
Eor .....	65
Every .....	66
Exit .....	66
Exp .....	67

False .....	68
Float .....	68
Font .....	68
Forever .....	68
Global .....	68
GotoXY .....	69
Has .....	70
Help .....	70
Home .....	70
If .....	71
Index and Index0 .....	72
Int .....	73
Inv .....	73
Is .....	73
IsFalse .....	74
Left .....	75
List .....	75
Local .....	75
Log .....	76
Make .....	77
Mod .....	77
New .....	78
Nil .....	79
Nop .....	80
Or .....	80
OrElse .....	81
ParamCount .....	81
Parameter .....	82
Print .....	83
Program .....	85
Private .....	86
Protected .....	87
Public .....	87
Repeat .....	87
Return .....	88
Right .....	88
Select .....	89
Sin .....	90
Sqrt .....	90

Start .....	90
Stop .....	91
Swap .....	93
Tan .....	93
Task .....	94
TextBlock .....	95
Then .....	97
This .....	97
To .....	98
True .....	99
Try .....	99
TypeOf .....	101
Unsigned .....	103
Wait .....	103
Word .....	104
While .....	104
Printf .....	105
<b>Object Types .....</b>	<b>107</b>
Locking .....	108
Printf .....	110
Common object properties .....	114
AlphaLCD .....	115
Analogue .....	119
Array .....	128
Buffer .....	134
CANBus .....	151
Class-default .....	162
CRCGenerator .....	166
DateTime .....	168
Digital .....	177
Encrypter .....	185
Ethernet .....	191
FileSystem .....	204
File .....	236
FTPClient .....	248
FTPServer .....	254
GraphicsLCD .....	261
HashGenerator .....	291
HTTPServer .....	293

I2CBus .....	320
IProt .....	326
Keypad .....	330
NIL .....	337
NumberReader .....	338
OnBoardLED .....	345
OneWire .....	348
OperatingSystem .....	355
PIDController .....	379
POP3Mailbox .....	384
PrintJob .....	387
PulseCounter .....	389
PulseWidthIn .....	391
PulseWidthOut .....	396
RandomNumberGen .....	402
RealTimeClock .....	404
SafeData .....	412
Semaphore .....	421
SerialPort .....	424
Shaft .....	434
SMSLink .....	437
SMTPSender .....	450
SPI .....	452
String object .....	456
Stopwatch .....	465
Task .....	467
TextAnalyser .....	469
UDProt .....	477
PPProt .....	484
TCProt .....	493
Timer .....	504
TouchScreen .....	507
TouchScreen: Button .....	522
WiFiLink .....	527
XMODEMLink .....	541
 <b>Pre-processor commands</b>	 <b>547</b>
#Define, etc .....	547
#If, etc .....	549

<b>TCP/IP Networking</b>	<b>553</b>
Notes on TCP/IP .....	554
IP Addresses .....	554
Example Code .....	555
Glossary .....	559
<b>Appendix</b>	<b>562</b>
A: Startup Sequence .....	562
B: Robust applications .....	563
C: Calling foreign code .....	564
D: Development Checklist .....	564
E: Error messages .....	565
F: FAQ .....	567
G: Glossary .....	568
H: Number Limits .....	572
I: Operator Precedence .....	572
J: Speed of Execution .....	573
K: Optimisation .....	574
L: ASCII Character Set .....	576
M: Memory Map (VM2) .....	578
N: Protecting your application .....	579
O: Updating Venom2 .....	579
S: Serial settings .....	579
<b>Credits</b>	<b>582</b>
<b>Index</b>	<b>583</b>

# **Venom2 Help**



---

## Venom2 Help

This is the Help File for the Venom2 language.

- [How to use this file](#)

### Version

V 2021 07 01

*Copyright © 2009-2021 Venom Control Systems Ltd*

*All rights reserved*

# How to use this help file

---

## How to use this help file

### About this help file

This help file contains a [description](#) of the Venom2 Language (for the VM2) and detailed information on every [keyword](#) and [object](#) in it. There is also an [overview](#) of the language, and various tables, charts, diagrams and [appendices](#).

### Finding what you want

There are several ways to find the information you want in this file:

- From this file, use the Contents, Index or Search tabs at the top of the panel to the left.
- From VenomIDE (our integrated development environment): put the cursor in a Venom keyword, right click it for a pop-up menu, and choose **Help on:** \_\_\_\_, or just hit **F1**.

### The most up to date information

NOTE Although this help file reflects the most current information possible, you should read the *Venom2 Release Note* for information that may not have been available prior to the publication of this version of the Help file. In particular, as Venom2 is being continually improved, the Release Note will document the added features and improved functioning available in any particular version of Venom2 over that described in this help file.

The *Release Notes* can be found online at [Venom Control Systems](#)

**WARNING:** Users of control equipment should be aware of the possibility of a system failure, and must consider the implications of such failure. Venom Control Systems Ltd. can accept no responsibility for loss, injury, or damage resulting from the failure of equipment we supply. Use of our products in applications where their failure to perform as specified could result in injury or death is expressly forbidden.

# Language Overview

## Language Overview

Here is an overview of the Venom2 Language

- [Description and heritage of Venom2](#)
- [The structure of Venom2](#)
- [New features in Venom2](#)

## Description

Venom2 is a language for writing control applications. Typical uses are

- Industrial automation
- Process control
- Intelligent instruments
- Security systems
- Hand-held devices

## Heritage

Venom2 is based on the original Venom and Venom-SC languages (used in the Scorpion K4 and VM1 control computers).

Venom was an object oriented version of 'the Scorpion language' that ran on the Micro-Robotics Scorpion Control Computer, which was itself loosely based on LOGO, a language used to teach computing to schoolchildren.

Venom2 is similar to Venom-SC, but has notable improvements, such as user defined Classes.

Both Venom-SC and Venom2 are 'Semi-Compiled'. This means that the code is partly compiled, but not all the way down to native machine code. The earlier languages developed at Micro-Robotics were interpreted; semi-compilation yields a considerable speed increase while maintaining the small code size and flexibility of an interpreter.

## Block Structured

Statements in Venom may be grouped together into a block equivalent to a single statement using `[` and `]`. Language constructs are based around the 'recursive' nature of statements.

## No statement separators

Venom has no statement separators equivalent to C's semicolon '`;`'. Instead the Venom parser relies on the syntactic structure of each keyword and operator to recognise where a statement ends. Venom statements will span any 'white space', including carriage returns. This allows indentation to reflect the block structure of code.

## Dynamically Typed

Variables in Venom are stored as a value and a type. The type of a variable is only set when the

variable has been assigned a value. Potentially any variable may hold a value of any type. This means the compiler cannot do any type checking on variables. All type checking is done at runtime rather than compile-time. This makes for a flexible and forgiving language for fast application development, but isn't so formally precise as strongly typed languages. Runtime type checking also has a small speed penalty.

All variables take eight bytes of storage. Four bytes are used to hold the value element of the variable, and one is used to hold the type. Another byte is used to indicate the write-protection status of the variable. The remaining two bytes are reserved for future use.

When a global variable is first created, it is given the special type 'Unassigned'. This allows the runtime system to error if there is an attempt to use a variable before it has been given a value. Local variables are initialised to the value integer zero, unless explicitly initialised at declaration.

## Name Scope

Venom has four main user-defined-name scopes: Global variables, local variables, Class members and macros. Global variable names are used for procedures and also integer, floating point and other types of variable. Global names are 'visible' from any procedure and any task in the multitasking system - unless local variable or class member names eclipse them.

[Local](#) variables are only visible from within the procedure where they are defined. If a local variable has the same name as a global, then the local takes precedence. Local variables are created when the procedure is called, and destroyed when the procedure is done. They are held on the stack.

Parameters passed to procedures are essentially local variables that have been initialised to the parameter values, and are equivalent to locals in their operation.

Class members are visible from inside and outside a class.

## Macros

[Macros](#) are bits of program text that have been given a name so that they may be instantiated many times, or simply to improve readability.

## Object Orientation

There are many pre-defined object types in Venom, and a set of pre-defined messages names, that are accepted by these objects. Because of the weakly typed nature of the language, any variable may potentially accept any message with any parameters, returning any, or no, result (which gives a kind of *polymorphism*). As with other type checking, this is all sorted out at runtime.

It is also possible to create new classes of object in Venom, using the keyword **Class**. A class may inherit from another class (*single inheritance*), though not directly from any of the pre-defined Venom classes.

A Venom value that 'holds' an object is actually more equivalent to a pointer to that object. If you copy the value then you haven't got two objects, you have two 'handles' on the same object. A handle is a value that uniquely identifies a particular object to the code that deals with that type of object. It may, or may not, be a memory pointer.

## Garbage Collection

Garbage collection is the term given to the freeing-up of memory, and other resources, that are no longer required by a program. In some high level languages garbage collection is done automatically, but it is not at all easy to combine automatic garbage collection with 'hard realtime' applications, which is what Venom2 is aimed at.

For this reason there is no automatic garbage collection system in Venom, so you will need to keep track of the objects you create yourself. This is most easily handled by creating all the objects your program will ever need at the start of your program. This is the best way in most Venom applications, but there are some applications where you can't predict how many objects of a given type you'll need at the start. In these cases you'll need to create and remove objects dynamically.

Objects created dynamically (i.e. created and destroyed during program operation, rather than just created at the start) are often held in local variables or in the members of a 'parent' object. In this case the [AutoDestruct](#) mechanism may be used to remove these objects when the procedure ends or the parent object is removed.

There is a built-in 'memory-leak detector' (or [Garbage Scanner](#)) that may be used to qualify Venom programs.

## Data Structures

All data structures are handled by objects. The data structures available are:

[Array](#) – a fixed-size array of initialised data. Arrays can take many types of data.

[Buffer](#) – a variable-sized list of variable data that can hold integers, floats, text. A special type of buffer (called 'Buffer of Any') may also contain mixed data of any type, including other objects. Buffers can perform sophisticated operations on the data they contain.

[String](#) – an array of text with a defined maximum size.

[TextBlock](#) - used to embed large amounts of text in a source file.

[Class](#) - user-defined classes may be used to implement complex, hierarchical data structures.

## String Handling

To handle textual data, Venom has [string constants](#), [String objects](#), [Arrays](#) of strings, and [text Buffers](#). It also has the [Print](#) command, which in Venom is used to do much more than print to an output device - it is used for sophisticated text manipulation. Venom also has a [PrintF](#) message, with similar syntax to C's printf() function.

## Multitasking

The Venom2 language and its forebears have multitasking built in. To the application writer, the task manager is a pre-emptive round-robin system. That is, all tasks have equal priority and run one after the other, being swapped out by the task manager with little external control.

Underneath, tasks run co-operatively – choosing when to swap out. This makes the Venom runtime system easy to manage and more efficient. Each part of the venom runtime system is required to obey rules that set the maximum period a task may run for. This then defines the

system latency as a the maximum time-slice period (1-2mS in Venom2) multiplied by the number of tasks.

Task scheduling is indicated to the task manager by using language constructs that wait.

Examples are [Wait](#), [Every](#) and any message to an object that needs to wait: e.g. **keypad . Get**.

## Printing

Printing is used to create a textual representation of a numeric value or an object, and send this text to an output device.

However, in Venom, printing is also used as the basis for most text or string manipulation.

There are two different ways to print within Venom2: the classic Venom [Print](#) statement, and sending the [PrintF](#) message to a text handling object.

### Print

Printing an object, or printing *to* an object are used to handle a many otherwise lengthy or tedious operations. For example you can show the date and time in an instantly recognisable format by [printing the RealTimeClock](#) object. Conversely you may also set the time in the clock by [printing a date and time to it](#).

There are a set of special printing keywords that represent operations like carriage return ([CR](#)), clear screen ([CLS](#)) and so on. Additional formatting of print output is handled by passing special parameters to the print command using the [colon](#) print format operator.

### PrintF message

The [PrintF](#) message is very much like C's printf(), fprintf(), etc functions. You provide a format string and a list of parameters, and the resultant formatted text is sent to the object.

Internally, Print and PrintF output is packaged up into 'print jobs'. These are discrete amounts of text with some extra coding to take care of the special print keywords that Venom uses.

Using print jobs reduces the overhead of sending print output to an output device character by character.

## Language Structure

The following is a list of all the elements of the language, building from the bottom up. Only the major structural layers are discussed here; the details of the structure of each keyword and symbol are discussed in the main body of the book.

### Character set

Venom2 uses the following characters, which have been arbitrarily divided into the following classes for discussion:

**Alphabetic:** A-Z a-z \_

**Numeric:** 0-9

**Operator Symbols:** + - \* / \ < = > @ ? ! ^

**Number base:** \$ % ~



```
Quote:  " '
Separator:  , . : ;
Bracket: () []
White space:<SPACE> <TAB> <CARRIAGE-Return>
Control: <CTRL-C>, <CTRL-T>
```

Any other character will cause a syntax error, unless it is part of a character constant or string constant.

## Constants

Constants may be integer (sub-divided into decimal, hex and binary), floating point, character or string:

```
12  $AF  %1101101
12.34  1.2E-5
'A'
"A string"
```

Integer constants start with a numeric character, or a \$ for hexadecimal numbers, or a % for binary numbers. Decimal constants may only use numeric characters. Hexadecimal constants consist of the characters 0-9 and A-F or a-f. Binary constants may only use 1's and 0's.

Floating-point constants start with a numeric character and have a decimal point somewhere within them. They may also optionally have an exponential part, indicated by an 'E' or 'e' followed a positive or negative integer exponent.

Character constants are a single character surrounded by single quotes: 'A'.

String constants are a group of characters surrounded by double quotes: "abcde".

The Venom2 compiler is capable of a high degree of constant folding, so many expressions involving constants are also themselves treated as constants by the compiler: 10 + 100 is treated as 110.

## Variable names

Variable names may be up to 64 characters long. They consist of alphabetic, numeric and underscore '\_' characters. Names may not begin with a number. Variable names may not be the same as any Venom keyword, class name, or message name.

## Variables

Variables in Venom are always eight bytes. Any variable may hold any of the various data types, including handles to any type of object. The value portion of the variable is four bytes wide (32 bits).

There are two main 'scopes' of variable: *global* and *local*. Global variables are created simply by using their name either on the command line or in a procedure. Any variable that is not explicitly defined as a [LOCAL](#) variable is global.

## Loop indexes

There are two 'variables' defined by the language: [Index](#) and [Index0](#). These are loop indexes: they hold the loop count for the looping construct currently running. Index starts counting at 1,

Index0 starts at 0. You cannot assign values to them. A distinct value for Index0 is held for each level of loop nesting.

## Expressions

Expressions consist of one or two sub expressions linked by an operator. The simplest sub-expressions are constants, variables and procedure calls. Expressions of arbitrary complexity may be built up from this 'recursive' structure. The order of evaluation of operations is defined by the [operator precedence](#). [Parentheses](#) ( ) may be used to override the precedence rules.

There are three main kinds of operator binding in Venom: binary, prefix, and postfix.

Binary operators take two operands, one before the operator and one after it. Prefix operators appear before the operand, and postfix operators appear after it.

## Pointer Expressions

This shouldn't really be a separate level of language structure ...

Venom supports many kinds of pointer expressions. In particular you may take a pointer to any variable (including procedures, objects, local variables) using the [@](#) ('at') symbol. You may then 'de-reference' a pointer using the [!](#) (pronounced 'pling') symbol. Thus you may do many of the sophisticated functions available in other modern languages like calling procedures using procedure pointers.

## Statements

A venom statement is relatively complicated to define. Venom statements are the most complicated of the structural layers of the language, closely followed by expressions.

A statement is either

- One of the system operations: Make, PRINT, DELETE, HELP, ... , etc.
- One of the Venom flow control constructions: If While Repeat Every ... etc.
- An assignment using the [:=](#) (becomes equal to) operator
- An expression (normally this is a procedure call or a message to an object)
- Square brackets, [ ], surrounding a set of sub-statements.

Detailing each of these...

Venom system operations and flow control constructions are all different, but many of them have one or more expressions or sub-statements as part of their structure. These are all documented later in this book. For example the While statement needs an expression and a sub-statement to complete it:

```
While <expression> <statement>
```

Venom assignment syntax is

```
<Left-hand side expression> := <expression>
```

A left-hand side expression must evaluate to either a variable, the address of a variable, a memory location, or a message to an object.

[Expressions](#) are covered elsewhere.

[Square brackets](#) [ ] are for grouping statements together so they are equivalent to a single statement:

```
[<statement><statement><statement> ... ]
```

As with expressions, statements of arbitrary complexity may be built up with this recursive structure.

## Comments

The Venom [comment](#) separator is semicolon ‘;’. The compiler ignores any further text until the next carriage return. There is currently no in-line comment structure like C’s /\* \*/. Comments may be placed anywhere within the code.

## Procedures

Simply speaking, a procedure is a named list of statements. A procedure is delimited by the keywords [To](#) and [End](#). A procedure may also have a list of parameter names, and a set of local variables. Internally, local variables and parameters are the same thing. The syntax for a procedure is

```
To <proc name> (<parameter name> , ...)
LOCAL <local variable name> := <expression> , ...
<statement>
<statement>
...
<statement>
End
```

Parameters may be declared as *optional* using [square brackets](#). Optional parameters are initialised to integer zero if they are not supplied, and the value ParamCount gives the number of actual parameters supplied.

[LOCAL](#) variable declarations may be separated by the keyword LOCAL or by commas. You may initialise local variables with any expression. The LOCAL declarations must all come before any other statements.

All procedures return a result: if you don’t explicitly set the return value using the [Return](#) keyword, then a default value is returned. Currently this is integer zero.

To call a procedure, use its name alone, or with parentheses ( ), after it. If the procedure takes parameters then you need to put expressions for these in parentheses after the name. For example:

```
my_proc
my_proc()
my_proc_with_params(1,2)
```

## Classes

A Class is a type of *object*. An object is a collection of data items, where each item of data has a name and a type, called *members*, together with a collection of procedures to operate on that data, called *methods*. A class by itself can't do much. To do something with it you have to create a new instance of the class - an object.

There are two basic kinds of class in Venom2: pre-defined classes, built into the Venom2 language, and user-defined classes.

Much of the power of the Venom language resides in the pre-defined classes, with yet more power available in the user-defined classes.

## Tasks

A task is a section of the overall application program that may run concurrently with other sections. In the simplest case there may only be one task running. However it is possible to create other tasks. These tasks will all run at the same time and are independent: in general, several tasks may carry out different operations at the same time.

The venom syntax for creating a new task is:

```
<task object> := Start <statement>
```

The sub-statement to [Start](#) will be run as an independent task.

Although the statement may be any Venom code it is good practice to keep it short: often it's just a single procedure call.

A task will run continuously until it runs out of code to execute, or if it is stopped using [Stop](#) or Stop ALL. If the task code is an infinite loop then the task will never end.

## Venom Stack

There is a procedure-call stack (called the Venom Stack) that stores the return address of the code that the procedure was called from and the *frame pointer* - that is a pointer to the current *stack frame*. A stack frame is an area of the stack that holds the values of all the parameters and local variables that are associated with a procedure. A new stack frame is added to the stack when a procedure is called and removed when the procedure is done.

The return address and frame pointer take 8 bytes of stack memory each, as do each of the parameters and local variables. The size of the Venom Stack is large enough for just about any sensible program that doesn't use recursion.

## New Features

Here are a few features that are new to Venom2 - i.e. they were not present in Venom-SC.

## Language structure

[User-defined classes](#)

[Optional parameters](#)

---

[Try/Catch](#) exception handling

## Operators

[AndAlso](#), [OrElse](#), [IsFalse](#)

[Bit-Shift](#)

[Inv](#)

## Objects and messages

[Semaphore](#) object

[PrintF](#) message

## Pre-processor

[Macro parameters](#)

[#Redefine](#), [# Undefine](#)

[#If conditional compilation](#)

# Language Keywords

## Language Keywords

This is the complete list of keywords in the Venom2 language in alphabetic order. Symbols are grouped according to function rather than ASCII order.







You will find all the keywords listed in the Contents panel to the left.

### How to use the Keywords section

The keywords are in alphabetic order, with the non-alphabetic symbols at the front, roughly grouped together by function rather than in ASCII order. Each keyword has a syntax description followed by information on what the keyword does, with code examples

```
Print net ; This is a code example
```

There may be supplementary information of various kinds, indicated by the following icons:

-  This gives information on any limits to the parameters or results of the keyword
-  This gives information on the hardware needed to use this keyword.
-  This gives information on the memory usage of the keyword
-  This gives information on timing aspects of the keyword
-  This gives pointers to other parts of the manual for related topics
-  This warns you to beware of putting bugs in your code

### Syntax descriptions

Under each keyword there is a semi-formal description of the way it is used in a program; both the syntax and the types of parameters and results are given.

The syntax descriptions are not intended to be fully precise, but instead to give a simple guide as to how the keywords and symbols should be used. The complexity of a fully precise description of each keyword might obscure rather than illuminate its operation.

Here is a guide to interpreting the syntax descriptions:

- Text in square brackets [ ] indicates an optional part of the construction.  
Where there might be confusion between the brackets in a syntax description and the brackets that are part of the language, the syntax description brackets will be in regular text.
- Text in <angle brackets> indicates a reference to a 'lower level' construct:
- <name> means any Venom name, and may be qualified by Global or Local
- <statement> means any Venom statement
- <expression> means any Venom expression, and may be qualified by a data type.
- The following in regular text indicate the data type of a value: Int, Float, String, Object, Pointer, Any. Any means that the data type might be any of the other types.

- The symbol  $\Rightarrow$  means the construction returns a result. The type is given in 'light' (not bold) text: e.g. `Int`
- Ellipses ... indicate optional repetition of the preceding construct.

As an example

```
Make d digital(Int channel, [Int attributes])
```

**+**

`< Int expression > + < Int expression >  $\Rightarrow$  Int`

`< Int expression > + < Float expression >  $\Rightarrow$  Float`

`< Float expression > + < Int expression >  $\Rightarrow$  Float`

`< Float expression > + < Float expression >  $\Rightarrow$  Float`

The + operator adds two numbers. If either operand is a floating-point number, the result will be floating-point.

See [operator precedence](#) for which operators are evaluated first.

▼ The limits to the operands and result of + are those that apply to all internal calculations. See [Number Limits](#)

**-**

`< Int expression > - < Int expression >  $\Rightarrow$  Int`

`< Int expression > - < Float expression >  $\Rightarrow$  Float`

`< Float expression > - < Int expression >  $\Rightarrow$  Float`

`< Float expression > - < Float expression >  $\Rightarrow$  Float`

`- < Int expression >  $\Rightarrow$  Int`

`- < Float expression >  $\Rightarrow$  Float`

When used as a binary operator (between two values), the - operator subtracts the right-hand operand from the left-hand. If either operand is a floating-point number, the result will be floating-point.

When used as a prefix (before a value), the value is made negative and retains its type.



See [operator precedence](#) for which operators are evaluated first.



The limits to the operands and result of - are those that apply to all internal calculations.

See [Number Limits](#)



If you use don't use comma's to separate parameters, then be careful of the unary minus operator – it can be confused with binary minus:

```
three_param_proc(1 -2 3) ; we've only sent two paramet
```

\*

```
< Int expression> * < Int expression> ⇒ Int
```

```
< Int expression> * < Float expression> ⇒ Float
```

```
< Float expression> * < Int expression> ⇒ Float
```

```
< Float expression> * < Float expression> ⇒ Float
```

The \* operator multiplies two numbers. If either operand is a floating-point number, the result will be floating-point.

See [operator precedence](#) for which operators are evaluated first.



The limits to the operands and result of \* are those that apply to all internal calculations. See [Number Limits](#)

/

```
< Int expression> / < Int expression> ⇒ Float
```

```
< Int expression> / < Float expression> ⇒ Float
```

```
< Float expression> / < Int expression> ⇒ Float
```

```
< Float expression> / < Float expression> ⇒ Float
```

The / operator divides the left-hand operand by the right-hand. The result is always floating-point, even if both operands are integers. Use [Div](#) if you want to do integer division.

See [operator precedence](#) for which operators are evaluated first.



The limits to the operands and result of / are those that apply to all internal calculations. See [Number Limits](#)

^

`<Int> ^ <Positive Int> ⇒ Int`

Otherwise...

`<Int or Float> ^ <Int or Float> ⇒ Float`

The ^ operator raises the first operand to the power of the second. If both operands are Int and the second is positive then the result will be an integer.

**Note:** ^ has the same precedence as \*, etc. See [operator precedence](#) for which operators are evaluated first.

▼ The limits to the operands and result of ^ are those that apply to all internal calculations. See [Number Limits](#)

"

`"<characters>" ⇒ String constant`

*<characters> is a string made up from any printable character apart from " and \. These and other special characters may be introduced with an [escape sequence](#).*

The double quote symbol is used to enter text strings into the program. It must always be balanced with another quote at the end of the string. All quoted strings are string constants, that is you may not change the characters within the string after it has been defined.

You may use string constants in fairly sophisticated ways. For example you may enable your application for different spoken languages. By running one of the procedures below at any time you can change your user interface language:

```
To english
  Yes_string := "Yes"
  No_string  := "No"
End
```

```
To french
  Yes_string := "Oui"
  No_string  := "Non"
End
```

...

```
If answer_correct
```

```
Print yes_string
...
```

If you need more sophisticated text handling (manipulation of characters within a string etc), use a [String](#) object or [Buffer](#) object.

### Similar to string objects

Note that string constants and [String](#) objects (variable strings) are very similar: string constants will accept many of the messages that string objects will. For example string constants can be asked for their length with **.Length**:

```
-->Print Yes_string.Length
3
```

### Concatenating quoted strings


Strings may be extend over more than one line: if two quoted strings are only separated by whitespace or comments then they will be run together.

These lines of code result in the string below them being defined:

```
"This string " ; This is a comment
"Runs into this one"
```

Resultant string:

```
"This string Runs into this one"
```

 To embed large strings of completely literal text within a Venom source file use [Embedded Text](#) or [TextBlock](#).

### Escape sequences

In Venom2 an escape sequence is introduced with the **\** character ('backslash'). The complete list of escape codes is

Sequence	Yields the character	ASCII	
<b>\\</b>	Backslash: \	92	\$5C
<b>\"</b>	Double quotation mark: "	34	\$22
<b>\\$hh</b>	Any ASCII character from hexadecimal digits	-	hh
<b>\a</b>	Causes a terminal to emit an audible alert	7	\$07
<b>\b</b>	Backspace	8	\$08
<b>\f</b>	Form feed	12	\$0C

<code>\n</code>	New line	10	\$0A
<code>\r</code>	Carriage Return (use new line, <code>\n</code> for most purposes)	13	\$0D
<code>\t</code>	Horizontal tab	9	\$09
<code>\v</code>	Vertical tab	11	\$0B

The sequence of `\` followed by any other character is ignored completely (no characters are entered into the string constant) – though additional sequences may be added in future.

## Graphics LCD

The following Escape sequences are used only when printing to a [GraphicsLCD](#), particularly when using [PrintF](#).

<code>\^hh</code>	Bitmap insertion; <i>hh</i> is the bitmap number as a hex pair
<code>\L</code>	Set Left justification
<code>\C</code>	Set Centre justification
<code>\R</code>	Set Right justification
<code>\Fhh</code>	Set Font - <i>hh</i> is the font number specified as hex pair

 [Print formatting for strings.](#)

=

**<Any expression> = <Any expression> ⇒ Int**

The 'Equals' operator takes two numbers and performs a comparison.

It evaluates to [True](#) (1) if its two operands are the same type and value, [False](#) (0) otherwise.

**N.B.** Unlike other operators, = does no type coercion, so a Float is *never* equal to an Int

For example,

```
-->Print 1 = 1.0
0
```

This limitation is used so as to preempt problems caused by Floats being imprecise in some circumstances. If you need to compare an Int and a Float, use the type conversion operators [As Int](#) or [As Float](#):

```
-->Print 1 As Float = 1.0
1
```

### Comparing floats

While it is possible, it is not usually a good idea, to compare two Float values using `=`. Two Float values may appear the same in some circumstances, but may be different in others. For example

```
-->Print 1/7 , CR
0.142857
-->Print (1/7 = 0.142857) , CR
0
```

i.e. the two values for `1/7` were not seen to be the same.

Instead use operators like `>` and `<`, etc. to compare Floats.

### Comparing strings

When `=` is used to compare strings, the comparison is made between the objects' 'handles' (pointers), not the text. Thus `=` can not usefully be used to compare two strings - instead use [String.Compare](#).

See [operator precedence](#) for which operators are evaluated first.

▼ The two operands can be of any type.

<>

`<Any expression> <> <Any expression> ⇒ Int`

The 'Not equal' operator takes two numbers and performs a comparison.

It evaluates to [True](#) (1) if its two operands are of different types or values, [False](#) (0) otherwise.

**In general, it is unwise to use `=` and `<>` with floating point numbers. See [=](#) for more information about this.**

See [operator precedence](#) for which operators are evaluated first.

▼ The two operands can be of any type.

🔍 [Equals](#) operator

<

`<Any expression> < <Any expression> ⇒ Int`

The 'Less than' operator evaluates to [True](#) (1) if the first operand is less than the second, [False](#)

(0) otherwise.

< can be used with mixed integer and floating-point operands.

See [operator precedence](#) for which operators are evaluated first.

▼ The limits to the operands of < are those that apply to all internal calculations. See [Number Limits](#)

>

**<Any expression> > <Any expression> ⇒ Int**

The ‘Greater than’ operator evaluates to [True](#) (1) if the first operand is greater than the second, [False](#) (0) otherwise.

> can be used with mixed integer and floating-point operands.

See [operator precedence](#) for which operators are evaluated first.

▼ The limits to the operands of > are those that apply to all internal calculations. See [Number Limits](#)

<=

**<Any expression> <= <Any expression> ⇒ Int**

The ‘Less than or equal’ operator evaluates to [True](#) (1) if the first operand is less than or equal to than the second, [False](#) (0) otherwise.

<= can be used with mixed integer and floating-point operands.

See [operator precedence](#) for which operators are evaluated first.

▼ The limits to the operands of <= are those that apply to all internal calculations. See [Number Limits](#)

>=

**<Any expression> >= <Any expression> ⇒ Int**

The ‘Greater than or equal’ operator evaluates to [True](#) (1) if the first operand is greater than or equal to than the second, [False](#) (0) otherwise.

>= can be used with mixed integer and floating-point operands.

See [operator precedence](#) for which operators are evaluated first.

▼ The limits to the operands of >= are those that apply to all internal calculations. See [Number](#)

## Limits

### << >> (Bit shift)

`<Int> << <Int> ⇒ Int`

`<Int> >> <Int> ⇒ Int`

The `<<` and `>>` operators shift binary bits left or right in the first operand, by a number of bits given by the second operand. Both operands must be integers.

The shift is always logical - i.e. a right shift operation will shift in 0 bits on the left. See below for an *Arithmetic Right Shift*.

If the right hand operand is negative the results are not defined.

`<<` and `>>` have the same precedence as `*`, etc. See [operator precedence](#) for which operators are evaluated first.

▼ The limits to the operands and result of `<<` and `>>` are those that apply to all internal calculations. See [Number Limits](#)

### Arithmetic right shift

There isn't an arithmetic right shift operator in Venom, but you can achieve the same effect using the `Div` operator.

So the arithmetic equivalent of

`a >> b`

would be

`a Div (1 << b)`

Because the Venom compiler folds constants, these two will compile down to the same number of bytecodes, and will execute with very similar speed.

### <<<: >>> (Embedded text)

`<<<:Embedded text>>> ⇒ String constant`

The embedded text syntax allow you to embed literal blocks of text in your code. This is very useful when writing web server applications which often require you to embed bits of HTML code in your venom procedures and methods.

The embedded text is literal: every character in the file is included exactly as it appears in the editor, apart from line endings, which are always converted to `'\n'` by the Venom compiler.

The embedded text doesn't have to occupy whole lines in the Venom code - it can start and end anywhere on a line.

There is a provision to include a description of the type of text that is being embedded - any printable characters between the opening <<< and the : are not part of the embedded text but may be used to signal to a syntax highlighter what kind of text follows, as it may have the ability to apply appropriate highlighting.

### Examples

```
Print <<<:<H1>This is a heading</H1>>>>
Printf(<<<:<H1>This is a %s</H1>>>>, "heading")
```

Both of these lines result in the following characters being printed:

```
<H1>This is a heading</H1>
```

Note that even though the closing </H1> tag ends with >, this is *not* seen as part of the closing symbol: >>>.

Here we have some embedded text that comprises several lines:

```
Print <<<:
<H1>This is a heading</H1>
And this is some text.
>>>
```

### Exceptions

In order to allow multi-line embedded text to look neater, if the colon is the last character on a line then the 'new line' character is not included in the text. The following example starts with the < character:

```
Print <<<:
<H1>This is a heading</H1>
And this is some text.
>>>
```

If you want your text to start with a blank line then you will need to do this:

```
Print <<<:

<H1>This is a heading</H1>
And this is some text.
>>>
```

### Limitations

Embedded text is stored inside the procedure or method it appears in. The size of a Venom procedure or method can't exceed 32KB, so this means that the total of all the embedded text in a single procedure, and all the other Venom code, should not exceed 32KB.



### VenomIDE Style tidier

The style tidier doesn't recognise the <<<: >>> construct and will try to tidy up the text inside the block.

### Future enhancements

We may add a provision to change the closing sequence from the default >>> to other user-defined strings.

 [TextBlock](#), [String concatenation](#).

\$

**\$<hex characters>** ⇒ Int

*<hex characters> is a number made up from the characters 0-9, a-f, A-F*


The dollar symbol is used to indicate hexadecimal (base-16) numbers. Hex numbers are often useful when accessing memory or hardware registers directly.

In base 16, the letters A-F represent the numbers 10-15. You can use upper or lower case. For example:

```
-->Print $A1f1
41457
```

Numbers can be printed in hexadecimal using ~.

▼ The maximum hexadecimal number that can be used is \$FFFFFFFF.

 See also [%](#), [~](#)

%

**%<zeros or ones>** ⇒ Int

*<zeros or ones> is a number made up from the characters 0 or 1 only*

The percent symbol is used to indicate binary (base-2) numbers. Binary numbers are often useful for accessing memory or hardware registers directly, and also for entering bitmaps. For example:

```
-->Print %101010
42
```

▼ The maximum positive number that can be entered in binary is %  
11111111111111111111111111111111 (31 1's).

The minimum, most negative, number that can be entered in binary is  
%10000000000000000000000000000000 (a 1 and 31 0's)

🔍 See also [§](#), [~](#)

'

'<character>' ⇒ Int

<character> is any printable character, or one of the escape code sequences.

The single quote is used to enter ASCII character constants into a program. It is just another way of expressing an integer.

For example

```
-->Print 'A' ;ASCII code for A
65
```

```
-->Print '\'' ;escape sequence for ASCII single quote
39
```

The full list of escape codes for other characters is

Sequence	Yields the ASCII code for	ASCII Value	
<code>\\</code>	Backslash: \	92	\$5C
<code>\'</code>	Single quotation mark: '	39	\$27
<code>\a</code>	Causes a terminal to emit an audible alert	7	\$07
<code>\b</code>	Backspace	8	\$08
<code>\f</code>	Form feed	12	\$0C
<code>\n</code>	New line	10	\$0A
<code>\r</code>	Carriage Return	13	\$0D
<code>\t</code>	Horizontal tab	9	\$09
<code>\v</code>	Vertical tab	11	\$0B

~

```
Print ... ~ <Int expression> ...
```

```
Print ... ~~ <Int expression> ...
```

The tilde modifier (pronounced tillda) is used to print integers in hexadecimal or binary.

## Hexadecimal


In hexadecimal the letters A-F will always be printed in upper case. For example:

```
-->Print ~41457
A1F1
```

Note that hexadecimal numbers are not printed with a \$ prefix. Hex numbers are always printed as 'unsigned' E.g.

```
-->Print ~-1
FFFFFFFF
```

The formatting of hexadecimal numbers is the same as decimal integers.

 See also [\\$, %](#), [:](#)

## Binary

Printing in binary is indicated using two tildes - ~~.


For example:

```
-->Print ~~42
101010
```

Note that binary numbers are not printed with a % prefix. Binary numbers are always printed as 'unsigned' E.g.

```
-->Print ~~-1
11111111111111111111111111111111
```

The formatting of binary numbers is the same as decimal integers. See [:](#) for more details.

 See also [\\$, %](#), [:](#)

## : Colon

```
Print <Any expression> : <Any expression> : ... , ...
```

The colon symbol is a *print format modifier* or *Class inheritance operator*.

We deal with print formatting here. See [here](#) for inheritance.

Zero or more colons are used after each item in a Print list to specify the formatting of the item being printed. Each colon is followed by a parameter. The way these formatting parameters are interpreted depends on the type of thing being printed:

[Integers](#)[IP dotted-quad style](#)[Floats](#)[Strings](#)[Objects](#)

## Other formatting options

There is a completely different way of printing available which involves sending the PrintF message to an output object (or the OperatingSystem object).

 [PrintF](#)

## Formatting Integers

Only one colon formatting parameter is used for integers, and this sets the minimum number of characters, or field width, used to print the number. This is used to make neat columns of right-justified numbers, for example:

```
-->Print 1:4, 123:4, CR, 99:4, 456:4
      1 123
      99 456
```

If the number takes more digits than the given field width, it will take as many characters as it needs. Hence to print a number with no minimum field width use :0 like this:

```
-->Print 1:0, CR, 1234:0, CR, -56:0
      1
      1234
      -56
```

If the colon is omitted the default field width of 6 places is used:

```
-->Print 1,CR,1234,CR,-56,CR
      1
      1234
      -56
```

If a negative field width is used, the number is printed in the same number of places as if the number was positive, but leading zeros are used instead of spaces to pad out the number. For example:

```
-->Print 1:-6,CR,1234:-6,CR,-56:-6,CR
000001
001234
-00056
```

## Formatting Floats

There are three ways of formatting floating point numbers. The type of format applied to the number depends on how many colons are used.

### : General

When just one colon value is given it simply specifies the total fieldwidth the number should be printed in. (If no colon is used then the default Float fieldwidth of 13 is applied).

Beyond that, the formatting is dependant on the value of the number, but will generally aim to represent the number in the most precise form possible.

```
-->Print 1.234,CR,10.324,CR,100000.0:11,CR,0.123,CR,1.123E8:16
      1.234
      10.324
      100000
      0.123
      1.123E+08-->
```

### : : Fixed Point

When two colons are used, the floating-point number is printed in 'fixed point' format. The first colon value gives the total fieldwidth and the second the precision (the number of digits after the decimal point). The fieldwidth will be exceeded if it is too small to represent the number correctly with the specified precision.

```
-->Print 3.1415:15:3, CR, 3.1415:9:3, CR, 3.1415:3:3
      3.141
      3.141
      3.141
```

If the fieldwidth is negative, then the number will be printed with leading zeros.

### : : : Exponential

When three colons are used the number is printed in exponential notation. The first colon value gives the total field width to print in. The second gives the precision to use. The third specifies exponential notation, irrespective of its value. Four characters are used to print the 'E', the exponent and its sign. The fieldwidth will be exceeded if it is too small to represent the number correctly with the specified precision. For example:

```
-->Print 3.1415:15:3:5, CR, 12.34:9:3:3, CR, 0.0019:3:3:3
      3.141E+00
      1.234E+01
      1.900E-03-->
```

## IP dotted-quad formatting

If you use the string "IP" to format any Venom variable, it will be printed in the standard IP address format of a 'dotted quad':

```
Print 1234567890:"IP"
73.150.2.210-->
```

 See also 'Print message' in the Index for printing each object type.

## Formatting Objects

You modify how an object should print itself by sending *print format parameters*. These follow the object in the print list and are separated by colons. Each object type attaches its own meaning to the parameter values. For example, to [print a Timer](#)'s period as HH:MM:SS, the following format is used:

```
-->Print timer_object :1 :1
01:30:00
```

For more information, see the Print message for each object type in [Object Types](#).

## Formatting strings

You can use the `:` operator to specify how a string is to be printed. One colon, `:n`, will print the leftmost `n` characters from a string. If the number is negative, you get the rightmost `n` characters:

```
-->str := "abcdefghij"
-->Repeat 10 Print "[",str:Index0-5,"]"
[fghij][ghij][hij][ij][j][][a][ab][abc][abcd]-->
```

Using two colon operators allows you to print any portion of a string you wish to, and additionally, will pad out the printed portion with space characters to a required width. This allows you to both select portions of the string and to implement scrolling text.

The first colon specifies where to start printing within the string, and the second specifies how many characters to print. If the start position is negative, or more characters are requested than are in the string, then space characters are printed.

```
-->Repeat 10 Print "[", "abcdefghij" :Index0-5:10 , "]" , CR
[   abcde]
[   abcdef]
[  abcdefg]
[ abcdefgh]
[ abcdefghi]
[abcdefghij]
[bcdefghij ]
[cdefghij  ]
[defghij   ]
[efghij    ]
-->
```

You can use colons to print text Buffers or [String objects](#) in exactly the same way.

Incidentally, you may find it useful to know that string constants understand the all same

messages as String objects.

,

... <list\_item> , <list\_item> ...

Commas separate items in a parameter list, a print list or other lists.

Parameter lists are used when calling procedures, sending messages or in [Make](#) statements or similar keywords. The use of commas in parameter lists is optional. However, you must use commas to separate items in a [Print](#) list.

```
fred(a,b,c,d)
Make vt100 TerminalFilter("VT100", serial2)
k:=keys.Key(3,1)
or
fred(a b c d)
Make vt100 TerminalFilter("VT100" serial2)
k:=keys.Key(3 1)
```

Comma may be used to separate items in [Local](#) declarations.

?

```
? <Int> ⇔ <Int>
?? <Int> ⇔ <Int>
???? <Int> ⇔ <Int>
```

The 'query' operators give direct access to the memory of the controller.

They may be used to access hardware registers, or directly read and write to the RAM or flash memory.

The size of values read or written depends on the number of ? symbols used. One ? reads and writes single bytes, ?? read and write 16-bit half words, and ???? read and write 32-bit words.

Hence to print the byte located at \$F0 in hexadecimal:

```
Print ~? $F0
```

To set the half-word at \$8100 to \$FF00:

```
??$8100 := $FF00
```

Note that the VM2 has a 'Little endian' processor, so in multi-byte accesses the least significant byte is at the lowest memory address. In the previous example, \$8100 is set to \$00 and \$8101

is set to \$FF.

To transfer 4 bytes from \$8100 to \$8104:

```
????$8104 := ????$8100
```



When you use the ? operator, no checking is done on what is being accessed, so using it carelessly is dangerous. In particular, writing to (or even reading) the wrong address can cause unpredictable errors in apparently unrelated parts of the system.



Multi-byte accesses are done atomically with respect to tasks and interrupts, so 'skewed' reads or updates will not occur.

!

```
! <Pointer> ⇒ Any
```

```
! <Pointer> := Any
```

```
(! <Pointer>) (<parameter list>) ⇒ Any
```

The ! (or 'pling') operator is used to follow a pointer, either to read the value of the thing pointed to, or to write it.

## Using Pointers

Pointers are created with @ followed by the name of a global or local variable. ! is used to follow the pointer, and so access the variable:

```
-->a := 99
-->p := @a
-->a := 100
-->Print !p,CR
100
-->!p := 101
-->Print a,CR
101
```

Pointers are usually used for 'pass-by-reference' parameters, getting multiple results from a function and so on.

## Procedure Pointers

Procedure pointers are really the same as other pointers. You just use @ with a procedure name instead of a

In the following example, ! is used to call a comparison routine to decide on the ordering of a sort.

```
;Used for an ascending sort
To ascending(a,b)
```



```

    Return a<b
End

;Used for a descending sort
To descending(a,b)
    Return a>b
End

;Sorts the array it is passed, using the function pointer
To sort(data, func_ptr)
    Local swapped,temp
    Do
        [ swapped := False
          Repeat data.Length - 1
            [ If (!func_ptr)(data.(Index0),data.(Index)) IsFalse
              [ temp := data.(Index0)
                data.(Index0) := data.(Index)
                data.(Index) := temp
                swapped := True
              ]
            ]
        ]
    While swapped
    Return data
End

Array costant_data (8,10)
    7,4,2,6,1,3,9,5,8,10
End

```

Here we test the sorting procedure at the command line:

```

-->cpy := costant_data.copy
-->sort(cpy, @ascending)
-->Print cpy
1
2
3
4
5
6
7
8
9

```

```

    10
-->sort(cpy, @descending)
-->Print cpy
    10
    9
    8
    7
    6
    5
    4
    3
    2
    1
-->

```

### Indirect message calls

`<object> . ! (<MessageRef>) [(<parameter list>)] ⇒ Any`

You can call messages indirectly using the **!** symbol. The syntax is shown above.

For example:

```

msgref := @.Get ; Form a reference to a message
object := New Buffer(Int)
val := object.!(msgref) ; Send a message using the reference.

```

Note that the **()** around the [message reference](#) expression are mandatory.

If you want to send parameters to the message call then you can do it like this:

```
object.!(msgref)(p1, p2)
```

The message reference can be any expression so long as it evaluates to the reference to a valid message.

For example you could store message references in an array:

```
object.!(array_obj.(Index0))(p1, p2)
```

### Current limitations

You can't *assign* to an indirect message send.

All indirect message sending is 'polymorphic' - this means you can't assert the class being called.

@

@ &lt;global name&gt; ⇒ Pointer

@ &lt;local name&gt; ⇒ Pointer

@ .&lt;message name&gt; ⇒ Int

The @ symbol is used to create pointers. These pointers are used to pass information to procedures or messages about how to fetch or set a result, without fetching it or setting it immediately. The detailed uses of this are described under each pointer type separately.

## Creating Pointers

A pointer will point to a variable. A pointer is created by prefixing the name of the variable with @. The pointer may be followed (or de-referenced) by using the ! operator.

One common use of pointers is for 'pass-by-reference' parameters for procedures. Normally the value of a parameter is calculated and this value is passed to the procedure. The procedure cannot 'reach back' and change the value of the original variable. For example:

```
To fail_to_swap(a,b)
  Local temp
  temp := a
  a := b
  b := temp
End

-->x := 1
-->y := 2
-->fail_to_swap( x , y )
-->Print x,y,CR
      1      2
```

Setting the parameter variables a and b in **fail\_to\_swap** does not have any effect on x or y, since they are local to the procedure. In some languages, such as Pascal, it is possible to define a procedure so that it can reach back to the variables in the parameter list. In others, such as C, you need to pass pointers to the procedure, and use pointer operations to read and set the variables. Venom is like C in this respect.

The swap routine written properly in Venom would be:

```
To swap(a,b)
  Local temp
  temp := !a
  !a := !b
  !b := temp
End

-->x := 1
-->y := 2
```

```
-->swap( @x, @y )
-->Print x,y,CR
      2      1
```

This ability to set the caller's variables can also be used to return more than one result from a function.

Note: If pointers to local variables are taken, it is important to ensure that the lifetime of the local variable they point to is not less than that of the pointer, otherwise you will be left with a 'dangling pointer'. This means that the pointer to a local variable must not be used after the procedure that created the local variable has finished. There is no such problem with pointers to global variables.

### Memory addresses

The pointer created by using @ is not the memory address of the variable. However you can find the memory address by converting the pointer to an integer using **As Int**.

You can use memory addresses to access memory directly using the [? operator](#).

### Creating Procedure Pointers

A procedure pointer is a pointer to a procedure, and is created by prefixing the name of the procedure with @.

```
To square(x)
  Return x * x
End

-->operation := @square
```

Procedure pointers are most often used where one of a number of similar procedures might be called with the same parameters.

See [here](#) for how to use a procedure pointer.

### Message references

@ .<message name> ⇒ Int

Message references allow you to effectively take a pointer to a message, so that you can send a message to an object *indirectly*. A message reference is a 16-bit integer.

To get a message reference put the symbols @ . in front of a message name. For example:

```
msgref := @.Put
msgref2 := @.MyMessageName
```

:=

<LHS> := <Any value>

This is the assignment operator, sometimes called *becomes equal to*.

<LHS> indicates a valid Left Hand Side expression and is one of

<variable name>

! <pointer>

<Object> . <message> (<parameter list>)

? <Int memory address>

The assignment operator may be used to set the value of variables, class members, [active variables](#) (i.e. a message to an object that may be assigned a value), or internal [memory locations](#).

()

Parenthesis, or round brackets, are used in three different contexts in Venom2:

### Evaluation order

( <Any> )  $\Rightarrow$  Any

Parentheses, or round brackets, are used to force the evaluation of an expression before normal [operator precedence](#). Parentheses may be nested, that is you may put parentheses inside other parentheses.

For example

```
-->Print 3*4+5, 3*(4+5), 4*(3-(4-5)),CR
        17      27      16
-->
```

Any legal expression of any type can be put in parentheses.

If you need to start a command with parentheses then you need to use square brackets to indicate the start of a command (this is a limitation of not having statement separators like `;` in C):

```
[ (!pointer) .Put (0) ]
```

▼ The maximum level of parenthesis nesting is currently set by the stack space available to the Venom compiler and is not defined as yet.

### Parameter list

( <Any> , ... )

### Formal parameter list

( <Local name> , ... )

Round brackets are used to indicate both formal and actual parameter lists.

A formal parameter list follows the procedure name in a [procedure definition](#). It lists the names of the parameters to the procedure.

An actual parameter list follows a procedure name in a call to a procedure. It lists the values to pass to the parameters of the procedure.

In both formal and actual parameter lists the commas separating the parameters are optional.

```
fred(1 2 a * b + c)
fred(1 , 2 , a * b + c) ;equivalent.
```

▼ The maximum number of parameters you may pass is 253. Each local variable used will reduce the number of parameters available by one.

## [ ]

Square brackets are used in two different contexts:

1. Grouping statements
2. Indicating optional parameters to a procedure

### Grouping statements

[ <statement> ... ]

Square brackets are used to group statements into blocks. The block is then equivalent to a single statement.

Because all 'white space' (spaces, tabs, carriage returns) is treated the same during parsing, this is the only way Venom knows which statements should be blocked together.

As an example, the following two pieces of code do quite different things!

```
If smoke_detected
[
    fire_alarm.On
    sprinklers.On
]
```

```
If smoke_detected
    fire_alarm.On
    sprinklers.On
```

In the latter (incorrect) case, the sprinklers will turn on whether or not there is any smoke.

### Indicating optional parameters

Procedures in Venom can be defined with optional parameters. The optional parameters are indicated by putting a pair of square brackets around them. The optional parameters must always be at the end of the parameter list. All the parameters to a procedure may be declared optional.

For example, in `proc1` there are two optional parameters: **b** and **c**. Parameter **a** must be supplied.

```
To proc1(a, [b, c])
...
End
```

In `proc2`, all the parameters are optional:

```
To proc2([a, b, c])
...
End
```

The Venom value [ParamCount](#) tells you how many parameters have been supplied to a procedure when it is called. The Venom function [Parameter\(n\)](#) returns the value of the *n*th parameter.

Parameters that were not supplied by the caller are set to the value *integer zero*.

### . [Dot]

```
<Object expression> . <Message> (<param list>) ⇒ Any
<Object expression> . <Message> (<param list>) := Any
```

The dot operator is used to send a message to an object.

See each of the [object types](#) for the list of message they accept, and what each one does.

(Note: the same character is also used in floating point numbers to represent the decimal point).

## Active Variables

An *active variable* is a notional value 'inside' an object which can both be read, and also written using `:=`.

An example is Digital's message `.Asserted`, which can be used both to read and write the state of a Digital channel:

```
-->Print Motor_Actuator.Asserted, CR
1
-->Motor_Actuator.Asserted := False
```

Some active variables can also take parameters, like `Buffer.Element(p)`.

For example, to set the tenth element of a Buffer called 'values' to 99, then read it out again:

```
-->values . Element(9) := 99
-->Print value . Element(9)
99
```

## Indirect message send

You can send a message indirectly by taking a [reference](#) to it, and later sending the message [indirectly](#) using the reference.

## ; [Comment]

```
; <characters>
```

*<characters>* is a list of any characters apart from Carriage Return.

The semi-colon symbol is used to introduce a comment into the program. The compiler will ignore text from the semi-colon until the start of the next line.

Comments serve the very important purpose of documenting the program, both for the author and anyone maintaining it.

Comments do not affect the code size or execution speed of your programs.

## Abs

```
Abs <Int expression> ⇒ Int
Abs <Float expression> ⇒ Float
```

The ABS operator returns the absolute value of its operand; in other words, it makes it positive.

For example:

```
-->Print Abs 99, Abs -99
99      99
```

See [operator precedence](#) for which operators are evaluated first.





The absolute value of the most negative integer (\$80000000) has no representation as a positive number within the 32-bit values allowed in Venom2. 0 is returned.

## Acos

**Acos** <Float expression>  $\Rightarrow$  Float

The **Acos** operator returns the inverse cosine of an angle in radians.

See [operator precedence](#) for which operators are evaluated first.



See also [Cos](#) [Sin](#) [Tan](#) [Asin](#) [Atan](#)

## All

All is used only in conjunction with other keywords, such as [List](#), [Stop](#), [Global](#).

## And

<Int expression> **And** <Int expression>  $\Rightarrow$  Int

The And operator returns the *bit-wise* And of its two operands.

A bit is set in the result if both of the corresponding bits in the operands are set. Because of this, And is often used to clear one or more bits while leaving the rest alone. For example

```
Print ~~%101010 And %001100
1000
```

Internally, what happened was this 32-bit operation:

```
And      00000000 00000000 00000000 00101010
          00000000 00000000 00000000 00001100
          00000000 00000000 00000000 00001000
```

See [operator precedence](#) for which operators are evaluated first.



See also [Or](#), [Eor](#), [Inv](#), [True](#), [False](#)



And is a numeric operator - if you really want a logical operator use [AndAlso](#).

## AndAlso

**<Int expression> AndAlso <Int expression>  $\Rightarrow$  Int**

The AndAlso operator is used to determine if two expressions are both *non-zero*.

If both of its operands are non-zero then it returns a non-zero value.

If either of its operands are zero (False) then it returns zero (False).

The AndAlso operator is also 'lazy' or 'short circuiting': if the first operand is zero (False) then it doesn't go on to evaluate the second operand, thus enabling efficient evaluation, and other benefits.

It is guaranteed to evaluate its left hand expression before its right hand expression.

AndAlso is often used in the expression of an If or While construct to make a more complex condition, e.g.

```
If cows_home AndAlso its_milking_time  
    machine . On
```

See [operator precedence](#) for which operators are evaluated first.

 [OrElse](#), [And](#)

## Any

**Any**

**Any** is used to specify a data type as 'any type' when creating a Buffer or defining a Class member.

For example:

```
Make bufferOfAny Buffer(Any)
```

or:

```
Class MyClass  
    MyMember Any  
    ...  
End
```

## Array

Note: the keyword Array can be used in two contexts:

1. To define *constant* arrays - i.e. lists of data values that are fixed
2. With Make or New to create [variable Arrays](#) that can hold *variable* data

Here we will look at *constant* arrays:

```
Array name (<type> [, Int Const size])  
    <type> Const data ,  
    <type> Const data ,  
    ...  
End
```

A Venom Array is a *data structure* - i.e. something that holds a collection of data. The data held by an array is all of the same type, and there is a fixed amount of it.

The kind of array considered on this page holds *constant* data - that is the values of the data never change during the lifetime of the program.

Constant arrays are created at *compile time* - i.e. they are written into your program code alongside, and in a similar way to, procedures. For example, a section of your code might look like this:

```
...  
To simple_procedure  
    led.On  
End  
  
Array simple_array (Int 8,5)  
    1,2,3,4,5  
End  
...
```

### Array names

Arrays are given a name, which has to obey the same rules as any name in Venom.

### Parameters to Array

After the Array keyword, and the name of the array, some parameters are supplied in parentheses:

```
ARRAY name (type , size)
```

**type** specifies the data type of the array elements:

Type	Element data type	Range of values element can hold
<b>Int 8</b>	8-bit integer (unsigned)	0 to 255
<b>Int 16</b>	16-bit integer (signed)	-32768 to +32767
<b>Unsigned Int 16</b>	16-bit integer (unsigned)	0 to 65535
<b>Int</b> or <b>Int 32</b>	32-bit integer (signed)	-2,147,483,648 to 2,147,483,647
<b>Float</b>	Floating point (IEEE single precision)	$\sim \pm 1.0E\pm 38$ , $\sim 7$ digits of precision
<b>String</b>	String constant	Any <a href="#">string</a> constant
<b>@dummy*</b>	Pointer to a global	Any pointer to a global variable

\* or any pointer to a global variable.

**size** (an integer) specifies the *number* of items in the array - i.e. how much data it will hold. **size** is optional - if you don't supply size then the array is made as big as is needed to hold all the data you list.

### Data items

After the array parameters have been supplied, the data that fills the array is listed. Every bit of data must be of the type specified, e.g. integer or floating point.

If there are fewer elements than were declared with **size**, then the array is filled to the end using the value of the last item. If there are too many items, then a warning is given.

You *must* use commas to separate negative numbers or quoted strings or you will get strange results.

### Accessing the array

Once created, a constant array is an Array object, and may be accessed using the same messages as for any kind of array; see the [Array object](#).

### Examples

Here are some examples of small Arrays.

```
;An array of 8-bit integers
Array lookup_data (Int 8)
```

```

    2 , 3, 5, 7 , 6
End

```

*;An array of six string constants, with the last 4 being the same*

```

ARRAY phrases (String, 6)

```

```

    "Hello",
    "Goodbye",
    "Ciao"

```

```

End

```

*;An array of procedure pointers*

```

Array handler (@dummy)

```

```

    @first_proc,
    @second_proc,
    @third_proc

```

```

End

```

See [here](#) how to call procedure pointers.

## As

```

<Float> As Int ⇒ Int
<Pointer> As Int ⇒ Int
<Int> As Float ⇒ Float

```

The As operator is used to 'cast', or change the type of, a value.

The As Int operator takes a floating-point number, and returns the integer part of it, truncating towards zero. For example:

```

-->Print 2.7 As Int, -2.7 As Int
      2      -2

```

If As Int is used on an integer, it has no effect.

As Int will also convert a value of type Pointer to an integer, in this case representing the actual address of the thing pointed to in memory.

For example, in the following code, addr is set to the address in memory of the value of the global variable var.

```

var := 1.0
ptr := @var
addr := ptr As Int

```

The As Float construct takes an integer and gives the floating point representation of it. For example:

```

-->Print 23 As Float

```

### 23.000000

If `As Float` is used on a floating-point number, it has no effect.

Note: Information may be lost when casting in either direction. `As Int` obviously loses the fractional part of the number, but also `As Float` will lose precision from integers larger than ~16,777,216.

See [operator precedence](#) for which operators are evaluated first.



The maximum float which can be cast to an integer for general calculation is 2.1474828E9.

The maximum float that can be cast to an integer for storage in a variable or passing to a procedure or message is 1.0737418E09. The maximum integer which can be cast to a float to be stored in a variable without losing precision is around 17,000,000.



See also [Number limits](#)

## Asin

**Asin** <Float expression>  $\Rightarrow$  Float

The `Asin` operator returns the inverse sine of an angle in radians.

See [operator precedence](#) for which operators are evaluated first.



See also [Cos](#) [Sin](#) [Tan](#) [Acos](#) [Atan](#)

## Assignment

**Assignment**  $\Rightarrow$  Int

**Assignment** is a special integer value that is **zero** when an [active variable method](#) is being *read*, and **non-zero** when an active variable method is being *written to*.

## Atan

**Atan** <Float expression>  $\Rightarrow$  Float

The `Atan` operator returns the inverse tangent of an angle in radians.

See [operator precedence](#) for which operators are evaluated first.



See also [Cos](#) [Sin](#) [Tan](#) [Acos](#) [Asin](#)

## AutoDestruct

**AutoDestruct**

This is an attribute that may be applied to a group of local variables in a procedure (or a method), or to a single member of a [Class](#). It's syntax is slightly different in each case.

**AutoDestruct** is used to indicate temporary objects which should be removed from memory when the entity that holds them is removed.

### AutoDestruct Local variables

**AutoDestruct** is applied to a *list* (or part of a list) of [Local](#) variables to indicate that they hold objects that should be destroyed when a procedure ends.

Any Local variables declared *after* the **AutoDestruct** keyword will be examined when the procedure ends. Each of them that holds an object will be sent the message Die.

Local variables listed *before* **AutoDestruct** are treated as normal: i.e. they are not examined, and objects they hold will not be destroyed.

This is particularly useful if you have to create temporary objects in your code, and don't want to have to think of all the possible places or mechanisms via which your code might exit the procedure, at each of which you would have to explicitly kill all the temporary objects.

#### Example

```
To proc
  Local a, b, c
  Local d
  AutoDestruct ; Objects held by the LOCALs below will be removed
  Local temp_string := New String(100) ; Temporary String object
  ...
  If condition
    Return 0
  ...
  If condition2
    Exit 100
  ...
  Return 1
End
```

**AutoDestruct** of local variables works in conjunction with [End](#), [Return](#), [Try](#), [Exit](#).

It does not act when an error stops a task, or when CTRL-C breaks into the main task.

#### When to use AutoDestruct

There is a simple rule that generally tells you when to use **AutoDestruct** on a local variable:

- If the variable is an object *created inside* the procedure, then use **AutoDestruct**
- If the variable is an object that was *created outside* the procedure, then don't use

## AutoDestruct



If you use **AutoDestruct** indiscriminately you will slow down the speed of Venom, and you may even end up removing objects that should not be removed (which will give the runtime error *Message to dead object*).



See also [New](#), [Zero-memory objects](#), [Garbage Scanner](#)

## AutoDestruct Class members

```
<member name> <type> Autodestruct
```

**AutoDestruct** is an attribute you can apply to any member of a Class that may hold an object.

When a [Die](#) message is sent to a user-defined Class object (and the Die message hasn't been over-ridden) any member that has the **AutoDestruct** attribute will be sent a **Die** message *automatically*. This is used to simplify 'garbage collection' when using Classes.

```
Class MyClass
  ID Int
  MyOwner Class
  MyList Buffer AutoDestruct
  ...
End
```

## When to use AutoDestruct

There is a simple rule that governs when to use **AutoDestruct** on a Class member:

- If the member object is *created inside* the Class (e.g. within **Initialise**), then use **AutoDestruct**
- If the member object is *passed into the Class from outside*, then don't use **AutoDestruct**

## Await

```
Await <Int expression>
```

The Await command waits indefinitely while the condition is false (i.e. zero). When the condition becomes non-zero, program execution resumes at the next statement in the program. For



example:

```
Await the_cows_come_home
milking_machine . On
```

While it is waiting Await will swap tasks.


It is functionally equivalent to, but slightly more efficient than,

```
While <condition> IsFalse
Swap
```

- ⌚ A task waiting in Await ‘swaps out\*’ as soon as it finds the expression is false (zero). Hence the overhead of a task waiting in Await is the time it takes to calculate the expression each time it gets scheduled. Await also allows the task manager to put the controller core to sleep, saving power.

\* ‘Swapping out’ means that the task passes control to the task manager which will find another task to run, or carry on with the current task if there are none.

 [Swap](#)

-  When Await is waiting for an event that is driven by interrupts it may sometimes wait up to 1 mS longer than necessary, dependent on the exact timing of other interrupts and/or the time taken by other tasks. To avoid this you can use **Until** or **While** instead of await:

```
Until condition []
```

## Base

**Base.<memberName>**

The **Base** keyword is used to specify access to a member of the base class.

For example,

```
Class baseClass
```

```
  To method
```

```
    ...
```

```
End
```

```
End
```

```
Class myClass : baseClass
```

```
  To method
```

```
    ...
```

```
    Base.method ; Call method in the base class.
```

```
...  
End
```

```
End
```

Note: if there is no explicit base class then the message will be sent as a [Class-default](#) message.

See also [This](#), [Derived](#)

## Beep

```
Print ... BEEP ...
```

The BEEP print item is used to cause an audible warning on the device printed to. It is equivalent to ASCII character 7 (BEL).

## Break

```
Break
```

Break is used to break out of loops. Break will immediately exit to the next statement past the end of any loop.

```
Every 1000  
[  
    done_it := do_something  
    If done_it = True  
        Break  
]  
carry_on
```

When **done\_it** is true, then execution will immediately carry on at **carry\_on**.

 See also [Forever](#), [Every](#), [While](#), [Do](#), [Repeat](#)

## BS

```
Print ... BS ...
```

The BS print item is used to move backwards one character on the device printed to. It is equivalent to ASCII character 8.

This character is intended to be non-destructive - i.e. it moves back one space, but does not rub out the previous character. To do a destructive delete operation, it is safest to send a BS, then a space, then another BS.

## Call

**Call** (<Int **address**> , <parameter list> , ...)  $\Rightarrow$  Any

**Call** will call external code written in C or Assembler. **Call** takes any number of parameters, but the first one must be the address of the code to call, e.g.

**Call (address\_of\_my\_code)**

Your called code should expect to receive one parameter, equivalent to a **void\*** in C. Your code is responsible for abiding by the ARM procedure call standard.

The parameter is a pointer into the Venom stack, which will have been loaded with the values of the parameters to **Call**, except that the code address (the first parameter) has been overwritten with the number of Venom parameters sent to **Call**. This first parameter position is where you should put any result you wish to return from your external code, via **Call**. Note the Venom stack grows upwards, i.e. it starts at low memory.

### Example

**Call (address , A , B , C)**

Would put the following on the venom stack

```
C ; Top of stack
B
A
⇒ 4 ;No. of Params and Return result.
```

Each Venom value on the Venom stack is 8 bytes wide. The first 4 bytes are the value (LSB first on a 'Little-endian' processor). The other 4 bytes hold the type and other information.

```
Byte 0  1  2  3  4  5  6  7
      [xx] [xx] [xx] [xx] [--] [--] [--] [TYPE]
```

### Example


This C function multiplies two numbers and sends the result back to **Call**.

```
long mult2ints(venom_value * stk)
{
    /* Do the multiplication */
    * stk = (stk + 1)->value.as_int * (stk + 2)->value.as_int;
    /* Set the type of the result to integer */
    * stk -> val_type = VENOM_TYPE_Int;
}
```

*See below for the definitions used.*

To call this code we can use:

```
#define mult2ints $87000001 ; The address of the C function.
result := Call (mult2ints, 10, 20)
```

 Our **CBuilder** application can create a list of **#Define** statements for the absolute addresses of symbols in your external C source code.

## Definitions

Here is the structure of a Venom variable:

```
typedef struct
{
    union
    {
        long as_int;           /* The actual value of an int... */
        float as_float;       /* ... or access it as a float */
        void * as_pointer;    /* or a pointer... */
        unsigned long long_word; /* or one 32-bit word... */
        unsigned short word[2]; /* or two 16-bit words... */
        char byte[4];         /* or four 8-bit bytes. */
    }value;
    char res1;                /* Reserved for future use */
    char res2;
    char write_protect;       /* If set, don't allow writes */
    char val_type;            /* The data type. */
}venom_value;

#define VENOM_TYPE_INT 0 /* The Type-value for integers */
```

## Task Swap Timing

Internally the Venom task manager relies on each task electing to 'swap out'. So as not to contravene Venom's task latency specification your external code should not run for more than 1mS before allowing other tasks to run.

A Venom Operating System call/macro is available: **SWAP\_IF\_I\_MUST**. This should be included in long-duration loops so your code task-switches when necessary.

## Case

```
Select Case <Int expression>
  Case <Int Const> , ...

    <statement>
...
Case Else

  <statement>
```

The Select Case construct allows the choice of one of many actions based on the value of an integer expression.

Case is used in three distinct places: immediately after the [Select](#) keyword that introduces the construct; at the start of each numbered action; and at the start of the Case Else, or default, action.

## Catch

See [Try](#), [Exit](#), [Appendix E: Error messages](#)

## Centre

This special print keyword centre-justifies text in [GraphicsLCD](#).

 [Left](#), [Right](#)

## Char

### Char

Char is used to specify the data type as 'character' when creating new buffers and files that contain text.

For example:

```
Make textBuffer Buffer(Char)
f := fs.Open("MyTextFile.txt", Char)
```

## Chr

```
Print ... Chr <Int expression> ...
```

The Chr print item is used to print 8-bit characters.

The first 128 8-bit characters are often interpreted using the ASCII standard. A table of ASCII values is given in an [Appendix](#).

The second 128 characters (from 128 to 255) are not covered by the ASCII standard, and are likely to be interpreted in widely differing ways by different systems.

▼ The character given must be between 0 and 255.

🔍 See also [Print](#)

## Class

The keyword **Class** allows you to [define your own object classes](#).

It may also be used as the object when you want to send a '[class default](#)' message.

### Class - defining

The keyword **Class** is used to define new object classes. It is followed by the class's *name*, a list of *members*, and it is terminated by the keyword **End**. Here is a very simple example of a class definition:

```
Class Person
  Name String
  Age Int

  To Initialise(n, a)
    Name := n
    Age := a
  End
End
```

You can create objects of this new class using **Make** or **New** in the usual way.

```
p := New Person("Fred", 42)
```

**Make** and **New** both create a new object of the given class, and then send that empty object an **Initialise** message, passing on the parameters from **Make** or **New**.

### Members

A Class is made up of a list of *members*, each of which has a different name, type and function.

Members are normally declared with a *member name* followed by a *data type* and optional

### attributes

Member names may be any normal Venom variable name, including the built-in Venom message names, such as **Reset**, **Put**, **Name**, etc.

### Member types

Member data types may be any of the following:

Type specifier	Type of data stored in the member	Default initial value
<b>Int</b>	Integer (32-bits, signed)	0
<b>Int 32</b>	Integer (32-bits, signed)	0
<b>Int 16</b>	Integer (16-bits, unsigned)	0
<b>Int 8</b>	Integer (8-bits, unsigned)	0
<b>Float</b>	Floating point number (IEEE Single precision)	0.0
<i>Any built-in Venom object type*</i>	Reference to an object	<i>Uninitialised</i>
<b>Class</b>	Reference to an object of a type defined by Class	<i>Uninitialised</i>
<b>Any</b>	Any Venom type (number, object, pointer, etc)	<b>Nil</b>
<b>New String(capacity)</b>	String	<i>Empty String</i>
<b>New Array(type, length)</b> <i>(type must be numeric)</i>	Array	<i>Array filled with integer or floating point zeros.</i>

\*Note that the types **Digital** and **Analogue** may be specified but will be converted to **Any** because of the way they are represented internally.

Each member has a specified data type; all values assigned to that member will be type-checked at runtime.

Integer values won't be range checked, but simply truncated.

### Member attributes

Members are 'Public' by default, but you can prefix any member (or method) with [Protected](#) or [Private](#) to limit its accessibility.

You can also apply the attribute [AutoDestruct](#) to a member, to implement 'garbage collection' in tree-like structures of objects.

## Default initial values

The default initial values for different member data types are shown in the table above. These are the values automatically assigned to members when an object is first created. They may be over-written by your own Initialise method.

## Methods

Methods are a special kind of member in that they are *procedures* rather than *values*. They are introduced with **To**, just like normal procedures. Methods are written in normal Venom code - see [below](#) for minor differences. Here is a simple class with one method:

```
Class Person
  Name String
  Age Int

  To Print
    Print "I am a person called ", Name, CR
    Printf("I am %i years old\n", Age)
  End

End
```

### Methods - differences from normal procedures

Methods are very similar to normal Venom procedures. There are some small differences:

1. The keyword **This** may be used inside a method to refer to the current instance of the class.
2. The Keywords **Base**, **Derived** and **Class** may be used inside methods.
3. Methods may be defined as '[active variables](#)'. The keyword **Assignment** may be used inside active variable methods to determine whether a *read* or a *write* is in progress.
4. **Initialise** methods implicitly return the value of **This** (the current object).
5. The code of a method is usually indented in the file to indicate it is inside the enclosing Class.



## Special methods

### Print

When an object is printed it is sent a message called **Print**. If you want your Class to print itself in a particular way then you should define a method called **Print**. Inside the **Print** method you can use **Print** and **PrintF** statements to generate the text output. These will send their text to the *current* print output stream - ie. the output stream defined by the printing command that printed the object. In the following snippet, the **Person** object **p** is printed to a **String** object, so the text output (generated by the **Print** and **PrintF** commands inside the **Person**'s **Print** method) will be sent to the **String**.

```
Class Person
  Name String
  Age Int
  To Print
    Print "I am a person called ", Name, CR
    PrintF("I am %i years old\n", Age)
  End
End

Make p Person("Jim", 30)
Make str String(100)
Print To str, p
```

You can use **Print To x** or **x.PrintF** *inside* a **Print** method, for example to debug the **Print** method; these statements will direct their print output to **x**, independently of the other print output.

### AcceptPrintJob

This method is called implicitly when you print to an object, using either **Print To** or **PrintF**. The **AcceptPrintJob** method should always take a single parameter, which is a **PrintJob** object (**PrintJob** objects only ever occur in this context). You can pass the **PrintJob** object on to another object as the parameter to an **AcceptPrintJob** message, or you can examine the contents of the **PrintJob** by sending it messages such as **Get**, **Queue** and **Status**.

## Class-default messages

Any class defined with **Class** recognises the set of [Class-default](#) messages.

## Terminating the class definition: End

A Class definition is terminated with **End**.

## Records

You can use classes as data templates, or records, when storing data in files and SafeData. See the *Venom2 Tutorial* for a how to do this.

## Inheritance

A new class can *inherit* the properties of an existing class. This is specified by a colon after the class name, followed by the name of the 'parent' class. The following class, **Employee**, inherits all the members and methods of the **Person** Class, but also adds the integer **PayRollNumber**. So an Employee has a Name, and Age, a method called Print, and also a PayRollNumber.

```
Class Employee : Person
    PayRollNumber
End
```

## Overriding members and methods

If a class defines members or methods with the same name as in a parent class then the new member or method *overrides* the old one.

```
Class Employee : Person
    PayRollNumber
    To Print ; Override Print.
        Print "My Pay Roll Number is ", PayRollNumber
    End
End
```

This means that if you simply send a message to an object of the new class you will by default invoke the new member/methods. However you can access the base class's member/method instead by using the [base](#) keyword.

```
...
To Print ; Override Print.
    Base.Print ; Print as a person first.
    Print "My Pay Roll Number is ", PayRollNumber
End
...
```

## 'Inheriting' Venom base Classes

You can for arrange any message sent to a Class-based object to be passed any one of the Class's members - this is called [message redirection](#), and it allows you to emulate the inheritance of a *Venom base Class*, such as String, Array or Buffer.

## Active Variable Methods

```
To MethodName [(<param list>)] := <name>
...
End
```

Methods of [classes](#) can be [active variables](#). You can declare that a method is an active variable by putting `:=` and a parameter name after any parameter list, or after the method name if there are no other parameters. For example

```
Class MyClass
  member Int
```

```
  To AcVar := val
    If Assignment
    [
      member := val
    ]
    Else
    [
      ...
      Return member
    ]
  End
```

*;This one uses two parameters as well as the assigned value*

```
  To AcVar2(a, b) := val
    If Assignment
    [
      member := val
    ]
    Else
    [
      ...
      Return member
    ]
  End
```

The name after the `:=` behaves just like an [optional parameter](#). A method that is an active variable can't have any other optional parameters.

You can detect whether the call to the method is 'reading' or 'writing' by using the keyword [Assignment](#) to detect whether the method is being read or written to.

### Return value

When reading an active variable method be sure to return a result using **Return**.

When writing there is no need to return a value.

## Message redirection

It is possible to arrange for a defined set of messages to a Class to be passed on (redirected) to a particular member (or members) of the Class. This is called *message redirection*. Message redirection is declared by listing the set of messages to be redirected after the Class member they are to be redirected to, each prefixed by a dot:

```
Class XYString
  XPos Int
  YPos Int
  ; A String member with redirected messages:
  str String
  .Put
  .Get
  .Empty
  .Print
  .AcceptPrintJob
  .PrintF

End
```

In the example above, any of the listed messages, when sent to an instance of the Class, will be passed on to the member `str`.

You can redirect messages to more than one member of a Class, but the sets of messages should not overlap: if any message name appears more than once then the newer one overwrites the older.

## Current limitations

Currently, message redirection only works as seen from 'outside' the Class.

From inside the Class, or a derived Class, you can't refer to redirected messages as if they are normal members of the Class, by their names alone. Instead you have to use explicit accesses, for example like this:

```
str.Empty
str.Put("text")
```

Or you can use this version, which emulates an 'external' view of the Class, though it is a bit slower:

```
Derived.Empty
Derived.Put("text")
```

However, these versions will not currently compile:

```
Empty
```


```
Put("text")

This.Empty
This.Put("text")
```

## Cls

```
Print ... CLS ...
```

The CLS print item is used to clear the screen of an output device. The text cursor is moved to the home position.

 See also [Home](#), [Print](#)

## Cos

```
Cos <Float expression> ⇒ Float
```

The Cos operator returns the cosine of an angle in radians.

See [operator precedence](#) for which operators are evaluated first.


 See also [Sin](#) [Tan](#) [Acos](#) [Asin](#) [Atan](#)

## CR

```
Print ... CR ...
```

The **CR** print item is used to move to a new line on the output device. Some output devices, such as SerialPorts, Files and TCP connections, translate it into the characters 13, 10 (CR, LF).

If you want to embed a new line or carriage return character within a quoted string see [escape sequences](#).

 See also [Print](#);

## Delete

**Delete** <global name>

The **Delete** command is used to set the type of a **global** variable to 'un-initialised'. If the variable refers to a procedure in RAM, its memory is released. If the variable refers to an object, this will be sent the message 'Die' to recover any memory or other resources it held.

Note that **Delete** is not usually used inside a program, but is more often used on the command line.

▼ The name given must be a global name.  
🔍 See also [List Word](#)

## Derived

**Derived.**<memberName>

The **Derived** keyword is used to specify sending a message to a derived class.  
For example,

```
Class baseClass
```

```
  To Action  
End
```

```
  To method
```

```
    ...
```

```
    Derived.Action ; Send the message Action to the derived class
```

```
    ...
```

```
End
```

```
End
```

```
Class derivedClass : baseClass
```

```
  To Action
```

```
    ...
```

```
End
```

```
End
```

See also [This](#), [Base](#).

## Div

**<Int expression> Div <Int expression>  $\Rightarrow$  Int**

The Div operator divides the first operand by the second, giving the integer part of the result. The operands and the result are always integers. Use '/' if you want a floating point result.

See [operator precedence](#) for which operators are evaluated first.

 See also [/](#)

## Do

**Do <statement> While <Int condition>**


The Do construct is used to loop at least once, with the loop condition tested at the end of the loop.

If more than one statement is used within the construct, they must be surrounded by square brackets []:

```
Do
[
    shout("Hoi")
    lean_on(gate)
    Wait a_while
] While the_cows_come_home
```

The special loop count values [Index0](#) and [Index](#) increment each time round the loop. These values are available in the statement and in the expression.

You may terminate any loop in Venom with the [Break](#) command.

 See also [While](#)

## Else

```
If <Int condition> Then
    <statement>
Else
    <statement>
```

Else is part of the If ... Then ... Else ... construct, which is described in more detail under [If](#). It is followed by a statement, which is performed if the condition evaluated to the value 0 (zero).

Else is also part of the [Select Case](#) construct.

## End

```
To <procedure name>
```

```
...
```

```
End
```

~Or~

```
Program End
```

~Or~

```
Array <array name>(...)
```

```
...
```

```
End
```

End is part of the definition of a procedure, described in more detail under [To](#). It serves to mark the end of the procedure.

It is also used as part of the [Program](#) and [Array](#) constructs to mark the end.

## Eor

```
<Int expression> Eor <Int expression> ⇒ Int
```

Eor calculates the bit-wise exclusive-or of its two operands.


A bit is set in the result if either of the corresponding bits in the operands are set. Eor is often used to flip one or more bits while leaving the rest alone. For example:

```
Print ~~%101010 Eor %001100
100110
```

Internally, what happened was this:

```
Eor      00000000 00000000 00101010
          00000000 00000000 00001100
          00000000 00000000 00100110
```

See [operator precedence](#) for which operators are evaluated first.

 See also [And](#), [Or](#), [Inv](#), [True](#), [False](#)



## Every

**Every** <Int expression> <statement>

The Every command is used to repeat code at regular intervals. The period is specified in milliseconds.

The special loop count values [Index0](#) and [Index](#) increment each time round the loop. These values are only available in the statement.

You may terminate any loop in Venom with the [Break](#) command.

If the statement takes longer to execute than the specified interval, Every will not attempt to regain the lost time.

- 🕒 As soon as the statement has been executed, the Every command swaps tasks to allow other tasks to run. If there are no other tasks ready to run, the processor core is put to sleep to save power.

🔍 See also [Forever](#), [Wait](#)

## Exit

**Exit** <Int expression>

Exit is used to generate a 'user exception' - basically a deliberate runtime error - that will immediately jump out of [Try](#) and may be handled by Catch.

## Exception handling

Exit allows your program to 'escape' to a higher level to handle an event that can't easily be handled at the point in the program where it happened. For example in a low level device driver you suddenly find the device you are driving is no longer present, then you can jump right out of the device driver.

```
Try
[
    complicated_device_driver_code_1
    If something_went_wrong
        Exit 103 ; our error code.
    complicated_device_driver_code_2
]
Catch exception
[
    Select Case exception
    ...
]
```

## Exit 0

If you use Exit with the value **zero** (0), then program control will exit from the Try/Catch construction, *but Catch will not catch the exception*.

This can be useful for a simple way to jump out of nested loops, as well as having other uses.

## Loop exit

Exit allows you to jump out of multiple layers of loop nesting, unlike Break which only breaks out of one level of loop;


```
Try
[
    Forever
    [
        Every 10
        [
            If condition
                Exit 0 ; Jump out of the loops
        ]
    ]
]
```

## Simulate runtime errors

Exit also allows you to simulate the effect of a runtime error:

```
Exit 1 ; Simulate CTRL-C
```


 See also [Try](#), [Appendix E: Error messages](#)

 You can Exit with any integer in the range 0 - 255, but if you are using Exit to generate your own exception values it will probably be useful to choose a range of number that is outside the [system runtime error codes](#). For example in the range 100 - 255.

## Exp

**Exp** <Float expression>


The Exp function returns the value of 'e' raised to the power of an expression.

 See also [Log](#)

## False

**False**  $\Rightarrow$  Int

False is a convenient constant exactly equivalent to integer zero.

 See also [And](#), [Or](#), [Eor](#), [Inv](#), [True](#)

## Float

**<Int expression> As Float**  $\Rightarrow$  Float

Float is used as part of the As Float operator to convert the given value to floating point. See [As](#) for more information.

It is also used as a type specifier in the [Class](#) construct and for [Buffers](#) and [Files](#).

## Font

**Print ... FONT <Int expression> ...**

The FONT print item is used to set the character font of text on the device printed to.

 See also [GraphicsLCD](#).


## Forever

**Forever <statement>**

The Forever command is used to repeat the following statement indefinitely.

The special loop count values [Index0](#) and [Index](#) increment each time round the loop. These values are only available in the statement.

You may terminate any loop in Venom with the [Break](#) command.

 See also [Every](#)

## Global

By default global variables are not accessible (i.e. they are *out of scope*) inside a class, and also global variables are overridden by local variables or parameters with the same name.

You can override these defaults in two different ways:

## Enforcing a global name

**Global**.<name>

Here **Global** is used to force the compiler to see a global variable where the same name might also be used for a [local](#) variable or a [Class](#) member.

For example:

```
Global.Width := 200
```

Note the use of a dot.

## Accessing globals inside a Class

**Global** <name> , <name> , ...

**Global** may be used to list all the global variables that are to be accessible from inside a particular class definition. **Global** must occur before any members or methods are declared:

```
Class MyClass
  Global lcd, touch, serial
  member Int

  To method
    Print To serial, touch.XPos, CR
  End
End
```

## GotoXY

**Print ... GotoXY (<parameter list>) ...**

The GotoXY print item is used to move the cursor to a particular location on the screen on the output device.

```
Print To glcd , GotoXY (10 , 40)
```

 See also [AlphaLCD](#), [GraphicsLCD](#).

## Has

`<user defined object> Has <message name> ⇒ Int`

The **Has** `<name>` postfix operator allows you to determine at runtime if a particular user defined object has a particular member or method.

**Has** will return **True** (1) if the object accepts the message, or **False** (0) if not.

If the first parameter is not a user defined object (e.g. it is an object of one of the pre-defined classes, or an integer, say) then **Has** will throw a [type mismatch error](#).

### Example

```
If obj Has Reset
[
    ...
]
```

## Help

**Help** `<anything>`

**Help** followed by a global variable name tells you something about what that variable contains. This is particularly useful for objects.

**Help** can also give information on keywords, etc, but these functions are now better supplied within the *VenomIDE* development environment.


 See also [Print](#), [List](#)

## Home

**Print Home**

The Home print item is used to set the cursor position to the top left on the output device. For example:

```
Print to lcd, Home, "Hello world"
```

 See also [Print](#)

## If

```

If <Int condition>
    <statement>
Else
    <statement>

```

The If... Else ... construct is used to conditionally execute a statement, or to choose between two statements, on the basis of the result of an expression (the 'condition').

If the condition is not False (ie. True or any non-zero integer), the first statement is executed. If the condition is False (i.e. integer zero), the statement following Else is executed.

The Else portion is optional.

For example:

```

To count10
Repeat 10
[
    If Index=5
        Print "    Five"
    Else
        Print Index
]
End

```

```

-->count10
      1      2      3      4  Five      6      7      8      9      10

```

## Else association

Note that if you have nested If constructs then any Else statement will be associated with the last If. The example below won't work in the way the indentation seems to imply:

```

If temperature < 23.5
    If switch.Asserted
        heater. On
Else
    heater. Off

```

To make it work you need brackets [ ], used like this:

```

If temperature < 23.5
[
    If switch.Asserted
        heater. On
]
Else
    heater. Off

```

## Optional 'Then'

There is an optional keyword, [Then](#), that may be used to clarify complicated If constructions. It goes between the If's condition expression and the If's statement, for example:

```
If a AndAlso b OrElse c IsFalse Then complicated_statement
```



See also [Select Case](#)



Don't use **And** in your condition expressions unless you actually intend to perform a bitwise combination of two bit patterns: use [AndAlso](#).

## Index and Index0

**Index**  $\Rightarrow$  Int

**Index0**  $\Rightarrow$  Int

Index and Index0 count the number of times round the current [Forever](#), [Every](#), [Do](#), [While](#) or [Repeat](#) loop. Index counts from 1, and Index0 counts from 0.

Only the Index of the most recently entered loop is available, so if you have nested loops and need the outer Index, you need to store it in a variable. The following example demonstrates this, and prints a times-table:

```
To times_table_up_to(max)
  Local outer_index
  Repeat max
  [
    outer_index := Index
    Repeat max
    [
      Print outer_index:1, " times ", Index:1, " is ", outer_index*Index
    ]
  ]
End
-->times_table_up_to(10)
1 times 1 is 1
1 times 2 is 2
1 times 3 is 3
...
2 times 1 is 2
2 times 2 is 4
...
10 times 9 is 90
10 times 10 is 100
```

▼ Index counts up to the maximum integer, \$7FFFFFFF. It then wraps round to the minimum integer, \$80000000, and continues back up through zero again. For a loop being executed 50 times a second, the wrap-round will occur after just under a year.

Hence if you use Index in extremely long-term loops, you should ensure that you allow for the case where it becomes extremely large. Using And to create a lower-range Index which repeats itself is one way of doing this.

## Int

**<Float expression> As Int** ⇒ Int

Int is used as part of the As Int operator to convert the given value to an integer. See [As](#) for more information.

🔍 See also [As](#)

It is also used as a type specifier in the [Class](#) construct and for [Buffers](#) and [Files](#).

## Inv

**Inv <Int expression>** ⇒ Int

The Inv operator gives the *bitwise complement* of its operand: each bit in the operand is ‘flipped’ to give the result.

Inv is most often used to invert the state of the bits in a number when manipulating bit patterns.

```
result := Inv inputbits And %10001
```

Because of the two's complement integer number system used by Venom, the following equation is always true:

```
Inv x = - ( x + 1 )
```



To to a logical inversion of a true or false value use IsFalse, not Inv.

## Is

**<Any expression> Is <Class name>** ⇒ Int

The Is operator tests an object's type, including any [inheritance](#). Is returns **True** if the



expression evaluates to an instance of a given Venom internal class, or a given user Class that either *is* or *inherits* the named Class.

For any other value of the expression **Is** will return **False**.

### Example

```
If obj Is MyClass
[
    obj.message
]
```

```
If obj Is Digital
[
    obj.Off
]
```

### IsFalse

**<Int expression> IsFalse**  $\Rightarrow$  Int

The IsFalse operator returns True for a zero (False) operand, and returns zero (False) for a non-zero operand.


It is most often used to construct conditions for If, While, etc:


```
If input_switch.Asserted AndAlso interlock.Asserted IsFalse
[
    ...
]
```

The two statements below are equivalent:

```
condition := x IsFalse
condition := (x = False)
```

**IsFalse** is a *postfix* operator - that is it comes after the operand.

 See [operator precedence](#) for the order in which operators are evaluated.

 To do a bitwise inversion of a bit pattern value use [Inv](#)

## Left

This special print keyword left-justifies text in [GraphicsLCD](#).

 [Right](#), [Centre](#).

## List

```
List <name>
List All
List Word
List Task
List Define
List Class
```

The list command prints a list to the terminal.

- **List** <procedure name> - lists a summary of a particular procedure
- **List** <class name> - lists a summary of a particular Class
- **List All** - list summaries of all global procedures
- **List Word** - list all global symbols
- **List Task** - lists all the active tasks
- **List Define** - lists all macros
- **List Class** - lists all defined classes in an inheritance tree

An exception is **List startup**, which will list the source of Venom2's *default* startup procedure. If you have written your own startup, then its source will not be listed.

 See also [Word](#), [Task](#), [Define](#)

## Local

```
Local <Local name>, <Local name>, ...
```

```
Local <Local name> := <Any expression>, ...
```

Local declares one or more local variables for a procedure. These are variables that can only be used within the procedure, and which are created each time the procedure is called.

Giving procedures their own local variables makes a program more reliable because you can be sure that another part of the program can't change the value of the variables by mistake.

If a procedure is recursive, or more than one task is using it, each instance of the procedure call has its own copy of the local variables.

All Local definitions must appear before any other statements in the procedure. They may be separated with commas, or with the keyword Local.

Local variables may be initialised with `:=` when they are defined. You can use any expression, including previously defined local values. Un-initialised local variables currently default to the value integer zero.

```
To proc
  Local val := 1.0 , val2 := 1.1
  Local loc_obj := New Digital(128)
  Local val2 ; not initialised
  ...
End
```

### Local names take precedence

Local variables will 'override' any global variables or Class members with the same name. That is, it is the local variable that is referred to by default when that name is used.

If you want to use the global or the member names then use [Global](#) or [This](#) to resolve the name conflict.

### Autodestruction

If you are using local variables to hold temporary objects then you can use the [AutoDestruct](#) keyword to indicate that the objects should be automatically destroyed when the procedure ends.

- ▼ The maximum total number of locals and parameters a procedure can have is 253.
- ⚙ Having local variables makes the procedure marginally slower to call, but they are marginally faster to access than global variables.

## Log

**Log <Float expression>**

The Log function returns the natural logarithm ( $\text{Log}_e$ ) of an expression.

🔍 See also [Exp](#), [Power of ^](#)

## Make

**Make** <global name> <Object Class> (<parameter list>)

The Make command is used to create an object of the given type, and put it in a global variable.

The parameters to make each type of object are given in the Creation section for each object.

As an example, the following statement creates a Digital object called motor\_power, on digital channel 128:

```
Make motor_power Digital(128)
```

By convention, Make is used to create permanent objects. I.e. the application never needs to kill them with Die. Where possible it's a good idea to put all your Make statements in the init procedure, which is normally called by startup shortly after power-on.

On the other hand, temporary objects (which may be created and deleted while the application runs) are, by convention, created with the [New](#) function.

### Make or New?

There is not much difference between Make and New. The choice of Make or New can indicate the way the object is to be used. As an aid to thinking about how an object is to be used, Make will not work with [Local](#) variables or [Class members](#).

 [New](#)

## Mod

**<Int expression> Mod <Int expression>**  $\Rightarrow$  Int

The Mod operator divides the first operand by the second, and gives the value of the remainder. Mod can only work with integers.

Mod is often used to make numbers 'wrap round'.

*If either operand is negative the sign of the result is implementation-dependant, but the following expression will always be true for all integer values of a and b.*

```
a = a Mod b + a Div b * b
```

The following example shows a simple pseudo-random number generator that uses Mod to stop the calculated number becoming too large and causing an integer overflow. The result of the random number generator is then used to calculate a dice throw from 1 to 6.

```
;sets the seed from the clock  
;the Mod prevents overflow later  
To set_random_number_seed
```

```

    seed := clock . Time Mod 259200
End

;Calculate next seed value with algorithm and constants
; taken from Numerical Recipes in C (Press et. al.) Ch. 7
;Randomness subject to warnings contained therein!
To large_random_number
    seed := ( seed * 7141 + 54773 ) Mod 259200
    Return seed
End

;Give a number from 1 to 6. We don't use Mod here
; because the bottom bits of the number are not random.
To dice_throw
    Return (6 * large_random_number) Div 259200 + 1
End
-->set_random_number_seed
-->Every 500 Print dice_throw
    3      6      4      2      5      3      5      2      4      1
    1      5      3      4      5      5      3      6      2      1      6
    1      4      4      3      5      5      3      4      2      4      3

```

See [operator precedence](#) for which operators are evaluated first.

🕒 If you want to Mod with a power of two, using [And](#) is faster:

$x \text{ Mod } 2N = x \text{ And } (2N - 1)$

eg. for  $x \text{ Mod } 16$  use  $x \text{ And } 15$ .

## New

**New <Object Class> (<param list>) ⇒ Any**

The New function is used to create an object of the given type and parameters. The parameters taken by individual object types are given under Creation for each object type in the [Object Types](#) section.

As an example, the following function creates a temporary **DateTime** object, which is then set to the current time. The object will be automatically removed when the procedure exits:

```

To now
    AutoDestruct
    Local dt := New DateTime(clock.Time)
    ...
End

```

New is generally used when creating temporary objects held in local variables, rather than setting

up permanent objects at the beginning of an application (where, by convention, [Make](#) is used). [Make](#) has been disabled from assigning to local variables as an aid to indicating the use of each object.

### Garbage collection

If you don't remove temporary objects after you have used them they will accumulate in memory, eventually using it all up, and so crash your application. To prevent this you can use the [AutoDestruct](#) mechanism to remove them automatically when a procedure ends.

If you don't use **AutoDestruct** then you have to keep track of temporary objects explicitly, sending them the `Die` message at each point your procedure might exit.

### Returning an object created with New

You should *not* use **AutoDestruct** for situations where a procedure returns an object it has created, for example:

```
;Returns a DateTime initialised to the current time.
To return_a_datetime
  Local dt := New DateTime(clock.Time)
  Return dt
End
```

### Nil

**Nil**  $\Rightarrow$  Nil

**Nil** is a special object that is used as a placeholder for any other type of object, or to represent 'no object'.

**Nil** will accept any message, and will return **False** to all of them.

It is useful for situations where, in the normal course of events, an object is used but sometimes you wish to allow for situations where 'no object' needs to be represented.

The most common example of this is substituting the value **Nil** for an output stream object when you want **Print** output to be discarded:

```
lcd := Nil ;all Print To lcd will now be discarded.
```

Testing for **Nil** is done with `=` and `<>`, for example:

```
If lcd = Nil
[
  ...
]
```

## Nop

### NOP

The **NOP** statement introduces a *no-operation* bytecode into the compiled Venom code. **NOP** doesn't do anything except taken some space in the code and take a short amount of time to execute.

### Runtime error reports - improving line number accuracy

**NOP** is most often used to improve the line-number accuracy in runtime error reports (because it introduces more embedded source-line numbers into the code)

E.g. if you were getting a runtime error somewhere in this block of code:

```
Make a AlphaLCD(20,2)
Make eeprom SafeData(2,1,162)
Make d digital(128)
```

you might do this - the runtime error would be narrowed down to just one of the Make statements.

```
NOP
Make a AlphaLCD(20,2)
NOP
Make eeprom SafeData(2,1,162)
NOP
Make d Digital(128)
NOP
```

## Or

**<Int expression> Or <Int expression> ⇒ Int**

The Or operator returns the *bit-wise* Or of its two operands.

A bit is set in the result if either of the corresponding bits in the operands are set. Or is often used to set one or more bits while leaving the rest alone. For example:

```
Print ~~%101010 Or %001100
101110
```

Internally, what happened was this:

```
Or      00000000 00000000 00101010
        00000000 00000000 00001100
```

00000000 00000000 00101110

See [operator precedence](#) for which operators are evaluated first.



See also [Or](#), [Eor](#), [Inv](#), [True](#), [False](#)



Or is a numeric operator - if you really want a logical operator use [OrElse](#).

## OrElse

`<Int expression> OrElse <Int expression> ⇒ Int`

The OrElse operator is used to determine if at least one of two expressions are *non-zero*.

If either of its operands are non-zero then it returns a non-zero value.

If both of its operands are zero (False) then it returns zero (False).

The OrElse operator is also 'lazy' or 'short circuiting': if the first operand is non-zero then it doesn't go on to evaluate the second operand, thus enabling efficient evaluation, and other benefits.

It is guaranteed to evaluate its left hand expression before its right hand expression.

OrElse is often used in the expression of an If or While construct to make a more complex condition, e.g.

```
If temp > 50.0 OrElse testing_fan
    fan . On
```

See [operator precedence](#) for which operators are evaluated first.



[AndAlso](#)

## ParamCount

`ParamCount ⇒ Int`

**ParamCount** is an integer value giving the actual number of parameters supplied to a procedure that has [optional parameters](#).


It may be used with **Select Case**, etc, to determine how a procedure should respond, given the number of parameters supplied.



Note: Those parameters that were not supplied in the call will be set to the value *integer zero*.

### Example

```
To proc(a,[b,c])
  Select Case ParamCount
    Case 1 ; Only one parameter supplied.
    [
    ]
    Case 2 ; Only two parameters supplied.
    [
    ]
    Case 3 ; All three parameters supplied.
    [
    ]
  End
```

 [Parameter\(n\)](#); [Select Case](#)

### Parameter

**Parameter**(Int n) ⇒ Any

**Parameter**(n) is a Venom function that returns the value of the Nth parameter passed to a procedure or method. It is most often used when there are [optional parameters](#).

The first parameter is **Parameter**(1) the last is **Parameter**(ParamCount). If you try to access parameters that don't exist then a Range Error is thrown.

### Example

```
To procedure(a,b,c)
  Repeat ParamCount
  [
    Print Parameter(Index),CR
  ]
End
...
procedure(3,"two",1)
  3
two
  1
-->
```



## Print

```
Print <print item> , ...
Print To <Object> , <print item> , ...
```

The **Print** command is used to send text to an output stream or text-handling object. You can also use the [PrintF message](#) to send text to an object.

By default print output is sent to Serial Port 1 (the default output device). During development this is usually connected to your terminal.

For example:

```
-->Print "Hello world"
Hello world
```

The Print keyword is followed by a list of print items separated by commas. These may be string constants, numbers, objects, expressions, or special print keywords, such as **CR** (Carriage Return) in this example.

```
-->Print "Hello world ", clock, " ", 2 * 3, CR
Hello world 2011-02-14 11:37:19 6
-->
```

## Print formatting

Each type of value will print in a default way - e.g. integers will print in a default field width of 1 digit. You can force different formatting of almost any kind of value type using the [colon formatting operator](#). E.g. here we print an integer in a field width of 10 digits:

```
Print 1234 : 10
1234
```

## Printing objects

Each object responds to being printed in its own way. Objects also respond to colon formatting operators.

```
-->Print clock , " ",clock : 4
2011-02-14 11:37:19 11:37:19 am-->
```

See [here](#) for how the clock object responds to colon printing formatters.

## Special print keywords

Print has a set of keywords for special actions. Not all of these are understood by every output device.


Print keyword	Action
<b>CLS</b>	Clear Screen
<b>Home</b>	Go to the Home position
<b>CR</b>	Carriage Return (new line)
<b>HTAB</b>	Tabulate horizontally
<b>VTAB</b>	Tabulate vertically
<b>GOTOXY</b>	Go to a position
<b>FONT</b>	Set the font style
<b>Left</b>	Justify to the left
<b>Right</b>	Justify to the right
<b>Centre</b>	Justify to the middle

## Print redirection


When **Print** is followed by **To**, the output can be sent to any object, for example:


```
Print To file, "Data readings", CR, data_array
```

You can also redirect print output by changing the [default output stream](#) object.

 Print will lock the output device for the duration of the Print command, and then unlock it afterwards. There is no further locking for 'nested' printing, where a procedure (or more likely, a [method](#)) that includes Print statements has been called from Print.

 The [colon format operator](#) : allows you to re-format print output

 The [Printf](#) message is accepted by any object that can accept print

 The [OperatingSystem.Output](#) message changes the default output streams.

## Program

```
Program "<filename>" Int line_number Int checksum Int
flags
... procedures ...
Program End
```

*Old syntax for this command is still supported and is listed [here](#).*

The **Program** command is used when downloading Venom2 source code files to the controller. **Program** is usually generated by the VenomIDE development toolset so you are not likely to need to use it directly.

When downloading program source text within the **Program/Program End** construct, echoed characters and prompts are turned off – instead a summary of each procedure is printed. Errors are reported as they occur, and are summed up at the end. **Program** will stop any background tasks.

Pressing CTRL-C will escape from **Program**.

### Parameters

The name of the file being downloaded should be supplied as a string constant.

The optional **line\_number** parameter gives the line within the file of the Program statement. If line number is not present the value used internally defaults to 1 to allow the Program command to be used explicitly within a text file (when you are not using VenomIDE).

The optional **checksum** parameter gives the checksum of all the characters sent between the Program ... and Program End statements. This is later used by Venom to check the integrity of the code that has been downloaded. If the checksum matches with the sum of the actual characters sent then 'Checksum OK' is reported at the end of the download. This parameter only likely to be used by VenomIDE.

The optional **flags** parameter is for fine tuning of the download process. The flags are sent as binary bits set in an integer value. The bit values are:

Bit number	Function	Description
0	Don't Echo Procedure Names	Don't report each procedure name as it is downloaded
1	Remove all procedures that were in this file before defining new ones	Helps to keep the program in VM2 RAM tidy by removing redundant code.
2	No Download report if OK	Don't report the end of download, or validation, etc, if it all went OK

3	Last file in batch	This is the last file in a multi-file download - a signal to analyse the code, etc.
4	Show Code analysis	Show an analysis of the code after the download
5	Show warnings	Switch on warnings in the code analysis
6 & 7	Handshake to use	Sets the serial handshake method to use during the download process. This is the same as the value used by <b>SerialPort.Handshake</b> .

### Program: Old syntax

**Program Start**

**Program End**

This is the old syntax for Program. You might need to use it if you don't have access to VenomIDE, but need to download code using a simple terminal emulator. Put these commands at the beginning and end of each Venom code file to suppress echoing back of the code as it is downloaded.

### Private

**Private** <membername> <datatype>

**Private To** <methodname>

**Private** specifies that a member or method in a class should only be visible from inside the class where it is defined.

See also [Public](#) and [Protected](#).

## Protected

**Protected** <membername> <datatype>

**Protected To** <methodname>

**Protected** specifies that a member or method in a class should only be visible from inside the class where it is defined, and derived classes.

See also [Public](#) and [Private](#)

## Public

**Public** <membername> <datatype>

**Public To** <methodname>

**Public** specifies that a member or method in a class should be visible from anywhere - inside and outside - the class. Public is the default accessibility setting for all members and methods, so it is not strictly necessary to use it.

See also [Private](#) and [Protected](#).

## Repeat

**Repeat** <Int count> <statement>

Repeat executes a statement a pre-determined number of times.

The count expression is evaluated only once, at the start of the loop.

If the count is less than 1, the statement is not executed at all.

The system variables [Index](#) and [Index0](#) count the loop iterations.

The following example shows Repeat being used to print a line of stars of a given length:

```
To stars(n)
  Repeat n
  [
    Print " * "
  ]
End

-->stars(10)
```

\*\*\*\*\*-->

▼ The maximum value of count is \$7FFFFFFF, the maximum positive integer.

🔍 See also [Break](#)

## Return

**Return** <Any>

The Return statement is used to return a value from a procedure. The procedure is exited immediately.

In the following example, Return is used to make a function that cubes its parameter:

```
To cube(x)
  Return x*x*x
End
-->Print cube(4),CR
    64
-->Print cube(4.6415888)
    100.0
```

Return can also be used to leave a procedure early, even if the procedure doesn't normally return a result. You can just use a dummy return value, like integer zero.

```
To proc
  If leave_early
    Return 0
  carry_on_processing
End
```

▼ The value returned may be of any type - including any numerical value or object type.

🔍 [To](#), [End](#)

☠ Beware of returning pointers to local variables. See [@](#) for more details on this.

## Right

This special print keyword right-justifies text in objects like the GraphicsLCD.

🔍 [Left](#), [Centre](#).

## Select

```

Select Case <Int>
  Case <Int Const> , ...
    <statement>
  ...
  Case Else
    <statement>

```

The Select Case construct allows the choice of one of many actions based on the value of an integer expression.

Each case in the construct is labelled with one or more integer constant values, and has one statement to execute. If the value of the expression matches one of the case values then the statement is executed. If it matches none of the cases, then the Else case statement is executed. The Else case is optional and may appear anywhere in the list of cases.

```

To test_select(n)
  Select Case n
    Case 1,2
    [
      Print "it was 1 or 2"
    ]
    Case 3
    [
      Print "it was 3"
    ]
    Case 200
    [
      Print "it was 200"
    ]
    Case Else
    [
      Print "it was none of them!"
    ]
  End

```

There is no check for duplication of the case values; matches are tested from the top down.

Note: You may nest Select constructs, but watch out: the case statements may get associated with the wrong Select statement unless you use square brackets `[]`, just as in If/Else.

- ⚙ The first cases are the fastest to check, but it's only ~3.5  $\mu$ S per case (measured on the VM2 with a 72MHz clock). The Else case is always reached after checking every other case, no matter where it appears in the list of cases.



⚠ Currently the case constants must be expressions that evaluate to a constant in the range - 32768 to 32767, and the number of cases must be less than 65535.

## Sin

**Sin** <Float> ⇒ Float

The Sin operator returns the sine of an angle in radians.

See [operator precedence](#) for which operators are evaluated first.

🔍 See also [Cos](#) [Tan](#) [Acos](#) [Asin](#) [Atan](#)

## Sqrt

**Sqrt** <Float> ⇒ Float

The Sqrt function returns the square root of an expression.

## Start

**Start** <statement> ⇒ Task

The Start command starts a new task. The statement is set running as an independent task, in parallel with existing tasks.

Start returns a result immediately. The result is a [Task](#) object, which may be used to control the task, or request information about it.

### Keep tasks simple

It is good practice to write applications where Start's statement is simply a procedure call, rather than a more complicated block of statements:

```
Start monitor_procedure
```

### Listing tasks

You can use [List Task](#) to get a list of all the tasks currently running.

If you don't have a command line because code is running, then you can hit CTRL-T to get the same effect (so long as [SerialEscape](#) is not disabled).

📄 Currently a task will take four blocks of memory. One is for the copy of the task's code. One is for the task object (~100 Bytes) and two are for the task's Venom stack (1600 bytes) and processor stack (1000 bytes).

⚠ There is no limit other than memory to the number of tasks you may use. However, if your

application needs to use more than four or so, you should consider re-writing it.

- ⌚ Each task is given slots of around 1mS (2mS max), in a round-robin cycle. Hence the more tasks there are, the less often any one of them gets a slot. If the task is waiting for something, such as in `Await`, reading a character or locking an object, it will swap out to the next task as soon as it realises it has to wait, and will not take the full slot time. Any task that is waiting on a timing event or an interrupt will cause the processor to sleep to save power if possible.

🔍 See also [Stop](#)

## Referring to Local variables in Start

Even though a new task runs in a completely different context (a different *stack*) to the context of `Start`, it is still possible to refer to local variables from the old context in the new task.

This is made possible because `Start` first copies local variables into the new context before actually starting the new task.

This means that the new task can only refer to *copies* of the local variables, taken at the point `Start` is executed, not their on-going values.

Note that the parameters to a procedure, and the object (**This**) in a [method](#) call, are also part of the local context and so may be referred to in `Start`.

Currently the `Index` and `Index0` of a loop containing `Start` *can't* be used this way.

Example:

```
To main
  Local ind
  Repeat 4
  [
    ind := Index0
    Start thread(ind)
  ]
End
```

## Stop

```
Stop <Task expression>
Stop <Int expression>
Stop All
```

The `Stop` command is used to stop another task. The [Task object](#) or the task's ID number may be used to identify which task to stop.

The task ID number is a unique 32-bit number assigned to each task when it is created. `List Task` will list all the tasks together with their task ID numbers.

The command line task (Task ID 0) cannot be stopped.

Stop All will stop all tasks except the command line task. If you type CTRL-C at an empty command line, then Stop All will be applied to any tasks running.

*Note that in rare circumstances (e.g. if the USB connection to the Flash File System is currently writing to the Flash memory) then the task that handles this writing, which is automatically started and stopped by the operating system, will not be stopped by Stop All. It will stop when it has finished writing to the Flash memory.*

 See also [Start](#), [List Task](#)

## Stopping tasks 'cleanly'

When a task is stopped with Stop or Stop All, it is sent a [special runtime error](#). This runtime error is just like any other except that it will not generate any error text, nor will it cause the controller to reset.

This error may be trapped using [Try / Catch](#), which is useful if you want to make your tasks clean up after themselves when they are asked to stop.

There are two kinds of clean up that you might want to do:

1. Unlocking any global objects used by the task. This can be handled using **Lock (0)** - see the [restored locking](#) scheme.
2. Removing 'local' objects created by the task. This can be handled using [AutoDestruct](#).

For example:

```
To a_tidy_task
  Local error_number
  AutoDestruct
  Local buff := New Buffer(Char)

  Try
  [
    Every 1000
      do_something_with(buff)
  ]
  Catch error_number
  [
    If error_number <> Task_DEATH_ERROR
      Exit error_number ; Handle other errors 'further up'.
  ]

  ; Tidy up when this task ends:
  make_outputs_safe
  ; Ensure these are unlocked if there is
```

```
; any chance they were left locked by this task.  
global_object_a.Lock(0)  
global_object_b.Lock(0)  
global_object_c.Lock(0)  
End
```

### Notes

If your task clean-up code doesn't unlock all the resources locked by a dead task then any task that attempts to lock a locked resource may wait for a little while before the system detects that the lock may be released to a new owner. You can optionally elect to have the system issue a runtime error in this situation: *Attempt to lock object held by dead task*. See [here](#) for how to turn on this runtime error.

If the tasks stopped by Stop or Stop All take time to 'clean up' then they may not have stopped by the time the Stop command has completed. If this is a problem for you then you can detect this with [Debug\(8\)](#), or by sending the Done message to the task object.

You can prevent all other tasks from running using [Debug\(20\)](#).

## Swap

### Swap

Swap will cause an immediate task swap. If there are no other tasks ready to run then Swap will allow the processor core to sleep: some processors have a SLEEP instruction that puts the processor core into a low power state. Interrupts are handled normally by waking the core.

Swap is not often needed as using a language construct that waits (which implies task swapping) is often a better solution.

Constructs such as Wait, Await, Every, and many messages to objects, such as Keypad.Get will swap tasks if they cannot return immediately.

 See also [Wait](#), [Await](#), [Every](#)

## Tan

**Tan <Float expression> ⇒ Float**

The Tan operator returns the tangent of an angle in radians.

See [operator precedence](#) for which operators are evaluated first.

 See also [Cos](#) [Sin](#) [Acos](#) [Asin](#) [Atan](#)

## Task

The keyword Task may be used to specify the current task, or to specify listing all tasks.

### The current task

**Task**  $\Rightarrow$  <Task object>

For example:

```
Task.State := New TaskState
Print Task.State.Value, CR
```

### Listing tasks

**List Task**

List Task lists out all the tasks currently running, giving some idea of where in the task the current execution point is.

```
Task 0:
in proc_a (myfile.vnm line 20)
in main (myfile.vnm line 40)
in startup (myfile.vnm line 139)
in the command line.

Task 1:
in proc_b (myfile.vnm line 200)
in a task started from main (myfile.vnm line 41).

Task 2:
in serial.Get
in proc_c (myfile.vnm lines 1-2)
in a task started from main (myfile.vnm line 42).
```

You can also type CTRL-T to show all the tasks even when the command line task is busy. CTRL-T is enabled and disabled using [serial.Escape](#), along with the CTRL-C Break function.

As always, you can double click on a file/line number in VenomIDE's Terminal to go to the execution point in your Venom source code.

### Task Blocking

The task list will also show if one task is 'blocked' by another - because of contention over a locked resource. In this case, the listing may look like this:

```
Task 0:
```

```
in proc_a (myfile.vnm line 20)
in main (myfile.vnm line 40)
in startup (myfile.vnm line 139)
in the command line.
```

---


**Task 1:**

**Blocked by Task 0**

```
in proc_b (myfile.vnm line 200)
in a task started from main (myfile.vnm line 41).
```

---

If you see two or more tasks each blocked by another in a circular fashion, then you have a *deadlock* situation. See the Tutorial for more information.

 See also [List](#), [Start](#)

## TextBlock

```
TextBlock <name> :
<lines of text>
<lines of text>
TextBlock End
```

TextBlock allows you to define large blocks of text in your code files. The blocks of text are treated exactly like large string constants.

Text blocks are always defined outside of any other context - that is, they can't be defined inside a procedure or Class definition.

One of the features of TextBlock is that the text you embed is completely literal - you don't need to escape out any characters at all: quotation marks, backslash characters, indentation and line endings are put into the string constant exactly as they appear in your source file. This can be very useful for embedding HTML, or other languages, in your Venom source files. It's also worth noting that you can use a **TextBlock** as the format string of [Printf\(\)](#) and any '%' characters in it will be interpreted in the usual way at run time by [Printf\(\)](#), providing a simple means to embed variable values into an HTML page.

```
TextBlock str:
  This is some text
    It can occupy many lines and be indented in any way
    and can include any number of ""quotes", \backslashes\\, etc.
TextBlock End ; the end of the text.
```

The block of text is introduced with the keyword **TextBlock**, and followed by the (global)

name you want give the text. This text block introduction must end with a colon symbol.  
Then follows the actual text - any characters you like, and any format.

## Termination

The text is ended with the 'termination string' **TextBlock End** (This is case *insensitive* - e.g. you can also use **textblock end**)

There are some other rules about this termination string:

- It must be at the start of the line it occurs on
- It must be followed by some kind of white space, before any other text on the line (you can put any text after the white space, but it will be ignored by the compiler)

## Other termination

In case the text **TextBlock End** can't be used to uniquely indicate the end of the text you can specify any other termination string (inside quotes).

Note that when you supply an explicit termination string it is *case sensitive*.

For example:

```
TextBlock str "+++":
blah blah blah
blah blah blah
TextBlock End ; This is part of the text block
blah blah blah
blah blah blah
+++ ; This is the termination string
```

In order to visually delimit your code you might still want to put **TextBlock End** at the end of your text blocks. The compiler will just ignore it. For example you can do this:

```
TextBlock str "+++":
blah blah blah
blah blah blah
TextBlock End ; This is part of the text block
blah blah blah
blah blah blah
+++ ; This is the actual termination string
TextBlock End ; This is ignored by the compiler
```

## VenomIDE Style tidier

Earlier versions of the style tidier don't recognise the TextBlock construct and will try to tidy up the text inside the block. Later versions stop tidying at the first TextBlock command.

 [Embedded text](#), [String concatenation](#).

## Then

```
If <Int condition> Then
    <statement>
Else
    <statement>
```

Then is used as part of the [If](#) construct to separate the condition and the If's statement.

Then is optional, and in fact is rarely used. It is sometimes useful for improving code clarity.

## This

```
This.<membername>
This ⇒ Any
```

**This** has two related uses or meanings.

Firstly, **This** forces the compiler to access a member/method name, overriding any local variable of the same name, when executing a [method](#) in a [Class](#).

Secondly, **This** holds the value of the current object, which is the current *instance* of a Class. Note that it may be an instance of a class *derived* from the current class.

## Example

```
Class Person : BaseClass
    Age Int
    Name String

    To SetAndCheckAge (age)
        If age < 200
            This.Age := age ; Override local name (1st use)
        End

    To Process
        MyBuffer.Put(This) ; Put this object in a buffer (2nd use)
    End
```



**End**

 See also [Base](#), [Derived](#).

## To

The keyword *To* is used to define procedures and also to [redirect print output](#).

### Procedures

```
To <global name> (<parameter name> , ...)
    <Local list>
    <statement>
...
End
```

<Local list> is as list of the following:

```
Local <local name> := <expression> , ...
```

A procedure is a named sequence of Venom statements.

Procedures may declare [local variables](#). Locals must be declared before any executable statement. Locals may be initialised to any value using `:=`.

Procedures may take parameters. The names of any parameters are specified in parentheses after the procedure name. Parameters behave just like local variables, but they are initialised by the code that calls the procedure. Parameters may be declared as optional using [square brackets](#).

Local variables and parameters will override any global variable with the same name - i.e. the global with the same name is invisible inside the procedure.

The [Return](#) statement can be used within the procedure to make it return a result.

The following example shows a procedure that gives the x to-the-power y (y positive) and shows how it is called:

```
To power(x, y)
    Local result := 1
    Repeat y
        result:=result*x
    Return result
End
-->Print power(10,3) ,CR
1000
-->Print power(2,16) ,CR
65536
```

There are many other examples of procedures throughout this document.

## Print redirection

See [here for print redirection](#)



The maximum number of parameters and local variables combined is 253.

[Print redirection](#)

## True

**True**  $\Rightarrow$  Int

**True** is a convenient constant exactly equivalent to integer 1.



Note that If, While, Await etc all treat any non-zero value as ‘true’.

See also [And](#), [Or](#), [Eor](#), [Inv](#), [False](#)

## Try

```
Try
    <statement1>
Catch <variable name>
    <statement2>
```

The Try/Catch construct is used to handle errors and exceptions in your program.

Try will try running the code in **statement1**. If there are no errors or exceptions then code execution will resume *after* **statement2**.

If there is a runtime error (or [Exit](#)) during execution of **statement1**, *or in any code called by statement1*, control will immediately jump to **statement2**, which is intended to handle the error.

The error will not be handled by the operating system, no error will be reported, and [ErrorAction](#) will not restart the controller.

You must specify a **variable name** after Catch - this variable will be set to the [runtime error code](#) or [Exit value](#) that occurred. The variable that holds the error number can be either a Global or a Local. If no error occurs then the variable is not set.

```
#Define Div_ZERO_ERR 6

To try_divide(a,b)
    Local r := 0
```

```
Local error_code ; variable to hold the error code
Try
[
    r := a Div b ; try the division.
]
Catch error_code
[
    Select Case error_code
    Case Div_ZERO_ERR
    [
        Print "caught div by 0",CR
    ]
    Case Else
        Exit error_code ; pass on other errors.
    ]
Return r
End
```

## Nesting

You can nest Try/Catch constructions to any depth. Exceptions will be handled first by the lowest level Catch. You can pass on any error or exception that you can't handle to the next level up using an Exit command in the Catch block.

## Error value isn't always used

If you don't need to use the value of the variable then use a 'dummy' variable name, e.g. **dummy**:

```
Try
[
    some_code
]
Catch dummy
[
    error_handling
]
```

## Catch is optional

The entire Catch part of the Try construction is optional. If you don't use it then errors and exceptions will simply jump out of the Try block.

```
To try_divide(a,b)
Local r := 0
Try
[
    r := a Div b ; try the division.
```

```

]
Return r
End

```

## User-generated exceptions

Try doesn't just handle normal runtime errors - it also handles *exceptions* that you might choose to generate in your low level code in order to 'get back' to some higher level code that can more easily handle the exception.

You can use Exit to generate your own exceptions. Exit essentially generates a runtime error with an error code you can specify, in the range 1 - 255.

If you use **Exit 0**, then Catch won't catch the exception. This can be used to jump out of nested loops.

If you use a value within the range of [standard Venom2 runtime errors](#) then it appears as if that error happened. For that reason it's usually best for you to choose a range of error/exception codes well outside this range for your own use - we recommend 100 - 255.

 [Exit](#), [Appendix E: Error messages](#)

## TypeOf

**TypeOf** <expression>  $\Rightarrow$  Int

The TypeOf operator returns an integer representing the *type* of the result of the expression, i.e. whether the expression is an integer, floating point number, string, pointer, object, defined [Class](#) etc. In general you don't need to know what these values are – you can just compare them as below:

```

To procedure (parameter)
  If TypeOf parameter = TypeOf 0 ; is it an integer parameter?
  [
    ;process the integer
  ]
End

```

## Object types

As expected, the type numbers returned for objects are different for each object type. However, you may be surprised to find that two objects that may appear to be of the same type will actually have different type numbers. Examples of this are Digital channels on the I2C Bus compared to Digital channels on the controller's on-board I/O ports.

If you want to test whether an expression is an object, then you can use TypeOfNil. The Nil object will always have a type value smaller than any object's type, but larger than any other

type.

```
If TypeOf thing >= TypeOf Nil ; is it an object of some kind?
[
]
```


## Class types

When you define a new class with Class, each class defined has its own unique type number.

You can use this like this:

```
Class my_class
...
End
...
my_object := New my_class
...
If TypeOf my_object = TypeOf my_class
...

```

 Also see the [Is operator](#).

## Constant folding

When it can, the compiler will optimise TypeOf expressions into a simple integer. Currently this applies to

```
TypeOf <integer constant>
TypeOf <float constant>
TypeOf <string constant>
TypeOf Nil
```


Thus, TypeOf <constant> may be used as the Case of a Select Case construct:

```
To procedure (parameter)
  Select Case TypeOf parameter
    Case TypeOf 1 ; Deal with integers
    [
    ]
    Case TypeOf 1.0 ; Deal with floats
    [
    ]
    Case TypeOf "" ; Deal with strings
    [
    ]
    Case Else ; Deal with other types...
    [
    ]
  End
```

## Other types

There is a non-object type that is visible to the Venom2 user but hasn't been mentioned: the Pointer type, created using `@variable_name`.

There are also types that are not visible to the Venom2 user, as they get trapped or converted before TypeOf can get hold of them (e.g. procedures, undefined variables), or because they will never be put in a venom variable.

 See also [As](#)

## Unsigned

### Unsigned

The type specifier **Unsigned** is used to indicate an unsigned integer data type. Currently it may only be used in specifying the data type of 16-bit Arrays.

For example:

```
Array data (Unsigned Int 16)
    1, 2, 3, 4
End
```

Or

```
a := New Array (Unsigned Int 16, 10, 0)
```

## Wait

### Wait <Int expression>

The Wait command is used to wait for a given number of milliseconds. This is used to build in delays between actions. For example, the following piece of code runs a motor for one second:

```
To one_second_run
    motor.On
    Wait 1000
    motor.Off
End
```

If you need to make something execute something at regular intervals, use [Every](#).

Timing over long periods can be done with [Stopwatch](#), [Timer](#) and [RealTimeClock](#) objects.

⚠ The minimum time usefully used by Wait is 0 milliseconds. The maximum time is \$FFFFFFFF milliseconds, or about 25 days.

⌚ While waiting for the time to elapse, the task manager will process other tasks. If there are

no tasks that can proceed the task manager will send the processor to sleep to save power. Wait has a resolution of 1mS. Wait 1 will wait for between 1 and 2 mS.

## Word

### List Word

List Word gives a list of all the global names seen by Venom, under headings giving what they are being used for. The following example shows the user names just after memory has been cleared:

```
Procedures:
startup init main
Integers:


Floats:

Strings:

Pointers:

Objects (inc. 'Nil'):
system serial net led clock s
Undefined:
```

The last category, unused, contains names which have been deleted, or which have been seen by Venom2 but never assigned a value. Unprintable words typed in moments of frustration also turn up here, so beware! Clearing the memory will get rid of these.

 See also [List](#)

## While

**While** <Int expression> <statement>

The While construct executes the statement while the value of the condition expression remains non-zero. If the condition is zero to start with, the statement is never executed. For a loop that executes at least once, see [Do](#).

In this example, the condition is `index0 < 10` and the statement is `Print index0`.

```
-->While index0 < 10 Print index0
      0      1      2      3      4      5      6      7      8      9-->
```

Normally when you use While you would indent the code in the statement, and the statement might well be a block delimited by `[]`.

```
While x <> 15
[
```

```
    process(x)
    deal_with(x)
]
```

Within the loop, [Index0](#) and [Index](#) count from 0 and 1 respectively, incrementing each time round the loop.



See also [Do](#), [Break](#)



Beware that all non-zero values are taken by the While construct as meaning True, but that it is possible for And to return False even when both its operands taken separately would be treated as True.

## Printf

```
Printf(String format, ...)
```

**Printf** is similar to C's *printf()* function. It can be more useful than [Print](#) in some situations - particularly when you need to embed variables within other text.

**Printf** used by itself looks like a command, but actually the **Printf** message is being implicitly sent to the **OperatingSystem** object - see [here](#) for more details.

### Example

```
-->Printf("Hello %i world\n", 5)
Hello 5 world
-->
```



# Object Types

## Object Types

Most of the work in a typical Venom application is done by Objects. Objects come in many different types. Different types of object respond to different messages to perform many different functions.

### How to use the Object Types section

This section of the manual contains detailed information on every object type currently in Venom2 for the VM2. It does not contain any hardware information; for this you should see the datasheet for your controller.

If you are new to Venom2 you are advised to familiarise yourself using the Venom2 Tutorial - this help file is designed for quick reference to enable you to find the exact syntax or function of a particular object type or message.


Each object type section has a header that gives a short description of the type and a summary of the messages available. Some object types have sub-objects - e.g. **Serial**.

#### **InputBuffer.**


After the header for each object type the messages are given in alphabetical order. The exceptions are Creation, which is always shown first - giving information on creating the object – and standard ‘system’ messages such as Printing, Accepting Print, and Die, which are put at the end.


Some messages do very similar things in many objects, or are always accepted because they are used internally by the system and must never give an error. These are only documented separately if they do something out of the ordinary. These ‘generic’ messages are, Lock, Unlock, TestLock, Owner, and Die. They are documented in general in the next section.

Each message has a syntax description, details of what the message does and sometimes some code examples. Following this there may be some paragraphs with icons indicating other information relevant to the object:


 Denotes information on any limits to the parameters or results of the message, or the number of objects of this type that can be created.


 Denotes information on the hardware needed to use this type or message.

 Denotes information on the memory usage of the message, or the memory used when the object is created.

 Denotes information on timing aspects of the message.

 Denotes pointers to other parts of the manual for related topics.

 Denotes information about if and when the object should be locked.

 This warns you to beware of putting bugs in your code

### Syntax descriptions

Each message has a syntax description, including the data types of parameters and results. In

order to be precise, some of these are quite complex. To quickly jog your memory it may be easier to look at one of the code examples. The format of the syntax description definitions is as follows:

**Message** (datatype **item** [, ...])  $\Leftrightarrow$  datatype

- Text in square brackets [ ] indicates an optional part of the construction.
- **bold** is used for message names and user-defined names. (Data types are shown in plain text.)
- Text in <angle brackets> indicates a reference to a 'lower level' construct:
- <name> means any Venom name, and may be qualified by Global or Local
- <statement> means any Venom statement
- <expression> means any Venom expression, and may be qualified by a data type.
- The following in regular text indicate the data type of a value: Int, Float, String, Object, Pointer, Any. Any means that the data type might be any of the other types.
- The symbol  $\Rightarrow$  means the message returns a result. The type is given in 'light' (not bold) text: e.g. Int
- $\Leftrightarrow$  indicates that the message may be used to either set or read a value, i.e. it is an [active variable](#).
- Ellipses ... indicate optional repetition of the preceding construct.

## Locking

There is a set of generic locking messages that are accepted by all objects:

**Lock**

**Unlock**

**TestLock**

**Owner**

Though every object must accept these messages, some objects may not do much in response.

For a description of how to use locking please see the *Venom2 Tutorial*.

The following paragraphs describe the messages and their return results and parameters.

### Lock

In general, calling lock will lock an object so no other task can use it. There are two basic locking schemes that may be used with Venom locks, and you can use both together if you want to.

**Obj . Lock**  $\Rightarrow$  Int

### Incremental locking scheme

Calling Lock with no parameters will attempt to lock the object for exclusive use by the current task:

- If the object is not locked by any task then the current task claims it, and the lock level is set to 1.
- If the object is already locked by the current task then the lock level is incremented.
- If the object is already locked by another task then the current task has to wait until the object is fully unlocked by the task that holds it.

When Lock returns it will return a value that may be useful later: the original lock level, i.e. the number of times the object had been locked originally. This is zero if the object had not been locked at all.

For example:

```
object.Lock ; Increment lock level
...
object.Unlock ; Decrement lock level
```

### Notes

▲ The maximum number of times an object may be locked to is 255. Attempting to lock any further than this will cause a runtime error.

### Restored locking scheme

**Obj . Lock** := Int

You can also Lock an object by assigning a value to the lock. This is used in the Restored Locking scheme described in the Tutorial. Basically, the internal lock level is set to the assigned value. If the value is zero, then the object is unlocked completely.

For example:

```
Local prev_lock_level := object.Lock ; Increment lock level and r
;...
object.Lock := prev_lock_level ; Restore previous lock level.#
```

### Notes

If an object is owned by another task then attempting to unlock it, by assigning a lock level of zero, will be ignored. This can be useful to tidy up in exception handling, when it's not clear if the current task owns an object or not.

If an object has no lock then assigning a value to Lock will do nothing, and Lock will always

return zero.

## Unlock

`Obj . Unlock`

Unlock will reduce the lock level by 1, assuming the object was 'owned' by the current task.

If the lock was not locked, or was owned by another task, then Unlock will result in a runtime error: **Locked/Unlocked too many times**.

## TestLock

`Obj . TestLock ⇒ Int`

TestLock is rather like Lock, except that it won't wait if it can't lock an object immediately. Instead it just returns a zero result to indicate failure.

If it succeeds in locking the object then it returns the resultant lock level. *Note: this is one higher than Lock would have returned, so one should be subtracted when using TestLock in the restored locking scheme.*

*If an object has no lock then TestLock will return the value 1 (one) always.*

## Owner

`Obj . Owner ⇒ TaskObject Or Nil`

Owner will return an object that represents the task that owns the locked object, or Nil if the object is not locked.

*If an object has no lock then Owner will return the value Nil always.*

## Printf

`obj.Printf(<String> format, [Any ... ])`

The **Printf** message is accepted by any object that the **Print To** command can be used on.

**Printf** is more useful than [Print](#) in many situations - particularly when you need to embed variables within other text.

If you just use the **PrintF** message *by itself* then the print output is sent to the *system* object, which then passes it on to the current default [output stream](#).

In operation the **PrintF** message is very similar to the **printf()** function in the C language, but has additions that make it more powerful in Venom2. See here for the [differences](#).

### Format string

The first parameter of **PrintF** is a format string. This should contain the plain text to be printed and optionally a sequence of value place-holders. These take the form of a % character and a format type character (see the table below). For each % place-holder, there should be a corresponding parameter in the **PrintF** parameter list, of a suitable data type.

(The format string doesn't have to be a string constant - it can be any string you like - even created by your program at runtime).

### Format type specifiers

Format type character	Data type of parameter	Examples
b	Binary integer	10011101
c	ASCII Character	A
d, i	Decimal integer	105
e, E	Scientific notation (mantissa/exponent) using e or E character	3.9265e+2
f	Decimal floating point	392.65
g, G	Use the shorter of %e (or %E) or %f	392.65
o	An object	
s	A string (string object or string constant)	Hello world
u	Print a decimal integer as an unsigned quantity	3432
x, X	Unsigned hexadecimal integer - using lower or upper case for letters	7f, 7F
p	A Pointer address in hexadecimal	2000b1c4
I	IP (internet protocol) address as integer	192.168.1.23
%	A % followed by another % character will write %	%

For example

```
-->Printf("Here is an integer %i", 13)  
Here is an integer 13-->
```

## Strings vs Objects

It is possible to print a String (constant, or object) using `%s` or `%o`. Each use has its own benefits.

## Additional flags

In addition to the format type characters, there are optional flags that indicate finer detail in the formatting. These characters always appear between the % and the format type character.

Flag character	Usage	
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).	
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.	
(space)	If no sign is going to be written, a blank space is inserted before the value.	
#	Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written.  Used with g or G the result is the same as with e or E but trailing zeros are not removed.	
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).	

## Fieldwidth and precision

The fieldwidth and precision of the printed item may be specified. The fieldwidth and precision take the form of decimal numbers between the % and the format type character (i, d, x, f, etc), but after other flags. The precision is separated from the fieldwidth by a `.` character.

The fieldwidth is the minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the printed result is larger.

(Note: for `%o` - objects - the fieldwidth is not necessarily used as the width of the print output)

The precision means different things for different format types:

Format type character	Usage
d, i, u, x, X (Integer)	Precision specifies the minimum number of <i>digits</i> to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0.
e, E and f (Float)	Precision is the number of digits to be printed after the decimal point.
g and G (General)	Precision is the maximum number of significant digits to be printed.
s (string)	Precision is the <i>maximum</i> number of characters to be printed. By default all characters are printed until the ending null character is encountered.
o (object)	The fieldwidth and the precision values are passed to the object as the first two format specifiers. See <a href="#">Formatting Objects</a> .
I (IP address)	Precision value is ignored

For example

```
-->Printf("A float: %6.2f",1.06234)
A float  1.06-->
```

### Control characters


To put line breaks in your output use \n in the format string.

See here for other [control characters](#).

For example

```
-->Printf("One line\nfollowed by another\n")
One line
followed by another
-->
```

### Locking

 Printf will lock the object it is sent to for the duration of the message, and then unlock it afterwards. Further locking does not occur in 'nested' printing, where a procedure that includes printing statements (Print, Printf) has been called from a printing statement.

### Differences from C

These are the main differences between the Printf message and **printf()** in C.

**%o** refers to printing objects - not octal, as octal is not commonly used.



`%n` is not supported.

### Additions in Venom2

There are new `%` format characters added by Venom2:

`%b`: print in binary - the parameter must be an integer.

`%o`: print an object - the parameter must be an object. The integer values of any (optional) width and precision values are passed to the object as if they were the [colon format specifiers](#) used in the [Print](#) command.

For example:

```
-->PrintF("The date is %5o and here's a number: %05i\n", clock, 1000000000)
The date is 10-02-09 and here's a number: 00100
-->
```

## Common object properties

This page lists some general properties that are common to some or all objects.

### Die

`Obj . Die`

The Die message will generally attempt to clean up any resources the object may have used, including memory.

If you send a message to a dead object the system will generally throw a runtime error. However, sending the Die message to an already dead object will not result in the runtime error.


Some objects don't use any memory, and some of these objects don't take any action in response to the Die message.

If an object does unusual things in response to Die then Die will be documented along with the other messages.

### Zero-Memory objects

Many objects don't take up any memory, or other resources, when they are created. An example of this is the Digital object. When a Digital is created with Make or New, an *object handle* is returned, which is simply a value. You can create an unlimited number of zero-memory objects. Sometimes it is a useful convenience to do this, for example:

```
To set_a_channel(chan, state)
  New Digital (chan) . Asserted := state
End
```

 Also see [AutoDestruct](#), which is useful for cleaning up temporary objects that do use memory when they are no longer needed.

## AlphaLCD

The AlphaLCD object type can drive alphanumeric displays based on the Hitachi HD44780U LCD driver and compatible devices.

### Summary of messages

**Make**  
**Bitmap**  
**Put**  
**Reset**  
**Value**  
**Print To**

### Creation

```
Make lcd AlphaLCD (Int linelength, Int rows [, Int loc
[, Int wrap]])
```

The first two parameters specify the number of characters per row, the number of rows on the LCD device.

The **location** parameter describes the electronic interface between the LCD and the VM2 - see below.

The **wrap** parameter indicates if you want text to wrap on to the next line on overflowing a line end. The default is *no wrapping* as this works best for most applications. Set this parameter non-zero for wrapping.

### Connecting an LCD

Location refers to the method of connecting the LCD to the controller: the VM2 Parallel Bus, or one of the I2C Buses.

- Parallel Bus: Alphanumeric LCDs will plug directly into the Parallel Bus Alphanumeric

LCD socket on some of our standard Application Boards. To design your own board refer to the circuit diagrams on our website.

- I2C Bus: To connect LCDs via an I2C Bus a PCF8574 (digital I/O port) IC is required to form the interface. The location parameter used can be any one of the equivalent [Digital channel numbers](#) for the PCF8574. A large number of LCDs may be connected to a single controller this way.

On creation, the display device is initialised and the display area cleared. The locations available for connecting an alphanumeric LCD are listed in the table below.

<i>Location</i>	<i>Location numbers</i>
Parallel Bus ( <b>Not available on VM2D</b> )	0 (Default)
First I2C Bus	128, 136, 144, ... , 248
Second I2C Bus	384, 392, 400, ... , 504

```
Make lcd AlphaLCD (20,4)
Make disp AlphaLCD (40,2,160)
```

The first of the above examples creates a 20 char x 4-line display on the Parallel bus, whilst the second creates a 40 char x 2-line display on the first I2C bus.

- ⚠ Multiple AlphaLCD objects may be created via the I2C buses at the various locations, but only one AlphaLCD may be created on the Parallel bus.

The parameter *rows* must be in the range 1-4.

- 🔌 The parallel bus is not brought out to pins on the VM2D.

## Bitmap

**Bitmap (Int address)**

This allows eight user-defined characters in the LCD controller IC to be defined. Each character is defined by eight bytes. Only the five least significant bits are used. *address* is the address in memory of 64 bytes of data defining all eight characters. The first eight bytes are for character 0, and the next eight for character 1 and so on.

The following example uses an array of data in binary format that defines an 'F'-shaped character, so you can see the relationship between the bits in the definition and the pixels on the LCD. Up to seven more characters could be defined.


```
Array user_chars (8,64)
  %11111, ;an 'F' character with an underline
  %10000,
  %10000,
```

```

    %11110,
    %10000,
    %10000,
    %00000,
    %11111,
    $FF      ;[Fill the rest of the array with $FF for now]
End

Lcd . Bitmap(user_chars . Pointer) ; define the characters
Lcd . Put (0) ; sends character 0 to the LCD.

```

 See also [Array](#)



Using character number zero may be difficult in some circumstances, e.g. when printing to a string or text buffer, which then prints to an LCD. The other characters don't have this problem.

## Put

**Put (Int)**

This message displays a character on the AlphaLCD.

The character could be an ASCII value or may have been defined using [AlphaLCD.Sprite](#).



Note: if using the AlphaLCD object in more than one task, it should be locked before sending the Put messages, and unlocked afterwards.

```

-->lcd.lock
-->Repeat 5 lcd.put(65 + index0)
-->lcd.unlock

```

## Reset

**Reset**

This message re-initialises the controller IC and refreshes the text on the display. Should you wish to clear the display use:

```

-->Print To lcd, CLS

```



See also [Accepting Print](#).

## Value

```
Value(int reg, int byte)
```

The value message allows you to write data directly to the LCD's command and data registers.

- If reg is 0 then byte is written to the command register.
- If reg is 1 then byte is written to the data register.

This command allows you to set the cursor and other functions not otherwise supported by the AlphaLCD object.

See the HD44780U, or other LCD driver, datasheet for the list of commands.

## Accepting Print

```
Print To lcd, <print list>
```

 *Note: you can also print to an LCD by sending it the [PrintF](#) message.*

Printing to an AlphaLCD object displays text on the LCD.

The object also understands the following printing keywords:

### CR

Go to the beginning of the next line, scroll up a line if necessary.

### Home

Go to the top left of the display.

### CLS

Clear the display and go to the top left.

### GOTOXY

Put the cursor at the given character position. The top left position is (0,0). The coordinates are clipped to the size of the display.

```
-->Print To lcd , "Hello" , GOTOXY (10 , 1) , "World"
```

 See the [wrap](#) parameter.

## Analogue

The Analogue object type allows read and write access to analogue I/O.

Devices currently supported are:

- VM2 on-board channels: 14 12-bit inputs, two of which may also be configured as 12-bit outputs.
- AD7998 and similar devices - on the I2C Bus.
- MAX1236/1238 devices - on the I2C Bus
- PCF8591 devices - on the I2C Bus

## Audio output

Channel \$14 may be used to generate audio output signals - e.g. to play WAV files. See [Send](#).

## Summary of Messages

**Make**  
**Period**  
**Send**  
**Value**  
**Print**

## Creation

There are several on-board and off-board analogue drivers available.

- [Onboard Analogue](#) I/O (12 bit, many inputs, 2 outputs)
- [AD7998 and variants](#) (I2C Bus, 12 bit, several inputs)
- [MAX1236/1238](#) (I2C Bus, 12 bit, several inputs)
- [PCF8591](#) (I2C Bus, 8-bit, 4 inputs, 1 output)

Other drivers may be written in Venom code, or added to Venom on request. Please contact us to find out more about this.

### On-board

#### Onboard Analogue input


**Make obj Analogue (Int channel)**

This creates an analogue input object that can read one of the analogue input channels on the VM2.

Channels in the ranges \$10 - \$17 and \$30 - \$35 are currently supported as 12-bit analogue inputs.

For example:

```
--> Make voltage_reading Analogue($30) ;an analogue input on the
```

 VM2 Channels that can be 12-bit analogue inputs are in the ranges \$10-\$17, \$30-\$35 and \$66-\$6A.

*Channels \$66-\$6A are not available on the VM2D.*

You can adjust the analogue input to match the source impedance of the voltage being read using [Period](#).

 Analogue is a [Zero-Memory](#) object

## Onboard Analogue output


```
Make obj Analogue (Int channel, 1 [, Int no_buf])
```

This creates an analogue output object that can drive one of the analogue output channels on the VM2. Channels \$14 and \$15 may be used for 12-bit analogue output.

For example:

```
--> Make control_voltage Analogue($14, 1) ;an analogue output on
```

By default there is a buffer amplifier put in the circuit which allows the output to drive relatively low impedance loads, but limits the output range to less than the supply rails. The optional parameter **no\_buf**, when supplied and non-zero, *disables* the DAC's internal output buffer amplifier. This allows the output to reach the supply rails but reduces the output drive considerably. See the STM32F103 datasheet for more details.

 VM2 Channels that can be 12-bit analogue outputs are \$14 and \$15.

 Analogue is a [Zero-Memory](#) object

AD7998, etc

### AD7998 and variants

*Variants include: AD7997, and the -0 and -1 addressing options.*

```
Make obj Analogue (0, Int bus, Int I2C_add [, Int
channel])
```

**bus** should be 1 or 2 for which I2C Bus you wish to use.

**I2C\_add** should be in the range 64 - 68. **Note: this clashes with the PCF8574 address range.**

**channel** is optional (it defaults to 0), or should be in the range 0-7.

For example, an object to read channel 2:

```
Make a Analogue(0,1,64,2)
Print a.Value
```

### Read any channel

If you make an analogue object like this on channel 0, you can later select which channel you wish to read when you call the [Value](#) message. This can sometimes be more convenient than defining the particular channel in the Make. Here is an object that will read channel 0 by default, but will read any channel if you so specify.

```
Make a Analogue(0,1,64) ;Set up to read channel 0
Print a.Value           ;Read channel 0
Print a.Value(2)        ;Ask to read channel 2
```

### Other options

There are many other options available on the AD7998 than are accessible through this driver. You can use the [I2C Bus object](#) to access them.

For your information, the Make command does not communicate with the device at all, and the Value message uses the device in Command Mode (Mode 2), and does a simple read of the selected channel.

 Analogue is a [Zero-Memory](#) object

MAX1236, etc

### MAX1236/1238

```
Make obj Analogue (Int type, Int bus , Int channel [,
Int setup])
```

**type** should be 4 for MAX1236 and 5 for MAX1238



**bus** should be 1 or 2 for which I2C Bus you wish to use.

**setup** is an optional parameter - it defines the setup register value sent to the device each time it is read. If **setup** is not supplied, a default value is used. See below.

**channel** should be in the range 0-3 for the 1236 and 0-11 for the 1238.

For example, an object to read channel 2:

```
Make a Analogue(4,1,2)
Print a.Value
```

### Read any channel

If you make an analogue object like this on channel 0, you can later select which channel you wish to read when you call the [Value](#) message. This can sometimes be more convenient than defining the particular channel in the Make. Here is an object that will read channel 0 by default, but will read any channel if you so specify.

```
Make a Analogue(4,1,0) ;Set up to read channel 0
Print a.Value          ;Read channel 0
Print a.Value(2)       ;Ask to read channel 2
```

### Other options

There are many other options available on the MAX1236/8 than are accessible through this driver. You can use multiple objects with different setup register values, or you can access the IC directly using the [I2C Bus object](#).

The setup register value is optionally selectable during Make (default value \$D2), however the configuration register value is fixed at \$61, plus the channel selection bits. See the device datasheet for the meaning of the bits in these registers.

For your information, the Make command does not communicate with the device at all, and the Value message sends the setup register value followed by the configuration register value, and then does a simple read of the selected channel.

 Analogue is a [Zero-Memory](#) object

## PCF8591

### PCF8591

```
Make obj Analogue (2, Int bus, Int I2C_add [, Int
channel])
```

**bus** should be 1 or 2 for which I2C Bus you wish to use.

**I2C\_add** should be in the range 144 - 158. (It is  $144 + A2*8 + A1*4 + A0*2$ , where A0-A2 are 0 or 1 depending on which state the address input pins on the device are set to).

**channel** is optional (it defaults to 0), or should be in the range 0-3.

For example, an object to read channel 2 of a device at I2C address 144:

```
Make a Analogue(2,1,144,2)
Print a.Value
```

### Read any channel

If you make an analogue object like this on channel 0, you can later select which channel you wish to read when you call the [Value](#) message. This can sometimes be more convenient than defining the particular channel in the Make. Here is an object that will read channel 0 by default, but will read any channel if you so specify.

```
Make a Analogue(2,1,144) ;Set up to read channel 0
Print a.Value           ;Read channel 0
Print a.Value(2)        ;Ask to read channel 2
```

### Set the output

To set the value of the single analogue output, create an object without a channel number, and set it's value:

```
Make a Analogue(2,1,144)
a.Value := 128
```

### Other options

There are many other options available on the PCF8591 than are accessible through this driver. You can use the [I2C Bus object](#) to access them.

 Analogue is a [Zero-Memory](#) object

### Period

```
Period (Int period)
```

Period allows the sample period of the *on-board ADC* reading to be set - which allows different source impedances to be accommodated. Higher source impedances need a longer sampling period to achieve the same accuracy.

The default value of period is 0 - the fastest - with the lowest source impedance requirement.

When the VM2 is running at 72MHz, the ADC is programmed to run at 12MHz (Main clock divided by 6). The table gives the maximum source impedance for a 0.25lsb error due to source impedance for each speed setting, and also the number of cycles taken to read an input.

period	ADC clock Cycles	Max Source Impedance (K Ohms)
0	1.5	1.2
1	7.5	10
2	13.5	19
3	28.5	41
4	41.5	60
5	55.5	80
6	71.5	104
7	239.5	350

⌚ The overall time taken to return a result is longer than the ADC clock cycles of the sample time, and includes the overhead of sending the message and controlling the measurement process.

## Queue

**Queue**  $\Rightarrow$  Int

Queue returns the number of audio output samples remaining to be sent out when [Send](#) is called.

## Example

```
audio.Send("mysound.wav")
Await audio.Queue IsFalse ; wait for audio output to end.
```

## Send

**Send**(String filepath)

**Send**(Int address, Int size)

The Analogue Send message can be used to send audio to a DAC output on the VM2. Currently this only works for channel \$14.

The Send message has two forms: one where it plays audio files from the Flash Filing System, and one where it takes an address of the data, anywhere in memory, and the size of the data, as two integers.

## Playing audio files

### Example

```
Make audio Analogue($14,1)
audio.Send("mysound.wav") ; Play file in root directory
audio.Send("mydir/mysound.wav"); Play file in sub directory
```

### Notes

Audio files must occupy **contiguous blocks** in the Flash Filing System memory, which entails loading them into the filing system in a [particular way](#).

The file formats supported are 'WAV' with mono, 8-bit unsigned PCM data. The sample rate has been tested up to 44KHz. If the file does not contain WAV header information that matches these parameters then the file will not play.

(Note: You can also play 16-bit WAV files if the data chunk of the file has been converted from *signed* to *unsigned*. The playback is actually at 12 bits due to the resolution of the VM2's DAC, but still the higher sample resolution reduces the noise heard on quieter sounds. Please contact us if you would like to try this option.)

You can convert other audio file formats into WAV formats using *Audacity*, which is available free on the Internet. We have been using version 2.0.5.

Before you save a file in the new format, set the bit rate you need in the **Project Rate** box (bottom left in the Audacity window). 11,025 Hz seems fine for most purposes.

To save a file in the correct format for the VM2 you should use Audacity's **File > Export ...** menu. You will need to select **Save as type**: 'Other uncompressed files', and using the **Options...** button set **Header** to 'WAV (Microsoft)' and **Encoding** to 'Unsigned 8 bit PCM'.



Note: You can't play wav files that are in the Flash File System at the same time as writing to or erasing files in the Flash File System. There are several different approaches to working around this. See [below](#).



The [Queue](#) message returns the number of samples remaining to be sent out. The audio output has finished when Queue is zero.

## Playing audio data direct from memory

Currently the only audio data format supported is 8-bit unsigned PCM data at 11,025Hz sample rate.

You should supply the address of the data in memory and the length of the data in bytes.

```
audio.Send(addr,len) ; Play audio data in memory
```

## Work-arounds: Playing audio and writing/erasing flash files concurrently

The basic problem is that audio data is accessed from the flash by an interrupt, and that this interrupt can run when the flash memory is in the middle of a program or erase operation. When the flash is in program or erase mode, data cannot be read from it.

There are three basic work-arounds.

1. Lock the flash file system while the sound plays
2. Play the sound from RAM instead of flash
3. Use the Flash file system for audio and other read-only file operations (bitmaps, fonts, ...), and use a different file system (RAM, SD Card, USB host) for any other filing requirements.

### 1. Locking the file system

Example code to play sounds from the flash file system only while it is locked:

This version waits for the file system to become unlocked before playing the sound. You may have to wait a long time if the file data being written is large.

```
ffs.Lock ; wait for file system to become free.
audio.Send("mysound.wav") ; Play file in root directory
await audio.Queue IsFalse
ffs.Unlock
```

Here a 'non-blocking' solution. If the file system is locked then playing the sound is ignored. Alternative action may be taken instead.

```
If ffs.TestLock
[
    audio.Send("mysound.wav") ; Play file in root directory
    await audio.Queue IsFalse
    ffs.Unlock
]
Else
[
    ;alternative action, if sound could not be played.
]
```

### 2. Playing the sound from RAM

For this you will need to copy the audio data from the flash file into an array (or RAM file) and pass the address of the data to Send.

### 3. Use the Flash file system only for read-only file operations

If you never need to write to files, or if you can use another file system for your writable files, you can play audio directly from Flash files.

Many applications will use the Flash File System as a read-only system - reading fonts and bitmaps, for example. You can ensure that the Flash File System is never written to by [creating it](#) with a cache size of zero in your **init** procedure.

It may also be possible to use RAM, SD Card, or USB Host file systems if you need to write files.


## Value

**Value**  $\Leftrightarrow$  Int

This [active variable](#) allows analogue values to be read from Analogue input objects, and analogue values to be written to Analogue output objects.

The value is not the voltage, but the 'step number' in the analogue range.

Note: the output impedance of the DACs is quite high and may need buffering externally to the VM2 - check the DAC drive capability in the device datasheet.

 See [Period](#) to adjust on-board analogue input to match the source impedance of the voltage being read

## Printing

**Print** <Analogue>

Prints the the analogue object in 'debug' format.

For example:

```
-->Print a  
[Analogue: 317]
```

Here the number is the value of the input or output.

## Array

**Array** is an object intended for storing a fixed amount of data. For example, Arrays may be used to store a look-up table for linearising a sensor, store a set of strings for implementing a generalised menu system, or hold a table of procedure pointers.

Arrays may be *constant* - that is their contents are defined at compile-time, they naturally reside in *Read Only Memory* (ROM), and their contents are fixed once you have written your program - just like procedures. Constant arrays are created in a special way: [Creating constant arrays](#).

Arrays may also be *variable* - that is they are created during program execution (using **Make**, **New** or **.Copy**), they naturally reside in *Read/Write Memory* (RAM), and their contents may be changed at any time after creation. See [Make](#) for how to create them.

ARRAYs can hold data of the following types:

- 8, 16 or 32-bit integers
- Floating point numbers
- Pointers (to global variables)
- Strings (variable or constant)

Each array holds only data of a single type.

*If you need an object that can hold any type of data, use [Buffer](#) of Any.*

## Summary of messages

**Make**  
**Address**  
**Copy**  
**Die**  
**Element**  
**Find**  
**Length**  
**Sort**  
**Print**

## Creation

Here we will look at creating *variable* arrays. See here for creating [constant arrays](#)

**Make obj Array (type , Int n , ... )**


**type** indicates the *type* of data that the array will hold:

Type	Element data type	Range of values element can hold
<b>Int 8</b>	8-bit integer (unsigned)	0 to 255
<b>Int 16</b>	16-bit integer (signed)	-32768 to +32767
<b>Unsigned Int 16</b>	16-bit integer (unsigned)	0 to 65535
<b>Int</b> or <b>Int 32</b>	32-bit integer (signed)	-2,147,483,648 to 2,147,483,647
<b>Float</b>	Floating point (IEEE single precision)	$\sim \pm 1.0E\pm 38$ , $\sim 7$ digits of precision
<b>String</b>	String constant	Any <a href="#">string</a> constant or <a href="#">String</a> object
<b>@dummy*</b>	Pointer to a global	Any pointer to a global variable

\* or any pointer to a global variable.

**n** is the number of elements in the array.

The ... indicate that you can put some initialising data in the parameter list. If there are fewer initialisers than elements, then the value of the last initialiser is used to fill the array. If no initialisers are present, then the array is not initialised: the data may be any random set. Take care not to put too many initialisers in as you may run out of space on the [Venom-SC stack](#). A sensible limit is 10 or so. If you need more initialisers than this use a [constant Array](#) and take a [copy](#) of it.

 Arrays use a single, contiguous block of RAM that is large enough to hold all their data, plus 18 bytes.



## Example code

Some simple arrays are created below:


```
Make b Array(Int, 25, 1, 0) ; Array of 25 32-bit integers, start
Make a Array(Int 8, 100, 0) ; Array of 100 8-bit integers, all in
Make c Array(Float, 10, 0.0) ; Array of 10 floating point numbers
```

## Using variable Arrays of strings

The following bit of code illustrates the use of a variable array of strings. Here we are using both String objects and string constants in the array - it can handle both. See if you can work out what is going on here; you may need to learn about String objects, array initialisers (above), and the Element message.

```
-->Make sa Array (String,5,"X") ;New array - each element is init
-->Repeat 3 [sa.(Index0) := New String(10)] ; 3 new 10-character
-->sa.(0).Put("Hello") ; Two of the new string objects have text
-->sa.(1).Put("Goodbye")
-->Print sa ; print out what's in there:
Hello
Goodbye

X
X
-->
```

 Please see the [Die](#) message for how arrays of strings are handled.

## Address

**Address**  $\Rightarrow$  Int

Address returns the memory address of start of data in the array.



Be very careful if you write to a raw memory address as it's possible to corrupt the system heap if you make any mistakes.

Data is arranged contiguously: elements are in order with no gaps. The address will always be aligned to the number of bytes required by the largest object used by the host processor. In the case of the VM2 it will be aligned to an even address.

Notes:

Constant arrays of strings are arranged in memory as a list of pointers, followed by all the string

data.

## Copy

**Copy**  $\Rightarrow$  Array

Copy returns a new, writable copy of an array.

For example:

```
variable_data := lookup_data.Copy
```



## String arrays

When you copy an **Array of Strings** then **Copy** doesn't make copies of the strings held by the original array, instead it makes copies of the *pointers* to the *same strings*.

## Die

**Die**  
**Die**(Int leave\_contents)

The **Die** message removes a variable Array from memory. However, an *Array of Strings* could 'contain' (point to) String objects. **Die**, by default also removes these String objects from memory.

This is an example of the default behaviour:

```
Make Array_of_strings Array(String,5)
str := New String(10)
Print To str, "Hello"
Array_of_strings.(0) := (str) ; Put the String in the Array.
...
Array_of_strings.Die ; Removes the Array and the String.
```

If for some reason you just want to remove the Array of Strings, and not any String objects it points to then you can pass a non-zero parameter to **Die**:

```
Array_of_strings.Die(1) ; Removes the Array but not the string it
```

Note that [AutoDestruct](#) will use the default behaviour, and so remove all String objects pointed to by the Array.

## Element

```
Element (Int num) ⇔ Any  
<object>.(Int num) ⇔ Any
```

Individual elements in an array are accessed using the Element message. The shortcut syntax, missing out the Element message name, is allowed.

The parameter **num** specifies the element number.

Elements will be checked for the correct type when they are written. For integer arrays, the written values will be truncated to the correct size by removing higher-order bytes.

## Examples

```
Array phrases (String,2)  
  "Hello"  
  "Goodbye"  
End  
  
-->Print phrases . Element (1)  
Goodbye-->
```

Venom abbreviation allows **.()** to replace **.Element()**, so you could also use:

```
-->Print phrases.()  
Hello-->
```

And...

```
-->Make data_list Array (8, 10, 0)  
-->Print data_list.()  
  0-->  
-->data_list.() := 10  
-->Print data_list.()  
  10-->
```

## Find

```
Find(target [, [Int startpos [, Int flags]]) ⇒ Int
```

Return Value	The first array index where the target was found, or -1 if not found
<b>target</b>	The value to find, which must be an integer, float or string and must match the array data type
<b>startpos</b>	Where to start the search. Default value of <b>nil</b> or -1 means search the whole array (i.e. 0 for forward search, (array length) - 1 for reverse search)
<b>flags</b>	This is a bitmapped combination of values. <div> <div>Bit 0</div> <div>(Linear) Search in reverse order</div> <div>(Binary) Array is sorted in descending order</div> <div>Bit 1</div> <div>Ignore case in strings</div> <div>Bit 2</div> <div>Use a faster binary chop search algorithm</div> </div>

Note that when using the binary chop search on an array of strings, the case sensitivity and direction settings for the search must match those used when the sorting was done.

For both linear and binary search, if there is more than one element matching the target value, the index returned is the lowest matching index for a forward search or the highest matching index for a reverse search.

## Length

**Length** ⇒ Int

Length returns the number of elements in the array.

## Sort

**Sort**([Int **sort\_type**])

Sorts an array, by default in ascending order and (if an array of strings) case-sensitively.

**sort\_type** Bit-mapped flags controlling the type of sort. Default value = 0

Bit 0 (value 1) sets descending order

Bit 1 (value 2) sets a case-insensitive sort for strings

**Sort** can be used for arrays of strings and all integer data types, and distinguishes correctly between signed and unsigned integer data types.

## PRINT

**Print** <Array>

Printing the array will print out each of the elements of the array in order, each element on a new line.

For example

```
-->Print a
      1
      2
      5
      7
-->
```

### Formatting print output

**Print** <Array>:**format**

If colon formatting is used, the specified format is applied to each element as it is printed.

```
-->Print a:1
      1
      2
      5
      7
-->
```

## Buffer

Buffers can be used to store lists of data: integers, floating-point values, text or even other objects. Buffers dynamically allocate as much memory as they need to store the data they hold, plus an overhead. They may be used to make FIFO (queue-like) and FILO (stack-like) structures.

### Text Buffers

Text Buffers (buffers containing textual information) may be used for string or text operations in Venom2. See also [String objects](#).

### Buffer of Any

There is a special kind of Buffer known as 'Buffer of Any' that can hold lists of any kind of entity, including lists of other buffers, numbers, arrays, strings, etc.

It is easy to convert a block of text containing many lines into a Buffer of Any containing a set of String objects, each String holding one line of the text. See [Accepting Print](#).

## Summary of messages

**Make**  
**Die**  
**Element**  
**Empty**  
**Find**  
**Flush**  
**Get**  
**GetLast**  
**Insert**  
**Length**  
**Put**  
**Queue**  
**ReadPoint**  
**Remove**  
**Reset**  
**Sort**  
**Value**  
**Print**  
**Print To**

## Creation

**Make** <object> **Buffer** (*type*)

The type parameter specifies the type of data to be stored in the buffer; it must be one of those shown in the following table:


type	Element data type
<b>Int 8</b>	8-bit unsigned integer
<b>Int 16</b>	16-bit signed integer
<b>Int 32</b>	32-bit signed integer
<b>Float</b>	Floating point
<b>Char</b>	Character (i.e. a text buffer)
<b>Any</b>	Any type

The following example creates a buffer with 8-bit integer elements:

```
-->Make buffer_object Buffer (Int 8)
```

And this line creates a text buffer, i.e. a buffer with character elements:

```
-->Make text_buffer Buffer(Char)
```

 Buffers use a minimum of 316 bytes of memory each. Each buffer has a header block of 54 bytes, and one or more data blocks of 262 bytes. Each data block can hold 256 bytes of data.

Buffers of integers hold data in 8-, 16- or 32-bit elements. Buffers of floats use 32-bit elements. Buffers of Any use 64-bit elements (8 bytes).

## Buffer of Any

A buffer that can hold any kind of entity is called a Buffer of Any:

```
-->Make buffOfAny Buffer(Any)
```

You can use this kind of buffer to hold lists of objects, including lists of Buffers, Arrays or Strings, for example.

### Printing to a Buffer of Any

One of the most common things a Buffer of Any is used for is to hold lists of String objects.

If you [print text to a Buffer of Any](#), then the text will be 'Put' into the Buffer of Any as a set of String objects, each String holding just one line of the text.

So to create a list of all the files in a file system, where each file name is held by a String object, and all the strings are held by a Buffer of Any you only have to do this:

```
Make listing Buffer(Any)  
Print To listing, ffs
```

After you have finished with the listing you can send `.Die` to it, and this will remove all the String objects it contains. Or you may be able to use [AutoDestruct](#).

## Buffer of Any Tutorial

Because Buffers of Any can hold objects, you can use them to do things that would otherwise take a lot of coding.

For example to send a Value message to an object held in the nth element of a buffer of any, you can do this:

```
buffOfAny.(n).Value
```

You can find the data type of something held in a buffer of any using the `TypeOf` operator:

```
Print TypeOf(buffOfAny.(n))
```

#### Loading a Buffer of Any with new objects

Often when using a Buffer of Any you will wish to load it with a set of new objects, for example a set of [String](#) objects.

This code loads the buffer with 10 new strings, each with a capacity of 20 characters.

```
Repeat 10
    buffOfAny.Put(New String(20))
```

#### Removing the Buffer and it's sub-objects

If at a later time you wish to remove the buffer and the strings it holds you can use this:

```
buffOfAny.Die
```

**Die** removes the buffer *and any sub-objects too*.

(There is a [version of Die](#) that doesn't remove the sub-objects, in case that is the behaviour you need.)

#### Memory leak examples

The following are all examples of how *not* to use the Buffer of Any, in that they will result in a [memory leak](#).

```
buffOfAny.(5) := New String(20)
...; some other processing
buffOfAny.(5) := <anything else>
```

In the code above, the element (5) of the buffer was overwritten with a new value, but the original string wasn't removed, so the string is lost, together with it's memory.

```
Repeat 10
    buffOfAny.(Index0) := New String(20)
buffOfAny.Empty
```

This time all the strings in the buffer were lost.

You can check for memory leaks in your code by using the [Garbage Scanner](#).



## Die

### Die

The Die message removes the Buffer from memory.

### Garbage collection in Buffer of Any

A **Buffer of Any** could 'contain' (i.e. hold references to) other objects. By default Die also removes all the contained objects from memory.

This is an example of the default behaviour:

```
Make buffOfAny Buffer(Any)
str := New String(10)
Print To str, "Hello"
buffOfAny.Put(str)
...
buffOfAny.Die ; Removes the buffer and the string.
```

### Shallow removal

If you just want to remove the **Buffer of Any**, but not remove any sub-objects it contains then you can pass a non-zero integer parameter to Die.

```
Die (Int shallow)
```

For example:

```
...
buffOfAny.Die(1) ; Removes the buffer but not the string.
```

Note that [AutoDestruct](#) sends Die with no parameter and thus removes all sub-objects.

## Element

**Element**(Int **item\_number**)  $\Leftrightarrow$  Any

<Buffer>.(Int **item\_number**)  $\Leftrightarrow$  Any

The **Element**() [active variable](#) allows both read and write random access to the contents of a buffer as if it were an array. Venom abbreviated syntax allows the use of **.()** to substitute for **.Element()**.

## Empty

**Empty**

Empty throws away any data held in a buffer, freeing up all but the minimum memory used by the Buffer.

However, a **Buffer of Any** could 'contain' (point to) 'sub-objects'. By default Empty also removes sub-objects from memory.

This is an example of the default behaviour:


```
Make buff_of_any Buffer(Any)
str := New String(10)
Print To str, "Hello"
buff_of_any.Put(str)
...
buff_of_any.Empty ; Empties the buffer and removes the string: St
```

## Garbage collection in Buffer of Any

**obj . Empty**(Int **shallow**)

If for some reason you just want to remove the **Buffer of Any**, but not remove any sub-objects it contains then you can pass a non-zero parameter to Empty:

```
Make buff_of_any Buffer(Any)
Make str String(10)
Print To str, "Hello"
buff_of_any.Put(str)
...
buff_of_any.Empty(1) ; Empties the buffer but leaves the string a
```

 See also [Flush](#).

## Find

### Text Buffers

```
Find (String str [, Int start_pos]) ⇒ Int
Find (Buffer buf [, Int start_pos]) ⇒ Int
```

Find searches a text buffer for the search string specified by either a string (**str**) or another text buffer (**buf**) starting at the optional element specified by **start\_pos** (or from the beginning of the buffer if no second parameter is supplied). If a match is found, the element position at which the specified text begins is returned, otherwise the value -1 is returned. The search string (**str**, or **buf**) is limited to 256 characters in length.

The following example finds the start position of the first occurrence of a string in the buffer:

```
Make b buffer(Char)
b.put("text text text")
pos := b . Find ("tex", 3) ; pos will be 5
```

The example below shows the Find message being used to locate the text contained in buffer b2 within buffer b:

```
Make b2 buffer(Char)
b2.Put("xt")
pos := b . Find (b2) ; pos will be 2
```

### Integer Buffers

```
Find (Int value [, Int start_pos]) ⇒ Int
```

Find searches an integer buffer for a particular value, starting from beginning, or an optional start position.

It returns the position the value was found at, or -1 if it wasn't found.

### Buffer Of Any

```
Find (any value [, start_pos [, int flags [, any xflags]]) ⇒ Int
```

For a buffer of any where the elements are all objects that have a suitable **Compare** method, this will find an element that "matches" the given value.

The search is much more sophisticated than those described above, with options for forward or reverse search, and also a "binary chop" search on sorted data, which is much faster than a linear search on all but the smallest buffer sizes.

Returned value is index of found element, or -1 if none found.

The objects can be strings or objects defined in the Venom program using the [Class](#) keyword.

<b>valu</b>	Any value compatible with the object's Compare method
<b>star</b>	Where to start searching. Forward searches are from this point to end of buffer, reverse search from this point to beginning (element 0)  If not specified, <b>nil</b> or -1, use default of 0 for forward search or last element of buffer for reverse.
<b>flag</b>	Bitmapped flags controlling type of search  Bit 0 = reverse search  Bit 1 = case insensitive search for strings (passed as 0 or 1 in second parameter to Compare method)  Bit 2 = use binary chop search algorithm on buffer that is in sorted order.  <b>Sort</b> direction and case sensitivity settings must match those used in <b>Find</b> .
<b>xfla</b>	Optional third parameter passed unchanged to the object's Compare method if it is supplied here.

#### Specification of Compare Method

**Compare**(value [, int ignore\_case [, xflags]]) ⇒ Int

Returns an integer which is:

- negative if the object is "lower" than the supplied value
- positive if the object is "higher" than the supplied value
- 0 if the object "matches" the supplied value

<b>value</b>	The value to compare with some element or property of the object
<b>ignore_c</b>	For string values, non-zero for a case-insensitive match (default or 0: case sensitive) based on bit 1 of the <b>flags</b> parameter passed to the <b>Find</b> message. The Compare method must accept this parameter, though it may not always be supplied.
<b>xflags</b>	Optional extra flags, number or other variable to specify how the comparison is done, e.g. you may want to select which of an object's member values to compare with the search target. The Compare method doesn't have to support this unless the <b>Find</b> is used with explicit fourth parameter.

Note that string objects comply with this specification, though they don't use the **xflags** parameter.

#### Type of First Parameter of Compare

If the Buffer is going to be sorted, the **Compare** method must accept an instance of the object

type as its first parameter for the sort message. For use by the buffer **Find** method, the first parameter is likely to be a string or numeric value. For both **Sort** and **Find** to work, the Compare method must therefore distinguish between the two data types of the first parameter. For example:

```

Class myclass
    strval string

    To Compare(x, [ignorecase])
        if Typeof(x) = Typeof(This)
            Return strval.Compare(x.strval, ignorecase)
        else
            Return strval.Compare(x, ignorecase)
        End
    End

    ;....
End ; Class

```

**Example of use of xflags**

**xflags** is used to select whether to compare with **strval1** or **strval2**

```

Class myclass
    strval1 string
    strval2 string

    To Compare(x, [ignorecase, xflags])
        If Typeof(x) = Typeof(This)
            If (xflags)
                Return strval1.Compare(x.strval1, ignorecase)
            Else
                Return strval2.Compare(x.strval, ignorecase)
            End
        Else
            if (xflags)
                Return strval1.Compare(x, ignorecase)
            Else
                Return strval2.Compare(x, ignorecase)
            End
        End
    End

    ;....
End ; Class

```

## Flush

### Flush

**Flush** removes the data items that are behind the current **ReadPoint**. If enough data items are removed, the memory they occupied is released back to the system.

The **ReadPoint** is updated when data is read from the buffer using **Get**, or by directly writing to the buffer's **ReadPoint**. It is also updated by messages such as **Flush** and **Empty**.

### Garbage collection in Buffer of Any



Note: Currently there is no garbage collection for Buffers of Any when **Flush** is called, unlike for [Die](#) and [Empty](#).



See also [Empty](#), [Get](#), [ReadPoint](#).

## Get

**Get** ⇒ Any

Get returns the data item from the buffer at the current readpoint and then advances the readpoint.



Attempting to read past the end of data will result in an 'Array index out of range' error.



See also [Put](#), [ReadPoint](#), [Flush](#)

## GetLast

**GetLast** ⇒ Any

GetLast removes one data item from the end of the buffer and returns it. It may be used to delete characters from the end of a text buffer, or to implement a FILO buffer, or 'stack': [Put](#) is used to push data on to the stack, and GetLast removes it. [Length](#) may be used to check how much data is held in the Buffer.

## Insert


```
Insert (Int character, Int pos)  
Insert (String str, Int pos)  
Insert (Buffer buf, Int pos)
```

This message is only available for buffers containing text. It allows text in the form of a single character, string or text buffer to be inserted into the buffer at the element position specified by **pos**. The following example inserts a string into a buffer:

```
Make b Buffer("text text text")  
b.Insert("XXX" , 3)  
PRINT b
```

... prints:

```
texXXt text text
```

 See also [Remove](#)

## Length

```
Length ⇒ Int
```

Length returns the number of data items currently held in the buffer.

 See also [Queue](#), [Put](#).

## Put

```
Put (Any item)
```

Put adds a data item to the end of the buffer.

```
Repeat 5 buff.put(index0)
```

The data item has to be of the correct type for the buffer to accept, else a [Type Mismatch error](#) is thrown.

Large integer values, that would overflow 8 or 16-bit integer buffer elements, are truncated.

**Put a block**


```
Put (string/array data , Int start, Int size)
```


This form of **Put** efficiently appends a block of consecutive elements from an integer or float array to a buffer of identical element type, or from a string to a text buffer.

If the optional parameters **start** and **size** are not given, the whole array or string is copied.

```
Make b buffer(Char)
b.Put("Hello")
```

This form of **Put** is faster than repeated single element **Put** message, but for integers the element types must be the same size (8, 16 or 32 bit).

 Note: if using the Buffer object in more than one task, you may wish to lock it before a group of Put messages are issued, and unlock it afterwards.

 See also [Accepting Print](#), [Buffer.Get](#), and also Locking in the *Venom2 Tutorial*.

**Queue**

```
Queue  $\Rightarrow$  Int
```

Queue returns the number of data items that may be processed by calling [Get](#).

 See also [Put](#), [Get](#) [ReadPoint](#)


**ReadPoint**

```
ReadPoint  $\Leftrightarrow$  Int
```

The **ReadPoint** holds the position (or index) of the data item that the next call to **Get** will return.

**ReadPoint** is an [active variable](#), so may be read and written to (or set).

When there are no more elements to read using **Get**, **ReadPoint** has the same value as **Length**.

 The readpoint may be set between zero and Length.

 See also [Reset](#), [Get](#), [Flush](#).



## Remove

**Remove** (Int **start**, Int **size**)

Remove will remove a section of text from any point inside a text buffer, starting from the position given by **start**, and number of characters given by **size**. If the section to remove exceeds the buffer boundaries, then an error is given.

 [Insert](#), [GetLast](#)

## Reset

**Reset**

Reset resets the readpoint of the buffer back to the start of the buffer.

 See also [ReadPoint](#).

## Sort

**Sort** ([Int **sort\_type**, Any **xflags**])

For a buffer of Any in which all the elements are objects of the same type having a **Compare** method, **Sort** rearranges the objects into alphabetical order.

The default action, equivalent to a **sort\_type** value of 0, is for sorting to be case sensitive and in ascending order.

Sort_type	Direction	Case
0 (default)	ascending	Case sensitive
1	descending	Case sensitive
2	ascending	Case insensitive
3	descending	Case insensitive

In a case sensitive sort, all upper case letters are treated as lower than all lower case letters.

The "ignore case" flag is passed on as an optional second parameter to the buffer element object's **Compare** method. In particular this makes is compatible with string objects; for user defined Classes a compatible **Compare** method must be written.

The **xflags** parameter, if present, is passed as a 2nd parameter to the object's **Compare** method.

### Specification of Compare Method

`Compare(obj2 [, int ignore_case [, xflags]]) ⇒ Int`

Returns an integer which is:

Negative if the object is "lower" than the supplied object reference `obj2`

Positive if the object is "higher" than `obj2`

0 if the object "matches" `obj2`

<b>value</b>	The value to compare with some element or property of the object
<b>ignore_case</b>	For string values, non-zero for a case-insensitive match (default or 0: case sensitive) based on bit 1 of the <b>flags</b> parameter passed to the <b>Find</b> message.
<b>xflags</b>	Optional extra flags, number or other variable to specify how the comparison is done, e.g. you may want to select which of an object's member values to compare with the search target.

Note that string objects comply with this specification, though they don't use the **xflags** parameter.

### Example

This exploits printing to a buffer of any - See [Accepting Print](#)

```
-->make b buffer(Any)
-->print to b, "XYZ", CR, "pqrst", CR, "abc", CR, "123", CR
-->b.sort
-->print b
123
XYZ
abc
pqrst
-->b.sort(1)
-->print b
pqrst
abc
XYZ
123
-->b.sort(2)
-->print b
123
abc
pqrst
XYZ
-->b.sort(3)
-->print b
```

```

XYZ
pqrst
abc
123
-->

```

### Practical Example

```

Make fs filesystem("FLA")
Make b buffer(Any)
Print to b, fs:0
b.sort(2) ; b now has list of file names in alphabetical order

```

### Value

```

Value ⇒ Int
Value(Int base) ⇒ Int
Value(Float) ⇒ Float

```

Value will convert the text in a text buffer to a numeric value. Beginning at the start of a buffer, it skips over any leading white space characters (space, tab, new line) and takes account of numbers prefixed with a '+' or a '-' sign. It stops reading numeric characters when it encounters a character that isn't valid for the kind of number being looked for.

If there are no parameters, **Value** looks for a decimal integer:

```

x := textbuffer.Value ; Convert to a decimal number

```

If an integer parameter (**base**) is given then this indicates what number base is to be used:

base	Convert text to
10	Decimal integer
2	Binary integer
16	Hexadecimal integer

```

x := textbuffer.Value(16) ; convert to a hexadecimal number

```

If the parameter is the type indicator **Float** then the text is converted to a floating point number:

```

x := textbuffer.Value(Float) ; convert to a floating point number

```

## Notes


Value does not 'Get' any characters from the buffer, nor does it use or alter the ReadPoint of the buffer - it only looks at the text in the buffer and returns the numeric value of the first number it finds.

Integer bases from 2 to 36 may be used to read numbers in any base from 2 to 36. In bases above 10, the characters A-Z or a-z are used for the digits above 9.

If no numeric characters are seen, the value returned is zero, either integer or float.

If an integer value exceeds the [maximum](#) absolute value for an integer, the largest value that has the correct polarity is returned.

Floating point numbers in the text buffer may include an exponent suffix, e.g. 1.0e+12

 See also [TextAnalyser](#)

## Accepting Print

### Printing to a 'Text Buffer'


```
Print To <Buffer>, <print item>, ...
```

Text buffers (buffers of character data) are able to accept print. The print output is simply appended to any existing text in the buffer.

Example:

```
Make b Buffer(Char)  
Print To b, "Hello World", CR
```

 See also [PrintF](#), [Insert](#).

 *Note: you can also print to an object by sending it the [PrintF](#) message.*

### Printing to a 'Buffer of Any'

```
Print To <Buffer>, <print item>, ...
```

[Buffer of Any](#) may also accept print. In this case, a new string object is created to hold each line of text in the print output, and the string object is [Put](#) at the end of the Buffer.

A line of text is delimited by the '\n' character or CR. If there is no '\n' or CR at the end of a Print statement then the end of the Print statement is assumed to be the end of a line of text.

For example, this code

```
Make b Buffer(Any)  
Print To b, "Hello World", CR "This is another line"  
Repeat b.Length
```

```
Printf("<%S>", b.(Index0))
```

gives this output

```
<Hello World><This is another line>
```

⚠ Note: the maximum length of any individual line is 200 characters, the size of Venom's internal 'Print Job' buffer. Lines overflowing this limit will appear in separate strings. The overall length of the text is only limited by the memory available.

## Garbage collection

When you print to a Buffer of Any, new string objects are created to hold the lines of text. These will most likely need to be removed from memory at a later time – when they and the buffer are no longer needed. This is easy to manage in most situations as a Buffer of Any will automatically remove all objects it references when it is sent the [Die](#) message. It is often useful to use [AutoDestruct](#) on the Buffer of Any to achieve this.

## Printing

```
Print <Buffer>
```

```
Print <Buffer> : nChars
```

```
Print <Buffer> : start : nChars
```

## Text buffers

Printing a text buffer prints out all the text in the buffer.

If the optional format parameters are used then any section of the buffer may be printed, just as with [strings](#).

One format parameter specifies that the first n characters are to be printed, or the last n if n is negative.

Two format parameters specify printing of any portion of the string: the first parameter is the start position, and the second is the number of characters to print.

Parameter values that imply characters before the start, or after the end of the buffer will result in the print output being padded with spaces at the start or end.

Example:

```
-->Make b Buffer(Char)
-->b.Put("The quick brown fox")
-->Print b:9
The quick
-->Print b:-9
brown fox
```

```
-->Print b:3:10
    quick bro
-->Print b:-5:7
    Th
```

### Numeric buffers

```
Print <Buffer> [: f1 : ...]
```

Printing a numeric Buffer prints out each of its elements in a vertical column, with optional format specifiers being used directly on each of the elements as if it were a single number being printed.

```
-->Make b buffer(1.0)
-->b.Put(1.324)
-->b.Put(100.0)
-->Print b:5:2
    1.32
    100.00
```

▼ The format parameters supplied must be acceptable to all the items in the Buffer, else an error is generated.

## CANBus

The CANBus object allows the VM2 to communicate over CAN – the widely used networking standard for Controller Area Networks.

The CANBus object implements low level CAN communication protocols.

It supports the CAN protocols version 2.0A and B.

It does not implement any of the higher level protocols such as CAN Open, CAN Kingdom, etc, but it may be used to implement higher level protocols.

### Summary of messages

**Make**  
**Debug**  
**Element**  
**Free**  
**Get**  
**Look**  
**Mapping**  
**Off**  
**On**  
**Put**  
**Queue**  
**Speed**  
**Status**  
**Value**

### Current implementation details

*This describes some of how the object interfaces with the CAN hardware module on the VM2's STM32F103 microcontroller. Other parts of the description are included where they are relevant to the each message.*

The CAN hardware module has three Tx mail boxes and two Rx FIFOs.

Currently we only use one Tx mail box to send outgoing messages.

Outgoing messages are put in a software queue by the Put message and are sent out when the CAN Tx mail box is empty (this sometimes uses an interrupt service routine). This means there is no prioritising of outgoing messages - they are sent in queue order only.

Currently we use just one Rx FIFO to receive incoming messages. All the messages filters are directed to use FIFO 0. When an incoming message passes a filter and reaches the Rx FIFO an interrupt service routine places it in a software queue, from where it is read using the Look and Get messages. If the software queue overruns, a counter is incremented to indicate the overrun and all the old messages in the queue are lost. The 'current' Rx CAN frame is cached so that an overrun doesn't result in inconsistent frame data. See **Debug (1)** for more information.

### Bus off management

In later versions of the driver (after 2019), the CAN hardware module is set up to automatically handle 'Bus-off' conditions. In earlier versions this is not set. Please consult the device manual and use the Value message to access the CAN module registers directly if you want to alter this behaviour.

## Creation

```
Make <object> CANBus (Int bitrate, Int extended) ⇒  
CANBus
```

Make creates a new CANBus object and carries out basic configuration.

## Parameters

**Bitrate** is an integer that represents the bit rate required on the CAN bus in bits per second. Not every bit rate is selectable. The values that are available are discussed more fully in [Speed](#).

**Extended** is an optional parameter that sets the default CAN frame type: 0: standard 11-bit identifiers, 1: extended 29-bit identifiers.

## Mode

Immediately after Make the device is left in INIT mode. This means the device is off the bus and you can perform further set up operations. To go to another mode use [On](#).

## Initial filter set up

To simplify initial testing a single filter is set up by Make that accepts all incoming messages. See [Mapping](#) to change this.

## Examples

```
Make can CANBus(1E6 As Int) ; CAN bus at 1MHz, not extended IDs
```

```
Make can CANBus(125E3 As Int, 1) ; CAN bus at 125KHz, extended IDs
```

## Debug

```
Debug  
Debug (Int) ⇔ Int
```

Debug allows you to set and read some of the debug features of the CAN hardware and CANBus object.

Currently there are only a couple of options.

## Debug by itself

Debug by itself returns some printout of the values of some of the internal registers and other state.

```
Print can.Debug
```



## Debug(0)

The active variable **Debug (0)** sets and reads the *Loop back mode*, and *Silent mode* of the CAN hardware module as a bit pattern.

Bit 0: Loop back mode on / off

Bit 1: Silent mode on / off

Example:

```
can.Debug(0) := %01 ; set loopback mode to test the CAN system.  
can.Debug(0) := %10 ; set silent mode to sniff the CAN system.
```

Silent mode allows the CAN controller to monitor the CAN Bus without putting any signal on to the bus.

Loop back mode allows the CAN controller to receive all frames it sends out, but none of the frames from the external CAN bus.

If both silent and loop back modes are selected together than the system can do a 'hot self test', where loop back won't send any signal to the external CAN bus.

**To set these modes the CAN module must be in INIT mode, and then put in normal mode before sending CAN test frames. See [Off](#) and [On](#).**

## Debug(1)

The active variable **Debug (1)** sets and reads the 'Rx buffer overrun count'. This is a counter that keeps track of the number of times the CANBus receive buffer overruns.

An overrun condition happens when frames are being written to the Rx buffer faster than they are being read out using Get, eventually newer frames start to write over the oldest frames in the circular buffer.

Example:

```
If can.Debug(1) ... ; Has the buffer overrun?  
can.Debug(1) := 0 ; reset the counter.
```

## Element

**Element(Int Offset, Int nBytes) ⇒ Int**

The Element message allows you to read single-byte or multi-byte values out of a CAN frame's data segment.

The parameter **offset** is the offset in bytes to the start of the value within the CAN frame's data segment, and **nBytes** is the size of the item you want to read: 1, 2 or 4 bytes .

If the value you want to read out of the packet is in the little endian format then use a negative value for nBytes.

As elsewhere in Venom, the **Element** keyword can be omitted, simply using **can. (0)**, for example.

Examples:

```
print can.Element(0, 1), can.Element(1, 1)
print can.Element(0, 2)
print can.(0, -2)
print can.(1, 2)
print can.(1, -2)
print can.(2, 4)
```

 [Get](#), [Queue](#), [Look](#)

## Free

**Free**  $\Rightarrow$  Int

The Free message reports the number of free frame spaces in the output frame buffer

⚠ The output frame buffer can hold 16 frames before it is full. If the output buffer becomes full then Put will wait for space to appear in the buffer.

 [Put](#)

## Get

**Get**

The Get message discards the current frame at the front of the frame input buffer. This is usually done after reading the frame contents with [Look](#).

 [Queue](#), [Look](#)

## Look

**Look(Int)** ⇒ Int

The look message is used to look at the first (oldest) frame in the frame input buffer. To discard the current frame and move to the next frame, use the [Get](#) message.

Look takes a single parameter and uses that as an index to access different parts of the frame.

```
can.Look(0) ; Frame ID
can.Look(1) ; Frame data length (0-8)
can.Look(2) ; Extended frame? (True or False)
can.Look(3) ; Memory address of the current CAN frame data bytes.
```

You can also use Look with no parameters:

```
can.Look ; Frame ID
```

## Accessing the data bytes

You can use the [Element](#) message to read data bytes from the frame.

### Older functionality

The following are superseded by **Element**, but are included for reference:

```
can.Look(4) ; First byte of CAN data
can.Look(5) ; Second byte of CAN data.
... and so on to
can.Look(11); - last byte of CAN data (if there are that many).
```

Reading past the end of the data yields undefined values.

 [Get](#), [Queue](#), [Element](#)

## Mapping

**Mapping** (Int **filt\_num**, Int **mode**, Int **scale**, Int **A** ,  
Int **B**)

The Mapping message sets the CAN message receive filters.

When you first create a CANBus object, a single filter (in filter 0) is set up to accept all incoming messages. Effectively, the following code has been called:

```
can.Mapping(0, 0, 1, 0, 0)
```

If you want to restrict incoming messages then you will need to overwrite filter zero, or reset all the filters and start again - see [below](#).

There are 14 *filter banks* in the hardware module, selected using the **filt\_num** parameter

(numbered 0-13).

Each filter bank can be programmed into one of four configurations using the **mode** and **scale** parameters. The four configurations are listed in the table:

Number of filters per bank	Type(s) of filter(s) in each bank	Mode parameter value	Scale parameter value
1	A 32-bit identifier and a 32-bit mask	0 (mask mode)	1 (32-bit)
2	Two exact 32-bit identifiers	1 (ID mode)	1 (32-bit)
2	Two 16-bit identifiers and two 16-bit masks	0 (mask mode)	0 (Dual 16 bit)
4	Four exact 16-bit identifiers	1 (ID mode)	0 (Dual 16 bit)

### Masks

When configured in **mask mode**, mask values are used to select which bits of an incoming frame ID must match the filter ID value.

A 1 in the mask means *match*, and a 0 in the mask means *don't care*.

### Scale

Each filter bank can operate in one of two 'scales': 32-bit or 16-bit.

When using **32-bit scale**, the **A** and **B** parameters are both 32-bit values. **A** is the filter ID. **B** is either a mask value or an additional filter ID, depending on the **mode**.

The bits are arranged inside each 32-bit word like this:

**[29 ID bits] [RTR] [IDE] [0]**

Bits 3 and 2 are IDE and RTR respectively. Bit zero is not used.

When using **16-bit scale**, the **A** and **B** parameters each hold two 16-bit values.

In mask mode, the most significant 16 bits of a word are the mask value, and the least significant 16 bits are the ID value.

In ID mode, both sets of 16 bits are IDs.

The bit assignment within the 16-bit values is more complex:

**[11 ID bits] [RTR] [IDE] [3 extended ID bits]**

## Example

The line below sets up filter zero in 'single 32-bit' scale and 'mask' mode to look for any identifier with it's ID, RTR and IDE bits like this, where **x** is *don't care*:

```
00000001xxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
can.Mapping(0,0,1,$10000000, $F0000000)
```

*For more information on how the CAN hardware module works please see the STM32F103C/D/E reference manual.*

*You can use all forms of the Mapping message in any mode - you don't have to set up INIT mode first.*

## Reset all filters

### Mapping

Using the Mapping message without any parameters resets all the filters to their inactive state. No incoming frames will be seen until new filters are set up.

```
can.Mapping ; reset all the filters
```

## Off

### Off

Off will put the device into init mode. This is the same as **Can.On(1)**.

### Example

```
can.Off ; Go to init mode
```



[On](#)

## On

### On

### On(Int)

On will put the device into normal mode where it may take part in normal CAN interactions on

the bus.

If a parameter is passed to `On`, then the operating mode is set to the value of the parameter:

Mode number	Name	Description
0	Normal mode	The device is actively connected to the CAN network.
1	Init mode	The device is in initialisation mode. It must be in this mode to set some of the internal registers.
2	Sleep mode	The device is in sleep mode. This isn't well supported in the software as yet.
3 and above	Undefined - do not use	

Example

```
can.On ; Go to normal mode
```

 [Off](#)

## Put

```
Put(Int ID, Int Addr, Int Nbytes [, Int Ext])
```

The `Put` message is used to construct and send CAN frames. The outgoing frames are put in the output queue. If there is no room in the queue then `Put` will wait.

## Parameters

**ID** is the ID field of CAN frame.

**Addr** is usually the memory address of some data within an array (see [Array Address](#)).

**Nbytes** is the number of bytes to read from the address and send out in the CAN frame. The CAN specification allows between 0 and 8 bytes of data per frame.

The frames are queued in FIFO order and are sent out using just one of the device's transmit mailboxes. There is no prioritising of messages.

The optional **Ext** parameter allows the *standard* or *extended* state, set during the `Make` of the CANBus object, to be over-ridden on a per-frame basis when required. Non-zero values indicate an extended address.

For example:

```
can.Put (%11110101001, my_array.Address, 8) ; Send 8 bytes of data
```

## Queue

**Queue**  $\Rightarrow$  Int

The Queue message reports the number of frames waiting to be read in the input frame buffer.

For example:

```
Await can.Queue ; wait for a message to come in.
id := can.Look(0)
data := can.Look(4)
can.Get ; we're done with this message.
```

⚠ The input frame buffer can hold 32 frames before it overflows. If it overflows, the receiving process just carries on writing to the circular buffer. This will mean the queue will transition from 32 to 0, whereupon all frames in the queue are lost. The receive process then carries on normally.

 [Get](#)

## Speed

**Speed**  $\Rightarrow$  Int

Baud allows the bus bit rate to be set to a value in Hz.

Not every value is settable. These are the values you can use:

If the main CPU clock speed is **system.Speed** (usually 72MHz) then **CANBus.Speed** can take the values

```
system.Speed / 24 / N
```

Where N is an integer: 1, 2, 3, 4, etc, up to 512.

Typical values for the CAN bit rate are 1MHz, 500KHz, 250KHz and 125KHz.

The equivalent values of Speed are 1000000, 500000, 250000 125000. These can also be expressed like this:

**1E6 As Int**, etc.

## Examples

```
can.Speed := 125E3 As Int; set speed to 125KHz.
```

To perform this operation the CAN module must be in INIT mode. See [Off](#).

## Timing configuration

The sub-bit-timing configuration used by the CANBus object is fixed.

(Though you can change it by accessing the CAN hardware modules registers [directly](#) if necessary)

Generally it aims to use 12 Tq per Bit Time, and set the sample point at 75% (i.e. after 9 Tq).

This is assigned like so:

```
SynchSeg: 1 Tq
PhaseSeg1: 8 Tq
PhaseSeg2: 3 Tq
```

The synchronisation jump width is set at 1 Tq.

## Status

```
Status ⇒ Int
Status(Int) ⇒ Int
```

Status returns the error status of the device.

```
can.Status (0) ;Bus off status: non-zero if off the bus
can.Status (1) ;Error warning status: non-zero if warning is true
can.Status (2) ;Error passive status: non-zero if error passive i
can.Status (3) ;Tx error count (0-511)
can.Status (4) ;Rx error count (0-255)
can.Status (5) ;Last error code (0-7)*
can.Status (6) ;Raw error register contents*
```

If the parameter is omitted it defaults to zero:

```
can.Status ;Bus off status: non-zero if off the bus
```

\*Please refer to the STM32F103 reference manual for the meanings of these bit patterns.



## Value

**Value** (Int)  $\Leftrightarrow$  Int

Value allows direct access to the CAN module's register contents.

Examples:

```
can.Value (offset) := X ; set register at offset with value X
X := can.Value (offset) ; read register at offset.
```

For the register offset values you should refer to the **bxCAN** module in the STM32F103 hardware reference manual.

To perform some of these operations the CAN module must be in INIT mode. See [Off](#).

## Class-default

All user-defined classes recognise a set of 'Class-default' messages, as well as the members and methods that you define.

You can override any of these Class-default messages with methods or members of your own, and these will take precedence when you send a message to your object.

However, if you wish to call a Class-default message explicitly you can specify this using special syntax:

```
<obj>.[Class]<msg>
```

## Examples

```
x := new MyClass
Print x.[ClassName], CR ; Debug: print the class name of x
...
To MyMethod
  Print This.[ClassName], CR ; Debug: print the name of this class
End
```

This section describes all the Class-default messages.

## Summary of messages

**Die**  
**Length**  
**Name**  
**Print**  
**PrintF**

## Die

**Die**

The Class-default **Die** message removes the object from memory; it also passes on the **Die** message to any members that have the [AutoDestruct](#) attribute.

To call the class-default message **Die** from within a method of the class, the recommended pattern is:

**Base.Die**

If for debug purposes you want to know you are calling the Class-default message, use

**This[Class]Die**

## Length

**Length** ⇒ Int

The Class-default Length message returns the number of bytes needed to hold a *binary record* representation of the object.

(Objects created with Class can be used as data templates to process binary or text format records in files and other storage media).

When a class contains no Strings or Arrays then every instance of the Class has the same length. If a class contains one or more Strings or Arrays then the length will vary depending on the actual lengths of the Strings or Arrays in the particular object.

If the Array members are defined as **New** then they will always have the same length in every instance.

## Name

**Name**  $\Rightarrow$  String

Name returns the name of the object's Class name.

## Example

```
Class Test
End

Make t New Test
Print t.Name
```

Prints:

```
Test
```

## Print

```
Print <object>
Print <object> : ","
Print
Print(",")
```

The Print message is sent internally (by the system) to any object when you Print it. Any colon formatting values you supply are sent as parameters to the Print message. You can also send the Print message directly - see below.

The *Class-default* Print message will print the user-class object in a different format depending on the formatting parameters you supply.

If no formatting parameters are supplied then a default 'debugging' style is used. This lists the class's name and memory address, followed by each member's name and value, eg:

```
-->Print p
Person (at $64000c64)
  Name = Fred
  Age = 35
```

However, if a colon formatting parameter of type **String** is supplied then the object is printed in one of several text formats.

If the string contains "INI" (case sensitive!) then the object is printed in 'INI file' format:

```
-->Print p:"INI"
Name="Fred"
Age=35
```

Otherwise a comma separated value (CSV) format is used; the first character of the string is

used as the separator character:

```
-->Print p:",",  
"Fred",35
```

To read data in CSV and INI file format you can use [File.Get](#).

### When can you send the Print message directly?

You can send the **Print** message directly to an object, but it will only work when there has already been a *print job* set up. A print job is 'claimed' when a **Print** statement starts, and 'released' when the **Print** statement ends, after printing all the items in the print list to the print job.

So you can send a **Print** message from inside any code called by a **Print** statement - e.g. a **Print method**.

However, if you call a **Print** message from outside a **Print** statement you may get a [runtime error](#): *No print job*.

### Printf

```
Printf(<String> format, [Any ... ])
```

**Printf** for user-defined Classes works in the same way as for any Venom object that can accept print. [See here](#).

However, if your Class is to accept print it must define an **AcceptPrintJob** method. (It is not usually useful to override the Class-default **Printf** message).

The **AcceptPrintJob** method should take **exactly one parameter** - which is a [PrintJob](#) object containing the text to be printed.

**PrintJob** objects are only ever seen in the context of an **AcceptPrintJob** method, so most of the time you won't need to know about them.

### Example

```
Class Label  
  XPos Int  
  YPos Int  
  Text New String(100)  
  
  To AcceptPrintJob(pj)  
    If pj.Status And 1  
      Text.Empty  
      Text.AcceptPrintJob(pj)  
    End  
  
End
```

```
-->l := New Label  
-->Print To l, "This is some text"
```

See also [PrintJob](#)

## CRCGenerator

This object is a wrapper for two different CRC generators:

- A 16 bit generator using the standard CRC-CCITT polynomial and initializer
- A commonly used 32 bit CRC generator
- A 16 bit CRC generator compatible with MODBUS CRC

A CRC (Cyclic Redundancy Check) is a number computed from a block of data such as a stream of text or the contents of a buffer or array, in such a way that any change in the content of the data is likely to change the value of the CRC. This has applications in checking data for corruption when transmitted through an error prone medium. The computation is done in such a way that most forms of simple data corruption (small numbers of incorrect bits, missing bytes etc) are very likely to change the CRC value and thus be detected.

The CRCGenerator is from time to time enhanced to include specific CRC algorithms for other industry standard communications protocols. These could be coded in Venom, but a version implemented through the CRCGenerator object would be much faster.

### Summary of messages

**Make**  
**Get**  
**Put**  
**Reset**  
**Value**

### Creation

**Make** <object> **CRCGenerator** (Int **type**) ⇒ CRCGenerator

Creates a CRCGenerator of the indicated type.

**type** is:

- 16 16 bit generator using CCITT-CRC16
- 32 32 bit generator
- 'M' MODBUS CRC - another 16 bit CRC using a different algorithm.

- 8     simple sum truncated to 8 bits
- 'S'   simple sum truncated to 16 bits

The **8** and **'S'** options offer a speed improvement over performing the same simple calculation in Venom code.

## Get

**Get**  $\Rightarrow$  Int

**Get**(buf, array or string **data**)  $\Rightarrow$  Int

If a string, buffer or array is supplied as a parameter, the CRC of its contents are calculated and returned.

The element type of the buffer or array must be text or 8 bit.

With no parameter, the CRC resulting from data in previous **crc.Put** messages is returned.

## Put

**Put**(Int **byte**)

*byte* is an integer value used to update the current CRC. Only the lowest 8 bits of the parameter are used.

**crc.Put**(**data** [, Int **start**, Int **length**]

**data**     An array, buffer or string.

           Array element type must be 8 bit.

           Buffer element type must be 8 bit or text

**start**     (default 0) 1st element to process

**length**    (default all) number of elements to process

All or the selected part of the data is entered into the CRC calculation.

This message can be repeated to build up a CRC over selected fields of a block of data.

## Reset

### Reset

Sets the generator to its initial state, the same as after Make.

## Value

### Value

Value is a synonym for [Get](#).

## DateTime

DateTime objects may be used to convert between the date and time in the familiar calendar form, and an integer value of time in seconds. This 'absolute seconds' value may be used for calculations involving real dates and times.



[RealTimeClock](#)

## Summary of messages

**Make**

**Day**

**DayOfWeek**

**Hour**

**Minute**

**Month**

**Adjust**

**Second**

**Time**

**Update**

**Valid**

**Year**

**Print To**

**Print**

## Creation

**Make** <object> **DateTime** [(Int seconds)]

A DateTime object is created with its time set to the optional value supplied.

If no value is supplied the time is set to 0.

For example:

```
Make dt DateTime
Make dt DateTime(clock.Time) ; Set the time to 'now'.
```

 Each DateTime takes a block of ~34 bytes from the heap.

 See also [RealTimeClock](#)

## Adjust

**Adjust**(Int part , Int increment)

Adjust allows you to implement ‘digital watch’ style methods to change the date in a DateTime or RealTimeClock object.

Every time Adjust is called it will increment or decrement a single part of the date or time, rolling over if the maximum or minimum value for that field is exceeded.

The part of the date (Day, Month, Year, Hour, Minute, Second) is specified by an integer parameter, or by a character constant (which is also an integer, actually).

Part of date	Number	Letter
Day – ranging 1-31	6	‘d’
Day – only correct date range	0	‘D’
Month	1	‘M’
Year	2	‘Y’
Hour	3	‘h’
Minute	4	‘m’
Second	5	‘s’

*All other values are ignored.*

For example

```
Date . Adjust(0,1)
```



```
Date . Adjust('D',1)
```

are the same – increment the day.

To decrement a part of the date, use an increment of '-1'.

If you use values of the increment larger than 1, then this value will be added or subtracted from the date element. However if the value rolls over, it will roll over to the exact maximum or minimum value for that part of the date.

## Coping with leap years with Adjust

There are two main ways of dealing with Feb 29<sup>th</sup> when entering dates.

The first (the simplest to program) is to let the DateTime object deal with it. You just write code to Adjust the DateTime, and print out the result. The down side to this approach is that you won't be able to enter invalid dates that you intend to correct later. Also if you enter, say 29<sup>th</sup> Feb 2008 (a leap year), and then increment the year, the DateTime object will print the new date as 1<sup>st</sup> March 2009.

The second method is to tell Adjust to allow invalid dates, and to make sure the date is valid after entry is complete.

To do this you will have to print out the date elements of the DateTime object individually, or use the special format option that allows [printing of invalid dates](#).

Use the Valid message to check if the date is OK after entry is complete.

The following procedure demonstrates both methods using characters from the serial port to Adjust a date. To use the procedure, run it and then enter any of the following characters to change the date: D, d, M, Y, h, m, s. Use the + or – characters to set increment or decrementing of the date.

To use the legal-dates-only method, never type a lower case 'd', and use the first printing option. Leave both Print statements in to compare methods.

```
To date_entry
  Local increment := 1
  Local date := New DateTime
  Forever
  [
    char := serial.Get
    Select Case char
    Case '-'
      increment := -1
    Case '+', '='
      increment := 1
    Case 13 ;Escape on seeing Carriage Return.
    [
      If date.Valid
        Break
      Print "Bad date - keep trying",cr
```

```

]
Case Else
[
    date.Adjust(char,increment) ;Increment the date.
    ; Use this Print if you don't
    ; allow invalid dates to be entered.
    Print date:0," "
    ; -Or- this if you do.
    Print date:0:1
    Print CR
]
]
End

```

 See also [Valid](#), [Print](#), [Update](#)


## Day

**Day**  $\Leftrightarrow$  Int

The *Day* [active variable](#) holds the ‘day of the month’ number for the DateTime object, counting from 1.

*Day* may be set to values too large for the particular month. It will be wrapped around to a real date if Update is called.

When printing a DateTime object, you can choose whether to print the date as you have entered it, or a guaranteed-real date. See [Printing](#) and [Update](#).

 Day can hold from 1 to 31.

## DayOfWeek

**DayOfWeek**  $\Rightarrow$  Int

DayOfWeek returns the day of the week, in days since Sunday. It cannot be set because it is dependent on the date. The day numbers are as follows:

Sunday	0
Monday	1
Tuesday	2
Wednesday	3
Thursday	4

Friday	5
Saturday	6

## Hour

**Hour**  $\Leftrightarrow$  Int

The Hour [active variable](#) holds the hours value of the DateTime object.

▼ Hour can hold 0 to 23.

## Minute

**Minute**  $\Leftrightarrow$  Int

The Minute [active variable](#) holds the minutes value of the DateTime object.

▼ Minute can hold 0 to 59.

## Month

**Month**  $\Leftrightarrow$  Int

The Month [active variable](#) holds the month-number of the DateTime object, counting from 1.

▼ Month can hold from 1 to 12.

## Second

**Second**  $\Leftrightarrow$  Int

The Second [active variable](#) holds the seconds value of the DateTime object. This is the number of seconds in the time and date representation, not the 'absolute seconds' value, which is given by Time.

▼ Second can hold 0 to 59.

## Time

**Time**  $\Leftrightarrow$  Int

The Time [active variable](#) holds the ‘absolute seconds’, the number of seconds since the datum: 00:00 on 1st January 1990.

The following example shows a DateTime being created and set to the current time from the RealTimeClock:

```
Make now DateTime
now.Time := clock.Time
```

⚠ Time can hold values representing the year 2089 and beyond. Currently other elements of the DateTime object make it unwise to use dates beyond 2089.

🔍 See also [RealTimeClock.Time](#).

## Update

**Update**

The Update message is for dealing with possible invalid dates. You can put invalid dates (such as 30 Feb 2009) into the Day, Month and Year values of a DateTime. However if you read Time then of course you will always get a value that represents a valid date. However, the Day and Month values will not have been updated to corrected values. Updating Day and Month only occurs when either Time is *set*, or Update is used, or during printing. The following two lines are equivalent and will fix invalid dates:

```
Now . Update
Now . Time := Now.Time
```

🔍 See also [Valid](#), [Print](#)

## Valid

**Valid**  $\Rightarrow$  Int

Valid returns True if the DateTime contains a real date, i.e. not one like 30th Feb.

🔍 See also [Update](#), [Print](#)

## Year

**Year**  $\Leftrightarrow$  Int

The Year [active variable](#) holds the years value of the DateTime object.

```
-->Make big_year DateTime
-->big_year.Year := 2002
-->Print big_year.Year
2002
```

⚠ Year can hold from 1990 to 2089

## Accepting Print

```
Print To <DateTime> , <print list>
```

Printing to a DateTime sets the time and date. It follows the same rules as for printing the time to the [RealTimeClock](#) object.

For example:

```
Print To dt, "2010-06-15 10:56:00"
```

💡 *Note: you can also print to an object by sending it the [PrintF](#) message.*

## Print

```
Print <DateTime>
```

Printing a DateTime object prints the date and/or time held by the object. It is printed in ISO format by default, but you can also specify the format in which the elements of the date and time are printed.

ISO date time format is the date: a four-digit year followed by a two-digit month and day, separated by dashes, and then the time: hours, minutes and seconds separated by colons. For example:

```
Print dt
2012-04-06 12:05:33
```

## Custom date and time formats

You can specify the printout of any date or time using a format specifier. The most versatile format specifier is a format string that uses special codes to encode the elements of the date or time you wish to print. The Venom syntax for printing with format specifiers is

```
Print <DateTime> : format_string
```

For example:

```
Print dt: "ddMMM'yy"
06Apr'12
```

The list of date and time formatting codes appears in the table below.

Code	Meaning
<b>aa</b>	<i>am</i> or <i>pm</i>
<b>d</b>	Day number 0-31
<b>dd</b>	Day number 00-31
<b>ddd</b>	Day, abbreviated name
<b>dddd</b>	Day, full name
<b>h</b>	Hour 1-12
<b>hh</b>	Hour 01-12
<b>H</b>	Hour 0-23
<b>HH</b>	Hour 00-23
<b>m</b>	Minute 0-59
<b>mm</b>	Minute 00-59
<b>M</b>	Month 1-12
<b>MM</b>	Month 01-12
<b>MMM</b>	Month, abbreviated name
<b>MMMM</b>	Month, full name
<b>o</b>	Day ordinal suffix: two characters placed after the day number as in: 1 <sup>st</sup> , 2 <sup>nd</sup> , 3 <sup>rd</sup> , 4 <sup>th</sup> , etc.
<b>s</b>	Seconds 0-59
<b>ss</b>	Seconds 00-59
<b>y</b> or <b>yy</b>	Year 00-99
<b>yyyy</b>	Year as 4-digit number
<b>&lt;text&gt;</b>	Embed literal text between <b>&lt;</b> and <b>&gt;</b>
<b>\</b>	The next character is literal. In practice this is only needed to enter literal <b>&lt;</b> characters. Note that two <b>\\</b> are needed in a quoted string to enter a single <b>\</b> . E.g. use " <b>\\&lt;</b> " to output a <b>&lt;</b> character

Any other characters in the format string will appear in the output without conversion.

For example:

```

Make dt DateTime
Print To dt, "2012-04-11 13:05:08"
Print dt : "yyyy-MMM-dd (ddd) <Time:> HH:mm:ss", CR
2012-Apr-11 (Wed) Time: 13:05:08

```

An example illustrating the use of \:

```

Print dt:"\\<MMMM>", CR
<April>

```

Note the closing > didn't need to be escaped-out because it is only a treated as special character after an opening <.

## Printing invalid dates

```
Print <DateTime> : format_string : f2
```

If a second format specifier of value 1 is used, then the date is printed in the same style, but instead of a fully validated date being printed, the contents of the date and time registers are printed as they are. This can often be useful when entering dates via a user interface that adjusts the month and the day of the month independently.

 [Valid, Update](#)

## Locale

It is possible to change the day and month names, and ordinal suffixes to suit different locales. See [here](#) for more information.

 See also [printing RealTimeClock](#).

## Old style formatting

There is another way to format dates and times which involves picking one of a set of fixed formats from a list given a format number.

The syntax is

```
Print <DateTime> : fn
```

Or you can use the following if you want to print invalid dates:

```
Print <DateTime> : fn : 1
```

The following examples all represent 1:50pm on 12 January 2011.

f	Description	Layout	Example
---	-------------	--------	---------

:0	ISO Date and time	YYYY-MM-DD HH:MM:SS	2011-01-12 13:50:00
:1	ISO Date and time, no seconds	YYYY-MM-DD HH:MM	2011-01-12 13:50
:2	ISO Date	YYYY-MM-DD	2011-01-12
:3	ISO Time	HH:MM:SS	13:50:00
:4	Hours(12), Mins, Secs, am/pm	HH:MM:SSxx	01:50:00pm
:5	Day, Month, Year	DD-MM-YY	12-01-11
:6	Month, Day, Year	MM-DD-YY	01-12-11
:7	Day of the week	ddd	Mon
:8	Month	mmm	Jan
:9			
:10	Day, Month, Year, H(24), M, S	DD-mmm-YY HH:MM:SS	12-Jan-11 13:50:00
:11	Day, Month, Year, H(24), M	DD mmm YY HH:MM	12-Jan-11 13:50
:12	Day, Month, Year	DD-mmm-YY	12-Jan-11

## Digital

The **Digital** object controls both individual digital channels, and groups of channels. The output state maybe set and/or the input state read. Several different digital I/O devices are supported.

The input state is generally read using the **Asserted** message.

The output state is generally set using **On** and **Off**, or by setting **Asserted**.

## Summary of messages



**Make**  
**Asserted**  
**High**  
**Low**  
**Off**  
**On**  
**Pulse**  
**Toggle**  
**Value**  
**Print**

## Creation

Digital objects may be created to drive the controller's on-board I/O ports, or PCF8574 ICs attached to the I2C Bus.

- [On-board digitals](#)
- [I2C Bus digitals](#)

## On-board

**Make obj Digital(Int channel [,Int Attributes])**

Make can take two parameters - the channel number and an optional 'attribute' code.

Each I/O channel on the VM2 is identified by a hexadecimal number - hence the \$ signs in the example code. See the VM2 datasheet for details of each channel.

Attributes are explained below, however the code for the most simple Digital objects are given here:

```
Make d digital($14)    ; Dig input on channel $14
Make d digital($11,1)  ; Dig output on channel $11
```

## Attributes

The Attributes parameter defines whether the channel is an *input* or an *output*, *active high* or

*active low.*

The VM2 has internal pull-up or pull-down resistors that can be used with inputs, and can operate in *push-pull* or *open drain* output mode. These modes are controlled by the attribute parameter too.

The default value for Attributes is 0 - which is an *active low input, pulled high internally*. This is the most commonly required type of digital input.

To create one of these you can omit the attributes parameter:

```
Make d digital($14) ;Dig input on channel $14, pulled high, active low
```

And the most commonly required type of digital output is defined like this:

```
Make d digital($14,1) ; Dig out, push-pull, active low.
```

For the less commonly required input or output attributes you have to set binary bits within the attributes parameter.

Note that all inputs have *even* attributes, and all output attributes are *odd*.

#### Attributes are fixed

These attributes can't be changed once the object is created. If you need to change a physical pin from one type to another, then you should re-create the object. This doesn't take very long.

```
Make d digital($14,1) ;Output
```

... and later ...

```
Make d digital($14) ;Input
```

#### Input attributes

For an input, the attribute bits are these:

	Bit 2	Bit 1	Bit 0
When 1	Floating	Active High	0
When 0	Pulled (to inactive state)	Active Low	

So to create a digital input that is floating and active high you can use either of these two:

```
Make d digital($14,6)
```

```
Make d digital($14,%110) ;Using Binary notation is useful.
```

#### Output attributes

For an output, the attribute bits are these:

	Bit 2	Bit 1	Bit 0
When 1	Open Drain	Active High	1
When 0	Push-Pull	Active Low	

So to create a digital output that is open-drain and active low you can use either of these two:

```
Make d digital($14,5)
Make d digital($14,%101) ;Using Binary notation is useful.
```

#### Output edge speed attributes

You can also set the 'speed' of the digital outputs by setting bits 3 & 4 of the attribute parameter. This speed governs the rate of change of output voltage when the output changes state, and is given in terms of the maximum frequency theoretically possible on the output.

Bit 4	Bit 3	'Speed'
0	0	2 Mhz
0	1	10 MHz
1	0	50 MHz
1	1	Reserved

```
Make d digital($14,%10011) ;Dig out 50MHz, push-pull, active high
```

#### Some useful Attribute constants

```
#Define IN_AL_PH %000 ; Input active low, pulled high.
#Define IN_AH_PL %010 ; Input active high, pulled low.
#Define IN_AL_FL %100 ; Input active low,floating.
#Define IN_AH_FL %110 ; Input active high,floating.

#Define OUT_AL_PP %001 ; Output active low, push-pull.
#Define OUT_AH_PP %011 ; Output active high, push-pull.
#Define OUT_AL_OD %101 ; Output active low, open drain.
#Define OUT_AH_OD %111 ; Output active high, open drain (not like
```

 Digital is a [Zero-Memory](#) object

#### I2C Bus

```
Make <object> Digital(Int channel)
```

Make normally takes just one parameter - the channel number. Channel numbers are listed in the datasheet for an I/O card, or in this [table](#).

For example

```
Make relay Digital(128)
```

It is possible to make multi-bit digital objects on the I2C Bus by specifying the start and end channels in the multi-bit port. These must both be located within the same group of 8 digital I/O pins.

For example

**Make relay Digital(128, 130) ; 3-bit wide digital I/O port.**

 Digital is a [Zero-Memory](#) object

## PCF8574 Channel table

Channels on PCF8574 ICs on the I2C Buses are detailed in the table below:

I2C Device	Address inputs: A2 A1 A0	Channel Numbers I2C Bus 1	Channel Numbers I2C Bus 2
PCF8574	000	128 - 135	384 - 391
PCF8574	001	136 - 143	392 - 399
PCF8574	010	144 - 151	400 - 407
PCF8574	011	152 - 159	408 - 415
PCF8574	100	160 - 167	416 - 423
PCF8574	101	168 - 175	424 - 431
PCF8574	110	176 - 183	432 - 439
PCF8574	111	184 - 191	440 - 447
PCF8574A	000	192 - 199	448 - 455
PCF8574A	001	200 - 207	456 - 463
PCF8574A	010	208 - 215	464 - 471
PCF8574A	011	216 - 223	472 - 479
PCF8574A	100	224 - 231	480 - 487
PCF8574A	101	232 - 239	488 - 495

PCF8574A	110	240 - 247	496 - 503
PCF8574A	111	248 - 255	504 - 511

Formula for channel number of a digital on a PCF8574 on an I2C Bus:

**Bus \* 256 - 128 + Address \* 8 + PCF8574A \* 64 + PCF8574\_Pn**

Where:

Bus is the I2C Bus number (1 or 2)

Address is the binary value of the address bits on the IC ( $A_0 + 2*A_1 + 4*A_2$ )

PCF8574A is 1 if the IC is an 'A' suffix, else 0.

PCF8574\_Pn the port number on the IC, 0 - 7.

## Reading the input value of a device that is not present

Note: if digital channels are assigned to a PCF8574 device that is not present on the I2C Bus, then the Make will not give an error, but reading the input value of the channels will yield 1 or 'not asserted' because the default state of the bus is pulled high.

## Asserted

**Asserted**  $\Leftrightarrow$  Flag

Asserted is used to read digital inputs and outputs, and also to set digital outputs.


When it is read, Asserted returns True or False depending on whether the channel is On or Off, respectively. Whether the channel is an input or an output, Asserted returns the state of the channel as if it were an input – i.e. it reads the actual voltage level rather than 'what it ought to be'.


An output is turned on when Asserted is set to True (or any non-zero integer) and off when Asserted is set to False.

**Asserted sent to a pulled input will make it pull to the state given.**

Example:

```
To thermostat
  Make heater digital(32)
  Every 1000
    heater.Asserted := temperature < 50
End
```


 See also [On](#), [Off](#), [High](#), [Low](#)

 You can set whether the input or output is active high or active low.

## High

### High

High sets a digital output to its high voltage state - as seen at the output of the VM2 or PCF8574.

 See also [On](#), [Off](#), [Low](#), [Asserted](#)

## Low

### Low

Low sets a digital output to its low voltage state - as seen at the output of the VM2 or PCF8574.


 See also [On](#), [Off](#), [High](#), [Asserted](#)

## Off

### Off

Off sets a digital output to its inactive state.

 See also [On](#), [High](#), [Low](#), [Asserted](#)

 You can set digital I/O to be active high or active low.

## On

### On

On sets a digital output to its active state.

 See also [Off](#), [High](#), [Low](#), [Asserted](#)



## Pulse

### Pulse

Pulse momentarily pulses an output to the opposite state.

**d.Pulse**

- 🕒 The pulse width is guaranteed to be longer than 300nS, but could be considerably longer than this, depending on how long it takes to write to the digital port.

## Toggle

### Toggle

Toggle inverts the state of the output. It does not work for inputs.

**d.Toggle** ; *invert the state of 'd'*

It is very useful on the command line to show that an output is working:

```
-->Make d digital($7F, 1)
-->Every 100 d.toggle
```

## Value

### Value ⇔ Int

Value is used to read and write multi-bit digital inputs and outputs (currently only available on I2C based Digital objects).

Any '1' in the binary representation of the value is equivalent to a high logic level on the I/O pin, and vice versa for '0' bits.

Example:

```
Make port Digital(128, 135) ; Make a multi-bit digital port.
...
port.Value := %10011000 ; Set the bits of the port
```

🔍 See also [Make multi-bit port](#)

## Printing

```
Print <Digital> :f1
```

If no format specifier is used, the state of the channel or port (as defined in [Asserted](#)) as "ON "or "OFF", always with 3 characters is printed

```
-->Make d Digital(128)
-->Print d
OFF
-->d.On
-->Print d
ON
```

If a format specifier greater than zero is supplied, then the object is printed as a 1 or a 0 depending on whether it is high or low. This is independent of the active high or active low attribute.

```
-->Print d:3
[Digital: 0]
```

 See also [Asserted](#)

## Encrypter

### The Encrypter Object

The Encrypter object can be connected to any Venom object that accepts **Put**, **Get** and **Queue** messages, or used as a standalone keystream generator.

Data passed though it is encrypted using the RC4 cipher. Passing encrypted data though the identical process with the same starting point (key) decrypts the data.

It is impossible to decrypt the data without knowing the key value used to encrypt it. The encrypter can thus be used to protect data from readability in files or in transit via communication media such as serial ports or TCP connections.

### Security and Efficiency

The RC4 cipher is fast and takes up very little memory. As with all good encryption schemes the algorithm is well known and documented. Encryption of any message or file is dependent only on a secret key known to authorised users of the encrypted data, and on the difficulty of recovering the key even when samples of known plaintext and its encrypted version are known.

RC4 is a widely used cipher (is still the most commonly used for secure web communications, for example) and is secure as long as it is used properly.

It does have a weakness in the key generation process: some information about the first few bytes of the key can be obtained if the plaintext and encrypted data are both known. In repeated



sessions or messages using the same key or keys with similar first few bytes it is quite easy, given enough known plaintext and cipher text, to work out the key. The WEP ("Wired Equivalent Privacy") encryption used in earlier wireless LANs suffers from this vulnerability, but the problem can be avoided in two ways:

1. By running the encryption process initially on blank data for a fixed and agreed number of bytes (256 is better than none, 768 and 1024 are commonly used figures). This is called RC4-drop $N$  where  $N$  is the number of bytes discarded. This can easily be done in Venom by connecting the encrypter to a **nil** object initially and getting and discarding the required number of bytes before using on real data.
2. By changing keys frequently, and in a way that shows no correlation between one key and the next. WEP was easily broken because it concatenated a shared secret key with a 24 bit counter which incremented for each packet sent. Key retrieval was easy because only a small part of the key ever changed. The Venom Encrypter object provides a simple method for combining a fixed secret with a changing value (such as a counter or time/date information) and using a 256 bit SHA-2 hash generator to produce a 256 bit key in which each bit has a 50% probability of changing even if there is only a single bit change in one of the supplied values.

## Message Summary

**MAKE**  
**Connect**  
**Get**  
**Key**  
**Put**  
**Accepting Print**

## Creation

**MAKE** <Encrypter> **Encrypter**([Any object])

The **object** parameter is any Venom object which can accept either the **Put** message, or the **Get** and **Queue** messages.

This creates an encryption layer which will either accept data to send to the object, or receive data from the object.

Example:

**Make Enc Encrypter(serial)**

If you are going to use the encrypter with an object that does not exist yet by using the [Connect](#) message later (e.g. a file that is not yet open), you can supply **nil** as the object when creating the encrypter. This is the default if no object parameter is specified.

An encrypter object created with **nil** or with no parameter can also be used to provide a key stream that is applied to any data by using the exclusive-OR (Venom EOR operator) on successive bytes of the data with byte values obtained with successive **Get** messages to the encrypter. This method might be easier for encrypting an array of byte values in place, for example.

A **nil** encrypter could also be used as a source of random byte values for any other purpose, and is faster than [RandomNumberGen](#) if only 8 bit values are required.

## Connect

**Connect**(*Any object*)

This connects the encrypter to a new object, replacing that specified (if any) when the encrypter was created.

Note that if the object is a file, it should be always be of 8 bit integer type. When text is encrypted, the encrypted form is not suitable for storing in a text file.

## Get

**Get**  $\Rightarrow$  **Int**

Get one decrypted/encrypted byte from the connected object by processing the returned value from a **Get** message to the object.

Returned value is an integer in the range 0 - 255.

Usually **Get** will be used for decrypting data e.g. read from an encrypted file or received on a serial port or TCP stream.

If the encrypter was created with **nil** or no parameter, the behaviour is as if it were encrypting an endless source of 0 values, and can be used for encrypting data by XOR-ing it with the data one byte at a time, or as a source of pseudo-random byte values for any other purpose.

```
enc.Get(string s [, Int use_queue])
```

In this form, **s** is a string variable. Data is read and decrypted from the connected object and replaces the string's previous contents. The decrypted data is expected to be text, and when a newline character ("**\n**" or 10) is received or the maximum string length is reached, reading stops. The newline character is not stored in the string.

**use\_queue** should be specified and non-zero if the connected object is a buffer or file, or any other where a **Get** message would result in a run time error when there is no more data available. If **use\_queue** is set, an end-of-input condition (Queue = 0) will be checked and treated as if a newline had been encountered.

## Key

```
Key(key1 [, key2 [, int iterations]])
```

Each key parameter can be:

- A String variable or fixed string
- A Buffer of text or 8 bit integers
- An Array of 8 bit integers

## If One Key Value is Supplied

The key value is used directly with the standard RC4 key scheduling algorithm. This is potentially insecure, but it enables communication with third party users of RC4, whatever means is used to generate or disseminate the key

## If Two Keys are Specified

The two keys are combined using the 256 bit SHA-2 hash algorithm according to the following key-strengthening scheme:

```
key = SHA256 (key1 . key2)      (where ' . ' denotes concatenation)  
Repeat <iterations> times:  
    key = SHA256 (key . key1 . key2)
```

The resulting 32 byte (256 bit) key value is then used with the standard RC4 key scheduling algorithm.

Default value for iterations is 1000.

The reason for using two key values is for convenience when you need to combine a long-term secret value with a once-per-session value to create a key for a file or message.

The reason for using the hash algorithm is to make sure that every bit of the resulting 256 bit key is uncorrelated with any part of the supplied key values.

The reason for the repeated iterations is to waste time. The extra work involved makes it harder for an attacker to find a key by brute force methods, since each guess needs the full number of iterations to calculate the derived key. With keys of length 20 bytes each and 1000 iterations the process takes 280 milliseconds on a VM2. An iterations value of 0 is valid and means the key will be a simple SHA256 of the two keys, which will take around 380 microseconds for 2 keys of 20 bytes each. This still gives a key value which is secure against the inherent key scheduling weakness of RC4.

## Examples

Very simple use:

```
f := fs.open("myfile.x", 8)
Make enc Encrypter(f)
enc.key("Venom Control Systems")
```

Combining a shared secret with a one-time value stored in the first 8 bytes of a file:

```
ARRAY secret(8, 16)
    12, 34, 56, 78, 11, 22, 33, 44,
    55, 66, 77, 88, 99, 123, 234, 200
END

f := fs.open("myfile.x", 8)
MAKE b buffer(8)
REPEAT 8 b.put(f.get)      ; 1st 8 bytes = encryption seed
MAKE enc Encrypter(f)
enc.Key(secret, b)
x := enc.Get               ; read remainder of file through decrypter
```

## Put

**Put**(Int)

Encrypts the supplied byte value and sends it to the connected object with an **Put** message.

## Queue

**Queue**  $\Rightarrow$  **Int**

Sending a **Queue** message to an encrypter object returns the number of bytes/characters available, based on the result of a **Queue** message sent to the connected object. In some cases a **Queue** message directly to the connected object will not return the same value because the encrypter may buffer some data internally. The encrypter's own **Queue** message will always return the total number of bytes available.

### Example

Printing an encrypted text file

```
f := fs.open("text.rc4", Int 8)
enc.connect(f)
enc.key(key1, key2)

; File was plain text encrypted; print the file contents
WHILE enc.Queue
    PRINT chr enc.get
```

Faster method using string get

```
; File was plain text encrypted; print the file contents
make s string(100)
WHILE enc.Queue
[
    enc.Get(s)
    printf("%s\n", s)
]
```

### Accepting Print

**PRINT to** <Encrypter>, **list**

**enc.printf**(str, values)

Normal printing of text data to an Encrypter results in encrypted text being sent to the connected object.

## Ethernet

The Ethernet object controls an Ethernet hardware interface. When VM2 Ethernet hardware is connected to a LAN, with the aid of the Ethernet interface all packets for addresses on the LAN or reachable via a gateway on the LAN will be routed through the Ethernet interface. All the IP based protocols such as UDP and TCP and higher level protocols that use these will then work over the Ethernet interface.

Any Ethernet device has an Ethernet address, commonly called a MAC (Media Access Control) address which should be unique in the world for each device, to guarantee that devices on the same network will all have individual addresses. The MAC address of each Ethernet interface we supply is programmed in during production and taken from a unique address block originally assigned to Micro-Robotics Ltd by the IEEE.

In IP networking, each device on a LAN will have an IP address too, which should be unique within the LAN but part of a block of related addresses. Many networks use a mechanism called DHCP (Dynamic Host Configuration Protocol) to assign IP addresses automatically and Venom supports this; alternatively IP addresses can be set explicitly in code.

The Ethernet object automatically handles the ARP (Address Resolution Protocol) to match IP and MAC addresses for devices on its network.

See also [TCP/IP Networking](#)

### Summary of messages

**Make**  
**Address**  
**Connect**  
**Count**  
**Debug**  
**ErrorAction**  
**Off**  
**On**  
**Status**  
**Valid**  
**Time**  
**Print**

### Creation

**Make** <object> **Ethernet**

This is all that is needed for many systems. It assumes the standard VM2 Ethernet circuit is connected, and a network with a DHCP server which will allow automatic allocation of the

Ethernet interface's IP addresses and assignment of a local name server and default gateway if needed and available.

```
Make <object> Ethernet([Int select [, address [, DNS [,  
  , DGW]]]])
```

This form allows for custom hardware and also enabled IP addresses to be specified explicitly.

Param	Default	Description
select	12	A bitmapped combination of values to select hardware addressing. Bits 0 and 1 are an SPI address (default 00) Bit 2 is 1 if addressing is to be used (default 1) Bit 3 is 1 for SPI2, 0 for SPI 1 (default 1 for SPI2) Bit 7 is 1 to use I2C bus 2, 0 to use I2C bus 1 (default 0 = bus 1) The I2C bus is used to address an EEPROM in which the Ethernet interface's MAC address and other data are stored. If <b>select</b> is not present, or set to <b>nil</b> or <b>0</b> , the default setting is used.
address	use DHCP	A non-zero integer or string that represents a valid IP address sets the IP address of the interface explicitly. If this parameter is not present, <b>nil</b> or <b>0</b> , DHCP is used to set the IP, DNS and default gateway addresses A string that does not represent a valid IP address is used for a host name or client ID in a DHCP request
DNS	0 or set by DHCP	If Parameter 2 specifies the IP address of the Ethernet interface, this parameter can optionally specify the IP address of a name server for DNS lookups.
DGW	0 or set by DHCP	If Parameter 2 specifies the IP address of the Ethernet interface, this parameter can optionally specify the IP address of a default gateway to locations outside the LAN.

## Examples

```
Make eth Ethernet ; Default hardware, get IP address from a DHCP  
Make eth Ethernet(12) ; Explicit hardware, get IP address from a D  
Make eth Ethernet(Nil, "192.168.1.54") ; Default hardware, explici
```


The last example might be used on an isolated LAN, where there is no name server (DNS) or default gateway (DGW).


## Hardware requirements

The Ethernet object requires the correct hardware to be present. This includes a 256-byte EEPROM device on I2C Bus 1 at address 164.

Only the first 20 bytes of the EEPROM are currently used by the Ethernet object; if you want to use this EEPROM for storing your own data we suggest you use locations **32** and higher to leave some room for unforeseen expansion.

If the hardware is missing or if the EEPROM contents have become corrupted, run time error 12 (Device not found) will result. If you suspect EEPROM corruption, contact us for advice on restoring the correct MAC address to your Ethernet interface.

 See Also: The VM2 Application Board 3 (5922) data sheet for details of Ethernet interface hardware.

 Warning: For typical VM2 hardware that has both memory card and ethernet interfaces, you should avoid leaving a memory card in the holder without making a Venom file system on that card. This would leave the card unpowered, causing data corruption on the Ethernet controller which shares SPI bus 2 connections.

## DHCP (Dynamic Host Configuration Protocol)

This is often used for configuring IP addresses and other parameters automatically. If you have a DHCP server on your network (the function may be included in a simple box like an ADSL router) you can use it to configure the following parameters automatically:

- IP address of the VM2's Ethernet interface
- IP address of the default gateway to destinations outside the LAN
- IP address of a DNS name server

In some networks the DHCP server is linked to a local name server. If a hostname parameter is given when creating the VM2's Ethernet interface, the name is sent to the DHCP server, and should be transferred to the DNS server so that another network user can reach it by name. For example, if the Ethernet is set up with the following command:

```
Make eth Ethernet(nil, "controller1")
```

and you create a web server in the VM2, you can reach it using a web browser and the URL:  
`http://controller1`

## Deferred Address Configuration

In all versions of Venom up to 2012 12 05, the DHCP configuration is attempted when the ethernet interface is created.

In versions of Venom from 2013 onwards, DHCP configuration is deferred until the interface is actually used to send or receive data, and is repeated whenever the link (i.e. the electrical ethernet connection) has gone down and come up again.



## Address

The address message is used for several different purposes within the Ethernet object:

- Get an IP address from a DHCP server on the network
- Set a fixed IP address
- Clear the routing table
- Set a default gateway address
- Set up a gateway for a specific address range
- Specify a name server
- Set up the VM2 as a DHCP server
- Enable Multicasting and received multicast packet filtering

### Get an IP address from a DHCP server

```
Address ('A' [, str hostname]) ⇒ Int
```

(Mnemonic: "Automatic")

This attempts to use DHCP to get or refresh the IP address and gateway and DNS addresses from a DHCP server on the local network.

The value returned is True (1) if address information was successfully obtained from a DHCP server, False (0) if there was no server response. If there was no server response, the IP address of the Ethernet interface remains unchanged.

This message is useful if an ethernet interface needs to be reconfigured from a static to a DHCP-assigned addressing configuration, or if the hostname needs to be changed.

In Venom versions up to 2012 12 05 it was also useful if an ethernet connection did not exist when the interface was created, but has since become available; however in later versions of Venom this now happens automatically when the electrical ethernet connection is detected.

### Address Conflict

It is possible for the DHCP server to offer an address that is already being used by another device. This (rare) condition is detected and the server will be asked for a different address until an acceptable offer is received.

### Set a fixed IP address

```
Address ('I' [, ipaddr addr [, ipaddr mask]]) ⇔ int
```

**addr** is an IP address to assign to the Ethernet interface.

**mask** is a mask to define the network part of the address.

The value returned is the IP address of the interface.

If mask is not specified a default value is assigned according to the standard Internet address classes convention:

Address Range	Mask
0.0.0.0 – 127.255.255.255	255.0.0.0
128.0.0.0 – 191.255.255.255	255.255.0.0
192.0.0.0 and higher	255.255.255.0

The value of addr is determined by the range of addresses used by the network. In an existing network, a network administrator should assign a suitable address and mask value for the device being attached. For private networks, these will usually be assigned from special ranges of addresses which can safely be used without reference to any internet authority. These are:

Address Range	Mask
192.168.0.x – 192.168.255.x	255.255.255.0
172.16.x.y – 172.31.x.y	255.255.0.0
10.x.y.z	255.0.0.0

In the lists above, all addresses in one network must have the same values in the parts designated by digits, and all must have different values denoted by x,y and z. In general, a 1 bit in the mask corresponds to part of the network address (constant within the LAN) and a 0 bit corresponds to part of the host address (different for each device within the LAN).

If you have previously set a gateway address it will be removed when you assign an IP address to the Ethernet interface. Therefore if you need a gateway, you should set the IP address first, then the gateway address.

#### Address Conflict

If you use this method to attempt to assign an address that is already being used by another device on the network, the address conflict is detected, with the following results:

- A run time error, unless [ErrorAction](#) was used to prevent this.
- The Status message will return a value of 3
- The link will not work (no packet will be transmitted)

#### DHCP

If you created the Ethernet object specifying DHCP, you should not need to use this message to set the IP address as the DHCP server will have sent an IP address to the VM2. It may be useful, however, to check the value of the IP address is non-zero to verify that DHCP configuration was successful - failure will result in an IP address of zero.

### Clear the routing table

**Address ( ' C ' )**

(mnemonic “Clear”)

This removes all IP routing table entries for this interface.

## Set a default gateway

**Address('D', addr)**

(mnemonic: "Default")

This sets a default gateway. All packets not routeable via an explicit route will be sent to this gateway in the LAN for forwarding to another network. If the **addr** parameter is missing, the current gateway address is returned, or 0 if there is none.

If you have previously set a gateway address it will be removed by assigning an IP address to the Ethernet interface. Therefore if you need a gateway, you should set the IP address first, then the gateway address.

## DHCP

If you created the Ethernet object specifying DHCP, you may not need this message, as the DHCP server should have sent the gateway address.

## Set up a gateway for a specific address range

**Address('G', addr, subnet [,mask])**

(mnemonic: "Gateway")

- addr** is the address of the gateway, which must be local to the network.
- subnet** is the base address of the range of addresses to be routed via that gateway.
- mask** if specified, sets an subnet mask to define the range of addresses. If not specified a default mask is used following the rules for **eth.address('I')**.

This sets up a gateway through which to route packets with addresses in a given range. It will not often be necessary to provide a gateway address for a range of addresses like this as most networks will use a default gateway for everything not local to the LAN, but this capability allows more complex setups when required. Note that the VM2 cannot obtain information on such networks by DHCP; you have to specify it yourself.

## Set Up Multicast Transmission

Multicast addresses can be used with UDP to send a single packet to a group of devices. The sender does not need to know which devices are receiving a multicast packet. Venom does not currently support any protocol (such as IGMP) for "subscribing" to a multicast group, so routing of multicast packets has to be controlled by manually configuring routers and switches, or more

simply but less efficiently by using cheap network switches that treat multicast packets as broadcast, copying them indiscriminately to every port.

**Address('M', Int Enable)**

If enable is 0, all multicast reception and transmission is disabled (the default state when the ethernet object is created)

If enable is 1, multicast transmission is enabled on any IP address in the range 224.0.0.0 to 240.255.255.255. This is the address range allocated by the IANA (Internet Assigned Numbers Authority) for all multicast addressing using IPv4.

## Set Up Multicast Reception

**Address('M' [, low [, high]]) ⇒ Int**

<b>low</b>	String or integer representing IP address. Sets lowest address for incoming multicast packets
<b>high</b>	String or integer representing IP address. Sets highest address for incoming multicast packets (default = <b>low</b> )

This allows the ethernet interface to receive selected multicast IP packets.

The range for **low** and **high** is 224.0.0.0 to 240.255.255.255, which is the address range allocated by IANA for all multicast addressing using IPv4. Addresses outside this range will cause a run time error (except for the integer values 0 and 1 described above)

Because the mapping of IP multicast addresses to Ethernet addresses only uses the low 23 bits, some multicast messages outside the range specified may still be received. Received multicast UDP packets should be filtered by inspecting the value returned by [udp.Address\(1\)](#).

## Selection of Address Range for Multicasting

The addresses in the IP multicast range are further subdivided for specific purposes, including some assigned to specific individuals or organisations. See <http://tools.ietf.org/html/rfc5771> for full details. In practice you are most likely to want to use the "Administratively scoped" range described in RFC2365 (<http://tools.ietf.org/html/rfc2365>) which uses addresses 239.0.0.0 to 239.255.255.255. Packets with addresses in this range are expected to be kept within a defined administrative boundary by configuring routers to block them from crossing that boundary, which means you don't have to register the addresses you are going to use with any organisation.

In practice, it is often actually safe to use any multicast address within a LAN, but ask the network administrator if you are not sure.

## Specify a name server

**Address('N', addr)**

(Mnemonic: "Nameserver")

Sets the address of a DNS nameserver to resolve domain names. If **addr** is missing, the current nameserver address is returned as an integer.

This message is the equivalent of sending the message **Find(0) := addr** to an IP object.

If you need to use domain names, you should set the address of a DNS server known to be accessible to your network. This will either be a server on your LAN itself or that of your Internet Service Provider.

## DHCP

If you created the Ethernet object specifying DHCP, you may not need this message, as the DHCP server should have sent the name server address.

## Get the MAC address

**Address('H', addr) ⇒ Int**  
**Address('L', addr) ⇒ Int**

(Mnemonic: "High", "Low")

These return the high and low 3 bytes respectively of the Ethernet interface's Ethernet address (MAC address).

You cannot set the MAC address this way, only read it.

As the MAC address is a 48 bit number, it needs two Venom (32 bit) integers to represent it.

The MAC address is stored in a EEPROM and programmed during manufacture.

## Set the MAC address

*As MAC addresses are globally unique they should not be set without great care!*

However, if you construct your own Ethernet circuit, and you have a MAC address you can use, you can set the MAC address for your hardware like this:

Call **Make** for Ethernet, but add **\$40** to the 'hardware select' parameter and specify the high and low parts of the MAC address as two 24-bit numbers in the next two parameters:

**Eth := New Ethernet(\$40 + \$0c, MAC\_H\_24, MAC\_L\_24)**

If you need to buy MAC addresses we can supply them from our pool.

## VM2 as a DHCP server

```
Address('S', low_addr, high_addr [, hostname [,
domain]])
```

(mnemonic: "Server")

<b>low_addr</b>	Int or string: lowest IP address in DHCP pool
<b>high_addr</b>	Int or string: highest IP address in DHCP pool
<b>hostname</b>	The VM2's own host name (default "vm2")
<b>domain</b>	The VM2's domain name (default none)

This sets up the VM2 as a DHCP server. The low and high addresses should have the same network address as the Ethernet interface itself, but the Ethernet interface address should NOT be within the range specified.

e.g. if the Ethernet has an address 192.168.1.1 it is a class C address (see above) with network address 192.168.1.0

The DHCP low and high addresses must be in the form 192.168.1.x but not include 192.168.1.1

e.g.

```
MAKE eth ethernet(nil, "192.168.1.1")
eth.address('S', "192.168.1.100", "192.168.1.199")
; VM2 can now be reached as "vm2" by any connected DHCP client.
```

In this mode, the VM2 also acts as a limited DNS name server: a request for its own host name will return its IP address.

For example with the default values, this means the fully qualified domain name is "vm2".

If the domain was specified as "mynet", the domain name would be "vm2.mynet".

## Applications of DHCP Server Mode

DHCP server mode is useful if you have a VM2 in the field and want to be able to connect to it intermittently with a visiting laptop computer, when the Ethernet port is not part of a local network so there is no other DHCP server. A CAT5 Ethernet cable connects the laptop direct to the VM2, and the VM2 will configure the network automatically. (Old laptops might need a "crossover" cable; modern laptops can use both crossover and straight cables as they use Auto-MDIX). On the laptop you can use a web browser or ftp client to connect to the VM2 with the hostname configured in the VM2, default "vm2".

## Connect

**Connect**  $\Rightarrow$  **Int**

(New from 2013 onwards)

When an ethernet interface has been created, no address configuration (i.e. DHCP or checking for conflict with a static IP address) is done until the ethernet is used for sending or receiving data.

This message waits for the physical link to come up (up to 3 seconds), then runs DHCP configuration if the link is newly up, and returns true (1) if successful.

If the link was already up and configured, it returns immediately.

A returned value of 0 means either the link is disconnected (e.g cable unplugged) or for an ethernet that uses DHCP, no DHCP server could be found.

This message enables you to verify that a connection is available, in case you want to take alternative action if it has failed.

cf. [Status](#) which always returns quickly but only tests the electrical connection of the ethernet cable.

## Count

**Count**(int n [, 0])  $\Rightarrow$  int

The value returned is a count of events as specified by n:

n = 0 Packets received

n = 1 Packets transmitted

n = 2 Missed received packets

n = 3 Transmit errors

n = 4 Collisions

If a second parameter with a value of zero is present, the counter is reset to zero.

## Debug

**Eth.debug**(0)  $\Leftarrow$  Int

Turns on (1) and off (0) packet level debugging.

With packet debugging on, a summary of every packet sent or received is shown on the serial terminal.

This can be useful for debugging network problems.

**Eth.debug(0)**  $\Leftarrow$  Int **n** ( $n > 1$ )

If  $n > 1$ , selective debug display of packet types is enabled. See [WiFi.Debug](#) for a table of bitmapped values corresponding to packet type.

**Debug(1)**

Shows the contents of the ARP table, which is the mapping of IP addresses to MAC addresses on the network.

It will list MAC and IP addresses of any network node to which the Ethernet interface has sent a packet recently.

## ErrorAction

**ErrorAction**  $\Leftarrow$  Int

Assigning a value of 1 to **ErrorAction** suppresses a run time error when a static IP address assignment results in an address conflict. It is effectively a promise that the condition will be checked by a Status message.

## Off

**Off**

Puts the ethernet controller chip into a low-power mode, using a power management feature in the chip. The ethernet interface cannot send and receive data in this state, but its configuration (e.g. IP address settings for filtering incoming packets) is retained so will resume operation when it gets an [On](#) message.

If the ethernet is not actually being used, this saves significant power consumption.

The chip is also put into low power mode when the ethernet object is killed, either by an explicit **Die** message or exit from a procedure where it was declared **Local**.

Note that the ethernet chip consumes power even if an ethernet object has never been created in software, so in power-sensitive applications it is actually worth creating a temporary ethernet object solely for the purpose of putting the chip into low-power mode:



```
New Ethernet(0, 12345).Die
```

(where 12345 is any non-zero dummy IP address)

Normal operation can be restored with the [On](#) message.

## On

### On

Turns the ethernet chip's internal power back on after it has been turned off with an [off](#) message.

## Status

**Status**  $\Rightarrow$  Int

### Venom version 2015 12 01 and earlier

This returns **true** (1) if the Ethernet link is active (i.e. cable connected and carrier detected), **false** (0) if not.

In 2016 and later version of Venom, that function is provided by [Valid](#)

### Venom versions from 2016 onwards

In 2016 and later versions of Venom, Status will return 0 for a good connection ([Valid](#) = 1) or an error code which indicates why a connection is unusable.

Status value	Meaning
0	Normal operation
1	No carrier (Ethernet cable connection problem)
2	DHCP failure
3	IP address conflict: another device is using the address assigned to this interface.

## Valid

**Valid**  $\Rightarrow$  Int

This returns **true** (1) if the Ethernet link is active (i.e. cable connected and carrier detected), **false** (0) if not.

This message is only available in versions of Venom released in 2016 and later.  
For earlier versions [Status](#) has identical functionality.

cf. [Connect](#).

## Time

**Time**  $\Rightarrow$  Int

This returns internet time as Venom time value, which is the time in seconds since midnight GMT on 1 January 1990, if a suitable time server can be contacted.

It behaves in exactly the same way as [udp.Time](#), and is provided to save the inconvenience of creating a [UDPProt](#) object purely to get the internet time.

## Example

Synchronise VM2 real time clock with internet time

```
make eth ethernet
clock.Time := eth.Time
```

## PRINT

**Print** <Ethernet> [: Int mode]

Printing an Ethernet object will display its MAC, IP address and related information in the following format:

```
eth0:
MAC   = 00:50:c2:3d:04:4b
IP     = 172.16.1.252
DNS    = 172.16.1.148
GW     = 172.16.1.199
host   = vm2
```

If a colon operator is present, it can have the following effects:

n	Description	Examples
0	Print information as above	<i>as above</i>
1	Print MAC address only	00:50:c2:3d:04:4b
2	Print IP address only	172.16.1.252

3	Print DNS server address	172.16.1.148
4	Print gateway address	172.16.1.199
5	Print host name(and domain if exists) if specified when setting up as a DHCP client or server, or the string "<no hostname>" if not defined.	vm2 vm2.mynet <no hostname>

## FileSystem

Venom 2 supports file systems on many different media. File systems can be created on:

- On board RAM. Up to 900k bytes are available for this purpose, if the memory is not needed for other purposes by the application. On a VM2 with a battery fitted, the RAM retains data when the power is off. RAM files are very fast, both reading and writing.
- On board Flash, with a capacity of about 7Mbytes. This retains its data without power. Reading is very fast; writing can be slow and excessive write cycles could wear out the flash memory. The Flash File System is also accessible as a 'Mass Storage Device' via USB - see [USB access to Flash File System](#).
- Memory Cards (SD or SDHC) with capacities up to 32Gb. Both reading and writing are slower than other media, and the media may be subject to wear from excessive writing. Memory cards are removable and so may be used to transfer data between a VM2 and other devices.
- External USB Mass Storage Devices - e.g. USB Flash Drives, or other USB disc drives. Flash-based drives have similar properties to memory cards.

The Venom file system is compatible with the Windows FAT12, FAT16 and FAT32 file systems, and has full support for long file names and subdirectories. Full Unicode in filenames is not supported, but 8 bit characters representing the first 256 code points of Unicode (also known as ISO 8859-1 or Latin-1) can be used.

## Summary of messages

**Make**  
**Address**  
**Adjust**  
**Connect**  
**Copy**  
**Count**  
**Debug**  
**Done**  
**Find**  
**Empty**  
**Flush**  
**Free**  
**Length**  
**Name**  
**Open**  
**Remove**  
**Reset**  
**Status**  
**Time**  
**Valid**  
**Print**

## Creation

There are four different file system media available on the VM2 and each has a slightly different creation syntax. Please see:

- [SD card file system](#)
- [Internal RAM file system](#)
- [Internal Flash file system](#)
- [External USB file system](#)

## SD Card Creation

```
Make <object> FileSystem(str type, [Int unit[ ,Int  
partition[, Int cachesize]])
```

<b>type</b>	" <b>SDI2C</b> " specifies an SD card in a socket with I2C interface for card detection hardware and power control, such as that fitted to the 5902 Application Board 2.
-------------	--

	"SD" specifies a fixed SD card with no IC2 "card present" or write protect de such as the micro SD card on the 5922 Application board.
<b>unit</b> (optional)	Hardware selection information. For standard hardware configurations leave this parameter out, or use the value 13. For other hardware configurations please contact us. <b>Prevent file system check on creation</b> If you set bit 7 e.g. by using <code>\$80 + 13</code> , the initial detailed self check of the file system is skipped (see <a href="#">Valid(0)</a> and <a href="#">Valid(1)</a> .) This saves time with high capacity cards with a large volume of file data, but is slightly dangerous as file system corruption is not detected and repaired when the file system is started up.
<b>partition</b> (optional)	(default: 0) selects a partition if the device has a partition table. SD cards are usually formatted with a partition table and a single partition which is selected as partition 0.
<b>cachesize</b> (optional)	Specifies the amount of memory in bytes to use for block cacheing. Default 20k; minimum 5k, maximum 100k.

- MMC, SD and SDHC cards and FAT12, FAT16 and FAT32 formats are supported, allowing card capacities up to 32GB.
- The SPI protocol is used: this should be supported by all the above card types.
- For removable types, if a memory card is not present, a file system framework will still be created, but a card must be present before any I/O is performed or a run time error will then result.
- A file system must already exist on the memory card; this driver cannot create a new one.

#### Example

```
Make fatfs FileSystem("FSD")
Make fatfs FileSystem("SD")
```



Warning: For typical hardware that has memory cards and ethernet or USB host connections, you should avoid leaving a memory card in the holder without making a file system on that card. This would leave the card unpowered, causing data corruption on the Ethernet and USB host systems which share SPI bus 2 connections.



When creating an SD card file system on the Application Board 3 (5922) you will need to first switch on the card's power supply. See the 5922 datasheet for details.

#### More than one SD card



For systems with more than 1 fixed SD card, you need to call [System.Debug](#)(22, unit1, unit2) to pre-initialise all the cards before making a filesystem on any of them.

#### Example

```
Debug(22, 13, 15)
```

```
Make fs1 FileSystem("fsd", 13)
Make fs2 FileSystem("fsd", 15)
```

### RAM FS Creation

```
Make <object> FileSystem("RAM", Int size)
```

<b>"RAM"</b>	Indicates that we are using a RAM disk for storage
<b>size</b>	Size of RAM disk in bytes. Minimum 3072 (3K), Maximum 921600 (900K). For predictable results, make this a multiple of 512.

### Example

```
Make ramfs FileSystem("RAM", 1024 * 10)
```

### Rough Guide to allocating RAM disk space

Table of approximate filing system overheads versus RAM disk size.


RAM disk size	Overhead	Root Directory Entries
3K - 29K	2K	32
30K - 79K	3K	64
80K - 346K	4K	96
347K - 688K	4.5K	96
689K - 900K	5kK	96


#### Notes:


- File data is allocated in 512 byte blocks, so allow extra space for "slack" in unfilled blocks at the end of files.
- Space for the root directory is fixed for a given RAM disk size, as shown in the table.
- A file may use more than one directory entry - see [Root Directory Limitation](#)

### Bug alerts

Both the RAM disk and the Flash File System use persistent data stored in non-volatile RAM. Because of this some particular limitations apply to these objects:

 After a reset, if you recreate the RAM disk with a different size then the old RAM disk will be overwritten and a new, empty RAM disk of the new size will be created. All your old files will be lost!

 If you need to create both a RAM disk and a Flash File System, you must create the RAM disk first (after a Venom Run or Reset).

 You should not try to create a RAM disk or a Flash file system more than once after a Reset, as this leads to undefined behaviour.

The best way to ensure that all of the above conditions are met is to create the RAM and Flash filing systems within the init procedure, and use **Run** (or **F10** in VenomIDE) to run your code.


### Flash FS Creation

Most VM2 controllers\* have an 8MB flash memory built in, 7MB of which is reserved for a file system. The Flash File System may be created in either Read-Write mode, or in Read-Only mode. Read Only mode uses much less RAM.

### Read-Write Flash Filesystem

**Make** <object> **FileSystem**("FLA" [, Int **cacheSize**])

"FLA" or "FLASH"	Indicates that a Flash File system is to be made
<b>cacheSize</b> (optional)	Default: 70k (70 * 1024) - recommended. Minimum: 5k (5 * 2014) Maximum: 100k (102400) Any non-zero value outside the limits results in default cache size. <i>Note: A cacheSize of 0 indicates Read-Only mode - see below.</i>

 The actual memory used for cacheing is the specified size + 64k, so the default allocation uses 133k of memory.

Using less than the default size will affect performance and wear out the flash memory faster if your application does a lot of file writing.

Using more than the default size may help performance if you have many files open for writing, deeply nested directory paths or directories with many files in them.

### Example

```
Make ffs FileSystem("FLA")
```

## Read-Only Flash FileSystem

**Make** <object> **FileSystem**("FLA" , 0)

If your application doesn't need to write to the flash file system, this option saves you 133k of memory compared with the default read-write file system. Any attempt to create or delete a file, or to write to an existing file will give run time error 7 "Write to read-only item".

### Example

**Make** **ffs** **FileSystem**("FLA", 0)

### Notes



The root directory has 512 directory entries and file data is allocated in 1k clusters in a standard FAT16 configuration.



The VM2L doesn't have an on-board Flash memory, and so can't support an on-board Flash file system.



Note that if you need to create both a RAM disk and a Flash File System, you must create the RAM disk first. Please see [here](#) for more information.

### USB FS Creation

**Make** <object> **FileSystem**("USB" [, Int **partition**, [, Int **cachesize**]])

<b>"USB"</b>	(not case sensitive) Indicates that we are making a file system for a USB device
<b>partition</b> (optional)	(default: 0) selects a partition if the device has a partition table. SD cards are usually formatted with a partition table and a single partition which is selected as partition 0.
<b>cachesize</b> (optional)	Specifies the amount of memory in bytes to use for block cacheing. Default 20k; minimum 5k, maximum 100k.

This will access the file system on a USB mass storage device (e.g. flash drive or digital camera) connected to the host connector (USB Type A) on the 5922 Application board or similar hardware using a MAX3421E chip on SPI bus 2. Currently there is no support for USB hubs or any other kind of USB device, so the device has to be plugged straight into the USB Type A socket with or without a cable.

The filesystem behaves like a removable SD card, including suspending the device when the cache is clean and there are no files open. On most removable storage devices this means an



LED on the device will be illuminated while busy and off when it is safe to remove.

## Example

```
Make usbfs Filesystem("USB")
```



Warning: For typical hardware that has memory card and USB host connections, you should avoid leaving a memory card in the holder without making a file system on that card. This would leave the card unpowered, causing data corruption on the USB host which shares SPI bus 2 connections.

## Address

**Address** (str filename)  $\Rightarrow$  Int

**Address** returns the address in memory of the data in a file, if possible.

**filename** is a text buffer or string containing the name of a file.

For certain applications like graphics, fonts or audio output, direct memory addressing streamlines the use of data from a file as it does not have to be copied to an area of memory first, and the file does not even need to be explicitly opened.

**Address** will return 0 if no address for the file's data exists. This will be for one of the following reasons:

1. The file system is not memory mapped (e.g. SD cards)
2. The file's data is not [contiguous](#) in memory
3. The file doesn't exist or is a directory

## Adjust

**Adjust**  $\Rightarrow$  Int

This causes the file system to reorganise its data so that all the free space is allocated in one block at the end of the storage medium or memory.

Its purpose is to ensure that any file created afterwards is guaranteed to have all its data in one contiguous block, as long as files are added one at a time.

Note that this is not quite the same as defragmenting. Files whose data blocks are interleaved on the storage media will remain in that condition.

The value returned is the number of clusters that were moved. If 0 is returned, then no changes were made.

This message will work on any filesystem, but is only useful on the onboard flash system where certain files need to be allocated all in one block,

for example, System [Protect\(2\)](#) and [Protect\(3\)](#) for OS and application updates, and GraphicsLCD [Bitmap](#) and [Fontdata](#) when used with files

The Adjust process generates a "percentage done" value which can be accessed from another task with the Filesystem [Done](#) message.



Note: all files in the FileSystem must be closed before using Adjust else there will be a runtime error.

## Connect

**Connect**(Int **connectstate** [, Int **port**])

**connectst** 1 or **True** = enable USB access to Flash File System

0 or **False** = wait until current USB access is complete then disable USB

2 Put fileSystem into 'file manager' mode

**port** For connectstate 2, specifies the serial port (default 1)

## Connecting to USB

On the Flash File System, this message allows an external host access to the file system via a USB connection such as the USB Type B connector on the Application Board.


You can continue to use the file system in Venom, but if your Venom program writes to the file system the USB host will see a "drive not ready" condition. This will persist until you send the filesystem a [Flush](#) message, after which the USB host will see a "drive ready" event and re-read the memory. This is necessary because the host has no other way of knowing that the data on the filesystem has changed.

While the USB host is writing to the file system any writes from Venom are held up until the USB write has finished. Host writing is considered finished after an idle period of 2 seconds.

In general, it is better not to attempt concurrent writing to the flash file system from both ends of the USB link.

Concurrent reading of files presents no problems.

The Memory card and RAM file systems are not currently available on USB.

 Note: do not use **Stop All** while Connect is set - a system task is created to deal with USB transactions and **Stop All** will stop that task.

## Connecting to Serial Port in 'File Manager' mode

**connect (2)**

**connect (2, 3)** ; Connect to port 3

This puts the file system into a mode where it receives commands via a serial port. These enable files to be transferred to and from the VM2, and some other file system operations.

File manager mode has a disconnect command which causes the **Connect** message to return. The disconnect command is a single byte value of 3, which you can send by pressing Ctrl+C if the VM2 has been left in file manager mode. The file manager uses a binary protocol and cannot be directed from a terminal.

Support for transferring files in this way is available in VenomIDE.

Please contact us for documentation on the details of the serial protocol, and a Windows command line program that uses this protocol, both exe file and source code.

## Copy

```
Copy(Str srcpath, Str destpath [ ,fs destfs])
```

**srcpath** The source path and file name

**destpath** The destination path and file name

**destfs** The destination file system, if different from the source file system

This copies a file or files from one location to another, in the same or a different file system.

If **srcpath** specifies a directory, the whole subtree is copied i.e. files and subdirectories

Any destination files must not already exist, or a run time error will result. If you want to overwrite existing files you must remove them first.

You cannot copy a directory to a subdirectory of itself and an attempt to do so will result in a run time error.

The file copying process generates a "percentage done" value which can be accessed in another task with the Filesystem [Done](#) message

The root directory can be specified as a source or destination, using **" / "**.

**Example**

```

MAKE ram FileSystem("RAM", 100*1024)
MAKE sd FileSystem("SD")

if ram.find("dir1") IsFalse
    sd.copy("dir1", "dir1", ram)

ram.remove("dir1-backup")
ram.copy("dir1", "dir1-backup")

```

**Example of copying complete filesystem**

This code copies the total contents of a USB stick into the Flash File System.

```

Make ffs FileSystem("FLA")
Make usbFs FileSystem("USB")
ffs.Remove("*")
usbFs.Copy("/", "/", ffs)
ffs.Flush ; Not strictly necessary.

```

**Count**

**Count**  $\Rightarrow$  Int

Returns the number of files in the root directory.

**fs.Count(string pattern)**  $\Rightarrow$  Int

Returns the number of files matching the pattern in one directory.

**pattern** Specifies a subdirectory and optionally also a filename or wildcard pattern. It can be a text buffer or fixed string.

Return value: the number of files in a directory.

Both plain files and subdirectories are counted, but not files within the subdirectories.

If the pattern specifies a directory all entries in that directory are counted.

If the pattern specifies a plain file name or a wildcard, the number of matching names is returned.

**Examples**

```
Print fs.Count, CR
```


Print the number of file in root directory

```
Print fs.Count("tmp"), CR
Print fs.Count("tmp/*"), CR
```

Print the number of files in directory "tmp". The above two examples have identical effect.

```
Print fs.Count("tmp/*.txt"), CR
```

Print the number of files in "tmp" whose names end with the characters ".txt"

 See also: [Wildcards and File Name Matching](#)

## Debug

### Debug

Lists debug options available.

```
Debug (Int a[, Int b]) ⇒ Int
```

## Memory Cards (SD/SDHC)

Parameters	Action	Applies to:
(1, n)	Read block n from device and display on screen	all
(2, 0) or (2)	Display file system information: size, cluster size, position of FAT and root directory etc.	all
(2, 1)	Show boot sector	all
(3, n)	Show details of FAT entry n	all
(4, n)	Show where block n is used (FAT, root dir, file etc.)	all
(6)	print status with explanatory text, e.g. stat=> 19 MEDIA CACHEWRITE FILES_OPEN	all
(10, 1)	Return no. of blocks in cache	SD, Flash and USB only
(10, 2)	Return memory used by cache	
(10, 3)	Return number of cache hits	

(10, 4)	Return number of cache misses	SD only
(12, 1)	Return no. of block reads from card	
(12, 2)	Return no. of block read errors	
(12, 3)	Return no. of block read resets	
(12, 4)	Return no. of block writes to card	
(12, 5)	Return no. of block write errors	
(12, 6)	Return no. of block write resets	

### Notes on block read/write statistics for SD cards:

If a read or write operation fails, an error is counted and the operation is retried several times.

Every two retries, the card is reset (re-initialised) and a "read reset" or "write reset" is counted.

We have found that some cards occasionally get a read error which recovers on the second attempt, and a rare instance of a write failure that required a reset. These count values may help to indicate the reliability, compatibility or state of wear of a particular card.

The counts are reset when the file system is created or recreated, including after sending a reset message to the file system or if the card is removed and replaced.

Use of Debug with other parameters is discouraged and subject to change, as these options are intended for internal use by the Venom developers.

## Done

**Done** ⇔ Int

If another task is performing a FileSystem [Copy](#) or [Adjust](#) operation, this message will return an estimate of the percentage of that operation completed. The returned value is therefore an integer in the range 0 - 100.

For example this could be used to show a bargraph indicator of progress on a graphics display. It is particularly useful on flash and SD card based file systems where the operations in question can sometimes take several seconds to complete.

Because of task timing uncertainties, it's possible to read a **Done** value of 100% before the task has actually started. To avoid this, either wait for a short time before checking the Done value, or preset the **Done** value to 0. Examples of both methods are shown.

### Example Using Wait

```
Start fs.Copy(source, dest)
Wait 10 ; allow time for task to get started
```

```
While fs.Done < 100
[
    printf("%u%% done\n", fs.Done)
    wait 1000
]
```

### Example Using Presetting the "Done" value to 0

```
fs.Done := 0
Start fs.Copy(source, dest)

While fs.Done < 100
[
    printf("%u%% done\n", fs.Done)
    wait 1000
]
```

## Find

**Find**(string name) ⇒ Int

**name** is a string or text buffer.

Returns:

- 0 if the named file does not exist
- 1 if the name matches a plain file
- 2 if the name matches a directory

The name can also be a wildcard specification, testing for the existence of at least one matching file if it returns non-zero.

 See also: [Wildcards and File Name Matching](#)

## Empty

**Empty**

This message wipes the FAT and root directory of the file system, leaving it empty. It is like a DOS/Windows "quick format".

Use of this message guarantees restoration of a valid file system if it has become corrupted by a program crash, card removal or power failure while the file system was in use, at the cost of loss of all files.

If you want to simply remove all the files from a file system then [Remove\("\\*"\)](#) is much faster.

## Flush

```
Flush ([Int fullflush])
```

### All file systems

Any open files have their directory entries updated with the file size and current time and date, as if each open file had been closed and re-opened or sent the [Update](#) message.


### Memory Cards and USB File Systems

All data in cache waiting to be written to the card is written to the card.

Any parameter is ignored.

### Flash File Systems

If optional parameter **fullflush** is present and non-zero, all data in cache waiting to be written to the flash memory is written to flash memory. This is not often required as the cache RAM on a VM2 is non-volatile if a battery is fitted.

 See also: Application Note "Data Logging with Memory Card File System" available from the our website, which discusses the issues of file data integrity and memory card wear. The issues are applicable to the on board Flash file system too.

## Free

```
Free ⇒ Int
```

Returns the amount of free space in the file system, in Kilobytes (1KB = 1024 bytes)

It is calculated from the number of FAT entries marked free.

On higher capacity FAT16 memory cards this can be quite slow, as it works by counting free blocks, of which there may be thousands.

On FAT32 file systems, a running estimate of free space is used, which is faster but could become inaccurate under fault conditions, but which is re-created when the filesystem is started



or when the [Valid\(n\)](#) message is used.

**Free(Int size) ⇒ Int**

**size** is the required amount of free space in Kilobytes. (1KB = 1024 bytes)

Returns True (1) if the free space in the file system is equal or greater than **size**, False (0) if the free space is less.

On high capacity FAT16 cards this can be significantly faster as the counting of free blocks stops as soon as a True condition is reached.

### Example

```
If fs.free(100) IsFalse
  Print "File system space is less than 100k", CR
```

### Length

**Length(str Name [, int units]) ⇒ Int**

Return the length in bytes of the file(s) named.

If the name is a wildcard pattern, returns the total size of all files whose names match, including any files in subdirectories.

If the name is a directory, returns the total size of all files in the directory and its subdirectories.

If no matching file exists, returns 0.

If a **units** value is present, it changes the units for the size value returned:

Units selector	Returned result
'K' or 'k'	Kilobytes (1024 bytes)
'M' or 'm'	Megabytes (1024 * 1024 bytes)
'G' or 'g'	Gigabytes (1024 * 1024 * 1024 bytes)
<i>any other</i>	bytes



It is possible for the length of a file or the total length of several files to exceed the capacity of a Venom 32 bit signed integer. The maximum value that can be represented is 2 Gigabytes. If the length requested exceeds this, an erroneous value will be returned without warning. Use of the units selector avoids this problem.

## Name

**Name**(Str **old**, Str **new**) ⇒ Int

Renames or moves file from old directory and name to new. Returns logical true if successful, meaning that a file with the old name existed. If the new name is invalid, a “File Access” runtime error (error no. 24) occurs.

**old** and **new** can be strings or text buffers.



The file must not be open for writing when renamed in this way. The programmer is responsible for this - the condition is not detected automatically by the file system.

Note that the full path of the new file name must be specified even if it is in the same subdirectory as before.

The old and new names can be in different directories in the file system, resulting in the file being moved without its data having to be copied. A directory can also be moved to a new location, in which case all files and lower directories under it appear in the same structure under the new location.

A directory cannot be moved to a subdirectory of itself and an attempt to do so results in a run time error.

## Open

**Open**(Str **name**, **type** [, Int **mode**]) ⇒ file object

Opens an existing file for or creates a new file, returning a [file variable](#).

By default an existing file is opened for append i.e. writing to the file adds data to the end.

Parameter	Type	Description
<b>name</b>	string or text buffer	The full path and name of the file to open
<b>type</b>	various	<div><b>Int 8</b>            8-bit unsigned integers</div> <div><b>Int 16</b>        16-bit signed integers</div> <div><b>Int</b> or <b>Int 32</b> 32-bit signed integers</div> <div><b>Float</b>           Floating point</div> <div><b>Char</b>           Characters - i.e. a text file</div> <div><b>'d'</b> or <b>'D'</b>      The file is a subdirectory</div>

<b>mode</b>	Int	default: file opened for append +ve values: limit length of file -1: read only -2: read only, no cache -3: create directory if it didn't previously exist
-------------	-----	---

## File names

Upper and lower case letters and numerals are allowed in the file name, as are all characters with code values over 127 and the following special characters:

! \$ % - \_ + ~ . / # ' @ ( ) { } & , ; = [ ]

The maximum path and file name length is 260 characters.

The case value of letters in the file name is stored unchanged when the file is created new, but the process of matching for an existing file name is not case sensitive.

A file can be created in a subdirectory by specifying its full path. If you do not specify a mode value of -3, the subdirectory must exist first. You can create a subdirectory by using the special type 'd' or 'D' (*see below*).

When specifying a file in a subdirectory, the character '/' is used to separate directory names. This is the Unix convention as opposed to the Windows/DOS convention and is more convenient as '\ ' is used as an escape character in Venom strings.

## Mode Parameter

The optional mode parameter affects the way the file is opened in several ways, described in more detail here.

Mode value	Description
Positive, >= 1024 (max length)	Sets a maximum length for the file. If created this way, writing beyond that length will cause data to be removed from the beginning of the file. This is useful for log files where only recent data is of interest, and in such cases can remove the need for extra housekeeping code to prevent the file system from running out of space.  Note that the <b>mode</b> parameter, when used to specify a length limit, is in bytes and is mainly intended for use with text files. This parameter, if specified, must not be less than 1024. ▲ The actual file size may be considerably bigger than specified as it is truncated to a whole non-zero number of FAT clusters, and on some systems a cluster can be as big as 32k.
-1 (read only)	File is opened read-only. This has the following consequences: <ul style="list-style-type: none"> <li>• You can open more than one read-only instance of a file without getting a run time error</li> <li>• Any attempt to write or modify the file will cause a run time error.</li> </ul>

- If the file did not exist, attempting to open it read-only will result in a run time error.
- 2  
(no cache)      File is opened read-only (see above) but in a mode that does not use the file system cache for the file's data blocks. This can lead to faster performance in cases where the file size is bigger than the cache, by saving time wasted searching the cache for blocks that will never be there, and by leaving more useful FAT and directory blocks in the cache to speed up the next file system operation. This mode is only recommended when reading the file linearly, not if you are intending random access with the **Element** message.
- 3  
(create directory)      Any required subdirectory (including nested subdirectories) will be created automatically if needed.

### Example

```
a := fs.open("abcde.txt", Char) ; a text file in the root directory
b := fs.open("/data/numbers", Float) ; a float file in the subdirectory
```

Empty file is created with read and write pointers at 0.

### Errors When Opening a File

Following are some possible reasons while opening a file could fail:

- Invalid file name - must only contain valid characters. See above.
- Too many slots in root directory. This can happen when you create many files with long names, especially in a RAM disk where root directory size is limited. see [Creation](#) (RAM disk) and table of root directory sizes. For solutions to this problem, try (a) fewer files, (b) shorter file names, (b) putting files in a subdirectory where directory size is limited only by total file system space or a maximum of 65535 directory slots.
- Subdirectory does not exist. When creating a file in a subdirectory, you must create the subdirectory first if it did not exist or use a mode value of -3.
- Trying to open an existing plain file as a directory or *vice versa*.
- Opening in read-only mode and file did not exist.
- File system full.

## Remove

**Remove**(str **pattern**) ⇒ Int

**pattern** is a string or a text buffer.

Files with names matching the pattern are deleted.

If the pattern matches a directory, the directory and all its contents, including subdirectories, are removed

The value returned is the number of files removed in the operation.

A run time error results if you attempt to remove a file that is currently open.

### Example

```
-->Print fs.remove("temp*"), " files deleted", CR
      3 files deleted
-->Print fs.remove("data/logs") log files deleted", CR
      10 files deleted
-->
```

(it is assumed that **/data/logs** is, for example, a directory containing 9 files)

### Removing all files

To remove all the files in a filesystem use the wildcard **\***.

```
fs.Remove("*")
```

## Reset

### Reset

This performs the same actions as when the memory card or other removable media are removed and re-inserted (or a different card inserted)

- Any open files are forced closed and become "dead" objects
- The memory used by the file system is freed
- The file system is recreated by detecting and reading the card or media

For RAM disks, only the first of the above actions is applicable.

## Status

**Status**  $\Rightarrow$  Int

This returns various separate pieces of information about the file system as bits of a single integer value:

Bit	Value	Meaning	Applicable File Systems
0	1	Valid media and file system	All
1	2	Media write protected (e.g. SD card 'lock' tab)	Removable Memory card and flash
2	4	media state changed (removed or replaced) <sup>1</sup>	Removable Memory card and USB
3	8	Data written to cache has not been copied to media yet (see <a href="#">Flush</a> message) <sup>2</sup>	Memory card, flash and USB
4	16	At least one file is open	All
5	32	File system connected to USB host ( <b>Connect (1)</b> state)	Flash, under control of external host by USB
6	64	USB active (host activity detected) <sup>3</sup>	Flash, under control of external host by USB

Notes:

<sup>1</sup> Note that the "Media state changed" bit will only be set in the first Status message after a memory card has been removed or inserted.

<sup>2</sup> If the flash file system has been written by Venom code, the data remains in cache until a **Flush (1)** message has been sent to the filesystem. In contrast, if the Filesystem has been written via USB, the cache is flushed (and this flag cleared) after a 2 second period of inactivity, which means this flag bit is a useful indication that an external write to the system has finished when the flag is 0.

<sup>3</sup> In practice, this provides a useful indication that the USB cable is physically connected as the bit will be set whether or not data is being transferred, unless the device has been suspended by the external host.

**Status (1)**  $\Rightarrow$  Int

This form of the Status message returns a number indicating the media type of the file system:

- 0     RAM disk
- 1     Memory Card
- 2     On Board Flash

### 3 USB Mass Storage Device (Flash Drive)

**Status (2)**  $\Rightarrow$  Int

This returns the number of currently open files as an integer.

## Time

**Time (str name)**  $\Rightarrow$  Int

**name** String or text buffer holding name of file

Returned value is time in seconds, compatible with [DateTime](#) and [RealTimeClock](#) objects.

The value corresponds to the time and date when the named file was created or last modified.


If the file does not exist, a value of 0 is returned.

If there was no real time clock hardware present when the file was created or last modified, the result will be 0.

This message allows the creation/modification time of a file to be found without having to open it.

The timestamp of a file cannot be changed this way. If you want to force a change to a file's timestamp, open the file and use the file [Time](#) message.

 See also: File [Time](#) message

 See also:  
[File Close](#)  
[RealTimeClock](#) object

## Valid

### 1. Simple Check for Valid Filesystem

**Valid**  $\Rightarrow$  Int

The returned value is True (1) if the file system is valid. In particular this tests for the presence of

removable media such as memory cards, and requires the file system to have a boot block containing sensible values from which a file system was able to be created.

Examples:

```
If fs.Valid
[
    f := fs.open("myfile.txt", Char)
    ; (code to access files)
]
Else
    Print "No valid file system found!"
```

## 2 Test Filesystem for Internal Consistency and Fix Errors

**Valid**(Int **fullcheck**) ⇒ Int

**Fullcheck** False or 0 : do a quick test for file system corruption (time limit on lost cluster check)

True or non zero : include a full check for lost clusters

Returned value: the number of errors fixed.

### Explanation

This performs a check on the internal consistency of the File System.

It detects and removes invalid directory entries, cross-linked files and lost clusters (blocks which are marked used but not found as part of any file)

On FAT32 file systems, the current free space estimate is recalculated by counting the free clusters.

With large capacity memory cards (e.g. 1GB and higher) it can take a long time to check every single allocation entry for lost clusters and in practice this check is rarely needed, so there are two options:

If **fullcheck** is zero, a time limit of 100ms is set for checking for lost clusters.

If **fullcheck** is non-zero, every cluster is checked. This has been timed at over 30 seconds on a 1GB card with FAT32.

Whenever a filesystem is created, and memory card is freshly inserted or a **Reset** message is sent to a file system, the shorter version of this check is performed, unless it's an SD card and the "skip FS check option" has been set in the **unit** selector (see [Creation](#))

Example:



```
errors := fs.Valid(0)
Print "File system check found/fixed ", errors:1, " errors", CR
```

 See also [Status](#) which returns more information about the file media.

### 3. Check if a String is a Valid File Name

**Valid**(str Name)  $\Rightarrow$  Int

Returns true (1) if the name conforms to the rules for FAT/FAT32 file names.

The result will be false (0) if the string is too long or contains illegal characters.

See [Open](#) message for details of allowed characters.

### PRINT

**Print** <FileSystem> [:string pattern] [: format]

Lists the files matching pattern in various formats dictated by the optional colon and number.

<b>pattern</b>	If this specifies a directory, the contents of that directory are listed.
(default: root directory)	If it specifies a plain file or wildcard, only matching entries are listed.
<b>format</b>	Specifies the listing format
(default: 2)	If it is a string, the '%' characters followed by various special symbols are replaced by information from the directory entry, as listed below.
	If it is an integer, it refers to one of four predetermined format strings as described below.

### Special codes in format string

code	Meaning
%a	DOS attributes 'A', 'D', 'V', 'S', 'H' and 'R'
%b	No of bytes in the file, including allocated size if it is a directory
%c	Cluster number of 1st cluster in file (for debugging)
%d	day of month as two digit number, from file's creation or last modification time
%f	UNIX "ls -l" style file mode symbols, fixed as "-rw-rw-rw-" for files and "drw-rw-rw-" for directories
%h	hour as two digit number, from file's creation or last modification time

- %m minutes as two digit number, from file's creation or last modification time
- %M month as two digit number, from file's creation or last modification time
- %n filename (excluding path)
- %N DOS style short file name
- %o Month as three letter abbreviation, from file's creation or last modification time
- %p directory path to file
- %r indicates that a recursive listing is required. This code can be anywhere in the format string
- %s seconds as a two digit number, from file's creation or last modification time
- %t Year of file modification/creation if longer than 360 days ago, otherwise time in format hh:mm
- %x DOS-style either file size or "<DIR>"
- %Y Year as two digits, from file's creation or last modification time
- %Y Year as four digits, from file's creation or last modification time
- %z File Type code 'F' = file, 'D' = directory, 'L' = volume label

## Listing Formats

### Basic Format

Form at	Description	Example
0	file names only, one per line "%n"	test1.txt test2.txt dir1
1	Normal listing (1st column: F = File, D=Directory, L = Volume Label)	F 1990-01-01 00:00:00 400 test1.txt D 1990-01-01 00:00:00 1024 dir1 F 1990-01-01 00:00:00 0 testdatafile2.txt
	"%z %Y-%M-%d %h:%m:%s %b %n"	
2	UNIX ls -l format (used internally by FTP)	-rw-rw-rw- 1 vm2 vm2 400 Jan 01 1990 test1.txt drw-rw-rw- 1 vm2 vm2 1024 Jan 01 1990 dir1 -rw-rw-rw- 1 vm2 vm2 0 Jan 01 1990 testdatafile2.txt
	"%f 1 vm2 vm2 %b %o %d %t %n"	
3	Debug information (includes starting cluster)	F 1990-01-01 00:00:00 400 c1=2 TEST1 TXT test1.txt D 1990-01-01 00:00:00 1024 c1=3 DIR1

and DOS-compatible file name)	dir1 F 1990-01-01 00:00:00 0 cl=0 TESTDATATXT testdatafile2.txt
%z %Y-%M-%d %h:%m:%s %b %c %N %n"	

Adding 8 to the format code shows the full directory path to each file

Adding 16 to the format code produces a recursive listing of the contents of subdirectories, and forces display of full directory path too.

E.g.

```
-->print fs:17
F 1990-01-01 00:00:00      400 test1.txt
D 1990-01-01 00:00:00    1024 dir1
F 1990-01-01 00:00:00      13 dir1/test11.txt
F 1990-01-01 00:00:00       0 testdatafile2.txt
```

### Each filename in a separate string

If you want a list of files as a set of separate strings (one filename in each string) then you can Print the filesystem to a ['Buffer of Any'](#), which will do the conversion for you.

You can then put the list in alphabetical order using [Sort](#).

## Notes on Using File Systems

### File Names

File names in the FAT system are Windows compatible as long as the ASCII or ISO-8859-1 (Latin-1) 8 bit character sets are used.

There is no support for 16 bit characters in either UTF16 or UTF8 encodings.

### Directory Separator

In filenames that specify directories, the character `'/'` is used to separate directory names from each other and from the final file name. This is the same convention as used in UNIX. Use of `"\"` as in DOS/Windows would be inconvenient in Venom as it is a special string escape character and would have to be written as `"\\\"` every time it was used. Use of `"/` is also compatible with URIs received by the HTTPServer object.

Any path may optionally include a `'/'` character at the beginning. The root directory is assumed regardless.

All filenames must specify a full path - there is no such thing as a "current working directory" in Venom.

The maximum length of a file name including path information is 260 characters.

## Wildcards and File Name Matching

Several of the file system and file messages can use wildcard specifications in file names. The special wildcard characters are:

- "?" matches any single character
- "\*" matches zero or more of any character

For example, the pattern

- "\*" will match any file name.
- "\*.txt" will match any file name that ends in ".txt".
- "abcd.???" will match any file name that consists of "abcd." followed by exactly three more characters.
- "\*2\*" will match any file name that contains a "2" anywhere.

Names are stored with case of letters preserved, but file name matching is case-insensitive.

## Root Directory Limitation on Number of Files

*Programs which create files dynamically and are liable to create a large number of files, especially with long names, should create them in a subdirectory.*

The reasons for this are:

- The root directory is fixed as 512 entries for the Flash File System and memory cards not using FAT32, and much less for [RAMdisk](#)
- Subdirectories can contain many more file entries, to a maximum of 65535 (theoretically) or as limited by RAM or memory card space.
- Note that file names not conforming to the DOS-compatible 8.3 format take more than 1 directory entry
- Frequent renaming, deleting and adding files can fragment the directory, further reducing its capacity

Note that on memory cards with FAT32 file system (generally anything over 2GB, and some 512MB and 1GB cards), this root directory restriction does not apply.

## Multiple access to same file

A file should be opened only once, so a file object has only one read and write point. A run time error is generated if you attempt to open the same file normally in two variables.

You can open a file more than once at a time if you open it read-only by specifying a third parameter of -1 in the Filesystem Open message.

You can copy a file variable or have multiple tasks access a file through the same global file variable. If one task is writing to a file and the other is reading, all updates and additions to the file can be seen by the reading task immediately.

If a file variable is copied to another variable and the file is then closed, access is no longer possible using either variable.

## Locking

The FAT file system is locked internally during the processing of any message that is likely to involve reading or writing the media, in order to serialize access to the media. This enables separate tasks to access files in the same system with no data corruption. You can lock the whole file system, for example if you need to guarantee that the contents of a directory will not be changed by another task during a sequence of operations. Locking a file is more often useful, so that different tasks can write to the same file. Printing to a file locks the file automatically for the scope of the Print statement..

## Storage Media Considerations with MMC/SD cards and On-board Flash

The file system works with block device drivers to access the physical media. Because reading and writing the physical media can be time consuming, and memory cards also have a limit on the number of writes they can handle in their lifetime, the memory card and on board flash drivers implement block cacheing so that recently and frequently used data blocks are retained in memory. This also means that updates to files and directories typically leave some data written to a cache buffer but not copied to the filesystem media. You can force writing to media by using the FileSystem [Flush](#) message at any time in order to ensure that the data is safe from loss by power failure or physical media removal.

On the flash file system, blocks that have been modified are held in an area of non-volatile memory. Writing is delayed to minimize the number of flash write cycles.

The driver can also tell the file system if the media is write protected (applies to MMC/SD only) and will block any attempt to write to the media.

## Removable Media

Special considerations apply to removable media such as memory cards.

With removable storage media, the device driver can inform the file system of media removal and the file system takes appropriate action including invalidating all the cached blocks.

A FileSystem object can be created with no media present. Similarly there is no problem if the media is removed or changed while no file is open. When the card is replaced or changed, the next access to the file system will silently reset and reload its parameters from the card.

If a memory card is removed while a file is open, even if the card is subsequently replaced or a different card inserted, the next attempt to access any open file will have the following results:

- all open file variables will die
- The block cache will be destroyed

The presence of a card can be detected without generating runtime errors by use of the [Status](#) or [Valid](#) messages.

If supported by the hardware, the file system will illuminate an indicator light whenever a file is open, data is being transferred or the cache contains data not yet written to the card, warning the user not to remove the media.

## Power Loss

When writing to a file, media removal or power loss can result in loss of data.

Closing a file and re-opening it will guarantee that the directory entry is updated with the file's size. The filesystem [Flush](#) message achieves this without the need for closing the file for all open files and also (depending on file system type and optional parameter) writes to the media all data that has been written to the cache.

The FileSystem [Flush](#) message involves time-consuming extra media access, so the programmer must make a judgment about how often and when to use it.

For RAM disk based file systems, the [Flush](#) message is still useful for updating the directory entry, but other data is written directly.

For the Flash File System, cacheing is implemented in non-volatile memory, so even if power is lost while unwritten data is in cache, it will be written to flash memory when the VM2 is powered up again. Note that a directory entry (including the file's size) is only as up to date as the last [Flush](#) or [Close](#) so this should still be used periodically if a file is being written over a long period of time.

## Errors

Run Time error messages generated in the FileSystem are listed here.

### Error 25 - Code checking error

Please report this to us: it means that an internal consistency or bounds check has failed.

### Error 26 - File Access error

This typically covers Venom Programming errors, for example:

- Read past end of file
- Access a closed file
- File name too long (max 260 characters) or contains illegal characters
- Attempt to open a file in a non-existent subdirectory
- Attempt to open too many files in root directory
- Attempt to open a file which is already open
- Attempt to delete a file which is currently open

### Error 7 - Write to Read-Only Item

- Device is write protected: SD card WP switch or Flash file system created read-only
- Attempt to assign a new name to an open file (this should be done the the FileSystem [Name](#) message when the file is closed)

### Error 12 - Device Not Found

This is generated:

- if Make specifies a device with the wrong interface type or a non-existent partition number
- on any attempt to access a file if the media was removed and/or replaced

Note that Make will succeed on a removable media device like a memory card holder even if there is no media present. The media must be present before the software attempts to open any file or directory. See [Status](#) and [Valid](#) filesystem messages for ways to check this.

### Missing Files, File Corruption and Startup Checks

A Venom application should be designed to be able to cope with missing files for a number of reasons:

- A program so designed is easier to run the first time it is used, otherwise a different program would have to be run to create the files.
- When the file system is started, the media is checked for integrity of the file allocation information. The checks make little attempt to repair invalid directory entries or free space allocation; instead, any files that are compromised by file system corruption are simply removed. An exception to this is where extra blocks have been allocated to a file but the length shown in the directory entry does not reflect the use of those blocks: in that case the file is preserved but the unused blocks are freed.
- It is impossible to guarantee that files will not be corrupted in RAM disk or on memory cards, if a program goes wrong (especially one that addresses system memory directly) or crashes as a result of any run time error or if the power is removed unexpectedly.

### USB access to Flash File System

The VM2 can be made to appear just like a USB Flash drive by allowing USB access to the VM2's Flash Filing System.

There are two methods for making the Flash File System visible on a USB connection

1. Program Mode USB access
2. Run mode USB access.

### Program Mode USB access: Production programming

While a VM2 is waiting at the **Clear RAM?** prompt it will automatically make its Flash file system available as a USB Flash Drive. This is useful for **production programming** of VM2s. No Venom application needs to be running for this to happen. If there was no file system in Flash memory, an empty one is created. All you have to do is start up the VM2 in Program Mode on a USB-enabled Application Board and connect it via a USB Cable to your PC or other host.

In Windows, you should see a new drive icon appear in the "My Computer" windows. You can open this as a window and drag and drop files into it.

Program Mode USB access is useful for loading the VM2's Flash File System with your firmware files, e.g.

- Venom [firmware update](#) files (**.vfu**)
- [Bitmap](#) image files (**.bmp**)
- Venom [font](#) files (**.vaf**)
- [Audio/Sound](#) files (**.wav**)

*You can produce vfu, etc files using [Protect](#). You can also download vfu files from our website.*

The recommend sequence of operations for loading a new OS or firmware is this:

1. Plug the VM2/Application board into your PC using a USB lead. You may need to configure the Application Board for USB - see the Application Board datasheet.
2. Reset or power up the VM2 in Program Mode. A new USB drive window should open up (or you may have to find the new drive)
3. **Important:** Make sure the VM2 drive window has no files in it, or delete them if there are any. [Explanation](#)
4. Drag and drop all your application firmware files into the new drive window.
5. Wait until the copy process ends - you should check that the LED on the VM2 has stopped flashing.
6. Press the Reset button on your application board (or power the VM2 off and then on)
7. The LED on the VM2 will flash while it reprograms itself with a new firmware. Do NOT REMOVE POWER OR RESET THE CONTROLLER AT THIS POINT. If you do you may have to re-load the operating system the slow way: over the serial port, using VenomIDE.
8. When the LED has stopped flashing then it should reset itself with the new OS and Application in its flash memory. The firmware update file will have been deleted.
9. Put the VM2 in Run Mode using the switch on the Application Board and then reset it. It should start running your application.



*Note: it is also possible to update a VM2's firmware remotely by transferring a vfu file to the Flash Filing System and then calling [Protect](#) to initiate the re-programming process.*

### Activity indicator

The VM2's on-board LED will flash rapidly during USB write activity to indicate that you shouldn't pull out the USB cable, reset the VM2 or interrupt the system in any other way.

### Run Mode USB Access

By default the Flash File system is not accessible via USB when a Venom programming is running. USB access is enabled by sending a [Connect](#) message to the file system. The USB host and the Venom program will both have access to the filing system.

### Hardware requirements

USB Filing System access is available on VM2 (5900) and VM2D (5907), but not on VM2L (5901).

The VM2 should be connected to a suitable USB hardware interface. Please see the application note *Designing VM2 Application Boards*, available on our website, for more details.

If you are using Application Boards 5902 or 5922 then the **USB** switch (sometimes labelled **Dflt Comm**) must be in the OFF position and the correct links must be fitted. See the datasheet for the 5902.

Currently only the 7 MByte Flash File System is accessible by USB. (The Memory Card and 'RAMDISK' filing systems may become accessible to USB later).

### Contiguous Files

Some applications for files in the Flash Filing System exploit the fact the file data can be directly addressed in memory, for example:

- Production programming of 'empty' VM2s with their application code and/or their Operating System
- Remote updating of VM2s with new application code and/or Operating Systems
- Audio files
- Bitmap image files
- Font data files

However, for the file data to be addressable in memory the file data must occupy a single (*contiguous*) memory block within the Flash Filing System.

### Creating contiguous files

There are two different ways to create contiguous files. Each is appropriate to a different situation.

#### 1. From your PC, over USB:

1. Delete all the files in the Flash Filing System
2. Create the files you want, either one at a time or in one or more drag-and-drop operations from Windows.
3. All files created in this way will occupy contiguous memory.
4. Note that once you have extended or deleted an existing file, files that have been extended and new files are not guaranteed to be contiguous.

This may be understood by considering that the FAT filing system allocates additional memory blocks to files by looking for unused blocks, always starting from the lowest address.

#### 2. From within Venom:

You can ensure that any *new* file you create from within Venom is contiguous either by

- Deleting all existing files in the Flash Filing System
  - Compacting the Flash Filing System
- before creating the new file

After deleting or compacting, any new file you create will occupy contiguous memory blocks. If

- You fully finish creating one file before starting to create the next
- You don't delete any files.

### Notes

To compact the filing system use the [FileSystem.Adjust](#) message.

Files that were contiguous before compaction will remain contiguous after compaction. The compaction process simply moves file data down in memory to fill empty blocks from the blocks immediately above.

Be careful to delete any file before compaction that has a name you might re-use after compaction.

You can test if a file is contiguous: if the file has a non-zero address then it is contiguous. Use [FileSystem.Address](#).

## File data memory address

To get the memory address of a file's data use [FileSystem.Address](#).

## File

*Please read about the [FileSystem](#) object before using files for the first time.*

A file object controls a file in a [FileSystem](#). A file is a sequence of data items of one of these types:

- 8 bit unsigned integer
- 16 bit signed integer
- 32 bit signed integer
- 32 bit floating point
- Text
- Objects created with user defined [Classes](#)

Data can be written to the end of a file, and can be read in sequence from the beginning of the file or from any selected point.


Any element of a file can be accessed by its numerical position in the file and read or changed.

Any file can be printed, in a format that depends on the data type, and a text file can also be printed to.


The interface is designed to be as similar as possible to that of the Buffer object type.

## Summary of messages

**Creation**  
**Close**  
**Element**  
**Empty**  
**Find**  
**Get**  
**Length**  
**Name**  
**[Open]**  
**Put**  
**Queue**  
**Readpoint**  
**Reset**  
**Time**  
**Help**  
**Print To**  
**Print**

 The Lock and Unlock messages can be sent to a file, however it is important to note that the normal file access functions (e.g. get and put) ignore locks. Printing to or from a file always locks it for the duration of the Print statement. If multiple tasks are writing formatted sequences of data to the same text file, using a single Print statement for each unbroken output record will keep it intact. For other file types, or where a single Print statement is not possible, use the Lock and Unlock messages.

Note that one task may write to a file while another reads it without any need for locking at all.

 See also Locking in the Tutorial.

## Creation

There is no **Make** or **New** for files. The only way to create a file (file reference object) is by sending an **Open** message to a **FileSystem** object.

This is documented in [FileSystem Open](#).

## Close

### Close

The file remains in the file system but is no longer associated with any variables that referred to it and no messages can be sent to it.

**Close** is defined to be the same as **Die**, and vice versa. This allows you to use **AutoDestruct** to close files automatically.

Closing a file updates the file's directory entry with the file's size. If a source of current date and time information is available, the directory entry is timestamped accordingly; if not the Venom time value of 0 results in a timestamp of 01/01/1990 00:00:00.

On MMC/SD cards, closing a file also results in buffered unwritten data being written to the physical media.

### Time Source

The VM2's internal clock-calendar function is used as the source of time and date information. If the VM2 has a connection to the internet, the internal timer can easily be synchronized to accurate internet time servers.

## Element

**Element**(Int **Elementnumber**) ⇔ Any

<File>.(Int **Elementnumber**) ⇔ Any

Sets or returns a single data element of the file.

For a text file the value is treated as an integer.

The file starts at Element 0.

It is relevant to know that in a text file created by Print To, a line break takes up two character elements (CR=13 and LF=10).

## Empty

### Empty

Removes all the data in the file, leaving read and write pointers at zero. The file is then in the

same state as if it had been newly created.

## Find

```
Find(String or Buffer pattern [, Int startpos [, Int  
uncased] ]) ⇒ Int
```

Searches a text file for a string supplied either as a fixed string or as the contents of a text buffer.

Find returns an integer showing the element position of the beginning of the first instance of the search pattern if found, or -1 if not found.

The search starts at the character position **startpos** in the file if specified, or else at the beginning of the file which is equivalent to a **startpos** value of 0.

If **uncased** is defined as 1, the search is case-insensitive.

The search pattern has a maximum size limit of 255 characters.

The Boyer-Moore search algorithm is used, which is very fast, especially with a long search pattern.

## Get

Get reads data from a file. Single values or multiple values may be read.

### Reading single values from a file

```
Get ⇒ Any
```

Get returns a value from the current read point in the file and advances the read pointer to the next element. It is an error to attempt to read past the end of the file. (See [Queue](#) Message for how to avoid this)

The type of data returned is:

- Integer for all types of integer file
- Float for a float file type
- Integer value of a single character for a text file

Note that a text file created with **Print To** and containing line breaks will return the sequence 13, 10 at the line breaks (ASCII CR and LF).

## Reading multiple values from a file

### Reading Arrays

`Get(Array a[, Int n]) ⇒ Int`

Where a is an array of the same data type as the file, this will read data from the file into the array. The transfer stops when either the end of file or end of array is reached. The optional parameter n limits the number of elements transferred. The file **ReadPoint** is left pointing to the next element after the last transferred, if there is any data remaining in the file.

The value returned is the number of elements transferred.

A runtime error is thrown if the array and file types do not match.

### Reading Strings (lines of text)

`Get(Str s) ⇒ Int`

Efficiently reads a line of text from a file into a String object. This method is much faster than reading one character at a time.

Any previous string contents are overwritten.

The returned value is the length of the string.

ASCII CR (13) in the text is ignored and not copied to the string.

ASCII LF (10) in the file is treated as a line terminator and not copied to the string.

All other values including control characters are copied to the string.

If the line is longer than the String object's capacity, the string is filled to capacity and the next file **Get** will continue from the next character on that line.

### Reading Classes (records)

When the first parameter to **Get** is a user-defined **Class** object, **Get** reads a record out of the file and loads this into the target object.

#### Choice of Binary or Text formats

Records may be stored in files in binary or text formats.

### Binary format records

`Get(Class obj) ⇒ Int`

Get will read the file and load the target object from the binary data it reads.

The exact binary format is not specified here, but in general the Class used to read a record out of a file must be exactly the same as the Class that was used to write it.

To write a record to a file in binary format use [Put](#). You can find the length of a record before you write an object to a file using the **Length** message.

Here is an example of some code to write and read back a record in binary format.

```
Class Record
  a Int 8
  b Int 16
  str New String(30)
End

rec := New Record(1,2,"Three")
file.Put(rec) ; Write the record to a file.
file.Get(rec) ; Read a record out of a file.
```

## Text format records

**Get(Class obj, String format\_string)**

If a second parameter is supplied to **Get** then a text format is specified. There are two text formats supported currently:

1. If the second parameter is **"INI"**, it means use **INI file format**.
2. Otherwise use CSV format; the format string specifies the CSV separator character - usually a comma.

For example:

```
file.Get(record, "INI") ; read INI file format
file.Get(record, ",") ; read comma-separated data
```

## INI file format

In INI file format, records are expected to be listed as in the example below, where two records are listed, along with header lines in square brackets that delimit and optionally identify each record.

```
[Person1]
Name="Jim"
Age=42
[Person2]
Name="Fred"
Age=56
```

When reading a record in INI file format

- Member names are not case sensitive.
- If a member value occurs more than once then it will be over-written with the last value



seen.

- Members that are not read retain their original value.
- Array elements are listed in comma-separated-value format, with a `\` to indicate continuation on the next line.
- Strings are in double quotes; no escape characters are supported currently.
- Lines starting with `;` or `#` are treated as comments and ignored.
- Member values will be read until the end of the file, or a line beginning with `[`, is seen (the next record's header line). The file is 'rewound' so that the `[` is the next character to be read from the file.
- It is the responsibility of the Venom programmer to read, process and interpret the header lines.
- `Get` returns the number of members it read.

### Creating INI files

If you [Print a Class](#) to a file using `"INI"` as the format specifier then it will be printed in INI file format. Header lines must be printed explicitly before each record.

### Data errors in INI file format

If you pass a third, non-zero, parameter to `Get` then it will throw a `"Script/Data error"` if

- A non-existent member name is seen
- A member is of a type that can't be represented in an INI file
- An array overflows

For example:

```
n := myfile.Get(p, "INI", True) ; read the object data, throwing e
```

### CSV Format

When reading a record in CSV file format

- No member names are included - member data must be listed in the order it occurs in the Class.
- Each record occupies exactly one line in the file.
- Strings must be inside double quotes.
- Array or String overflows will result in a runtime error.

### Creating CSV files

You can create a CSV file by [printing](#) user-defined Class objects to a file using the format

specifier string that contains the separator character.

## Length

**Length**  $\Rightarrow$  Int

Returns the length of the file in elements of the file's specified type.

For text files, this is the number of characters, counting each line separator as the two characters CR, LF.

## Name

**Name**(str s [, Int modifier])

**s** string variable into which to copy the file name

**modifi**Select which part of name to copy:

0 (default) full path and name

1 - omit path

2 - only copy the "file type" extension i.e. the last '.' and the characters following it

Copies the file's full directory path and name to a string variable.

If the string variable is not long enough to hold the file name, the name will be truncated.

A file name can never be longer than 260 characters.

**N.B. This is a change from previous behaviour and syntax of this message.**

You cannot rename a file this way: use [FileSystem Name message](#) for that.

## Example

To print a file's name:

```
-->Make s string(260)
-->f := fs.open("dir1/test.txt", Char)
-->f.name(s)
-->print s, CR
dir1/test.txt
-->f.name(s, 1)
-->print s, CR
test.txt
-->f.name(s, 2)
-->print s, CR
```

```
.txt  
-->
```

## [Open]

Files do not have an **Open** message. Instead, files are opened by sending an **Open** message to a **FileSystem** object.

 See [FileSystem Open](#)

## Put

**Put**(Any value) ⇒ Int

The data type of the parameter general has to match the file type. Integer values must be written to a file of one of the integer types. Values will be truncated if necessary to the size of the file data type. For the floating-point file type, the value must be a floating-point type.

The data is appended to the end of the file.

For a text file, the data can be a single character, fixed string or an entire text buffer whose length must not exceed 256 characters.

## Putting Arrays

You can **Put** data from an Array when it has the same data type as the file.

**Put**(Array a, [Int start, Int count])

Optional parameters specify the first element to copy and the number of array elements to be transferred.

The default action is to copy the whole array.

## Putting classes (records)

You can use user-defined classes to write 'records' to a file.

### Binary format

You can **Put** an object of a user-defined [Class](#) to an 8-bit integer file. A binary record representing the object is written to the file. For example:

```
rec := New Record(1,2,"Three")  
f.Put(rec)
```

The Class-default message **Length** (sent to the record template object) gives the number of bytes the object will require to be stored as a record in the file.

The record may be retrieved using [Get](#).

**INI file or CSV format**

If you want to use a text format (either CSV or INI file) for storing records then you should [Print the class](#) to a text file, e.g.:

```
Print To f, myClassObj: ", ", CR
```

**Queue**

**Queue**  $\Rightarrow$  Int

Returns the number of elements remaining to be read from the file. For a text file, an element is a single character.

**Readpoint**

**Readpoint**  $\Leftrightarrow$  Int

Gets or sets the point at which the next Get message will read data from the file. The file starts at position 0.

**Reset**

**Reset**

Resets the read pointer to the beginning of the file.

This is equivalent to `f.Readpoint := 0`.

**Time**

**Time**  $\Leftrightarrow$  Int

Returns or sets a time value in seconds, compatible with the [DateTime](#) and [RealTimeClock](#) objects (i.e. time in seconds since 00:00 on January 1, 1990)

The time value returned corresponds to the date and time when the file was created or last modified.

Modification times are updated only when the file is closed or flushed.

If the real time clock hardware was not initialised when the file was created or last modified, the returned value will be 0.



Note that to set the time and date successfully to a different value from the current time, this

must be done immediately before closing the file. Any further write to the file would cause the current date and time to be applied at the time of closing the file.

 See also: File [Close](#), Filesystem [Flush](#)

## Update

### Update

The File **Update** message updates the file length, time and date entries in the directory entry for the file, as if the file had been closed and re-opened. The effects are as described below:

#### All Filesystems

**fs.Length (name)** and **fs.Time (Name)** will return the newly-updated values for the file.

#### RAMdisk and Flash FileSystem

This will make a permanent update. If power is lost and the system is started up again and creates the same file system, the directory information will be up to date with when the **Update** message was last sent.

## Relationship with Filesystem Flush Message

The Filesystem [Flush](#) message performs the equivalent of an **Update** on all files currently open, before (where applicable) writing any blocks to the physical device.

## Help

```
HELP file f
```

It is worth noting that the standard help message, which shows the type of a variable, additionally shows the file name for a file variable.

## Example

```
-->HELP a  
It is a text file named "abc.txt"  
-->
```

## PRINT TO

**Print To** <File>, **list**

A text file can be the destination of a print operation. The print output is appended to the end of the file in exactly the same way as by a series of Put messages.

When printing to a file, **CR** sends a CR LF (13, 10) sequence to the file.



## PRINT

**Print** <File>[: Int **n1**[: Int **n2** [: Int **n3**]]

Lists the contents of the file.

Text Files are displayed in their normal text format

Other file types are displayed one item per line.

If the colon(s) and format code(s) are present they are interpreted in the same way as when printing buffers of various types:

File type	1st format parameter	2nd format parameter	3rd format parameter
Text	If positive, print first n1 characters of file If negative, print last (-n1) chars of file	Not present	Not used
Text	First char position in file to print	Number of chars to print	Not used
Text	“+” - print entire file adding CR in front of any LF encountered (UNIX -> Windows conversion)	Not present	Not used
All Integer types	Minimum field width (number of characters to print per number)	Not used	Not used
Float	Minimum field width	Number of decimal places	Exponential format

In the FAT file system, a file opened as a directory can be printed. A numerical colon operator can be used to control the display format in the same way as when [printing a FileSystem object](#)

## FTPClient

This object is an FTP (File Transfer Protocol) client which connects to FTP servers and can get a server directory listing and send and receive files.

N.B. FTP is an old and widely supported protocol on the internet, but it is not secure. User names, passwords and file data are transferred unencrypted and file transfers can be intercepted in either direction.

The Venom FTP objects support a small but commonly used subset of the FTP protocol as documented in RFC959.

See also [TCP/IP Networking](#)

### FTPClient Messages and equivalent FTP Protocol commands

This is probably only interesting if you are already familiar with the workings of FTP.

<b>Open</b>	USER PASS
<b>Get</b>	PORT or PASV RETR
<b>Put</b>	PORT or PASV STOR APPE
<b>Remove</b>	DELE
<b>Name</b>	RNFR and RNT0
<b>Go</b>	CWD
<b>Close</b>	QUIT
<b>Print obj:0</b>	PWD
<b>Print obj:1</b>	NLST
<b>Print obj:2</b>	List

The FTPClient object can request binary or ASCII file type. As both the Venom filesystem convention and that of FTP ASCII transfer is to use CRLF for line endings there is no difference in the processing at the Venom end, but the server may do ASCII translation of text from its local system convention when ASCII mode is requested.

**Make**  
**Close**  
**Debug**  
**Get**  
**Go**  
**Name**  
**Open**  
**Put**  
**Remove**  
**Timeout**  
**Print**

## Creation

**Make** <object> **FTPclient**([Int mode])

### Parameters

**mode** 'A' for active, 'P' (default) for passive - determines the mode for opening the data connection

### Active and Passive Modes

When the data connection is opened, it can be initiated from either the server end or the client end.

The terms Active and Passive are from the server's point of view, so:

Active means the server opens a data connection to the client

Passive means the client opens the data connection to the server

Passive mode is used by typical web browsers and is appropriate if the client is behind a firewall that restricts incoming TCP connections. Occasionally a server behind a firewall will need to use active mode for similar reasons.

### Port Numbers (Technical Note for Firewall Administrators)

If you are using a VM2 which makes ftp connections outside a LAN through a firewall, and which for some reason must use Active mode, you may find it useful to know that the Venom FTP client's PORT command will use data connection port numbers in the range 2000-3999, so the firewall can be configured to allow incoming connections to those ports.



## Put

```
Put(file f, str name [Int append]) ⇒ Int
```

### Parameters

**f** An open file to send

**name** Fixed string or text buffer with the name of the file to be created or replaced on the FTP server (can include path)

**append** If present and non-zero, this requests the server to append the transmitted contents if the named file already exists, instead of replacing the file

### Returned Value

True (1) for successful transfer, False (0) for any failure.

Sends a file's contents to the server. By default the server replaces any existing file of the same name with the new contents. If the **append** parameter is included with a non-zero value, the new contents are appended to any existing file of that name on the server.

The data type for the FTP transfer is ASCII if the local file was opened as a text file, binary for any other file type. This makes no difference in the VM2, but it might affect the way the file is stored on the server: in particular, for an ASCII transfer a Unix-like server will use LF ("\n") for line endings, while a Windows server will use the sequence CR LF ("\r\n") for line endings.

## Close

### Close

Closes the connection with the server, first sending an FTP QUIT command, then closing the control TCP connection.

Harmless if FTP connection already closed.

## Debug

```
Debug(Int Value)
```

### Parameter

0 - no debug output

## 1 - display control connection dialogue

When debug mode is enabled, all commands to the server and responses back from the server are displayed on the serial 1 terminal. This is useful when any FTP operation doesn't work, as the server responses usually contain explanatory text.

### Get

`Get(file f, str name) ⇒ Int`

#### Parameters

- f** An open file in which to receive the file data
- name** Fixed string or text buffer with the name of the file on the FTP server (can include path)

#### Returned Value

True (1) for successful transfer, 0 (false) for any failure.

Gets a file from the server, storing its data in the open file. Any existing file contents are replaced with those of the retrieved file.

The data type for the FTP transfer is ASCII if the local file was opened as a text file, binary for any other file type.

### Go

`Go(str path) ⇒ Int`

#### Parameters

- path** specifies new directory path

#### Return Value

True(1) if changed working directory successfully, false (0) if not.

## Name

**Name**(oldname, newname)  $\Rightarrow$  Int

**oldname** String or text buffer: contents must match the name of an existing file on the server

**newname** String or text buffer containing the new name for the file

Returned value: True (1) if rename was successful, False (0) otherwise.

Renames a file on the remote server to a new name.

If **newname** matches an existing file on the server, that file is deleted before the other file is renamed to it.

Note that this message is useful when updating a file on a server where a process on the server may be reading the file at any time. By uploading to a temporary file and then using rename, the update to the real file is a single instantaneous operation and the reading process will never see a partially filled file.

## Open

**Open**(site, str user, str password)  $\Rightarrow$  Int

### Parameters

**site** Integer or string IP address, or hostname. String can be fixed string or text buffer

**user** fixed string or text buffer holding user name for logging in to FTP server

**password** fixed string or text buffer holding user's password

### Return Value

True(1) for successful connection and login, false(0) if failed for any reason

## Remove

**Remove**(name)  $\Rightarrow$  Int

**name** String or text buffer containing name of file to be removed

Returned value: True (1) if remove was successful, False (0) otherwise.

Removes a file or directory from the remote server.

## Timeout

**Timeout**  $\Leftrightarrow$  Int

Reads or sets a timeout value which is applied the the FTP client's control and data TCP connections.

Default is 0, meaning no timeout limit is applied.

This will set a timeout limit for received data or for acknowledgements when sending data. By default there can be a delay of over 3 minutes before a link is dropped because of lack of response. In some cases, like a local area network, it is reasonable to assume that lack of response within a few seconds indicates a non-responsive host; setting a lower timeout discovers this condition faster.

## PRINT

**Print** <FTPclient>

Prints [FTPCCLIENT] followed by some state information

**Print** <FTPclient>:0

Prints the servers's current directory as a string, if a control connection is open  
Otherwise prints a single question mark

**Print** <FTPclient>:1[:str **path**]

**path** optional string or text buffer specifying directory path to list

Prints a directory listing of the server's current directory or specified path.

The format of the listing is one file name per line separated by CR LF and is thus designed to be usable by a program.

If the connection is closed, prints a single question mark.

```
Print <FTPclient>:2[:str path]
```

**path** optional string or text buffer specifying directory path to list

Prints a directory listing of the server's current directory or specified path

The format of the listing is server-dependent and designed to be human-readable.

If the connection is closed, prints a single question mark.

## FTPServer

The FTP (File Transfer Protocol) Server object runs a server that requires no program input from the controlling Venom program once set up.

It will work with any of the VM2's file systems.

The essential services of FTP are:

- User authentication
- listing directories
- transferring files to or from the server

See also [TCP/IP Networking](#)

The following FTP commands are understood:

USER	Login: user name
PASS	Login: password
NOOP	Null command for testing and suppressing inactivity timeout: returns "200 OK".
CWD	Change working directory.
CDUP	Change to parent directory.
PWD	Print working directory.
QUIT	Log off and disconnect.
PORT	Client says which IP address and Port it is listening on (active mode FTP).
PASV	Request Passive mode transfer.
TYPE	File data type: Any type accepted. VM2 does not treat binary and ASCII differently as its file system stores ASCII compatibly with FTP ASCII mode.
RETR	Retrieve a file from server and transfer to client.
STOR	Store content transferred from the client in a file in the server's file system, replacing any existing file of the same name.
APPE	Store content transferred from the client in a file in the server's file system, appending to

	any existing file of the same name.
LIST	List files in default directory listing format.
NLST	List file names only separated by CRLF.
SYST	Replies with "WIN32", which is a recognized OS name.
SIZE	Size for a file, error message for a directory.
STAT	Shows system status info: Venom version, logged in user.
RNFR	Rename "from" file name.
RNTO	Rename "to" file name.
DELE	Delete file.
XCWD	Synonym for CWD.
XPWD	Synonym for PWD.
XCUP	Synonym for CDUP.

See the description of the [Run](#) message for more details on how each command is handled.

The FTP server has been tested successfully with:

- Unix and Windows command line FTP clients
- SmartFTP (Windows graphical FTP client)
- Mozilla Firefox
- Microsoft Internet Explorer.

#### Limitations

- ▼ • The server doesn't allow clients to create directories
  - Only one connection can be handled per server object, but multiple Venom tasks can run a server object each.
  - Only ASCII and binary types and the streaming mode are supported (but the alternatives are rarely used)
  - Numerous less frequently used FTP commands are not supported

#### Summary of Messages

**Make**  
**Debug**  
**Go**  
**Run**  
**Print**


## Creation

**Make** <object> **FTPServer**(FileSystem **fs**, str **name**, str **password** [,


<b>fs</b>	The <a href="#">FileSystem</a> to use for file transfer operations.
<b>name</b>	String or text buffer with valid username to log in to FTP server
<b>password</b>	String or text buffer containing password required for successful login
<b>directory</b>	Optional false root directory for FTP operations. All file system content outside this directory is completely invisible to an FTP client. Default is root directory.

Both the username and password may contain wildcard characters: '?' matches any character, and '\*' matches any string.

If a false root directory is specified, it must exist on the file system.

 See [Run](#) description for more notes on false root directory and server operation

## Memory Usage

 Creating an FTPServer object uses about 2430 bytes of memory.

A file or listing transfer opens another TCP connection which uses approximately 5k.

TCP/IP and file system activity also use unpredictable amounts of memory temporarily.

## Debug

**Debug**(Int **mode**)

**mode**    0 - no debug (default)  
           1 - enable debugging output

When debugging output is enabled, the ftp object sends to serial port 1 a message describing every command received and every response sent back to the client.

A typical sample of output looks like this:

```
ftp: sending response: 220 VM2 FTP server
cmd: "USER vm1" : USER
ftp: sending response: 331 OK, send password
cmd: "PASS secret" : PASS
ftp: sending response: 230 Logged in
cmd: "PASV" : PASV
ftp: sending response: 227 passive 172,16,1,150,15,132
cmd: "List" : List
ftp: sending response: 150 Opening data connection
ftp: sending response: 226 List done.
cmd: "SYST" : SYST
ftp: sending response: 215 WIN32
```

## Run, Go

```
Run [(Int time)]
Go [(Int time)]
```

**time** Number of milliseconds to run

Runs the server for the specified number of milliseconds. If no parameter is given, runs the server for ever.

**Go** is a synonym for **Run**.

If a file transfer is requested during the period specified, the **Run** message does not return until the file transfer has been completed.

While the server is running, one user at a time may log in and perform file operations. To enable simultaneous access by more users, separate ftp server tasks can be created.



## Notes on FTP

### Notes on Supported FTP Commands

The full details of these commands, their parameters and response codes are documented in RFC959 "File Transfer Protocol". The descriptions below summarize the set of commands supported by the VM2. In practice you should not need to know these details as you should be able to use the FTP server with many existing FTP client programs (including web browsers that can handle FTP), but if anything goes wrong, the information below may help to diagnose the cause.

If the FTP server was created with a false root directory, all references to the filesystem's directory structure are relative to that directory. E.g. if the filesystem has a directory `"/ftp"` which is used as a false root, an ftp request for `"/file1.txt"` will access `"/ftp/file1.txt"` on the file system. When the FTP current working directory is `"/"` it would be `"/ftp"` on the file system, and the FTP command `"CWD .."` or `"CDUP"` at that point would not result in a change of directory.

Use of a false root directory is recommended for any system connected to the internet as it protects all files outside that directory from any malicious user that has gained ftp access.

### USER, PASS

These must be sent in the correct order and with the correct user name and password. As a measure against brute force password guessing attacks, a series of failed logins causes a response delay whose length increases with each failure. After 1 minute of failed login attempts or inactivity the control connection is disconnected and the failure response delay reset to 0.

When the FTPServer object is created, both user name and password can be specified with wildcards. `'?'` matches any character; `'*'` matches any string.

### NOOP

Can be used by an FTP client to check that the server is listening or to prevent an inactivity timeout.

Only works when logged in.

Response is 215 OK

**CWD dir****XCWD dir**

A directory name beginning with ' / ' is taken to be an absolute directory.

Otherwise it is a subdirectory of the current directory.

The special string ". ." is interpreted as "change to parent directory". It is accepted but has no effect when used in the root directory.

**CDUP****XCUP**

Equivalent to CWD . . or XCWD . .

**PWD****XPWD**

Print working directory. The FTP root directory is the root of the file system.

" / " is shown as the root directory.

**QUIT**

Disconnects the control connection, allowing another incoming connection and login to start.

**PORT <text> h1,h2,h3,h4,p1,p2**

Indicates that data transfers are to be carried out in active mode. When a data transfer is requested the server will connect to the IP address and port specified by the numbers h1-h4 and p1, p2.

**PASV**

Causes the server to send a response detailing the IP address and port number on which it will listen for data transfers, and to set up a listening (passive) TCP connection when a data transfer is required.

The IP address will be the same as used for the control connection.

The port number will be in the range 2000-3999.

**TYPE x**

Specifies file data type. *x* is usually 'A' for ASCII or 'I' for Image (binary). The server accepts any value, but as the format for storage of ASCII text in files on the VM2 is the same as the format for ASCII mode FTP transfer, no special processing is required for either data type. The command therefore has no effect but produces a response

200 TYPE OK

### **RETR filename**

Retrieve a file - the file is sent to the client. The filename may be a simple name for a file in the current directory, a full path starting with ' / ', or a relative path from current directory.

### **STOR filename**

Stores a file from the client in the location and name specified. The filename and path are as for the RETR command. If a file with the same name existed, it is overwritten.

### **APPE filename**

Like STOR but if a file with the same name exists, the data from the client is appended to the file.

### **List filespec**

For a FAT filesystem, filespec may be:

- a directory
- a directory followed by a wildcard file specification
- a wildcard file spec only

The returned data is a list of files in UNIX "ls -l" format, because this format is expected by semi-automatic clients like web browsers.

The filesystem Print message is used for this function.

 See [FileSystem \(FAT\) Print](#).

### **NLST filespec**

Produces a list of file names only, separated by CRLF. For the FAT filesystem, *filespec* can be used and can contain directory path and wildcards.

### **RNFR filespec**

#### **RNTO filespec**

These set up source and destination names for a file rename operation. **RNFR** must be sent first and followed by **RNTO** with no intervening command. Either **filespec** can be a full path (starting with ' / ') or a name within the current directory, and can be the name of a file or directory. If the "to" and "from" paths are different, the file or directory is moved to the new position in the directory structure.

**DELE filespec**

Deletes a file. If **filespec** refers to a directory, all its files and subdirectories are deleted too.

**SYST**

Produces the response

215 WIN32

The response has to be a recognized string as defined in the IANA "assigned numbers" document and also needs to be one likely to be recognized by web browsers. "WIN32", whilst wildly inaccurate, at least correctly reflects the capabilities and limitations of the FAT file system used in the VM2 and is likely to be widely recognized by ftp client programs.

**SIZE**

For a plain file, produces the response

230 *nnnnn*

Where *nnnnn* is the file size in bytes.

For a directory, the response is:

550 Not a plain file.

This information seems to be used by some clients (e.g. web browsers) as an *ad hoc* way of guessing whether a name in a directory listing is a plain file or a subdirectory.

**Timeouts**

Any inactivity of more than 60 seconds, or failure to login successfully within 60 seconds, causes the control connection to be disconnected.

No file data transfer timeout is currently defined.

**GraphicsLCD**

GraphicsLCD manages text and graphics on many types of graphical LCD displays.

It supports

- Colour and monochrome displays
- Several built-in fonts, both proportional and monospaced
- User defined fonts - font conversion utilities are provided on our website
- Word wrapping
- Text justification (left, right and centred)
- Bitmap images

- Graphical primitives (lines, rectangles, 3D relief boxes, etc.)

### Where to look first

To make the most of the GraphicsLCD object see [TextBox](#) and the [print keywords](#), as they are central to almost everything you will do.

We also show example Graphics LCD code on our website.

Most applications for the GraphicsLCD also use a touchscreen. Please see the [Touchscreen](#) object.

### Indirect update

In order to optimise the speed of creating and modifying images on the LCD device, graphic primitives (lines, text, etc) to be drawn are first written to an internal 'scratchpad' RAM area. This image buffer is only transferred to the actual display device when the [Update](#) message is called. Update will try to optimise the process by only updating parts of the image that have changed.

### Summary of messages

**Make**  
**Bitmap**  
**Box**  
**FontData**  
**Format**  
**Line**  
**Off**  
**On**  
**Pen**  
**Reset**  
**TextBox**  
**Update**  
**Xpos**  
**Ypos**  
**Print To**

### Creation

The GraphicsLCD object can drive many different graphical display devices. Devices are divided into two major types:

1. TFT devices driven by **VM2D** or **VM2-D2**

2. Other devices, driven by [VM2](#)

### Graphic displays on VM2D and VM2-D2

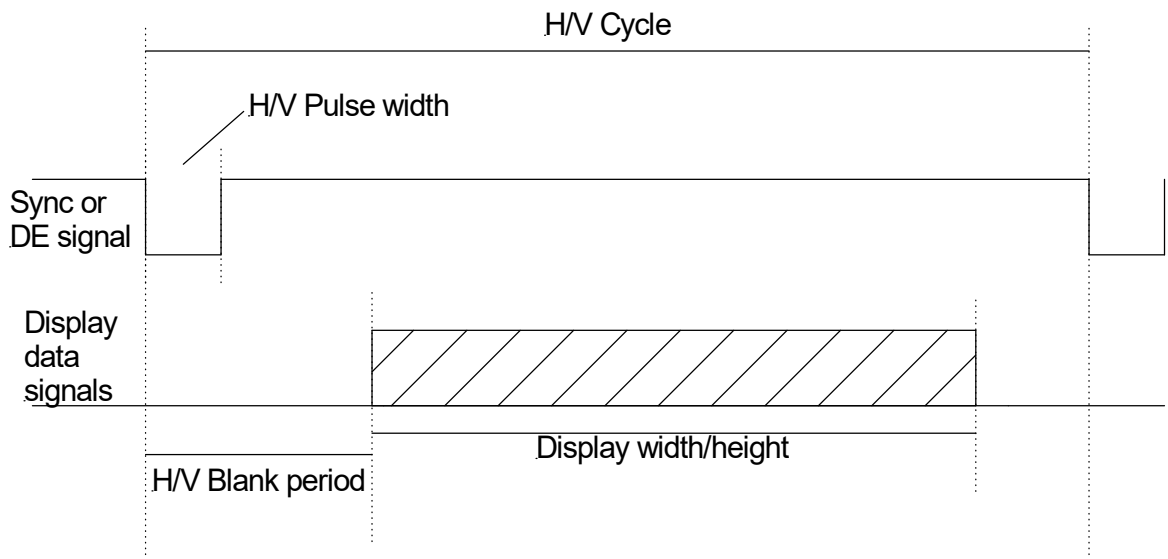
**Make** <object> **GraphicsLCD** (<Array(Int 16)> **parameters**)

The parameters to configure the onboard TFT driver are passed as an array of 16 x 16-bit integers to allow the flexibility to drive different displays.

The parameters are listed in the table below.

Element	Parameter name	Comments
0	Display driver type	1 = VM2D; 2 = VM2D2.
1	Display width	The width of the display in pixels, e.g. 640 for VGA.
2	Display height	The height of the display in pixels e.g. 480 for VGA.
3	Rotation	Rotation of the display. 0 = no rotation, 1 = 90 degrees, 2 = 180 degrees, 3 = 270 degrees.
4	Frame rate	The frame rate in Hz. Typically 60.
5	Horizontal cycle	The total number of clock pulses in a display line, including blank periods.
6	H. blank period	The blank period from the start of the H sync pulse to the first display data (pixel clock units).
7	H. sync pulse	The horizontal sync pulse width (pixel clock units).
8	Vertical cycle	The total number of horizontal lines in a display frame, including blank periods.
9	V. blank period	The blank period from the start of the V sync pulse to the first display line (in horizontal lines).
10	V. sync pulse	The vertical sync pulse width (in horizontal lines).
11	Signal Polarity bit flags	The polarity of various signals. Use <b>0</b> for most displays. The meaning of each bit when set: <b>Bit 2: PCLK clocks data on +ve edge.</b> <b>Bit 1: HSync pulse active high.</b> <b>Bit 0: Vsync pulse active high.</b>
12-15	<i>Reserved</i>	<i>Reserved for future expansion - please set to zero.</i>

The diagram below shows how the parameters in the array relate to the display signal timings. The horizontal and vertical timings have similar characteristics, so one diagram shows both.



Some datasheets specify timings such as *back porch* and *front porch*. The usual relationship between these parameters and the timing values we use is:

- Cycle = display width/height + sync pulse + back porch + front porch.
- Blank period = sync pulse width + back porch

### Example

```

MAKE g GraphicsLCD(StandardVga) ; Make the LCD object.
...

; Parameters for standard VGA TFT display.
Array StandardVga (Int 16,16)
    1, ; Display driver type.
    640, ; Display width
    480, ; Display height
    0, ; Rotation.
    60, ; Frame rate.
    766, ; Horizontal cycle
    100, ; Horizontal blank period
    30, ; H Sync Pulse - can be short in DE mode
    525, ; Vertical cycle
    35, ; Vertical blank period
    3, ; V Sync Pulse - can be short in DE mode
    0, ; Reserved, set to zero.
End

```



## Graphic displays on VM2

**Make** <object> **GraphicsLCD** (Int **type**)

### Parameters

**type**: the lcd controller type. See the table below for the display types supported.


### Example

**Make** **glcd** **GraphicsLCD**(3) ; *ST7565R-based display.*

### Currently supported displays

*If the display you want to use is not listed here please contact us - we may be able to write a driver for it free of charge.*

Type	External Display Controller	Display device	Display resolution	Comments
1	<i>None</i>	QVGA TFT	320 x 240	Requires VM2D.
3	ST7565R		128 x 64	Requires standard VM2
4	ILI9325C	MCT024D12TW2 40320PML	240 x 320	Requires standard VM2
5	ST7565R - via SPI bus 1	Electronic Assembly DOGL128-6	128 x 64	Use SPI Bus 1 on VM2 or VM2D; SPI1_MISO is used as A0 signal.
6	<i>As above</i> - via SPI bus 2	<i>As above</i>	<i>As above</i>	Use SPI Bus 2 on VM2 or VM2D; SPI2_MISO is used as A0 signal.
7	T6963C	Any T6963C based display with 128x64 pixels	128 x 64	Requires standard VM2
8	T6963C	Any T6963C based display with 240x128 pixels	240 x 128	Requires standard VM2

 The amount of memory used is dependent upon the display size. One bit of memory is taken for every bit in the display, plus a few Kbytes. For example, a 16-bit-per-pixel, QVGA display (320 x 240 pixels) takes around 154 K Bytes.

## Error messages

If the object detects the appropriate hardware is not present then you will get a runtime error message such as **Device Not Found** or **Feature not supported**.

## Bitmap

### Drawing bitmaps

There are two main ways to draw a bitmap image on to a display:

1. Use the Bitmap message to draw a bitmap at a position relative to the current TextBox.
2. Print the bitmap as part of some text output. See [here](#).

We deal with drawing bitmaps directly into a TextBox here.

```
Bitmap(Int bitmap_ref, Int x, Int y)
```

### Parameters

The **x** and **y** coordinates specify the position of *bottom left-hand corner* of the bitmap, relative to the current [TextBox](#) origin. *Note: the bitmap isn't confined to the TextBox area.*

**bitmap\_ref** is usually the *memory address* of the bitmap image. If the bitmap data is contained within an Array or a file in the Flash Filing System then you can use the [Address](#) message to get the address of the bitmap. For example:

```
lcd.Bitmap(ffs.Address("myimage.vbm"), x, y)  
lcd.Bitmap(myarray.Address, x, y)
```

Note: **bitmap\_ref** can also be the bitmap number [registered](#) with the GraphicsLCD object.

### Bitmap data formats

There are two major bitmap format families supported currently

1. The Windows Bitmap format (BMP files)
2. Venom Bitmap format (VBM files)

Bitmap images may be defined in Venom Arrays, but large bitmaps are best defined in Windows or Venom Bitmap files (extension .BMP or .VBM) loaded into the Flash File System. Note that the file data must be [contiguous](#).

## Comparing file formats

### Windows Bitmap (.BMP)

This is the most convenient format to use, because many graphics programs support it. However this format takes more space in the file system and is slower to plot.

The 32 bits-per-pixel variant supports transparency, so images with rounded corners or edges display better.

See [below](#) for useful macros.

*Note: Currently only the **24 bit per pixel RGB** and **32 bits per pixel ARGB** sub-formats of BMP are supported.*

### Venom Bitmap (.VBM)

The native Venom bitmap file format (VBM) takes less space in the memory or file system, is faster to plot, but it's not as convenient as the Windows BMP format. Also the VBM format doesn't currently handle transparency well. Our website has a utility you can download to convert BMP files into the VBM format.

See [below](#) for useful macros.

## Details of the VBM format


*Note: we supply a utility that can convert Windows bitmaps to Venom Bitmaps - you can download it from the **Resources** page on our website.*

### Monochrome Venom Bitmaps

One-bit per pixel Bitmaps may be drawn on both monochrome and colour displays

They will draw in the current display **foreground** and **text background** 'colours', just like printing text: binary '1' bits in the bitmap image draw in the foreground colour and Binary '0' bits draw in the background colour.

A *transparent* foreground or background can be used. On a monochrome display the pen colour 'INVERSE' can also be used.

 See [Pen](#) for more information on foreground and background colours.

### Colour Venom Bitmaps

Colour bitmaps use 16 bits of colour data per pixel. They can only be drawn on 16 bit per pixel displays.

Colour bitmaps can have transparent areas. These are defined by enabling Transparency and assigning a transparent 'colour' value in the bitmap header. Pixels with this value 'colour' aren't drawn on the display, but instead the original display shows through. The transparent 'colour' can be defined uniquely in each bit map and can be any 16 bit value. For example you might decide to use the value \$FFFF as your transparent 'colour', if you knew that \$FFFF was not used for any real colour in the bitmap. If you need to set a transparent colour in a bitmap, then

the 'transparency flag' must be set in the bitmap data.

Note: This is different to the TRANSPARENT 'colour' you can set for any of the [Pens](#).

The bitmap data is usually supplied in the form of a file (in the Flash Filing System) or an [Array](#). The data format is as follows:

- bitmap width in pixels - 16-bit value
- bitmap height in pixels - 16-bit value
- bitmap colour depth in bits per pixel (e.g. 1 or 16) - 16-bit value. If bit 8 is set then Transparency is enabled.
- 'Transparent' colour - the colour value in *this* bitmap assigned to be transparent if Transparency is enabled.
- Bitmap data - in rows of bytes or 16-bit words
  - For 1 bit-per-pixel bitmaps: the MSBit of the byte appears on the left, and the LSB on the right, in the image
  - The first byte or word of the data is at the top-left of the image.
  - The next byte or word continues across the top of the image, until the end of the first line.
  - A 1 bit-per-pixel image is left justified in the data if it doesn't have a pixel width that is a multiple of 8.
  - If the Array is of 8-bit values then remember that all the values above are 'little endian' - i.e. the Least Significant Byte comes first.

*(All 16-bit values are Little Endian - ie. least significant byte first).*

The following example sets a couple of bitmap images:

```
; Bitmap for the tick that says a tickbox is selected.
Array tick(8)
  $06 ; 6 pixels wide
  $00
  $06 ; 6 pixels high
  $00
  $01 ; 1 bit per pixel
  $00
  $00 ; [No transparent 'colour' in mono bitmaps]
  $00
  %00000100 ; The bitmap data...
  %00001100
  %00011000
  %10110000
  %11100000
```

```

    %01000000
End

```

If you are creating 1-bit-per-pixel bitmaps in an array by hand it's often useful to use the binary notation (e.g. %10101010) within 8-bit data Arrays.

```

; Bitmap to illustrate colour and transparency.
Array image(16)
    6 ; 6 pixels wide
    7 ; 7 pixels high
    16 + %100000000; 16 bits per pixel, transparency needed.
    $8000 ; The colour we will use here as 'transparent'.
    $8000,$8000,$3974,$3974,$8000,$8000,
    $8000,$3974,$3974,$3974,$3974,$8000,
    $3974,$3974,$6342,$6342,$3974,$3974,
    $3974,$3974,$6342,$6342,$3974,$3974,
    $3974,$3974,$3974,$3974,$3974,$3974,
    $8000,$3974,$3974,$3974,$3974,$8000,
    $8000,$8000,$3974,$3974,$8000,$8000
End

```

```

To init
    Make glcd GraphicsLCD(2)
    glcd.Bitmap(0) := tick.Address ;Register bitmap 0
    glcd.Bitmap(1) := image.Address ;Register bitmap 1
    Start Every 100 glcd.Update ;task to update display
End

```

```

To main
    glcd.Bitmap(0,30,30) ;plot bitmap 0 at position (30,30)
    glcd.Bitmap(1,50,30) ;plot bitmap 1 at position (50,30)
End

```

## Macros

### Useful macros for Windows bitmaps

```

#Define IS_BMP(ADD) (??(ADD) = $4D42) ; Is BMP format.
#Define BMP_W(ADD) ??(ADD + 18) ; BMP width
#Define BMP_H(ADD) ??(ADD + 22) ; BMP height

```

The ADD parameter is the address of the bitmap in memory.


### Useful macros for Venom bitmaps

```

#Define VBM_W(ADD) ??(ADD) ; VBM width

```

```
#Define VBM_H(ADD) ??(ADD+2) ; VBM height
```

 See also [Array](#).

## Box

```
Box (Int x, Int y, Int dx, Int dy [, Int border])
```

Draws a box on the display with an optional border. The box area is filled with the current fill [Pen](#), which can be 'TRANSPARENT'.

The first two parameters (x, y) specify the pixel coordinates of one corner of the box, and the second two parameters (dx, dy) specify the size of the box in pixels relative to (x, y); dx and/or dy may be negative.

The optional **border** specifies the type of border to use - see below. If the **border** parameter is not present or is zero no border is drawn.

## Box Borders

There are three major border types: 'Flat', '3D' and 'rounded corners'

### Flat borders

Flat borders are simply one or more lines drawn around the Box with the current foreground [Pen](#).

The flat boarder styles are based on binary patterns. E.g. %101 would produce a border consisting of 2 black lines separated by a white line. The least significant bit (bit 0) controls the *outermost* border line.

▼ The largest flat border number is 255 (\$FF)

### 3D borders

3D borders are arrangements of lines drawn in shades of (normally) grey, to give the impression of a raised or lowered region, or a region bounded by a groove. The colours used to draw these are given by the [Pen](#) message.

These borders are generally intended to be used against a background of light grey, and the box is generally filled with light grey. Some borders can also work well filled with white or another light colour.

The border types are given in this table, and an idea of what each border looks like is shown in the image below.

Border name	Border number code	Example usage
Raised Button	\$100	For dialogue box buttons (Currently the same as \$102)
Lowered Button	\$101	For dialogue box buttons when pressed. (Currently the same as \$103)
Raised 2 pixels	\$102	For dialogue boxes
Lowered 2 Pixels	\$103	For text boxes, tick boxes, etc
Groove	\$104	For separating out a panel within a dialogue box
Raised 1 pixel	\$105	For separating out a panel within a dialogue box
Lowered 1 pixel	\$106	For separating out a panel within a dialogue box



### Rounded Corners

These consist of a 1 pixel outline in the current foreground (Pen 0) colour, with rounded corners. This type of border is selected by a value of  $\$200 + r$ , where  $r$  is the radius of the corners and have a maximum value of 255.

The radius cannot be greater than half the width or height of the box, and is automatically reduced if too big, resulting in a circle if the original box was square.



See also [Pen](#)

## Format

**Format** (Int **index**) := Int **n**

**Format**

Format allows you to set various formatting options for printing to the GraphicsLCD.

Form at inde x	Formattin g applied	Description	Value range	Def ault val ue	Reset conditions (Reset to default value)
0	Word wrapping	When this is 0 ( <i>off</i> ) each line of text is clipped by the TextBox. If it is non-zero ( <i>on</i> ) the text word-wraps inside the TextBox.	0 or non-zero	0	<b>TextBox</b> message
1	Explicit CR depth	If this is non-zero then <b>CR</b> will drop the cursor <b>n</b> pixels independently of the current font's size.	0-255	0	<b>TextBox</b> message
2	Print at monospac e pitch	If this is non-zero then text will print with each character centered inside monospaced 'cells' <b>n</b> pixels wide. This can be useful for printing columns of numbers or spacing out text.  <i>This feature does not work with the newer anti-aliased fonts (font format 5, see <a href="#">FontData</a>).</i>  <i>This feature is not usually needed if the numeric characters in a font are already set in cells of constant width, which is true for many fonts including the internal fonts in the later releases of Venom.</i>	0-32 pixels for colour displays. 0-24 pixels for monochrome.	0	<b>TextBox</b> message



3	Background padding	When the text background colour is not transparent, a rectangle of the background colour is drawn, over which the text is printed. This format value specifies how many pixels the background rectangle extends beyond the text in all directions.	0-255	0	<b>TextBox</b> message
4	Tab spacing	This sets the spacing between tab positions in pixels. When a tab character is printed to the display the cursor moves to the right, to the next available tab position. Tab positions are measured relative to the left hand side of the current TextBox, or the last explicit cursor movement (GotoXY, Htab, Vtab, Home, ...)	0-255	10	
5	Set up capture for position of Nth character	This sets a counter so that when the given number of characters have been printed to the display, the position of the Nth character is captured and later available in the properties Xpos and Ypos.	16-bit signed		<b>Format (5) := -1</b> <b>TextBox, CLS, Home</b>

**obj.Format(0) := True ; turn on word wrapping.**

### Resetting Format options

If you send the **Format** message with no parameters then all the default values are re-applied.

**obj.Format ; Reset all Format options.**

All format options are also reset on the next **TextBox** message.

### Rotated display

You can rotate the whole display on some devices. See here: [Rotated Display](#).

## FontData

**FontData** (Int **Font\_no**)  $\Leftrightarrow$  Int **memory\_address**  
**FontData**

The FontData message allows extra fonts to be registered with the GraphicsLCD object, in addition to, or instead of, the embedded 'system' fonts.

The Font Manager built into VenomIDE can convert TrueType fonts into the VAF format used in Venom.

### Parameters

**Font\_no** is the new font's font number. It must be in the range 0-255. You can overwrite the [system font](#) numbers (0, 1 & 2) if necessary.

**memory\_address** is the address (in memory) of the font data. This data is usually held in a file in the VM2's flash filing system, where the address of the data may be found using the [Address](#) message.

Reading **FontData** returns this address, so it is possible to 'move' the system fonts to new font numbers.

### Example

```
glcd.FontData(3) := flash_file_system.Address("MyFont.vaf") ; fon
```



Note: if the font data file is not found then 'address' returned will be zero; in that case system font 0 (the smallest) will be substituted so you can still see the text.

### Revert to internal fonts

When the **FontData** message is sent with no parameters it clears all the fonts that have been registered and defaults back to the embedded 'system' fonts.

*This may be useful when updating font files during a firmware update process, during which time the font files will cease to exist, and attempting to print them will result in a runtime error.*

### Example

```
glcd.FontData ; Revert to internal fonts.
```



Up to 256 fonts can be registered, numbered in the range 0 - 255.



For a detailed description of the font data formats used see [here](#).

### Font data formats

*This information is provided for completeness - you don't need to know this if you use our font conversion utility.*

Here is a description of the 'Venom' bitmap font formats used in Venom2:

### Header for Font types 4, 5 and 6

The first 10 bytes are a 'header' that lists general data about the font. After this are the character map (that indexes each character in the font), and the font bitmap data itself.

Byte #	Function
0	Font data format: 4: Proportional, 5: Monospaced, 6,7,8: Anti-aliased (VAF format)
1	Font character width, this value is only used in Monospaced fonts, though it may be present in proportional font data converted from other formats.
2	Font height (Total number of pixel lines in a character)
3	ASCII code of first character defined in the font (in the range 0-255, often 33: '!')
4	ASCII code of last character to be defined in the font (in the range 0-255)
5	Ascent: Number of pixel rows above the baseline pixel row. The baseline is the pixel row above any decenders in the font.
6	Inter-character gap: Number of pixels between each printed character. Only necessary where there are no blank pixels included in each character.
7	Width of the space character in pixels – only used in proportional fonts.
8	'External Leading': extra lines above the font that are added to the font's height to give more separation between lines on carriage return. Can usefully be set to 0.
9 - 11	Reserved value – set to 0
12 onwards	Character map & font bit-map data. This differs depending on the type of font.

### Character map for 1-bit per pixel proportional fonts

Byte #	Function
0	Character width in pixels.
1	Reserved – set to zero.

2	Character data offset - LSB
3	Character data offset - MSB

The character map allows each proportional font character to be found within the bitmap data. Each map entry holds the character width and the offset of the bitmap data. The offset is relative to the start of the array or file.

Because there is one character map entry for each character in the font, the offset for the bitmap data for each character may be determined:

*For a font in which 100 characters are defined, the first character bitmap has an offset of  $10 + 4 \times 100 = 410$ , or \$9A, \$01 as two bytes in hex.*

Monospaced fonts have no character map - the header is followed directly by the font bitmap data. in the same format as the bitmaps for Proportional font type 4. The width of every bitmap is the value found in the font header (see above).

### 1-bit bitmap data

Each pixel in a character is printed as either the foreground or the background colour.

Each character is represented by a number of bytes -  $n \times m$  - where  $n$  is the number of bytes in a row, and  $m$  the number of rows.

The data is listed starting with the first row, and then each subsequent row follows.

Each row of pixels in a character occupy  $n$  bytes - where  $n$  bytes is big enough hold the pixel width of the character, where 1 bit represents 1 pixel. With narrow characters only 1 byte is needed per row.

The left-most pixel in a row maps to the most significant bit of the first byte in the row.

The number of rows in each character is given by the *font height* in the header listed above.

Any character set can be created in this way, although in the case of some (such as Chinese, Arabic etc.) the characters created will not relate directly to the character code used to print them. You can define any group of consecutive characters between 33 and 255.

### Font Type 6, 7 & 8 - Anti-aliased Fonts

These fonts have a numerical value per pixel, which determines the proportion of background and foreground in the colour of the pixel, from 0 for pure background to \$FF (types 6, 7) or \$3F (type 8) for pure foreground. The fonts can be created from TrueType fonts using the Font Manager built into VenomIDE. This type of font can create much better looking text than 1-bit fonts, with smoother edges on curves.

### Character map for Anti Aliased fonts

The character map contains 32-bit values, least significant byte first. Each value in the map is the offset (bytes from the start of the file) of the bitmap data structure for the corresponding character.

### Bitmap Data for font type 6

The Bitmap data for each character consists of a fixed format header followed by variable number of pixel bytes.

Byte #	Function
0	width of bitmap in pixels = width in bytes (unsigned)
1	number of rows of pixel data (unsigned)
2	Height of top row above cursor position (signed)
3	offset from cursor position of left edge of bitmap (signed, positive = right)
4	cursor advance in pixels, to next character (unsigned)
5	reserved, set to 0
6 +	(rows x width) bytes of pixel data from left to right, top row first

### Header for Font Types 7 and 8

As with types 4, 5 and 6, the first byte always identifies the font type.

Byte #	Function
0	Font data format: 7: 16 bit metrics, 8: as 7 but with run length compressed bitmap data
1	Unused; padding to align 16 bit data on even memory addresses.
2,3	Width: maximum width of any glyph (LS byte first)
4,5	Height: height required to accommodate all glyphs
6	First code point in table
7	Last code point in table
8,9	Ascent above baseline, LS byte first
10, 11	Width of space char, LS byte first
12, 13	Extra leading; space between rows.
13,14	Padding to make header size of multiple of 4 for more efficient memory alignment

<i>12 onwards</i>	Character map & font bit-map data. This differs depending on the type of font.
-------------------	--

### Bitmap Data for font type 7, 8

The Bitmap data for each character consists of a fixed format header followed by variable number of pixel bytes.

Element	Length (in bytes)	Description
width	2 unsigned	Width of each row of pixel data, in bytes
rows	2 unsigned	Number of rows of pixel data
top bearing	2 signed	height of top row above cursor (baseline)
left bearing	2 signed	offset of left edge from cursor
advance x	2 signed	cursor x advance in pixels
advance y	2 signed	cursor y advance in pixels
bitmap	width x rows (type 7) variable (type 8)	8 bit pixel data from left to right, top row first (see below)

#### Bitmap (type 7)

Each byte corresponds to 1 pixel in the rectangular bitmap. The value shows the proportion of foreground and background colour for that pixel: 0 is 100% background and 255 is 100% foreground.

#### Bitmap (Type 8)

This data is run-length compressed.

The pixel values are 6 bit: 0 for 100% background, 63 for 100% foreground.

the top two bits of each byte determine the use of the remaining 6 bits:

Bit 7	Bit 6	Bits 5 - 0
0	0	0 - 64 representing 1 - 64 bytes of 100% background colour
0	1	0 - 64 representing 1 - 64 bytes of 100% foreground colour
1	0	fg/bg proportion for 1 pixel: 0 = 100% background; 63 = 100% foreground
1	1	Reserved

## Line

**Line** (Int **x1**, Int **y1** [, Int **x2**, Int **y2**])

Draws a line from the end of the previous line to the display coordinates (x1, y1), or if the optional parameters x2 and y2 are present, from (x1, y1) to (x2, y2).

The line is drawn in the current foreground colour (see [Pen](#)).

*Note that (0,0) is the pixel at the bottom left of the display.*

▼ Coordinates outside the edge of the display are individually limited at the edge of the display, i.e the gradient of the line is not preserved.

🔍 See also [Pen](#)

## Off

**Off**

The GraphicsLCD Off message turns off the PWM signal generated by the VM2-D2 TFT driver IC.

This PWM signal is intended for backlight control.

🔍 See also [On](#)

## On

**On**

**On** (Int **Width**, Int **Period**)


The GraphicsLCD On message turns on the PWM signal generated by the VM2-D2 TFT driver IC.

This PWM signal is intended for backlight control.

If On is called without parameters then the PWM signal is turned on at the maximum duty cycle (255/256) at a default PWM period. This PWM period is equivalent to a frequency of about 250Hz when driving a typical VGA display. The basic PWM period is derived from the LCD driver clock and so depends on the LCD parameter settings.

If On is supplied with parameters, the first parameter sets the duty cycle in the range 0-255 out of 256, where 0 is off and 255 is maximum. The second, optional, parameter (in the range 0-255) sets the period. A value of 0 gives the maximum frequency (minimum period). Period values over about 10 (on a VGA display) may result in visible flickering of the screen.

If the period parameter is omitted then the default value is used (see above).

 See also [Off](#)

## Pen

### Pen

**Pen** (Int **FGcol**, Int **BGcol**)

**Pen** (Int **pen\_number**)  $\Leftrightarrow$  Int **colour**

The Pen message allows you to set the colour of each of the different 'pens' used in the display.

There are nine pens in all, but only the first three are likely to need changing frequently. These three are the current *foreground* pen - used to draw text and lines; the current *fill* pen - used to fill Boxes and TextBoxes, and the current *text background* pen - sometimes used to provide a background for text.

Any of these pens may be set to 'transparent'. In particular, the text background pen is automatically set to transparent when any [TextBox](#) is defined.

### 'Default' colours

The next two pens hold the *default* foreground and fill pen colours. The *current* foreground and fill pens are reset to these colours when CLS is used to clear the display.

### '3D' colours

The last four pens are only used when drawing the 3D box borders and are usually left at their default values.

## Setting pen colours

You can set or read any of the pen colours using each pen's number, for example:

```
lcd.Pen(1) := RED ; Set the fill colour
fg := lcd.Pen(0) ; read the foreground colour
```

## Quick setting

You can quickly set the foreground and background pen colours by using Pen with two parameters, for example:

```
lcd.Pen(RED, BLUE)
```

## Resetting

If you send the Pen message with no parameters then Pens 0 - 2 will reset to their default values.

- Pen(0) will be loaded with the value of Pen(3)
- Pen(1) will be loaded with the value of Pen(4)



- Pen(2) will be set to Transparent.

For example:

`lcd.Pen`

### Background colour

Setting Pen(4) is the best way to set the background colour for the whole display.

`Pen(4) := BLUE_WASH`

Pen number	Name	Usage	Reset
0	Current Foreground pen	Draw Text and lines	Reset to <i>Default Foreground pen</i> on <b>CLS</b> and <b>Pen</b> with no parameters
1	Current Fill pen	Fill Boxes and TextBoxes	Reset to <i>Default Fill pen</i> on <b>CLS</b> and <b>Pen</b> with no parameters
2	Current Text Background pen	Fill text and 1BPP bitmap backgrounds.	Reset to the <i>transparent</i> colour on <b>TextBox</b> , <b>CLS</b> and <b>Pen</b> with no parameters
3	Default Foreground pen	Sets the value of Pen(0) when the display is cleared with CLS	This is set to <i>black</i> on creation of the GraphicsLCD object.
4	Default Fill pen & background colour	Sets the value of Pen(1) when the display is cleared with CLS, also used as the background colour when the display is cleared.	This is set to <i>light grey (in RGB565)</i> on creation of the GraphicsLCD object.
5	Default dark grey for 3D borders	The dark shadow colour for 3D borders.	This is set to an appropriate grey (in RGB565) on creation of the GraphicsLCD object.
6	Default mid grey for 3D borders	The mid shadow colour for 3D borders.	<i>As above</i>
7	Default light grey for 3D borders	The light shadow colour for 3D borders.	<i>As above</i>
8	Default highlight colour for 3D borders	The bright lines in 3D borders.	<i>As above</i>

## Colours

Colours in Venom2 are represented by 32-bit integer values. The colours are different for 1 bit-per-pixel displays and 16 bit-per-pixel displays - and are sometimes different between some 16 bit-per-pixel displays.

### 16 bit-per-pixel colours

For 16 bit-per-pixel displays the colours depend on the internal display controller data format. One format, used in VM2D, is called RGB565 - where the colours red, green and blue are packed into a 16-bit word with resolutions of 5, 6 and 5 bits each. The *transparent* 'colour' is defined outside of the 16-bit range.

The following macros show how you can define colours:

```
#Define TRANSPARENT $10000

; Macro to help with colour declarations.
; (If the macro parameters are constant then this will evaluate to a constant)
#Define RGB(red,green,blue) (red << 11 + green << 6 + blue) ; This is a macro

;Primary colours.
#Define BLACK $0000
#Define WHITE $FFFF
#Define RED RGB(31,0,0)
#Define GREEN RGB(0,31,0)
#Define BLUE RGB(0,0,31)
#Define YELLOW RGB(30,31,0)

;'Washes'
#Define BLUE_WASH RGB(16,16,31)
#Define YELLOW_WASH RGB(31,31,16)
```

### Examples

```
g.Pen(0) := YELLOW ; Set foreground colour to yellow.
g.Pen(1) := TRANSPARENT ; Set fill colour to transparent.
```

### One bit-per-pixel colours

The following macros represent 'colours' that may be used with 1 bit-per-pixel displays:

```
#define TRANSPARENT 0 ; Pixels are unchanted
#define BLACK 1 ; Pixels are set to 'on'
#define WHITE 2 ; Pixels are reset to 'off'
#define INVERT 3 ; Invert the pixel state.
```

## Reset

### Reset

Not implemented yet.

## TextBox

### TextBox

```
TextBox(Int x, Int y, Int wid, Int hgt, [Int border,  
[Int font, [Int padding,]])
```

TextBox defines a rectangular area within which all subsequent text is printed - until another TextBox is defined.

If **word wrapping is turned off** then the text is clipped (on the right) to fit the width of the TextBox.

If **word wrapping is turned on** then the text will wrap on to subsequent lines so as to fit inside the TextBox. Text will wrap at a space character, or in the middle of a word if there is no space character.

Text is clipped at the bottom of a text box, though not at the top (so you can have text that is displayed above the text box).

The first two parameters (**x**, **y**) specify the pixel coordinates of one corner of the TextBox, and the second two parameters (**wid**, **hgt**) specify the width and height of the TextBox in pixels; **wid** and/or **hgt** may be negative. The coordinate origin is always relative to the bottom left corner of the TextBox.

TextBox will also draw a visible box on the display (as a visible background or container for the text) if required.

If the current fill [Pen](#) is not transparent then the TextBox is filled with this pen colour - just like a normal [Box](#).

If the optional **border** parameter is supplied then a [border](#) is drawn within the TextBox rectangle and the text margins are set immediately inside the border. (If the border is negative then the text margins are set to the edges of the box rather than inside the border. This can be useful during development.) For 'rounded corner' border style, the text margins are set so the text will not collide with the corners.

If the optional **font** parameter is supplied then this sets the current font.

If the optional **padding** parameter is supplied then the left and right hand margins are indented by this number of pixels inside the border.

When the TextBox is created, the text background pen is set to TRANSPARENT, justification is set to Left and the cursor is placed at the Home position *for the current font*.



If you see text that prints in one position the first time, and then prints in a slightly different position subsequently, it may be because the font was changed after setting the TextBox.

You can use Left, Right, Centre, HTAB, VTAB, GOTOXY, [etc](#) to move the text cursor around and otherwise format text within the TextBox.

Note: though it is not possible to print text below the bottom of a TextBox, it is possible to use VTAB, [etc](#), to position text at or above the top of a TextBox.

Note that nothing is seen on the display until an [Update](#) message is issued.

### Default text area is the whole screen

If TextBox is called with no parameters this resets the current text area to the whole of the LCD display area with no background, border or padding.

**g.TextBox**

### Examples

The following example creates a TextBox and then prints some text into it:

```
Make g GraphicsLCD(1)
Start Every 100 g.Update
g.TextBox(20, 20, 100, 100, $102)
Print To g, "Hello World"
```

## Update

### Update

The Update message forces all the changes that have been made (to the local-memory copy of the display data) to be sent to the display itself. Nothing will be displayed on the physical display device until an Update message is issued. Whilst individual Update messages can be used, e.g.:

```
Print To w, "Hello"
panel.Update
```

When you first start to use the GraphicsLCD it can be useful to start a separate task to keep the LCD updated:

```
Start Every 100 panel.Update
```

However, for most applications it often makes sense to put explicit updates in your code so the display gets updated at exactly the right point.

- 🌀 The GraphicsLCD driver objects will only update the areas of the display that have been written to since the last Update.

There may be different algorithms used by the different drivers for each display:

The most common algorithm keeps track of a rectangular area that encloses all the changes that have been made. The speed of display activity can therefore be improved by updating relatively quickly, before changes have been made over a large area.

A less common algorithm, usually used for smaller displays, keeps track of changes in horizontal bands across the display. The speed of display activity can therefore be improved by confining fast changing graphics to groups of horizontal areas.

## Xpos, Ypos

```
XPos ⇒ Int
YPos ⇒ Int
```

**XPos** and **YPos** return the position of the internal text cursor - that is, the position that the next character will print at – relative to the current [TextBox](#) origin.

## Capture position of Nth character

**XPos** and **YPos** can also capture the position of a particular character by using [Format\(5\)](#) := N to capture the Nth character.

The 'captured position' is turned off at the next **TextBox**, **CLS**, **Home** - or by setting **Format(5)** to -1, whereupon **Xpos** and **Ypos** will report the current text cursor position.

This feature may be used to position a visible text cursor for text editing on the display.

## Example

```
To TextEditorUpdate
  lcd.TextBox(...)
  lcd.Format(0) := True ; Wordwrap on.
  lcd.Format(5) := CursorPosition ; Set up cursor position capture
  Print To lcd, Text ; Print text
  Print To lcd, GotoXY(lcd.XPos, lcd.YPos), "| " ; Print cursor
```

**End**

## PRINT TO

**Print To** <GraphicsLCD>,<print list>

Print may be used to print text to the GraphicsLCD object. You can also print text using the [PrintF](#) message.

Printing text on the GraphicsLCD is relatively complicated because there are many different formatting possibilities, including:

- Position of the text on the display
- Font size and typeface
- Justification (left, right or centre)
- Word wrapping on or off
- Text foreground and background colours
- Margins

Some of these format options are specified by using special Print keywords - see below - and others are specified by sending messages to the GraphicsLCD object.

These messages are

- [TextBox](#) - specify the area within which to contain text output
- [Pen](#) - specify the colours to print with
- [Format](#) - specify some other miscellaneous format options

## Print keywords

These print keywords control printing from within a [Print](#) statement. You can use as many of them as you need in a single Print list.

### CLS

Clears the entire display using the default fill [Pen](#). It also resets the current [TextBox](#) to the whole display area with no border or padding.

Almost all text formatting is reset: Left justification is applied, the cursor is sent to the Home position for the current FONT (which is left unchanged), and all [.Format](#) options are reset.

The current pens are reset. See [Pen](#).

No change will be seen until an Update message is sent.

**Print To** **glcd**, **CLS**

### Home

Positions the text cursor at the top left of the current [TextBox](#), correct for the current font, and resets the justification to Left.

```
Print To glcd, Home
```

### CR

Positions the text cursor at the beginning of the next line down, according to the current font size, [TextBox](#) padding and justification. Any text that overflows the bottom of the TextBox then it is not printed.

```
Print To glcd, "Hello", CR
```

You can force the vertical movement of CR independently of the font size, using the [Format](#) message.

### Left, Right, Centre

These set one of these justification styles until the next justification command, TextBox or CLS/Home. They also put the cursor at the left, centre or right of the current [TextBox](#).

```
Print To glcd, Right, "Some text", CR
```

If you want to print justified text relative to a point in the TextBox, use GOTOXY, VTAB or HTAB to position the cursor *after* the justification command.

```
Print To glcd, Centre, HTAB(-20), "abc", CR
```

Note: you can also change justification by using an embedded [escape sequence](#).

### HTAB (Int pixels)

This moves the cursor horizontally across the display in pixel units. The cursor is moved relative to the *anchor position*, which is set by the last explicit cursor movement (such as HTAB, VTAB, Left, Right, Centre, CR, Home). Negative values will move the cursor leftwards.

CLS, Left, Right and Centre set the anchor position to the left, right or centre of the current [TextBox](#).

CR and Home also set the anchor position.

The HTAB position will always be limited to inside the current TextBox border and padding.

The **pixels** parameter is a signed 16-bit integer (range -32768 to 32767)

### VTAB (Int pixels)

This moves the cursor vertically up or down the display in pixel units. The cursor is positioned relative to the *anchor position*. (The anchor position is described in HTAB above).

Negative values will move the cursor *upwards*. The downward movement is limited to the bottom of the [TextBox](#), but the upward movement is not limited - you can print text at or above the top of a TextBox.

The **pixels** parameter is a signed 16-bit integer (range -32768 to 32767)



When a TextBox is first set the tab anchor position is set at the Home position *for the current font*. To avoid confusion it's best to set the font in the TextBox parameters, or at least before the TextBox is set.

#### **GOTOXY** (Int x, Int y)

Positions the text cursor at the coordinates x,y *relative to the current [TextBox](#) origin*

GOTOXY (0,0) refers to the bottom left hand corner of the TextBox, inside any border and padding.

#### **FONT** (Int font)

Set the current font to print to the display. Several embedded fonts are supplied. These are listed in the table below, and you can [add more fonts](#).

Font Number	Font style	Proportional / Monospaced	Character Height (pixels)	External Leading* (pixels)
0	Sans Serif	Proportional	10	1
1	Sans Serif	Proportional	15	2
2	Sans Serif	Proportional	19	3

\* *External leading is the amount of blank space left between lines of text when printing CR.*

Fonts may be changed at any point in a Print statement.

```
Print To glcd, FONT 0, "Hello", FONT 1, " World"
```

You can also specify the font when setting the [TextBox](#), or change fonts with an embedded [escape sequence](#).

If you specify a font that hasn't been defined then FONT 0 is substituted.

You can use the [Format](#) message to arrange for any font to print in a 'monospaced' format - eg. for printing columns of numbers.



## Bitmaps embedded in text

You can embed bitmap images in your text, as if the bitmap were a single character, though you can also use this method to display bitmap images that are *much* larger than a character if you like.

Bitmaps are embedded in a quoted string by placing `\^hh` in the text at the point at which the bitmap should appear (where *hh* is the bitmap's number represented as a 2-digit hexadecimal number). You can set this number by registering the bitmap with the GraphicsLCD object. See [below](#).

For example, the following would write `20°C` to the graphics LCD provided that a bitmap of the degrees symbol ° had previously been created and registered as bitmap number 3.

```
Print To glcd, "20\^03C"
```

Note: due to improvements in Venom, printing a degrees symbol might now be better handled by using a font containing the degrees symbol, and the following code:

```
Print To glcd, "20°C"
```

## Registering bitmaps for printing

```
obj . Bitmap (Int bitmap_ref) := Int address
```

This form of the Bitmap message registers a bitmap with the GraphicsLCD object so it may later be printed. You have to supply a unique reference number (in the range 0-255), which you can use to refer to the bitmap later.

### Example: registering bitmaps

```
To init
  Make glcd GraphicsLCD(1)
  glcd.Bitmap(3) := degrees.Address ;Register bitmap 3
  Start Every 100 glcd.Update ;task to update display
End
```

```
To main
  Print To glcd, "20\^03C"
End
```

```
; Bitmap for a small degrees symbol.
; Note: a degrees symbol may be present in the font you are using
; so this would then be unnecessary.
```

```
Array degrees(8)
  $04 ; 4 pixels wide
  $00
  $04 ; 4 pixels high
  $00
  $01 ; 1 bit per pixel
```

```

$00
$00 ; [No transparent 'colour' in mono bitmaps]
$00
%01100000 ; The bitmap data...
%10010000
%10010000
%01100000
End

```

 See also [Bitmap](#), [Printf](#)

## HashGenerator

The HashGenerator object enables a secure message digest ("hash") to be made from any set of data from an empty string to a large file.

The hash value generated is 256 bits (32 bytes) long and the algorithm (known as SHA-2) is designed to make it almost impossible to tamper with the data in any way that would not change the digest value, and equally difficult to find out the full original data content even if the hash value and part of the data is already known.

It is also a useful way of generating a numerical key for encryption purposes from text such as a password or phrase of any length.

### Summary of messages

**Make**  
**Get**  
**Put**  
**Reset**  
**Print**

### Example of use

```

To init
  Make hash Array(8, 32, 0) ; for storing hash result
  Make sha HashGenerator
  Make fs Filesystem("SD") ; file system on SD card
End

To main
  Print SHA, CR ; SHA for empty string

  sha.Put("abc")

```

```
Print SHA, CR    ; SHA for string "abc"

sha.Put("abc")
sha.get(hash)    ; get the hash value into array this time
                  ; print the value from the array (should be the same as above)
Repeat 32 Print ~hash.(index0):-2 Print CR

; Make the SHA-256 of a file, also illustrates Printf with %o
f := fs.open("myfile.txt", Char)
Repeat f.length sha.put(f.get)
Printf("SHA of myfile.txt is\n%o\n", sha)
End
```

This produces the result:

```
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
BA7816BF8F01CFEA414140DE5DAE2223B00361A396177A9CB410FF61F20015AD
SHA of myfile.txt is
6c9dc57ad9b3bef88ea57b454bb678246d5de6748b711c71fabae7af5539147
-->
```

## Creation

**Make** <object> **HashGenerator**

Creates a SHA-256 bit Hash generator in its initialised state.

## Get

**Get**(Array **hash**)

The parameter must be an array of 32 bytes

This message finalises the hash creation process and writes the result to the array supplied as a parameter.

After this has been done, the object is reset.

## Put

**Put**(Int n)

The parameter is treated as a single byte or character. Only the low 8 bits are used.

**Put**(object s)

The parameter can be a fixed string, string object, array of 8 bit integers or buffer of 8 bit integers or text.

If it is a buffer it must have no more than 256 elements.

In both cases the current state of the hash generator is updated with the data supplied.

You can repeat the **Put** message any number of times; the resulting hash will be that of all the **Put**

data concatenated.

## Reset

**Reset**

The Hash Generator is set to its initial state.

This is not often needed as the generator is reset automatically by Print and Get.

## PRINT

**Print** <HashGenerator>

This finalises the hash computation, prints the result as a 64 digit hex string and resets the hash generator.

## HTTPServer

The HTTP object can be used as the basis of a web server thread. It creates a TCP listener, parses incoming http requests and creates the necessary headers and other protocol for the response. The application program thus only has to identify the URI (Uniform Resource Identifier) requested and create a response either by printing HTML or by sending buffer or file

contents to the HTTP object.

The HTTP protocol object can work automatically with a local file system, so that an incoming request for a file will result in that file being sent without any Venom program involvement at all. The object will guess the correct MIME data type for common file naming conventions and will send sensible default cache control headers to improve efficiency.

Sending HTML code in response to a request is a simple matter of using Print statements to generate the required text.

The object supports http GET and POST requests enabling HTTP form data to be sent, so a variety of browser-generated data or commands can be sent to the HTTP server.

Cookies are implemented, enabling the illusion of a persistent connection ("session") with a browser. For example, after logging in with a user name and password, a user can continue to use the server which remembers the logged-in state of the user.

The creation of HTML code is a wide-ranging topic that is not documented here.

See also [Notes on Use](#) and [TCP/IP Networking](#)

## Summary of messages

**Make**  
**Address**  
**Cookie**  
**Count**  
**Debug**  
**Flush**  
**Format**  
**Get**  
**Mapping**  
**Match**  
**Name**  
**Period**  
**Put**  
**Redirect**  
**Valid**  
**Value**  
**Print**  
**Put**  
**Queue**  
**Print To**

## Notes on Use

### Example of a Minimal Web Server

You can make the VM2 serve HTML code from files very easily if no server-side processing is required.

Here is the code for a minimal web server, using files in Flash memory. If you have ever created a simple web site using html code, images and (optionally) a CSS stylesheet you can copy the site contents to the Flash filesystem using USB and this Venom code will serve it. To access it from a browser, put in the URL "vm2server".

Note we create two server tasks; you may create as many as resources will allow but it's advisable to have more than one server

thread running, especially if your html code has embedded images or is likely to be accessed from more than one location simultaneously.

```
TO init
  Make fs filesystem("fla")
  Make eth Ethernet(nil, "vm2server")
END

TO main
  Start server ; start two server tasks
  Start server
END

TO server
  LOCAL http := New HTTPServer(fs)
  LOCAL path := New string(100)

  FOREVER
  [
    http.get(path)

    ; if you need to process requests not matching an existing fi
    ; the code to do so is inserted here. E.g. uncomment the next
    ; display a directory using the function listed below
    ; If fs.Find(path) = 2 ; path is a directory
    ;   ServeDirectory(http, "VM2 Server", path)

    http.Flush
  ]
END
```

## Listing Directories

If a request is made which matches a directory in the file system, this is always returned by the HTTP **Get** message and must be processed by Venom code. A simple function to do this is shown below. To use it, uncomment the lines indicated in the code above.

The code below assumes that **fs** is the name of the file system used by the HTTP server. If you use this code and your file system global variable has a different name, you'll have to edit it. You can, of course, also edit it to produce listings in many different formats. See [Print Filesystem](#).

```
To ServeDirectory(http, title, dirpath)
    http.printf("<html><head><title>%s: %s</title></head><body>\n"
    http.printf("<h1>Directory of %s: %s</h1>\n<pre>", title, dirp
    print to http, fs:dirpath:"%z %Y-%m-%d %h:%M:%s %b <a href=\"%
    http.printf("</pre></body></html>")
End
```

## VHTML - HTML with embedded Venom Variable Names

A special variant of HTML code allows Venom global variable names to be embedded in what would otherwise be an HTML file. Such a file is identified by its name ending with the string ".vhtml".

Inside the file, variable names are enclosed between backslash ('\\') characters.

If a literal backslash is needed in the HTML code it is represented by an empty name i.e. two consecutive backslashes.

When the file is sent in response to an http request, the global variable table is searched for a matching name. Type conversions are performed according to the C language `printf()` formatting conventions as follows:

- integer: "%ld"
- float: "%g"
- strings and text buffers are sent verbatim.
- Unmatched variable names are substituted with the string "?xxxx", where xxxx is the name.
- Variables whose type is invalid are substituted with the string "?type:xxxx" where xxxx is the name.

It's worth emphasising that variables substituted in this way do not necessarily have to be in ordinary display text. They could appear in the attributes for an HTML tag, or in part of an embedded CSS style specification for example. This is useful for changing default values of form input elements, and offers creative possibilities such as data-driven changes of colour or bar-graph charts.

 See also [Printf and TextBlocks](#)

## Notes on URIs

An http request is typically of the form:

```
GET http://www.example.com/index.html?test=abc HTTP/1.1
```

or

```
GET /index.html?test=abc&rate=10 HTTP/1.1
```

The first form is an absolute URI with the scheme (`http://`) and host (`www.example.com`) included. The second form is called a relative URI. Either may be sent by a browser; the http protocol object will skip the host part if present.

The '?' character and following string are optional. This query string, if present, contains data that can be processed by the server. It is typically sent when an HTML form is submitted, in which case it is a sequence of "name=value" pairs separated by '&'. Because certain characters are unsafe to send in a GET query like this they are escaped; the HTTP Protocol object translates these escape codes when the data is retrieved by a **Value** message.

## HTTP Protocol Considerations

### Persistent connections

The HTTP object supports persistent connections, i.e. when a request has been received and the response sent the TCP connection remains open so further requests can be sent in the same connection. The connection is closed if no further requests are received for 5 seconds, or if the remote end closes the connection.

Note that an application using the HTTP object should run several web server tasks. To communicate with one browser needs 2 tasks as the HTTP standard (RFC 2616) recommends the use of a maximum of two simultaneous connections per user agent. More are needed if you expect multiple users to access the VM2's web server concurrently.

### Headers

The HTTP object takes note of the following headers in the request, ignoring others and using default settings.

Cookie	This can be retrieved with the <a href="#">Cookie</a> message
Content-Type	In a POST request, this can be set to <code>application/x-www-form-urlencoded</code> or <code>multipart/form-data</code> . The latter format is usually selected by web browsers when sending a file from a form, as it allows large blocks of data and binary data to be sent.
Transfer-Encoding	In a POST request, a value of <code>chunked</code> can be accepted; otherwise, a <code>Content-Length</code> header is expected.
Content-Length	In a POST request, this implies no transfer encoding: data is sent as-is.
Connection	A value of <code>close</code> is recognised; otherwise the default is <code>keep-alive</code> .
If-None-Match	Used with an Etag value, and when a file is requested, this enables a



	special response to be sent indicating that the client may use a previously cached copy of the file because the file's contents have not changed.
--	---

The headers sent in responses are:

Server: VM2/yyyyymmdd	yyyyymmdd is the venom release date code
Date: Tue, 28 Nov 2006 14:23:12 GMT	The standard format for HTTP dates
Transfer-encoding: chunked	Date generated by Venom <b>Put</b> , <b>Print To</b> and <b>Printf</b> and in a .vhtml file is sent in this format, as the content length is not known at the time the headers are sent.
Content-Length:	For files other than .vhtml, this is sent and the chunked encoding described above is not used.
Cache-control:	Either a max-age value or "no-cache" (see <a href="#">Period</a> message)
Content-type:	See <a href="#">Format</a> message for some default automatic types
Set-Cookie:	Replicates received cookies and also sets a new cookie if requested by a <a href="#">Cookie</a> message.
Etag	For files, this is set to an 8 digit hex number (actually the CRC32 of the file's directory entry) which changes when the file changes in any way.

### HTTP Request Methods

GET and POST methods are supported.

The **Value** message will retrieve variables sent using either GET or POST.

### Uploading Files

A file of any type of data can be sent with the POST method if the http object was created with an associated file system.

The file data is stored in a temporary file whose name is in the form XXXXXXXX.tmp, where XXXXXXXX is a random hexadecimal number. The file name can be retrieved from the HTTP object by **Name(0, str)**, and if desired the original file name can be retrieved with **Name(1, str)**. (see [http.Name](#))

Note that you can use the [Mapping](#) message to restrict the location of the uploaded temporary file to a chosen directory.

### Limitations with Files and POST

Only one file at a time may be uploaded. A future version of Venom may allow multiple files to

be uploaded in a single POST message.

Other than file contents, POST is not suitable for sending pure binary data: in particular, the presence of a NUL (0) byte in the data will cause incorrect results. This limitation may be lifted in a future version of Venom.

When creating an HTML form to upload a file, the `<FORM>` element **must** include the attribute `enctype="multipart/form-data"`

e.g. `<form method="POST" action="process_upload.html" enctype="multipart/form-data">`



Forgetting to do this is a very common error, and results in the form data being silently sent with no file attached.

## Creation

```
Make <object> HTTPServer([filesystem fs, [, Int port]]
)
```

<b>filesystem</b>	Specifies a file system to be searched for files to match incoming http GET requests for serving automatically. If this parameter is missing or <b>nil</b> then no file system will be used.
<b>port</b>	If specified, defines the port number for the TCP listener. By default the object will listen on port 80, which is the standard port number for http.

The **make** statement creates an http server object and associated TCP listener.

If you specify a File system, you may restrict the way it is used by the **HTTPServer** object using the [Mapping](#) Message.



An HTTP Protocol object with default TCP buffer sizes uses 11710 bytes of memory when created

## Address

**Address**  $\Rightarrow$  Int

The **Address** message returns:

- 0 if the http object is in the listening state
- the IP address of the originator of the request if the http object has returned from a **Get**

message or a [Queue](#) message has returned a value of 1

### Example

```
path := http.Get
Print "request from ", http.Address:"IP", CR
```

## Cookie

**Cookie**(<'Text'> name, String val) ⇒ Int

**name** Name of cookie; can be a fixed or variable string or text buffer

**val** String object to write cookie value to, if a cookie with matching name has been set

If a cookie with the given name has been set, sets the string variable to the value of the cookie and returns True (1)

Otherwise, Returns False.

**Cookie**(String name) := String value

Sets a cookie with the given name and value.

This message must be used only after an http.get has returned i.e. after an incoming request has been received.

The cookie is set with a default expiry time of 10 minutes. If a source of calendar time is available

(either the real time clock or an internet time server), the expiry is set as an actual time and date with the cookie "Expires:" attribute. Otherwise a "Max-age" attribute is set with a value of 600 seconds.

The cookie expiry can be changed by a [Period](#) message with the cookie's name as a parameter.

Note that any cookies received are sent back with the default "Expires:" or "Max-Age:" value if they were not processed with the Cookie and Period messages.

### Simple Example of Use

In this code we expect a session id named "SID"

```

LOCAL reqpath := new string(100)
LOCAL temp := new string(100)

http.get(reqpath)
if http.cookie("SID", temp) IsFalse ; if we haven't received a
[
    temp.printf("%08x", system.time(0)) ; could be a random number
    http.cookie("SID") := temp ; set a new "SID" cookie
]

```

### Brief Explanation of Cookies

Cookies are a protocol that enables a web browser to identify itself so that a series of requests to the same server can be recognised by the server as coming from the same source. E.g. if a user had to log in, a cookie will identify that user in subsequent requests so the login screen is not displayed again, also so that the content displayed or other actions taken may be tailored to the user's ID.

### Count

**http.Count**(Str name[, Int post]) ⇒ Int

<b>name</b>	A variable name
<b>post</b>	not present: count both post and GET variables with matching name = 0: count GET variables only = 1: count POST variables only

The **Count** message counts the number of instances of a POST or GET variable with the given name in a request just received.

If no variable with a matching name is found, it returns 0.

See [Value](#) message for more information about GET and POST variables.

### Debug

**Debug** ⇐ Int

This internal flag can be set True (1) or False (0). When True, the http object will show the headers on outgoing responses

(you can show incoming request headers with the [Print](#) message) and occasional other useful information.

## Flush

### Flush

Transmits any remaining data in the response buffer, then returns the http object to a listening state.

If no data was sent to the object following a successful **Get**, the **Flush** message causes a pre-packaged HTTP Error 404 (Not Found) response to be generated.

A **Flush** message must always be sent to the http between **Get** messages, even if the first **Get** returned a request for something non-existent.

## Format

**Format**(String/Buffer **mimetype**)

**mimetype** Fixed string specifying the MIME (Multipurpose Internet Mail Extension) data type for the response string.

The MIME type data must be a recognised string in the general format *type/subtype*.

This message is only necessary to override the default MIME type generated by the HTTP object, based on the URI received. By searching the URI for certain strings the following types are inferred by default, and if the request is for a file that exists the indicated MIME-type will be sent automatically.

URI substring	Default MIME type	Use
.png	image/png	Graphics images, especially drawings and solid colours
.jpg .jpeg	image/jpeg	Graphics, especially photographic images
.gif	image/gif	As PNG, older standard
.htm, .html	text/html	HTML code

URI substring	Default MIME type	Use
.vhtml	text/html	HTML code with Venom variable names embedded. See <a href="#">VHTML</a>
.txt	text/plain	text not to be interpreted as HTML, usually displayed in a fixed size font
.css	text/css	Style sheets to define presentation style of HTML pages
.log	text/plain	Log files are assumed to be plain text
.xml	application/xml	XML data
.js	application/javascript	Code to be executed by Web browser
.csv	text/csv	Comma Separated Variables, which can be imported by many spreadsheet programs.
.svg	image/svg+xml	Scalable Vector Graphics - useful for software-generated images like graphs.
<i>default</i>	application/octet-stream	A generic stream of bytes - unknown application

A full list of registered MIME media types is available at <http://www.iana.org/assignments/media-types/>.



Minor bug: the parameter must be a fixed string. It should be allowed to be a text buffer also.

### Setting new default file type/MIME type associations

```
http.Format(str file_ending, str MIME_type, Int Max-Age)
```

<b>file_ending</b>	A file name ending string like ".pdf" (must include the dot)
<b>MIME_type</b>	The MIME_TYPE to be sent in the HTTP Content-Type header
<b>Max-Age</b>	The cache lifetime to send as a Cache-control: Max-Age=nnn Header for all files of this type

This can be used to modify or extend the list of default values shown above, applicable when files served directly from the file system have names and data type not in the default list.

Only one global table of filename ending/MIME type associations is kept. This saves space, and it's unlikely that you'd need to run two web servers simultaneously with different tables.

## Get

**Get**(String s)

The path is stored in the String object, s, supplied as a parameter.

The resource path, in simple terms, is the part of the URI that looks like a file name. For example, if the URI was

`http://www.mysite.com/search.html?keyword=VM2`

The path would be `search.html`

If no file system was specified when the HTTP object was created, then every incoming request except for "test" will cause **Get** to return. Otherwise during the **Get**, any incoming request whose path matches a file name in the file system and which has no GET or POST variables will be serviced automatically. This includes the processing of [VHTML](#) files. The **Get** will not return until something is requested which does not match a file name or which has variables attached.

Variable Values can be retrieved from the query string with the [Value](#) message.

## Match

**Match**(String name, String value) ⇒ Int

<b>name</b>	Name of a variable that may or may not have been set in a POST or GET request
<b>value</b>	Variable is tested for equality to this string

Returned result: true (1) if variable exists and matches the value given, False (0) otherwise

In some applications a string value is expected to have one of a limited set of known values. This message finds out if a variable exists with the name and value given, returning true if it does. It is less cumbersome than creating a local variable and testing its value with a string Compare.

Common uses include:

- For an "action" variable, with values like "list", "update", "reset", "save"
- For checkbox variables, which either don't exist or have the value "on"

## Example

```
If http.Match("action", "update")
[
    testmode := http.Match("testmode", "on")
    ...
]
Else If http.Match("action", "list")
[
    ...
]
; etc.
```

## Name

### 1. Retrieving Names for Uploaded Files

**Name**(Int **select**, String **var**) ⇒ Int

<b>select</b>	0 = get name of temporary file where uploaded file data is stored 1 = get name of original file at client end 2 = set name to store downloaded data as a file (see <a href="#">section 2</a> below)
<b>var</b>	String variable in which to store the name or (when select = 2) string to use as file name

Returned value: 1 if the indicated file and name existed, 0 if not.

When a file is uploaded using the POST method, this messages enable two file names to be retrieved.

Uploaded file data is stored locally in a temporary file whose name is chosen by the HTTPServer object and guaranteed to be unique. This name is returned when **select** = 0. If the [Mapping](#) message was used to specify a directory for the temporary file, the returned string



includes that path.

When a file is uploaded, the name of the original file at the client (browser) end of the link is returned when `select = 1`.

The Name message may only be used After `http.Get` has returned and before sending `http.Flush`.

The temporary file is removed (if it still exists) by the `http.Flush` message. If you wish to retain the file, it must be renamed.

### Temporary file Name

The temporary file is created in the root directory of the file system or in the directory specified by the [Mapping](#) message, and is in the form `xxxxxxxx.TMP` where `xxxxxxxx` is a randomly generated 8 digit hexadecimal number. Your Venom application should avoid creating files with similar names.

### Example

This example saves the uploaded file to its original name by renaming the temporary file.

```

TO server
  LOCAL tfname := New string(100)
  LOCAL fname := New string(100)
  LOCAL path := New string(100)
  LOCAL http := New HTTPServer(fs)

  FOREVER
  [
    http.get(path)
    if path.compare("upload.html") = 0
    [
      http.Name(0, tfname)
      http.Name(1, fname)
      fs.remove(fname)
      fs.Name(tfname, fname)
    ]
    Else if path.compare(...) ; other paths requested
      ... ; code for other requests

    http.Flush
  ]
END

```



See [Mapping](#) which is recommended for use when files are to be uploaded.



Also see [Uploading Files](#) in Notes on Use.

## 2. Setting File Name for Downloaded Data

```
http.Name(2, Str name)
```

When the first parameter has a value of 2, the second is a string (variable or fixed) that sets name of file for downloading dynamically generated data.

This will cause a browser to download the data and store it with the file name given.

Typically it would be used in conjunction with the [http.Format](#) message to specify the data type if the requested resource name implied a different file type from the data returned.

This form of the **Name** message can only be used after an **http.Get** has returned and before sending any data to the http object.

### Output

```
Output(Int mode)
```

When mode = 0, this sets printed output to the http object to be sent as http headers.

When mode = 1 (the default state), output is sent as part of the normal data returned in an http response e.g. html text.

### Restrictions on Use

This message enables headers to be generated which are not automatically produced by the HTTP object.

It must be used after any messages which would affect the automatically generated headers, such as

```
Period, Valid, Cookie, Name(2, ...)
```

Header data can be output using any of:

```
http.Put(int or string)
http.printf
Print To http
```

Every header sent this way must be terminated with "**\n**"

### Example

To make a page refresh itself every 5 seconds:

```
Forever
[
  http.get(path)
  http.Output(0)
  http.printf("Refresh: 5; url=%s\n", path)
```

```
    http.Output(1)
    ...           ; code to send the HTML page data
    http.Flush
]
```

Note that the above example could also specify a different URL instead of **path**, with the result that the page will display for 5 seconds, then another page will be visited.

## Period

### (1) Cache Control

**Period**  $\Leftarrow$  Int

This sets the period of time in seconds for which the returned data can be allowed to remain in a browser or proxy cache. The purpose of this is to speed up response when the same data is requested more than once and has not changed. Web browsers usually cache responses: a resource may be fetched the first time with this time period set in a `Cache-control` header, then during that period any further requests will use the cached copy instead of requesting it again from the VM2. An expiry value is assigned automatically by the http server object and the **Period** message is only needed if the default is to be overridden. The **Period** message should be sent after **Get** and before sending any response data, and only applies to the current request.

A period of 0 is valid and desirable if the response contains variable data that is likely to change every time it is requested, or if the request contains variable data intended to update the state of the VM2 in any way or trigger an event. In either case, a **Period** setting of 0 guarantees that the browser will send a request every time.

For versions of Venom after April 2016, the **Period** value defaults to 0 whenever **Get** returns, as any request being handled by Venom code is likely to involve dynamically-varying data in either the request or the response, and browser caching is undesirable in this case.

If the Venom code is handling a request for a file whose contents don't change, or for constant data supplied from an **array** or **textblock**, a non-zero period can be set to suppress repeated sending of identical data.

#### Technical Detail

The relevant http protocol headers are:

`Cache-control: max-age=nnn` (for non-zero period values)

or

`Cache_control: no-cache` (period = 0)

See also [Format](#) message, where default expiry values can be set for different data types when the file is served automatically.

## (2) Cookie Expiry

**Period(Str name)  $\Leftarrow$  Int**

This sets the expiry for a cookie whose name matches the one given.

The value assigned is the number of seconds in the future when the cookie should expire. If a source of calendar time is available, the cookie is sent to the client (browser) with an `Expires:` attribute set to the corresponding date and time, otherwise a `Max-age:` attribute is sent with the given time value in seconds.

As with the cache control **Period** message, The cookie **Period** message should be sent after **Get** and before sending any response data, and only applies to the current request.

If a fresh Cookie is being generated i.e. no cookie was sent by the client, The **Period** message must be sent after the cookie has been set by the server, otherwise you'll get a run time error trying to set the period for a cookie whose name is not yet recognised.

### Example

```
LOCAL cookieval := New string(30)
...
http.Get(req)
If http.Cookie("MYCOOKIE", cookieval) IsFalse
    http.Cookie("MYCOOKIE") := "cookiedata"
http.Period("MYCOOKIE") := 3600    ; expires in an hour

; code to send response to http
http.printf(...
```

## Put

**Put(data)**

**Put** puts a character, buffer, string, array or file into the response buffer.

A buffer must be text or 8 bit integers.

An array must be 8 bit integers.

All integer types are sent as 8 bit binary or character data.

Text buffers and strings are sent unmodified as 8 bit characters.

Files of any type are sent as a stream of bytes as stored in the file, starting at the file's current read point and finishing at EOF, except that if the file name ends with the string ".vhtml" (the test is not case-sensitive) it is subjected to [VHTML](#) processing. The file is left open with the readpoint at the end.

```
Put(String text[, Int type])
```

<b>text</b>	A String to be sent to the HTTP link
<b>type</b>	0 (default) = send as html; 1 = make VHTML substitutions

In this special case: the optional second parameter (default value 0) defines whether the text should be sent as normal HTML (0) or using the VHTML convention (1) where global variable names can be embedded between backslashes. See [VHTML](#).

## Redirect

```
Redirect(str url)
```

<b>url</b>	a string defining the new location
<b>flush</b>	If set non-zero, perform an http.Flush operation afterwards. Default 0 (no flush)

This message causes the server to send back a response code ("303 See Other") and headers that will cause a redirection to a different location.

In addition to the headers and response code, a brief HTML page is sent containing the redirection information as an `<a href=...>` link (as recommended in RFC2616 "HTTP/1.1") but in normal use with most browsers this would never be seen as the browser will redirect to the new location immediately.

No messages that affect http headers should be used before **Redirect**, and no data should be sent to the http object after it, but a **Flush** message must be sent after it, just as with any other response.

Typically this behaviour would be controlled by GET or POST variables sent when the current page is the target of an HTML `<form>` element.

It enables you to set which of several different pages to go to depending on a form variable's value.

## Example

**Forever**

```

[
    http.Get(path)
    if path.Compare("formaction.html") = 0
    [
        If http.Match("action", "redirect")
        [
            If http.Value("url", url) > 0
                http.Redirect(url)
        ]
        Else... ; other action values
        [
            ...
        ]
    ]
    Else... ; other path values
    [
        ... ; other code
    ]
    http.Flush
]

```

## Queue

**Queue**  $\Rightarrow$  Int

Returned value:

- 0 if no incoming request requiring service by the Venom program is present
- 1 if an incoming request requires processing by the Venom program

If a request can be processed internally this is done and **Queue** then returns a value of 0.

The returned value thus always indicates whether a subsequent **Get** will return immediately.

**Queue** can be used in a loop where it is required that **Get** will never block. e.g.

```

forever
[
    if http.Queue > 0
        process_request(http.Get)
    ; other tasks in loop...
    ...
]

```

## Valid

**Valid**  $\Leftrightarrow$  Int

Gets or sets the numerical response code to be used in the response headers.

This is usually set automatically, but if you want to generate a customised error message to accompany an error response you can do so by setting Valid to an appropriate value such as 404 (not found) and printing the HTML response text to the http object.

The values generated by the HTTP object automatically are:

200 - OK : normal code for data response

303 - See Other: indicates a redirect to a different URL (see Redirect message)

304 - Unmodified : sent when an "if none-match" header was received and the file requested has not changed

400 - bad request

403 - Forbidden e.g. attempting to upload a file when there's no file system, or 2nd parameter of [Mapping](#) was **nil**

404 - Not found e.g. file not found

413 - Entity size too large e.g. POST data more than 10000 bytes (other than a file)

414 - request line too long : max 1022 chars

501 - Not Implemented e.g. a request other than GET or POST was received

503 - Unavailable: generated when file system media, e.g. memory card, has been removed and a file was requested

See Internet RFC 2068 "Hypertext Transfer Protocol" for the full list of codes and their correct usage, but most values are unlikely to be useful.

## Value

```
Value(name, Int default [, Int post, [Int select]]) ⇒
  Int value
Value(name, Float default [, Int post, [,Int select]])
⇒ Float value
Value(name, String container [, Int post, [,Int
select]]) ⇒ Int length
Value(0, String container) ⇒ Int length
```

<b>name</b>	A string or text buffer containing the name of an identifier that may have been sent as part of the query string in an incoming request. An integer value of 0 here copies the requested path (as returned by <b>Get</b> ) to the string variable
<b>default</b>	An integer or floating point value
<b>container</b>	A String object large enough to hold the result
<b>post</b>	Zero: only search GET variables Non zero: only search POST variables unspecified: try GET, then POST if not found
<b>select</b>	An optional integer (default = 1) selecting which instance of a variable to retrieve when there is more than one with the same name

This retrieves values from POST or GET variables typically resulting from submitting an HTML <form> element.

POST data is contained in the body of an HTTP request.

GET variables are in a query string, which is the part that follows a '?' in the URI. e.g.

`http://mysite.com/somepage.html?num=123&temp=32.4&action=update`

The query string contains assignments of values to named variables, separated by '&', e.g. the above is taken to mean

num is set to the value 123

temp is set to the value 32.4

action is set to the value "update"

## Read an Integer or Float

The quantity is expected to be a string of digits which will be returned as an integer or floating point number depending on the type of **default**.

If the name does not appear in the GET or POST variables, the value of **default** is returned.

If the variable is set but is empty or contains no valid digits, a value of 0 or 0.0 is returned,



depending on the type of **default**.

## Read a String

You have to pass a **String** object (**container**) that the value is placed in.

If the name is that of a POST or GET variable, **container's** contents are replaced with the associated value string and the length of the value is returned.

Otherwise **container** remains unchanged and 0 is returned.

If the value is longer than the capacity of **container**, it is truncated.

## Multiple Values with the Same Name

It is possible to have multiple values with the same name in a single request. This happens normally if an HTML form contains a `<SELECT>` element with the `multiple` attribute set and more than one option was selected when the form was submitted. The optional **select** parameter chooses one of these values by positional count. The [Count](#) message will tell you how many instances of a variable exist with one name.

## GET and POST variables with same name

If the same name is defined as both GET and POST variables and this message is used with only two parameters, the GET value takes precedence. Set the **post** parameter to 0 or 1 if you need to be specific.

## Limitations

1. The identifier (variable name) must not contain an equals sign ('=' character), even in its URLEncoded form as "%3D", nor may it contain a NUL character, even encoded as "%00". The '=' can appear in the value of a variable without problems and will be encoded as "%3D" by web browsers processing form data and decoded normally by the HTTPserver object.
2. The value of a variable must not contain the NUL character, even encoded as "%00".

## Examples of usage

```
Make s string(100)
http.Value("stringval", s)
xval := 20
xval := http.Value("xval", xval)      ; unchanged if not specified
intvar := http.Value("intval", 0)      ; 0 if not specified
fltvar := http.Value("fltval", 1.0)    ; 1.0 if not specified
```

A suitable URI to produce results from the above would be something like:

```
formresponse.html?stringval=abcdef&intval=1234&fltval="123.456"&xval=10
```

## Example of Multiple values with select multiple

HTML form code snippet:

```
<form action="procform.html" METHOD=GET>
  <select name="animal" multiple>
    <option value="dog">Dog</option>
    <option value="cat">Cat</option>
    <option value="goat">Goat</option>
    <option value="horse">Horse</option>
  </select>
  <input type="submit">
</form>
```

Partial contents of Venom code for "procform.html"

Example URI: procform.html?animal=cat&animal=horse

```
s := New string(20)
http.put("animals selected<br>")
n := http.Count("animal")
If n = 0
  http.put("(none)<br>")
Else
[
  http.printf("%u animals selected<br>", n)
  Repeat n
  [
    http.Value("animal", s, 0, index)
    http.printf("%s ", s)
  ]
  http.put("<br>")
]
```

## PRINT

```
Print <HTTPServer>
Print <HTTPServer>:0
```

If the http object is listening, printing it produces the text: "http listening"

If a Get has returned a request and before a Flush message has been sent, printing the http

object produces the text "http GET " followed by the requested URL

```
Print <HTTPServer>:1
```

After a GET has returned, this lists all the received http headers. It's mostly useful for debugging.

```
Print <HTTPServer>:2
```

After a GET has returned, this lists any variables received either as GET or POST variables e.g. from a form. This is very useful when debugging the processing of html form data, as it shows you what variables were sent from the form.

## PRINT TO

```
Print to <HTTPServer>, list
```

Any data can be printed to an HTTP object using the usual conventions for text devices. The print output is appended to the content part of the response. This is the normal way to generate an HTML page in response to an http GET request.

When printing to an http object, **CR** sends a CR LF (13, 10) sequence

## Using Printf with Embedded Text and TextBlock

Like all objects that accept **PRINT TO**, the HTTPServer object can also be sent a **Printf** message, and it's worth noting that [embedded text](#) or a [TextBlock](#) may be useful ways to supply the majority of the HTML text to be returned by making it the format string for a **Printf** message. This keeps the HTML structure separate from the code that generates the variables, which may make the code more readable and helps with getting the HTML code itself correct.

### Example 1

In this example, a whole HTML page is in a **TextBlock**. A **TextBlock**, or several, could also be used as part of a page, of course.

Note that the values displayed are the result of function calls, not global variables, so we can't use VHTML here.

```
; this is just a big format string for a printf message  
TextBlock page1:  
<html>
```

```

<head>
  <meta http-equiv="refresh" content="2">
  <title>
    VM2 printf() with TextBlock demo
  </title>
</head>
<body>
  <h1>VM2 printf() with TextBlock demo</h1>
  <table>
    <tr>
      <th>Unit</th><th>Pressure</th><th>Temperature</th><th>Speed</th>
    </tr>
    <tr>
      <td>%u</td><td>%u</td><td>%u</td><td>%u</td>
    </tr>
    <tr>
      <td>%u</td><td>%u</td><td>%u</td><td>%u</td>
    </tr>
  </table>
</body>
</html>
TextBlock END

```

TO **servertask**

```

LOCAL http := new HTTPServer(nil)
LOCAL s := new string(100)
FOREVER
[
  http.Get(s)
  http.period := 0
  If s.Compare("demo.html") = 0
  [
    http.printf(page1, 1, read_pressure(0), read_temperature(0),
      2, read_pressure(1), read_temperature(1), read_speed(1))
  ]
  http.flush
]
END

```

#### Example 2 - the same using embedded text

In this example, a whole HTML page is in embedded text in the Venom code.

```

LOCAL http := new HTTPServer(nil)
LOCAL s := new string(100)

```

```

FOREVER
[
    http.Get(s)
    http.period := 0
    If s.Compare("demo.html") = 0
    [
        http.printf(<<<HTML:
<html>
    <head>
        <meta http-equiv="refresh" content="2">
        <title>
            VM2 printf() with TextBlock demo
        </title>
    </head>
    <body>
        <h1>VM2 printf() with Embedded text demo</h1>
        <table>
            <tr>
                <th>Unit</th><th>Pressure</th><th>Temperature</th><th>Sp
            <tr>
            <tr>
                <td>%u</td><td>%u</td><td>%u</td><td>%u</td>
            </tr>
            <tr>
                <td>%u</td><td>%u</td><td>%u</td><td>%u</td>
            </tr>
        </table>
    </body>
</html>>>>, 1, read_pressure(0), read_temperature(0), read_speed(
    2, read_pressure(1), read_temperature(1), read_speed(1))
    ]
    http.flush
]
END

```



## Mapping

Mapping(str getdir, str postdir)

<b>getdir</b>	Directory for retrieving files requested with with http GET or POST messages 0 or nil to suppress searching of the file system.
<b>postdir</b>	Temporary directory for uploaded files sent using POST

0 or nil to forbid file uploads
---------------------------------

By default, if an HTTPServer object is created associated with a file system, any requested resources are searched for in the root directory of that file system, and uploaded file data (using `html<input type="file">` in a form) is stored in a temporary file in the root directory.

Use of this message modifies the paths used for these purposes, and also enables either use of the file system to be suppressed if not desirable.

The message should be sent after creating the **HTTPServer** object, and before any **Get** message is sent.

The result is that an area of the file system can be reserved for use by the web server, and files outside of that area cannot be accessed by any external HTTP access. For the **postdir** parameter, it enables you to create and use a special directory exclusively for uploaded temporary files, which cannot overwrite other files in the system.

For both parameters, If the directory does not exist on the file system, it will be created, but the path to the directory must already exist, e.g. if you specify `"/server/html"` then `"/server"` must exist.

To specify the root directory of the file system (the default setting) you can use either an empty string or `"/"`.

If you specify either parameter as `nil` or `0`, then the file system will not be used. For **getdir**, this means any **http.Get(s)** message will return the requested resource name in its string parameter, as if no file system had been associated with the http object, and for **postdir**, it means any attempt to upload a file will return an HTTP 403 "Forbidden" response to the HTTP client (e.g. web browser) originating the request, along with a brief HTML page explaining the problem.

### Example

VM2 is set up with host name `"vm2.localnet"`

```
TO server_task
  Local req := New string(100),
  tempname := New String(100),
  http := New HttpServer(fs),
  f ; file for uploads

  http.Mapping("web", "web/tmp")
  Forever
  [
    http.get(req)
```

```
; code to process requests
If req.Compare("upload.html") = 0
[
    http.Name(0, tempname)
    ; code to process uploaded file
    f := fs.open(tempname, char)
    ...
    ...
]
End
```

A browser request directed to <http://vm2.localnet/index.html> will cause the server to look for

web/index.html in the file system.

A POST request to <http://vm2.localnet/upload.html> will cause the server to run the **http.Name** statement shown, and if a file was uploaded with the POST data, **tempname** will be set to something like "web/tmp/30AB6C92.TMP" which can be used directly by a filesystem **Open** message.

## I2C Bus

This object allows communication with devices on an I2C Bus. Usually you won't have to use this object, as individual driver objects will often handle the I2C Bus for you. For example, communication with PCF8574 digital I/O ports is handled entirely by the Digital object type. You will only need to use the I2C Bus object to communicate with devices where Venom doesn't have built-in drivers, or to find out which devices are connected to the bus.

The default implementation of the I2C Bus object doesn't handle the I2C Repeated Start condition. If you need to use Repeated Start then please see this [page](#).

## Summary of messages

**Make**  
**Die**  
**Find**  
**Get**  
**Locking**  
**Put**  
**Reset**  
**Send**  
**Print**

## Creation

```
Make <object> I2CBus  
Make <object> I2CBus (Int bus_no)
```

Create an I2CBus object. If the parameter is not present then bus number 1 is assumed.

For example:

```
Make obj I2CBus
```

When a parameter is supplied it can take the values 1 or 2.

Note: an I2CBus object for Bus number 1, called **net**, is created in the default startup procedure.

## Die

**Die**

Die will reset the internal I2C Bus peripheral so that it no longer controls or responds to the SCL and SDA channels, and so it draws no power.

The SDA and SCL channels are both set to floating inputs in the expectation that I2C Bus pull ups will pull them up, if necessary.


If you need the channels to change to a different state then you will have to do this explicitly.



## Find

**Find** (Int address)

This checks for the presence of a device on the network, returning True or False. It addresses the device and checks for an acknowledge pulse.

 Find does its own locking, so explicit locking is not necessary.

## Get

**Get**(Int address [, Int nbytes]) ⇒ Int

Get takes the address of the I2C device and a number of bytes to retrieve from it (1 to 4), and returns the bytes packed into an integer value. The optional parameter **nbytes** defaults to 1.


### Big Endian

Normally the first byte received from the bus is considered to be the *most significant byte* of the result - i.e. it is *Big Endian* (most I2C devices seem to be big endian).

### Little Endian

However, you can get a Little Endian result (the first byte received is considered to be the *least significant byte* of the result) by using a negative number for **nbytes**, for example:

```
net.Get(162, -2)
```

 Get does not lock the bus.

## Repeated start condition

If you need to use a Repeated Start Condition please refer to this [page](#).

## Locking

**I2CBus** objects have a lock built into them. For the sake of good programming practice, we recommend that all *explicit* transactions using an I2CBus object be locked, even if you only have one task accessing the I2C Bus, as you may add another task later.

## Example

To write

```
net . Lock
net . Put(160, 12, 2)
net . UnLock
End
```

### Detailed description

Objects that use the I2C Bus *implicitly*, e.g. I2C digital, will handle all the locking for you.

Explicit locking of the I2C Bus is only necessary if both the following are true:

- You are using an **I2CBus** object directly, by sending messages to it, such as **net.Put**, ect.

~ And ~

- You are using the same I2C Bus, either explicitly or implicitly through a client object, in another task.

If both the above are true then you should lock around every *explicit* use of the I2CBus object. You don't need to lock around implicit usage.

 See also Locking in the Tutorial

### Put

```
Put (Int address, Int data [, Int nbytes]) ⇒ Int
Success
```

Put sends 0-4 bytes of data over the I2C Bus. You can also use [Send](#).

The parameters are:

**address**: the I2C address of the device (this is an 8-bit address, i.e. twice the 7-bit address typically given in device datasheets).

**data**: the data to send. If you are sending more than one byte then **data** holds all the bytes as it is a 32-bit quantity.

**nbytes**: (optional) the number of bytes of **data** to send. Defaults to 1 byte. It is the least significant of the bytes that are sent. If a value of 0 is used then the device is addressed, but no bytes are sent. This is roughly what [Find](#) does.

### Return value

The value returned by Put indicates whether it was successful in sending the bytes to the device.

## Little Endian

If **nbytes** is negative it means send the bytes in *Little Endian* mode - i.e. the *least significant* of the bytes is sent first.

## Examples

```
net.Put(144, val) ; Send the byte val to the device at 144.  
net.Put(162, address << 16 + data , 3); send two-byte address and  
;(Note: address is Big endian and data must fit in 8 bits.)
```



Put does not lock the bus.



To send larger blocks of data, see the [Send](#) message.

## Reset

### Reset

Reset will force a re-initialisation of the I2C Bus peripheral module inside the MCU.

This will take account of the current [system clock speed](#) to produce the correct I2C Bus clock speed.



Reset will not respect the I2C Bus lock. You should use [Lock](#), etc, around this message if you want to respect the I2C Bus lock.

## Send

```
Send (Int address [, Int data1 ...]) ⇒ Int Success
```

```
Send (Int address, Array data [, Int nbytes[, Int  
Offset]]) ⇒ Int Success
```

**Send** transmits larger amounts of data over the I2C Bus. It is an alternative to [Put](#).

You can supply the data either as a list of individual parameters (up to 10), or as an array of 8-bit values.

(The first parameter to **Send** is always the I2C address of the device you wish to send data to.)

### Sending an array

When an array is specified, all the data bytes in the array will be sent unless you supply the optional parameter **nbytes**, which limits the number of bytes to send. You can also specify an offset into the array, to where your data starts, with the optional parameter **Offset**.

**Return value**

The value returned by Send indicates whether it was successful in sending all the bytes to the device.

**Examples**

```
net.Send(144, val) ;Send the byte val to the device at 144.
net.Send(162, val1, val2, val3);send three bytes
net.Send(144, my_array) ;Send all the bytes in the array.
net.Send(144, my_array, 2) ;Send two of the bytes in the array.
net.Send(144, my_array, 2, 3) ;Send two of the bytes in the array
```



Send does not lock the bus.

**Printing**

```
Print <I2CBus>
```

Printing an I2CBus object generates a list of all the devices connected to bus:

```
-->Make net2 I2CBus(2)
-->Print net2
Devices on the I2C network No.2:
```

Number	Channels	Device	Description
-----	-----	-----	-----
124	496-503	PCF8574A	8 digital I/O lines
126	504-511	PCF8574A	8 digital I/O lines
162	PCF8582/83...		RTC/EEPROM...

Note that this feature can only guess the devices attached to the bus, as there are some devices that map onto the addresses of the more common devices.

**Software based driver**

Because the hardware-based driver doesn't easily handle the Repeated Start Condition we've also implemented a software-based driver for the I2C Bus.

To use this driver you must do the following:

If your code has already defined any I2CBus objects, remove them:

```
net.Die ; remove any existing I2CBus objects.
```

Then signal that you want to use the software-based drivers from now on:

```
Debug(25) := True ; flag to force creation of software-based bus.
```

Then remake the bus:

```
Make net I2CBus(1) ; Remake the I2CBus object.
```

To use the Repeated Start you first have to signal that an I2C packet should end without sending a Stop Condition.

You can do this by sending the I2C Put message with an extra parameter that has the value **True** or **1**, for example:

```
net.Put(126,$55,1,1)
```

Because the previous packet ended without a Stop Condition, the next packet's Start Condition will be a Repeated Start:

```
net.Get(127,1)
```

## IProt

IP stands for Internet Protocol. The protocol is concerned with addressing and routing packets on the Internet, and with certain test and housekeeping functions.

An IP object does not have to be created in order for IP to be used for routing and addressing incoming and outgoing IP datagrams, nor for TCP or UDP connections. The IP module autonomously handles routing, accepting IP address registration from the PPP module and Ethernet modules, creating a loopback interface on 127.0.0.1 and routing packets between TCP clients, the PPP links and (if/when available) the Ethernet interface. The module also replies to any incoming ICMP echo (ping) requests.

A Venom IP object can be created to enable certain services:

- Testing the network using the Time message, which acts like the well known "ping" utility.
- Converting Dotted Quad IP addresses and domain names into numerical IP addresses.
- A general TCP/IP interface polling function.
- Assigning names to local IP addresses.

See also [TCP/IP Networking](#)

## Summary of messages

**Make**  
**Address**  
**Debug**  
**Find**  
**Go**  
**Print**  
**Time**

## Creation

**Make** <object> **Iprot**

Creates an IP object. The object can be used as a tool for finding information about the network, but is not essential for the operation of IP based protocols such as TCP and UDP.

## Address

**Address** (str name)  $\Leftarrow$  **ipaddress**

**name** can be a string or a text buffer.

This stores the association of a symbolic name with a numerical IP address (integer or dotted quad string) in a local table. This may be useful for local networks so hostnames can be used without a DNS server. Any number of addresses can be assigned this way.

## Example

Note the code examples in this section also illustrate the "**IP**" Print modifier which displays an integer in the dotted-quad notation conventionally used for IP addresses.

```
-->ip.Address("printer") := "192.168.1.250"
-->Print ip.Address("printer"): "IP", CR
192.168.1.250
-->
```

**Address** (host)  $\Rightarrow$  **Int**

Converts a hostname or IP address in dotted quad notation to a numerical IP address. If the host string contains only numerals and dots it is assumed to be a dotted quad IP address, otherwise is it treated as a host name and will either return an address that was previously assigned by the first form of the **Address** message or if not in the list will be found by DNS lookup like **ip.Find**.

## Examples

```
--> Print ip.Address("www.venomcontrolsystems.co.uk"): "IP", CR
109.75.171.120
-->
-->a := ip.address("80.10.20.30")
-->print a, CR
1342837790
-->print a: "IP", CR
80.10.20.30
-->
```

**Address (str addr, 0) ⇒ Int**

As above, except the string must only be in numeric dotted-quad notation. Any invalid string will generate run time error 26, which can be trapped with [Try/Catch](#) enabling this to be used for validation of IP address strings.

## Debug

**Debug (0, file f)**

Enable logging of information about packets sent and received. **f** must be an open writable text file.

**Debug (0, 0)**

Stops logging of packet info.

**Debug (1)**

Prints the routing table (same as **Print ip**)

**Debug (2)**

Prints the list of host names and IP addresses created by **ip.Address**.

## Find

**Find (String str) ⇒ Int**

The parameter can be a fixed string or text buffer.

Performs a DNS query to convert a domain name or hostname into an IP address. The names "localhost" and "loopback" are recognised and converted automatically to 127.0.0.1. All other

names require an internet connection so the names can be looked up with a nameserver.

**Find**(Int **n**)  $\Leftrightarrow$  Int

Returns IP address of name server in use, or assigns the address of a nameserver to use. n can be 0 or 1 denoting the two nameserver addresses that are tried in succession.

Both PPP and Ethernet have mechanisms for finding the address of a usable nameserver, so a nameserver does not usually have to be assigned this way.

## Go

**Go**(Int **milliseconds**)

Polls all IP interfaces at regular intervals for the length of time specified.

If no parameter, or a value of 0 is supplied, polls forever and does not return. This is sometimes useful if you want interfaces to respond to activity like "ping" requests, but do not have have a TCP or UDP interface active or listening all the time.

## PRINT

**Print** <Iprot>

Prints the routing table. The default routing table only contains the localhost address 127.0.0.1. When a PPP connection is made, its address is added to the routing table and becomes the default route. The Ethernet object can also add network and host addresses and a default route to the routing table. (see [Ethernet.Address](#))

## Time, Period

**Time** (Any **host**[, Int **Timeout**])  $\Rightarrow$  Int

**Period** (Any **host**[, Int **Timeout**])  $\Rightarrow$  Int

Returns time to respond to ICMP echo request ("ping") in milliseconds. If there is no response after 10 seconds a value of -1 is returned.

**host** is either a string containing a domain name or dotted quad IP address, or an IP address as an integer

**Timeout** if specified, overrides the 10 second default timeout in milliseconds

**Period** is a synonym for **Time**.



## Keypad

The Keypad object is used to interface to a variety of keypads. Most keypad drivers consist of one or more PCF8574 ICs on the I2C Bus controlling the rows and columns of a key matrix. However, one type of keypad scans 'virtual' keys on a [TouchScreen](#) object.

By switching on the Keypad's [InputBuffer](#) function you can automate many of the functions most applications need a keypad to perform.

### Summary of messages

**Make**  
**Asserted**  
**Flush**  
**Get**  
**GetLast**  
**InputBuffer**  
**Key**  
**Period**  
**Queue**  
**Time**  
**Update**

### Creation

```
Make <object> Keypad (Int type, Int chan1 [, Int chan2]  
)
```

```
Make <object> Keypad (TouchScreen ts)
```

There are two forms of the Make command for Keypad, one for matrix keypads, and one for use with a TouchScreen 'virtual keypad'. In operation the two types of keypad are very similar.

### Matrix Keypads

The parameters are as follows:

**type**: keypad type; see the table below.

**chan1**: Channel number indicating a PCF8574 IC

**chan2**: Channel number indicating a second PCF8574 IC (only for 8 by 8 and 12 by 4 matrixes)

Matrix	Type Parameter	No. of PCF8574 needed	How rows and columns map onto PCF8574(s)
4 by 4	0	1 needed	First four channels: columns Last four channels: rows
8 by 8	1	2 needed	First PCF8574: columns Second PCF8574: rows
12 by 4	2	2 needed	Channels 4-7 of the 1st PCF8574: rows 0-3 Channels 0-3 of the 1st PCF8574: columns 8-11 2nd PCF8574: columns 0-7

Key numbering starts at 0 (zero) for the key at the first row/column position. The first row (or column) is the one with the lowest channel number. Key 1 is on the same row, next column along, and so on.

Any subset of a matrix may be supported, for example a 12 by 3 matrix may be implemented by the 12 by 4.


Examples:

```
Make kpd1 Keypad (0,168) ;4x4 keypad, PCF8574
Make kpd2 Keypad (0,248) ;4x4 keypad, chip 1, PCF8574A
Make kpd3 Keypad (0,240) ;4x4 keypad, chip 2, PCF8574A
Make kp Keypad (1,240,248) ;8x8 keypad, chips 1 & 2, PCF8574A
Make kpd Keypad (2,240,248) ;12x4 keypad, chips 1 & 2, PCF8574A
```

## Touchscreen Keypad

In this case there is only one parameter to the Make: the TouchScreen object. See [TouchScreen](#) for details of how to define the Touchscreen object and its virtual keys.

```
Make kpd Keypad (ts)
```

 Each Keypad uses around 28 bytes of memory.

## Asserted

**Asserted**  $\Rightarrow$  Int

Asserted returns True if any of the keys on the keypad are currently being pressed. The GetLast message will tell you which key it was.

 See also [Key](#), [GetLast](#), [Get](#)

## Flush

### Flush

Clears out the keypad's input buffer.

## Get

**Get**  $\Rightarrow$  Int

Get waits for a key press and then returns the result. If no key is pressed then Get will wait indefinitely.

The operation is slightly different depending on whether you are using the InputBuffer functionality.

### Normal operation

Get waits for no keys to be pressed on the keypad, then waits for a key to be pressed, and returns the logical key number of the key pushed.

### InputBuffer operation

Get waits until a key press appears in the input buffer. For this to happen you must have a separate task repeatedly sending the Update message to the keypad. For this reason it's not usual to use Get when you are using an InputBuffer.

 [InputBuffer](#)

*Note: When two keys are pressed at the same time a special 'key' value is returned, guaranteed to be larger than any legal key value.*

## GetLast

**GetLast**  $\Rightarrow$  Int

Returns the key number of the key that caused the last Asserted message to return True.

*Note: When two keys are pressed at the same time a special 'key' value is returned, guaranteed to be larger than any legal key value.*

 See also [Asserted](#).

## InputBuffer

```
InputBuffer(Int rpt_pd [, Int rpt_del])
```

### **InputBuffer**

There is an 'Input Buffer' function inside a Keypad object that provides higher functions that the basic Keypad doesn't have: *Auto Repeat* and *Key Press Buffering*. These can help to simplify your keypad input routines.

*Auto repeat* is when multiple key presses are requested by holding a key down.

*Key press buffering* is used to detect and store key presses for later use.

For the InputBuffer functions to work correctly, the Keypad must be given regular calls to its [Update](#) message, say from inside an Every loop.

Currently the key buffer holds only one key press - this is usually the most that is needed.

To turn on the input buffer function the **InputBuffer** message is sent to the Keypad object.

This message takes two parameters: the auto-repeat period, and auto-repeat delay.

**rpt\_pd** is the time between each auto-repeat of the key when a key is held down. It is measured in multiples of the Keypad's Update period. If this parameter is not present, then it defaults to zero. If the repeat period is set to zero auto-repeat is turned off.

**rpt\_del** is the time taken before auto-repeat starts, also in multiples of the Update period. If this optional parameter is not present it defaults to the repeat period.

### **Turn off auto-repeat**

If you want to just turn off auto-repeat, but keep key press buffering you should call use this:

```
kpd . InputBuffer(0) ; Turn of Auto-Repeat
```

### **Turn off all input buffer functions**

Calling InputBuffer with no parameters turns off input buffering completely (both auto-repeat and key press buffering)

```
kpd . InputBuffer
```

### **Example**

The example below uses an InputBuffer to provide auto-repeat on some keys that alter the value of the variable **target\_temp**.

Note that the keypad scanning (**kpd.Update**) is done in the same task as the key actions. Sometimes you may need to put it in a separate task, but this is rare.

```

Make kpd Keypad(0,496) ;Make the Keypad object
kpd.InputBuffer(2,15) ;Turn on key buffering
...

#define DN_KEY 0
#define UP_KEY 1
#define Exit_KEY 3

Every 50 ;Keypad scan period
[
  kpd . Update ;Scan the Keypad
  the_key := kpd.Key
  If the_key >= 0 ;Any key pressed?
  [
    Select Case the_key ;Select an action for each key
    Case DN_KEY
    [
      If target_temp > MIN_TEMP
        target_temp := target_temp - 1
    ]
    Case UP_KEY
    [
      If target_temp < MAX_TEMP
        target_temp := target_temp + 1
    ]
    Case Exit_KEY
    [
      Break ;Drop out of this menu
    ]
    Case Else
    [
      buzz_wrong_key
    ]
  ]
]

```

## Key

**Key**  $\Rightarrow$  Int

Key has two different modes of operation, depending on whether [InputBuffer](#) has been set up.

### With InputBuffer

Key will return the number of any key press that *has been detected and stored in the buffer*, or -1 if no key press was detected. The Update message has to be called regularly for this to work.

 [Code Example](#)

*Note: When two keys are pressed at the same time a special 'key' value is returned, guaranteed to be larger than any legal key value.*

### Without InputBuffer

Key returns the number of any key that is *currently being pressed*, or -1 if no keys are being pressed.

Example:

```
Every 50
[
  Select Case kpd.Key
  Case 0
  [
    ; Do key 0 action
  ]
  Case 1
  [
    ; Do key 1 action
  ]
  Case Else
  [
    ; 'No key' action
  ]
]
```

*Note: When two keys are pressed at the same time a special 'key' value is returned, guaranteed to be larger than any legal key value.*

 See also [Keypad.InputBuffer](#).

## Period

**Period**  $\Leftrightarrow$  Int

Period allows the value of the auto-repeat period to be set and read. Note that the new value of Period won't take effect until the next auto-repetition.


This may be used to implement an accelerating auto-repeat function, for example:

```
Make kpd Keypad (0,496)
kpd.inputbuffer(10,20)

Start Every 30 kpd.Update

Every 10
[
    If kpd.Time >= 50
        kpd.Period := 2
    Else
        kpd.Period := 10

    Select Case kpd.Key
    Case 0
        Print "key0",kpd.Time, CR
    Case 1
        Print "key1",kpd.Time, CR
]
```

 See also [Time](#)

## Queue

**Queue**  $\Rightarrow$  Int


Returns the number of keys in the buffer: only zero or one currently.

## Time

**Time**  $\Rightarrow$  Int

Time returns the number of Update cycles the current Key has been held down for. This doesn't get reset when the key is released (only when the next is pressed), so it is also the time the last key pressed was held down for.

This may be used to implement an accelerating auto-repeat, as well as other useful functions.

 See also [Period](#), which includes a code example.

## Update

### Update

This message is only relevant if an [InputBuffer](#) has been set up. Update checks the keypad for key-down and key-up events and uses them to update the InputBuffer.

Update should be called regularly, say from within an Every loop. The loop may contain other code, so long as it doesn't upset the loop timing while you are scanning for keys. It's usually OK to break the normal loop timing to act on the key presses that are detected, since most applications don't care about the next key press until the previous action has been completed.

A typical loop period is 50mS though you should experiment with your set up to determine the optimum period.

See here for [example code](#).

## NIL

Nil is a special object that is used as a placeholder for any other type of object, or to represent 'no object'.

Nil will accept any message, and will return False to all of them.

It is useful for situations where, in the normal course of events, an object is used but sometimes you wish to allow for situations where 'no object' needs to be represented.

The most common example of this is substituting the value Nil for an output stream object, for when you want Print output to be discarded:

```
lcd := Nil ;all Print To lcd will now be discarded.
```

Testing for Nil is done with = or <>, as follows:

```
If lcd = Nil  
[  
  ...  
]
```

## Summary of messages

*All messages are accepted*



## Creation

```
obj := Nil
```

The value Nil is defined in Venom so you don't need to use Make or New to get a Nil object.

 Nil is a [Zero-Memory](#) object

## NumberReader

NumberReader is for entering numbers, typically using a [Keypad](#), as on a calculator or telephone. For visual feedback the digits of the number may be printed to a display device (e.g. an LCD) as the number is being entered.

It is easy to create number-entry routines to enter integer, floating-point, or hexadecimal numbers, or [secret](#) PIN numbers. The displayed text may be initialised to any text string to prompt the user – either with a default value, or with any other text.

## Summary of messages

```
Make
Close
Empty
Length
Mapping
Output
Put
Reset
Value
Width
Print To
Print
```

## Creation

```
Make <object> NumberReader
Make <object> NumberReader ([Int base [, Int max_width
]])
```

The optional parameter **base** determines what number base to use for converting numbers. The default base of 10 is assumed, for decimal numbers. For hexadecimal numbers use 16.

The second (optional) parameter **max\_width** allows a maximum number of digits in the displayed number (including any decimal point and minus sign characters) to be specified. The default width is 10 digits. Once the NumberReader is 'full', further key presses will be ignored.

On creation, the NumberReader's buffer is set to "0", and the buffer is '[Closed](#)' - i.e. will be over-written at the next [Put](#). This gives calculator-like behaviour. If you need the buffer to be empty, or to show some other text, use [Empty](#), or [Print](#) into the NumberReader.

## Examples

The first of the following examples creates a decimal NumberReader with the default maximum field width; the second example creates a hexadecimal NumberReader that will only accept numbers with up to 4 digits.

```
Make n1 NumberReader
Make n2 NumberReader (16,4)
```

The [Put](#) message is what does most of the work of a NumberReader.

 See also [Conversion](#), [Width](#), [Accepting Print](#); the [Keypad](#) object

## Close

### Close

Close indicates to the NumberReader that the current buffer contents should be over-written at the next [Put](#).

## Empty

### Empty

Empties the NumberReader's buffer. This is different to [Reset](#).

 See also [Print](#), [Accepting Print](#)

## Length

**Length** ⇒ Int

Length returns the number of characters in the [Print](#) output buffer.

## Mapping

Since different keypads have different mappings between key *function* (as indicated by the identifier on the key cap) and the *logical key number* we need a way to specify to the NumberReader which keys are to be used for which purpose. The Mapping message does this.

(The *logical key number* is the number returned by [Keypad.Key](#), etc, when you press a key. On a 4 x 4 matrix keypad the logical key numbers typically range from 0 - 15. )

Note that if there are keys mapped to the Decimal Point and/or Minus functions then places are reserved for these characters in the number buffer [width](#).

You can either map all the key functions in one call to Mapping, or set them individually.

### Mapping all the keys at once

```
Mapping
(
    Int Decimal_key,
    Int Minus_key,
    Int Delete_key,
    Int Dig0, Int Dig1, Int Dig2, Int Dig3, Int Dig4,
    Int Dig5, Int Dig6, Int Dig7, Int Dig8, Int Dig9,
    Int DigA, Int DigB, Int DigC,
    Int DigD, Int DigE, Int DigF
)
```

The 13 or 19 parameters to Mapping are explained here:

- The first three parameters list the logical key numbers of the special function keys **decimal point**, **minus** and **delete**.
- The next ten parameters list the logical key numbers of the decimal digit keys, that represent **digits 0-9**.
- The last six parameters list the logical key numbers of any extra hexadecimal digit keys, that represent **digits A-F**.

The three special functions are listed, in parameter-list order, in the table below, together with details of their actions:

Function	Action
Decimal	Enter a decimal point; only one will be allowed in any number
Minus	Adds or removes a minus sign to the left of the number
Delete	Removes the last digit entered; may be done repeatedly

If any particular special function is not required then the entry in the Mapping parameter list should be '-1' to indicate 'no key'.

Mapping may be re-sent at any time to change the functions of the keys.

Here is an example of the use of Mapping. Note how the Delete function is disabled by being assigned to key '-1':

```
#define DECIMAL_KEY 3
#define MINUS_KEY 12
#define DELETE_KEY -1

nr . Mapping ;Assign functions to keys on the keypad.
(
    DECIMAL_KEY,
    MINUS_KEY,
    DELETE_KEY,
    ;Digits 0-9 on these keys:
    13,0,1,2,4,5,6,8,9,10
)
```

### Mapping individual keys

```
obj.Mapping(Int index) := Int key_value
```

You can also access individual keys in the key map:

```
nr.Mapping(0) := 5 ; Set the logical key for decimal point.
```

### Output

```
Output (Int dummy_char)
```

Output sets a dummy character to be printed instead of the actual digits entered, so that numbers may be entered secretly.

Setting Output to zero turns off this 'secret mode'.

```
nr.Output := '*' ; Print stars instead of real digits.
```

## Put

**Put** (Int **Key**)  $\Rightarrow$  Int

Put is used to send logical key numbers to the NumberReader, which it interprets as digits or special functions using the mapping given by the [Mapping](#) message.

Put takes one parameter: the logical key number to be sent to NumberReader. Key numbers are often read from a [Keypad](#) object, but can come from any other source.

Put returns True if the Key was acted on. If the

To provide visual feedback of the number being entered, the partially assembled number may be displayed by [printing](#) the NumberReader object to an output device.

### Example Code

```
; Keypad Map:
; Key Tile Legend
; 1      2      3      F
; 4      5      6      E
; 7      8      9      D
; A      0      B      C
; Associated Function
; 1      2      3      'Float' (dec point).
; 4      5      6      Enter
; 7      8      9      Delete
; '-'    0 [spare] Cancel
; Logical Key number
; 0      1      2      3
; 4      5      6      7
; 8      9      10     11
; 12     13     14     15

;Put names to some logical key numbers.
#define ENTER_KEY    7
#define CANCEL_KEY   15

#define DELETE_KEY   11
#define DECIMAL_KEY  3
#define MINUS_KEY    12

To init
  Make lcd AlphaLCD (20 2 0) ;The display device
```

```

Make kpd Keypad (0,248)      ;The keypad device
Make nr NumberReader (10,8)
nr . Mapping      ;Assign functions to keys on the keypad.
(
    DECIMAL_KEY,
    MINUS_KEY,
    DELETE_KEY,
    ;0-9 on these keys:
    13,0,1,2,4,5,6,8,9,10
)
End

To test_num_reader
    Local key_num
    Local def := 123.45      ;Our default value
    Print To nr, def:5:2      ;Set default value text
    ; nr.Reset ;[use this if line above not used]
    Do
    [
        Print To lcd, Home, nr      ;Visual feedback on LCD
        key_num := kpd.Get          ;Get a key number
        nr . Put(key_num)           ;Give it to the NumberReader
    ]
    While key_num <> ENTER_KEY
        Or key_num = CANCEL_KEY ;Exit when 'Enter' or 'Cancel' key s
    Print nr.Value , CR          ;Report the result
End

```

The code reads digits at the keypad and displays them on an LCD. When either the ENTER or CANCEL key is pressed the code reports the value of the number entered.

The number is given a default value of 123.45.

## Reset

### Reset

Resets NumberReader's buffer to "0".

It also 'Closes' the buffer, so that the next key press will overwrite the "0", just like on a calculator display.

If you want to completely empty the buffer then use [Empty](#).

 See also [Print](#), [Accepting Print](#)

## Value

**Value**  $\Rightarrow$  Int or Float

Returns the value of the number currently held in the Number Reader buffer.

Value will return an integer if there was no decimal point in the assembled number, otherwise it will return a floating-point value.

- If you only want integers then don't assign a decimal point key in [Mapping](#).
- If you only want floating-point values then use obj.Value As Float to read the value.
- If you want to detect the type of the returned value use the TypeOf function:

```
If TypeOf nr.Value = TypeOf 1.0 ;is it a float?  
[  
...  
]
```

By detecting the type of the value it is possible to prompt the user to enter a decimal point if you require them to do so.

*For visual feedback it is better to print the number reader object itself, rather than Value.*

 See [Printing](#)

## Width

**Width**  $\Leftrightarrow$  Int

Sets (or reads) the maximum number of characters allowed in a number, including any minus sign or decimal point. Any key presses that would increase the number of characters beyond this width are ignored. If the Minus or Decimal functions are [mapped](#), then a space is always reserved for each of these characters: numeric digits won't be allowed to fill the whole width.

▼ The maximum value for width is 34.

 See also: [Creation](#), [Put](#), [Accepting Print](#)

## Accepting Print

Printing to a NumberReader will put text into the NumberReader's buffer. This text need not always be a valid number.

However if you put an invalid number into the buffer then it is advisable to use [Close](#) so that it is overwritten by subsequent keypresses.

Examples:

```
def_val := 123.45
nr . Width := 8
Print To nr1, def_val:4:2...


-->Print nr
    123.45
```

Or...

```
Print To nr1, "Power"
...
-->Print nr
    Power
```

Text printed to a NumberReader is processed as follows:

- Space characters at the start of the text are stripped off.
- If the text exceeds the current fieldwidth of the NumberReader then the rightmost characters are lost.


 [PrintF](#) may also be used to send text to a NumberReader.

## Printing

```
Print <object> :fw
```

Printing a NumberReader prints the characters of current number it has assembled. If a fieldwidth (**fw**) is supplied then it will print right justified in the given fieldwidth, with any padding to the left being made up of zero or more space characters.

NumberReader holds the digits it has assembled in a buffer so that the number may be printed as it is being assembled, and afterwards. This buffer is not overwritten until [Reset](#) or [Empty](#) is sent, or the NumberReader is '[printed-to](#)'.

 See also: [Value](#)

## OnBoardLED

OnBoardLED controls the behaviour of the LED mounted on the VM2 board, and also channel 15, which the LED is attached to.

The LED may be told to turn on or off, flash various patterns and at various rates, so it may be used to signal the current state of the VM2 to users.

## Summary of messages



**Make**  
**Asserted**  
**Flash**  
**Off**  
**On**  
**Toggle**  
**Print**

## Creation

**Make** <object> **OnBoardLED**

The OnBoardLED object is created with the LED turned off. (Note that in the default startup routine, the OnBoardLED object is created, and then set to [flash pattern \\$80](#) if the VM2 is in run mode, and then turned off if the application ever terminates back to the command line).



OnBoardLED is a [Zero-Memory](#) object

## Asserted

**Asserted**  $\Leftrightarrow$  Int

An [active variable](#) that allows the state of the OnBoardLED object to be read or set. It returns True if the LED is on, and False otherwise. When setting the OnBoardLED via Asserted - True turns the LED on, and False turns it off.

## Flash

**Flash** (Int **pattern**)

Makes the LED flash in a pattern.

The flash pattern is given by the binary bit pattern of the parameter.

(There are two exceptions: 0 means turn the LED off, 1 means turn it on constantly).

Everywhere there is a 1 in the binary representation of the pattern, the LED is turned on for ~1/8th of a second (actually 128mS), else it is turned off for the same time.

The pattern is shifted to the right - and ends when there are no more 1 bits left.

Example patterns are given below - but you can create your own.

Pattern in	Pattern in binary	Description
------------	-------------------	-------------

hexadecimal		
\$80	%0000000	Short flash approximately once a second
\$A0	%0100000	Two short flashes approximately once a second
\$A8	%0101000	Three short flashes approximately once a second
\$A9DDCA80	%01010011101110111001010	Signals "SOS" in Morse Code
\$3AE77700	%0011101011100111011101110	Signals "OK" in Morse Code

## Off

### Off

Turns the LED off.

## On

### On

Turns the LED on.

## Toggle

### Toggle

Inverts the state of the LED.

## Printing

### Print <OnBoardLed>

Prints "ON " or "OFF" (always three characters) depending on the state of the LED.

## OneWire

The OneWire object is a Dallas 1-Wire™ Bus driver. This bus uses a single wire to carry power and data to a distributed network of devices. The communication protocol is detailed in Dallas documents and device datasheets.

### Summary of messages

**Make**  
**Checksum**  
**Find**  
**Get**  
**High**  
**Low**  
**Put**  
**Reset**  
**Print**

One of the most important features of the 1-Wire bus is that every device manufactured for the bus has a completely unique 64-bit serial number burned into it's 'ROM'. This means that any device connected to your application can always be addressed uniquely.

The first 8 bits of the 64-bit serial number are the family code that indicates what capabilities the device has; i.e. what commands it will obey, and maybe what memory capacity it might have.

The last 8 bits of the serial number are a CRC (Cyclic Redundancy Check) of the previous 56 bits. This provides a high level of data-validation when using the serial number to address devices.

### 1-Wire Protocol

Briefly, the 1-Wire bus master (VM2) reads and writes data on the bus by pulling the bus low for long or short pulses to indicate 1's and 0's. Timing is not very critical.

1-Wire bus communications always follow a very simple sequence:

#### **Reset**

This initiates a communication by putting all the devices on the bus into the initial state. Reset will elicit a response from every device on the bus called the 'presence pulse'. This may be used to determine whether there are any devices connected.

#### **ROM command**

This addresses a unique device on the bus – either by assuming it is the only device present, or by addressing a device by its unique serial number.

SEARCH ROM and MATCH ROM should be used when more than one device could be connected to the bus.

Command	Code	Decription
SKIP ROM	\$CC	Address all the devices on the bus at once – or just one device if only one is present.
MATCH ROM	\$55	Address just one device by sending its ROM data to the bus after this command.
READ ROM	\$33	The next 8 bytes read are the ROM data of the one device on the bus.
SEARCH ROM	\$F0	[Use the <a href="#">Find</a> message to do this for you, as SEARCH ROM can't be used directly]

### Transport layer command

This is generally a data transfer operation where one or more bytes of data are transferred either to or from the bus.

Command	Code
READ MEMORY	\$F0
WRITE SCRATCHPAD	\$0F
READ SCRATCHPAD	\$AA
COPY SCRATCHPAD	\$55

The details of how to use these commands are beyond the scope of this help file. Please read the Dallas 1-Wire bus documentation or the device datasheet for further information.

The bus communication can end at any point in this 3-stage sequence. For example you may just want to issue a Reset command to determine whether any devices are present.

### New device on the bus

When a device is first connected to the 1-Wire bus it will issue a presence pulse without being prompted by a reset from the bus master.

If you want to detect these new device presence pulses then it is possible to use a [PulseWidthIn](#) object to detect them. This simple application prints out the devices it finds on the bus every time a new one is added.

```
To init
  Make detect PulseWidthIn ($18, 1)
  Make b OneWire($18)
End
```

```

To main
  Forever
  [
    detect.go
    Await detect.Done
    Wait 1 ; Debounce
    Print b
  ]
End

```

## High-current Pull-up

Some 1-Wire devices need a high-current pull-up during certain parts of the 1-Wire communication in order to function correctly. See [Creation](#) and the [High](#) and [Low](#) messages.

## Limitations

This implementation of the 1-Wire bus does not cover 1-Wire interrupts of either type, nor does it cover Overdrive speed.

It has not been determined whether it covers the programming of 12V EPROM devices.

## Creation

```
Make <object> OneWire(Int data [, Int pullup])
```

**data** is the channel to use for the data line.

**pullup** is the optional channel to control the high-current pull up transistor required by some 1-Wire devices such as temperature sensors. See [High](#) and [Low](#).

Which ever channels are used are configured by the Make as Open Drain outputs and will need pull up resistors fitting in order to make them work correctly.

Currently the only valid values for data and pullup are \$18 and \$78:

```

Make owb OneWire ($18)
Make owb OneWire ($18 , $78) ; use active pull-up.

```

## Checksum

```

Checksum ⇔ Int
Checksum(Int select) ⇔ Int

```

Checksum is an [active variable](#) that sets and reads either of two CRC accumulators kept by the OneWire object. The 1-Wire bus uses two types of CRC extensively in its operation: an 8-bit CRC and a 16-bit CRC. The optional parameter **select** chooses which CRC is accessed.

Every Put to the bus or Get from the bus using the OneWire object updates **both** of these CRCs with the value of the data byte read or written. This can then be used to implement the protocols for data validation described in the Dallas literature.

In particular, when reading the 64-bit ROM data from a device, the last 8-bits are an 8-bit CRC of the previous 56 bits. The definition of the Dallas 8-bit CRC is such that the total CRC of all 64-bits will be zero.

The [Find](#) message and [Print](#) use this CRC internally to determine the validity of the data they are reading.

The 16-bit CRC (or CRC-16) is generally used for validating the larger amounts of data transferred during data storage/retrieval operations on the 1-Wire bus.

The CRC-16 is normally used in an inverted form, where the two bytes of CRC sent after the data are the value of this expression:

```
Inv owb . Checksum(1)
```

(The least significant byte should be sent first.)

When reading a block of data including the inverted 16-bit CRC, the final CRC value read will be \$B001.

The CRC-16 is also normally initialised to the value of the page number of the memory being addressed to provide security of addressing as well as data.

See the Dallas literature for a deeper explanation of these issues.

## Examples

```
owb . Checksum      ; returns the 8-bit CRC value
owb . Checksum(0)  ; returns the 8-bit CRC value
owb . Checksum(1)  ; returns the 16-bit CRC value

owb . Checksum := 0    ; sets the 8-bit CRC value
owb . Checksum(0) := 0 ; sets the 8-bit CRC value
owb . Checksum(1) := val ;sets the 16-bit CRC value
```

## Find

Find uses the 1-Wire bus SEARCH ROM command to get each unique 64-bit ROM serial number for each of the devices connected to the 1-Wire bus. The 1-Wire bus's MATCH ROM command can then be used to address any one device individually.

```
Find ( Int max_devices , Int data_addr) ⇒ Int
```

**data\_addr** is an address in RAM to put the ROM data. The best way to get some memory is to create a writable [Array](#) and use its [Address](#) message.

Take care when passing the value of `data_addr` as getting it wrong may cause the system to misbehave in an unpredictable manner.

**max\_devices** limits the amount of serial number data and should be set in relation to the maximum capacity of the array. Since a 1-Wire ROM is 8 bytes long, **max\_devices** should be less than the array capacity divided by 8.

**Find** will return the number of device ROMs it has found, even if that number exceeds **max\_devices**. If there were no devices present, or there was a CRC error in any of the data then **Find** will return zero.


After a successful **Find**, the Array will have each device's unique ROM data in successive 8-byte blocks.

The order of bytes in within the 8-byte block is least-significant byte first. The ordering of devices within the whole array is based on the 1-Wire bus SEARCH ROM algorithm, where zero-bits in the least significant bit positions take the highest priority. This will automatically tend to sort devices according to family type, though not in normal numerical order.

```
To init
    Make b OneWire ($18)
    Make data_store array (8, 64, 0)
End

To main
    roms_found := b . find(data_store.length Div 8 , data_store.poi
    Print "found:",roms_found,cr
End
```

 See also [Print](#), [Put](#)

 Find is currently limited to 255 devices.

## Get

**Get** will read one byte, or a block of bytes, from the 1-Wire bus:

```
Get ⇒ Int
```

... reads a single byte from the bus.

```
Get (Int n_bytes , Int data_addr)
```

... reads **n\_bytes** from the bus and puts them in memory at address **data\_addr**.

Every byte read from the bus is combined with both CRC accumulators held by the object to


keep a running total of the overall CRCs. These CRC values may be accessed using [Checksum](#). The `data_addr` parameter is usually found by taking the [Address](#) of an Array. Take care when passing the value of `data_addr` as getting it wrong will cause the system to misbehave in an unpredictable manner.

 See also [Checksum](#)

## High


High will turn on the active pull-up channel immediately after the next byte written to the bus. If a suitable MOSFET is connected then a high current pull-up is applied to the bus until the Low message is issued.

```
b . High           ;high-current pull-up on after ...
b . Put(byte)      ; ... this byte.
Wait 11           ;wait for a minimum of 10 ms.
b . Low           ;pull-up off.
```

 See also [Low](#)

## Low

This immediately turns off the high current pull-up channel.

 See also [High](#)

## Put

Put will write one or many bytes to the 1-Wire bus.

```
Put (Int byte)
```

... writes a single byte to the bus.

```
Put (Int n_bytes , Int data_addr)
```

... writes `n_bytes` from the memory address `data_addr` to the bus.

You can use both of these when building up commands, such as MATCH ROM:

```
To match(n)
b.reset           ;initiate communication.
b.put($55)       ; '$55' is the MATCH ROM command.
b.put(8 , data_store.pointer + 8*n) ;send the ROM id
                                           ;for the device.
;we have now addressed a single device...
End
```



Every byte written to the bus is combined with both CRC accumulators held by the object to keep a running total of the overall CRCs. These CRC values may be accessed using [Checksum](#).

 See also [Checksum](#)

## Reset

**Reset**  $\Rightarrow$  Int

Reset puts a reset pulse on to the 1-Wire bus and returns a positive number if there was a 'presence pulse' detected – i.e. there were one or more devices detected on the bus.

If there was no presence pulse detected Reset will return zero.

## Example

```
Make b OneWire($18)
If b . Reset ; Any devices present?
[
    ...
]
```

Reset is used to initiate all command sequences on the 1-Wire bus.

 See also [New device on the bus](#).

## PRINT

Printing the OneWire object tells you the object type and lists all the devices connected to the bus. The format emulates the CRC, family code and serial number stamped on to the stainless steel can on Dallas iButtons:

```
-->Make b OneWire($18)
-->Print b
One Wire Bus:
00          14
000000ABCE06
7F          14
000000AC3B4D
17          01
0000082413C4
2F          01
00000822DB02
AD          23
000000175AAA
1A          23
```

```
0000000283B1
C1          23
0000001760C3
7 Devices found.
```

If there was a CRC error during the print, then this will be indicated:

```
-->Print b
One Wire Bus:
00          00
000000000000
CRC-error - try again.
```

## OperatingSystem

The OperatingSystem object system, defined in the default startup routine, provides access to system-wide features of the hardware, operating system and compiler.

Note that Venom2 allows some system messages to be called by just typing the message name alone:

- Run
- Reset
- Debug
- PrintF
- Protect

So

```
Reset
```

*Is the same as*

```
system.Reset
```

## Summary of messages

**Make**  
**Checksum**  
**Copy**  
**Count**  
**Debug**  
**ErrorAction**  
**Free**  
**Key**  
**Low**  
**Output**  
**Protect**  
**Reset**  
**Run**  
**RunMode**  
**Speed**  
**Time**  
**Valid**  
**Print**

## Creation

**Make** <object> **OperatingSystem**

An object called system of type OperatingSystem is made by the default startup routine:

**Make** **system** **OperatingSystem**

## Checksum

**Checksum**  $\Rightarrow$  int

Computes a checksum of all the operating system code.

This is used by us to set up the correct internal value for the [Valid](#) message.

## Copy

**obj.Copy**(Int **dest**, Int **source**, Int **size**)

The **Copy** message is intended for situations where you need to move some data within memory, but Venom doesn't provide a suitable mechanism.

The **Copy** message will move a given number of bytes from the source memory address to the destination memory address. The move operation will work correctly even if the source and

destination overlap.

### Parameters

**dest**: the destination address

**source**: the source address

**size**: the number of bytes of memory to move

### Warning



The source and destination addresses, and the size, are not checked for safety. If they are not correct then the results are unpredictable.

### Count

**Count**  $\Leftrightarrow$  **Int**

This system variable is incremented every time the system restarts. It is used internally to make sure that certain processes do not initialise identically each time the VM2 is restarted or reset, and it can also be used to count the number of times the VM2 has been restarted.

The count value is only valid if the system has a battery-backed external RAM. It will contain a random value if you have not initialised it.

**Count** is a signed 32 bit value.

### Example

Reset the count by hand:

```
-->System.Count := 0
```

Print the count at startup:

```
To main
  Print system.Count, CR
End
```

### Debug

**Debug** ( Int **n**, ... )  $\Leftrightarrow$  Any

The debug operating system message has options allowing the internal state of the Venom2 compiler & OS to be viewed and modified, as well as other more sophisticated functions.

**Debug** is allowable shorthand for **System.Debug**.

Typing **Debug** with no parameters will list all the available options, e.g:

```
-->debug
System Debug Command
Debug (0) Heap Dump, Debug (0, 1) check heap => error code
Debug (1, flgs) Garbage scan
Debug (2,n) Code position
Debug (3) Runtime debug flags
Debug (4, addr[,lines]) Mem Dump
Debug (5) Keywords
Debug (6) System Version
Debug (7) Compile code from object
Debug (8) No. of tasks running
Debug (9,n) Stack usage
Debug (10) Analyse code
Debug (11) SRAM test
Debug (12) Ext. Flash device code
Debug (13) Runtime error list
Debug (14) Reset-source flags
Debug (15) Erase on board Flash
Debug (16) Set/read internal flash Write Protection state
Debug (17) := Decimal point character
Debug (18, days, months) Day and Month names
Debug (19) VM2D?
Debug (20) := Disable multi-tasking
Debug (21) List I/O states
Debug (22, addr1, addr2...) Pre-initialise FSD cards
Debug (23) Break to debugger.
Debug (24) := Ctrl+C -> debugger.
Debug (25) := S/W I2C Bus flag
```

In detail:

<b>Debug(0 [,flags])</b>	Dump a listing of the heap(s) to the main serial port. Not likely to be very useful to the Venom programmer. Optional second parameter: BIT0/1: Select which heap(s) to dump. BIT2: List to terminal. Returns error code or 0.
<b>Debug(1, flags)</b>	The Venom Garbage Scanner. See <a href="#">below</a> for details.
<b>Debug(2 [,1])</b>	List the current position in the code to the terminal, including the chain of procedure calls. If a second parameter is supplied then the Venom stack is dumped.

	If the Debug (3) flags are set then dump other information too.
<b>Debug (3)</b>	Set and read the value of the Runtime Debug Flags as a 32-bit binary number. The flags values are listed <a href="#">below</a> .
<b>Debug (4, addr[,</b>	Dump memory contents starting at <b>addr</b> , optionally listing <b>n</b> lines of 16 bytes, in Hex and ASCII formats. List is dumped to main serial port.
<b>Debug (5)</b>	Keywords: list all the Venom2 keywords, class names, message names, etc.
<b>Debug (6)</b> <b>Debug (6, 1)</b>	Return the system version number as an integer representing a reversed date, e.g. <b>20100615</b> is the 15th June 2010.  If the optional second parameter is present with value 1 then return the system version that the current application in flash was compiled with. If no app in flash then returns -1.
<b>Debug (7)</b>	Compile code from a source of text during Runtime. See <a href="#">below</a> for details.
<b>Debug (8)</b>	Return the number of tasks running
<b>Debug (9, n)</b>	Control the task stacks: See <a href="#">below</a> .
<b>Debug (10)</b>	Analyse the compiled code for internal inconsistencies and print a report.
<b>Debug (11)</b>	Tests the external SRAM by exercising the unallocated block in the heap
<b>Debug (13)</b>	Runtime error list: Lists all the runtime error codes and their error text.
<b>Debug (14)</b>	Return flags indicating the <a href="#">source of the last Reset</a> .
<b>Debug (15)</b>	Erase the on-board 8MB Flash memory - removing all Flash files and erasing the 1MB Protected Application Area.
<b>Debug (16)</b>	Set and read the internal flash memory write protection state. The internal flash is where the Venom2 Language and Operating System is stored. (it's also where your application code is stored in VM2L).
<b>Debug (17)</b>	Sets the decimal point character used in all printing of floating point numbers. This defaults to '.' but may be set to any ASCII

	character, e.g. <b>Debug (17) := ' , ' .</b> Note that all reading of floating point numbers still uses the '.' character.
<b>Debug (18)</b>	Sets the strings used for days, months and ordinal suffixes when printing DateTime and RealTimeClock objects. See <a href="#">here</a> for more information.
<b>Debug (19)</b>	Return <b>True</b> (i.e. 1) if the VM2D's Display Driver IC is detected. This may be used to distinguish between a VM2D and a VM2-D2.
<b>Debug (20)</b>	Disables multitasking. <b>Debug (20) := True</b> While this is set non-zero no task swaps will happen.
<b>Debug (21)</b>	Lists the configuration state of all the VM2's I/O pins. This may be used to check that all the VM2's ports are configured into the states you expected. Please contact us if you require more information.
<b>Debug (22 , a1 , a2 . .</b>	Pre-initialises fixed SD cards, needed if you have more than 1 a1, a2 etc are the select codes used in the Filesystem <b>New</b> or <b>Make</b> command (see <a href="#">Filesystem Creation</a> )

## Garbage Scanner

The Garbage Scanner finds memory blocks that have been lost to the system due to memory leaks. A memory leak is where your application program loses track of an object and so can never delete it.

During normal operation of both the Venom2 compiler and your application, memory is requested from, and later returned to, the memory management system or 'Heap'. However, in some circumstances memory can be lost. This kind of bug can be latent in your application, only showing itself after the system has been running for some time, or when it does certain operations.

The most common causes of this kind of 'memory leak' are

1. When you create a temporary object that needs memory (within a procedure, say) then forget to remove it (explicitly with [.Die](#), or [AutoDestruct](#)) before returning from the procedure.
2. When you assign a new value to the only variable that refers to an object.

If you do this enough times then the system will eventually run out of memory. A memory leak, typically, only becomes apparent when your application code stops or resets on a RAM Full

runtime error. Initially you may not know it's a runtime error as you might not be there to observe it, so it may appear as random resetting of the controller.

We recommend that if your program is at all complicated, that you use the Garbage Scanner as part of your code validation process. To do this, put in a call to the Garbage Scanner somewhere in your code (in the main task only). Let your system run for some time, and exercise as much of its functionality as you can. Then signal to your code to run the Garbage Scanner.

#### Operational details

**Debug (1)** returns the number of leaked blocks found:

```
-->Print Debug(1)
3
```

If you include the optional 2nd parameter with a value of 1, it will list out the details of the lost blocks. The block listing looks like this:

```
-->Debug(1, 1)
Garbage block list:

30 bytes at $200AD8: Buffer?
262 bytes at $200AFC: <Unknown>
46 bytes at $2020BE: Sub-class of GraphicsLCD?
Found 3 Garbage blocks
```

The list of garbage blocks indicates the size and address of the block, and also a good guess as to what kind of object may have used the block.

*Note: the 'Unknown' block is actually a second block used by the Buffer object that didn't contain enough information for the garbage scanner to identify it.*

There are legitimate heap blocks that sometimes get into the garbage list:

1. Entering a command at the `-->` prompt that contains a string constant (e.g. `-->Print "a string" , CR`).
2. CTRL-C breaking out of a program while it is running - any temporary objects that were held in local variables will show up as garbage, as even if you wrote code to destroy them after use, this won't have been called.
3. CTRL-C breaking out of a procedure definition part way through compilation. This leaves compiler 'garbage' in the system that is harmless but will show up in a garbage scan.
4. Sometimes, in a multi-tasking system, an object may use some heap blocks temporarily and then free them after a short while. You may see these blocks in the garbage list. Only blocks that consistently appear in the garbage list need to be taken seriously.

Make sure that none of these is giving you false indications of a memory leak.

Also note that **Debug (1 . . .)** will NOT respect task swap timing, so don't use it as part of a



normal application unless you take this into account.

## Runtime Debug Flags

The runtime debug flags enable debug options in the runtime system. The flags are arranged as a set of binary bits, within the value of **Debug (3)**.

Bit value	Operation	Description
1	Stack Dump	Whenever a runtime error occurs, dump the Venom stack.
2	List Error Bytecode	Whenever a runtime error occurs, list the exact code address and bytecode at that address.
4	System Stack Dump	Whenever a runtime error occurs, dump the system stack. Probably not useful to the Venom programmer.
8	Enable Runtime Error: <i>Attempt to lock object held by dead task.</i>	Turn on <i>Attempt to lock object held by dead task</i> runtime error. See below.
16	Heap Dump	Whenever a runtime error occurs, dump the Venom Heap(s).

For example you might put the following line in your **init** procedure:

```
Debug(3) := 1+2+16 ; Dump V.stack, bytecode and heap.
```

### Enable *Attempt to lock object held by dead task* runtime error

If a task holds a lock on a resource, then that resource may not be locked by any other task. If the task ends while still holding the lock then the resource can never be unlocked from within your program. However the operating system will detect this situation and silently unlock the resource, though the task requesting the lock may have to wait for up to around  $255 * \text{Number of Tasks}$  mS before the situation is resolved.

If you suspect this is happening (unexplained short pauses in your system) then you can force the OS to generate a runtime error instead of silently unlocking the resource, to see if that is what is causing the pause. If the pauses are unacceptable then you will have to recode your application so that the dead task doesn't hold any locks.

## Last Reset Source

**Debug (14)** will return a number containing a set of binary flags indicating what event last reset the VM2. The flags are:

Binary Bit No.	Bit value	Reset Source
7	128	Low voltage detector
6	64	Window watchdog (not used in VM2 family)
5	32	Watchdog
4	16	Software Reset
3	8	Power on Reset
2	4	Reset pin

### Runtime compiler

The runtime compiler allows you to compile code from any text stream, from within a running application. This may be useful to temporarily change the behaviour of an application to suit particular circumstances.

The source text can come from any object that returns characters from a Get message.

Compilation can be done from any task (though for compiling larger procedures it is recommended that the main task be used as it has a larger stack, and it's important to make sure that two tasks are not compiling at the same time). The syntax for compiling is currently:

**Debug (7, obj)**

**obj** is an object containing the text for new procedures. The **Debug (7 . . .)** call has to be done once for each procedure in obj.

### Notes

1. When an application is stored in flash, any newly compiled procedures will be held in RAM. Newly compiled procedures will temporarily 'overlay' existing ones that have the same name. At the next startup all new procedures will be deleted and the originals will be restored from flash.
2. The maximum line length allowed in Venom is 255 characters. Lines can be broken by using CR or any valid line termination sequence (\r, \n, \r\n). The last line should end with CR, etc, to make sure no attempt is made to read past the end of the object's text.
3. Any syntax errors that occur in the code will be sent out of the system output stream object (usually the main serial port) - it's best to make sure there are no syntax errors in the code first. Any runtime errors that occur when accessing the object will result in a runtime error report but program execution will continue.

## Task stack usage

Each task in Venom is given standard amounts of stack to use. However there may be some situations where these values need to be changed.

Debug (9) allows you to monitor how much stack each task in your program has used, and then set the amount of stack given to each task, for example if you need to save memory or provide more stack for complex code.

### Monitoring stack use

Each task has two stacks - the *Venom stack* and the *System stack*.

**Debug (9,0)** is an active variable that allows you to turn *stack use reporting* on and off. When it is turned on CTRL-T or List Task will report the memory allocated to each stack, and the amount used by the task up to this point.

**Debug(9,0) := 1 ; turn on stack use reporting.**

You should turn this feature on early in your application, make sure your code has been fully exercised, then use CTRL-T or List Task to show you how much stack each task uses. You may then adjust how much stack each task is allocated using the options listed below.

### Set new task stack allocation

**Debug (9,1)** and **Debug (9,2)** are active variables that allow you to set the sizes of the stacks in new ('child') tasks created from the current ('parent') task.

- **Debug (9,1)** sets (or reads back) the Venom stack size (in bytes) for next task started by the current task.
- **Debug (9,2)** sets (or reads back) the System stack size (in bytes) for next task started by the current task.

For example

```
Debug(9,1) := 500 ; Set new task's Venom stack size to this
Debug(9,2) := 400 ; Set new task's System stack size to this
Start my_new_task
```

Each task inherits the values of Debug(9,1) and Debug(9,2) from its 'parent' task.

The main task (Task ID: 0) is not created using Start, so you can't set its stack's sizes using this. The following mechanism is used instead:

### Set size of main task Venom stack (Not available in VM2L)

**Debug (9,3)** will set or read the size of the *main task's* Venom stack. *This should only be called from the main task, and it should be called before any other tasks have been*

*started* (actually, before any other blocks have been allocated on the *system heap*) - i.e shortly after startup. If these conditions are not met a runtime 'Resource' error is generated, and the system will be in an unstable state and should be reset.

For example:

```
Debug(9,3) := 2000 ; Set main task Venom stack size
```

Note: don't turn on stack use reporting before setting the size of the main task Venom stack.

Note: the size of the main task System stack is fixed.

## Internal Flash Write Protection State

```
Debug(16) ⇔ Int write_protection_state
```

**Debug(16)** sets and reads the internal flash memory write protection state. The internal flash is where the Venom2 Language and Operating System are stored. (it's also where your application code is stored in VM2L).

Setting **Debug(16)** to **\$FFFFFFFF** will write-protect the internal flash, and setting it to **0** will un-protect it.

(Actually the value you use is a bit pattern contained in a 32-bit number. Each bit corresponds to a page in the flash).

Note that setting **Debug(16)** will always cause the controller to reset, as the new write protection state is only valid after a reset.

### Use with Protect(3)

In order for **Protect(3)** to be able to re-program the Venom2 Language and OS you have to make sure that the internal flash is unprotected with

```
Debug(16) := 0
```

Because this resets the controller, you might want to use code like this in your init procedure so that the controller doesn't need to reset just before calling **Protect(3)**:

```
To init
...
;Make sure VM2 internal flash is not write protected:
If Debug(16)
    Debug(16) := 0 ; Clear write protection; resets controller.
...
End
```

## Date format locale

You can change the day month and ordinal suffix strings, as printed by [DateTime](#) and [RealTimeClock](#), from the English defaults to any text you want.

Once set, these strings will remain current until they are set again or the next time the controller restarts. The syntax is as follows:

```
Debug (18 ,  
  Int  short_day_strings ,  
  Int  short_month_strings ,  
  Int  long_day_strings ,  
  Int  long_month_strings ,  
  Int  ordinal_strings ,  
  Int  ordinal_indexes)
```

*Note: All parameters are optional; though you can't miss out a parameter, you don't have to supply all of them.*

You must define some arrays to hold the new names, and then pass the addresses of their data to **Debug (18)**.

You can also set the ordinal strings - 1<sup>st</sup>, 2<sup>nd</sup>, etc, in English. This is done using two arrays - an array of strings to carry each of the ordinal strings that exist in the language, and an array of indexes to index into the strings based on the day number (starting from 0, though this value is never used).

Note that there is a dummy string at the beginning of the month names so that month numbers starting at 1 will index the correct name.

Note that the day of week numbers start at 0 = Sunday.

The code below will replicate the English day, month and ordinal strings. Modify this for your own use.

```
To ChangeLocale  
  Debug (18 ,  
    short_day_strs.Address ,  
    short_month_strs.Address  
    long_day_strs.Address ,  
    long_month_strs.Address  
    ordinal_strs.Address ,  
    ordinal_inds.Address)  
End  
  
Array short_day_strs (String, 7)  
  "Sun"  
  "Mon"  
  "Tue"  
  "Wed"  
  "Thu"  
  "Fri"
```

```
"Sat"
```

```
End
```

```
Array short_month_strs (String, 13)
```

```
"Jan"
```

```
"Feb"
```

```
"Mar"
```

```
"Apr"
```

```
"May"
```

```
"Jun"
```

```
"Jul"
```

```
"Aug"
```

```
"Sep"
```

```
"Oct"
```

```
"Nov"
```

```
"Dec"
```

```
End
```

```
Array long_day_strs (String, 7)
```

```
"Sunday"
```

```
"Monday"
```

```
"Tuesday"
```

```
"Wednesday"
```

```
"Thursday"
```

```
"Friday"
```

```
"Saturday"
```

```
End
```

```
Array long_month_strs (String, 13)
```

```
"January"
```

```
"February"
```

```
"March"
```

```
"April"
```

```
"May"
```

```
"June"
```

```
"July"
```

```
"August"
```

```
"September"
```

```
"October"
```

```
"November"
```

```
"December"
```

```
End
```

```
;Array to hold ordinal suffix strings so  
;day numbers can print as '1st', '2nd', etc.
```

```
Array ordinal_strs (String, 4)
```

```
"st", ; 0
```

```

    "nd", ; 1
    "rd", ; 2
    "th", ; 3
End

; This array is used to convert from the day number (1-31)
; to one of the ordinal strings defined in the array above.
Array ordinal_inds(Int 8, 32)
    3, ; 0
    0,1,2,3,3,3,3,3,3,3, ; 1-10
    3,3,3,3,3,3,3,3,3,3, ; 11-20
    0,1,2,3,3,3,3,3,3,3, ; 21-30
    0 ; 31
End

```

## ErrorAction

**ErrorAction**  $\Leftrightarrow$  Int

If ErrorAction is 0 then control returns to the command line prompt when a runtime error occurs.

If it is set to the value '1' a runtime error will force Venom to reset the controller hardware.

*Meaningful error action values may be expanded beyond 0 and 1 in future releases.*

Note: The default startup procedure sets `System.ErrorAction` depending on the state of the program mode switch, to give the runtime error behaviour most commonly required. You can override this safe, default behaviour by setting `System.ErrorAction` in your code if you need to.



If the default ErrorAction behaviour is changed, your application may not reset on runtime errors when it is in the field.

## Free

**Free**  $\Rightarrow$  Int

Returns the total amount of free RAM in the controller (in the main heap).

Using a parameter, it will also report on other aspects of the heap:

**Free**(Int **n**)  $\Rightarrow$  Int

<b>Free (0)</b>	Heap memory free, total
<b>Free (1)</b>	Largest free block in the heap
<b>Free (2)</b>	The size of the 'unallocated block' of free memory at the top of the heap.
<b>Free (3)</b>	Total size of the main heap

## Key

**Key**(Int **which**)  $\Rightarrow$  Int

**which** has the value 0, 1 or 2 and selects which of three system ID numbers to return

The VM2's CPU has a unique factory-programmed system ID, which is different for each unit manufactured and cannot be changed. It consists of 96 bits of information which can be accessed as 3 x 32 bit integers using this system message.

The meaning of the bits is not specified, but empirically it seems that Key(2) contains a serial number which varies from unit to unit.

The serial number information may be useful as a basis for generating codes like encryption keys, or for positively identifying VM2 units in a network.

## Low

**Low**

This message will look for all the VM2 I/O channels that have not been explicitly set to an I/O state and set them to 'input pulled low'.



This helps to reduce the current consumption in critical applications, especially when using [STOP mode](#).

It is typically used at the end of an Init procedure - after all the main I/O objects have been defined.

### Example

```
To init
  Make s2 SerialPort(9600,2)
  Make relay Digital($10,1)
  ...
  system.Low ; Pull everything else low.
End
```

### Inputs that have already been set to float

**Low** will not set any I/O channels that have been explicitly or implicitly (by creating an I/O object) set to 'floating'.

You will have to deal with these explicitly. For a list of the I/O states of each channel you can type **Debug (21)** at the command line. Any channels that are in the Floating Input state can be set to any state you wish using something like:

```
New Digital(channel_number, %10) ; Pull an input low
```

Doing this will not affect the logical operation of the I/O object.

(Note that **Debug (21)** may list I/O channels that are used internally to the VM2 - any of these that are floating will not need to be set explicitly).

### Output

**Output**  $\Leftrightarrow$  Any

**Output(Int n)**  $\Leftrightarrow$  Any

### Redirecting text

**System.Output(n)** hold the various output streams associated with the OperatingSystem object. Changing their values allows these text streams to be redirected.

Output	Text redirected	Default output stream
--------	-----------------	-----------------------

<b>System.Output</b>	Text from Print, List and HELP	serial
<b>System.Output(0)</b>	<i>Same as above</i>	serial
<b>System.Output(1)</b>	Runtime Error reports	system

### Redirecting normal text

Setting Output(0) redirects the text from Print, List and HELP:

```
System.Output(0) := new_output_device
```

If you redirect the output to Nil then the output is simply discarded:

```
System.Output(0) := Nil
```

### Redirecting runtime error text

Setting Output(1) redirects the text from runtime error reports, for example to an error logging file. It is usual to use the 'maximum size' feature of error logging files to prevent them becoming too large.

```
System.Output(1) := err_file
```

When redirecting error output try to use an object that is not used by other parts of the system: if the object you are redirecting text to is locked by another task then the text output will stall waiting for the object to become unlocked. This could prevent you from seeing error reports or from using control-C to break out of a program.

To get around this problem in normal use, error text is directed by default to the system object, which simply sends it directly out of the main serial port without respecting the serial object's locks.

*It is possible to redirect any print to the system object but this is not normally recommended, as applications normally require locking to be respected.*

### Reading

Reading Output returns the value of the output stream object.

### Protect

```
Protect (Int code) ⇒ Int
```

**Protect** sets protection from accidental loss of your finished Venom application, turning the 'software' you have developed in RAM into 'firmware' in the Flash memory.

**Protect** also allows you to create a distributable firmware file and to use this to update other

VM2s.

When you download code into a VM2 it is compiled and the held in the battery-backed RAM. This gives a very fast compile-execute code development cycle because there is no need to erase and reprogram a Flash memory. However, it is not safe to send your VM2 out into the field with its application code held in RAM. You should protect it in the *Protected Application Area*, located in non-volatile Flash memory.

**Protect** is allowable shorthand for **System.Protect**.

**Protect** may also be used to

- Erase an application from the Protected Application Area
- Create a firmware file for production or distribution
- Reprogram a VM2 using a firmware file

The table details the functions of the Protect message:

<b>Protect (0)</b>	Erases the Protected Application Area
<b>Protect (1, [flags])</b>	Copies the application from RAM into the Protected Application Area. The optional flags parameter allows you to set security levels - see below.
<b>Protect (2, appfn, o</b>	Creates binary distribution files (.vex and/or .vos). <i>No longer recommended. See Protect(4).</i>
<b>Protect (3)</b>	Looks for a firmware update file (extension .vfu) in the root directory of the Flash Filing System and uses it to update the VM2's firmware. (The older .vex and .vos file format are still supported.)
<b>Protect (4, filename)</b>	Creates a distributable firmware file. This file is a combination of your application code and the Venom system that will run it, ensuring that the two are always matched. See <a href="#">below</a> for how to use Protect(4).

It is normal to use **Protect** with parameters 0, 1 and 4 at the command line, rather than as part of a program, whereas **Protect(3)** is normally used as part of an application program.

#### **Protect(1) flags**

Venom applications in Flash are quite secure from reverse engineering: Venom is compiled down to bytcodes, so your source code can never be read back out of a VM2. However, some

users may want to secure their application code and flash files even further. **Protect(1)** has an optional **flags** parameter that allows you to set different levels of protection over your application. This table sets out the meanings of the binary bits in **flags**.

Binary flag value	When set
<b>1</b>	No command line. There is no way to gain access to a Venom command line, even in Program Mode. The only option is to erase the application. <i>Note: you will need to ensure your application code never terminates.</i>
<b>2</b>	No automatic USB access to the Flash File System at startup in Program Mode. This means no one can see or alter the files in the Flash Filing system.

#### Example

**Protect(1,3) ; No command line, no USB access.**

### Create binary executable files for production and distribution

*Not possible in VM2L*

```
Protect (4, String filename, String text, [int c_addr, int c_size]
```

#### Parameters:

**4:** Specifies firmware update file production.

**filename:** the name of the firmware update file to be produced.

**text:** some text you can embed in the file.

*Optional:*

**c\_addr:** the address of any auxiliary C code you want to include in the firmware.

**c\_size:** the size of any auxiliary C code you want to include in the firmware.

**Protect(4)** creates a binary file that combines your Venom application and the Venom RTOS that runs it. These files normally have the extension **.vfu** (Venom Firmware Update).

A vfu file can be loaded into another VM2 to 'clone' your firmware. This is useful during production, and it is also useful for distributing new versions of your firmware to your customers. Note that Venom source code is not included in a vfu file.

This is the suggested sequence of operations to use **Protect(4)** :

1. If necessary, clear the Protected Application Area using **Protect(0)**.
2. Download your application code into RAM in the usual way
3. (Optionally use **Protect(1, flags)** if you want to set the [security flags](#)).

4. Type `Protect(4, "myapp.xxx", "a description")`; you specify a filename, including a dummy extension, for the new distribution file, and you have the option of embedding some text into the file that may be used to indicate what it contains.
5. The file will be created with the name you supplied in the root directory of the VM2's Flash Filing System.
6. Note how we are using use a *dummy* filename extension at this stage, so we can't accidentally use the file to reprogram *this* VM2.
7. The optional text might be used to record the application name and version number. You can later read this text, from the VM2, or by opening the file in a text editor, but you can't change it as the file is validated using a CRC.
8. Connect the VM2 to a PC using a USB connection and copy the new file to your PC.
9. On your PC, rename the file with the correct extension: `.vfu`
10. This file is the distribution firmware for that version of your application - keep it in a safe place.
11. Remember to keep you application source code safe too.

#### Example

```
Protect(4, "my_app.xxx", "Heating application v20120524")
```

See below for how to use the `.vfu` file you have created.

## Updating your firmware

*Not possible in VM2L*

There are several ways to update a VM2's firmware.

### Production Programming

The first way is to load a firmware update file (\*.vfu) into the Flash Filing System of a VM2 (using a USB connection) then reset the VM2 in Program Mode. This is useful for when you have the VM2 to be updated sitting in front of you. See [here](#) for more details.

### Under program control

The second way is to transfer the firmware update file into a VM2 (using USB, FTP, Serial File Transfer, SD Card or other means) and then have the VM2 application code call `Protect(3)`. You must ensure that the transferred file is [contiguous](#), and you may need to ensure that the internal flash memory is not [write-protected](#). After calling `Protect(3)` the controller will reprogram itself and then restart running the new application and operating system. Please see the code snippet *Remote Firmware Update* [on our website](#) for an example of how to do this.

Both of the above methods interrogate the Flash Filing System to see if it has a file with extension vfu (or vex/vos) in the root directory, and then uses the first one that it finds. The file is

deleted if it is used.



*Bug alert: You must make sure that there are no other tasks running when you call `Protect(3)`. This is because when the Application Area is erased the other tasks will have their code deleted, causing them to crash. One way to ensure this is to call `Protect(3)` from the main task, having stopped other tasks with **Stop All**. It may also work to call `Protect(3)` from any task, but first turn off multi-tasking using [Debug](#). Note that you will not see this kind of bug until the code is run from Flash.*

### Caution

Note: if the VM2 is either **reset** or **powered down** while it is updating it may lose either the application code, the operating system code, or both. If this happens the only way to recover is to download a new application or operating system using VenomIDE or similar. This cannot be done remotely.



There is 1M Byte of application area available in the VM2 and VM2D, and 64K Bytes in the VM2L. Venom2 is very efficient with application memory, so 1MB should be enough for very large applications.



[ROMing](#), [Copy](#)

### Reset

#### **Reset**

Resets the controller. If it is in run mode then the application will run, else you will get the startup banner.

The RESET\ signal on the VM2 controller will be pulled low for around 28uS.


**Reset** is allowable shorthand for **System.Reset**.

### Run

#### **Run**

Resets the controller, and then runs the application as if in Run Mode, even if the Program Mode switch is on.

**Run** is allowable shorthand for **System.Run**.

 Note that some of your external circuits may not be connected to the RESET\ signal, but other circuits may only be reset by the removal of the power supply.

## RunMode

**RunMode**  $\Rightarrow$  Int  
**RunMode (1)**  $\Rightarrow$  Int

When no parameters are supplied, **RunMode** returns **True** if the system is in **Run Mode**, or **False** if the system is in **Program Mode**.

```
IF system.RunMode ; Are we in Run mode?
[
]
```

## Read state of Prog Mode switch

If a non-zero parameter is supplied then **RunMode** returns **True** if the Prog Mode switch is *off*, and **False** otherwise.

There is a difference because the system is in **Run Mode** when the user sends **Run**, even though the Prog Mode switch may be *on*.

```
IF system.RunMode(1) ; Read the actual state of the switch...
[
]
```

## Speed

**Speed**  $\Leftrightarrow$  Int

The Speed message controls the master core clock speed on the VM2.

You can set the speed in increments of 8MHz from 16 to 72MHz.

The VM2 will always startup with Speed set to 72MHz.

If you lower the clock speed then both the processing speed and the power consumption of the controller will be lower roughly in proportion. However both of these effects are non-linear.

You will get more processing power than expected as you go below 48 and 24MHz because fewer wait states are used in accessing the flash memory. Also you may get less saving in power than expected because of leakage currents.

### Example

```
Speed := 16
```

## Permanently low speed systems

If you want your VM2 to always run at a low speed it may be simplest to put the speed change very early in your application code, before any other objects are created. I.e. in the startup procedure. If so you should **List startup**, copy it into your source file, and modify it like this:

```
To startup
  Make system OperatingSystem
  System.ErrorAction := New Digital($20).Asserted + 1
  System.Speed := 16
  Make serial SerialPort(115200,1,1)
  Make net I2Cbus
  Make led OnBoardLED
  Make clock RealTimeClock
  If Runmode
  [ led.Flash($80)
    init
    main
    led.Flash(0)
  ]
End
```

The VM2 will always start up at 72MHz, but will slow down when it gets to the Speed command and run the rest of the application at the new speed.

## Dynamic speed control


However, if you want to change speed dynamically you can do this, but you will have to reset the speeds of some objects, like serial ports or I2CBusses, as their speeds will have been defined relative to the original clock speed. The procedure below shows how this can be done:

```
To change_speed(sp)
  Local temp := serial.Speed ; Record the original serial speed.
  System.Speed := sp ; Set the master clock speed.
  serial.Speed := temp ; Now reset the serial speed.

  net.Reset ; Reset the I2C Bus to take account of new system speed
End
```

Note that system commands and objects that depend on Venom2's internal millisecond timing, like Wait, Every Timer and Stopwatch, don't need to be adjusted - they take account of the new clock speed automatically. The RealTimeClock is not affected as it has its own independent clock source.



 See *Speed message* in the index to find objects whose speed may need to be adjusted.

 Note that the USB Device port will only operate if Speed is set to 72 or 48 MHz. The USB Host port (included on some Application Boards) has it's own clock and so is not affected.

## Time

```
time ⇒ Int
time(Int type) ⇒ Int
```

Returns the value of one of two 32 bit counters that are incremented at regular intervals

Type = 0          Return microsecond counter value  
(default)

Type = 1          Return millisecond counter value

These values can be viewed as 32 bit counters, both initialised to 0 then the VM2 starts, one incremented every microsecond and the other incremented every millisecond.

Both can be used for simple timing measurements by subtracting a stored earlier value from a later value. Even if the timer wraps round between readings, the subtracted value will be valid as long as there was only 1 wrap round event and the difference is positive, corresponding to a maximum measured time interval of about 35 minutes for the microsecond timer and about 24 days for the millisecond timer.

**System.Time(0)** is the only way to measure microsecond times, and System.Time(1) is similar to the use of a [Stopwatch](#) object.

Using **System.Time(0)** to time external events like keystrokes or network response times might be a useful way of generating unpredictable values e.g. to seed a random number generator or create an encryption key.

## Example

```
; measure a floating point division time in microseconds
To Test
  Local a, b, t, x
  a := 100.0
  b := 12.0
  t := System.time
  x := a / b
  t := System.time - t
  Printf("Operation took %u microseconds\n", t)
```

**End**

Note that there are overheads - in the time reading process, and in loading a and b and storing in x. You can attempt to separate some of these overheads from the division operation by using the line

```
x := a
```

instead, and running the test again.

## Valid

**Valid**  $\Rightarrow$  int

Returned value is true (1) if the operating system's computed checksum (see [Checksum](#) message) matches the value stored in the ROM when it was created.

**System.Valid** is False (0) if the checksum does not match, indicating the possibility of ROM corruption if the OS is a released version of Venom in which the internal checksum has been set properly by us.

## PRINT

**Print** <OperatingSystem>

Printing the system object gives the size of the symbol table and global area, and the amount of the heap memory free. Other general system information may be added from time to time.

```
-->Print system
Symbol table 55 bytes
8 Global variables
99814 Heap bytes free (biggest block 99700)
NV RAM area 0 bytes (0 unused)
-->
```

## PIDController

The PIDController object performs the functions of a classic PID controller. A PID control system is a form of feedback loop for controlling continuous processes.

A PID controller reads the current state of a 'set point' and a 'process variable' (such as a desired temperature and an actual temperature) and manipulates the process it controls to try to bring the process variable equal to the set point. It does this by applying negative feedback of three different kinds: *Proportional*, *Integral* and *Differential*.

1. Proportional control is feedback that is proportional to the current *error* (the difference between the set point and the process variable).
2. Integral control is feedback proportional to the *sum of all previous errors*. This removes offset errors, which proportional control typically can't do.
3. Differential control is feedback proportional to the *rate of change of the error*. This is not often used, but where it is used it may be seen as having a 'damping' effect.

Proportional control may be seen as correcting for errors in the *present*; integral control may be seen as correcting for errors in the *past*, and differential control may be seen as correcting for errors in the *future*.

The PID controller implemented by this object is arranged in the form

$$V_m = G * (err + \Sigma err / T_{int} + \Delta err / \Delta T * T_{diff})$$

Where  $V_m$  is the 'manipulated variable' (the output of the PID controller, or the input to the system to be controlled);  $G$  is the overall gain of the feedback loop,  $err$  is the difference between the set point and the process variable, and  $T_{int}$  and  $T_{diff}$  are the time constants for the integral and differential actions within the PID controller.

This form of the the PID loop allows the integral and differential coefficients to be expressed in terms of *time*: the time over which past errors are corrected for, and the time over which future errors are anticipated.

## Additions to the classic PID

This implementation of the PID controller has the following additions:

### 1. Anti-windup

If the output of the PID controller (or, actually, the input to the process) saturates then the integral term can accumulate without limit to values that don't reflect physical reality. This is called 'integral wind-up', and should be avoided. The PIDController object has anti-wind-up action in the form of output saturation limits, which limit the PID's output value, and beyond which the integrator doesn't accumulate. Optionally, while the output is saturated at Min or Max, the integral term will decay towards the saturation limit.

This anti-wind-up action can't compensate for wind-up due to the process input saturating, so you should take care that the PIDController's saturation limits are set inside the saturation limits of the process input.

### 2. Digital filter

To avoid the differential term generating excessive spikes in the output, the input to the differential term may be digitally filtered with an exponential decay response.

## Live control

All of the external parameters, and some of the internal state of the PID controller may be altered live, while the object is running. See [Value](#) and [Reset](#) and [Debug](#).

## Summary of messages

**Make**  
**Debug**  
**Reset**  
**Value**  
**Update**

## Creation

```
Make <object> PIDController (Float Gain, Float Min,  

Float Max, [Float Tint, [Float Taw, [Float Tdiff,  

[Float Tfilt]]])
```

## Parameters

**Gain:** This is the overall gain of the controller feedback loop

**Min, Max:** The saturation limits; the output value (as returned by [Update](#)) is kept within Min and Max.

**Tint:** this is the Integral Time.

**Taw:** this is the '[anti-windup](#)' relaxation time: while the output is saturated at Min or Max, then the integral term decays towards the saturation limit with a time constant of **Taw**.

**Tdiff:** this is the differential term time constant.

**Tfilt:** this is the digital filter time constant.

## Example code

This code illustrates using the PIDController; some parts of the code are not shown:

```
; PID Parameters:      Gain  Min  Max    Ti    Taw  [Td, Tf not  
MAKE PIDC PIDController(20.0, 0.0, 100.0, 75.0, 10.0)  
  
To PidLoop  
  SetPoint := 50.0  
  EVERY 100  
  [  
    OvenTemperature:= ReadThermometer  
    ; Calcluate the PID output:
```

```
    HeaterPower := PIDC.Update(SetPoint, OvenTemperature)
]
End
```

## Suggested values

**Gain:** There is no suggested value for this as it is very dependent on the system you are controlling.

The value of the output due to the gain is this:  $\text{output} = \text{error} * \text{gain}$ .

**Min, Max:** These should definitely be set to the values of the output that are meaningful. E.g. if a heater is controlled using a value ranging from 0-100 then min should be 0 and max should be 100.

Setting **Max** and **Min** wider than this will result in no control over windup.

**Tint** is likely to need to be much greater than **1.0**.

You can think of it as the time constant, in units of your control loop time, over which the integral term tries to correct offset errors.

**Taw** is usually  $1.0 < \text{Taw} < \text{Tint}$ .

You can think of it as the time constant, in units of your control loop time, over which the anti-windup action reduces windup in the integral term if the output of the PID controller saturates at Min or Max.

If you set it too short, any small saturation of the output will wipe out the integral term, which will then need to be built up again.

If you set it too long, any integral windup that occurs will not be corrected quickly.

**Tdiff:** *no suggested value at this time.*

You can think of it as the time constant, in units of your control loop time, of the damping is applied to the control output, to correct for overshooting the set point at high values of Gain.

## Turning off I, D and other terms

If any of **Tint**, **Taw**, or **Tfilt** are less than **1.0** then their associated action is turned off.

If **Tdiff** is **0.0** then differential action is turned off.

## Debug

**Debug** (Int **index**)  $\Rightarrow$  Float

The Debug message allows you to probe some of the working values inside the PIDController object.

Index	Internal value returned	
0	Output before Min/Max limits applied	
1	Proportional term (or the 'error' term)	
2	Integral term	
3	Differential term	
4	Digital filter output value	

## Reset

**Reset** [(Float **integral\_term**)]

Reset allow you to force the PIDController to behave as if it had just been created. Specifically, it sets the internal value of the 'integral term' to 0.0, or to any value you supply in the optional parameter, and it sets a flag so that the differential term and digital filter don't see any discontinuity in the process variable value.

## Value

**Value** (Int **index**)  $\Leftrightarrow$  Float

The Value active variable allows read/write access to any of the external parameters of the PIDController object. These are the same parameters that are passed when the object is [created](#), numbered from 0.

## Update

**Update** (Float **set\_point**, Float **process\_variable**)  $\Rightarrow$  Float

Update calculates the next iteration in the PID control algorithm and returns a new value for the 'manipulated variable'. This is the value that should be applied to the control system to attempt

to move the process variable towards the set point. For example in an oven control system, it might be the heater voltage. In this case the set point would be the desired temperature, and the process variable would be the measured temperature.

Note that Update returns a floating point value - and that many (all?) outputs likely to be controlled by the PIDController will take an integer value to control them (e.g. a DAC or PWM).

So the float value must be converted to an integer using [As Int](#).


## Parameters

**setpoint:** this is the target value for the control system to maintain.

**process\_variable:** this is the actual (usually measured) value of the system.

## Avoiding discontinuities

The first time Update is called after the object is created, or after the object is [Reset](#), the differential calculation and the digital filter are initialised with 'T-1' or 'previous' values equal to the value of the current process variable.

 Update typically takes around 20uS to calculate.

## POP3Mailbox

POP (Post Office Protocol) version 3 is a protocol for fetching mail from an external mailbox. It is a popular mechanism for users on the end of a dialup link to collect mail from a mailbox on their ISP's server.

A POP3 object is a holder for a mailbox list. When opened it accesses a list of incoming messages corresponding to the mailbox on the ISP's server. The messages are referenced by number, and the POP3 object allows the program to retrieve headers and/or message body for local processing and to delete messages from the remote mailbox.

A POP3 client must log in to a POP3 server, supplying a user name and password, in order to prove that it is entitled to read and delete messages belonging to that user.

See also [TCP/IP Networking](#).

## Summary of messages

**Make**  
**Close**  
**Count**  
**Element**  
**Length**  
**Open**  
**Print**  
**Remove**

## Creation

**Make** <object> **POP3Mailbox** (Str **server**)

**server** is a string or text buffer containing the domain name of the mail server that will be used. Typically this information is supplied by an ISP when the account is opened. A typical value might be "**pop3.ukfsn.org**".

This creates a mailbox object. Its initial state is empty and closed.

## Close

**Close**

Close a mailbox. The TCP connection to the server is closed, and afterwards no access is possible to the mailbox contents. A side-effect is the physical deletion of any messages on the server that were previously marked for deletion with the **Remove** message.

## Count

**Count**  $\Rightarrow$  **Int**

If the mailbox is open, returns the number of incoming messages listed

## Element

**Element**(Int **msgno**, Str **tag**)  $\Rightarrow$  **String**

**msgno** is the message number.

**tag** is a string or text buffer to match the name or first part of the name of a header element.

A header element is one of a sequence of lines of text which precede the message and contain control information. Common examples are:



- **From:** shows the email address from which the message was sent
- **To:** shows the email address of the intended recipient of the message
- **Subject:** What the message is about
- **Reply-to:** Who to reply to, if not same as **From:**

The header element name is terminated with a colon. The tag parameter to the Element message must match the first part of the tag. The matching is not case sensitive. E.g. "Date" and "date: " will both match "Date: "

The value returned is the address of a fixed string which is valid as long as the POP3 object is open. The string contents are the header line starting from the first non-blank character after the colon.

#### Example

```
-->print pop3.element(3, "from"), CR
sales@venomcontrolsystems.co.uk
-->
```

## Length

**Length( Int msgno) ⇒ Int**

**msgno** is the message number assigned by the server.

This message returns the number of characters in an email message, including the headers.

## Open

**Open(Str username, Str password) ⇒ Int**

**username** defines the user whose mailbox is to be opened.

**password** must be the correct password for that user.

Each parameter can be a string or text buffer.

This message makes a TCP connection to the mail (POP3) server specified when the POP3 object was created. The value returned is:

- 0 or positive number: the number of messages in the mailbox
- -1 : unable to make a TCP connection
- -2 : POP3 authentication failed
- -3 : some other error or timeout during the exchange of messages after logging in

## Remove

### Remove (msgno)

Marks the specified message for deletion. Runtime error if out of range. Note that after marking a message for deletion all the messages are still in the mailbox and have the same numbers. Actual deletion occurs when the mailbox is closed.

## PRINT

**Print** <POP3Mailbox>

Prints a list of message numbers and the size of each message

**Print** <POP3Mailbox>:msgno[:maxlines[:maxchars]]

Prints the message contents or the message headers, controlled by colon parameters.

**msgno** Message number, starting at 1 for the first in the mailbox.

Runtime error if out of range.

**maxlines** 0 : print the message headers instead of the body.

Non-zero: print the message body up to the maximum number of lines specified.

Default: 100

**maxchars** The maximum number of characters to print from the message body. It will override the **maxlines** setting if this limit is reached first.

Default: 5000

## PrintJob

PrintJob objects are created by the system and the only context you will ever see them is if you define an [AcceptPrintJob](#) method within a [Class](#) you have created.

Most of the time you will not need to send any messages to a PrintJob object - you'll only need to pass it on to another object as a parameter to an AcceptPrintJob message.

## Summary of Messages

**Get**

**Queue**

**Status**

## Get

**Get**  $\Rightarrow$  Int

Get will fetch a character from the PrintJob and return it as an integer.

If there are no characters left in the PrintJob then Get will return **-1**.

See also [Queue](#)

## Escape sequences in PrintJobs

Some Print keywords including **Font**, **GotoXY**, **Left**, **Right**, **Centre**, **Htab** and **Vtab** are encoded as escape character sequences within the print job. These sequences begin with the ESC character (ASCII 27) and are followed by up to 5 further characters:

Print keyword	Sequence of characters
<b>GotoXY</b>	27, 'G', XL, XH, YL, YH
<b>Font</b>	27, 'F', (FN + 1)
<b>Embedded Bitmap</b>	27, 'S', BMN
<b>Left justification</b>	27, 'L'
<b>Right justification</b>	27, 'R'
<b>Centre justification</b>	27, 'C'
<b>Htab</b>	27, 'H', PL, PH
<b>Vtab</b>	27, 'V', PL, PH

## Queue

**Queue**  $\Rightarrow$  Int

Queue returns the number of characters remaining in the PrintJob.

See also [Get](#)

## Status

**Status**  $\Rightarrow$  Int

When a large amount of text is printed to an object then the text will be split up into multiple print jobs.

It is useful to know if the print job is the first, last or intermediate part of the total text being sent.

Status will return an integer with bits set according to the table:

Bit	Value	Meaning
Bit 0	1	First print job in a Print To statement
Bit 1	2	Last print job in a Print To statement

### Examples

- If the text is contained in a single print job, then the Status value for this job will be 3.
- If the text is split up among two print jobs, the Status values for these two jobs will be 1 & 2.
- If the text is split up among four print jobs, the Status values for these will be 1, 0, 0, 2.

So an **AcceptPrintJob** method might look like this:

```
To AcceptPrintJob(pj)
  If pj.Status And 1 ; First PJ?
    name.Empty
    name.AcceptPrintJob(pj)
End
```

### PulseCounter

PulseCounter is used to count pulses using the STM32F103's internal timer hardware. It actually counts 'edges' - that is when the input signal changes from one state to the other. The pulse counter object can be set up to count positive-going, negative-going edges. It keeps track of the count as a 32-bit number.

#### Options available

The input may be pulled (up or down) or floating and hardware digital filtering may be applied to the input signal.

### Summary of messages

**Make**  
**Count**  
**Reset**  
**Print**

## Creation

```
Make <object> PulseCounter (Int chan, Int attributes [,
Int filter] )
```

A new PulseCounter object is created with a zero pulse count.

**chan** is the VM2 channel to use - it must be one of: \$10, \$16, \$18, \$26, \$36.

**attributes** sets up the pulse measuring input. It has the following commonly used values:

<b>attributes</b>	<b>Description</b>
0 or %0 <b>x</b>	Count on Falling edge
2 or %1 <b>x</b>	Count on Rising edge

(**x** = don't care)

In more detail, the binary bits in the **attributes** parameter have these significances:

	<b>Bit 2</b>	<b>Bit 1</b>	<b>Bit 0</b>
<b>When 1</b>	Floating input	Count on Rising edge	-
<b>When 0</b>	Input pulled to inactive state	Count on Falling edge	-

These are exactly the same [input attributes](#) as used for Digital.

The optional **filter** parameter may be used to set up digital filtering of the input signal.

[Read more...](#)

## Examples

```
Make p_in1 PulseCounter ($18,2)      ;channel $18, Rising edge.
Make p_in2 PulseCounter ($18,0)      ;channel $18, Falling edge.
```

▼ The maximum number of PulseCounter objects is 5

🔍 See also [PulseWidthOut](#), and [Shaft](#).

📏 Each PulseCounter object takes around 40 bytes.

## Count

**Count**  $\Rightarrow$  Int

Returns the current pulse count.

```
-->Print p . Count
3245-->
```

⚠ The PulseCounter object can keep track of over 2 billion pulses - i.e. it uses a 32-bit count register. When the count exceeds  $2^{31}-1$ , then the value wraps round to  $-2^{31}$ .

🔍 See also [Reset](#).

## Reset

**Reset**

Resets the pulse count to zero. For example:

```
-->Print p . Count,CR
3245
-->p . Reset
-->Print p . Count,CR
0
-->
```

🔍 See also [Count](#)

## Printing

**Print** <PulseCounter>

Prints the object's type and current pulse count in square brackets:

```
-->Print p
[PulseCounter: 10]
```

## PulseWidthIn

PulseWidthIn is used to measure the width or period of incoming pulses using the STM32F103's internal timer hardware. Pulses or periods from several  $\mu$ S to over 35 minutes can be measured with a resolution of 1  $\mu$ s, or smaller.

### Options available

Both rising and falling edges may be detected. The input may be pulled (up or down) or floating

and hardware digital filtering may be applied to the input signal.

The resolution of pulses measured is normally 1uS, but may be set to other values. See the [Speed](#) message.

## Summary of messages

**Make**  
**Done**  
**Go**  
**Period**  
**Print**

## Creation

```
Make <object> PulseWidthIn (Int chan, Int attributes [,
Int filter] )
```

A new PulseWidthIn object is created.

**chan** is the VM2 channel to use - it must be one of: \$10, \$16, \$18, \$26, \$36.

**attributes** sets up the pulse measuring input. It has the following commonly used values:

<b>attributes</b>	<b>Description</b>
0 or %00	Measure period: falling to falling edges
1 or %01	Measure low width: falling to rising edge
2 or %10	Measure high width: rising to falling edge
3 or %11	Measure period: rising to rising edges

In more detail, the binary bits in the **attributes** parameter have these significances:

	<b>Bit 2</b>	<b>Bit 1</b>	<b>Bit 0</b>
<b>When 1</b>	Floating input	1st edge Rising	2nd edge Rising
<b>When 0</b>	Input pulled to state before 1st edge	1st edge Falling	2nd edge Falling

These are exactly the same [input attributes](#) as used for Digital, apart from Bit 0.

*The optional **filter** parameter may be used to set up digital filtering of the input signal. See below*

### Examples

```
Make pw1 PulseWidthIn ($36, %10) ;VM2 channel $36, High pulse width
Make pw2 PulseWidthIn ($18, %11) ;Chan $18, Period between falling
```

▼ The maximum number of PulseWidthIn objects is 5.

📏 Each PulseWidthIn object takes around 40 bytes.

⌚ Because this object uses interrupts as part of the measurement, if the input pulse edges repeat too quickly then the measurement may become invalid. If you are using other interrupt-based objects concurrently then the minimum pulse width measurable will increase. Unfortunately it is difficult to be completely precise about the minimum measurable in any particular circumstances.

🔍 See also [Digital](#), [PulseWidthOut](#), [Shaft](#).

### Digital filtering

The STM32F103 has digital filtering hardware built into its timer modules.

The **filter** parameter allows you to set up the digital filter on the input signal so that glitches aren't confused with real input signals.

The digital filters work by recording the state of the input signal (high or low) for N samples, and only detecting a change of state when the last N samples all indicate a state change.

Both the sample rate and the length of the filter are selectable from a range of values.

Feature	Bits in the value	Description
Input filter setting	<b>Bits 3:0</b>	<p>This bit-field sets both the frequency and filter length used to validate the input signal. The frequency is defined in terms of either Ckint - the internal clock frequency of the internal TIMER module, or Fdf - the digital filter frequency defined below.</p> <pre> 0000: No filter, sampling is done 0001: Fsample=Ckint, N=2. 0010: Fsample=Ckint, N=4. 0011: Fsample=Ckint, N=8. 0100: Fsample=Fdf/2, N=6. 0101: Fsample=Fdf/2, N=8. 0110: Fsample=Fdf/4, N=6. 0111: Fsample=Fdf/4, N=8. 1000: Fsample=Fdf/8, N=6. </pre>



		1001: $F_{sample} = F_{df}/8$ , $N=8$ . 1010: $F_{sample} = F_{df}/16$ , $N=5$ . 1011: $F_{sample} = F_{df}/16$ , $N=6$ . 1100: $F_{sample} = F_{df}/16$ , $N=8$ . 1101: $F_{sample} = F_{df}/32$ , $N=5$ . 1110: $F_{sample} = F_{df}/32$ , $N=6$ . 1111: $F_{sample} = F_{df}/32$ , $N=8$ .
Digital filter frequency, $F_{df}$	<b>Bits 5:4</b>	This bit-field sets the digital filter sampling frequency, $F_{df}$ 00: $F_{df} = C_{kint}$ 01: $F_{df} = 1/2 \times C_{kint}$ 10: $F_{df} = 1/4 \times C_{kint}$ 11: <i>Do not use this value</i>

Notes: **Ckint** is the input clock frequency to the timer module used for each pulse channel. This is usually the same as the CPU speed (normally 72MHz), unless the internal clock tree has been altered outside the control of Venom2.

## Done

**Done**  $\Rightarrow$  Int

Returns **True** (1) when the measurement cycle is complete, i.e. both the leading and trailing edge have been seen. Returns **False** (0) if this condition has not yet been met.

## Go

**Go**

Starts a measurement cycle. The time measurement will start at the next leading edge as specified by mode when the object was created, and the measurement is complete as soon as the correct type of trailing edge is encountered.

## Period

**Period**  $\Rightarrow$  Int

Returns the measured period. The unit and resolution of pulses measured is normally **1uS**, but may be set to other values. See the [Speed](#) message.

The Period message can be used in two ways:

1. By itself, the Period message will initiate a measurement cycle and wait for the result before returning. The current task is blocked while waiting.

2. In conjunction with the Go and Done messages, the task can loop and execute other code while waiting. When the Done message returns true, the next Period message will return immediately with the value just measured. If a Period message is sent any time after a Go message and before the measurement cycle is complete, the task will wait.

Example of 1st usage

```
pwwidth := pw.Period    ; task is suspended while waiting
```

Example of 2nd usage

```
pw.Go
While pw.Done IsFalse
[    ; other code in loop executed while waiting
]
pwwidth := pw.Period
```

Or, this uses less power when there is nothing particular to be done in the current task while waiting:

```
pw.Go
Await pw.Done
pwwidth := pw.Period
```

▼ See [here](#) for minimum period limitation. The maximum period measurable is around 2 billion units - which is around 35 minutes if the units are 1uS.

## Speed

**Speed** ⇔ Int

This allows you to set the clock 'prescaler' for the timer module which implements the PulseWidthIn object.

The clock prescaler divides the system clock (normally 72 MHz) by an integer value before passing it on to the timer module.

Thus when Speed is set to 72 (the default value), the timer is clocked at 1MHz, and so the resolution of the PulseWidthIn is 1uS.

▼ Speed may be set in the range 1 - 65356, though numbers much higher than 72 are unlikely to be useful. Speed does not change the limitation due to [speed of interrupts](#).

## PRINT

**Print** <PulseWidthIn>

Prints, inside square brackets, the text 'PulseWidthIn : ' followed by the last measured value, followed by a new line.

No measurement cycle is initiated. If no measurement has been made since the object was created the value printed is 0.

The Print message is not recommended for general programming with PulseWidthIn objects.

Example:

```
-->print pw  
[PulseWidthIn: 456]  
-->
```

## PulseWidthOut

PulseWidthOut generates pulse trains with variable Mark/Space ratio and frequency using the STM32F103's internal timer hardware.

### Options available

A PulseWidthOut object may be set to generate a continuous signal, or a pulse train containing an exact number of pulses.

The output may be active high or active low, and may be 'push-pull' or 'open drain'.

The resolution of pulses generated is normally 1uS, but may be set to other values. See the [Speed](#) message.

Some channels support a dual output, where there are two waveforms on different channels that have the same period, but different widths. This gives up to eight pulse outputs in total on a VM2.

## Summary of messages

**Make**  
**Asserted**  
**Count**  
**Off**  
**On, Go**  
**Period**  
**Queue**  
**Speed**  
**Width**  
**Print**  
**Die**

## Creation

```
Make <object> PulseWidthOut (Int channel, Int period,  

Int width, Int attributes [, Int count [, Int  

attributes2]])
```

A new PulseWidthOut Object is created. Note that the pulse train will not actually start until the Go or On message is sent.

The table below describes each of the parameters.

Parameter	Range of values	Purpose
<b>channel</b>	\$10, \$16, \$18, \$26, \$36	The VM2 Channel to use for the pulse output
<b>period</b>	2 to 65,536	The period of the output signal (in uS unless Speed is changed)
<b>width</b>	0 to 65,536	The width of the output signal (in uS unless Speed is changed). If <b>width</b> is 0 then the signal goes to the OFF state; if <b>width</b> >= <b>period</b> then the signal goes to the ON state.
<b>attributes</b>	<i>Simple options:</i> 'Off' is Low, Pulse High: <b>3</b> 'Off' is High, Pulse Low: <b>1</b>	These are exactly the same <a href="#">output attributes</a> as used for Digital, apart from Bit 0, which is 'don't care', but set to 1 here for consistency. This includes the output edge 'speed' attributes.
<b>count</b>	0 – 2,147,483,647	<i>Optional:</i> The number of pulses to generate. If this parameter is not present, or has a value

		of 0 then the signal is continuous.
<b>attributes2</b>	<i>Simple options:</i> 'Off' is Low, Pulse High: <b>3</b> 'Off' is High, Pulse Low: <b>1</b>	<i>Optional:</i> Dual outputs are possible on these channel pairs: (\$10, \$11); (\$16, \$17) and (\$36, \$37). To specify a dual output, set <b>channel</b> to the first channel in the pair, and set the <b>attributes2</b> parameter for the attributes of the second channel (it must be non-zero). See <a href="#">Width</a> for how to control the second channel.




### Example

```

;Create a 1:10 mark:space PWM waveform
Make pwm pulsewidthout($18,10000,1000,1)
pwm . On ;turn it on;Create a 1:10 mark:space PWM waveform

;Create PWM outputs on two channels (same period, initially same
Make pwm pulsewidthout($18,10000,1000,1, 0, 1)
pwm . On ;turn them both on

```

	<i>Warning: When generating a defined-length pulse train using count, PulseWidthOut uses interrupts to count the pulses. If the period becomes too small the VM2 may start to behave erratically. The exact threshold for this depends on interrupt loading. Unfortunately it is difficult to be completely precise about the minimum pulse width attainable in any particular circumstances.</i>
	See also <a href="#">PulseWidthIn</a> , <a href="#">PulseCounter</a> , <a href="#">Shaft</a> .
	Each PulseWidthOut object takes around 40 bytes.

### Asserted

**Asserted**  $\Leftrightarrow$  Int

Setting Asserted True starts a pulse train (like On), and setting it False stops the train (like Off). Asserted returns True if the pulse train is currently active, False otherwise.

 [On](#), [Off](#)


## Count

**Count**  $\Leftrightarrow$  Int

**Count** allows you to specify that the PulseWidthOut object generates a pre-determined number of pulses in a pulse train.

Setting the **Count** takes effect the next time the train is triggered (using **On**). It has no effect on a currently active pulse train.


Reading the **Count** returns the number last set during creation or using **Count**. Use [Queue](#) if you want to find the number of pulses remaining to be sent.

 See also [Creation](#)

## Off

**Off**

Off turns off the pulse output immediately. It leaves the Width and Period settings untouched, so that On will turn on the pulse output with the same pulse timings as before.

 See also [On/Go](#)


## On, Go

**On**  
**Go**

**On** turns on the pulse output. The pulse output will not start until the **On** message has been sent. Every time **On** is sent, the number of pulses in a numbered pulse train is reset to the value of **Count**. If this was zero, then the pulse train keeps going indefinitely.

**Go** is an alias for **On**.

 See also [Off](#) and [Count](#)

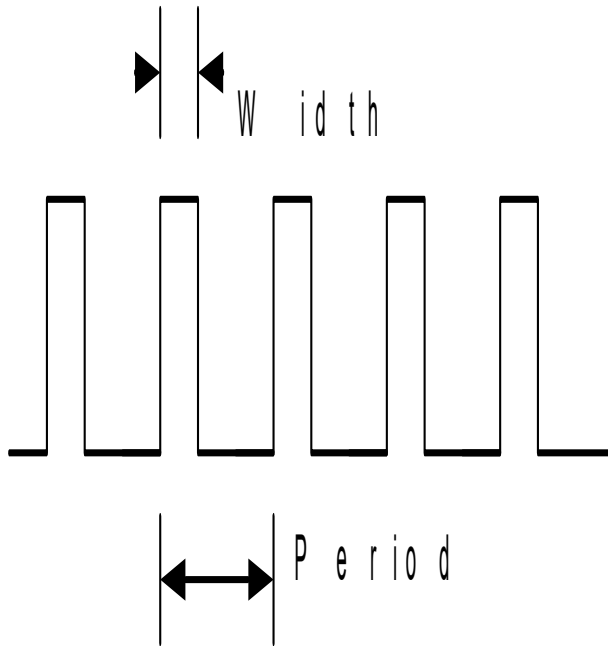
 There is a bug in the Silicon of the VM2 processor IC that leads to this behaviour: if Width >= Period, the On message will not turn the output on.

## Period

**Period**  $\Leftrightarrow$  Int

An active variable that allows the overall period of the waveform to be read or set. The unit and resolution is normally **1uS**, but may be set to other values. See the [Speed](#) message.

The diagram below illustrates the relationship between period and width for an active-high waveform.



The minimum Period value is 2, but see the warning notice in [Creation](#).

If Period is set to less than or equal to the current Width then the output will go to the 100% duty state. As soon as Period is greater than Width again, pulses will reappear.


If you have Count set when Period is less than Width, (or a Width of 0) counting of pulses will continue even if no real pulses are generated.

 See also [Creation](#), [Count](#).

## Queue

**Queue**  $\Rightarrow$  Int

If the pulse train is active and count has been set, **Queue** returns the number of pulses still to be generated. If the pulse train is inactive or the pulse train is continuous, or no count has been set it returns 0.

 See also [Count](#)

## Speed

**Speed**  $\Leftrightarrow$  Int

This allows you to set the clock 'prescaler' for the timer module which implements the PulseWidthOut object.

The clock prescaler divides the system clock (normally 72 MHz) by an integer value before passing it on to the timer module.

Thus when Speed is set to 72 (the default value), the timer is clocked at 1MHz, and so the resolution of the PulseWidthOut is 1uS.

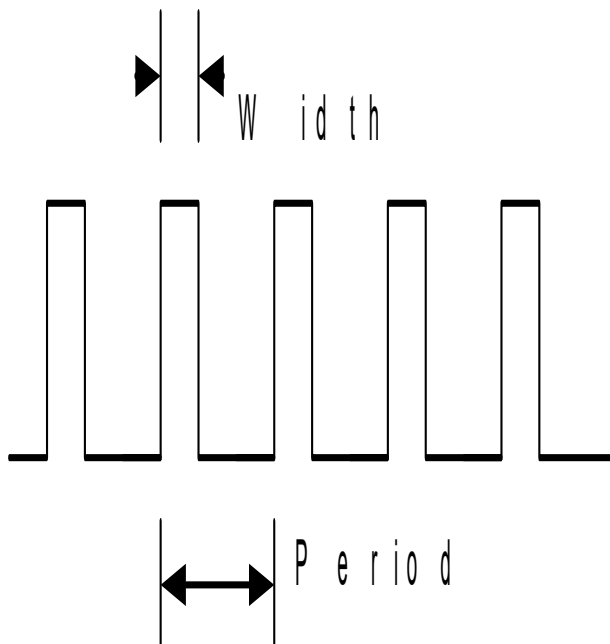
⚠ Speed may be set in the range 1 - 65536. Speed does not change the limitation due to [speed of interrupts](#).

## Width

**Width**  $\Leftrightarrow$  Int

Allows the width, or 'mark' period of the waveform to be read or set. The unit and resolution is normally **1uS**, but may be set to other values. See the [Speed](#) message.

The diagram below illustrates the relationship between period and width for an active-high waveform.



If Width is set to zero or less, then the output will go to the 0% duty state, i.e. pulses will not be



generated.

If Width is set greater than or equal to the current Period then the output will go to the 100% duty state so the output is turned on continuously, no pulses will be generated. As soon as Width is set less than Period, pulses will reappear.

If you have Count set when Width is zero, or Width is  $\geq$  Period, the count will continue even if you can't see pulses.

 See also [Creation](#), [Period](#)

## Dual outputs

```
obj . Width(Int index) ⇔ Int
```

If the object has [two outputs](#) associated with it, each has an independent width setting. Using Width without an index parameter will access the main output. However you can use Width with an index to access either output. The two outputs are numbered:

0. The main output
1. The auxiliary output

## Printing

```
Print <PulseWidthOut>
```

Prints the object type and current width and period within square brackets.

```
-->Print pwm  
[PulseWidthOut: 1000/10000]
```

 See also [Period](#), [Width](#)

## Die

Die turns off the pulse train immediately and removes the object.

## RandomNumberGen

The Random number generator object uses a triple Tausworthe 32 bit generator whose results pass many stringent tests of randomness.

The sequence length is  $2^{88}$ .

It can be used to generate integer or floating random values.

 The integer generator takes about 12 microseconds per new value.

## Summary of messages

**Make**  
**Get**  
**Print**  
**Reset**  
**Value**

## Creation

**Make** <object> **RandomNumberGen**(**type**, [various **seeds**])

type        **Int** or **Float**

Seeds        If no seed value is specified, an initial state for the generator is created from system information in a way that is designed to be unrepeatable and unpredictable.  
  
Otherwise 1-3 integers or a string or text buffer contents define a starting pattern which is repeatable if the same seed value is used each time.

Each random number generator object stores its own state.

## Examples

```
Make r1 RandomNumberGen(Int)  
Make r2 RandomNumberGen(Float)
```

## Get

```
Get ⇒ Int or Float  
Get([Int min, Int max]) ⇒ Int  
Get([Float min, Float max]) ⇒ Float
```

Returns a value of the type defined when the generator was created.

Integer values have a uniform distribution in the range -2147483648 to +2147483647.

Floating point values have a uniform distribution between 0.0 and +1.0

If min and max values are specified, the generator will return numbers in that range, including the min and max values.

## PRINT

The Print operator on a random number generator produces output like this:

```
[rand:0.652303]      (float)
[rand:-1553008633]   (int)
```

## Reset

### Reset

**Reset** (Int or Str **seed**)

**Reset** (Int **seed1** [, Int **seed2** [, Int **seed3**]])

If no seed value is specified, an initial state for the generator is created from system information in a way that is designed to be unrepeatable and unpredictable.

Otherwise 1-3 integers or a string or text buffer contents define a starting pattern which is repeatable if the same seed value is used each time.

## Value

Value is a synonym for [Get](#) for all types of random number generators.

## RealTimeClock

The RealTimeClock object uses the VM2's built in RTC module to keep an accurate count of time in seconds. This number may be converted to a meaningful date by [printing](#) the clock or by using a [DateTime](#) object.

RealTimeClock can keep track of the date and time until the year 2089, and it can be adjusted to compensate for inaccuracies in the crystal oscillator.

When the controller is powered down the real-time clock is kept powered by the Lithium backup battery.

## Summary of messages

**Make**  
**Adjust**  
**Element**  
**Line**  
**Reset**  
**Time**  
**Timeout**  
**Valid**  
**Print To**  
**Print**

## Creation

**Make** <object> **RealTimeClock**

The clock object is created without affecting the time it holds internally.

If the real-time clock has not had its time set previously then this will need doing before it can be used properly: see [Time](#).

▼ The clock can hold times from 1990 to 2089

## Adjust

**Adjust** ⇔ Int

Adjust allows you to correct the clock if it runs fast or slow.

The units of adjustment are approximately equal to ppm, but are actually 1 part in  $2^{20}$ , or ~0.9537ppm.

The adjustment value is in the range 0-127. An adjustment value of 64 will make the clock run at the exact speed set by the clock crystal circuit. A lower value will make the clock run faster, and a higher value will make the clock run slower. The maximum adjustment range is about +/- 5.3 seconds per day, and the precision of adjustment is ~0.082 seconds per day.

The Adjust value is initialised to the midpoint value (~64) the first time you set Time.

 [GetLast](#)

## Example code

*; Apply a correction, in Seconds, to the clock's time.*

```

; Correction is +ve to set the clock forward, and -ve to set it b
; Also, adjust the clock's speed to take account of the applied c
To ApplyClockCorrection(correction)
  Local elapsedTime
  Local lastSet := clock.GetLast ; Read when the clock was last s
  elapsedTime := clock.Time - lastSet
  clock.Time := clock.Time + correction ; set the clock to the ne
  speedUpClockBy(correction/elapsedTime) ; Adjust the clock. Spe
End

; Adjust the speed of the clock.
; adjustment is a float value.
; +ve values make the clock go faster.
To speedUpClockBy(adjusment)
  Local NewAdjust
  Local AdjustValue := (adjusment * (2 ^ 20)) As Int ; Convert t

  ; Find the new Adjust value.
  ; [Increasing the Adjust value slows the clock down,
  ; so we have to subtract from the existing value].
  NewAdjust := clock.Adjust - AdjustValue

  ; Limit the range to valid values:
  If NewAdjust > 127
    NewAdjust := 127
  Else If NewAdjust < 0
    NewAdjust := 0
  ;Set the new adjustment
  clock.Adjust := NewAdjust
End

```

## GetLast

**GetLast** ⇒ Int

GetLast returns the date/time (in Venom Seconds) at which the clock was last set.

## Element

**Element**(Int **index**)  $\Leftrightarrow$  Int

<RealTimeClock>.(Int **index**)  $\Leftrightarrow$  Int

The Element message allows access to the STM32F103's 'backup registers'. These are 42 registers, each of 16 bits, that hold their contents so long as the clock battery is present.

They may be used to hold non-volatile parameters and settings for your application.



**index** can take the values in the range 1 - 42, but see the note below.



NOTE: registers 1 and 2 are used by the RealTimeClock object to hold the [GetLast](#) value. If further registers are used by the system they will be taken from the front of the register block.

## Line

**Line**(Int **channel**) := Int **edge\_flags**

The Line message allows you to set up one or more input channels so that they wake the VM2 when it is in [Stop Mode](#).

Before using Line you should Make the channel into a Digital input of some kind.

Any VM2 channel may be used to wake from Stop Mode, but not all at the same time.

The limitation is given by the secondary channel number grouping: channel numbers are expressed as two-digit hexadecimal numbers, for example **\$2C**. The secondary grouping is given by the second hexadecimal digit of the channel number. Thus channel **\$2C** is part of secondary group **C**.

You may use one channel from each secondary group. I.e. *none of the channels you use to wake up can have the same second digit as any other.*

This is further restricted in that the CTS input signals used by serial ports 2,3,4 & 5 use the same mechanism as for wake up - so you should include them in your calculations of which channels may be used.

The channels use by the SerialPort CTS inputs are given in the table

Serial Port	VM2 Channel	Channel group
-------------	-------------	---------------

	No.	
1	\$73	<i>None</i>
2	\$43	<b>_3</b>
3	\$75	<b>_5</b>
4	\$77	<b>_7</b>
5	\$78	<b>_8</b>

The value **edge\_flags** is used as a set of binary flags. Bit 0 is set if you want to detect falling edges and bit 1 set for detecting rising edges. Set both bits for both edges.

### Example

Say we are using serial ports 1, 2 & 4 with hardware handshaking. The other serial ports we aren't using at all, or at least we not using hardware handshaking.

Thus these channels are used up:

- CTS2, or channel \$43
- CTS4, or channel \$77

This means the groups **\_3** and **\_7** are in use, leaving groups **\_0**, **\_1**, **\_2**, **\_4**, **\_5**, **\_6**, **\_8**, **\_9**, **\_A**, **\_B**, **\_C**, **\_D**, **\_E** & **\_F** free to use.

So we might choose to use channels **\$10**, **\$2C**, **\$34** as groups **\_0**, **\_C** and **\_4** are free.

### Example code

```

Make wakeup_input1 Digital($10) ; make an input
Make wakeup_input1 Digital($2C) ; make an input
Make wakeup_input1 Digital($34) ; make an input
...
clock.Line($10) := %10 ; Wake on Rising edge
clock.Line($2C) := %01 ; Wake on Falling edge
clock.Line($34) := %11 ; Wake on Either edge
...
Forever
[
    clock.Timeout(10) ; Go into Stop Mode for ~10 Seconds.
    check_wake_lines ; code to check the inputs to see if one of th
]

```

## Reset

### Reset

Reset will reset the RTC hardware module inside the VM2 microcontroller, stopping the 32768 Hz oscillator. Valid will subsequently return False.

This might be used to reduce battery drain, or to simulate the behaviour of a new system where the clock has never been set.

## Time

**Time**  $\Leftrightarrow$  Int

Time is an active variable that holds the seconds count of the clock. This may be converted to a date using a [DateTime](#) object, or by [printing](#) the clock. If the time has not been set since the Lithium battery was plugged in, then the time returned will be zero.


### Startup time

It can take around three seconds to first start up the clock's crystal oscillator: you may notice this as a delay when setting Time for the first time.

If the clock crystal oscillator doesn't start up then a Device not found error is generated.

You can shut down the clock oscillator using [Reset](#).

You can set the clock's time by writing a seconds value to it (perhaps using a [DateTime](#) object to calculate that value), or by [printing to](#) it.

 Generally the clock will keep time to a few seconds a day. If the controller is used near the temperature extremes then this accuracy will be affected. You can calibrate the clock using [Adjust](#). Adjust is initialised to its mid point (64) the first time you set Time.

 See also [Reset](#), [Valid](#)

## Timeout

**Timeout**(Int `sleep_period`)

The **Timeout** message puts the VM2 into the STM32's low power 'Stop Mode' for a period of `sleep_period` Seconds.



During Stop Mode the VM2 takes very little power (~55uA). All internal clocks other than the Real Time Clock are stopped, the I/O state is retained, and the contents of memory are retained. This means the outputs remain at exactly the state they were on entry into Stop Mode; similarly inputs remain as inputs. When Stop Mode ends the VM2 carries on from where it left off. The internal time used by Wait, Stopwatch, Timer and other subsystems is frozen during Stop Mode.

The maximum sleep period is 15 seconds, however it is simple to program around this limit - see the example code below.

▼ **sleep\_period** is in the range 2 to 15 seconds, though it is easy to extend the effective sleep time. The upper limit is given by STM32F103 watchdog timer; the lower limit is set to ensure the real time clock alarm is never set so early it might be missed. If you supply a period outside these limits it will be constrained.

### Minimising power consumption

It may be useful for some VM2 applications to spend much of their time in Stop Mode, consuming little current, and waking up only when necessary.

In order to attain the minimum overall current during Stop Mode it will be necessary to do most or all of the following things:

- Turn off all circuits external to the VM2 that take significant current
- Turn off all outputs on the VM2 that may be sinking or sourcing current into an external device, including the onboard LED.
- Make sure no inputs are floating. All inputs should be driven to a valid logic state - either high or low. See [system.Low](#).
- Make sure that 'pulled' inputs are not being driven against their pull up/down resistance.
- Turn off the ADC and DAC modules within the VM2 (method to be implemented later)
- Wait until there is no traffic on the Serial Ports, I2C Bus, etc - else characters may be missed or the bus held in a high current state.

### Other wake mechanisms

It is possible to wake up on the change of state of one or more input channels. See [Line](#).

### Example code

```
To main
Print clock,CR
wait 5 ; to let the serial characters out before we sleep.
sleep_until(clock.Time + 60)
Print "We have woken up!", clock, CR
```

End

```

; (This code has a 2S resolution.)
To sleep_until(wake_time)
  Local sleep_period
  Forever
  [
    sleep_period := wake_time - clock.Time
    If (sleep_period) <= 0
      Break
    clock.TimeOut(sleep_period) ; Low power sleep 2-15 seconds.
  ]
End

```

## Valid

**Valid** ⇒ Int

Valid returns True if the clock holds a valid time - i.e. it has been set and the clock hardware module seems to be working.

## Accepting Print

The clock's time and date can be set using Print To. E.g.:

```
Print To clock, "2010-06-15 10:56:00"
```

Printing to the clock must obey these rules:


- ISO format is used: **YYYY-MM-DD HH:mm:ss**
- You have to provide all the elements of the date, but you can miss out the least significant time elements, e.g. **YYYY-MM-DD**, or **YYYY-MM-DD HH:MM**
- You can use any single non-numeric characters to separate the elements of the data and time, e.g. **YYYY/MM\*DD HH-mm SS**

For example:

```

Print To clock, "2010-06-15 10:56:00" ; ISO format
Print To clock, "2010 06 15 10 56 00" ; Using spaces to separate t

```

 Note: you can also print to an object by sending it the [PrintF](#) message.

## Printing

```
Print <RealTimeClock>
```

Printing the RealTimeClock object shows the current time and date in ISO format.

```
-->Print clock  
2012-04-11 15:03:55
```

## Formatting

You can also specify the format you want to use by using a format specifier string.

```
Print <RealTimeClock> : format_string
```

For example:

```
-->Print clock : "h:mmaa, dd MMM yyyy",cr  
3:03pm, 11 Apr 2012
```

This uses the same system that is used for printing the [DateTime](#) object.

(The old system that uses a numeric format specifier is also still available - see Printing [DateTime](#) )

## DST, BST and Time zones

If you use a second (integer) format parameter to **Print** for the **RealTimeClock** then this will be used to offset the printed date/time. This is useful for implementing systems that can display daylight saving time and different time zones. Using this method the clock's time is not changed when moving in and out of DST, but the displayed time is changed by supplying a different offset.

This line prints the clock's time, but offset by one hour:

```
Print clock: "HH:mm:ss":3600 , cr
```

## Locale

It is possible to change the day of week and month names to suit different locales. See [here](#) for more information.

## SafeData

SafeData allows easy storage of important non-volatile data - e.g. device settings and calibration data - in an external I2C EEPROM.

 You might also want to use [RealTimeClock.Element](#) to store non-volatile data.

## Usage notes

[Put](#) and [Get](#) are probably the best access messages to use for most applications as these allow the simplest interface to store and retrieve lists of values of different types and sizes. [Element](#) is also provided for random access into the SafeData store. There is some example code [here](#).

## Summary of messages

**Make**  
**Address**  
**Checksum**  
**Element**  
**Get**  
**Put**  
**Reset**  
**Print**

## Creation

**Make** <object> **SafeData** (Int **type** , Int **bus** , Int **address**)

**type** gives the type of non-volatile storage device: see the table below.

**bus** gives the I2C Bus the device is attached to: either 1 or 2.

**address** gives the I2C Address of the device: usually 160, 162, ... and upwards.

There are two I2C EEPROM device types supported:

Type	Description
1	EEPROMs compatible with the M24C02. The significant common feature is they take one byte of address within the device (maximum storage 256 bytes).
2	EEPROMs compatible with the M24C32, M24C64, etc. The significant common feature is they take two bytes of address within the device.

The bus and address parameters specify which I2C bus the EEPROM is on (1 or 2), and which address it is located at on the bus (160, 162, ...). For example:

**Make s SafeData (1 , 1 , 162)**

 SafeData takes an small block from the heap to keep local information

## Address

**Address**(Int **p**)  $\Leftrightarrow$  Int

Address allows you to set and read the address within the storage device that the Get message reads from and the Put message writes to. This address is auto-incremented on Put and Get.

## Checksum

**Checksum**(Int **start** , Int **end**)  $\Rightarrow$  Int

Checksum returns the sum of all the bytes in the storage device from start to end-1, as a 32-bit unsigned integer. It can be useful for validating the contents of the device.

*Note: you don't have to use all 32 bits of the returned value of the sum.*

## Example code: using user-defined class as a record

This code uses a user-defined class to hold all the settings, and uses a checksum to validate them.

```

To init
    ;...
    Make eeprom SafeData(1,1,162)
    Make Settings SettingsClass
    SettingsChanged := False ; Set this flag whenever a setting has
    ;...
End

To main
    LoadPersistentSettings
    ;...
End

; Load the settings from EEPROM. Reset settings if the checksum is
To LoadPersistentSettings
    If ReadSettings IsFalse
        ResetSettings
    End

; Save the settings in the EEPROM if SettingsChanged flag is set.

```

```

; Call this at points in the code when you want the current setti
; It only writes to the EEPROM when the settingsChanged flag has
To SaveSettings
    If SettingsChanged
        WriteSettings
End

; A Class to hold persistent settings.
Class SettingsClass
    Calibration Float
    Delay Int 8
    Count Int 8
End

#define XSUM_EXTRA 42 ; Extra value added to checksum.

; Store settings in the EEPROM, and set a checksum.
To WriteSettings
    Local xsum
    eeprom.Reset
    eeprom.Put(Settings)
    ; calc and write checksum...
    xsum := eeprom.Checksum(0, eeprom.Address) + XSUM_EXTRA
    eeprom.Put(xsum, Int 16)

    SettingsChanged := False
End

; Read settings from the EEPROM and return true if checksum is go
To ReadSettings
    Local xsum, xsum2
    eeprom.Reset
    eeprom.Get(Settings)
    xsum := eeprom.Checksum(0, eeprom.Address) + XSUM_EXTRA
    xsum2 := eeprom.Get(Int 16)

    SettingsChanged := False

    Return (xsum = xsum2) ; return true if xsums agree.
End

; Reset the settings to their default values.
To ResetSettings
    Settings.Delay := 10

```

```
Settings.Calibration := 100.0
Settings.Count := 0

SettingsChanged := True
End
```

### Example: storing separate items

Here we show how to store the values of a set of variables (named a, b, c, d and e), set their checksum, retrieve them from storage, and check their validity.

A significant simplifying feature of this code is that the values are always stored and retrieved in the same order, so that even though they are stored using different numbers of bytes, your code doesn't have to know the exact addresses they are stored at.

```
#Define XSUM_CONST 42 ; any non-zero constant value.

To init
  Make sd SafeData (1 , 1 , 162)
End

To main
  load_settings
  ... ; make some changes to the variables
  store_settings
End

To Set_default_values
  ;Some dummy data to test with:
  a := 1
  b := 2
  c := 3
  d := 4
  e := 5.12345
End

;Store the data using the SafeData object.
To store_settings
  Local posn

  sd.Address := 2 ; [First two bytes are reserved to hold size of
  sd.Put(a)
  sd.Put(b,16)
  sd.Put(c,16)
  sd.Put(d,32)
```

```

    sd.Put(e, 1.0)
    posn := sd.Address ; the next empty position.
    sd.Element(0, 16) := posn ; record (at the start) how much data
    sd.Put(sd.Checksum(0,posn)+XSUM_CONST,16) ; Store the checksum
End

;return True if the stored data matches it's checksum.
To stored_data_ok
    Local posn := sd.Element(0,16) ; read how big our stored data is
    Return sd.Element(posn, 16) = sd.Checksum(0,posn) + XSUM_CONST
End

To load_settings
    If stored_data_ok IsFalse
    [
        Print "Settings lost!", BEEP, CR
        Set_default_values
        store_settings
    ]
    Else ; The stored data looks OK...
    [
        sd.Address := 2 ; This is where the actual data starts.
        a := sd.Get
        b := sd.Get(16)
        c := sd.Get(16)
        d := sd.Get(32)
        e := sd.Get(1.0)
    ]
End

```

A constant value, called here **XSUM\_CONST**, is used to get round the condition that if the data is all zeros, then the checksum will match even if all zeros is not correct data. Use any non-zero integer value, such as '42'.

## Storing strings

Variable-length Strings may be stored in the SafeData similarly to the variables shown above, by writing all the characters of the string into the SafeData. You should either store the length of the string before the data, or store a zero byte to indicate the end of the string.



## Element

```
Element(Int addr , type) ⇔ Int/Float
<SafeData>.(Int addr , type) ⇔ Int/Float
```

The element message allows 'random' read and write access to the storage device. The parameters to Element are the byte address within the device, and an optional parameter that tells the object what type of data is being stored.

type	Data storage
None	8-bit unsigned integers
<b>Int 8</b>	8-bit unsigned integers
<b>Int 16</b>	16-bit signed integers
<b>Int 24</b>	24-bit signed integers
<b>Int</b> or <b>Int 32</b>	32-bit signed integers
<b>Float</b>	Floating point numbers

The **.Element** message has a standard shortcut in Venom2 which is **.()**

For example:

```
s.(0, Int 8) := 24
```

Writes 24 to one byte at location zero.

```
s.(2,Float) := 231.3438
```

Writes a floating-point number to four bytes at location two

You should be careful to take account of the size of the data you are writing when you choose the start address, so you don't get overlaps or holes in the device's memory space.

```
s.(0,Int 8) := 24
s.(1,Int 16) := 1232
s.(3,Int 16) := 23232
s.(5,Int) := 323323222
s.(9,Float) := 1.23232
```

Reading data out of the device is very similar to writing it.

```
Val := s.(0, Int 8)
```

Reads a byte out of the device from location zero.

```
s.(2, Float) := 231.3438
```

Writes a floating-point number to four bytes at location two.

*There is currently no validation of data written to the storage device so you may want to check the correct data has been written.*

## Get

**Get**  $\Rightarrow$  Int

**Get(type)**  $\Rightarrow$  Int

Get reads an item of data, at the current [address](#), from the storage device, and returns it's value.

The address is advanced by the number of bytes in the item read.

The optional **proto** parameter sets the kind of item to read:

type	Data item read from device	Address increment
<i>None</i>	8-bit unsigned integer	1
<b>Int 8</b>	8-bit unsigned integer	1
<b>Int 16</b>	16-bit unsigned integer	2
<b>Int 24</b>	24-bit unsigned integer	3
<b>Int</b> or <b>Int 32</b>	32-bit signed integer	4
<b>Float</b>	Floating point number	4
A user-defined Class object*	A <i>record</i> as defined by a <a href="#">class definition</a>	<a href="#">Length</a>

*\*Note: when supplying a user-defined class object as the 'type' parameter, data is read from the storage device into the user-defined class object. The Get message return value is undefined.*

## Examples

```

b := s.Get           ; read byte
n := s.Get(Int 16) ; read a 16-bit integer
n := s.Get(Int)   ; read a 32-bit integer
f := s.Get(Float) ; read a floating point value
s.Get(MyClassObject) ; read data into MyClassObject

```

More example code is shown in [Checksum](#).

## Put

**Put**(Int **item**, [**type**])

Put writes a data item to the storage device, at the current [address](#). The address is advanced by the number of bytes in the item written.

The type parameter indicates the type and size of the item to write

type	Data item written to device	Address increment
<i>Not supplied</i>	8-bit unsigned integer	1
<b>Int 8</b>	8-bit unsigned integer	1
<b>Int 16</b>	16-bit unsigned integer	2
<b>Int 24</b>	24-bit unsigned integer	3
<b>Int</b> or <b>Int 32</b>	32-bit signed integer	4
<b>Float</b>	Floating point number	4
A user-defined Class object*	A <i>record</i> as defined by a <a href="#">class definition</a>	<a href="#">Length</a>

*\*Note: when writing a user-defined class object, only one parameter should be supplied: the object.*

*There is currently no validation of data written to the storage device so you may want to check the correct data has been written.*

## Examples

```
s.Put(2) ; Write a byte
s.Put(3.1234, Float) ; Write a float value
s.Put(1232, Int 16) ; write a 16-bit integer value.
s.Put(12388734, Int) ; write a 32-bit integer value.
```

More example code is shown in [Checksum](#).

## Reset

**Reset**

Reset resets the internal [address](#) for reading and writing to 0.

## PRINT

Printing the SafeData object prints some or all of the contents of the storage device as a list of bytes in hex format, each line up to 16 bytes long.

If you supply colon format parameters, then you can specify the amount of data printed, and the start position of the data to print. One colon parameter specifies the amount of data to print from the start; two colon parameters specify the start point and then the amount of data to print.


If no colon parameters are present, 256 bytes from the start of the device are printed.

### Example

```
Print s:16:32
```

```
SafeData
```

```
$01 $02 $03 $04 $04 $04 $04 $04 $04 $04 $00 $00 $42 $C8 $76 $C9
$41 $41 $41 $41 $41 $41 $41 $41 $41 $41 $41 $41 $41 $41 $41
-->
```

 [Print formatting](#), in general; [Print formatting for Objects](#).

## Semaphore

The Semaphore object in Venom2 has two closely related uses.

Firstly, it is an object that may be *locked* - and as such may be used to control access to critical parts of your code in a multitasking application, as described in The Venom Tutorial. This use of the Semaphore object uses the **Lock**, **Unlock**, **TestLock** and **Owner** messages.

Secondly, it also implements the *classic semaphore* function, enabling more sophisticated resource control systems to be implemented.

This functionality uses the **Get** and **Put** messages, to claim and replace resources.

The semaphore holds an internal *count* value, which is usually initialised when the semaphore is created to a positive value (although a value of zero can be useful too).

The amount of a particular resource in your application can be represented by the Semaphore's count value.

Tasks may claim integer amounts of resource using the **Get (n)** message, and relinquish that resource using the **Put (n)** message.

**Get** waits, swapping tasks, until the Semaphore's count is at least n, then takes n from count (thus n always remains zero or positive).

**Put** simply adds n to the count value.

## Summary of messages

**Make**  
**Count**  
**Get**  
**Put**  
**Lock**  
**Owner**  
**TestLock**  
**Unlock**  
**Print**

## Creation

**Make** <object> **Semaphore** [ (Int **initial\_count**) ]

**initial\_count** is optional - it sets the initial value for the count held by the Semaphore.

## Examples

```
Make s Semaphore  
Make s1 Semaphore(10) ; make one with an initial count of 10
```

## Count

**Count**  $\Leftrightarrow$  Int

Reading the value of **Count** returns the count value held by the Semaphore.

**Count** can also be set, but this should normally only be done as a 'reset' operation, not as part of claiming and relinquishing of resources.

## Get

**Get**  $\Rightarrow$  Int  
**Get**(Int **n**)  $\Rightarrow$  Int

**Get** waits, swapping tasks, until the Semaphore's count is at least **n**, then removes **n** from the count.

If **n** is not supplied then **n** defaults to 1.

## Put

**Put**

**Put** (Int n)

Put adds n to Semaphore's count.

If n is not supplied then n defaults to 1.

## Lock

**Lock**  $\Rightarrow$  Int

**Lock** (Int n)  $\Rightarrow$  Int

Locks the Semaphore object.

This message accesses the Semaphore's standard 'Venom resource lock' properties, which are completely unrelated to its other, classic semaphore properties.

See [here](#) for details of the locking messages.

## Owner

**Owner**  $\Rightarrow$  TaskObject Or Nil

Returns the owner of the Semaphore object.

This message accesses the Semaphore's standard 'Venom resource lock' properties, which are completely unrelated to its other, classic semaphore properties.

See [here](#) for details of the locking messages.

## TestLock

**TestLock**  $\Rightarrow$  Int

Locks the Semaphore object, and returns the result.

This message accesses the Semaphore's standard 'Venom resource lock' properties, which are completely unrelated to its other, classic semaphore properties.

See [here](#) for details of the locking messages.

## Unlock

**Unlock**

Unlocks the Semaphore object.

This message accesses the Semaphore's standard 'Venom resource lock' properties, which are completely unrelated to its other, classic semaphore properties.

See [here](#) for details of the locking messages.

## PRINT

**Print** <Semaphore>

PRINTing the Semaphore will print something like

[Semaphore 5]

which indicates you are printing a Semaphore object that holds the value 5 in its internal count.

## SerialPort

SerialPort objects provide interfaces to serial communication ports. The ports can operate at standard rates up to 115,200 baud, or higher non-standard rates. There is configurable handshaking available: hardware (RTS/CTS), software (XON/XOFF), or none.

## Summary of Messages

**Make**  
**Die**  
**Empty**  
**Escape**  
**Format**  
**Free**  
**Get**  
**Handshake**  
**Look**  
**On**  
**OutputBuffer**  
**Put**  
**Queue**  
**Speed**  
**Timeout**  
**Valid**  
**Print To**

## Creation

```
Make <object> SerialPort(Int baud_rate, Int port [, Int handshake])
```

When making a SerialPort object the baud rate, the port and, optionally, the handshaking are specified.


The VM2 uses a group of four channels for each serial port: Transmit, Receive, Handshake input and Handshake output. If hardware handshaking is not enabled then the handshake channels are generally free to be used for other purposes. See the VM2 datasheet for more information on these channels.


Here we create a serial communication object on port 1 talking at 115200 baud, with hardware handshaking:


```
Make serial SerialPort(115200,1,1)
```

*An object like this is created by the default startup routine.*

- The port values are 1 - 5.
- The baud rates available lie in the range 1200 - 4,500,000 (port 1) or 600 - 2,000,000 (ports 2-5)\*
- Handshaking takes values of:
  - 0 - none
  - 1 - hardware handshaking with CTS input floating
  - 2 - software handshaking
  - 3 - use RTS for [RS-485 half-duplex mode control](#)
  - 4 - hardware handshaking with CTS pulled low)
  - 5 - hardware handshaking with CTS pulled high)
- The input and output buffers are both 256 bytes long. This is not configurable.

 You may need to connect RS232, RS485 or other line transceivers to your VM2 in order to use this object to communicate with other equipment. Some Application boards include transceivers on one or more serial ports.

 When the SerialPort is created the Rx channel is pulled High internally and any CTS input is set to floating.

 See the VM2 Datasheet for details of which ports use which VM2 channels.

\*Lower speeds are possible if the system clock speed is reduced. At 16MHz system clock speed the speed on ports 2-5 can go down to 150baud.

 Also see [Handshake](#), [Speed](#), [Format](#)



## Die

### Die

Die will reset the internal serial port peripheral so that it no longer controls or responds to its I/O channels, and so it draws no power.

The TX channel is set to input pulled high, to emulate an inactive transmitter. The I/O states of other I/O channels are not changed.

If you need the channels to change to a different state then you will have to do this explicitly.

## Empty

### Empty

This message will empty the serial input buffer, discarding any characters in it.

## Escape

### Escape $\Leftrightarrow$ Int

**Escape** governs the Ctrl-C (break) and Ctrl-T ([list tasks](#)) functions on the main serial port (**Escape** only applies to the 'main' serial port - the one used for programming - i.e. port 1). The **Escape** message is ignored when sent to the other serial ports.

When the main serial port is created **Escape** is set to 'on' - i.e. Ctrl-C & Ctrl-T are enabled. Note however that the **startup** procedure may modify **serial.Escape**.

**Escape** may be set to on or off by setting it to True or False (or, 1 or 0); similarly when reading its state, 1 is on and 0 is off.

When escape is off, Ctrl-C & Ctrl-T are treated as normal characters.

## Format

### Format $\Leftrightarrow$ Int

The Format [active variable](#) allows the serial communication format to be changed. The integer value is a set of flags coded as bits in a binary number.

Bit	Value	Meaning...	...When 0	...When 1
0	1	Odd Parity	No parity when both these bits are 0	Odd parity
1	2	Even Parity		Even parity

2	4	Data length	8 bits, <i>or</i> 7 bits data + 1 bit parity	9 bits, <i>or</i> 8 bits data + 1 bit parity
3	8	stop bits	1 stop bit	2 stop bits

The default format is '8-NONE-1' which has a format value of zero.

### Example

For a format of 7 data bits, even parity, one stop bit the value of the binary number is **%0010**, or in decimal, **2**.

```
serial.Format := 2 ; Set '7-EVEN-1' format.
```

### Free

**Free** ⇒ Int

Free returns the free space for characters in the serial input buffer.

If you want to know the Free value for the serial *output* buffer, see [OutputBuffer](#)

### Get

#### Get a single character

**Get** ⇒ Int

The Get message returns a character from the serial port, but waits if there isn't one available yet.

Note: CTRL-C characters can be trapped by the Escape function and so not seen in Get. If these characters need to be received, Escape may be turned off with [serial.Escape](#).

### Get a line

```
obj . Get(String s [, Int termination])
```

This will get a whole line of text and put it in string **s**.

By default the line is terminated with CR and input is not echoed back to the serial line.

If the optional parameter **termination** has the value 1, the line is expected to be terminated with CRLF or just LF (CR is ignored)

Neither CR nor LF is included in the stored string.

If the string capacity is reached, Get returns without collecting any further input.

### Get an array

```
Obj.Get(array a [, Int start, Int size])
```

**a** An array of 8 bit integers

**start**(default 0) element number at which to start transferring data to array

**size** (default whole array) how many bytes to transfer

This gets a fixed number number of characters into an array very efficiently.

### Handshake

```
Handshake ⇔ Int
```

Handshake allows the handshaking function of the port to be set.

Value	Handshaking
0	NONE
1	Hardware
2	Software
3	RS-485 <a href="#">half-duplex</a>
4	Hardware, with CTS pulled LOW
5	Hardware, with CTS pulled HIGH

Software handshaking is performed with the XON and XOFF characters in the ASCII character set. If enabled, these characters assume their control function and may not be sent over the serial link.


Note that setting software handshaking will always cause the immediate sending of an XON character. For this reason, if you want to send both **Handshake** and **Speed** messages to configure a serial port, you should set **Speed** first, then **Handshake**.

Hardware handshaking uses ports designated as RTS and CTS for each serial channel.

## Automatic handshake setting during download

When code is downloaded via VenomIDE the VM2 will automatically adjust its serial handshake setting to match VenomIDE.

This allows you to set serial handshaking to whatever setting your application requires (e.g. to **None** for serial debug output), without having to reset it every time you download new code.

 See also: [Timeout](#).

▼ The handshaking 'high and low water marks' inside the input buffer are set at 40 characters away from the buffer limits i.e. at 40 and 215 characters. Thus handshake output is asserted 'off', or XOFF is sent, when there are more than 215 characters in the input buffer, and these signals are turned on again when there are fewer than 40 characters.

On the other side of the serial ports, when handshake input is asserted off, or XOFF is seen coming in, then any character currently being transmitted is completed and no more are sent after that.

## Pulling CTS High or Low

Modes 4 and 5 only apply when the serial port is connected at logic levels to a device, i.e. without RS-232 level shifters in between. They define the behaviour of the transmit flow control when the device is physically disconnected.

4 (pull CTS LOW) means data can be transmitted (and will be lost) while disconnected.

5 (pull CTS HIGH) means data will not be transmitted while disconnected.

## RS-485 Half-Duplex Mode

The SerialPort object can drive a half-duplex RS-485 interface, i.e. one that uses a bidirectional transceiver which is switched between transmit and receive mode. This function is enabled by setting the **Handshake** value to 3, either using the Handshake message, or at [Make](#).

The SerialPort object uses its RTS signal for automatic control of the transceiver mode. See the circuit schematic for Application Board 3 (5922) for one way to do this.

Sending an [On](#) message to the SerialPort sets the transceiver to transmit mode. You should then send your data packet, using [Print](#) or [Put](#), etc.

When a stream of data has been transmitted and the port becomes idle, the direction control is immediately set to receive mode.


Another [On](#) message is required before transmitting again.

Example

**To** **init**

```
Make RS485 SerialPort(115200, 4, 3)
End

To SendPacket(packet)
Task.Off ; Turn of multitasking temporarily
RS485.On ;Transmit mode
RS485.Put(packet) ; Send the data
Task.On ; Multitasking restored
End
```

 Note that the transceiver will be reset to receive mode immediately the serial output buffer becomes empty, even if only momentarily. To guard against this happening in the middle of a packet you should ensure your packet of data is placed in the serial output buffer faster than it can be transmitted. Currently the best way to do this is to turn off multi-tasking around the operation.

If you use this method you should ensure that the packet you are sending is less than 256 bytes so that the serial buffer doesn't fill up, blocking your task, and consequently delaying other tasks in the system.

## Look

**Look**  $\Rightarrow$  Int

Look fetches, and returns the value of, the next character in the serial input buffer. If there was no character waiting it returns -1.

 [Get](#)

## OutputBuffer

**OutputBuffer**  $\Rightarrow$  SerialOutputBuffer

This message allows you to monitor the *output* side of the serial port.

The following messages usually refer to the serial *input buffer*:

**Free**  
**Queue**

However, you may refer to the serial *output buffer* like this:

**serial.OutputBuffer.Free**

`serial.OutputBuffer.Queue`

### Example

`Await serial.OutputBuffer.Queue = 0 ; wait for characters to tran`

### On

#### On

If the serial port handshake mode has been set to 3, this message turns the RS-485 Transceiver direction control to transmit mode.

See [RS-485 half-duplex mode](#) for more details.

### Put

The Put message sends one or more characters to the serial output buffer. It will block (i.e. wait) if the buffer is full.

Put can send:

- Single characters
- Arrays
- Strings

See below for each of these.

### Send a single character

`Put(Int character)`

Sends a single character to the serial output buffer, waiting if the buffer is full.

⚠ *character* will be masked to an 8-bit value, resulting in a character in the range 0 - 255

### Send an Array, or Array segment

`Put(array a [, Int start, Int size])`

Sends some or all of the bytes in an array, waiting if the buffer is full

**a**      An array of 8 bit integers

**start**(default 0) element number at which to start transferring data from array

**size** (default whole array) how many bytes to transfer

## Send a String

**Put** (String s)

Sends the contents of a string constant or string object to the serial port, waiting if the buffer is full

## Queue

**Queue**  $\Rightarrow$  Int

Queue returns the number of characters waiting in the serial input buffer.

If you want to know the Queue value for the serial *output* buffer, see [OutputBuffer](#)

## Speed

**Speed**  $\Leftrightarrow$  Int

The Speed [active variable](#) controls the baud rate of the serial port.

To ensure a baud rate change happens smoothly you should ensure that the serial output buffer is empty before changing baud rate.

The baud rates available on the serial ports range from 300 to 4,500,000 subject to the limitations in the table below, though the highest 'standard' baud rate is 115,200.

For some higher speeds the exact baud rate is not always possible so the closest alternative will be selected. So long as it is within a couple of percent of the correct value it will be acceptable. Reading Speed gives the exact rate used, subject to the accuracy of the system clock crystal oscillator.

The actual range of speeds available depends on which serial port is used and varies in proportion to the the system clock speed. The upper and lower limits are shown in the table below.

system clock	serial port 1 speed range	serial ports 2 - 5 speed range
16MHz	300 - 4,000,000	150 - 2,000,000
72MHz	1200 - 4,500,000	600 - 2,250,000

### Code Snippet: Keeping the Serial port Speed when changing System Clock speed

If you change the system clock speed the serial port will change speed in proportion; to change the clock speed on the command line without losing control of the VM2 you should change the serial port speed in the same line of code, e.g. assuming your serial 1 speed is currently 115200:

```
speed := 16 serial.speed := 115200
```

Or if you don't know the current serial port 1 speed:

```
temp := serial1.speed speed := 16 serial1.speed := temp
```

This will return to the command line without any serial data corruption.

### Timeout


**Timeout**  $\Leftrightarrow$  Int

This [active variable](#) controls the timeout for software handshaking. If an XOFF character shuts off the serial port's transmitter, and the subsequent XON is missed for some reason, the transmitter would stay off forever. However if timeout is set to a non-zero value, the transmitter will be turned on again after the timeout period.

Timeout defaults to 10,000 (10 Seconds) currently. It may be turned off entirely by setting it to zero. Its value is set in milliseconds, up to

the maximum positive integer value which would correspond to about 24 days.

Timeout is set and stored in milliseconds but the latency on checking the timer is normally 80ms.

 See also: [Handshake](#).

### Valid

**Valid**  $\Rightarrow$  Int

*Valid* is used to detect errors in serial reception. It returns five error flags as bits in a binary number. The flags are all reset to zero once Valid has been read.

Bit	Value	Meaning	Description
0	1	Parity Error	Incorrect parity in the received character
1	2	Framing error	The Stop bit was 0 (should be 1), caused by garbage or incorrect line speed (baud rate)
2	4	Noise error	The UART detected invalid changes of state on the serial line



3	8	UART overrun	The UART buffer received another character before the interrupt routine could remove the previous one.
4	16	Buffer overrun	The input buffer overflowed.


You should be able to use Valid to detect serial breaks, i.e. when the serial line is put into the 'active' state for a period (much) longer than the character frame. If Valid continuously indicates a framing error, and no new characters have been received then there is a serial break.

## Accepting Print

**Print To** <SerialPort> , <print list>

Printing to the SerialPort object sends the printed characters out of the serial port. The following print keywords are supported; others are passed on unprocessed in their internal representation and so should not be used.

<b>BEEP</b>	ASCII 7
<b>BS</b>	ASCII 8
<b>CLS</b>	ASCII 12
<b>CR</b>	ASCII 13 & 10
<b>Chr <i>n</i></b>	ASCII character <i>n</i>

 *Note: you can also print to an object by sending it the [PrintF](#) message.*

## Shaft

Shaft monitors quadrature-phase inputs to keep track of the position of rotary (shaft) encoders using the STM32F103's internal timer hardware. Up to 3 shaft encoders may be monitored at very high frequencies.

### Options available

The inputs may be pulled (up or down) or floating, the direction of counting may be changed and hardware digital filtering may be applied to the input signals.

### Future options

(It may be able to use inputs in the form UP/DOWN and CLOCK, though this has not been implemented yet).

## Summary of messages

**Make**  
**Count**  
**Reset**  
**Print**

## Creation

```
Make <object> Shaft (Int chan [, Int attributes [, Int filter]])
```

A new Shaft object is created, monitoring channels **chan** and it's associated channel (**chan + 1**).

**chan** gives the value of one of the VM2 channels to use for the shaft encoder input - it must be one of: \$10, \$16 or \$36.

The other channel used is implied by **chan**, and is always **chan + 1** - ie. \$11, \$17 or \$37 respectively.

**attributes** allows you to set the direction of counting of the Shaft object and also whether the inputs are pulled high, low, or not at all.

In more detail, the binary bits in the **attributes** parameter have these significances:

	Bit 2	Bit 1	Bit 0
<b>When 1</b>	Floating input	'Active high'	Reverse counting
<b>When 0</b>	Input pulled to inactive state	'Active low'	Normal counting



These are exactly the same [input attributes](#) as used for Digital, except for Bit 0, which allows you to reverse the direction of counting.

*The optional **filter** parameter may be used to set up digital filtering of the input signal.*

[Read more...](#)

## Example

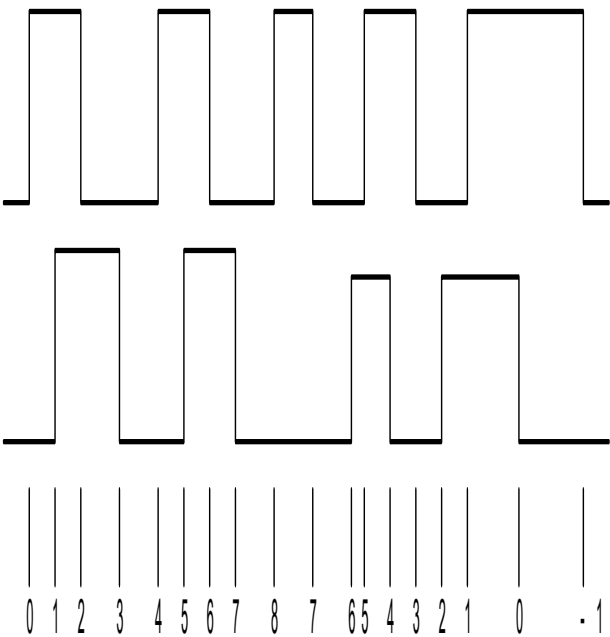
```
Make encoder1 Shaft($36) ; Encoder input on channels $36 and $37.
```

	Each Shaft object takes around 40 bytes.
	Shaft objects can count edges at over 1MHz.



Count

Count ⇔ Int

An active variable that holds the count of quadrature phase edges seen by the Shaft object. The count can be set or read. The diagram below gives an example of input waveforms and the count that they might produce. Note that the count will change on every edge.




If the count goes in the wrong direction when you rotate the encoder you can reverse it by setting attributes in [Make](#).

	Count can hold values up to around $\pm 2$ billion.
	See also <a href="#">Reset</a> , <a href="#">PulseCounter</a>

Reset

Reset

Resets the count to zero.


 See also [Count](#).

## Printing

**Print** <Shaft>

Prints the current value of the Count like this:

**[Shaft: 1330]**

 See also [Count](#).

## SMSLink

The SMS controller object simplifies the sending and receiving of messages using any GSM device that complies with the ETSI standards and supports PDU mode. It has been tested with a Siemens (now Cinterion) MC-35 GPRS terminal and should work with many other GSM terminals or mobile phones with suitable serial adapters.

The MC-35 has 35 message memory locations. An incoming message fills the lowest numbered empty memory location. The SMS object can read and delete any of these locations by number. Other devices may have different message memory sizes - consult the documentation when designing your code.

## Summary of Messages

**Make**  
**Address**  
**Empty**  
**Format**  
**Get**  
**Length**  
**Remove**  
**Send**  
**Time**  
**Print To**  
**Print**

## Character Set

For most purposes, all you need to know is that the SMS object will allow you to pass ASCII messages to and from strings or text buffers.

The rest of this section describes various provisions for data other than ASCII text.

There are 3 different character encodings in common use on GSM networks:

- GSM 7 bit encoding which includes ASCII and some European characters and is the default for text.
- UCS2 (16 bits) which is a two byte representation of the Unicode code points up to \$FFFF (Covers many European and Middle Eastern alphabets and symbols) and can also be used for binary data.
- GSM 8 bit encoding which is usually used for binary non-text data such as graphics, ringtones and SIM updates

The SMS object can use all three of the above encodings in GSM messages.

For communication with the Venom program, it uses text or 8 bit and 16 bit encodings, translating to and from the GSM 7 bit code as needed:

8 bit ISO-8859-1 with the addition of code point 0x80 for the Euro ( € ) symbol, borrowed from the WIN-1252 character map. ISO-8859-1 is the first 256 code points of Unicode and a superset of ASCII.

16 bit UCS2. The only character translated between this and text data types is the Euro Symbol. If a 16 bit message is received into, or transmitted from, a 16 bit Venom buffer or array variable, all values are passed unchanged.

### **Sending in GSM7 Format**

When sending messages, the message passed to the SMS object as a string or text buffer is assumed to be an ISO-8859-1 string and converted where possible to GSM 7 bit. The following codes will be translated successfully:

All ASCII printable characters except ` (back quote, code 96 or \$60).

ASCII CR (13) and LF (10)

These are usually available from your keyboard.

The following ISO-8859-1 and Win-1252 characters, listed with their decimal and hex codes for convenience.

char	dec	hex	char	dec	hex
€	128	80	ß	223	DF
ì	161	A1	à	224	E0
£	163	A3	ä	228	E4
¤	164	A4	å	229	E5
¥	165	A5	æ	230	E6
§	167	A7	è	232	E8
¿	191	BF	é	233	E9

Ä	196	C4	ì	236	EC
Å	197	C5	ñ	241	F1
Æ	198	C6	ò	242	F2
Ç	199	C7	ö	246	F6
Ñ	209	D1	ø	248	F8
Ö	214	D6	ù	249	F9
Ø	216	D8	ü	252	FC
Ü	220	DC			

The following characters belong to the GSM set but not to ASCII or ISO-8859-1, but their GSM codes may be inserted into an outgoing message and will display correctly on a mobile phone.

char	dec	hex
Δ	16	10
Φ	18	12
Γ	19	13
Λ	20	14
Ω	21	15
Π	22	16
Ψ	23	17
Σ	24	18
Θ	25	19
Ξ	26	20

### Receiving in GSM7 Format

When receiving messages encoded with the GSM encoding, the characters described above will always be received as sent.

### UCS2 Format

#### Receiving

If a received message uses the UCS2 character encoding and is restricted to the low 256 code points, the entire set of ISO-8859-1 characters as tabulated below will be received correctly as text. Additionally the Euro symbol (\$20AC in UCS2) is translated to \$80, which is not a defined code point in ISO-8859-1 but mapped to the Euro Symbol in the commonly used WIN-1252 character set so it will display correctly on many terminal configurations.

**Sending**

All values from 0 to \$FF are sent unchanged as 16 bit values, with the exception of \$80 which is translated to \$20AC, the UCS2 code point for the Euro symbol.

NUL 0 0 <sup>*[1]</sup>	@ 64 40	€ 128 80	À 192 C0
SOH 1 1	A 65 41	129 81 <sup>*[1]</sup>	Á 193 C1
STX 2 2	B 66 42	130 82	Â 194 C2
ETX 3 3	C 67 43	131 83	Ã 195 C3
4 4	D 68 44	132 84	Ä 196 C4
5 5	E 69 45	133 85	Å 197 C5
6 6	F 70 46	134 86	Æ 198 C6
BEL 7 7	G 71 47	135 87	Ç 199 C7
BS 8 8	H 72 48	136 88	È 200 C8
TAB 9 9	I 73 49	137 89	É 201 C9
LF 10 A	J 74 4A	138 8A	Ê 202 CA
11 B	K 75 4B	139 8B	Ë 203 CB
FF 12 C	L 76 4C	140 8C	Ì 204 CC
CR 13 D	M 77 4D	141 8D	Í 205 CD
14 E	N 78 4E	142 8E	Î 206 CE
15 F	O 79 4F	143 8F	Ï 207 CF
16 10	P 80 50	144 90	Ð 208 D0
XON 17 11	Q 81 51	145 91	Ñ 209 D1
18 12	R 82 52	146 92	Ò 210 D2
XOFF 19 13	S 83 53	147 93	Ó 211 D3
20 14	T 84 54	148 94	Ô 212 D4
21 15	U 85 55	149 95	Õ 213 D5
22 16	V 86 56	150 96	Ö 214 D6
23 17	W 87 57	151 97	× 215 D7
24 18	X 88 58	152 98	Ø 216 D8
25 19	Y 89 59	153 99	Ù 217 D9
26 1A	Z 90 5A	154 9A	Ú 218 DA
ESC 27 1B	[ 91 5B	155 9B	Û 219 DB
28 1C	\ 92 5C	156 9C	Ü 220 DC
29 1D	] 93 5D	157 9D	Ý 221 DD
30 1E	^ 94 5E	158 9E	Þ 222 DE
31 1F	_ 95 5F	159 9F	ß 223 DF

32 20	` 96 60	160 A0	à 224 E0
! 33 21	a 97 61	¡ 161 A1	á 225 E1
" 34 22	b 98 62	¢ 162 A2	â 226 E2
# 35 23	c 99 63	£ 163 A3	ã 227 E3
\$ 36 24	d 100 64	¤ 164 A4	ä 228 E4
% 37 25	e 101 65	¥ 165 A5	å 229 E5
& 38 26	f 102 66	¦ 166 A6	æ 230 E6
' 39 27	g 103 67	§ 167 A7	ç 231 E7
( 40 28	h 104 68	¨ 168 A8	è 232 E8
) 41 29	i 105 69	© 169 A9	é 233 E9
* 42 2A	j 106 6A	ª 170 AA	ê 234 EA
+ 43 2B	k 107 6B	« 171 AB	ë 235 EB
, 44 2C	l 108 6C	¬ 172 AC	ì 236 EC
- 45 2D	m 109 6D	173 AD	í 237 ED
. 46 2E	n 110 6E	® 174 AE	î 238 EE
/ 47 2F	o 111 6F	¯ 175 AF	ï 239 EF
0 48 30	p 112 70	° 176 B0	ð 240 F0
1 49 31	q 113 71	± 177 B1	ñ 241 F1
2 50 32	r 114 72	² 178 B2	ò 242 F2
3 51 33	s 115 73	³ 179 B3	ó 243 F3
4 52 34	t 116 74	´ 180 B4	ô 244 F4
5 53 35	u 117 75	µ 181 B5	õ 245 F5
6 54 36	v 118 76	¶ 182 B6	ö 246 F6
7 55 37	w 119 77	· 183 B7	÷ 247 F7
8 56 38	x 120 78	¸ 184 B8	ø 248 F8
9 57 39	y 121 79	¹ 185 B9	ù 249 F9
: 58 3A	z 122 7A	º 186 BA	ú 250 FA
; 59 3B	{ 123 7B	» 187 BB	û 251 FB
< 60 3C	124 7C	¼ 188 BC	ü 252 FC
= 61 3D	} 125 7D	½ 189 BD	ý 253 FD
> 62 3E	~ 126 7E	¾ 190 BE	þ 254 FE
? 63 3F	127 7F *[1]	¿ 191 BF	ÿ 255 FF

\*[1] Code values \$00-1F, \$7F-9F are not defined in ISO-8859-1 but will be received and may result in characters displayed according to the font used in your terminal, or with the usual ASCII control character meanings.



## Binary Data

The SMS object can also use 8 bit and 16 bit **Buffer** and **Array** objects for sending and receiving binary data using the GSM 8 bit and 16 bit (UCS2) data coding schemes. When this is done no code translation at all is performed.

## Creation

```
Make <object> SMSLink(Int serialport)
```

**serialport** 1 - 5 to select serial port to which GSM device is connected

Returned value: an SMSLink object variable.

The serial port must exist as a venom object before an SMSLink object is made using it.

## Example

```
Make serial SerialPort(38400, 2, 0)  
Make sms SMSLink(2)
```

## Debug

```
Debug  $\Leftarrow$  Int
```

Turns debugging mode on (value=1) or off (value=0, default)

In debugging mode, all output from the GSM device is echoed to serial port 1.

This will only work if the GSM device is not connected to serial port 1.

## Example

```
-->Make sms SMSlink(2)  
-->sms.debug := 1  
-->sms.get(2)  
AT+CMGF=0  
OK  
AT+CMGR=2  
+CMGR: 1,,33  
0791449737709399040C914477195524380000113092414135400FF4329E0E12D  
  
OK  
-->sms.debug := 0
```

```
--> sms.get(2)
-->
```

In this example, the long string of hex digits represents a message stored in location 2.

## Address

**Address**  $\Rightarrow$  String

Returns the senders address (i.e. phone number) of the last message read. This is a fixed string whose contents remain valid until a new message has been read or any message is deleted.

### Example:

```
print sms.address, CR
447791554283
-->
```

Note: the number format includes the international code in the first two digits. This example shows the equivalent of the UK number 07791 554283

## Empty

**Empty**

Clears the internal buffer holding the result of **Print To sms** statements.

This is not usually needed, as sending a message clears the buffer anyway.

## Format

**Format**  $\Leftarrow$  Int

This sets the format for encoding sent text messages which can be in 7 bit GSM code (default) or 16 bit Unicode

A value of 16 causes any text messages (from string or text buffer) to be encoded as 16 bit Unicode (UCS2).

A value of 7 causes any text messages (from string or text buffer) to be encoded using GSM 7 bit encoding.

Messages sent from 8 bit and 16 bit integer buffers or arrays are always sent in 8bit or 16 bit mode respectively and unaffected by the **Format** setting.

Example:

```
Make sms SMSlink(2)
sms.Format := 16
Print To sms, "This is a test message in 16 bit format"
sms.send("447791554283")
sms.Format := 7
Print To sms, "This is a test message in 7 bit format"
sms.send("447791554283")
```

It is unlikely that 16 bit mode will be required for text, especially as Venom does not currently support Unicode in strings. If you need to send messages containing characters with code points over 255, load the message into an array or buffer of 16 bit integers and use that instead.

## Get

**Get**(Int **n** [, Buffer/Array/String **buf**) ⇒ Int

- n** Message store number, starting from 1.
- buf** Buffer to which the message contents will be appended  
or array to which the message contents will be written  
or string to which the message contents will be written

Returned value indicates the type of message found in the message store.

- 1 Error (invalid message number)
- 0 No message
- 7 Received GSM 7 bit encoded text
- 8 Received 8 bit binary message
- 16 Received 16 bit text or binary message

Reads a message from the numbered store location in the GSM device

Stores the date/time, originator and text of the message in the SMS object. (see [Time](#), [Address](#) and [Print](#))

## Optional Buffer, String or Array Parameter

If a buffer or Array parameter is specified, its data type can be text or any integer type.

32 bit arrays are not permitted

The array must be big enough to hold the message.

If a string variable's allocated size is too short for the message, the message is truncated.


In certain cases with Buffers, conversion for the Euro symbol is performed between 8 bit (€ = \$80) and UCS2 (€ = \$20AC), as described in the following table.

#### Buffers and Strings

Message type	Buffer type	Conversion
7	16 or 32	Euro symbol: \$80 converted to \$20AC
16	text or string var.	\$20AC converted to \$80
16	8	integer values over \$7f are truncated to 8 bits
all other combinations		simple copy

#### Arrays

For arrays, the data is copied unmodified, except that a 16 bit message read to an 8 bit array will be truncated.

 See also: [Character Set](#)

#### Examples

```
-->Make sms SMSlink(2)
-->sms.get(3)
-->print sms
[ SMS from 447791554283 ]
2011-03-29 14:15:03
print txt [€]
-->

-->Make s string(160)
-->print sms.get(3, s), CR
7
-->print s, CR
print txt [€]
-->
```

#### Length

**Length** ⇒ Int

Returns the length of the GSM 7 bit encoded message stored in the SMS object as a result of **Print To sms** statements.

If the sms object is set for 7 bit text sending (the default) this length takes into account a handful of characters that have to be encoded as a two-character escape sequence.

For a message to be sent successfully using 7 bit GSM encoding, this value must not exceed 160.

For a message to be sent using 16 bit encoding, the maximum is 70.

The following characters result in an escape sequence and require two characters in 7 bit mode:

`^ { } \ [ ~ ] €`

**Length**(Str) ⇒ Int

Performs the same length calculation on a fixed string or text buffer.

**Length**(Int msgno) ⇒ Int

**msgno** Message number

This form of the Length message reads an SMS message from the GSM device's memory just like **Get**, but the returned value is the length of the message.

#### Example

```
-->print sms.length
0-->print to sms, "test "
-->print sms.length
5-->print to sms, "message"
-->print sms.length
12-->
-->n := sms.length(13)
-->print "message memory 13 has a received message of length ", n
message memory 13 has a received message of length      37
-->
```

#### Remove

**Remove**(Int msgno)

**msgno** Message store location to be deleted

- Deletes numbered message from GSM device's message memory
- Clears the current message

**Example**

```
-->Print sms.length(13)
37-->Print sms.get(13)
7-->sms.remove(13)
-->print sms.length(13)
0-->print sms.get(13)
0-->
```

**Send**

**Send(dest[, message]) ⇒ Int**

**dest** String or text buffer holding the recipients phone number

**message** String or buffer holding the text of the message


Returned value: True (1) if successful, False (0) if sending failed.

**Effect of 2nd Parameter on Message Data Encoding**

Param	Max length	Encoding
none	160	The message is the result of any preceding <b>Print To sms</b> statements, encoded as for string or text buffer.
string	160 (7 bit)	The message is sent by default using GSM 7 bit coding, unless the <b>Format</b> message has been used to change this to 16 bit UCS2 coding.
text buffer	70 (16 bit)*	
8 bit buffer or array	140	Contents are sent using GSM 8 bit encoding. This is not usually used for text messages in mobile phones, but for binary messages such as ringtones, SIM updates and graphics.
16 bit buffer or array	70	Contents are sent in a message that specifies UCS2 encoding in the header. Mobile phones display this using the UCS2 mapping, or it can be used for binary data.

\*16 bit mode is enabled by the [Format](#) message to the SMS object

Sending a message has no effect on the current stored incoming message.

 The Send message will wait for confirmation that the message has been sent successfully. If no confirmation is received, it will timeout after 20 seconds.

 See [Length](#), [Print To](#), [Format](#)

**Example**

```
; two ways of doing the same thing
```

```
; (1)
Print To sms, "this is a test message"
If sms.Send("447791554283")
    Print "sent OK",CR
Else
    Print "sending failed"
; (2)
If sms.Send("447791554283", "this is a test message")
    Print "sent OK",CR
Else
    Print "sending failed"
```

## Time

**Time** ⇒ **Int**

Returns the timestamp of the current (most recently read) message in Venom time i.e. seconds since the beginning of 1990.

It is sometimes useful to use this in conjunction with the [DateTime](#) object to convert date and time to components, but the simple integer form returned here simplifies comparison with other time values.

### Example

```
-->make dt datetime
-->sms.get(1)
-->dt.time := sms.time
-->print dt
2011-03-29 14:14:44-->
```

Note you can also get formatted time and date of a received message using [print](#)

```
Print sms:1
```

## Accepting Print

**Print To** <SMSLink>, **printlist**

The print output is appended to an internal buffer for subsequent sending.

If the limit of 160 characters is exceeded the extra characters are discarded without warning.

 See also: [Send](#), [Empty](#), [Length](#), [Printf](#)

**PRINT**

**Print** <SMSLink>:**select** [: **format**]

Select Value	What's Printed
--------------	----------------

none	Print originator, date/time and message text on three separate lines
0	Print originator (i.e. sender's phone number)
1	Print a string containing the date and time of the message. If a second colon operator is present, it controls the format of the date and time info in exactly the same way as the first colon operator when printing a <a href="#">DateTime</a> object.
2	Print the message text

If the message is empty, the string: "[SMS: empty]" is always printed.

```
-->print sms
[ SMS from 447791554283 ]
2011-03-29 14:15:13
test str [€]
-->print sms:0
447791554283-->print sms:1
2011-03-29 14:15:13-->print sms:2
test str [€]-->
```

**Code Example****Sending Data Readings by Text**

In this example, the VM2 monitors two voltage readings via 10 bit analogue inputs.

On receipt of a text message consisting of the word "READINGS", it sends them in a text message by reply.

If V1 exceeds a critical level, a message is sent to an alarm number immediately, and repeatedly every 5 minutes .

```
#Define v1_alarm_level 614      ; alarm level
#Define alarm_number "0797626xxxx" ; mobile # for alarm message

To init
  Make v1 Analogue(40)      ; the voltage measurement inputs
  Make v2 Analogue(41)
  Make serial2 SerialPort(38400, 2, 0) ; set up serial port
```



```

    Make sms SMSController(2)           ; SMS object uses serial
    Make msg String(160)                ; for received message
    Make alarmtimer timer(300000)       ; 5 minute timer for ala
End
To main

Every 5000      ; poll for request every 5 seconds
[
    print "v1 ", v1.value * 0.00488:1:2, " v2 ", v2.value * 0.004
    Repeat 4      ; poll 1st 4 message memories
        if SMS.Get(index, msg) > 0
            [
                Print "Received:",CR, sms, CR
                If msg.Compare("READINGS") = 0
                    [
                        Print To sms, "V1 = ", v1.Value * 0.00488:1:2, CR,
                            "V2 = ", v2.Value * 0.00488:1:2
                        sms.Send(sms.Address)      ; Reply to sender of incoming
                    ]
                sms.Remove(index)                ; clear message memory
            ]
        ; check alarm level
        if v1.Value > v1_alarm_level And alarmtimer.done
            [
                Print "v1 > alarm",CR
                alarmtimer.go                    ; restarts the timer
                Print To sms, "ALARM: V1 = ", v1.Value * 0.00488:1:2
                sms.send(alarm_number)
            ]
        ]
    ]

End

```

## SMTPSender

SMTP stands for Simple Mail Transport Protocol. Its most common usage is for personal computers to send email to a server for forwarding to its final destination. Hence the Venom SMTP object is for sending email.

See also [TCP/IP Networking](#)

## Summary of messages

**Make**  
**Debug**  
**Send**

## Creation

**Make** <object> **SMTPSender**(**server**, **id** [, **username**,  
**password**])

**server** String or text buffer containing the host name of a SMTP server which will be able to accept mail from the VM2.

**id** An optional hostname for the VM2 itself. If supplied as an empty string, the SMTP object will send its IP address in square brackets as its id when talking to the server, in compliance with the SMTP standard. The ID string must be present even if only as an empty string.

**username** Username for authenticating with the SMTP server, if required.

**password** Password for authenticating with the SMTP server, if required.

This creates an SMTP object that can then be used for sending email messages.

## Authentication

Most SMTP servers do not require authentication if they belong to the same ISP that provides your internet connection; however if you are using a different mail server you may still be allowed to use it as long as you identify yourself as a permitted user. There are several authentication mechanisms: the VM2's SMTP object only supports the mechanism known as "AUTH PLAIN".

## Debug

**Debug**  $\Leftarrow$  Int

The default value is 0.

Setting a value of 1 will echo the exchange of commands and responses with the SMTP server to serial port 1, useful for confirming that the object has been correctly set up for the server with which it will be used.

## Send

```
Send(Str fromaddr, Str toaddr, Str subject,  
      Var headers, Var body) ⇒ Int
```

Sends an email message.

Returns result value as follows:

- 0 - sent OK
- 1 - unable to connect to server
- 2 - Error in server commands
- 3 - Error sending headers
- 4 - Error sending content

<b>fromaddr</b>	string or text buffer	originator's email address.
<b>toaddr</b>	string or text buffer	recipient's email address.
<b>subject</b>	string or text buffer	"Subject:" line of message
<b>headers</b>	string, text buffer or file	optional headers, such as "Sender: ", "Reply-to: "
<b>body</b>	string, text buffer or file	the message contents

Email addresses may be in the form of these examples:

```
smith@abc.com  
<smith@abc.com>  
John smith <smith@abc.com>
```

The third form is preferred, if for no other reason than scoring fewer points on spam filters.

## SPI

SPI objects control the VM2's two SPI interfaces.

Each SPI object controls four VM2 channels. These are connected to the CS, CLK, MISO and MOSI lines.

The connections from the VM2 to a device are as follows. See the VM2 data sheet for the channel numbers:

VM2 Function	Connect to
MISO	Device's data out*

MOSI	Device's data in
CLK	Device's serial clock
CS	Device's chip select

*\*Not necessary if no data comes back.*

### Addressing multiple devices on a single SPI bus

**SPI bus 2** has two digital ports associated with it called **SPI ADDR 0** and **SPI ADDR 1**, on channels \$70 and \$71. These may be used (in conjunction with some external logic gates) for addressing up to four devices on SPI bus 2. Some or all of these addresses may be used by the Ethernet, one or more Memory Cards, or USB Host Filing system objects, for example.

Note that even though they are physically located on pins near to SPI bus 1, these address lines are intended for use with SPI bus 2.

You can emulate this address functionality for SPI bus 1 by choosing your own address line(s) and controlling them with Digital objects.

### Different modes

You can also connect multiple devices where each device has a different bus speed, clock mode and address. To do this you should create as many SPI objects as you have devices. Each object is used to control a particular device. When an SPI object's On message is sent, this selects the correct speed, clock mode and address. Note that for SPI bus 1 you will have to set the address yourself.

### Summary of messages

**Make**  
**Off**  
**On**  
**Put**

### Creation

```
Make <object> SPI (Int bus [, Int speed [, Int clk_mode
[, Int Address]])
```

### Parameters explained

**bus**: This selects which bus to use - either 1 or 2.

**speed**: (default: minimum) sets speed for SPI clock in kHz. The actual speed is the system

clock divided by a power of 2, and will be set to the requested value or the nearest value below it from the table below

For a VM2 running at 72MHz, the actual speeds available are as shown in the table. For lower system speeds the values in the table scale down proportionally.

Speed (kHz)	
140	(bus 2 only)
281	
562	
1125	
2250	
4500	
9000	
18000	
36000	(bus 1 only)

**clk\_mode**: This determines the clock polarity and phase as in the table below. The default value is 0.

clk_mode	Clock polarity	CPOL	Clock phase	CPHA
0	Clock idle low	0	Capture on 1st clock edge	0
1	Clock idle low	0	Capture on 2nd clock edge	1
2	Clock idle high	1	Capture on 1st clock edge	0
3	Clock idle high	1	Capture on 2nd clock edge	1

Sometimes the clock mode is call the *SPI transfer mode* and is represented by two variables: **CPOL** (clock polarity) and **CPHA** (clock phase). These two variables can each have the value 0 or 1. The clk\_mode parameter is equal to **CPOL\*2 + CPHA**.

**Address**: This specifies the address set up on channels \$70 and \$71 when using SPI bus 2. This address may be used to decode access to several devices on the one SPI bus by using it to gate the CS signal. Note that the Ethernet, Memory Card and USB Host interfaces usually use the first three of these addresses, but you should check to see which addresses are free on the hardware you are using. Address is in the range 0 to 3.

 SPI is a [Zero-Memory](#) object



Warning: If you are using SPI bus 2 on typical VM2 hardware that has memory card interfaces, you should avoid leaving a memory card in the holder without making a file system on that card. This would leave the card unpowered, causing data corruption on the SPI bus 2 connections.

## Off

### Off

This pulls the Chip Select signal high to end a communication.

## On

### On

This pulls the Chip Select signal low to start a communication.

Also, **On** will re-program associated hardware to select the correct address (SPI2 only), clock speed, clock mode, etc.

## Locking

SPI objects use Venom resource locks to prevent different tasks interfering with each other's use of the SPI hardware. There are two locks; each lock is associated with an SPI bus hardware module (rather than the SPI object.)

If you are using the same SPI hardware in more than one task (e.g. other tasks might be accessing Ethernet or a Memory Card, or another SPI object on the same SPI port) then you should lock the hardware (via the object) while you are using it. The best places to apply and then release the lock are just before On and just after Off. For example:

```
Make spi2 SPI(2, 18000)
...
spi2.Lock
spi2.On
spi2.Put(byte)
spi2.Off
spi2.Unlock
```

See [Locking](#) for more information.

## Put

```
Put(Int data[, Int nbytes]) ⇔ Int
```

The Put message both sends and receives data. Every byte you send out also results in a byte being returned.

If you only want to transmit data then you can ignore the result. If you only want to receive data then you can use a dummy value for **data**.

For sending and receiving a single byte of data use Put with a single parameter:

```
result := s.Put(data)
```

You can send and receive up to four bytes in a single Put by using the nbytes parameter to specify how many bytes of the value data to send:

```
result := s.Put(data, 3) ; send three bytes of the value 'data'.
```

These bytes are sent in Big Endian format - ie. the most significant byte is sent first.

## Little Endian

If you want to send and receive data in Little Endian format then you should use a negative value for nbytes:

```
result := s.Put(data, -3) ; send three bytes of the value 'data',
```

Note: Data is clocked in and out MSBit first.

## String object

String objects are strings of *variable* text - that is you can assign the text they are to hold at runtime, unlike string constants, e.g. "**A string**", which are defined at program time.

A string object has a *capacity*, or maximum length - this is the memory reserved for it when it is created. The actual text held in the string at any time may be less than the capacity of the String object, but it can never be more.

The text of a string object is null-terminated, i.e. there is a character of value zero at the end of the text.

## String constants

Constant (i.e. [quoted](#)) strings and String objects (or variable strings) are very similar - they are treated the same in almost every situation in Venom2, however you cannot do any write operations to a constant string, and the **Get**, **Readpoint**, **Reset** and **Queue** messages aren't accepted.

## Summary of messages

**Make**  
**Address**  
**Compare**  
**Element**  
**Empty**  
**Find**  
**Get**  
**Free**  
**Flush**  
**Length**  
**Put**  
**Queue**  
**Readpoint**  
**Reset**  
**Value**  
**Print**  
**Print To**

## Creation

**Make** <object> **String** (Int **max\_size**)

Create a string with a capacity of **max\_size** characters. The string is initialised to be empty.

Usage note: Don't bother using String objects where you could use a variable that refers to one or more string constants:

```
s := "some text"  
;...and maybe later:  
s := "some other text"
```

## Removing Strings

Sometimes you will wish to create a string for temporary use, and then discard it later. You can remove the string using the **.Die** message. This returns the memory taken by the string. Also take a look at [AutoDestruct](#) and [DELETE](#) as these may be useful.



[String constants](#)



⚠ The maximum capacity of a string is ~64K (65535) bytes long.

📄 Strings use up the memory taken by the text plus an overhead of up to 20 bytes. Eight of these bytes are the memory taken by any object that is held on the heap and the rest are used to hold information about the String object. Strings require contiguous memory in the heap - so if you try to create a very large string in system that has been running for a while with a lot of dynamic memory allocation going on (creating and killing objects, for example) then you may find that there is not enough contiguous memory to create the string. In that case Make will generate a RAM Full Error.

☠ BUG ALERT: Note the following Venom code snippet **does not** put the contents of the string constant into the string variable - it *overwrites* **s** with the string constant, and, critically, it loses the string object you made, together with the memory it took. This is an example of a *memory leak*, which, if it is repeated enough will result in your application running out of memory. If you suspect your application has memory leaks in it, you can use the [garbage scanner](#) to detect them.

**Do not do this:**

```
Make s String(10)
s := "some text" ; BUG!
```

One way of doing it correctly is

```
Make s String(10)
s.Put("some text")
```

## Address

**Address** ⇒ Int

Returns the memory address of the first character in the string.

## Compare

**Compare** (String or Buffer **str** [, Int **uncased** [, Int **length**]]) ⇒

Compare compares two strings. This allows you to put strings in alphabetical order. A Text Buffer may also be used as the second string.

The integer result returned depends on whether the second string is 'less than', 'greater than', or the same as the first string. Here, 'greater than' and 'less than' refer to comparing the strings character by character until one has a higher ASCII value than the other (or they turn out to be identical strings).

You can think of the **Compare** message as a minus sign between the two strings.

The return result is:

> 0            If the second string is 'less than'  
(Positive)  
= 0 (Zero)    If the second string is the same  
< 0            If the second string is 'greater than' the first  
(Negative)

For example:

```
-->Print "The first string" . Compare("The second string")
-13-->
```

Though of course normally you wouldn't be comparing two string constants.

### Case-insensitive comparison

The optional, second parameter determines whether the comparison is case sensitive or insensitive. The default value (0) means case sensitive. If the second parameter is 1 then the comparison is done in a case-insensitive manner:

```
-->Print "The first string" . Compare("THE FIRST STRING", 1)
0-->
```

Note that the case-insensitive compare is done in UPPER Case, which leads to the following characters being treated as greater than any alphabetic character:

```
[ \ ] ^ _ `
```

### Length-limited comparison

The optional third parameter, length, will limit the comparison to the first *length* characters of the string. If length is missing or zero the length of the comparison will be unlimited.

### Element

```
Element (Int item_number) ⇔ Int
```

```
<String>.(Int item_number) ⇔ Int
```

The Element() [active variable](#) allows both read and write random access to the contents of a string as if it were an array. Thus you may read or write individual characters in the string.

If you try to access elements outside the boundaries of the string you will get an *Array index out of range* Error.

Note: Venom abbreviated syntax allows the use of `.()` to substitute for `.Element()`.

Note: Write access is only permitted for variable strings, and not string constants; if you attempt to write to a string constant you will get an *Assignment to write-protected item* error.

## Empty

### Empty

Empty sets the string contents to an empty string of length 0 by writing a null character at the start.

The string is in the same state as if it had just been created.

Empty does not remove the string from memory. To do thus use the message **Die**.

## Find

```
Find (String str [, Int start_pos]) ⇒ Int
```

```
Find (Buffer buf [, Int start_pos]) ⇒ Int
```

Find searches a string for occurrences of the search text.

The search text is held either in another string or in a text buffer.

The search starts at the optional position specified by `start_pos` (or from the start of the string if no second parameter is supplied).

If a match is found, the start position of the found text is returned, otherwise the value -1 is returned.

The following example finds the start position of the first occurrence of a string constant in the string:

```
Make s string(100)
s.Put("text ttx txt text")
pos := s . Find ("tex", 3) ; pos will be 13
```

⚠ If the search text is held in a text buffer, then only the first 256 characters are used.

## Flush

### Flush

Flush removes any text from before the readpoint, and then copies the remaining text to the front of the string. The readpoint is reset to 0.

The overall capacity of the string remains the same, but the Length will be reduced if any text is moved 'forwards'.

## Free

**Free**  $\Rightarrow$  Int

Free returns the amount of space left in this string - i.e. how many more characters may be Put or PRINTed to it before it is full.

## Get

**Get**  $\Rightarrow$  Int

Returns the character from the string at the current [Readpoint](#) and increments the Readpoint. Array index out of range error if at the end of the string.

## Length

**Length**  $\Rightarrow$  Int

Length returns the current length of the string. This is not the same as the capacity of the String object.

To find the capacity of a String, you should add [Free](#) to Length.

## Put

**Put** (Int **character**)  $\Rightarrow$  String  
**Put** (String **str**)  $\Rightarrow$  String

Put will append a single character, string [constant](#) or string object the text currently held by the string, but only up to the capacity of the String object. Text that won't fit into the String is ignored.

## Dot chaining

Put returns the original String object, so you can use it like this:

```
Local s := New String(100).Put("(Initial text)")
```



Put is only allowed for variable string objects.

 [Print To](#)

## Queue

**Queue**  $\Rightarrow$  **Int**

Shows how many characters are available to read with **Get** message, i.e. between [Readpoint](#) and the end of the string.

## Readpoint

**Readpoint**  $\Leftrightarrow$  **Int**

Set or gets the current Readpoint, i.e. the position at which the next **Get** will read a character.  
The string starts at Readpoint = 0

You will get an Array index out of range error if you attempt to assign Readpoint past the current string length.

 [Get](#)

## Reset

**Reset**

This simply resets the [Readpoint](#) to 0.

## Value

**Value**(Int **base**)  $\Rightarrow$  Int

**Value**(Float)  $\Rightarrow$  Float

Extracts a numeric value from the start of a string. You can extract floating point values and integer values of different number bases.

Parameter	Interpretation
<i>None</i> or <b>10</b>	Decimal integer
<b>2</b>	Binary integer
<b>16</b>	Hexadecimal integer
<b>Float</b>	Floating point number

Any white space characters at the start of the string are skipped, and the conversion stops at the first character that is not a valid part of the type of number being evaluated, or at the end of the string if encountered first.

Leading plus and minus signs are correctly interpreted.

## Example

```
Make s String(10)
Print to s, "123"
Print s.Value
123
Print s.Value(16)
291
Print s.Value(Float)
123.
```

## Other number bases

You can specify any number base up to base 36, assuming use of the letters A-Z or a-z for the digits above 9.

## Accepting Print

```
Print To <String>, <print item>, ...
```

Strings are able to accept print. The print output is simply appended to any existing text in the string, up to the capacity of the string.


Text that won't fit into the capacity is ignored. For example:

```
Make str String(10)  
Print To str, "Some text to over-fill the string"  
Print str
```


Gives the result:

```
Some text
```

## Printf

 *Note: you can also print to an object by sending it the [Printf](#) message.*

When printing to a string, **CR** appends an LF (ASCII code 10) only to the string.

 **Print To** is only allowed for variable string objects. You can't print to a string constant - you will get a runtime error.

 [Put](#), [Printf](#)

## Printing

```
Print <String>
```

```
Print String : n
```

```
Print String : start : n
```

Printing a string prints out all the text in the string.

If the optional format parameters are used then any subsection of the string may be printed. See [formatting strings](#) for details.

## Example

Extract a subsection of a string and put it in another string:

```
Print To s2, s1:5:3
```

## Stopwatch

Stopwatches are for timing events. Once created they immediately start counting time in milliseconds. You can reset the time, or set it to any value you like, including negative values.

### Summary of messages

**Make**  
**Reset**  
**Time**  
**Print**

### Creation

**Make** <object> **Stopwatch**

A new Stopwatch object is created. Its Time is set to zero, but immediately starts incrementing at 1000 per second. For example:

```
-->Make s Stopwatch  
-->Print s:1  
00:00:03-->
```

 See also [Timer](#), [RealTimeClock](#), [PulseWidthIn](#).

### Reset

**Reset**

Resets the Stopwatch's time to zero.

### Time


**Time**  $\Leftrightarrow$  Int


Reads or sets the Stopwatch's time in milliseconds. Time may be set to any value, including negative values.

▼ You should be aware that after ~25 days (2,147,483,648 ms), the number of milliseconds timed by the stopwatch will wrap round and become negative. You should therefore reset any Stopwatches before this period if you need to rely on their Time



value.

 Stopwatches count in milliseconds. The accuracy depends on the VM2's crystal oscillator. See the VM2 Datasheet

 See also [PulseWidthIn](#) for timing between edges on signals.

## Printing

```
Print <Stopwatch> : Int f1 : Int dp
```

Prints the time or the period of the Stopwatch in different formats. **f1** gives the format style to use.

<b>f1</b>	<b>Format</b>	<b>Key</b>
:0	DD:HH:MM:SS.sss	D is a day digit (0-24)
:1	HH:MM:SS.sss	H is an hours digit (0-23)
:2	MM:SS.sss	M is a minutes digit (0-59)
:3	SS.sss	S is a seconds digit (0-59)
		s is a fraction of a second digit, specified by <b>dp</b> .

**dp** gives the number of decimal places to print the seconds value to. If **dp** is zero then no decimal point or fractional digits are printed.

*Tip: if you want to print out any ordinary integer in the format used by Stopwatch or Timer, you can create a Timer object, set its Period to the value you want to print, and then print it, remembering to specify printing the Period.*

▼ **dp** has a maximum value of 3.

▼ If no format is supplied the value of the Stopwatch is printed in the form:

**[Stopwatch: Time in mS].**

## Task

A Task object is used to represent and/or control a task.

New tasks may be started with the [Start](#) command; Start returns the task object.

A task object is returned by the **Start** command, and also, the **Task** keyword will return the currently running task.

You can't create a task object using Make or New.

When a task comes to an end (either by running out of code to execute, or with Stop, the task object may still be sent messages. However it 'knows' that the task it refers to does not exist anymore.

### Summary of messages

**Creation**

**Die**

**Done**

**State**

**Print**

 See also [Stop](#) & [List](#).

### Creation

There is no Make for Task objects - they are created by [Start](#).

### Die

**Die**

Die will stop the task. It is equivalent to Stop obj.

**tsk . Die**

### Done

**Done** ⇒ Int

Done returns True when the task is no longer running.

### Off

**Off**

**Off** turns off multitasking. There will be no more task switching until the **On** message is sent.

## On

### On

**On** turns multitasking back on after an **Off** message.

## State

**State**  $\Leftrightarrow$  Any

**State** is a read-write property of Task objects that can hold 'state' associated with a particular task. It may be used to send signals between tasks.

It often makes sense to set a task's State to an instance of a user-defined [Class](#).

**State** is initialised to [Nil](#). It is not AutoDestructed when the task dies, so you should take care of this using an AutoDestruct Local.

## Example code

Here is some simple template code to set up a task with some state.

Note: This code takes advantage of the property of **Nil** that it will return **False** when sent any message.

```
To main
  Local t := Start taskcode ; Start a new task and hold a reference to it
  Await t.State <> Nil ; Wait for the task to get itself in order
  ;...
End

;This procedure is run in another task
To taskcode
  AutoDestruct
  Local st := New taskstate

  Task.State := st
  ;...
End

;This Class is used to hold a task's state:
Class taskstate
  Value Int
  ;...

  To Initialise
    Value := 10
```

```
    /...  
End  
End
```

## PRINT

**Print** <Task>

Print or List will print the details of a task, for example

```
-->Print task  
Task 1:  
in this_procedure(line 2)  
in a task started from main (line 2).  
[During bytecode 'BRA' at $2603FE]
```

---

## TextAnalyser

The TextAnalyser object is used to extract meaningful numbers, words, strings, etc from an input stream object.

Any object that returns ASCII character values in response to a **Get** message may be used as the input stream, for example, **SerialPort**, **Buffer**, **String**.

## Summary of messages

**Make**  
**Find**  
**Get**  
**GetLast**  
**InputBuffer**  
**Look**  
**Queue**  
**Reset**  
**Valid**

## Creation

**Make** <object> **TextAnalyser** (Any **Source**, Int **QueueMode**)

### Parameters

- Source** The **Source** can be any object that accepts a **Get** message that returns an ASCII character, e.g. a **SerialPort** or a **Buffer(Char)**.  
If **QueueMode** is 1 or 2, the object must also accept a **Queue** message.
- QueueMode:** **QueueMode** determines how the **TextAnalyser** responds when there is no data available in the source.
- 0** For 'infinite' input streams, where **Get** will wait for more data to arrive if there is none currently.  
E.g. **SerialPort**
  - 1** For 'finite' input streams where **Queue** returns 0 to indicate the end of the data.  
E.g. text **Buffers** and **String** objects.
  - 2** For 'infinite' input streams where **Queue** must be non-zero before **Get** may be called.  
E.g. when reading a shared **Buffer** or **file** that is being written to by another task.

## Examples

Firstly, some useful definitions of the **queuemode** parameter values:

```
;TextAnalyser queue modes:
#Define WAIT_IN_GET 0
#Define END_ON_QUEUE 1
#Define WAIT_ON_QUEUE 2
```


Creating text analysers to read from different input sources:

Read from serial port 1, wait for input if none:

```
Make t1 TextAnalyser (serial, WAIT_IN_GET)
```

Read from a buffer, treat end of buffer as an 'end of input' condition:

```
Make t2 TextAnalyser (my_buff, END_ON_QUEUE)
```

 Creating a TextAnalyser Object uses ~50 bytes of memory.

## Find

**Find**(str **pattern** [, Int **uncased**])

Find reads characters by sending **Get** to the source until a string of characters matching the input has been found

If found, the next character to be processed will be the first character following the matched string.

If not found all input from the source will have been used.

Returns 1 (true) if pattern found, 0 (false) if end of input reached without finding pattern.


**pattern**: A string containing the pattern to find. Must be no more than 255 characters.


**uncased**(default = 0) optional boolean flag to make matching case insensitive

### Example

Input: 123456ABC789

**ta.Find**("ABC") ; will return 1; "789" will be next characters pro

 The **Find** message temporarily allocates 512 bytes of heap memory while it is processing.

 The search string can be no longer than 255 characters

## Get

**Get** analyses characters it reads from its source object, and can convert them to other kinds of value.

The **Get** message has several variations, each of which does a particular kind of textual analysis.

There are variations that read

- [Integers](#) of different kinds (Decimal, Hexadecimal, etc)
- [Floating point](#) numbers
- [Delimited text](#): a set of characters terminated by a delimiter
- The next character (see below)
- Read past any characters that occur in a given group (see below)

To read characters until a given string is seen, use [Find](#).

## Reading single characters

`Get`  $\Rightarrow$  `Int`

`Get` without any parameters will read the next character from the source object. For example:

```
c := ta.Get
```

## Reading past given characters

`Get('x', String skip)`  $\Rightarrow$  `Int`

`Get` with a first parameter of '**x**' will skip characters specified by the second parameter, a string.

`Get` returns the number of characters skipped. The next character to be processed is the next character in the input text that is not in the skip group.

For example:

```
ta.Get('x', " \t\r\n0-9") ; Skip whitespace and digits
```

The skip-character group follows the rules for defining [character groups](#) in **TextAnalyser**.

## Reading numbers and strings

To read numbers or strings see the following topics:

- [Reading numbers](#)
- [Reading strings](#)

### Reading numbers

#### Read an integer

`Get(Int base)`  $\Rightarrow$  `Int`

`Get(Int base, Int length)`  $\Rightarrow$  `Int`

`Get` with a single integer parameter (which specifies the number base) will read an integer in that base, e.g. Hexadecimal:

```
Print ta.Get(16) ; Read a hex number
```

You can also specify the maximum number of valid number characters to read:

```
Print ta.Get(10, 3) ; Read a decimal integer, max three characters
```

See [here](#) for more details.

## Read a floating point number

`Get(Float [, Int length]) ⇒ Float`

Get with the type parameter **Float** will read floating point numbers:

```
Print ta.Get(Float) ; Read a float
```

You can also optionally specify the maximum number of valid number characters to read:

```
Print ta.Get(Float, 8) ; Read float, max 8 chars
```

See [here](#) for more details.

## Reading numbers - detailed operation

Space characters are skipped until a valid character or '-' or '+' is encountered.

For number bases up to 10 , only numerals are valid characters.

For number bases over 10, numerals and all letters are read as valid characters

For base-n numbers, the letters A-Z or a-z (not case-sensitive) are accepted for digit values above 9.

For float values, '.', 'E', 'e', '-', '+' and numerals are accepted as part of the number; any other character is a delimiter.

Floating point values may include an exponent which may be signed e.g.

```
876.432 -1.2 1.23E6 -4.0E-10 4.85e+5
```

If a length is specified in the second parameter, it sets the maximum number of valid characters to read *after* skipping initial spaces.

If the first non-blank character encountered is not a valid number character, a value of 0 or 0.0 is returned and the TextAnalyser's **Valid** state is set to false.

The first delimiter character encountered is consumed from the input and stored where it can be retrieved by a **GetLast** message.

## Reading strings

*This page describes how to collect characters from the input until a delimiter is seen. To skip input until a given string is seen, use [Find](#).*

## Read a delimited string

`Get (String dest [, String delimiters]) ⇒ Int`

Get supplied with a parameter that is a **String** object will copy characters into the string from the source until one of the default delimiters is seen. The delimiter is *not* read into the string, but



is read from the source object. You can discover its value using [GetLast](#).

The default group of delimiters is the set of control characters, i.e. characters with an ASCII value of less than 32. Examples are carriage return, line feed and tab.

You can also supply an explicit delimiter set as [character group](#).

Example:

```
Make str String(100)
ta.Get(str, "\r\n*") ; String ending with line end or '*'
```

It is also possible to specify that the string be read in [CSV format](#).

### Detailed operation

- The destination **String** object is emptied and reset before reading begins
- **Get** returns the number of characters read into the string.
- The delimiter character is not read in to the string, but is not the next character to be processed. Instead its value is available using [GetLast](#).
- If the destination string is filled to capacity before a delimiter is seen then reading stops and **Get** returns.



*There is a difficult-to-check-for situation where the string in the source is exactly the same length as the destination string capacity. In this case the delimiter remains in the input stream as the next character to be read. This issue will be addressed by a functional improvement at a later date.*

### Groups of characters

The following rules describe a shorthand way of defining characters sets used in the **TextAnalyser**.

- "-" between two characters indicates a range e.g. "a-zA-Z0-9". To use '-' explicitly, put it at the beginning or end of the list.
- "^" at the start of the string inverts the meaning: a delimiter is anything NOT in the list
- Control characters should be explicitly defined using \, e.g. "\r\n"
- Other ASCII characters may be specified using \hh, see [escape sequences](#).

### Examples

" / : , \ t "	Space, slash, colon, comma, tab are delimiters
" ^ - 0 - 9 "	Anything except ' - ' sign and numerals is a delimiter
" ^ 0 - 9 a - z A - Z "	Delimiter is anything non-alphanumeric

### CSV - Comma Separated Variables

This is a widely used file and data format, though there is no fixed official standard for it.

CSV can be exported by many spreadsheet programs and is also the data format used in NMEA data from GPS units and other navigational aids.

CSV mode is selected by specifying a delimiter list as the exact string ",," (two consecutive commas) or any other pair of the same character which is used as the separator.

The protocol analyser makes the following interpretations in CSV mode:

- Input fields are terminated by comma ', ' (or other chosen separator character), a line feed '\n' or end-of-input condition.
- Unquoted leading and trailing spaces are ignored
- All unquoted control characters except linefeeds are ignored.
- Any data enclosed in double quotes may contain leading and trailing spaces, commas and non-printing characters which are copied into the data string.
- Within double quotes, a pair of consecutive double quotes inserts a double quote character into the field.
- The comma or other delimiter is consumed and is not seen by a subsequent **Look** or **Get** message but can be retrieved by a **GetLast** message.

Note that CSV mode will correctly return empty strings if there are consecutive commas, or commas separated by nothing but unquoted space.

## GetLast

**Getlast** ⇒ **Int**

This message retrieves the character that terminated the last string or numeric read.

Typically this is:

- ' , '     comma or other specified separator between items in a line
- 10     LF in CSV mode if string was at end of line terminated with CRLF
- 13     CR in string (not CSV) mode if line ended with CRLF
- 1     If an end-of-input condition was met

## InputBuffer

**InputBuffer** (Object [,Int Queuemode])

This resets the Protocol analyser to use a different source of data from that specified when first created..

This includes the effect of a **Reset** message so the next character to be read will always be from the new source.

By default the end-of-input policy remains unchanged; this can be overridden with the Queuemode parameter which, if present, has the same values as the Queuemode parameter when a new **TextAnalyser** is [created](#).

## Look

**Look**  $\Rightarrow$  Int

Returns the next character to be processed.

If there is no more input and the object was created with **queuemode** value of 1 (see [Creation](#)), -1 is returned.

By definition this returns the same value that **Get** would, but without consuming the character from the input stream.

## Queue

**Queue**  $\Rightarrow$  Int

The **Queue** message returns the numbers of characters known to be available in the input stream. It uses the **Queue** message to the input object, and adds any character buffered by the protocol analyser itself.

## Reset

**Reset**

**Reset** is needed when the input stream source object has changed internally. This could occur if the input stream object has a readpoint that has been changed or if the data inside the input stream object has been overwritten.

The **TextAnalyser** object caches a single character from the input stream when it reads a delimiter character that is not subsequently used. **Reset** clears out this cache.

## Valid

**Valid**  $\Rightarrow$  Int

Returns True (1) or False (0) depending on whether the last Get was successful.

In particular, a return value of 0 means an empty string or numeric value was read (end of input or delimiter)

## UDPProt

UDP stands for User Datagram Protocol. UDP datagrams do not require the setting up of a connection. The transmission is typically fast but survival of the data in transmission is not guaranteed. UDP is useful for some types of application:

- Real time updates of continuously varying data, where any missing value is shortly overwritten by the next.
- Simple query-and-reply protocols with short messages that will fit in a single packet each. DNS queries normally use this as it is much faster than TCP.
- Applications that need tight control over acknowledgement timing.
- Any application requiring broadcast data.

UDP is used internally by Venom for:

- DNS lookups
- Getting the time and date for sending email with the SMTP object

See also [TCP/IP Networking](#)

## Summary of messages

**Make**  
**Address**  
**Close**  
**Debug**  
**Empty**  
**Find**  
**Get**  
**Length**  
**Open**  
**Put**  
**Send**  
**Queue**  
**Time**  
**Print**  
**Print To**

## Creation

**Make** <object> **UDProt**  $\Rightarrow$  udp object

Creates a UDProt object. This can then be used to send and receive UDP packets.

To send packets use [Put](#) and [Send](#)

To receive packets, use [Open](#), [Queue](#), [Get](#) and [Empty](#).

## Address

**Address** (Int **type**)  $\Rightarrow$  Int

<i>type</i> = 0	Returns Source IP address of received packet
<i>type</i> = 1	Returns Destination IP address of received packet

**udp.Address(0)** is the same as **udp.Get(1)**

**udp.Address(1)** is useful in systems where multicast packets are received, as the VM2 can receive multicast packets for more than one multicast address (even though the Ethernet object can do some filtering) and needs to know which is which to process or ignore them correctly.

## Close

### Close

Closes a UDP listener and frees any existing transmit and receive buffers.

## Debug

### Debug (1)

Displays a list of UDP objects, showing listening port number where applicable. Note that the list appears in reverse order of creation.

### Debug (2)

Lists the DNS cache, which contains a maximum of 100 most recently looked-up domain names. Entries are expired after an hour. The expiry time in minutes is shown for each entry. The most recent and frequently used entries are at the top of the list.

### Debug (3)

Shows information about the udp object, including the contents of any outgoing or received packets.

Any other parameter value will display a brief help message.

## Empty

### Empty

Clear received packet, leaving buffer ready for next to be received.

Note that the UDP object can only hold one received packet at a time. While the receive buffer is occupied, any further received packets are ignored. After the Empty message has been sent, the next received packet will be placed in the buffer.

## Find

**Find**(string **host**) ⇒ Int

Performs a DNS query to get the IP address of an internet host  
host is a string containing the domain name to look up.

The returned value is the IP address of the host, or 0 if the query failed.

The names `localhost` and `loopback` are recognised internally and converted to the standard loopback IP address 127.0.0.1.

Note that a connection to a nameserver must be available for this to work for names other than `localhost` and `loopback`. When a PPP connection is made to the internet, the ISP usually provides a nameserver address during the initial link negotiation. When connected to an Ethernet, DHCP can be used to discover a nameserver, or its address can be set explicitly by sending the correct form of **Address** message to an IP object.

This DNS mechanism uses a local cache of names and addresses to save time on repeated lookups. In the current version the cache will hold 100 entries, each entry expires an hour after it was first obtained and least recently used entries are discarded when the cache becomes full.

## Get

**Get(strobj) ⇒ Int**

This form copies the data contents of any UDP packet to a string variable, up to the maximum string length of the variable if this is less than the amount of data in the packet. It would be usual to expect the packet to contain readable ASCII text when using this form of Get. If it is used on binary data the string would be considered terminated prematurely on the first zero data byte.

The value returned is the length of the string.

**Get(1) ⇒ Int**

Returns the IP address of the sender of a received UDP packet.

**Get(2) ⇒ Int**

Returns the source port number of a received UDP packet.

**Get(3) ⇒ Int**

Returns the destination port number of a received UDP packet.

**Get(4) ⇒ Int**

Returns the data length of a received UDP packet. See also [Length](#).

**Get(5, Int offset, Int length) ⇒ Int**

Returns a component of the data in a received UDP packet.

offset is the number bytes from the start of the packet data (offset 0 is the first data byte)

length is 1, 2 or 4 and is the size of the data item to return. Values of 1 or 2 bytes are converted into standard 32 bit Venom integers using unsigned arithmetic, so a byte value of \$FF is returned as 255 and a two byte value of \$FFFF is returned as 65535, not as -1.

**udp.Get**(Array/buffer **a**)  $\Rightarrow$  Int


Returned value: the number of elements transferred

This form reads the incoming packet's data into an array or buffer, much faster than repeated **udp.Get(5, x, y)** messages.

The array or buffer can be of any data type.

For an array, the number of bytes transferred is the size of the UDP packet data or the size (in bytes) of the array, whichever is smaller.

For a buffer, the entire data contents of the UDP packet are appended to the buffer.

 [Open](#) - To configure the UDP object to listen for incoming packets.

[Queue](#) - To check whether a packet has arrived.

[Empty](#) - To clear the received packet buffer, leaving it ready for the next packet to be received.

## Length

**Length**  $\Rightarrow$  Int

Returns the length of the data part of a received UDP packet. It is a synonym for **udp.Get(4)**

## Open

**Open**(Int **port**)  $\Rightarrow$  Int

Allocates a receive buffer and enables the object as a UDP listener. A specific port number can be given, or 0 which means accept packets with any destination port number.

The returned value is true or false: a false value means that new listener was not created because a listener already exists with the same port number.

## Put

**Put**(Int)

Appends a character or byte value to the object's transmit buffer.

**Put**(Array **a** [,Int **start** , Int **size**])



<b>a</b>	The array to append to the udp packet
<b>start</b>	start position of array data (default 0)
<b>size</b>	number of elements to transfer (default: to end of array)

Appends the contents of the array to the data in the UDP outgoing packet.

The array may be of any data type.

If start and size parameters are given, a subset of the array is copied.

▼ Note that the UDP object imposes an internal limit of 592 bytes for the user data. If you create outgoing packets using 16 bit or 32 bit arrays, bear in mind that these occupy 2 or 4 bytes per element respectively.

## Send

```
Send(host, port [, myport])
```

Sends a packet containing the data assembled by **Put** and **Print To**

**host** is a host (domain) name or IP address to which to send the packet.

**port** is the destination port number.

**myport**, if specified, is the source port number. A value of zero is used by default, which means the recipient is not expected to have any use for a source port number, i.e. will not be replying.

## Time

```
Time ([Int Force [, str server] ⇒ Int
```

**Force** If true (non zero), force resynchronisation to a time server. You don't normally need this as it's done automatically.

**server** This will set a time server to be used in future UDP Time messages, instead of the default server list

Returns internet time as a Venom time value, which is the time in seconds since midnight GMT on 1 January 1990. This is the same time base as used by the [RealTimeClock](#) object.

For this message to work, there must be an existing connection to the internet to set the internal

clock. Once the clock has been set successfully from an internet NTP (Network Time Protocol) server, the time is estimated using Venom's internal millisecond timer. If more than about 10 hours has elapsed since the clock was last synchronised to the net, it is resynchronised if a connection is currently available.

If there has never been a successful connection to a time server, a value of 0 is returned.

Four ntp servers are used by default. They are tried in succession until one returns a response, so that the system is robust if any time server should fail, and so as to spread the load evenly between servers. The servers used are:

```
0.uk.pool.ntp.org
1.uk.pool.ntp.org
2.uk.pool.ntp.org
3.uk.pool.ntp.org
```

You can choose a different server by specifying its domain name. This has to be done only once when your program starts; the same server will then be used for all time synchronisation requests instead of the defaults listed above. This is useful if your system is not running in the UK and you want to use a closer NTP server, or is not connected to the internet but does have access to a LAN with a local NTP server. For an international list of NTP servers, visit <http://ntp.org> and look for public time server lists.

### Use with RealTimeClock

The Real Time Clock in the VM2 can be synchronised to GMT using this service very simply:

```
...
Make   rtc RealTimeClock
Make   udp UDPProt
...
rtc.Time := udp.Time
```

### Queue

**Queue**  $\Rightarrow$  Int

Returns 1 if a listener has received a UDP packet or 0 otherwise.

Note: there is only space for one packet in the receive buffer. Any packets that are sent while there is a packet in the receive buffer are lost.

 [Empty](#)

## Print

**Print** udp

Shows the number of bytes in a packet waiting for output, and the UDP object's listener state, which is one of:

- listening (no packet received)
- not listening
- listener has received data (shows how many bytes and where from)

## PRINT TO

**Print To** <UDProt>, <print list>

The transmit buffer of a UDP object can be used as the destination of a Print operation.

When printing to a UDP object, **CR** sends an LF (ASCII 10) character to the buffer.



[PrintF](#)

## PPPProt

PPP stands for Point to Point Protocol and is a widely used standard for encapsulation of packets and link management on a [serial](#) line.

A PPP object makes a connection via serial line and modem or GPRS terminal, usually to an internet service provider. The object handles all LCP and IPCP negotiation.

See also [TCP/IP Networking](#)

## Features

CHAP or PAP authentication

IPCP negotiation to obtain dynamic IP address and DNS server addresses.

Host mode also available, where the VM2 emulates a modem and the "server" end of a dialup connection.

## Summary of Messages

**Make**  
**Close**  
**Debug**  
**Name**  
**Open**  
**Period**  
**Reset**  
**Timeout**  
**Valid**  
**Print**  
**Print To**

### [Code Example](#)

#### Creation

```
Make <object> PPPProt(serialobject)
```

The `serialobject` parameter is a `SerialPort` object previously created. PPP has been extensively tested on serial port 2 but should work on any of the VM2's serial ports. The port is usually expected to be connected to a modem or GPRS terminal.

```
Make <object> PPPProt(serialobject, ip_address  
my_address, ip_address remote_address)
```

**serialobject**     The serial port on which to create the PPP object  
**my\_address**       String or integer representing the VM2's IP address  
**remote\_address**   String or integer representing IP address of remote end that connects to VM2

This form creates a PPP object in Host Mode, enabling another VM2 to connect to it with a normal PPP object and a serial interconnect cable.

It should also work with any host expecting a dialup modem connection. The VM2 emulates a modem and PPP dialup connection.

At present, VM2 host mode does not require any authentication, so a user and password are not defined.

IP addresses should be chosen which will not conflict with any other addresses used by the connecting host, i.e. they should not be in the network address of any connected LAN.

### Configuring a Host for dialup connection to a VM2 in Host mode

If a phone number has to be specified, it can be anything.

No authentication is required, so no user name or password should be needed.

The VM2 responds to most "AT" modem commands with "OK"

It responds to any command starting "ATD" with "CONNECT"

The VM2 will allocate IP addresses for both ends of the link.

## Close

### Close

Disconnect.

Sends LCP terminate request, and then attempts to disconnect the modem by sending the standard disconnect sequence:

```
<1 second pause> "+++" <1 second pause> "ATH0" <CR>
```

**Off** is a synonym for **Close** .

## Count

### Count(Int type) ⇔ Int

This sets or returns one of four counter limits used in configuring the PPP link. These limits prevent negotiation from continuing indefinitely without converging, i.e. the two ends cannot agree on a mutually acceptable set of options.

If one of these values is exceeded during a PPP [Open](#) process, the open will fail.

For most purposes the default values will work, but in exceptional cases they may need changing, and RFC1661 requires them to be configurable.

The types of count and default values are as follows:

Type	Default Value	Name	Meaning
0	10	Max-Configure	Maximum number of configure requests sent
1	5	Max-Fail	Maximum Number of configure NAK or REJ sent
2	5	Max-Auth	Maximum number of attempts to send authentication data
3	2	Max-Term	Maximum number of Terminate Requests sent

## Debug

**Debug**  $\Leftrightarrow$  **int**

Enables (1) or disables (0) debug output during the connection phase.

The DEBUG option produces a trace, for a terminal connected to serial port 1, of the modem control and LCP packet exchange, which may help when testing the connection to an ISP with a modem or GPRS unit. Typical output will look like this:

```
user_dbg=1
ATZ
OK
AT+CGDCONT=1,"IP","orangeinternet"
OK
ATD*98*1#
waiting for CONNECT

CONNECT
user Orange secret
Tx:LCP:CONFREQ id=180 len=4
Rx:LCP:CONFREQ id=3 len=29 MRU=2000 ACCM=000a0000 PFC ACFC MAGIC=ac14e9f8 AUTH=CH
Tx:LCP:CONFREQ id=3 len=24 MRU=2000 ACCM=000a0000 PFC ACFC MAGIC=ac14e9f8
Rx:LCP:CONFACK id=180 len=4
Rx:LCP:CONFREQ id=5 len=9 AUTH=CHAP
Tx:LCP:CONFACK id=5 len=9 AUTH=CHAP
Rx:CHAP challenge id=1 len=25:
Tx:CHAP response id=1 len=27:
Rx:CHAP success id=1 len=4:
Tx:IPCP CONFREQ id145: [6]IPADDR=0.0.0.0
```

## Mapping

**Mapping(Int Map)**  $\Leftrightarrow$  **Int**

PPP allows all ASCII control characters to be mapped to a two-character escape sequence to prevent confusion where local serial links use control characters for other purposes.

By default the Venom PPP object maps the characters XON (17 or Ctrl/Q) and XOFF (19 or Ctrl/S) so that XON/XOFF flow control protocol can be used with the modem if desired.

The **Mapping** message allows any selection of control character to be mapped to escape sequences.

The **Map** parameter is a 32 bit integer where each bit is set to 1 where its bit number (numbering from 0 for LSB to 31 for MSB) is the code for an ASCII control character to be mapped.

For example the default mapping value is \$000a0000, where bits 17 and 19 are set, corresponding to XON and XOFF respectively.

A value of \$ffffffff would map all ASCII control chars to be escaped, which is the default

for PPP if no option is negotiated to change it, but which adds about 12% overhead to the number of bytes transferred because of the two-character escape sequences used.

A value of 0 would allow all control characters to pass though unchanged.

Note that reading `ppp.Mapping` after opening a connection may give a different value from that set initially, because the remote end may negotiate for extra mappings.

It is almost always safe and efficient to leave this setting at its default value.

## Name

**Name** (Str `username`, Str `password`)

This sets the user name and password for authentication.

Most PPP connections require authentication before use.

PPP can use either PAP (Password Authentication Protocol) or CHAP (Challenge Handshake Authentication Protocol). The choice is negotiated automatically and you don't have to specify which is used.

## Client Mode

In the usual client mode of PPP, repeating this message will replace any previous user name and password with new values.

## Host Mode

In a ppp host, any number of username/password combinations may be set by using the name message repeatedly.

The connecting client may identify itself with any of those user names, and must use the corresponding password.

**Name** (Stringobj `s`)

In host mode, after a connection is open this form of the message will set `s` to the name of the user that connected. This enables the host to select different behaviour or capabilities per user, or to use the user name in connection with any data storage activity.

Note the list of users/passwords can be cleared by `ppp.Reset`

## Open

**Open**  $\Rightarrow$  **IP address or 0**

Starts connection and returns IP address of negotiated connection if successful, zero if failed.

**On** and **Go** are synonyms for **Open**.

- Commands defined by printing to the PPP object are sent to the modem.
- The response from the modem is monitored for certain strings to speed up the response time, and the timeouts depend on the command sent to the modem.
- The strings "ERROR", "NO CARRIER", "NO DIALTONE" are recognised at any time and result in immediate failure return.
- The string "OK" is recognised and results in the next command being sent immediately.
- If the command contains the string "ATD" there is a 60 second time limit waiting for a response
- All other commands have a 2 second time limit waiting for a response

After the modem has made a successful connection there are exchanges of LCP packets to establish a valid connection to the internet:

1. LCP negotiation to establish that the VM2 performs no compression, and escapes all ASCII control characters. At this stage the VM2 also finds out whether the remote end prefers PAP (Password Authentication Protocol) or CHAP (the more secure Challenge Handshake Authentication Protocol). The VM2 can use either.
2. Authentication: the chosen protocol is used, with the username and password, to establish the VM2's identity
3. IPCP (Internet Protocol Control Protocol) determines the local and remote IP addresses of the ends of the PPP link, and attempts to obtain primary and secondary nameserver (DNS) addresses.

## Period

**Period**  $\Leftrightarrow$  **Int**

Returns and optionally sets the period in milliseconds of inactivity after which LCP ECHO messages are sent.

Setting a value of 0 disables sending echo messages and checking for timeouts.

The default value is 0 (echo testing disabled)



If this value is set non-zero, the PPP protocol will send an LCP echo request message when no packet has been received for the period specified, and while no packets are received will send more echo requests at time intervals of the same period. If no response is received after a timeout period is exceeded, the link is assumed to be down and marked internally as dead, requiring a new **Open** message to restart it. The default timeout period is 120 seconds, which can be changed with the **Timeout** message.

Normally an echo request should cause an echo response to be sent back immediately from the other end of the link.

The **Period** value should be set much smaller than the **Timeout** value so that several echo requests can be sent before the link is declared dead. Typical figures might be:

Period	Timeout	no. of echo requests
10000 (10 sec)	60000 (1 min)	5
30000 (30 sec)	120000 (2 min)	4

 See also: [Timeout](#)

## Timeout

**Timeout**  $\Leftrightarrow$  **Int**

**Timeout(0)**  $\Leftrightarrow$  **Int**

1. Returns and optionally sets the timeout period in milliseconds for opening the link. If the link becomes physically disconnected this may be the only way to detect an [Open](#) failure.
2. If echo requests have been enabled by setting **Period** to a non-zero value, this value is also the time after which the link is assumed dead if no packets have been received.

[Period](#) description for hints on suitable values.

The default timeout value is 120000 (120 seconds).

**Timeout(1)**  $\Leftrightarrow$  **Int**

Returns or sets the timeout period in milliseconds for retransmitting configuration requests during LCP and IPCP negotiation when the connection is being opened.

The default value is 3000 (3 seconds) but some communications media may benefit from a longer or shorter value, e.g. an Iridium satellite data link was found to negotiate more reliably with the timeout set to 10 seconds.

 See Also: [Period](#)


## Valid

**Valid**  $\Rightarrow$  Int

Returns True (1) or False (0) depending on whether the link is open and working.

A false value can mean any of the following:

- Initial state, not connected
- sent a Close message
- received a TERMREQ packet from other end
- Packet reception timeout, only if echo requests were enabled with the Period message

 see also: [Close](#), [Period](#), [Timeout](#)

## Reset

**Reset**

Clears all text sent to the object by **Print To ppp** statements, and any stored user name and password info.

The PPPprot object is in the same state as when initially created.

## PRINT

**Print** <PPPProt>

Lists all the text that was sent to the PPP object by Print, consisting of modem commands.

e.g.

```
-->Print ppp
AT&F0
AT&E0
ATD0555123456
-->
```

## PRINT TO

**Print To** <PPPProt>, print list

All commands required for setting up a modem or GPRS terminal are defined by printing them to the PPP object, where they are stored until an **Open** message is sent. Each command is terminated by a new line (**CR** in the print list)

e.g.

```
Print to PPP, "AT&F0", CR, "ATD0555123456",CR
```

## Code Example

```
; Program to demonstrate simple use of the PPP object
```

```
To init
```

```
    Make serial2 Serialport(115200, 2, 1)
```

```
    Make ppp PPPProt(serial2)
```

```
End
```

```
To main
```

```
; this happens to use a dialup number for Plus Net
```

```
; the '9' is to dial out through a switchboard
```

```
Print To ppp, "AT&F0M0", CR, "AT&D1", CR,
```

```
"ATDT908451400103", CR
```

```
ppp.Name("myusername", "mysecretpassword")
```

```
repeat 3
```

```
    if ppp.Open
```

```
        break;
```

```
if ppp.Valid
```

```
[
```

```
; here you can put in some code
```

```
; that uses your internet connection
```

```
; ....
```

```
]
```

```
else
```

```
    Print "unable to open PPP connection", CR
```

```
End
```

## TCPProt

A raw TCP object type is available to enable the Venom programmer to implement proprietary or existing application protocols that use a TCP link.

A TCP connection can be viewed as a bidirectional conduit for reliably transmitting and receiving streams of bytes of any length. Any loss or corruption of data results in retransmission, so that while data integrity is guaranteed, timing is not.

See also [TCP/IP Networking](#)

### Summary of messages

**Make**  
**Close**  
**Debug**  
**Flush**  
**Free**  
**Get**  
**Open**  
**Put**  
**Queue**  
**Reset**  
**Status**  
**Timeout**  
**Print To**

### Creation

```
Make x TCPProt([ Int txbufsize [, Int rxbufsize]])
```

**txbufsize** : transmit buffer size, default 2048

**rxbufsize** : receive buffer size, default 2048

Creates a TCP object, allocating memory for this and the transmit and receive buffers.

### Buffer Sizes

Minimum 1024

Default 2048

Maximum 32768

The buffer sizes are optional. They should be specified as an exact power of 2. If not, they will be increased to the nearest next power of 2, e.g. specifying 5000 will allocate 8192 bytes.

Increasing these sizes can improve performance, especially if there is much delay in the physical link, but uses memory.

### Example

```
Make eth Ethernet ; Create a connection for TCP to use.
Make tcp TCPProt
Make s string(200)
If tcp.Open("www.venomcontrolsystems.co.uk", 80)
[
    Print To tcp, "GET /test.html HTTP/1.1",CR,
    "host: www.venomcontrolsystems.co.uk",CR, CR
    tcp.Flush ; send our request string without delay
    while tcp.get(s) > 0
        print s, CR
]
```

### Close

#### Close

Closes an existing connection made by the object. The TCP attempts to go through the usual sequence to close a connection. Note that closing a connection in TCP means that there is no more data to send. The connection may remain open in the receiving direction and continue to receive data. It is up to the remote end to initiate a close when it has no more data to send. See also [Reset](#).

### Debug

```
Debug(0) ⇐ File or 0
Debug(1) ⇐ File or 0
```

If you have file system, you can use it for logging information to help with debugging a TCP connection.

To enable logging, you should open a text file (type **char** in the filesystem **Open** message) and assign the file variable to **tcp.Debug(0)** or **tcp.Debug(1)**.

Management of files, including closing files, flushing the file system or emptying an old log file before use, is entirely up to the Venom programmer. By default, logging data is simply appended to the file after any data that previously existed.

It is legitimate to associate the same file with more than one tcp connection. Each connection created is given a numeric ID to help distinguish tcp connections logged in the same file.

- **Debug (0)** Controls logging of all TCP packets, one per line in a compact format.
- **Debug (1)** Controls logging of significant TCP events. Entries are created when connections are opened and closed, and for all exceptions such as resets or timeouts
- A value of 0 stops logging.

If you are having trouble with a TCP connection during program development, this feature may help you to understand what is going on, or if you send the logged results to us it may help us to provide technical support.

### Example

This sends an HTTP request for a small test file on our web site. If you run this on a VM2 connected to LAN with internet access this should work unchanged.

```
To init
  make fs Filesystem("ram", 100 * 1024)
  Make eth Ethernet
  make tcp TCPProt
  f := fs.open("tcp.log", char)      ; open log file
  f.empty                            ; fresh log each run
  tcp.Debug(0) := f                  ; Tell TCP to log packets
End

To main
  ; send a message to some device on the network that will send a
  ; response back and then close the connection
  if tcp.Open("www.venomcontrolsystems.co.uk", 80)
  [
    tcp.printf(<<<:
GET /test.html HTTP/1.1
Host: www.venomcontrolsystems.co.uk
Connection: Close

>>>)
    tcp.Flush                        ; don't wait 1/2 second for more
    tcp.close
    printf("response:\n")
    while tcp.queue >= 0             ; print response until TCP close
    repeat tcp.queue
      serial.put(tcp.get)
    printf("\n====\n")
  ]
  else
    printf("TCP open failed\n")
```

```

printf("Packet log:\n")
print f
f.close
End

```

If the program runs successfully the terminal output will look like this:

```
-->Run
```

```

response:
HTTP/1.1 200 OK
Date: Tue, 07 May 2019 11:00:49 GMT
Server: Apache
Vary: Accept-Encoding
Content-Length: 75
Connection: close
Content-Type: text/html; charset=UTF-8

```

```

<!DOCTYPE html>
<html><head></head><body>test.html web 2016</body></html>

```

```
=====
```

```

Packet log:
0.000 1 25653->80 s:4f00-4f01( 1) a:0000 w:2047 SYN
0.029 1 25653<-80 a:4f01 s:781a-781a ( 0) w:29200 SYN ACK
0.029 1 25653->80 s:4f01-4f01( 0) a:781b w:2047 ACK
0.033 1 25653->80 s:4f01-4f54( 83) a:781b w:2047 ACK PSH
0.061 1 25653<-80 a:4f54 s:781b-781b ( 0) w:29200 ACK
0.062 1 25653->80 s:4f54-4f54( 0) a:781b w:2047 ACK FIN
0.065 1 25653<-80 a:4f54 s:781b-7914 (249) w:29200 ACK PSH
0.065 1 25653<-80 a:4f54 s:7914-7914 ( 0) w:29200 ACK FIN
0.066 1 25653->80 s:4f55-4f55( 0) a:7915 w:1797 ACK
0.090 1 25653<-80 a:4f55 s:7915-7915 ( 0) w:29200 ACK
-->

```

## Die

### Die

If the Die message is sent to a TCP object, the memory used by it is freed and any open connection will be aborted.

If a TCP object is created as a local variable in a Venom procedure or function, it must be sent a Die message before the function exits, or there will be a memory leak and problems with the existing connection persisting.

The Die message could be useful to reclaim the memory used by TCP object no longer in use, or to change the buffer sizes between one connection and another.

## Flush

### Flush

Forces any locally buffered data to be sent and waits for it to be acknowledged. The packets are sent with the TCP PSH (Push function) flag set. This is intended for interactive use, where a response is expected from the other end once all the current data has been sent. The intended function of the PSH flag is to force the TCP object to send data immediately when it might otherwise have refrained from doing so because the resulting data segment would be small.

### Example

```
Make eth Ethernet ; Create a connection for TCP to use.
Make tcp TCProt
Make s string(200)
If tcp.Open("www.venomcontrolsystems.co.uk", 80)
[
    Print To tcp, "GET /test.html HTTP/1.1",CR,
    "host: www.venomcontrolsystems.co.uk",CR, CR
    tcp.Flush ; send our request string without delay
    while tcp.get(s) > 0
        print s, CR
]
```

## Free

**Free** ⇒ Int

Returns the free space in the transmit buffer.

If Free returns a negative value it means the connection is closed or was never opened.

## Get

The **Get** message reads characters or data bytes from a TCP object. There are three forms of the Get message:

- **single byte read**
- **string read**
- **array read**

### Single byte

**Get** ⇒ Int

Gets a single character or byte from the incoming data stream.



If the connection is open but there is no data, the TCP object will wait indefinitely for data to arrive or until the timeout limit has been reached, if a timeout was set.

With no parameter, the data is normally a byte value in the range 0 to 255.

A returned value of -1 means there is no connection e.g. it has been closed or reset.

A returned value of -2 means the timeout expired.

#### Example:

Receiving bytes until connection closes

```
While tcp.queue > 0
[
    c := tcp.get
    if c >= 0
        process_received_byte(tcp.get)
    else if c = -1
        Print "connection closed", CR
    else
        Print "timed out", CR
]
```

### String

**Get**(stringobj **s**) ⇒ Int

Reads a line of input terminated by CRLF from the TCP incoming data stream into the string object specified in the parameter.

The LF character is detected as the end of line marker, but neither CR nor LF is included in the returned string.

Transfer also stops if the capacity of the string object is reached first.

The returned value is the length of the received string if positive or zero. A negative returned value indicates a closed connection or timeout.

#### Example:

Receiving lines of text until connection closes

```
Make s string(200)
While tcp.get(s) >= 0
    process_received_line(s)
```

### Array

**Get**(arrayobj **a**, Int **nBytes**) ⇒ Int

Reads bytes from the TCP connection into an **Int 8** Array, specified in the first parameter.

The number of bytes to read is indicated by **nBytes**. If nBytes is larger than the array then an

Array Index error is thrown.

Get returns the number of bytes read, or -1 for a closed connection, or -2 if there is a timeout.

**Example:**

```
; In the init procedure:
Make dataArray Array(Int 8, 100, 0) ; 100-byte array to store incoming data

; In the main code:
returnCode := tcp.Get(dataArray, 50) ; Read 50 bytes into the array
if returnCode >= 0
[
    process(dataArray)
]
else
[
    ...
]
```

## Open

There are different forms of the Open message, for different purposes.

### Active Open

**Open**(host, **Int** port) ⇒ **Int**

host : Integer, string or text buffer representing the remote domain name or IP address to connect to

port : Integer representing remote port number to connect to.

Opens an active connection (i.e. the VM2 initiates a connection) to an address and port number on the net. Returns when established.

The return value is True (1) if the connection was made successfully, or False (zero) if it failed.

### Passive Open

**Open**(**Int** Port)

port is an integer port number.

When one parameter is specified, Open sets up the TCP object as a passive listener. The message returns immediately and the object continues to wait for an incoming connection.

A passive Open message does not return a value.

A run time error may result if the TCP object is in an incompatible state, e.g. if it already has a connection open, or the port number is invalid.

## Test for Open Connection

**Open**  $\Rightarrow$  Int

Return value:

-1            if closed/unused

0            if listening

IP address   when connected

With no parameters, the Open message is used as a test to find out whether a connection is open. After a passive open message has been sent, this will tell if an incoming connection has been received.

Always returns immediately with value according to state.

### Example 1 - Active Opening

```
If tcp.open("192.168.1.23", 123)
[
    ; send and receive data over TCP connection
]
Else
    Print "connection failed"
```

### Example 2 - Passive Listener

```
tcp.open(123)        ; listen on port 123
Forever
    If tcp.open
    [
        ; handle incoming connection
    ]
    Else
    [
        ; do something else
    ]
```

## Put

```
Put(Int)
Put(string)
Put(buffer)
Put(array a[, Int size])
```

Puts a byte, fixed string or entire buffer or array contents into the outgoing data stream.

The buffer or array can be of any integer or float data type. 16 or 32 bit integers and floats are sent as stored internally in Venom 2, least significant byte first. Using arrays or buffers gives fast data transfer rates. Note that sending 16 or 32 bit integers and float data this way is only useful if you know that the receiving system stores these values in the same way in memory.

"Buffer of Any" and arrays of strings are not currently supported, but you can [Print](#) an array of strings to TCP.

For arrays, an optional second parameter limits the size of data sent to TCP. The data is always sent from the beginning of the array.

## Queue

```
Queue ⇒ int
```

Return the number of bytes received and unread.

If Queue returns a negative value, it means the connection has been closed or reset. Note that when a receiving connection has been closed, Queue will continue to show a positive value while there is still unread data in the buffer, but will return negative once the last byte has been read.

## Reset

```
Reset
```

The Reset message forces a TCP object immediately into the closed state. Any packet received from the other end of a prior open connection with that TCP object will be rejected with a TCP RST (reset) packet.

All unread data in the receive buffer or untransmitted data in the transmit buffer is lost. The object can then be reopened for a new connection.

## Status

### Status

**Status**  $\Rightarrow$  **Int**

The value returned has the following meanings:

- 0 Normal connection.
- 1 Host unreachable – it was impossible to route packets to the remote destination.
- 2 Timeout when sending – no acknowledgement received after several retransmissions of the same data.
- 3 Connection attempt was refused.
- 4 DNS lookup failure when opening an active connection to a host specified by name.
- 5 TCP connection closed normally

This provides more information when the return value from a queue, Free or Get message indicates that the connection is closed.

Note that a value of 5 is returned any time after the remote end has initiated a close, but does not necessarily mean that the connection is fully closed. See [tcp.Valid](#) for a more detailed check on half-closed states.

## Timeout

**Timeout**  $\Leftrightarrow$  **Int**

Set a timeout value in milliseconds for the Get message. A value of 0 disables the timeout, which is the state when the TCP object is created.

This message also sets a maximum limit for acknowledgement timeouts when sending data. By default there can be a delay of over 3 minutes before a link is dropped because of lack of response. In some cases, like a local area network, it is reasonable to assume that lack of response within a few seconds indicates a non-responsive host; setting a lower timeout discovers this condition faster.

## Valid

**Valid**  $\Rightarrow$  **Int**

This returns a bitmapped value with the following bit meanings:

Bit 0 (1)	1 = TCP connection is open for receiving data, 0 = other end has closed connection
Bit 1	1 = TCP connection is open for sending data, 0 = this end has closed connection

(2)

## Explanation

When a TCP connection is first opened, data can be sent in either direction.

When one end closes the connection, it can no longer send data, but it can still receive data sent from the other end, and the connection is not fully closed until both ends have requested a TCP close.

This message provides a convenient means to determine that a connection is fully closed in case you want to open a new connection with the TCP object. This condition is met only when **tcp.Valid** = 0 and is exactly equivalent to **tcp.Queue** and **tcp.Free** both returning -1.

## Example 1

Determine when other end has closed connection

```
tcp.Open("example.com", 80)
... ; send data, receive data
tcp.Close

While tcp.Valid ; wait for other end to close connection
    Wait 1

tcp.Open("example2.com", 80) ; we may now open a new connection
```

## Example 2

Determine when listening connection has opened

```
tcp.Open(123) ; listen on Port 123

while tcp.Valid = 0 ; wait for incoming connection
    wait 1

... ; incoming connection
```

See also [tcp.Open](#) and [tcp.Status](#)

## PRINT TO

### Print To TCP object

The **Print** statement can be used to print anything to a TCP object as if the TCP object was a text buffer or serial port. The Keyword **CR** results in the sequence CR LF (13, 10) being sent to the TCP data stream, which is usually the correct line ending sequence in text-based protocols over TCP, such as SMTP, POP3 and HTTP.

Other **Print** keywords are not recognised and should be avoided.



## Timer

Timer objects are software countdown timers. Once started, a Timer object ‘runs’ for a given number of milliseconds, and then stops.

### Summary of messages

**Make**  
**Done**  
**Go**  
**Period**  
**Reset**  
**Time**  
**Print**

### Creation

**Make** <object> **Timer** (Int **period**)

A new timer object is created. Once the timer is started it will ‘run’ for period milliseconds before stopping. A timer is created in the stopped state.

For example:

```
Make tmr Timer(100000) ; a 100 second timer
```

⚠ The maximum value the period may be set to is the largest positive integer (~2billion), which is equivalent to ~24 days.

🔍 See also [Stopwatch](#), [RealTimeClock](#); [Wait](#), [Every](#)

📊 You can make as many timer objects as you require. Each one will take a small amount of memory.

### Done


**Done** ⇒ Int

Done returns [True](#) if the timer has stopped, and [False](#) if the Timer is running.

## Go

### Go

Starts the timer, effectively loading Period into Time.

 See also [Period](#), [Time](#)

## Period

**Period**  $\Leftrightarrow$  Int

Period holds the millisecond value that gets loaded into Time when the timer is started with Go, etc.

Period is an active variable.

▼ The maximum value Period may be set to is the largest integer (~2billion), which is equivalent to ~25 days.

⌚ All timing is done in milliseconds. The accuracy of the timing depends on the controller's crystal oscillator. See the VM2 Datasheet.

 See also [Creation](#), [Time](#), [Go](#), [Printing](#)

## Reset

### Reset

Reset immediately stops the timer, reducing Time to 0, and Done becomes True.

 See also [Creation](#), [Go](#), [Time](#)


## Time

**Time**  $\Leftrightarrow$  Int

Time is an active variable holding the countdown time: the time left before the timer stops. Time may be set – this allows you to alter the period before the timer stops. If the timer has stopped then setting Time non-zero will start it.

▼ The maximum value Time may be set to is the largest integer (~2billion), which is equivalent to ~25 days.

⌚ All timing is done in milliseconds. The accuracy of the timing depends on the VM2's crystal oscillator. See the VM2 Datasheet.

 See also [Creation](#), [Period](#), [Go](#), [Printing](#)



## Printing

```
Print <Timer> : Int f1
Print <Timer> : Int f1 : Int dp
Print <Timer> : Int f1 : Int dp : Int select
```

Prints the time or the period of the timer in different formats. **f1** gives the major format style to use:

<b>f1</b>	<b>Format</b>
:0	DD:HH:MM:SS. <i>sss</i>
:1	HH:MM:SS. <i>sss</i>
:2	MM:SS. <i>sss</i>
:3	SS. <i>sss</i>

<b>Key</b>
<b>D</b> is a day digit (0-24)
<b>H</b> is an hours digit (0-23)
<b>M</b> is a minutes digit (0-59)
<b>S</b> is a seconds digit (0-59)
<b>s</b> is a fraction of a second digit, specified by the second formatting parameter <b>dp</b> .

**dp** gives the number of decimal places to print the seconds value to. If **dp** is zero then no decimal point or fractional digits are printed.

**select** chooses whether the time printed should be the Time or the Period. If **select** is missing or zero, then the Time is printed, if **select** is 1 then the Period is printed.

*Tip: if you want to print out any number in the format used by Stopwatch or Timer, you can create a Timer object, set its Period to the value you want to print, and then print it, remembering to specify printing the Period.*

▼ **dp** has a maximum value of 3.

▼ If no format is supplied the Timer is printed in the form: [Timer: Period : Time].


## TouchScreen

The TouchScreen object interfaces to a TSC2003 Touchscreen Controller IC via the I2C Bus. The IC may be used to detect the position of a finger or stylus touching a resistive touchscreen.

Normally a touchscreen is placed in front of a graphics LCD, functioning as the input device in a Graphical User Interface. The TouchScreen object is able to convert the touch position on the screen into the coordinates used on the Graphics LCD.

Normally a graphical user interface (GUI) built around the Touchscreen object will make extensive use of the [Button](#) sub-object.

Please see the Code Snippets pages on our website for examples of how to use the Touchscreen and Button objects.

 This object requires a TSC2003 IC to be present on the I2C Bus.

 See also [GraphicsLCD](#).

### Summary of messages

**Make**  
**Adjust**  
**Asserted**  
**Button - create**  
**Button - get**  
**Count**  
**Event**  
**Key**  
**Mapping**  
**Remove**  
**Time**  
**Timeout**  
**Value**  
**XPos**  
**YPos**

### Creation

**Make** <object> **TouchScreen**(Int **type**, Int **Bus**, Int **I2C\_addr**)

**Type** gives the type of touchscreen hardware to be driven. Only type 0, the TSC2003, is currently supported.

**Bus** is the number of the I2C Bus: Must have the value 1 currently.

**I2C\_addr** is the address of the TSC2003 on the I2C Bus and must be in the range 144-150 (even numbers).

*The 'Venom' I2C address is obtained by doubling the 7-bit address usually given in a device datasheet.*

A touchscreen object will normally be used in conjunction with a [GraphicsLCD](#) object.

## Example

```
Make t Touchscreen(0, 1, 144)
```

## Adjust

```
Adjust(Int ReadWrite, GraphicsLCD lcd, SafeData safe,
Int address) ⇒ Int
```

Adjust is used to calibrate the touchscreen. There are two different forms of the message, depending on the value of the **ReadWrite** parameter:

Read Write	Action
0	<b>Read</b> calibration data out of a SafeData object, use it to calibrate the Touchscreen, and return a non-zero value if the calibration data was correctly validated by a checksum.
1	Perform a calibration, and <b>write</b> the calibration data into a SafeData object along with a checksum to validate it.

The other parameters are:

**lcd**: the [GraphicsLCD](#) device associated with the Touchscreen.

**safe**: the [SafeData](#) object to store the calibration data in. The calibration data takes **12 bytes** (six 16-bit values, including the checksum)

**address**: the address (within the SafeData object) of the start of the calibration data .

Adjust will draw crosses on the LCD for the operator to touch in order to calculate the calibration data, and will over-write each cross with a square box once a sufficiently long touch is detected.

The crosses and boxes are drawn in the LCD's current foreground and background colours.

### Example code

This will calibrate a touchscreen if the checksum for the stored calibration data isn't valid, or if the operator is touching the screen when this procedure is called.

```

Make lcd GraphicsLCD(1)
Make touch TouchScreen(0,1,144)
Make eeprom SafeData (1,1,162)
...
To calibrate
  ; Do we need to recalibrate?
  If touch.Asserted ; If operator touching screen...
  OrElse touch.Adjust(0, lcd, eeprom, 0) IsFalse ; ...Or Stored a
  [
    Print To lcd, FONT 1, CLS, "Touchscreen calibration. Touch th
    touch.Adjust(1, lcd, eeprom, 0)
    Print To lcd, CLS, "Calibration Done"
    Wait 500
  ]
End

```

### Asserted

**Asserted** ⇒ Int

Asserted returns True if the touchscreen is being touched.

When Asserted detects a touch it records the X,Y coordinates of the touch so that they may be read out using [XPos](#) and [YPos](#).

### Button - create

#### Creating buttons

```

Button
(
  Int Key, String Name,
  Int XPos, Int YPos, Int Width, Int Height,
  [Pointer Draw, [Int Active, [Any Element(0), ...]]]
) ⇒ Button

```

Buttons are created by sending a Button message to a Touchscreen with at least six parameters. There are also two optional parameters of defined type, and then an unlimited number of

optional parameters of any type.

*Note: when using the user-class based GUI Library, Buttons are only used to associated a GUI object with an active area on the touchscreen (defined by the Button's rectangular extent). Only the parameters XPos, YPos, Width, Height and Element are used.*

⚠ Currently there is a limit of 64 buttons, but this will be removed later if it proves too restrictive.

## Parameters

Name	Type	Description
<b>Key</b>	Int	The 'Key value' for the Button. It is usually used by your program to determine what action to take when the button is pressed. <i>Deciding exactly what the Key value is to represent is often the central decision to be made when designing a complex menu page.</i>
<b>Name</b>	String	Button name. Usually used to add a text label to the button when it is drawn.
<b>Xpos</b>	Int	Position of the left hand edge of the Button
<b>YPos</b>	Int	Position of the bottom edge of the Button
<b>Width</b>	Int	Width of the Button
<b>Height</b>	Int	Height of the Button

*Note: the coordinates and size of the Button are in Touchscreen coordinates, which are the pixel coordinates of the underlying GraphicsLCD if the Touchscreen has been calibrated.*

## Optional parameters

<b>Draw</b>	Pointer	Procedure pointer to Venom procedure that 'knows' how to draw this button. Different 'types' of button may be implemented by using different Draw procedures for each type, so giving a different drawn appearance and behaviour for each type.
<b>Active</b>	Int	This is usually used to signal to your code whether a button is 'active' or not. Usually, inactive buttons are drawn differently, and don't respond to touches, but behaviour is determined by your Venom code.
<b>Eleme</b>	Any	This is an unlimited set of extra values that are stored in the Button, and that

nt(n)	you can use in any way you wish.
-------	----------------------------------

Note that any of the Button parameters may be accessed by sending a message of the same name to the [Button object](#).

### Return value

The Button message returns a [Button object](#). It's not often necessary to use this value.

### Example

```
ts.Button(5, "OK", 10, 10, 50, 30, @draw_normal_button)
ts.Button(4, "Cancel", 10, 70, 50, 30, @draw_normal_button)
```

The code above will create two button objects, giving them names, as in the quoted strings, and key numbers 5 and 4, followed by the XY positions and extent (width and height) of each button, and then a pointer to the procedure which is to draw the buttons. Simple menus are built up from a list of buttons created like this.

More complex menus may use Venom code to automatically generate lists of buttons.

Sometimes it's useful to keep a reference to the Button object created; this example shows you one way to do that:

```
OK_button := ts.Button(5, "OK", 10, 10, 50, 30, @draw_normal_b
```

 Also see [Button - get](#) and [Find](#).

### Removing buttons

To remove all the buttons from a Touchscreen, use the [Remove](#) message.

### Button - get

#### Getting the Button associated with an event

Button events are sensed by regularly sending the [Event](#) message to a Touchscreen. If the Event message returns with an event code (E.g. *Button Down*), then the button associated with that event may be retrieved by sending the Button message with no parameters:

```
Button ⇒ Button
```

For example:

```
button_obj := ts.Button
```

#### Getting a Button using its index

You can get any Button object by accessing the list held by the Touchscreen object. You have

to use the button's 'index' to specify which button you want. The index is given by the value of [Count](#) just before the button is created, or *Count-1* just after the button was created, or may be calculated in some other way, since the index number increments from zero.

**Button**(Int **index**) ⇒ Button

For example:

**button\_obj** := **ts.Button**(0)

This gets button number 0 - the first button created, since buttons are numbered from 0.



Note: a button's index is *not* the same as it's 'Key value'. If you want to find a Button given it's Key value then use [Find](#).

## Count

**Count** ⇒ Int

Returns the number of Button objects currently held by the Touchscreen.

[Remove](#)

## Event

**Event** ⇒ Int

The event message is used to scan the Touchscreen and report any Button activity detected.

The table shows the events that may occur, and the values associated with them.

No Event	0
Button Down	1
Button Up	2

## Scan order

Buttons are scanned in *reverse order* : last created, first scanned. If Buttons overlap then the

first one that contains the coordinates of the touch point is returned.

To find the button that caused the event, call the [Button](#) message.

### Example

```
touch_event := touch.Event
```

### Find

*Not needed when using user-class based GUI Library.*

```
Find(Int Key_value) ⇒ Button
```

Find will look for the first Button with the given Key value and return it.

The Key value is the value of the first parameter supplied when the Button was [created](#).

If no Button with a matching Key value is found then Find returns Nil.

Note: Find is different to [getting a Button](#) using it's 'index' number.



If there is more than one Button with a matching Key value (not recommended) then only the first match will be found.

### Key

*This feature is no longer recommended for new designs.*

```
Key (Int x1, Int y1, Int dx, Int dy)
```

#### Key

The Key message allows an unlimited number of 'virtual keys' to be defined on the touchscreen.

Each key is a rectangle defined by the coordinate (x1, y1) & the sizes dx and dy - these are the same values that are used to define a [Box](#) or [TextBox](#).

Note that dx and dy may be negative.

Each time the Key message is called with four parameters, a new key is added. The keys are numbered from zero upwards in order of creation.

In this example four keys are created. We also draw button graphics on the the display, g, and



label the buttons 0 to 3.

 See also [TextBox](#).

```
#define KEYBD_X 10
#define KEYBD_Y 10
#define KEYSIZE 20
#define KEYSPPACING 25

To keys
  Local x:=KEYBD_X, y:=KEYBD_Y

  t.Key ;reset the key list.
  Repeat 4
  [
    g.TextBox(x , y , KEYSIZE , KEYSIZE , $100)
    Print To g, FONT 0, Centre, Index0
    t.Key(x , y , x+KEYSIZE , y+KEYSIZE)
    x := x + KEYSPPACING
  ]
End
```

## Scan order

When keys are scanned (using the Keypad object), the scan is done in **reverse** numerical order starting with the last key to be defined. A touch position is compared with each key in turn, and the first key to match is returned. This is important to consider if you define keys with overlapping areas. It can be useful to define overlapping keys. For example you may want the background to be a default 'key' that is detected if no others are.

## Removing keys

If you call the Key message with no parameters then all the keys are discarded, and new ones may be defined if desired.

```
t.Key ;reset the key list.
```

## Using the Keys

To use the keys as a keypad, create a keypad based on the touchscreen like this:

```
Make t TouchScreen (0, 1, 144)
Make kpd Keypad (t)
```


Then use the Keypad object in the normal way to detect key presses, etc.

*Usage Note: If you redefine a TouchScreen virtual keypad when, say, moving from menu*

*to menu, you should be aware of how the Get message for Keypad's InputBuffer functionality will behave, as it is currently implemented: when the old set of keys is removed from the TouchScreen object, but before the new set has been defined, the InputBuffer may detect no key being pressed. If the operator is still pressing the touch screen when the new keypad is defined, this will be interpreted as a new key press. One solution is to not change the virtual keys until the operator has stopped touching the screen.*

### X Y coordinates

When the touch screen is used as a virtual keypad, the X and Y coordinates are available, using the Value message, for each touch of a key so you can determine exactly where the touch was made.

 See also [Keypad](#)

### Mapping

*This feature is no longer recommended for new designs.  
Please see [Adjust](#).*

In each individual touchscreen assembly the touchscreen system has to be calibrated so that the X, Y coordinates it returns coincide with the screen coordinates of the graphics LCD. This calibration is handled automatically by the Mapping message.

The Mapping message has two different forms:

1. The first form is used to set up the calibration.
2. The second form allows the four calibration constants to be read out of, or written into, the TouchScreen object. The calibration data can in this way be stored for repeated use so the actual calibration need only be done once.

### Initial Calibration

*This form of the message is used to calibrate the touchscreen object:*

**Mapping (Int phase, Int xpixel, Int ypixel)**

The basic procedure to calibrate the TouchScreen object is:

1. Draw a cross near the bottom left corner of the LCD, at position (X1, Y1).
2. Send TouchScreen the message **Mapping (1, X1, Y1)** – i.e. using the same coordinates as the cross. The Mapping message will wait for a touch.
3. The user should touch the cross accurately

4. Sound a beeper, or otherwise let the user know the touch was detected.
5. Draw a 2nd cross near the top right corner of the LCD, at position (X2, Y2).
6. Send TouchScreen the message **Mapping (2, X2, Y2)**
7. The user should touch the 2nd cross accurately
8. The calibration constants are calculated and stored by the object. All subsequent readings of the touch position will be calibrated to screen coordinates.
9. The four values that represent the calibration may be accessed, using the second form of the Mapping message (discussed below), and preserved in non-volatile storage.
10. Sound a beeper, or otherwise let the user know the touch was detected.

Example code appears at the end of this page.

## Reading and Writing calibration data

**obj . Mapping (Int index) ⇔ Int**

When sent with one parameter, the Mapping message reads or writes the calibration constants. There are four constants (index is in the range 0 – 3), and each constant is a two-byte (16-bit) number.

The constants represent X-offset, Y-offset, X-scale, Y-scale.

When the Touchscreen object is first created these constants are given default values that result in the touchscreen reporting touches as the raw output values from it's 12-bit ADC - i.e. values in the range 0 - 4095.

## Storing a Calibration

The four calibration constants may be stored in, and subsequently retrieved from, [non-volatile storage](#), making it unnecessary to perform a calibration every time the system is turned on.

## Example code

```
;==== Touch screen calibration code =====
;
;Calibrate the touch screen and store the calibration data in EEPROM
; or read a previous calibration out of EEPROM.
;This procedure should be called at the start of the Main procedure
;Remember to turn the backlight on first!
;
;
; Parameters:
;=====
; recalibrate - if True then force a calibration. If false then c
```

```

; a calibration is necessary.
; touch - the Touchscreen object
; lcd - the GraphicsLCD object
; lcd_width - the pixel width of the LCD
; lcd_height - the pixel height of the LCD
; sounder - a sounder object that takes messages ON and OFF. Use
; safe - a Safedata object - to store calibration data.
; safe_addr - the byte address within 'safe' to start the calibra
;
; Data storage.
; This uses 5 x 16-bit values: 10 bytes in the SafeData object.

```

**To calibrate\_touchscreen(recalibrate, touch, lcd, lcd\_width, lcd\_height)**

```

; Define the crosses drawn on the screen.
#Define CAL_INSET 20
#Define CAL_CROSSSZ 10

safe.Address := safe_addr ; Reset the SafeData store.

;Check to see if a calibration data is missing or corrupted, or
;(If calibration data is OK then just load it into the Touchscreen)
If recalibrate IsFalse AndAlso safe.Checksum(safe_addr, safe_addr)
[
    Repeat 4
        touch.Mapping(Index0) := safe.Get(16)
    Return True
]

;Set up pen colour
lcd.Pen(0) := 0 ; black

lcd.TextBox ; reset textbox to whole screen

;Print a message to the lcd device.
Print To lcd
    , FONT 1,CLS,"Touch screen calibration"
    , FONT 0, CR, "Touch each cross accurately"

cal_draw_cross(lcd, CAL_INSET,CAL_INSET);Draw the first '+' - touch
lcd.Update ; Update the lcd - this may be redundant.
touch.Mapping(1, CAL_INSET,CAL_INSET) ; Initiate phase 1 of calibration
sounder.On Wait 100 sounder.Off ; beep to say it's done.

cal_draw_cross(lcd, lcd_width-CAL_INSET,lcd_height-CAL_INSET) ;

```

```

lcd.Update ; Update the lcd - this may be redundant.
touch.Mapping(2, lcd_width-CAL_INSET,lcd_height-CAL_INSET) ; In
sounder.On Wait 100 sounder.Off ; beep to say it's done.

;Store the calibration data in the SafeData object.
Repeat 4
[
    safe.Put(touch.Mapping(Index0),16)
]
safe.Put(safe.Checksum(safe_addr, safe_addr+8),16) ; Set a chec

Print To lcd, FONT 1, CLS, "Calibration Done"
lcd.Update ; Update the lcd - this may be redundant.
Wait 1000 ; wait so message above is visbile - this may be redu
End

;Draw a cross on the lcd, centred on (x,y)
To cal_draw_cross(lcd,x,y)
    lcd.Line(x-CAL_CROSSSZ,y, x+CAL_CROSSSZ, y)
    lcd.Line(x,y-CAL_CROSSSZ, x, y+CAL_CROSSSZ)
End

;==== End Touch screen calibration code =====

```

 See also [SafeData](#)

## Remove

### Remove

Remove removes all Buttons from the Touchscreen allowing a new set to be redefined. This is usually when you want to create a new menu page.

```

touch.Remove
touch.Button(5, "OK", ...)
touch.Button(9, "Cancel", ...)
...

```

## Re-create buttons in the same menu

**Remove**(Int continuous)

If the optional **continuous** parameter is non-zero then Remove allows the last *button press* to

be carried though from the old set of Buttons to the new set.

This means that a new button that contains the position of the last touch event will be set to [Asserted](#).

This is useful when you have a menu page that needs to re-create all its Buttons when one of its buttons is pressed, and where you will usually want to draw the newly created buttons as if the button that was pressed is still pressed.

```
touch.Remove(True)
```

## Time

```
Time ⇒ Int
```

```
Time(Int ind) ⇒ Int
```

Time returns the time (in mS) since the last 'real' (i.e. not an auto-repeat) Button Down [event](#).

```
t := touch.Time
```

Alternatively, if you give it a parameter of value 1, it will return the number of *auto-repetitions* (including the initial touch) since the initial touch.

```
n := touch.Time(1)
```

Both of these may be useful when designing the [auto-repeat](#) behaviour of a menu.

## Timeout

```
Timeout(Int delay, Int period)
```

Timeout is used to implement an 'Auto-repeat' function for touchscreen buttons. If a Button is held down for longer than **delay** mS, then [Event](#) will be forced to return *Button Down* even though the Button has not been released and re-touched. If the Button remains held down then further **Button Down** events will be forced every **period** mS.

You can implement different auto-repeat behaviour for each button on a menu page by choosing whether to send the Timeout message, or what values of **delay** and **period** to use.

## Example

The following code was extracted from some QWERTY keyboard menu code. Notice the different auto-repeat behaviour for the Delete, Shift and Letter buttons.

```
Select Case key_value
Case -1 [] ; no key, no action.
Case 0 ;OK key
[
    Return 0
]
Case 1 ;delete key
[
    delete_character
    touch.Timeout(400,100) ; Delete auto-repeat
]
Case 2 ; shift key
[
    do_shift
    ; No auto-repeat
]
Case Else ; Letter keys.
[
    process_letter(key_value)
    touch.Timeout(400,200) ; Typing auto-repeat
]
```

## More details

If you set **delay** to zero then auto-repetition will not start.

If you set **period** to zero (0) then auto-repetition will not continue.

If you don't call **Timeout** then auto-repetition will not start (or continue).

The [Time](#) message may be useful when implementing more sophisticated auto-repeat behaviour.

## Value

**Value** ⇔ Int

The value message is associated with the sensitivity of the Touchscreen.

- When read, Value returns a number that gives some kind of indication of the pressure of

a touch. Because this value has not been processed, and is dependent on the XY position of the touch, this is not very useful other than to aid in setting Value, as below.

- When set, Value sets the minimum value of touch 'pressure' that the touchscreen object will register as a touch. This value may be increased to reduce the sensitivity of the touchscreen. If it is increased too much it will start to limit the area of the screen that will detect touches. The default value is around 5

### Example

```
ts.Value := 10
```

### XPos

**XPos**  $\Rightarrow$  Int

Returns the X ordinate of a touch. This will be in Graphics LCD coordinates if the touchscreen is [calibrated](#) to the graphics LCD.

The X and Y coordinates are set each time either [Asserted](#) or [Event](#) detects a touch on the touchscreen.

This example draws a line to each new touch on the screen:

```
To lines
  Every 50
  [
    If t.Asserted
    [
      w.Line(t.XPos, t.YPos)
      g.Update ; update GraphicsLCD object
    ]
  ]
End
```

### YPos

**YPos**  $\Rightarrow$  Int

Returns the Y ordinate of a touch. This will be in Graphics LCD coordinates if the touchscreen is [calibrated](#) to the graphics LCD.

The X and Y coordinates are set each time either [Asserted](#) or [Event](#) detects a touch on the touchscreen.



See [here](#) for example code.

## TouchScreen: Button

### Use with user-class based GUI library

Note: Since the introduction of the **user Class-based GUI library**, much of the functionality of the **Touchscreen:Button** object has become redundant.

The GUI Library still uses the **Touchscreen:Button** object, but only uses part of its capability: to define rectangular 'active' areas on the display, and to associate these active areas with user-defined objects in the GUI system. The only message used is [Element](#).

### Easiest to start with template code

Because of the relative complexity of GUI systems, it's probably easiest to adapt an existing GUI template project rather than start writing code from scratch. Please see the Code snippets page on our website for template code examples.

### Previous description of Button

*This is the previous description of the Button object:*

The Button object is a powerful tool for creating Graphical User Interfaces (GUIs).

Button is a sub-object of Touchscreen. You cannot Make a Button - the only way to create a Button is to send the 'create' version of the [Button](#) message to a Touchscreen object.

A typical GUI consists of a set of *menu pages*, and each menu page consists of a set of *Buttons*.

Each Button object holds data that help your program to know how to draw it in different states of being pressed or active, and what action to take when it is pressed. In fact a Button object doesn't do much more than hold data values in a convenient form.

### Summary of messages

**Creation** (Button,in Touchscreen object)  
**Active**  
**Asserted**  
**Draw**  
**Element**  
**Height**  
**Name**  
**Key**  
**Width**  
**XPos**  
**YPos**  
**Print**

## Active

*Not needed when using user-class based GUI Library.*

**Active**  $\Leftrightarrow$  Int

The value of Active may be used in your Venom code to control how a button is displayed, and whether a press on that button is acted on.

For example, you may decide to draw inactive buttons in grey colours, and block them from taking any action or making any sound if they are pressed.

Active is initialised to the value of the optional parameter **Active**, supplied when the button is [created](#), or to the value 1 by default.

You can set it to any value at any time.

## Asserted

*Not needed when using user-class based GUI Library.*

**Asserted**  $\Rightarrow$  Int

Asserted returns whether a particular button is pressed or not. It does not actually scan the touchscreen hardware, but instead uses information gathered the last time the [Event](#) message was sent to the parent TouchScreen object. Only one Button can ever be Asserted at any one time - and this will be the Button associated with the last [Event](#).

It is also possible for no (zero) Buttons to be Asserted at a particular time.

Asserted is useful when you need to draw a button differently depending on whether it is pressed or not.

## Draw

*Not needed when using user-class based GUI Library.*

**Draw**  $\Rightarrow$  Pointer

The Draw message returns a procedure pointer - the one that was supplied to the Button when it was [created](#).

It holds a pointer to a Venom procedure that 'knows' how to draw a particular 'type' of button in your application.

You will have to write this procedure, or use an example from the code snippets on our website.

## Example

This code gets a procedure pointer from a button object and then calls the procedure, sending it one parameter - the button object.

This is almost always the way that the Draw message will be used. It's a complicated Venom statement - but you probably don't need to understand it fully - just use it as it is shown here.

```
[(!button_obj.Draw) (button_obj)] ; Draw the Button.
```

You can also usefully write a procedure to draw *all* the Buttons held by a Touchscreen object:

```
To draw_all_buttons
  Local button_obj
  Repeat touch.Count ; For every Button held by touch...
  [
    button_obj := touch.Button(Index0) ; Get the Nth
    Button in the list
    [(!button_obj.Draw) (button_obj)] ; Call the Venom
    procedure to draw it.
  ]
End
```

## Element

**Element**(Int **index**)  $\Leftrightarrow$  Any

<Button>.(Int **index**)  $\Leftrightarrow$  Any

*Note: when using the user-class based GUI Library, Element is used to hold a reference to the GUI object associated with the area defined by the Button's rectangular extent.*

Element allows you to access any user-defined values in the Button object. These user-defined values are set up when the button is first [created](#) by appending extra parameters to the parameter list.

Element will throw a runtime error if you try to access values that haven't been defined.

These extra values might be used to store alternative text for the button, font numbers, bitmap images or colours - to customise each button on a menu page. Element is writeable, so these values can be changed whenever needed.

### Example

```
OK_button := ts.Button(5,"OK", 10, 10, 50, 30, @draw_normal_button)
```

Here we've added a couple of user-defined values to the normal parameter list: 5 and "Not OK".

It's up to you how you use these in your application, but this code shows how you might access them.

```
-->Print OK_button.Element(0)
      5-->
-->Print OK_button.Element(1)
      Not OK-->
```

Note we used the 'longhand' version of the Element message above. The shorthand version would look like this:

```
-->Print OK_button.(0)
      5-->
```

### Height

*Not needed when using user-class based GUI Library.*

**Height** ⇒ Int

Returns the height of the Button object, set when the Button was first [created](#). This is used in drawing the button, typically to define the height of a [TextBox](#).

## Name

*Not needed when using user-class based GUI Library.*

**Name** ⇒ String

Name is an active variable initialised by the Name parameter when the Button is [created](#). You can change a Button's name at any time.

Name is usually used to hold text that labels a button when it is drawn.

## Example

```
To draw_simple_button(b)
    lcd.Pen ; reset to default colours.
    lcd.Format(0) := 1 ; Set 'word wrapping'.
    lcd.TextBox(b.XPos,b.YPos,b.Width,b.Height,1,0) ; Draw a
    simple button
    Print To lcd, Centre, b.Name
End
```

## Key

*Not needed when using user-class based GUI Library.*

**Key** ⇒ Int

Key returns the 'Key value' of a button. This is initialised to by the **Key** parameter when a button is first [created](#).

Key is usually used to hold a number that identifies what action the button is associated with. It is often used as the control value in a **Select Case** construction.

*Deciding exactly what the Key value is to represent is often the central decision to be made when designing a complex menu page.*

## Width

*Not needed when using user-class based GUI Library.*

**Width** ⇒ Int

Returns the width of a Button object, set when the Button was first [created](#). This is used in drawing the button, typically to define the width of a [TextBox](#).

## XPos

*Not needed when using user-class based GUI Library.*

**XPos**  $\Rightarrow$  Int

Returns the X position of a Button object, set when the Button was first [created](#). This is the horizontal position of the left hand edge of the Button.

This is used in drawing the button, typically to define the X position of a [TextBox](#).

## YPos

*Not needed when using user-class based GUI Library.*

**YPos**  $\Rightarrow$  Int

Returns the Y position of a Button object, set when the Button was first [created](#). This is the vertical position of the bottom edge of the Button.

This is used in drawing the button, typically to define the Y position of a [TextBox](#).

## Print

*Not needed when using user-class based GUI Library.*

**Print** <Button>

A button object will print out some of the values of it's stored data.

The format is not yet fixed.

## Example

```
-->Print button_obj, CR  
[Button: 5 'OK' (10,10) (50,30) @?? 1]  
-->
```

## WiFiLink

The WiFiLink object gives the VM2's TPC/IP networking system access to a WiFi network by controlling a WiFi module through a serial port.

Any module running Gainspan's IP2WiFi application firmware is supported. So far the WiFiLink object has been tested with Gainspan GS2011 and GS2100 modules.

In general, the behaviour and interface of the WiFiLink has many features in common with the Ethernet interface and uses similar syntax and naming conventions.

### WiFi Modes

The WiFiLink can be set up as an infrastructure station (connecting to an existing wireless LAN)

or as a limited access point (creating a wireless LAN for others to connect to).

### Security

WPA2 security is enabled by default. The WiFiLink object provides simple passphrase management.

### Scanning for Access Points

The WiFiLink can be made to produce a list of locally detected access points to assist with choosing a network.

### Serial Interface and Speed

The serial port is run at 921600bps, the highest speed supported by the Gainspan module for serial access.

Hardware flow control is needed, and configured in the serial port and the WiFi module, so RTS and CTS must be connected.

### Glossary of Wireless Networking Terms

<b>Access point</b>	A WiFi device configured to create and control a network, and which acts as a hub through which all data is passed.
<b>Station</b>	A WiFi device configured to connect to an existing network by associating with an access point.
<b>SSID</b>	"Service Set Identifier", meaning the name of a wireless network, used to define which one we are connecting to when there are several within range.
<b>Passphrase</b>	A short string of text, used as the basis for generating encryption (security) keys. If encryption is enabled, the passphrase is set at the access point, and any client must know the passphrase to be able to connect to the access point and use the network.
<b>DHCP</b>	(also used in wired Ethernet) stands for Dynamic Host Configuration Protocol. It is the commonest method used by computers to set up network connections automatically, and relies on the network having a DHCP server that manages IP addresses, and knows the IP address of a DNS server and gateways if they exist.
<b>DNS</b>	Domain Name Service: The mechanism that translates domain names like <code>www.google.com</code> into numeric IP addresses.

### Summary of messages

**Make**  
**Address**  
**Connect**  
**Debug**  
**Find**  
**Protect**  
**Status**  
**Value**  
**Print**

## Creation

```
Make wifi WiFiLink(Int port [, addr/hostname[, dns[, gateway]]])
```

Parameter	Type	Default Value	Description
<b>port</b>	int	-	Serial Port Number. You do not have to create a serial port object; the WiFiLink object will set up the selected port automatically.
<b>addr/hostname</b>	int or string	<i>No hostname, use DHCP</i>	If numeric or a string that looks like a valid IP address, sets IP address of WiFi interface. Any other string is assumed to be a hostname: DHCP is used to obtain an IP address and DNS and gateway information, and the hostname is sent to the DHCP server in case a local DNS service can use it.
<b>dns</b>	int or string	<i>0 (no DNS)</i>	IP address of DNS server, if fixed IP address was set by the second parameter. 0 means no DNS is available.
<b>gateway</b>	int or string	<i>0 (no gateway)</i>	IP address of default gateway to networks outside the LAN, if fixed IP address was set by the second parameter. 0 means no gateway is available.

Where IP addresses are supplied as a string, they are in the conventional "dotted-quad" notation used for IPV4 addresses, e.g. "192.168.1.200"



## Examples

(Also see [Address](#) and [Connect](#))

### Simple Station Mode

<code>Make wifi WiFiLink(5)</code>	Serial port 5 Suitable for connecting in Station mode Use DHCP to get network information
<code>wifi.Connect(0, "MySSID"</code>	Connect to an existing WiFi network

This is suitable for use when connecting to an existing access point, e.g. office or factory network, and where the VM2 will be initiating data connections but not receiving them.

### Station Mode with Hostname

<code>Make wifi WiFiLink(5, "c</code>	Serial port 5 Reachable on the network by the name "controller1" Suitable for connecting in Station mode Use DHCP to get network information
<code>wifi.Connect(0, "MySSID"</code>	Connect to an existing WiFi network

This is suitable for use when connecting to an existing access point, e.g. office or factory network, and where the VM2 may be receiving incoming connections, e.g. running a web server which can be accessed with the host name "controller1". This will only work automatically if the network has linked DHCP and local DNS servers.

### Creating an Access Point

<code>Make wifi WiFiLink(5, "192.1</code>	Serial port 5 Has fixed IP address 192.168.1.1 Suitable for connecting in Station mode Use DHCP to get network information
<code>wifi.Address ('S', "192.168.</code> <code>"192.168.1.199", "control</code>	Set up VM2 as DHCP server, with hostname "controller1"
<code>wifi.Connect(2, "MySSID", "M</code>	Create a Wireless network using channel 6

Another device can now associate with the network created by the VM2, and can reach the VM2 itself by the hostname "controller1". For example, a smartphone or laptop could use its web browser to communicate with a web server based application on the VM2.

### Station Mode - Fixed IP

<code>Make Wifi WiFiLink(5, "192.1</code>	Serial port 5
---	---------------

<code>"192.168.1.10", "192.168.1.1"</code>	Set IP address, DNS server and default gateway
<code>wifi.Connect(0, "MySSID", "MyPassphrase")</code>	Connect to an existing WiFi network

"Fixed IP" networking like this might be useful if the VM2 needs to receive incoming connections (e.g. it's running a web server application) and the network does not have a linked DNS and DHCP servers. The IP address would have to be manually selected to be in the local network IP range but not in the DHCP pool (if any exists), and the VM2 could then be reached by its IP address. The correct DNS and Gateway addresses to use, if needed, must be found out from the local network configuration. They won't be needed if the VM2 is only using the local network.

## Address

**Address**(Int **type**, various)

See the Ethernet [Address](#) message, which has identical functionality and in fact uses the shared code for both wired ethernet and WiFi.

Note that the WiFiLink object does not currently support Multicast operation.

## Connect

### Opening and Closing a Previously Used Connection

**Connect**(0) ⇒ Int

**Connect**(1) ⇒ Int

Returned value: 1 on success, 0 on failure.

The first two forms of the **Connect** message simply associate (parameter = 1) and disassociate (parameter = 0) the Wifi object from a previously associated access point, or enable and disable operation if the WiFi object is configured as an access point.

### Associating with an Access Point by Name

**Connect**(0, Str **SSID**, Str **passphrase**[, Int **channel**) ⇒ Int

Configures the mode, SSID and Passphrase in the object and attempts to associate with the named access point.

If a channel is specified, only that channel will be used.

Returns 1 on success, 0 on failure

## Creating an Access Point

```
Connect (2, Str SSID, Str passphrase [, Int channel) ⇒  
Int
```

Configures the mode, SSID and Passphrase in the object and creates an access point.

If a channel is not specified, the module will choose a channel to use.

The passphrase must be between 8 and 63 characters long.

Returns 1 on success, 0 on failure

## Associating With a Found Access Point

```
Connect (AccessPoint ap, Str passphrase) ⇒ Int
```

Uses an **AccessPoint** object returned by the **Find** message to connect to the corresponding access point. In cases where the same SSID is available on more than one access point, channel, this will request a connection to the one whose MAC address and channel match those contained in the **AccessPoint** object.

Returns 1 on success, 0 on failure

## Debug

```
Debug (Int sel [, param])
```

Allows various debugging options.

sel	param	Description
0	0 or 1	Turn on (1) or off(0) packet debug, in which a summary of every packet sent and received is sent to the serial terminal, one per line.
0	value > 1	Selective packet debugging. The parameter can be any combination of these values, each using one binary bit of a 16 bit integer: \$0002 ARP           Address resolution: IP to MAC addresses \$0004 UDP           User Datagram Protocol packets (IP) \$0008 IP            Internet Protocol packets \$0010 TCP           All TCP packets (IP) \$0020 IPV6          Not supported by Venom

		\$0040 IPX Not used by Venom \$0080 ICMP "Ping" as in <a href="#">IP.Time</a> message \$0100 IGMP Group Management - not used by Venom \$0200 DNS Domain Name Resolution (UDP, IP) \$0400 HIP Host Identification - not used by Venom \$0800 DHCP Dynamic Host Configuration Protocol (see <a href="#">DHCP</a> ) \$8000 Unknown Anything else: show numeric protocol identifier.
1	0 or 1	Turn on (1) or off (0) I/O debug. I/O debug shows all command interaction with the module ("AT..." commands and their responses) with the exception of commands sent periodically to check the status of the link if it has been silent for a while.
2	"AT" command	Send command to module and display results on terminal (command is a string usually starting with "AT")
3	-	Show ARP table (association of MAC addresses with IP addresses.)

Of the listed debug actions, **Debug (0, n)** (or **Debug (0) := n**) is the only one likely to be useful in tracing problems.

If you have a copy of the Gainspan IP2Wifi Application Programmer's Guide you can find commands to control features like power saving, but use of these is not supported by Venom.

## ErrorAction

**ErrorAction**  $\Leftarrow$  Int

Assigning a value of 1 to **ErrorAction** suppresses a run time error when a static IP address assignment results in an address conflict. It is effectively a promise that the condition will instead be checked by a [Status](#) message.

## Find

### Make a List of available Access Points

**Find**(Buffer **buf**)  $\Rightarrow$  Int

Scans for access points within range.

Returns the number of access points found.

**buf** must be created as **Buffer (Any)**.

On return **buf** contains a list of [AccessPoint](#) objects.

## Find The Best Access Point for a known SSID

**Find**(Str **SSID**) ⇒ AccessPoint or Nil

Scans for access points within range, and then selects one with matching SSID. If more than one access point matches the SSID, the one with highest signal strength is chosen.

The returned value is an **AccessPoint** object, or **nil** if no suitable access point found.

## Protect

**Protect**([Int **mode**]) ⇔ Int

Gets or sets the Security/encryption protocol for the WiFi connection.

The current value is returned, and can optionally be set by supplying it as a parameter.

When used as an Access Point, the WiFiLink object uses WPA2-PSK by default.

## Security Modes

Value	Meaning	Description
0	Auto	Client will use the best protocol supported by both itself and the access point.
1	None	No encryption is used at all. Not recommended.
4	WEP	The original Wireless security protocol, now deprecated because it is very insecure. Not supported by this software.
8	WPA-PSK	WPA is widely used and a huge improvement on WEP. WPA2 is a further enhancement to WPA and available on most new equipment.*
16	WPA2-PSK	PSK = Pre Shared Key, meaning the security passphrase must be configured separately into the client and the access point. This is sometimes described as "WPA Personal" or "WPA2 Personal"
32	WPA Enterprise	Here, "Enterprise" refers to a different mechanism for setting up encryption keys using a central authentication server. It is only used in large networks.
64	WPA2 Enterprise	

\*WPA and WPA2 can be configured to use TKIP or AES encryption. AES is the more modern

encryption scheme, more secure and in tests we have also found it to give better performance and reliability with the hardware we have used.

#### Example: Setting Up an Open Network

To override the default security mode in an access point, set **Protect** to a different value before sending the **Connect** message, .g. to set up the VM2 as an open access point, use code as shown below. Note that a dummy passphrase must be used because

```
wifi.Protect(1)
```

```
wifi.Connect(2, "myssid", "dummyspassphrase")
```

### Status

**Status**  $\Rightarrow$  Int

Returns 0 if the Wifi module is associated with an access point or has been activated as an access point and is correctly configured and working properly.

(the same condition as when [Valid](#) returns true)

If there is not a usable connection, the value indicates what is wrong.

Status value	Meaning
0	Normal operation
1	Not associated
2	DHCP failure
3	IP address conflict

### Valid

**Valid**  $\Rightarrow$  Int

Returns 1 if the Wifi module is associated with an access point or has been activated as an access point and has been properly configured, else 0.

## Value

**Value**  $\Rightarrow$  Int

Returns the Received Signal Strength on the currently associated (connected) network. It is a negative number expressing the signal strength in dBm and has a typical range from -30 (strong) to -100 (too weak to use).

### Example

```
-->Make wifi WifiLink(5)
-->wifi.Connect(0, "testwap", "ZZyg2k196P0")
-->print wifi.value,CR
      -61
-->
```

## Print

**Print wifi** [: Int option]

Without a colon option, this lists several pieces of information about the WiFi interface in the following format.

```
WiFiLink using serial port 5
wifi0:
MAC   = 20:f8:5e:c1:ea:a5
IP    = 172.16.1.222
DNS   = 172.16.1.148
GW    = 172.16.1.199
host  = <no hostname>
mode: STATION security: WPA2-PSK
SSID: testwap
Passphrase: sn678K45tXpu
+WSTATUS result:
MODE:0 CHANNEL:2 SSID:"testwap"
BSSID:4c:e6:76:e0:37:04 SECURITY:WPA2-PERSONAL
```

## Colon Options

These select individual items from this list:

Option	Value printed
0	full summary as above

1	MAC address (48 bit Ethernet address)
2	IP address
3	DNS server address
4	Gateway address
5	Hostname[.domain]
6	SSID (network name)
7	Passphrase
8	Channel number, 0 if not known.
9	station <b>or</b> access point
10	associated <b>or</b> not associated

## AccessPoint

The AccessPoint object stores information about a remote WiFi Access point. It can only be created from a WiFiLink object (using the [Find](#) message) and its main uses are

- to display information to enable manual selection of an access point.
- to specify an access point in the WiFiLink [Connect](#) message.

The data stored in an AccessPoint object can be accessed in several ways.

**Creation**

**Channel**

**Compare**

**Key**

**Name**

**Value**

**Protect**

**Print**

## Channel

**Channel**  $\Rightarrow$  Int

This is a value from 1 to 14 representing a Wireless channel mapped to a centre frequency in the 802.11 standard 2.4GHz WiFi spectrum.

### Note on 802.11 Channel Use

When setting up a network in an environment where there are several other wireless networks, it



is beneficial to choose a channel not used by others, and since numerical adjacent channels can also interfere it is better still to keep as far away from other channels as possible.

In some parts of the world there are restrictions on the use of channels 12-14.

See [https://en.wikipedia.org/wiki/List\\_of\\_WLAN\\_channels](https://en.wikipedia.org/wiki/List_of_WLAN_channels) for more detailed information.

### Creation

An **AccessPoint** object can only be created from the WiFiLink **Find** message which makes the WiFi module perform a radio scan for local access points.

The Find message is supplied with a **Buffer (Any)** object, which is filled with **AccessPoint** objects, or an SSID string in which case it will return a single **AccessPoint** object describing the access point with the highest signal strength whose SSID matches that supplied.

### Compare

**Compare** (various **x** [, Int **uncased** [, Int **type**]]) ⇒ Int

Compares the SSID or signal strength of an **AccessPoint** object with either a given value or the same property of another **AccessPoint** object.

Parameter	Type	Default	Description
<b>x</b>	AccessPoint	-	Compare with another AccessPoint
	Int	-	Compare signal strength with this value
	string	-	Compare SSID with this value
<b>uncased</b>	int	0	For SSID compare, ignore case if nonzero
<b>type</b>	int	0	0 : compare SSID 1 : compare signal strength

These variations of the compare message enable a buffer of **AccessPoint** objects (as created by the **WiFiLink.Find** message) to be sorted by SSID or signal strength, or searched with a **Find** message. See **Buffer.Sort** and **Buffer.Find** for more information about how this works with object **Compare** messages.

### Examples of use with Sort

```
-->Make wifi WiFiLink(5)
-->Make aplist Buffer(Any)
```

```

-->wifi.Find(aplist)
-->aplist.Sort(2)      ; sort in alphabetical order of SSID
-->print aplist
18:1e:78:2b:d0:54    -93dB Ch11 WPA2-PSK "BTHub4-ZK36"
18:1e:78:2b:d0:57    -88dB Ch11 OPEN "BTWifi-with-FON"
18:1e:78:2b:d0:59    -92dB Ch11 WPA2-PSK "BTWifi-X"
e0:46:9a:0f:f8:6e    -74dB Ch 6 WPA2-PSK "Micro-Robotics"
02:7b:ef:45:d5:de    -75dB Ch10 WPA-PSK "Prospect Research"
4c:e6:76:e0:37:04    -71dB Ch 2 WPA2-PSK "testwap"
20:c9:d0:1d:4b:05    -97dB Ch11 WPA2-PSK "Zippin Pippin Guest"
-->
-->aplist.sort(1, 1)   ; sort in descending order of signal stren
-->print aplist
4c:e6:76:e0:37:04    -71dB Ch 2 WPA2-PSK "testwap"
e0:46:9a:0f:f8:6e    -74dB Ch 6 WPA2-PSK "Micro-Robotics"
02:7b:ef:45:d5:de    -75dB Ch10 WPA-PSK "Prospect Research"
18:1e:78:2b:d0:57    -88dB Ch11 OPEN "BTWifi-with-FON"
18:1e:78:2b:d0:59    -92dB Ch11 WPA2-PSK "BTWifi-X"
18:1e:78:2b:d0:54    -93dB Ch11 WPA2-PSK "BTHub4-ZK36"
20:c9:d0:1d:4b:05    -97dB Ch11 WPA2-PSK "Zippin Pippin Guest"

```

#### Notes:

In the first example, `Sort(2)` specifies a case-insensitive sort, which is what you probably want for SSID names.

In the second example, the first parameter of `Sort(1,1)` is set to 1 for a descending order, and the second parameter is set to 1 to request sorting on signal strength.

#### Key

**Key**  $\leftarrow$  str

Sets the passphrase for the access point from any string type.

The **AcessPoint** object may then be passed to a **WiFiLink** [Connect](#) message to associate with the specified AP.

#### Name

**Name**(str ssid)

This messages copies the AcessPoint's SSID to **ssid** which must must be a string variable.

**Value**

**Value**  $\Rightarrow$  Int

Returns the signal strength of the access point. It is a negative number expressing the signal strength in dBm and has a typical range from -30 (strong) to -100 (too weak to use).

**Protect**

**Protect**([Int mode])  $\Rightarrow$  Int

Gets the security (encryption) protocol for the access point.

**Security Modes**

Value	Meaning	Description
0	Auto	Client will use the best protocol supported by both itself and the access point.
1	None	No encryption is used at all.
4	WEP	The original Wireless security protocol, now deprecated because it is very insecure.
8	WPA-PSK	WPA is widely used and a huge improvement on WEP. WPA2 is a further enhancement to WPA and available on most new equipment.
16	WPA2-PSK	PSK = Pre Shared Key, meaning the security passphrase must be configured separately into the client and the access point.
32	WPA Enterprise	Here, "Enterprise" refers to a different mechanism for setting up encryption keys using a central authentication server. It is only used in large networks.
64	WPA2 Enterprise	

**Print**

**Print ap**

Prints a 1 line summary of the **AcessPoint**'s data, e.g.

```
-->make wifi wifilink(5)
-->ap := wifi.find("Micro-Robotics")
-->print ap,cr
e0:46:9a:0f:f8:6e  -67dB Ch 6 WPA2-PSK "Micro-Robotics"
```

-->

## Print Modifiers (Colon Operators)

```
Print ap:n
```

```
printf("%no", ap)
```

These enable selected pieces of information about the AccessPoint object to be printed.

n	Prints	Example
0	Everything (same as no modifier)	e0:46:9a:0f:f8:6e -67dB Ch 6 WPA2-PSK "Micro-Robotics"
1	SSID	Micro-Robotics
2	Signal Strength	-67dB
3	Channel	6
4	Security	WPA2-PSK
5	MAC address	e0:46:9a:0f:f8:6e

## XMODEMLink

XMODEM is a very simple and widely supported file transfer protocol for use on direct serial links. It does not use TCP, IP or PPP. It is useful for transferring files between a VM2 and a host computer such as a PC using a terminal program like Tera Term Pro or Hyperterminal, or a UNIX or Linux system using rx and tx (rz and sz with the -X option).

### File sizes

XODEM transfers files in fixed size blocks of 128 bytes. If a file is not an exact multiple of 128 bytes long, the last block is padded with data to make it up to 128 bytes. The VM2 XMODEM protocol uses zero (ASCII NUL) for this purpose when sending a file. Some implementations use Ctrl/Z (chr 26 or ASCII SUB) as padding for text files. The XMODEM protocol itself has no means of specifying exact file length.

### File Names

XMODEM does not transfer any file name or data type information. The program or user must supply a file name and determine the type if the destination or source is a file.

## Checksums, CRC and YMODEM

There are numerous extensions to XMODEM, notably XMODEM-CRC which uses a 16 bit CRC instead of an 8 bit checksum for better error detection, and variations with larger packet sizes and the ability to transfer file names and perform batch transfers. This implementation attempts to use the more robust CRC method, dropping back automatically to checksum mode if the other end of the link does not support CRC. This behaviour can be controlled by an optional Make parameter

## Using XMODEM with the VM2 File System

Note that the File objects can use XMODEM directly to send or receive a VM2 file through a serial port. In order to do this, it is NOT necessary to create an XMODEM protocol object.

 See [File.Put](#), [File.Get](#)

## Summary of messages

**Make**  
**Flush**  
**Free**  
**Get**  
**Put**  
**Queue**  
**Reset**

## Creation

```
Make <object> XMODEMLink(Int portnumber [, Int  
crcmode) )
```

**portnumber** A serial port number. The serial port must be created first and set set for 8 bits, no parity.

**crcmode** 0 - only use the older checksum method.  
1 (default) - try CRC mode when receiving, fall back to checksum mode if no response  
2 - only accept CRC mode when receiving.

The link is created in an initial state where it is neither transmitting nor receiving. The first [Put](#) or [Free](#) message puts it into transmit mode, and the first [Get](#) or [Queue](#) message puts it into receive mode.

## Some Examples of use

### 1. Simple blocking send

```

Make x XMODEMLink(1)
f := fs.open("myfile.txt", "")
While f.Queue > 0
[
    ok := x.Put(f.Get)
    If ok = 0 break
]
If ok
    x.Flush

```

### 2. Receive in a polling loop (COPY)

```

Make x XMODEMLink(1)
f := fs.open("rxfile.dat", 8)
f.empty
Forever
[
    some_polling_function
    some_other_polling_function
    If x.Queue > 0
        f.Put(x.Get)
    If x.Queue < 0
    [
        if x.Queue = -1
            ; Transfer completed
            ...
            ; file receive failed...
    ]
]

```

## Flush

**Flush** ⇒ Int

Returned value:

True (1) if successful

False (0) if failed, e.g. because no acknowledgement was received

Flush completes an outgoing file transfer. Any current incomplete packet is padded out to 128 bytes with 0 (NUL) bytes and then sent. The process waits for the last packet to be acknowledged, sends an EOT byte to indicate the end of the transfer, and waits for that to be acknowledged, returning True if the process completed without timeout or cancellation.

## Free

**Free**  $\Rightarrow$  Int

Returned value:

- 0-128 : free space in transmit buffer
- 2 : Connection cancelled by other end
- 3 : Transfer failed with timeout
- 4 : Connection closed

Free returns the number of free bytes that can be written to in the transmit buffer. A positive value shows the number of bytes that can be written using the Put message without blocking. Zero means that a Put message will wait until free space is available. A negative value indicates that something has gone wrong and the transfer will not complete.

## Put

**Put**(Int **byte\_to\_send**)  $\Rightarrow$  Int

Returned value: True (1) : sent OK

False (0) : failed

Put waits if necessary for any current full packet to be acknowledged, then places the byte in the transmit buffer. If the buffer becomes full as a result, a packet is transmitted.

A returned value of 0 means that the transfer has either timed out or been cancelled by the other end of the connection.

## Queue

**Queue**  $\Rightarrow$  Int

Returned value:

- 0-128 : bytes available
- 1 : End of file received
- 2 : Connection cancelled by other end
- 3 : Transfer failed with timeout
- 4 : Connection closed

Queue returns the number of bytes available in the receive buffer. A positive value shows the number of bytes that can be read using the Get message without blocking. Zero means that a Get message will wait for an incoming packet. A negative value indicates that something has gone wrong and the transfer will not complete.

## Reset

### Reset

The Reset message returns the XMODEM object to the same initial state as when created. If a transfer was in progress it is cancelled. After a transfer has completed, a Reset is necessary to enable a new transfer to take place.

## Get

**Get**  $\Rightarrow$  Int

Returned value:

+ve : next received byte

-1 : End of file

-2 : Connection cancelled by other end

-3 : Timeout

-4 : Connection closed (possibly because of an earlier error)

Get waits until there is incoming data available, then returns the next byte of incoming data or an error code in the event of failure or normal end of file.



# Pre-processor commands

## Pre-processor commands

Venom2 has a 'pre-processor' built into it - rather like that used by the C language.

This allows you to define [macros](#) and do [conditional compilation](#).

All pre-processor commands are introduced using the **#** symbol. The **#** must be the first non-white-space character on the line.

### #Define, etc

```
#Define <macro_name>[(<parameter list>)] <text of
macro> [;<optional comment>]
```

#Define is used to define macros.

Macros are pieces of program text that have been given a name. They may be used to define constants, or larger fragments of code.

The macro name obeys the same rules as any other Venom2 name - i.e. it must start with a letter or underscore, and may only contain letters, decimal digits and underscore.

The macro text can be any text you like, including quoted strings.

The macro declaration must be the only text on the line, apart from indentation with space or tab characters and an optional comment at the end.

Examples:

```
#Define PI 3.14159
#Define clock_present net.Find(160)
#Define Age .Element(5) ;invent new message name
```

### Macros with parameters

Macros may take parameters. These are similar to the parameters to a procedure. Every time the parameter name appears in the macro text, it is replaced by the text of the parameter supplied when the macro is used.

E.g.

```
#Define SQUARE(A) (A * A)
#Define HYPOTENUSE(A,B) (Sqrt(SQUARE(A)+SQUARE(B)))
Print HYPOTENUSE(3,4)
```

### Parameters in the macro definition

Macros can have any number of parameters.

The 'formal' parameters to a macro must be contained within **()**, and must be separated by commas.

No space is allowed between the end of the macro name and the ( at start of the parameter list.

Macro formal parameter names obey the same rules as for any name in Venom.

Macro parameter names are only 'in scope' inside the macro definition - so you can reuse any existing local, global, macro or parameter names.

### Parameters when the macro is used

The text of the actual parameters to a macro must be contained within ( ) and separated by commas.

The actual parameter 'values' can be *any text*, including quotes, parentheses, semicolons and commas, so long as

- The quotes are balanced
- The parentheses are balanced
- Any commas are inside quotes or parentheses

Macro parameters are expanded before starting to parse the macro.

### Listing Macros

Macros may be listed out:

```
List Define ; lists all macros
List <name> ; lists out the given macro
List Word ; lists out all symbols by type, including macros.
```

### Removing macros

Macros may be removed using

```
#UNDEF macro_name
```

### Redefining macros

Macros may be redefined using **#Define**, but if the macro text is at all different from the original definition then the compiler will issue a warning. To redefine a macro without getting a warning use **#REDEFINE** instead.

**#REDEFINE** can be used even if the macro hasn't been defined yet.

### Constant folding

Because the Venom compiler will evaluate expressions involving constant values before generating the code for them, wherever possible, there is no performance loss on using macros that evaluate to complex expressions.

For example, though the macro hours (below) will expand to the text **((1000 \* 60) \***


60), the compiler will evaluate the expanded text to the single value 3600000 before using it.


```
#Define hours (minutes * 60)
#Define minutes (seconds * 60)
#Define seconds 1000 ; (One second in mS)

Wait hours * 2 ; Wait for two hours...
```

## Macro limitations

- Macros may only be one line long

 You must make sure there is no space between the end of the macro name and the opening parenthesis ( for the parameter list. If there is a gap then the parameter list is seen as part of the macro text.

 It's useful to put parentheses() around macros that expand to an expression - to ensure the correct calculation precedence when they are used. For example,

```
#Define twice(A) (A+A)
```

If you didn't put the expression in parentheses then if the macro was used in another expression, the + operator might come after another operation.

## #If, etc

```
#If <Int constant expression>
#ElIf <Int constant expression>
#Else
#EndIf
```

The **#If** set of preprocessor commands are used to hide sections of your Venom code from the Venom compiler, depending on the value of expressions that can be calculated by the compiler. The general structure of #If is shown below:

```
#If <conditionA>
  Compile this line if conditionA is non-zero
```

```
#ELIF <conditionB>  
    Compile this line if conditionB is non-zero and previous condit  
#Else  
    Compile this line if the previous conditions were all zero  
#ENDIF
```

**#If** is used to start a block of conditional compilation.

**#ENDIF** is required to end a **#If** block.

**#Else** indicates a block of code that should be compiled if the **#If** condition evaluated to 0 (False).

**#ELIF** is a shortcut, so you don't have to do **#If** and **#Else**. You can use as many of these as you like. The first **#If** or **#ELIF** block with a non-zero expression will compile; all further **#ELIF** and **#Else** blocks will not compile.

Only one pre-processor command may appear on any line. It must be the first non-white-space on the line. Other text on the line after the command is generally ignored. The pre-processor commands are not case sensitive, just like the rest of Venom, so you can use **#if**, **#else**, etc.

The expressions passed to **#If**, etc, must evaluate to a constant integer. Any expression that the compiler can evaluate at compile time, and that evaluates to an integer constant, may be used, including [macros](#). The expressions follow the same rules as in the Venom language.

## Nesting

You can nest **#If**, etc, to many levels - current the maximum nesting is 31 levels.

## Examples

### Example 1

```
; Select mode:  
#Define debug_mode True  
  
; Conditionally compile code depending on mode:  
#If debug_mode  
    Print value,CR  
#Else  
    store(value)  
#EndIf
```

### Example 2

```
; Set the product we are compiling code for:
#Define PRODUCT_CODE FULL_SPEC

; Define some product numbers:
#Define FULL_SPEC 1
#Define MID_RANGE 2
#Define CUT_DOWN 3

; Conditionally compile some code:
#If PRODUCT_CODE = FULL_SPEC
    Print To screen, "Full"
#Elif PRODUCT_CODE = MID_RANGE
    Print To screen, "Mid-range"
#Elif PRODUCT_CODE = CUT_DOWN
    Print To screen, "Cut down"
#Else
    Print To screen, "No Product!"
#EndIf
```



If you don't use enough **#EndIf** commands then you may find that the VM2 doesn't display the **-->** prompt at the end of a download. You can get out of this state by hitting Ctrl-C.

You should, of course, also fix your code.

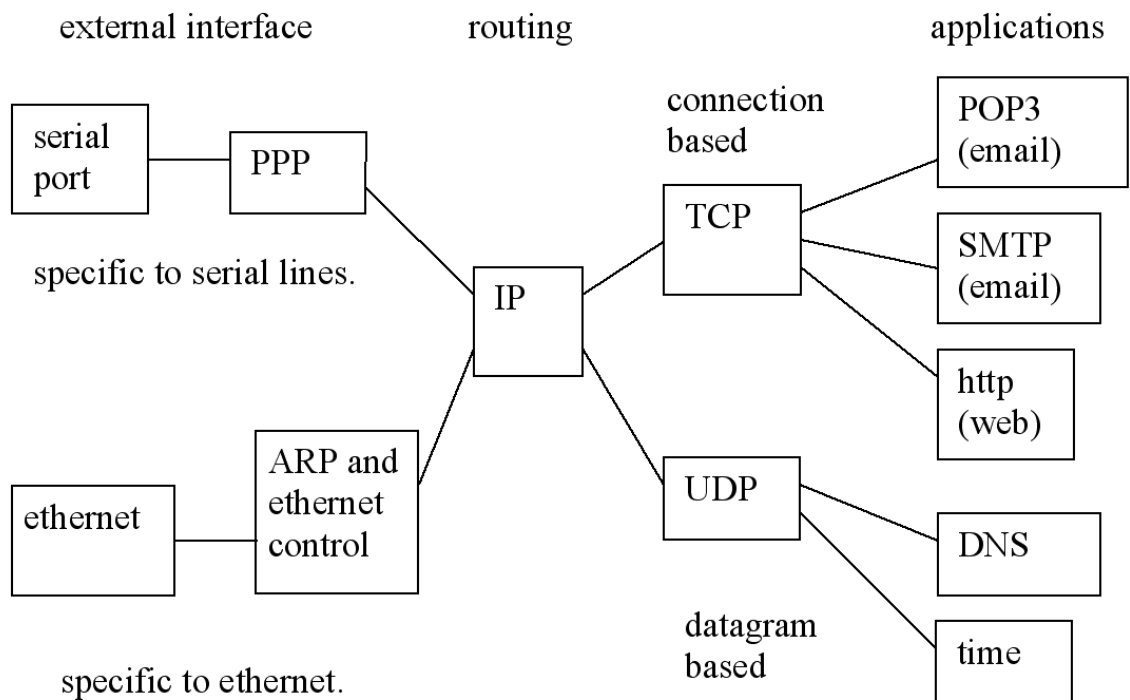
# **TCP/IP Networking**

## TCP/IP Networking

TCP/IP networking is covered in Venom by a number of related object types. This section shows how they are related and includes general networking and programming information.

### How the protocols are related

#### IP Protocol Family



#### TCP/IP Objects Supported by Venom 2

Object	Use	Description
<a href="#">PPPProt</a>	PPP	Link management protocols for serial link to modem or GPRS terminal.
<a href="#">Ethernet</a>	Ethernet	Local Area Networking - hardware driver and low level protocols: ARP, DHCP.
<a href="#">IPProt</a>	IP	Miscellaneous tools.
<a href="#">TCPProt</a>	TCP	General character stream link between two processes.
<a href="#">UDPProt</a>	UDP	General packet based link between two processes.
<a href="#">FTPClient</a>	FTP	File Transfer Protocol client.
<a href="#">FTPServer</a>	FTP	File Transfer Protocol server.
<a href="#">HTTPServer</a>	HTTP	Web server.



<a href="#">SMTPSender</a>	SMTP	Send email.
<a href="#">POP3Mailbox</a>	POP3	Receive email.
<a href="#">WiFiLink</a>	WiFi	Wireless Networking interface, similar to Ethernet.

## Notes on TCP/IP

### Notes on TCP/IP for Venom Programmers

#### Multitasking and Timing

Once a connection has been made and incoming packets can be expected, it is up to the Venom programmer to send object messages that result in those packets being seen and processed.

The serial ports have a buffer of 255 characters. Calls must be made sufficiently frequently to prevent the input buffer from overflowing, or packets will be lost and performance will suffer dramatically. As an example, at a connection speed of 38400 bps, the input buffer can fill up in about 64 milliseconds.

The Ethernet interface can receive packets much faster than this, but has a buffer capacity of 4k which will hold 7 packets so as long as TCP's inherent flow control is operating properly this should not overflow unless several different sources are trying to send lots of data to the VM2 at once and the TCP connections have been set up with large receive buffers.

As long as any message is sent to any TCP/IP networking object sufficiently frequently, all existing connections will be serviced properly. For example, a typical program might satisfy this requirement by polling a TCP receiver with Queue messages at sufficiently frequent intervals, or a simple TCP Get message will do the same while it is waiting for data.

In practice this is not usually a problem. A server task is listening constantly anyway, and a client task is not expecting any incoming packets unless it has sent a request and is waiting for a response. A minor exception is where you want the VM2 to respond to a "ping" request but not to participate in any data connection; in that case an IP Go message will allow that to happen.

## IP Addresses

### IP Addresses and domain names

An IP address is a 32 bit number. It is conventionally written as a "Dotted Quad" of decimal numbers representing one byte each e.g. "**152.158.91.245**" but can also be represented by an integer in Venom.

Any message parameter described as *host* can be either

- An integer representing the IP address
- A string representing the address as a dotted quad

- A string containing the domain name, assuming the VM2 has TCP/IP access to a name server, which is usually the case if an internet connection is available.
- A string containing a local host name assigned with the IP.Address message.

The following code illustrates how an IP address is constructed, and the use of the [“IP” formatting specification](#) in a Print statement.

```
-->a := 23
-->a := a * 256 + 42
-->a := a * 256 + 159
-->a := a * 256 + 212
-->print a:"IP",CR
23.42.159.212
-->
```

The preceding calculation can also be done by sending the Address message to an IP object (see [ip.Address](#)).

## Example Code

### 1. Simple Ethernet LAN “ping” Test

In three lines of code this creates an Ethernet and IP object and tests the existence and response time of another node in the network, using the equivalent of the well-known “ping” utility.

1. Make an [Ethernet](#) object. The default address is that of the Ethernet interface on our Application board, and we are going to use DHCP to allocate an IP address, so no parameters are required.
2. Create an [IProt](#) object to receive the Time message and create and receive the necessary ICMP messages.
3. Send [Time](#) message to measure the response time of another device on the LAN and print the result.

```
Make eth ethernet
Make ip IProt
Print ip.Time("172.16.1.199"), " ms response time",CR
```

### 2. Getting the VM2 to respond to a "ping" Test

The VM2 will send a response to any incoming ICMP packet as long as interfaces are being polled. Any message activity involving TCP/IP objects will enable this; in addition the [Go](#) message sent to an [IP object](#) will cause the interfaces to be polled without any other side effects. The simplest way to set up a VM2 to respond to ICMP packets on a LAN for test purposes is this:

```
Make eth Ethernet
Make ip IProt
```

```
ip.Go
```

This will now respond to ICMP (ping) packets. Press Ctrl/C to return to a command prompt.

### 3. Pinging an Internet site through a LAN gateway

This does the same as (1) above, but to a remote host. It is assumed that:

- A gateway to the internet exists on the LAN
- A DNS name server is available (either local or remote)
- DHCP can supply the address of the gateway and the nameserver.

Note: The string “www.google.com” is recognised as not being a numeric IP address, so the IP system software will transparently make a DNS request for the address.

```
Make eth Ethernet
Make ip IProt
Print ip.Time("www.google.com"), " ms response time",CR
```

### 4. If you don't have a DHCP Server

If there is no DHCP server on your network you will have to assign an available IP address to the VM2, and set the IP addresses of your name server and default gateway if you need them. The addresses shown are just examples; you'll have to use something appropriate for your LAN.

```
Make eth Ethernet(12, "192.168.1.52")
eth.address('N') := "192.168.1.200" ; local name server
eth.Address('D') := "192.168.1.199" ; default gateway to internet
```

You can also do this all at once in the Make statement:

```
Make eth Ethernet(12, "192.168.1.52", "192.168.1.200", "192.168.1.199")
```

### 5. Simple Dialup Program to fetch a Web Page

This program is shown in two alternative versions: either make a dialup connection to an ISP via a modem or set up an [Ethernet](#) connection to a LAN with internet gateway. Both then fetch a test page from our web site (the test page exists and is at <http://www.venomcontrolsystems.co.uk/testpage.html>) and display the text from the page. You can run this if you have either a dialup account with an ISP or a LAN connected to the internet.

The procedure `Init` creates the three objects needed. This version is for a dialup connection.

```
To init
Make modem SerialPort(38400, 2, 1)
Make ppp PPProt(modem)
Make tcp TCPProt
ppp.name("myusername", "mysecretpassword")
Print To ppp, "AT&F0", CR, "AT&D1M0", "ATDT908456042086"
ppp.Debug := 1 ; be verbose (this can be changed to 0)
```

**End**

The main program procedure opens the dialup connection, then calls the http fetch procedure, closing the dialup link when it's finished.

Note that when we make the PPP connection, we don't even need to know what our IP address is (it's the value returned by `ppp.open`, or 0 if it failed), nor do we concern ourselves with the address of a DNS server, though both these pieces of information have been assigned to us by the ISP and will be used internally by later code.

```
To main
  Print "TCP/IP demo",CR
  if ppp.Open
  [
    fetchtestpage
    ppp.Close
  ]
  else
    Print "dialup connect failed",CR
End
```

This alternative version uses Ethernet and will work if your LAN has a gateway to the internet and a DHCP server.

```
To init
  Make eth Ethernet
  Make tcp TCPProt
End

To main
  fetchtestpage
End
```

This procedure is a very simple implementation of the client end of the http connection.

1. Open a TCP connection to port 80 (standard port number for http) on the web server for the VCS web site.
2. Send two lines of text containing a valid http request, followed by an empty line.
3. Receive and display all the headers and text returned. We stop when the remote end closes the connection, indicated by a negative [`tcp.Queue`](#) value, or if the stipulated timeout of 10 seconds is exceeded.

Note:

- We simply put the domain name of the site in the [tcp.Open](#) message. A DNS query will be performed invisibly to look up the IP address of the site. The IP address is available to the Venom code, but rarely needed.
- we can close the sending end of the connection straight after sending the request; the receiving side of the connection remains open until closed by the remote web server.

```
; simple test page connect
To fetchtestpage
Local c
  Print "TCP connect to http://www.venomcontrolsystems.co.uk...",

  If tcp.open("www.venomcontrolsystems.co.uk", 80)
  [
    tcp.printf(<<<:
GET /test.html HTTP/1.1
Host: www.venomcontrolsystems.co.uk
Connection: Close

    >>>)

    ;Print To tcp,
    ;"GET http://www.venomcontrolsystems.co.uk/test.html HTTP/1.1
    ;CR, "host: www.venomcontrolsystems.co.uk", CR,CR

    tcp.close
    tcp.timeout := 10000
    While tcp.queue >= 0
    [
      c := tcp.get
      if (c > 0)
        Print chr c
    ]
    If tcp.queue = -1
      Print CR,"connection closed",CR
    Else if c = -2
      Print "*** timed out", CR
  ]
  else
    Print "FAILED",CR
End
```

## Glossary

### Glossary of Abbreviations

Name	Expansion	Description
<b>ARP</b>	Address Resolution Protocol	Matches up 48 bit Ethernet addresses with IP addresses
<b>CHAP</b>	Challenge Handshake Authentication Protocol	Part of PPP – a more secure method of verifying a user's credentials
<b>CRC</b>	Cyclic Redundancy Check	Verifies that data in a packet has not been corrupted in transmission.
<b>DNS</b>	Domain Name Service	Relates domain names to IP addresses
<b><u>FTP</u></b>	File Transfer Protocol	list directories, send and receive files
<b><u>HTTP</u></b>	HyperText Transfer Protocol	The protocol used between web browsers and web servers.
<b>ICMP</b>	Internet Control and Monitoring Protocol	Used by IP devices to test and report on the state of the network.
<b><u>IP</u></b>	Internet Protocol	Concerned with addressing and routing
<b>IPCP</b>	Internet Protocol Control Protocol	Part of PPP – deciding IP addresses of the ends of a link and DNS addresses
<b>ISP</b>	Internet Service Provider	Usually the entity at the other end of a PPP link, providing a connection to the rest of the internet.
<b>LAN</b>	Local Area Network	Network sharing a common geographical location and address range
<b>LCP</b>	Link Control Protocol	Part of PPP – setting up connections, negotiating options
<b>MAC</b>	Media Access Control	The lowest level Ethernet protocol. MAC address = Ethernet hardware address
<b>NTP</b>	Network Time Protocol	Synchronises time-of-day clocks accurately and efficiently across the net
<b>PAP</b>	Password Authentication Protocol	Part of PPP – verifying a user's credentials
<b><u>POP3</u></b>	Post Office Protocol 3	For fetching email from a mailbox on a POP3 server.

<b><u>PPP</u></b>	Point to Point Protocol	Family of protocols for using serial lines for packet transfer
<b><u>SMTP</u></b>	Simple Mail Transport Protocol	The method used for sending email, usually through a server which passes the mail on.
<b><u>TCP</u></b>	Transmission Control Protocol	Setting up sessions between two endpoints; error-free transfer of streams of byte data between the two points
<b><u>UDP</u></b>	User Datagram Protocol	For speedy short messages and time-critical data streams
<b>URI</b>	Uniform Resource Identifier	Standard method of naming a resource on a network and the protocol for retrieving it
<b>URL</b>	Uniform Resource Locator	As URI, but more specific about location of resource

# Appendix



## Appendix

The appendices contain extra information that doesn't fit into the keywords and object types sections.

See the Contents tab to the left of this page for each section.

### A: Startup Sequence

This bit of 'pseudo code' is intended to show what happens when the VM2 starts up. (This code has been simplified to show the essence of the logic).

```
To startup_sequence
;This bit is just for Program Mode:
If (program_mode)
[
; Check to see if we should be reprogramming the VM2 from the
; Flash File System.
#If VM2L IsFalse
If (USB_SWITCH_ENABLED)
[
If reprogrammed_vm2_from_flash_files
Reset
]
#ENDIF

startup_banner ;Display the startup banner

If clear_memory_question = 'Y'
[
erase_ram
If (application_in_flash)
erase_flash
]
]

;This bit is for Run Mode and Program Mode:
If (application_in_flash)
[
erase_ram ; Always clear RAM when running from Flash.
load_application_from_flash
If (program_mode)
Print "Loaded application from Flash.", CR
]
Else ; application is in RAM
```

```

[
    validate_code_in_ram      ; Check the code stored in RAM is OK
    create_default_procedures ; Make sure default procedures exist
]

; Run the startup procedure, And thus the whole application.
run_command("startup\n");

command_line ; If startup ends then process the command line.
End

```

## B: Robust applications

This appendix deals with how to create robust Venom2 applications, i.e. those that have the least chance of going wrong and needing attention when they are in the field.

### Protecting your Application Code

While you are developing your application program, your procedures are held in battery-backed RAM. This is fine for development, but not suitable for a finished application in the field: there are many ways to lose a program from battery-backed RAM.

Finished applications should be copied into the Flash memory. Flash memory is 'non-volatile' - it keeps its contents even when not powered. Once the application is in Flash memory the application is very secure.

 See [\(OperatingSystem\) Protect](#).

### Dealing with runtime Errors

A Runtime error in your code would cause your program to halt forever if it returned to a Venom command line. This is usually unacceptable for an embedded control application in the field.

To prevent errors from causing your program to halt use [Try](#) to trap any errors that you know how to handle.

To deal with errors that you haven't thought about, and so don't know how to handle explicitly, use the [ErrorAction](#) system message.

```
System . ErrorAction := 1
```

This restarts the Venom application on any error not handled by Catch.

*Note: The default **startup** procedure sets ErrorAction according to the Prog Mode switch: When the Prog Mode switch is set, runtime errors will be reported to the programmer. When the Prog Mode switch is not set runtime errors will result in the program restarting.*

*You are advised not to alter the the default startup procedure.*

### Serial Break

Most finished applications should turn off [Serial.Escape](#) (which controls Ctrl-C break), as this could potentially halt an application.

To turn off Ctrl-C Escape, use

```
Serial.Escape := False
```

[Ctrl-C Escape is treated as a runtime error, so if Ctrl-C Escape is allowed, but [ErrorAction](#) is set, the Venom application will be restarted].

*Note: The default **startup** procedure sets Serial.Escape according to the Prog Mode switch:*

*When the Prog Mode switch is set Ctrl-C is enabled.*

*When the Prog Mode switch is not set Ctrl-C is disabled.*

*You are advised not to alter the the default startup procedure.*

### Watchdogs

A watchdog is a hardware device that has control of the reset input to the controller. If the program does not 'kick' the watchdog every so often, then the watchdog will reset the controller. This is to halt and restart a crashed micro-controller.

In the VM2 controller, the Venom task scheduler kicks the watchdog. This is sufficient to guard against most bugs in the Venom language, or processor crashes.

## C: Calling foreign code

See [Call](#) keyword.

## D: Development Checklist

The steps involved in developing a typical Venom application are presented here. You may have completed some of these already.

1. Satisfy yourself that the controller and/or application board have the hardware interfaces that you require. See the datasheet for the controller. Often customers will buy the controller from us, and design the application board themselves. However, we can design and manufacture custom application boards.
2. Get familiar with the Venom language and basic Object Types by reading the *Tutorial* and by trying out your ideas on your development system.
3. Use this reference manual for reference on Venom2 keywords and object types.
4. Using development hardware, write key sections of your application to make sure that they are viable.

5. Design and build the real application hardware in conjunction with the controller's datasheet and example circuits.
6. Write the complete application program.
7. Read through the Appendix on [Robust Applications](#) to make sure your application is as robust as possible.
8. Test the application hardware and software.
9. Protect your application from erasure by burning it into the onboard flash memory. See the system message [Protect](#).
10. Go into production with the application hardware and the application code.

## E: Error messages

Here are all the runtime error messages in the Venom2 runtime error system.

Each error has a number and associated error text. Sometimes some extra text is added to the error text in the table to pin point the error more accurately, e.g. error 5, Un-initialised variable will usually name the variable concerned.

You can ask Venom to list all of it's error codes using **Debug (13)** .

0	Exit with no error	<i>Escape via Exit 0. This is not really an error.</i>
1	Escape via CTRL-C	<i>Escape via CTRL-C. This is not really an error.</i>
2	Ram full	When there is no RAM left to perform an operation. Often caused by repeatedly MAKEing objects
3	Value out of range	A parameter or operand is outside a required range
4	Type mismatch	A parameter or operand is of a type that can't be handled
5	Un-initialised variable	Attempt to read a variable that hasn't yet been given a value
6	Division by Zero	An attempt to divide by Zero
7	Write to read-only item	Attempt to write to a write-protected item, e.g. a procedure name, or a read-only message in an object
8	Array/buffer index out of range	An index is outside limits, e.g. reading off the end of a Buffer, Array or other indexed item.
9	Illegal bytecode	An illegal bytecode is seen by the <a href="#">BCI</a> . Usually caused by a bug in the Venom compiler or corrupt memory. Please contact us.

10	Message sent to non-object	Attempt to send a message to a non-object, for example a. Put, where a is an integer.
11	Parameter list applied to non-procedure	When a variable or expression is followed by a parameter list, but isn't a procedure, so can't use parameters
12	Device not found	Could not find the hardware needed for the operation
13	Venom stack overflow	The Venom Stack is used up; usually caused by <a href="#">recursion</a> in your Venom program.
14	Stack wrong at End	The <a href="#">BCI</a> finds the stack is not correct. Usually caused by a bug in the Venom compiler or corrupt memory. Please contact us.
15	Unexpected parameter number/types/values	No type of object is possible with the parameter values and types supplied to Make or New
16	Wrong no of parameters supplied	Procedure or message could not accept the number of parameters passed
17	Message not recognised	The object didn't accept this message
18	Attempt to lock object held by dead task	A task locked an object, then later died without unlocking it; subsequently another task tried to lock it. See <a href="#">here</a> for how to block this error.
19	Message to dead object	A message was sent to a dead object
20	<i>[Unused error number]</i>	
21	Hardware fault	A fault has been detected in the hardware
22	Too many	Too many items have been created: Some objects, etc, have a fixed number of sub-object that they support.
23	Code checking error	Internal code check failed (like assertion failure in C). Please contact us.
24	File access error	File system specific
25	Resource error	Memory or other resource ran out (but not the controller's SRAM, which is reported with Ram full)

26	Script/Data error	Invalid data value or syntax error in a script
27	Protocol error	Various TCP/IP errors typically caused by performing actions in wrong sequence
28	Heap error	An error was detected in the heap structure during a heap operation. This could be because of a bug in Venom2, or because your code accessed the heap memory incorrectly (using <code>?</code> , <code>Call</code> or other mechanisms that use a raw memory pointer).
29	Flash programming error	There was a failure while programming one of the internal flash memories.
30	Locked/Unlocked too many times	An object has had <code>.Lock</code> or <code>.Unlock</code> called too many times. The maximum lock level is 255. Also caused by unlocking an object that was locked by another task.
31	Device Configuration Error	A device was not configured to perform the requested operation.
32	<i>Task Stop request</i>	<i>When <a href="#">Stop</a> is used, a task is forced to exit with this 'error'. This may be trapped using <a href="#">Try</a>, but it will never appear in a runtime error report.</i>
33	Feature not supported	Thrown when a program tries to use a feature not supported by the current Venom version. Usually when it tries to create an object.
34	No Print Job	Thrown when a Print message is sent to an object and no Print Job has been set up. Usually when the Print message is sent explicitly, and not inside a Print statement.

## F: FAQ

### Text manipulation

Both Strings and text Buffers can be used for manipulating text:

#### Extract a sub-string

Use **Print text:start:nchars** to extract a section of text from a String or text Buffer

#### Append text

Use **obj.Put**, where *obj* is a String or text Buffer.

**Print To** a text Buffer or String.

#### Insert text

Use **Insert** to insert text into a text Buffer

#### Find a search string

Use **Find** to find text within a String or text Buffer.

#### Divide into lines

**Print To** a Buffer of Any to divide text into individual lines of text, one line per String.

Example:

```
To GetFiles
  AutoDestruct
  Local BuffAny := New Buffer(Any)
  Print To BuffAny, FileSys:"*.txt":0 ; Each file name matching *
  Print BuffAny.(0) ; Print the first matching name.
End
```

For more Frequently asked questions refer to our website.

## G: Glossary

<b>Active variable</b>	An active variable is a message whose value may be read and written to. An example is Digital. Asserted.
<b>Analogue</b>	An analogue signal is one that can take a range of values rather than the On and Off of digital signals. The range is divided up into discrete steps. See Resolution.
<b>Application</b>	The application program is the your Venom code, consisting of the procedures you have written.
<b>ASCII</b>	ASCII stands for American Standard Code for Information Interchange, and gives a standard numbering system for the letters, numbers and symbols used in computing. For example, the ASCII code for the letter 'A' is 65.
<b>BCI</b>	Byte Code Interpreter - see interpreter.
<b>Binary</b>	Binary is the number base 2, where everything is expressed as Bits of either 0 or 1. Fundamentally, this is what computers work in, but it is rather unwieldy, so hexadecimal is often used instead.
<b>Bit</b>	A bit is a single binary digit, which can either be 0 or 1. Several bits may be combined to give a number.
<b>Bit-wise</b>	A calculation is performed bit-wise if the same operation is performed on every binary bit of a number individually. In Venom, the And, Or, Eor and Inv operators are bit-wise.
<b>Boolean Operator</b>	Boolean operators are named after George Boole, who first formally investigated the operation of the 'logical connectives' And, Or and NOT. (The corresponding operators in Venom2 are

AndAlso, OrElse and IsFalse). Later, the operator Eor was added for mathematical completeness. Unlike And, Or and NOT, Eor isn't used in normal speech. A.A. Milne is the only writer ever to have used Eor successfully in an English sentence.

<b>Byte</b>	A group of 8 bits, which can represent numbers in the range 0 to 255. This is the standard unit of memory for computer systems, chosen because it was enough to store an ASCII code. In hexadecimal, each hexadecimal digit represents half a byte, or one nybble.
<b>Bytecode</b>	A Bytecode is an instruction that the Venom Virtual Machine understands. They are like virtual machine codes. The Venom2 Compiler turns your Venom code into bytecodes. The Java language also uses bytecodes as it makes it easy to port to different platforms.
<b>Call</b>	When a procedure is executed, either by including its name in the program, or by typing its name in at the command line, this is referred to as a procedure call. Call is also a keyword in Venom, used to call code in Assembler, C or other languages.
<b>Channel</b>	A channel is a signal used for input and output. It has been given a reference number as a convenience in Venom. The controller datasheet shows you which channels are available where.
<b>Class</b>	A class refers to a <i>kind</i> of object. In Venom there are two distinct groups of classes - those pre-defined by the language (such as Digital and DateTime) and those defined by the Venom programmer using the keyword Class.
<b>Code</b>	A piece of Code is a program or section of a program.
<b>Command Line</b>	The command line is the part of the Venom system that sends the --> prompt to the host computer and then accepts characters typed in by the user.
<b>Digital</b>	A digital signal is one that can either be On or Off. The voltage levels equivalent to On and Off depend on the hardware concerned. This variation also applies to the polarity: Off is often the higher voltage!
<b>Dot Chaining</b>	Dot chaining is a shortcut that allows you to send a message to the result of a previous message. Obviously this result must be an object, and this result should not use up any memory.
<b>Expression</b>	An expression is a piece of program that performs a calculation. Each expression has a value: the result of the calculation. The smallest expression is just a value by itself.
<b>Firmware</b>	Firmware is software that is permanently stored in the computer by being placed in a non-volatile memory (e.g. Flash).
<b>Flag</b>	A Flag is a name for an integer chosen to only represent True or False. The name is believed to come from railway signalling. In Venom, True is represented as 1, and False is 0.
<b>Flash</b>	This is a 'writable' ROM. A flash device on the controller holds the Venom2 Language, and also your application code when you 'Protect' it.
<b>Float</b>	A float is a floating-point number, which allows decimal fractions to be represented as well as whole numbers. Floats in Venom are IEEE single precision.
<b>Fragmentation</b>	Fragmentation occurs if the memory blocks used by the application are taken and returned out of order. As this continues, instead of there being one large area of free memory, there are many small areas, separated by memory blocks that are still being used. Only programs that do a lot of complex, dynamic creation and destruction of objects are likely to risk this problem.
<b>Function</b>	A function is a name for a procedure that returns a result. By definition, a pure function is one that only returns a result, and does not have any side effect on any other part of the system.
<b>Garbage</b>	Garbage refers to heap memory that has been allocated, but that there is no longer a reference to, so it can not be de-allocated (or freed) directly. This memory is not useable and may cause a program to halt due to lack of sufficient memory.



	<p>In Venom this can happen when an object is created and then the variable that referenced the object is overwritten, e.g.</p> <pre> a := New DateTime a := Nil </pre>
<b>Garbage collection</b>	<p>In some computer languages an automatic garbage collector runs from time to time to recover 'garbage'. However it is very difficult to implement an automatic garbage collector in a Real Time system. Instead, Venom provides the <b>AutoDestruct</b> attribute for local variables and Class members.</p> <p>You can detect garbage in your program by using the garbage scanner: Debug(1,...)</p>
<b>Global variable</b>	<p>A global variable is a variable that is accessible throughout the program. Global variables are created just by assigning a value to a name, e.g: var := 1</p>
<b>Hardware</b>	<p>Hardware is the physical electronics that makes up the computer and the devices that are connected to it.</p>
<b>Heap</b>	<p>The heap is an area in the controller's RAM that is reserved for use by your application, and also the Venom system, in ways that can't be predicted ahead of time.</p>
<b>Hexadecimal</b>	<p>Hexadecimal is the number base 16, which is used as a shorthand way of writing binary. The numbers 10-15 are represented by the letters A-F. Each hexadecimal digit represents 4 binary bits. Hexadecimal is usually used when accessing a computer's hardware directly, since hardware is usually laid out in round numbers in binary.</p>
<b>Host</b>	<p>The host is the computer (usually a PC) that runs a terminal emulation program so that the user can communicate with the controller.</p>
<b>Instance</b>	<p>When many objects of a single type are created, each of these objects is referred to as an <i>instance</i> of the object type. When a procedure is called more than once at the same time by itself, or by different tasks, each called procedure is referred to as an <i>instance</i> of that procedure.</p>
<b>Int</b>	<p>An Int is an integer: i.e. a whole number. Integers in Venom are stored in 32 bits, representing roughly <math>\pm 2</math> billion.</p>
<b>Interpreter</b>	<p>The Interpreter is a very fast program that reads and executes bytecodes. The Venom2 compiler turns your Venom code into bytecodes. The Venom2 interpreter is sometimes called the Bytecode Interpreter, or the Venom Virtual Machine. VM2 stands for <i>Venom Machine 2</i>.</p>
<b>K, Kilobyte</b>	<p>A kilobyte is 1024 bytes. The unusual number is chosen because it is a nice round number in binary.</p>
<b>Keyword</b>	<p>A keyword is one of the basic elements of the Venom language, such as Repeat and If. You cannot give procedures and variables the same names as any keyword.</p>
<b>Local variable</b>	<p>A local variable is a variable that is only accessible within a certain instance of a procedure. They will 'eclipse' any global variable of the same name for the duration of the procedure. Local variables are created with the Local statement.</p>
<b>Locking</b>	<p>Locking is a way of ensuring that only one task has access to a particular object at any time, to prevent transactions from two different tasks becoming confused.</p>
<b>Machine Code</b>	<p>Machine code is the set of instructions that the microcontroller obeys. Programs written in machine code run faster than those written in the Venom.</p>
<b>Member</b>	<p>A member is an item of data defined inside a user-defined Class. Members may be of any data type, including numeric types, strings and objects.</p> <p>Procedures defined inside a Class are called Methods, but may usefully be considered as members of type 'Procedure'.</p>

<b>Message</b>	A message is something that is sent to an object, consisting of a message name, such as On, Value, DefaultOutput etc., and some parameters. Each object type may respond to each message in a different way.
<b>Method</b>	A method is a procedure defined within a user-defined Class. It is almost identical in its effect to a message of a Venom object.
<b>Object</b>	An object is an element of the system, containing both code and data, that is used to interface to the outside world, store data, etc. All objects have a type, which defines how they act in response to messages. There can be more than one instance of some types, and these act independently as they hold different data.
<b>Operand</b>	An operand is a value acted on by an operator. In the expression 'a + b', a and b are operands.
<b>Operator</b>	An operator is a language symbol or keyword that acts on one or two operands to produce a result. Each operator has a precedence that governs the order in which operations are evaluated.
<b>Parameter</b>	A parameter is a value that is passed to a procedure or message. This value is operated on by the procedure or message.
<b>Pixel</b>	A pixel is the smallest element of a display screen that can be individually controlled. The greater the number of pixels, the higher the resolution of the screen, and the better quality of the image.
<b>Pointer</b>	A pointer is a value that holds a reference to some data, rather than the data itself.
<b>Postfix operator</b>	A postfix operator is an operator that comes after the value it modifies. Examples include As Int and As Float.
<b>Precedence</b>	Precedence sets the order of execution of operators in a complex expression. The higher the priority, the earlier it is executed. See the appendices.
<b>Prefix operator</b>	A prefix operator is an operator that comes before the value it modifies. Examples include ! and Inv.
<b>Procedure</b>	A procedure is a set of commands grouped together as one unit. See the To keyword. A procedure can take parameters, and return a result.
<b>RAM</b>	RAM stands for Random Access Memory, but the name is a relic. RAM now generally refers to memory that is <i>writable</i> as well as readable.
<b>Recursion</b>	This is where a procedure calls itself, either directly or indirectly. This can be very useful for solving some kinds of problems, but is often done unintentionally. In the unintentional cases it usually leads to a stack overflow error.
<b>Resolution</b>	<p>The resolution of a device is the smallest change detectable, or controllable, by it. When used with analogue I/O, resolution refers to the size of the smallest voltage step between possible input readings or output levels.</p> <p>The resolution of a display screen is the number of pixels it has - this governs the quality of the image. The lower the resolution, the more grainy the image becomes.</p> <p>The resolution of a timing operation (such as PulseWidthOut.Width) is the smallest interval in time that the timing can change by.</p>
<b>ROM</b>	ROM stands for Read-Only Memory. Usually these days, ROMs are mostly Flash memories. Flashes are actually writable – though there may be safeguards to stop this occurring accidentally.
<b>Routine</b>	Routine is another word for procedure or function.
<b>Software</b>	Software is the program that the hardware executes. If software is contained in a ROM, it is called firmware.
<b>Statement</b>	A statement is a complete piece of program that performs some action.

<b>String</b>	A string constant is a series of characters, enclosed by quotes, such as "A string". You can use String objects and text Buffers for variable text.
<b>Syntax</b>	Syntax is the structure of the language: how the elements of the language fit together.
<b>Task</b>	A task is part of a program that runs as though it were executed by an independent processor with access to the same memory. In VM2 a single processor is shared by all the tasks.
<b>Tri-state</b>	A tri-state output is a digital output which, as well as being on or off, can go into a high-impedance state where it doesn't drive either way.
<b>Type</b>	All venom variables are of one type or other, for example Int, Float, String. Type can also refer to what kind of object something is; in this case the type indicates what messages the object understands, and what action it takes on receipt of each message. Examples of objects types are Buffer, PulseWidthOut etc.
<b>Variable</b>	A variable is a place to store a value. See also Global variable and Local variable.

## H: Number Limits

Integers are signed 32-bit quantities:

Maximum	Minimum
2,147,483,647	-2,147,483,648
\$7FFFFFFF	\$80000000

Floats are standard IEEE single precision: 1 sign bit, 8 exponent bits and 23 mantissa bits.

Maximum	Least non-zero	Precision
±3.39E+38	±1.18E-38	~7 decimal digits

## I: Operator Precedence

The following table shows the order of precedence of the operators in Venom2. The operators at the top of the table are more tightly binding than those at the bottom.

Those on the same line have the same precedence.

Operators with the same precedence level are evaluated from left to right in an expression, apart from prefix operators, which are evaluated from right to left.

Parentheses	( )
Postfix	• As Int As Float IsFalse Is Has
Prefix	- Abs Inv TypeOf ? ? ? ? ? ! Sin Cos Tan Asin Acos Atan

	<b>Sqrt Exp Log</b>
Multiplicative	<b>* / ^ Div Mod &gt;&gt; &lt;&lt;</b>
Additive	<b>+ -</b>
Comparative	<b>&gt; &lt; &gt;= &lt;= = &lt;&gt;</b>
Logical & Bitwise	<b>And Or Eor AndAlso OrElse</b>

The following examples show some cases where precedence is important, and how they are resolved by Venom:

Expression	Is interpreted as if it were written as...	To get the other interpretation, use...
<b>2+3*4</b>	<b>2+(3*4)</b>	<b>(2+3)*4</b>
<b>x And 1 = 1</b>	<b>x And (1 = 1)</b>	<b>(x And 1) = 1</b>
<b>2.1 * x As Int</b>	<b>2.1 * (x As Int)</b>	<b>(2.1 * x) As Int</b>
<b>! object .Value</b>	<b>! (object . Value)</b>	<b>(! object) . Value</b>
<b>!proc_ptr(a,b)</b>	<b>! (proc_ptr (a,b) )</b>	<b>(!proc_ptr) (a,b)</b>

If you start a statement with ‘(‘ then you should enclose the statement in square brackets so that the Venom parser (which doesn’t have statement separators to tell it where a statement ends) can parse it correctly. Otherwise it may interpret ‘(‘ as the beginning of a parameter list.

So

```
(!proc_ptr) (a,b)
should be written
[ (!proc_ptr) (a,b) ]
```

## J: Speed of Execution

Venom2 is a *semi-compiled* language, like Java. This means it compiles your code to a set of bytecodes. These codes are then interpreted by the Venom runtime system to run your application. Semi-compiled code runs faster than interpreted code, but not so fast as native machine code. Typically, a single bytecode will execute in 1-2µS on the VM2. A bit of code like **a := a + 1** will take ~4.5µS.

### Measuring Execution Times

The following code allows you to measure the execution time of a bit of Venom code.

```
To measure_time(n,c)
  Local t
```

```

AutoDestruct
Local stop_watch := New Stopwatch

stop_watch.Reset
Repeat n
[
    ;commands to be timed
]
t := stop_watch.Time As Float
Print (t / n - c):10:4, " milliseconds", CR
End

```

The parameter n is the number of times the loop is repeated - increasing it increases the accuracy of the result. The parameter c is a constant adjustment that is used to take into account the time taken to execute the [Repeat](#) command.

Firstly, the procedure should be run with n = 1000; c = 0 and the Repeat command empty. This will then print the value to use for c.

Then put the code under test into the Repeat, choose a value of n, and use the value of c you just found.

## K: Optimisation

Optimisation is the automatic or manual alteration of code to make it run faster, occupy less space, or use less electrical power.

### Code Optimisation

The Venom compiler automatically performs some optimisations on the Venom code you write.

Because Venom2 is semi-compiled, the size of the code it produces is typically much smaller than either assembly code or fully compiled code.

Venom also does some more explicit optimisation. Currently this is limited to constant folding. Constant folding is where an operation on one or more constants may be calculated at compile time rather than at run time. For example, the first line could be written as the second, but the first line may be more understandable and maintainable.

```

a := 5 * 4
a := 20

```

When Venom compiles these lines of code, it is able to notice the possible optimisation, and compiles as if the second line had been written.

Constant folding is performed on most operations. In order to ensure folding happens, enclose the operations in parentheses:

```

5 * a * 4    ;will not be folded (the compiler's not that clever!)
5 * 4 * a    ;might be... (implementation dependent)
a * 5 * 4    ;might be... (implementation dependent)
a * (5 * 4)  ;definitely will be.

```

Here is the current list of the operations that may be optimised by constant folding.

```
+ - * ^ / Div And Or Eor Mod << >>
= >= <= < > <>
IsFalse Inv - negative
As TypeOf for Integer, Float, and String constants & Nil
Sqrt Log Exp Sin Cos Tan Asin Acos Atan
```

## Power saving

The Venom operating system automatically uses the SLEEP instruction on the host processor. The controller is put into a power-saving mode if there are no tasks requiring any processing power. Interrupts are not affected as they automatically wake the controller from its SLEEP instruction.

In order to make best use of this, make your tasks wait if they can do so without compromising the responsiveness of your code.

For example you could wait for a digital input like this:

```
While dig.Asserted IsFalse []
```

However if you don't mind being up to NmS late in the detection of the input you can save power by using something like

```
While dig.Asserted IsFalse [Wait N]
```

The Wait command will let the controller idle while it's waiting.

Await will also allow the controller to sleep while it's waiting, with a minimal loss of responsiveness

```
Await dig.Asserted
```

All commands and messages in Venom that are waiting for an interrupt or for a millisecond time of any sort will allow the controller to idle. Other things will also allow idling.

Examples are **Wait**, **Every**, **SWAP**, **serial.Get**, **keypad.Get**, **any\_object.Lock...**

You can check the effect of running various bits of code if you have a power supply with a current meter on it.

## Defined logic levels

There are more power savings to be had by making sure that every IO pin on the VM2 is pulled to a defined logic level. This is most important if you are have a very power sensitive application, especially one that uses [stop mode](#).

The operating system message [system.Low](#) will set all uninitialised IO pins to the state 'input pulled low' to make sure every uninitialised IO pin is pulled to a defined state.

Usually, the best time to call this is at the end of your init procedure, after all the IO objects have been defined.

(Your init procedure is called by the default *startup* procedure).

## L: ASCII Character Set

The following table shows all of the characters in the ASCII character set, giving the decimal character number, the hexadecimal character number and the character itself. In the case of unprintable characters, either a description is given, or the box is left blank.

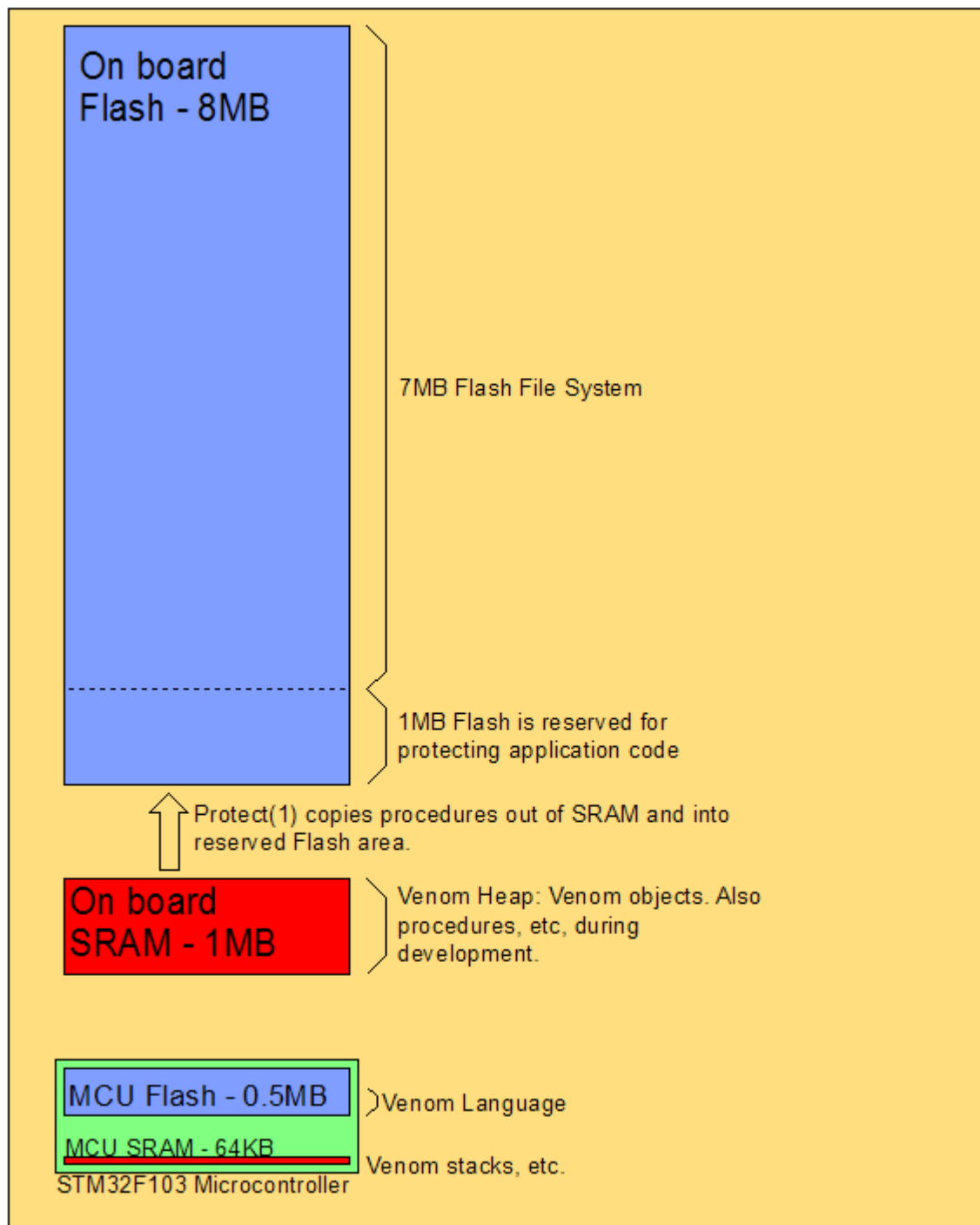
Dec	Hex	Char		Dec	Hex	Char		Dec	Hex	Char		Dec	Hex	Char
0	0	NUL		32	20	SPC		64	40	@		96	60	`
1	1			33	21	!		65	41	A		97	61	a
2	2			34	22	"		66	42	B		98	62	b
3	3	BRK		35	23	£		67	43	C		99	63	c
4	4			36	24	\$		68	44	D		100	64	d
5	5			37	25	%		69	45	E		101	65	e
6	6			38	26	&		70	46	F		102	66	f
7	7	BEEP		39	27	'		71	47	G		103	67	g
8	8	BS		40	28	(		72	48	H		104	68	h
9	9			41	29	)		73	49	I		105	69	i
10	A	LF		42	2A	*		74	4A	J		106	6A	j
11	B			43	2B	+		75	4B	K		107	6B	k
12	C	FF		44	2C	,		76	4C	L		108	6C	l
13	D	CR		45	2D	-		77	4D	M		109	6D	m
14	E			46	2E	.		78	4E	N		110	6E	n
15	F			47	2F	/		79	4F	O		111	6F	o
16	10			48	30	0		80	50	P		112	70	p
17	11	XON		49	31	1		81	51	Q		113	71	q
18	12			50	32	2		82	52	R		114	72	r
19	13	XOFF		51	33	3		83	53	S		115	73	s
20	14			52	34	4		84	54	T		116	74	t
21	15			53	35	5		85	55	U		117	75	u
22	16			54	36	6		86	56	V		118	76	v

23	17			55	37	7		87	57	W		119	77	w
24	18			56	38	8		88	58	X		120	78	x
25	19			57	39	9		89	59	Y		121	79	y
26	1A			58	3A	:		90	5A	Z		122	7A	z
27	1B	Esc		59	3B	;		91	5B	[		123	7B	{
28	1C			60	3C	<		92	5C	\		124	7C	
29	1D			61	3D	=		93	5D	]		125	7D	}
30	1E			62	3E	>		94	5E	^		126	7E	~
31	1F			63	3F	?		95	5F	_		127	7F	DEL



## M: Memory Map (VM2)

This is a memory map for the VM2 and VM2D controllers, with memory areas drawn to scale.



### VM2L

The VM2L controller doesn't have the on-board Flash nor the on-board SRAM fitted. There is

no Flash File System. The MPU's internal Flash and SRAM are the only memories available.

Addresses of the memory areas:

Onboard Flash: \$60000000  
Onboard SRAM: \$64000000  
MCU SRAM: \$20000000  
MCU Flash: \$08000000

## N: Protecting your application

See [OperatingSystem.Protect](#)

## O: Updating Venom2

There are several ways to update the Venom2 Language & Operating System on a VM2 Controller to the new version .

The first two can only be used if you have the VM2 on the bench in front of you:

1. Use VenomIDE to send the new version over the COM port. This is documented in the language Release Note and the VenomIDE Help File.
2. [Program Mode USB access](#)

There is a third method that can be use to *remotely* update a VM2 in the field - you will have to write some Venom code that uses [Protect\(3\)](#). This can also be used on the bench.

## S: Serial settings

These are the serial port settings you will need to talk to a VM2's command line from a terminal emulator:

**115Kbd, 8-NONE-1**

Or:

Baud rate	115,200
Data bits	8
Parity	None
Stop bits	1

Note: make sure the VM2 is in Program Mode by setting the Prog Mode switch or pulling down the Prog Mode pin.

# Credits

---

## Credits

Venom2 was written by the following people:

Compiler	Karl Lam
Runtime system	Karl Lam
Operating System	Karl Lam
TCP/IP stack	Anahata
Filing systems	Anahata
Other objects	Karl Lam and Anahata

The Venom2 Help File was created by Karl Lam and Anahata.

# Index

- -

' 27

- - -

- 17

- ! -

! 33

- " -

" 19, 95

- # -

#DEFINE 547

#ELIF 549

#ELSE 549

#ENDIF 549

#IF 549

#REDEFINE 548

#UNDEF 548

- \$ -

\$ 26

- % -

% 26

- ( -

( 38

- ) -

) 38

- \* -

\* 18

- , -

, 32

- . -

. 40

- / -

/ 18

- : -

: 28

:= 38

- ; -

; 41

- ? -

? 32

- @ -

@ 36

- [ -

[ 39

- ] -

] 39

- ^ -

^ 19

- ~ -

~ 28

- + -

+ 17

- < -

< 22

<< 24

<<< (embedded text) 24

<<<: (embedded text) 24

<= 23

<> 22

- = -

= 21

- > -

> 23

>= 23

>> 24

>>> (embedded text termination) 24

- 1 -

1-Wire Bus 348

- A -

ABS 41

AcceptPrintJob method 165

Accessibility

private 86

protected 87

public 87

AccessPoint object 537

ACOS 42

Active message

TouchScreen: Button 523

Active mode (FTP) 248

Active variable 41

detecting assignment 47

method 60

Address message

Array 130

Ethernet 194

FileSystem 210

HTTP 299

IP 327

SafeData 414

SMS Protocol 443

String Object 458

UDP 478

WifiLink 531

Adjust message

DateTime 169

File System 210

RealTimeClock 405

TouchScreen 508

ALL 42

Alpha LCD 118

cursor 118

AlphaLCD 115

Analogue 119

AD7998, etc 121

MAX1238, etc 121

object 119

on-board 119

PCF8591 122

source impedance 123

AND

bitwise 42

logical 43

AndAlso 43

Any 43

Arc-cosine 42

Arc-Sine 47

Arc-Tangent 47

Array 128

constant 44

multi-dimensional 136

variable 129

AS 46

ASCII 55, 576

Character set, table 576

ASIN 47

Assembler 52

Assembler 52  
     calling 52  
 Assembler code  
     calling 564  
 Asserted message  
     Digital 182  
     Keypad 331  
     OnBoardLED 346  
     PulseWidthOut 398  
     TouchScreen 509  
     TouchScreen: Button 523  
 Assignment  
     keyword 47  
     operator 38  
 At @ operator 36  
 ATAN 47  
 Audio output 124  
 AUTODESTRUCT 47  
 Auto-repeat  
     for Keypad 333  
     for TouchScreen Buttons 519  
     number of auto-repetitions 519  
 AWAIT 49

## - B -

Background  
     Text colour padding 273  
 Backlight control 280  
 Backspace 51  
 Base 50  
 Baud rate setting 432  
 Becomes equal to 38  
 BEEP 51  
 Binary  
     constants in Venom 26  
     print formatting 28  
 Binary executable files 371  
 Bitmap  
     drawing on GraphicsLCD 267  
     embedded in text 290  
     finding size of 270  
     macros 270  
     registering for printing 290  
 Bitmap message  
     AlphaLCD 116  
     GraphicsLCD 267

Bitwise  
     and 42  
     exclusive or 65  
     inversion 73  
     not 73  
     or 80  
     shift left/right 24  
 Block  
     of code 39  
     structure 6  
 BMP - See VBM (venom bitmap files) 267  
 Box message  
     GraphicsLCD 271  
 BREAK 51  
 Break up text into lines 149  
 BS 51  
 BST - British Summer Time 412  
 Buffer 134  
     convert to number 148  
     object 134  
     size of, protocols 554  
 Bus  
     1-Wire (Dallas) 348  
     CAN 151  
     I2C 320  
     RS485 425  
     SPI 452  
 Button  
     drawing - Venom procedure pointer 524  
     greyed-out; inactive 523  
     'key' value 526  
     label 526  
     object 522  
     pressed 523  
     user-defined elements 524  
 Button message  
     TouchScreen 509, 511  
 Button object 522

## - C -

C code  
     calling 52, 564  
 Calendar 404  
 Calendar date calculation 168  
 Calibrate  
     real time clock 405



- Calibrate
  - touchscreen 508
- Call 52
- CanBus 151
  - Filtering 156
- Carat 19
- Carriage Return 62
- CASE 54
- CATCH 99
- CENTRE 54, 287
- Channel message
  - AccessPoint 537
- Character constant 27
- Character Set
  - ASCII 576
  - SMS Protocol 437
- Checksum
  - generator 166
- Checksum message
  - OneWire 350
  - OperatingSystem 356
  - SafeData 414
- CHR 55
- Circle 271
- Class
  - accessing global variables 68
  - Class-default messages 162
  - Has operator 70
  - inheritance checking operator 73
  - interface checking operator: Has 70
  - structure 13
  - user-defined class 55
- Class keyword 55
- Clear screen 62, 287
- Clock 404
  - gearing 376
  - Real Time Clock 404
  - real time, calibrate 405
  - speed, software control 376
- Close message
  - File 238
  - FTPClient 250
  - NumberReader 339
  - POP3 385
  - PPP 486
  - TCP 494
  - UDP 479
- CLS 62, 287
- Code Example
  - SMS Protocol 449
- Colon 28
- Colours 281
- Columns
  - number printing 273
- COM Ports 424
- Comma 32
- comma separated values 474
- Comments 41
  - structure 12
- Compare message
  - AccessPoint 538
  - String Object 458
- Compiler
  - runtime compilation 363
- conditional compilation (#IF, etc) 549
- Connect message
  - Encrypter 187
  - Ethernet 200
  - FileSystem 211
  - WiFiLink 531
- Constant 10
  - character 27
  - folding 574
  - macro 547
  - string 19
  - string - large 95
  - types of 10
- Contiguous files 234
- Control flow
  - IF 71
  - SELECT 89
- Convert
  - between INT, FLOAT, Pointer, ... 46
  - text to number 148
- Cookie message
  - HTTP 300
- Copy
  - files 212
  - moving/copying memory contents 356
- Copy message
  - Array 131
  - FileSystem 212
  - OperatingSystem 356
- COS 62

Cosine 62  
 Count message  
   Ethernet 200  
   FileSystem 213  
   OperatingSystem 357  
   POP3 385  
   PulseCounter 391  
   PulseWidthOut 399  
   Semaphore 422  
   Shaft 436  
   TouchScreen 512  
 CR 62, 287  
 CRC 166  
   32-bit 166  
   CCITT-CRC16 166  
   MODBUS 166  
 CRCGenerator 166  
 Creation  
   AccessPoint 538  
 CSV format  
   reading protocol 474  
   reading records 241  
   writing records 164  
 Ctrl-C Break 426  
 Cursor  
   AlphaLCD 118  
   Move around display 287  
   moving 69  
 Cursor movement  
   GraphicsLCD 288  
 Cursor position  
   GraphicsLCD 286

## - D -

Data structures 8  
   all 8  
   array: constant 44  
   array: variable 128  
   Buffer of Any 136  
   Buffer, text buffer 134  
   String object 456  
 Data type  
   conversion 46  
   floating point 68  
   integer 73  
 Datagrams 477  
 Date 168  
   DateTime object 168  
   Real Time Clock 404  
 Date stamp  
   file 224  
 DateTime 168  
 Day message  
   DateTime 171  
 DayOfWeek message  
   DateTime 171  
 Deadlock 94  
 Debug message  
   CANBus 153  
   Ethernet 200  
   FileSystem 214  
   FTPClient 250  
   FTPServer 256  
   HTTPServer 301  
   OperatingSystem 357  
   PID Controller 383  
   PPP 487  
   SMS Protocol 442  
   SMTP 451  
   TCP 494  
   UDP 479  
   WifiLink 532  
 DEFINE 547  
 Delete  
   command 63  
   File System 216  
   files 222  
   macros 548  
 Dereference pointer 33  
 Derived 63  
 Development  
   checklist 564  
 DHCP 193  
   Ethernet 194  
 Dial up  
   example code 555  
 Die message 114  
   all objects - general discussion 114  
   Array 131  
   Buffer 138  
   Class-default 163  
   I2CBus 321  
   PulseWidthOut 402

- Die message 114
  - SerialPort 426
  - Task 467
  - TCP 496
- Digital 177
  - I2C (PCF8574) 180
  - On-board 178
- Display
  - Alphanumeric 115
  - Graphic 261
- Distribute
  - application code 232
- DIV 64
- Divide
  - Floating point 18
  - Integer 64
- DO 64
- Dollar 26
- Done message
  - File System 215
  - PulseWidthIn 394
  - Task 467
  - Timer 504
- Dot operator 40
- Dotted quad
  - converting to integer 327
  - printing integer as 30
  - Use in IP addresses 554
- Download
  - source code 85
  - Venom language 579
- Draw message
  - TouchScreen: Button 524
- DST - Daylight Saving Time 412
- Dump
  - heap 362
  - memory contents 357
  - stack 362
- E -**
- EEPROM 412
- Element message
  - Array 132
  - Buffer 139
  - CANBus 154
  - File 238
  - POP3 385
  - RealTimeClock 407
  - SafeData 418
  - String object 459
  - TouchScreen: Button 524
- ELSE 64
- Email 384, 450
  - fetching 384
  - sending 450
- Embedded
  - text 95
- Embedded text 24
- Empty message
  - Buffer 139
  - File 238
  - File System 216
  - NumberReader 339
  - SerialPort 426
  - SMS Protocol 443
  - String object 460
  - UDP 479
- Encrypter 185
- END 65
- EOR 65
- Equal
  - assignment 38
  - equality expression 21
- Equality
  - equal 21
  - not equal 22
- Erratum
  - tutorial 13
- Error
  - messages, table 565
- error handling 99
- ErrorAction message
  - OperatingSystem 368
- Escape
  - character 27
  - character in strings 20
- Escape message
  - SerialPort 426
- Ethernet 191
  - setMAC address 198
- Event message
  - TouchScreen 512
- EVERY 66

Example code  
 TCP/IP 555  
 Exception generation 66  
 exception handling 99  
 Exclamation mark operator 33  
 Exclusive OR 65  
 Executable files 371  
 EXIT 66  
   loop 51  
 EXP 67  
 Exponential 67

## - F -

FALSE 68  
 FAQ 567  
 FIFO 134  
 File 236  
   close 238  
   copy 212  
   delete 222  
   empty (remove contents) 238  
   length 243  
   length remaining 245  
   name 243  
   open 219  
   random access 238  
   read 239  
   read point 245  
   rename 219  
   rewind 245  
   search text 239  
   size 218  
   timestamp (of file by name) 224  
   timestamp (of open file) 245  
   write 244  
 file name matching 229  
 File System 204  
   compact/defragment 210  
   connect to USB 211  
   connect via serial port 211  
   copy file 212  
   create 205  
   debug information 214  
   file address in memory 210  
   find file 216  
   Flash, internal 208

flush cache 217  
 free space 217  
 list 226  
 number of files 213  
 progress indication 215  
 RAM disk 207  
 remove all files 216  
 reset 222  
 SD card 205  
 status 223  
 timestamp of file by name 224  
 USB - external device 209  
 validation 224

File transfer  
   over serial port 211  
   over USB 211

File Transfer Protocol 254

Files  
   contiguous in memory 234  
   number of 213  
   transfer, XMODEM 541

Filesystem 204

FILO 134, 143

Find message  
   Array 132  
   Buffer 140  
   File 239  
   FileSystem 216  
   I2C Bus 322  
   IP 328  
   OneWire 351  
   String object 460  
   TextAnalyser 471  
   TouchScreen 513  
   UDP 479  
   WiFiLink 533

Firmware update 579

Flags  
   runtime 357

Flash file system 208

Flash memory  
   protect application in 371

Flash message  
   OnBoardLED 346

FLOAT 68

Floating I/O - find and set to defined state 369

Floating point

Floating point  
     conversion 68  
     number from text 148, 469  
     precision 572

Flush message  
     Buffer 143  
     FileSystem 217  
     HTTP 302  
     Keypad 332  
     String object 461  
     TCP 497  
     XMODEM 543

Font  
     changing 68  
     data formats 276  
     defining new 275  
     pre-defined 289

FONT print keyword 68

FontData message  
     GraphicsLCD 275

Foreign code 52  
     calling 564

FOREVER 68

Format  
     DateTime print output 174  
     font data 276  
     RealTimeClock print output 412

Format message  
     GraphicsLCD 273  
     HTTP 302  
     SerialPort 426  
     SMS Protocol 443

Formatting 28  
     binary 28  
     floating point numbers 30  
     hexadecimal 28  
     integers 29  
     IP dotted-quad style 30  
     objects 31  
     Printf 110  
     strings 31

Free message  
     CanBus 155  
     FileSystem 217  
     OperatingSystem 369  
     SerialPort 427  
     String object 461  
     TCP 497

XMODEM 544

FTP 248  
     FTPClient 248  
     FTPServer 254  
     Function 88

## - G -

Garbage collection  
     AUTODESTRUCT attribute 47  
     Buffer of Any 138  
     description 8  
     garbage scanner 360

Get message  
     Buffer 143  
     CanBus 155  
     CRCGenerator 167  
     Encrypter 187  
     File 239  
     FTPClient 251  
     HashGenerator 292  
     HTTP 304  
     I2CBus 322  
     Keypad 332  
     OneWire 352  
     PrintJob 388  
     RandomNumberGen 403  
     SafeData 419  
     Semaphore 422  
     SerialPort 427  
     SMS Protocol 444  
     String object 461  
     TCP 497  
     TextAnalyser 471  
     UDP 480  
     XMODEM 545

GetLast message  
     Buffer 143  
     Keypad 332  
     RealTimeClock 406  
     TextAnalyser 475

Global name assertion 68

Global variable 10

Global variables  
     accessing inside a Class method 68

Glossary 568  
     networking 553

GMT 412  
 Go message  
     FTPClient 251  
     FTPServer 257  
     IP 329  
     PulseWidthIn 394  
     PulseWidthOut 399  
     Timer 505  
 GOTOXY 69, 287  
 GPRS 484  
 GraphicsLCD 261  
     backlight control 280  
     text cursor position 286  
 Greater than 23  
 GSM 7 bit characters  
     SMS Protocol 437  
 GUI  
     graphical displays 261  
     touchscreen input 507

## - H -

Half-duplex 429  
 handle  
     errors and exceptions 99  
 Handshake message  
     SerialPort 428  
 Has operator 70  
 HashGenerator 291  
 Heap  
     dump 362  
 Height message  
     TouchScreen: Button 525  
 HELP 70  
     File 246  
 Hexadecimal  
     constants in Venom 26  
     number from keypad 338  
     number from text 469  
     print formatting 28  
 High message  
     Digital 183  
     OneWire 353  
 HOME 70, 287  
 Hour message  
     DateTime 172  
 HTAB 287

HTML  
     embedded in venom code files 95  
     embedding in Venom 24  
 HTTP 293  
 HTTP minimal web server 295  
 HTTPServer 293

## - I -

I2C Bus 320  
     repeated start condition 325  
     software based bus 325  
 IF 71  
 Impedance  
     analogue input 123  
 INDEX 72  
 INDEX0 72  
 Inheritance 59  
     checking - 'Is' operator 73  
     of a Venom base class 61  
 INI file format  
     reading 241  
     writing 164  
 Input stream  
     tokenise 469  
 InputBuffer message  
     Keypad 333  
     TextAnalyser 476  
 Insert message  
     Buffer 144  
 INT 73  
 Integer  
     max, min values 572  
     number from text 148, 469  
 Interface  
     User Class, 'Has' operator 70  
 INV: bitwise inversion operator 73  
 Inverse  
     bitwise 73  
     logical 74  
 Invert colour 281  
 IP 326  
 IProt 326  
 Is operator 73  
 IsFalse 74

## - J -

Jumping out of nested loops 66  
 Justification 287  
   CENTRE 54  
   LEFT 75  
   RIGHT 88

## - K -

Key message 330  
   AccessPoint 539  
   Encrypter 188  
   Keypad 334  
   OperatingSystem 369  
   TouchScreen 513  
   TouchScreen: Button 526  
 Keypad 330  
 Killing objects 114

## - L -

LAN 191  
 language 2, 4  
 Lazy  
   and 43  
   or 81  
 LCD  
   Alphanumeric 115  
   Graphic 261  
 LCD PWM backlight control 280  
 Leak  
   detector, for memory leaks 360  
 LED 345  
 LEFT 75, 287  
 Length message  
   "string constant" 19  
   Array 133  
   Buffer 144  
   Class-default 163  
   File 243  
   Filesystem 218  
   NumberReader 339  
   POP3 386  
   SMS Protocol 445  
   String object 461

UDP 481  
 Less than 22, 23  
 Line message  
   GraphicsLCD 280  
   RealTimeClock 407  
 Line of text  
   read from serial port 427  
 Lines  
   break up text into 149  
 LIST 75  
   keywords 357  
   memory contents 357  
 LOCAL 75  
 Locale  
   day and month names 365  
   decimal point character 359  
 Lock  
   Semaphore 421  
 Lock message  
   any object 108  
   Semaphore 423  
 LOG 76  
 Logarithm 76  
 Logical  
   and 43  
   not 74  
   or 81  
 longjmp(), equivalent 66  
 Look message  
   CanBus 156  
   SerialPort 430  
   TextAnalyser 476  
 Looping  
   DO 64  
   EVERY 66  
   FOREVER 68  
   INDEX 72  
   loop count, automatic 72  
   REPEAT 87  
   WHILE 104  
 Low message  
   Digital 183  
   OneWire 353  
   OperatingSystem 369  
 Low Power Mode  
   Stop Mode 409

# - M -

- MAC address 191
  - Set 198
- Macros 547
  - redefine 548
  - undefine, remove 548
- Magnitude 41
- MAKE 77
  - AlphaLCD 115
  - Analogue 119
  - Array 129
  - Buffer 135
  - CanBus 153
  - CRCGenerator 166
  - DateTime 169
  - Digital 178
  - Encrypter 186
  - Ethernet 191
  - FileSystem 205
  - FTPClient 249
  - FTPServer 256
  - GraphicsLCD 262
  - HashGenerator 292
  - HTTP 299
  - I2CBus 321
  - IP 327
  - Keypad 330
  - NIL 338
  - NumberReader 338
  - OnBoardLED 346
  - OneWire 350
  - OperatingSystem 356
  - PID Controller 381
  - POP3 385
  - PPP 485
  - PulseCounter 390
  - PulseWidthIn 392
  - PulseWidthOut 397
  - RandomNumberGen 403
  - RealTimeClock 405
  - SafeData 413
  - Semaphore 422
  - SerialPort 425
  - Shaft 435
  - SMS Protocol 442
  - SMTP 451
  - SPI 453
  - Stopwatch 465
  - String object 457
  - Task 467
  - TCP 493
  - TextAnalyser 470
  - Timer 504
  - Touchscreen 507
  - UDP 478
  - WiFiLink 529
  - XMODEM 542
- Map
  - of VM2 memory 578
- Mapping message
  - CanBus 156
  - HTTPServer 318
  - NumberReader 340
  - TouchScreen 515
- Match message
  - HTTPserver 304
- Maximum integer 572
- Memory
  - direct access 32
  - dump contents 357
  - leak detector 360
  - moving/copying contents 356
- Memory map 578
- Memory pointer
  - to file 210
- Message
  - indirect message send 35
  - reference to message ('message pointer') 37
  - send message (dot) 40
- Message redirection 61
- Method
  - active variable 60
- Methods 57
- Minimum integer 572
- Minus 17
- Minute message
  - DateTime 172
- MOD 77
- MODBUS CRC 166
- Modem 484
- Modulo 77
- Monospaced 289
- Monospaced text 273



Month message  
     DateTime 172

Move  
     moving/copying memory contents 356

Multi-bit ports  
     creation 181  
     reading and writing 184

Multicast  
     Ethernet 194

Multiply 18

Multitasking  
     description 8  
     disable 360  
     language structure 13  
     switching off 467  
     switching on 468  
     syntax 90  
     use with network protocols 554

## - N -

Name message  
     AccessPoint 539  
     Class-default 164  
     File 243  
     FileSystem 219  
     FTPClient 252  
     HTTP 305  
     PPP 488  
     TouchScreen: Button 526

Name scope  
     global scope assertion 68  
     local names take precedence 75  
     member name scope assertion 97

Names  
     declaration (local) 75  
     scope 7

Networking  
     guide 553

NEW 78

New features 13

NIL 79, 337

No operation 80

Non-proportional text 273

Non-volatile storage  
     EEPROM 412  
     STM32 backup registers 407

NOP 80

Not operator  
     bitwise 73  
     logical 74

Notes on Operation  
     FTPServer object 258

Notes on Using File Systems  
     File Systems 228

Nothing 79

Number  
     input from keypad 338  
     input from text 148, 469  
     limits 572

Number entry  
     PIN 341  
     secret 341

NumberReader 338

Numbers  
     printing in columns 273

## - O -

Object  
     creation 78

Object orientation 7

objects 6

Off message  
     CanBus 158  
     Digital 183  
     ethernet 201  
     GraphicsLCD 280  
     OnBoardLED 347  
     PulseWidthOut 399  
     SPI 455  
     Task 467

On message  
     CanBus 158  
     Digital 183  
     ethernet 202  
     GraphicsLCD 280  
     OnBoardLED 347  
     PulseWidthOut 399  
     SPI 455  
     Task 468

OnBoardLED 345

OneWire Bus (Dallas) 348

Open

Open  
   file 219

Open message  
   FileSystem 219  
   FTPClient 252  
   POP3 386  
   PPP 489  
   TCP 499  
   UDP 481

Operating System 355

OperatingSystem object 355

Operator  
   precedance 572

Optimisation 574

Optional  
   parameters 39  
   parameters count 81

OR  
   bitwise 80  
   logical 81

OrElse 81

Output message  
   HTTPserver 307  
   NumberReader 341  
   OperatingSystem 370

OutputBuffer message  
   SerialPort 430

Overlay  
   code 363

Owner message  
   any object 108  
   Semaphore 423

## - P -

ParamCount 81

Parameter - keyword 82

Parameter list  
   actual 39  
   formal 39  
   get parameter by index 82  
   optional parameters in Venom procedures 39

Parenthesis 38

Parity 426

Parse text 469

Passive mode (FTP) 248

password

PPP 488

Pen message  
   GraphicsLCD 281

Percent 26

Period message  
   Analogue 123  
   HTTP 308  
   IP 329  
   Keypad 336  
   PPP 489  
   PulseWidthIn 394  
   PulseWidthOut 399  
   Timer 505

PID Controller 379

PIN entry 341

Ping  
   example code 555

Pling 33

Plus 17

Pointer  
   creation 36  
   dereference (follow) 33  
   procedure - create 37  
   procedure - dereference 33  
   to a message / method 37

POP3 384

Ports  
   create multi-bit 181  
   reading and writing multi-bit 184

Power consumption  
   minimising 410

Power of (^ operator) 19

PPP 484

Precision  
   floating point 572

Pre-processor  
   conditional compilation 549  
   macros (#define) 547

Print  
   "string constant" 31  
   AccessPoint 540  
   general 9  
   HashGenerator 293  
   PPP 491  
   PRINT command 83  
   PRINT TO - print redirection 84  
   redirection 99, 370

- Print
  - WiFiLink 536
- PRINT message
  - Analogue 127
  - Array 134
  - Buffer 150
  - Class-default 164
  - DateTime 174
  - Digital 185
  - Ethernet 203
  - File 247
  - FileSystem 226
  - formatting 28
  - FTPClient 253
  - HTTP 315
  - I2CBus 325
  - IP 329
  - NumberReader 345
  - OnBoardLED 347
  - OneWire 354
  - OperatingSystem 379
  - POP3 387
  - PulseCounter 391
  - PulseWidthIn 396
  - PulseWidthOut 402
  - RandomNumberGen 404
  - RealTimeClock 412
  - SafeData 421
  - Semaphore 424
  - Shaft 437
  - SMS Protocol 449
  - Stopwatch 466
  - String object 464
  - Task 469
  - Timer 506
  - TouchScreen: Button 527
- PRINT TO message
  - AlphaLCD 118
  - Buffer 149
  - Class, user defined 165
  - DateTime 174
  - Encrypter 190
  - File 247
  - GraphicsLCD 287
  - HTTP 316
  - NumberReader 344
  - ppp 492
  - RealTimeClock 411
  - SerialPort 434
  - SMS Protocol 448
  - String object 464
  - TCP 503
  - UDP 484
- PrintF
  - command 105
  - TextBlock format string in HTTP 316
- PrintF message
  - [for all objects] 110
  - Class-default 165
- PrintJob object 387
- Private 86
- Procedure
  - entering 98
  - pointer 37
  - pointer - dereference 33
  - stack 13
  - structure 12
- Production programming of VM2s 232
- PROGRAM 85
- Programming
  - production programming of VM2s 232
- Progress indication
  - file operations 215
- ProtAnalyser 469
- Protect message
  - AccessPoint 540
  - OperatingSystem 371
  - WiFiLink 534
- Protected 87
- Protected Application Area 371
- Protocol
  - analyser 469
- Public 87
- Pulse generation 396
- Pulse measurement 391
- Pulse message
  - Digital 184
- PulseCounter 389
- PulseWidthIn 391
- PulseWidthOut 396
- Put message
  - AlphaLCD 117
  - Buffer 144
  - CanBus 159
  - CRCGenerator 167

Put message  
 Encrypter 189  
 File 244  
 FTPClient 250  
 HashGenerator 293  
 HTTP 309  
 I2CBus 323  
 NumberReader 342  
 OneWire 353  
 SafeData 420  
 Semaphore 423  
 SerialPort 431  
 SPI 456  
 String object 461  
 TCP 501  
 UDP 481  
 XMODEM 544

PWM  
 input 391  
 output 396  
 VM2-D2 backlight signal 280

## - Q -

Quadrature 434  
 Query operator 32  
 Question mark 32  
 Queue message  
 Analogue 124  
 Buffer 145  
 CanBus 160  
 Encrypter 190  
 File 245  
 HTTP 311  
 Keypad 336  
 PrintJob 388  
 PulseWidthOut 400  
 SerialPort 432  
 String Object 462  
 TCP 501  
 TextAnalyser 476  
 UDP 483  
 XMODEM 544

Quote  
 double 19  
 single 27

## - R -

RAM File system 207  
 Random numbers 402  
 RandomNumberGen 402  
 Read  
 from memory 32  
 line of text from serial port 427  
 Read a number  
 from a string 463  
 Readpoint message  
 Buffer 145  
 File 245  
 String Object 462  
 Real Time Clock 404  
 calibrate 405  
 RealTimeClock 404  
 creation 405  
 Records  
 retrieving from file 239  
 retrieving from SafeData 419  
 storing in file 244  
 storing in SafeData 420  
 Recursion 571  
 REDEFINE 548  
 Redefining macros 548  
 Redirect  
 HTTPServer 310  
 Redirection  
 all output 370  
 PRINT output 99  
 Reference  
 to a message ('message pointer') 37  
 Reformat  
 File System 216  
 Register - write to directly  
 AlphaLCD 118  
 Remove message  
 Buffer 146  
 FileSystem 222  
 FTPClient 252  
 POP3 387  
 SMS Protocol 446  
 TouchScreen 518  
 Removing objects 114  
 Rename

Rename  
     file 219  
 REPEAT 87  
 Repeated Start Condition  
     I2C Bus 325  
 Reset  
     source of last reset 362  
 Reset message  
     AlphaLCD 117  
     Buffer 146  
     CRCGenerator 168  
     File 245  
     FileSystem 222  
     GraphicsLCD 284  
     HashGenerator 293  
     I2C Bus 324  
     NumberReader 343  
     OneWire 354  
     OperatingSystem 375  
     PID Controller 383  
     PPP 491  
     PulseCounter 391  
     RandomNumberGen 404  
     RealTimeClock 409  
     SafeData 420  
     Shaft 436  
     Stopwatch 465  
     String object 462  
     TCP 501  
     TextAnalyser 476  
     Timer 505  
     XMODEM 545  
 Result 88  
 RETURN 88  
 RIGHT 88, 287  
 Robust applications 563  
 ROMing applications 563  
 RS232 424  
 RS485 424, 429  
 RTC 404  
 Run message  
     FTPServer 257  
     OperatingSystem 375  
 Runmode message  
     OperatingSystem 376  
 Runtime error  
     improving line number accuracy 80

messages, table of 565

## - S -

SafeData 412  
     records 419, 420  
 Scan for access points (Wifi) 533  
 SD Card 205  
 Second message  
     DateTime 172  
 Secret number entry 341  
 SELECT 89  
     Case 54  
 Semaphore 421  
 Semicolon 41  
 Send message  
     Analogue 124  
     I2C Bus 324  
     SMS Protocol 447  
     SMTP 452  
     UDP 482  
 Serial FTP 211  
 Serial port settings 579  
 Serial ports 424  
 SerialPort 424  
 setjmp(), equivalent 99  
 Settings  
     terminal 579  
 SHA-2 291  
 SHA-256 291  
 Shaft 434  
 Shaft encoder 434  
 Shift - bitwise 24  
 Signal Strength 536  
 SIN 90  
 Sine 90  
 Sleep 409  
 SMS  
     Protocol Object 437  
 SMTP 450  
 Sort message  
     Array 133  
     Buffer 146  
 Sounds - playing 124  
 Source file annotation 85  
 Source impedance

Source impedance  
 analogue input 123

Speed  
 execution, of code 573

Speed message  
 CanBus 160  
 OperatingSystem 376  
 PulseWidthIn 395  
 PulseWidthOut 401  
 SerialPort 432

SPI 452

SQRT 90

Square brackets 39

Square root 90

Stack 134  
 dump 357, 362  
 overflow 571  
 set task stack size 364  
 user 143  
 Venom 13

START 90  
 Task object 467

Startup  
 sequence 562

State message  
 Task 468

Statements 11

Status message  
 CanBus 161  
 Ethernet 202  
 FileSystem 223  
 PrintJob 388  
 TCP 502  
 WifiLink 535

STOP 91

Stop bits 426

Stop Mode 409

Stopwatch 465

String 456  
 blocks of text 24, 95  
 capacity - find 461  
 compare 458  
 concatenate 461, 464  
 constant: "quoted string" 19  
 convert to number 148, 469  
 extract a substring 464  
 find a substring 460

handling - general discussion 8

object 456

variable 456

SWAP 93

switch statement - equivalent in Venom2 89

Syntax descriptions  
 keywords 16  
 objects 107

System  
 OperatingSystem object 355

## - T -

Tab  
 GraphicdLCD, explicit CR depth 273  
 GraphicsLCD - horizontal and vertical 288

TAN 93

Tangent 93

Task  
 blocking 94  
 deadlock 94  
 listing active tasks 94  
 number of tasks 359  
 object 467  
 reinstate task switching 468  
 stack use 364  
 starting 90  
 state variable 468  
 stop task switching 467  
 stopping 91  
 swapping 93

TCP 493

TCP/IP 326, 493  
 example code 555  
 Notes 554

TCProt 493

Terminal settings 579

TestLock message  
 any object 108  
 Semaphore 423

Text  
 append 567  
 break into lines 149  
 buffer 134  
 convert to number 148, 463, 469  
 divide into lines 567  
 embedded 95

- Text
    - embedding 24
    - find string within 567
    - handling 567
    - in-line 95
    - insert 567
    - manipulation 134
    - substring - extract 567
  - Text clipping 273
  - Text cursor
    - GraphicsLCD 286
  - TextAnalyser 469
  - TextBlock 95
  - TextBox message
    - GraphicsLCD 284
  - THEN 97
  - This 97
  - Tilda 28
  - Time message
    - DateTime 173
    - Ethernet 203
    - File 245
    - FileSystem 224
    - IP 329
    - Keypad 336
    - OperatingSystem 378
    - RealTimeClock 409
    - SMS Protocol 448
    - Stopwatch 465
    - Timer 505
    - TouchScreen 519
    - UDP 482
  - Time zones 412
  - Timeout message
    - FTPClient 253
    - RealTimeClock 409
    - SerialPort 433
    - TCP 502
    - TouchScreen 519
  - Timer object 504
  - Timestamp
    - of file by name 224
    - of open file 245
  - Timing
    - Stopwatch 465
    - Timer 504
  - TO 98
  - Toggle message
    - Digital 184
    - OnBoardLED 347
  - Token
    - read 469
  - Touchscreen 507
    - events 512
    - position of touch 521
    - sensitivity 520
  - Transparent colour 281
  - TRUE 99
  - TRY 99
  - Tutorial
    - additions and corrections to first edition 13
  - Type
    - checking - 'ls' operator 73
    - conversion 46
    - finding the type of a value 101
  - Typeface 68
  - TypeOf operator 101
  - Types
    - weakly-typed language 6
- ## - U -
- UART 424
  - UDP 477
  - UNDEF 548
  - Unlock message
    - any object 108
    - Semaphore 423
  - Unsigned 103
  - Unused I/O - define logic states 369
  - Update
    - File 246
    - Venom language 579
  - Update message
    - DateTime 173
    - GraphicsLCD 285
    - Keypad 337
    - PID Controller 383
  - Upgrade
    - Venom language 579
  - USB
    - access to Flash File System 232
    - access to Flash File System - runtime 211
    - VM2 as mass storage device 232

USB external file system 209  
 user name  
   PPP 488

## - V -

Valid message  
   DateTime 173  
   Ethernet 202  
   FileSystem 224  
   HTTP 312  
   OperatingSystem 379  
   PPP 491  
   RealTimeClock 411  
   SerialPort 433  
   TCPProt 502  
   TextAnalyser 477  
   WifiLink 535

Value message  
   AccessPoint 540  
   AlphaLCD 118  
   Analogue 127  
   Buffer 148  
   CanBus 162  
   CRCGenerator 168  
   Digital 184  
   HTTP 313  
   NumberReader 344  
   PID Controller 383  
   RandomNumberGen 404  
   String object 463  
   Touchscreen 520  
   WiFiLink 536

Variables  
   active 41  
   global 10  
   local 10, 75  
   names 10  
   scope 7

VBM - Venom Bitmap files 267

Venom2  
   description 6

Venom-SC  
   language structure 9

Version  
   read as integer 357

vfu files 371

VTAB 287

## - W -

WAIT 103

Waiting  
   AWAIT 49  
   WAIT 103

Wake from Stop Mode 407

WAV files 124

Web server 293, 554

WHILE 104

Width message  
   NumberReader 344  
   PulseWidthOut 401  
   TouchScreen: Button 526

WiFi security 534

WifiLink object  
   WifiLink 527

wildcard 229

Wireless Networking 527

WORD 104

Word wrapping (GraphicsLCD.Format) 273

Write  
   to memory 32

Write protection  
   internal flash - setting, reading 365

## - X -

XMODEM 541

XOFF 428

XON 428

XPos message  
   GraphicsLCD 286  
   Touchscreen 521  
   TouchScreen: Button 527

## - Y -

Year message  
   DateTime 173

YPos message  
   GraphicsLCD 286  
   Touchscreen 521  
   TouchScreen: Button 527



---

## - Z -

Zero Power 409

Zero-memory objects 114

