Venom2 Tutorial

Written by Karl Lam, Micro-Robotics Ltd

Table of Contents

Venom2 Tutorial

Written by Karl Lam, Micro-Robotics Ltd

Document D043 Version 2018 01 02

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

Micro-Robotics Ltd.
The Old Maltings
135 Ditton Walk
Cambridge
CB5 8QB
Tel: +44 (0) 1223 523100
Fax: +44 (0) 1223 524242
sales@microrobotics.co.uk
www.microrobotics.co.uk

Getting Started

This section introduces you to the VenomIDE development system, and also to a little of the Venom2 language.

If you can't or don't want to use VenomIDE, please start the tutorial here.

Getting started guide

The exact details of connecting your VM2 controller to your PC are given in the paper document *Getting Started Guide* in the starter kit. Please refer to this now, if you haven't already. The *Getting Started Guide* will take you as far as seeing the Venom startup message in the Terminal window:

```
VM2 Control Computer running Venom2 at 72MHz
Version 2013 06 03
Copyright 2008-2013 Micro-Robotics Ltd.
Clear RAM?
```

Type Y – this tells the controller to clear its RAM – the memory where programs are held during development.

You may need to click in the terminal window before it will accept any characters you type.

The controller then creates a few default procedures for you; you will find out more about these later:

```
Loading procedure startup... Procedure Defined
Loading procedure init... Procedure Defined
-->_
```

The flashing cursor, _, will be positioned just after the **-->** arrow. This arrow is called the *prompt* and means Venom is waiting for your instructions.

At this stage you may want to make the IDE window bigger, and the terminal window larger within it, so you can see more lines of text, and longer lines too.

Simple Commands

Try hitting the Enter key on the PC keyboard a few times. You will notice that Venom replies with a new prompt on a new line. This is a quick way of checking that Venom is talking to you.

Now try typing the following (press Enter at the end of the line). Here, the bits in **bold** are what you type, and the rest is Venom's response.

```
-->Print "hello"
hello-->
```

Venom responds to the command by printing the string you gave it back to the terminal window.

Now try the command below. Don't forget to type the dot between the two words.

If you make a mistake in your typing, then you can use **Backspace** (←) to remove the characters you have entered.

```
-->led.On
-->
```

To see the effect of this command you will need to look at the small LED indicator on the VM2 controller.

The LED will light up. If you repeat the command substituting the word **Off** for **On**, the LED will be turned off.

Objects

An object is a part of the Venom language that will control a device in response to a fixed set of messages. In the example above, **led** was the object responsible for controlling the LED device on the controller. **On** was the message sent to the led object. The dot (**.**) tells Venom that a message follows. Objects will be covered in much greater detail later. For now it is enough to know what it looks like when an object is being used.

Incidentally Venom commands are not case sensitive, so you can use any combination of UPPER and lower case letters when writing Venom code.

The Command Line

The text that you type in at the **-->** prompt is called the *command line*.

Errors

If you made any mistakes in the examples above, Venom probably issued an error message. In case you haven't seen an error message yet, type in **led..On**. You will see:

```
-->led..on
       ^
Syntax Error: Expected message name
Command line not executed.
-->
```

Venom issued a Syntax Error message, meaning it didn't understand the command. The offending line is listed together with a pointer to where Venom thinks the error is (the **^** character), and the reason Venom didn't like it.

Syntax errors, like the one above, will only show up when code is sent to the VM2. There is another type of error that can occur: *runtime errors*. These will be dealt with later.

Simple Procedures

The commands shown above were very simple. Commands may be grouped together into *procedures* that perform more complicated functions. Try the following line, taking care to include the dots and spaces.

```
-->To blip led.On Wait 1000 led.Off End
Procedure defined
-->
```

The keywords **To** and **End** tell Venom that the commands in-between should be treated as a single command (or procedure) called **blip**. Incidentally, the **Wait 1000** command tells Venom to do nothing for 1000 milliseconds.

Try issuing blip as a command:

```
-->blip
-->
```

The LED should turn on for one second then turn off again. The new prompt will only appear once the procedure has finished.

Blip could also be issued as a command from within a procedure. The following procedure 'calls' blip once, waits for a second and then calls blip again. Try entering it and then typing 'double'.

```
-->To double blip Wait 1000 blip End
```

It is not necessary to enter procedures on a single line. The blip procedure could have been entered as below, or in any form where the spaces are replaced by new line or tab characters - i.e. any 'white space' is allowed as a separator.

```
-->To blip
02>led.On
03>Wait 1000
04>led.Off
05>End
Procedure Defined
-->
```

You will notice that the prompt is different during entry of the procedure. This tells you that Venom will not act on the commands you type immediately, and also lists the line numbers of the procedure.

Program files

Simple procedures may be typed in at the command line as shown above. When you want to write a full program it is useful to be able to keep all your procedures together in one or more files. This can be done using the text editor in the IDE. We have adopted the file extension **.vnm** for Venom program files.

To create a Venom file follow these steps:

1. Create a new Venom file using the menu **File ▸New**.
2. Give it a name by saving it: **File ▸Save As...**.
3. If you don't give the file an extension, it will be saved as a Venom file (**.vnm**), which is what we need.
4. You may want to create a new folder somewhere on your PC to save the file into. You can do this from within the **Save As...** dialogue.
5. Now type the code of one the procedures above into the Venom file. This is now your Venom program file.
6. If you want your code to run as an application, define a procedure called **main**.

Example file

This is an example for the contents of your first program file; you can copy and paste this text into your own file.

```
; Main is called at startup.
To main
  Print "Hello world", CR
  blip
End

;Blip the LED.
To blip
  led.On
  Wait 1000
  led.Off
End
```

Syntax highlighting

Notice that the text in your file is colour coded: this is Venom syntax highlighting. It allows you to see the structure of a Venom program more easily by highlighting different elements of the language with different styles. It can also show you when you have misspelled a Venom keyword, or when you are trying to use a Venom reserved word for one of your own variable names.

Downloading a file

Once you are happy with your program file, use **Terminal ▸Download** (shortcut **F7**) to send it to the terminal. This is equivalent to typing in the procedure, but much faster. Note that when you download a file like this you don't see each individual line of the procedure - just a short report.

```
-->PROGRAM "New1.vnm" 0 $00001B9F 7
-->
```

You can now call any of your procedures just by typing their name at the command line as before:

```
-->blip
-->
```

Or you can run your whole program by hitting **F10** or clicking on the **Run** icon in VenomIDE.

Syntax errors during download

Any syntax errors in the code will be reported on the terminal window as the file downloads.

Each syntax error will indicate the filename and line number that the error occurred in - see the line shown in **bold** below (it won't be bold in the terminal window):

```
-->PROGRAM "New1.vnm" 0 $00000F46 7
 led.on wait 1000 led..on
                      ^
Syntax Error: Expected message name (new1.vnm line 2)
1 Syntax Error(s): procedure not defined.
End of file "new1.vnm"
1 error(s)
-->
```

Try introducing such an error into your program and downloading it... then try double clicking on the line in the terminal window that lists the filename and line number (shown bold above)

Double clicking on any line in the terminal window that has a filename and a line number on it takes you to that place in your program - so you can instantly see where your errors are coming from - and then correct them.

Help

There are several sources of help available:

Help Files in CHM format

The firstly there is the the Venom Tutorial: that's what you are looking at now.

There are also other Help Files:

- the Venom2 Help File, for help on the Venom2 Language
- the VenomIDE Help File, for help on the VenomIDE development system.

All of these are available from within VenomIDE, in the menu **Help ▸ Venom2 language help ▸ ...**

You can also get help on a specific Venom keyword by placing the cursor on a word in the editor, right-clicking and choosing

Help on:

Interrogating Venom

Another source of help is in Venom language itself: Venom has a simple on-board Help command that allows you to interrogate the runtime system. It may not always have the information you are looking for, but it can be useful. Try this:

```
-->Help led
 It is the OnBoardLED. (Printing it may give more information)
```

Help will tell you information about the word you type after it - typically what type of thing it is - and how to get more information.

In Venom, printing something will often give you information about it. For example, `System` is a predefined object that represents the Venom operating system:

```
-->Print system
Source files:
 working.vnm
Symbol table: 76 bytes
11 Global variables
Heap: Total 1038336, Free 1035484 (Contig. 1035048), Used 2852.
```

SUMMARY

- You have seen how to talk to Venom, issue commands, build simple procedures and edit them.
- You have seen how to use some of the basic features of the IDE to communicate with the controller, write programs and download them.

What next?

You should now go on to read the next chapter of Venom language tutorial, Repeating and Deciding.

Later, you might also like to learn more about the VenomIDE development tools by reading about them in the *VenomIDE Help File*.

Part 1:Venom Language Tutorial

This part of the tutorial takes you through the features of the Venom programming language (as distinct from the library of objects covered in Part 2).

Carry on

Repeating and Deciding

It is often desirable for a command to be carried out several times or for it to be carried out only if certain conditions are met. This is called Flow Control, and the language keywords that do this are described in the following pages.

Repeating Commands: Repeat, Forever

Often it is useful to simply repeat a command or a set of commands – there are three 'constructs' that do this: `Repeat`, `Forever` and `Every`.

Repeat is used to execute a command a pre-determined number of times. For example the following line prints the string 5 times:

```
-->Repeat 5 Print "And again", CR
And again
And again
And again
And again
And again
-->
```

The keyword `CR` at the end of the `Print` command tells Print to send a Carriage Return after the string.

Similarly `Forever` repeats a command forever.

```
-->Forever Print "And again", CR
And again
And again
And again
```

… *(and so on)* …

Stopping your program

Whenever Venom is executing a command, it may be asked to stop by pressing **Ctrl-C** on your PC keyboard. The next example shows the effect of stopping a command:

```
-->Forever Print "And again", CR
And again
And again
And again
…
```

(user presses **Ctrl-C***)*

```
Runtime error 2: Escape via CTRL-C
in the command line.
-->
```

Escape – though not really an error – is handled as a runtime error by Venom as this allows it to use all of the error handling features in the language.

Shortcuts

Note: you can hit Esc, F9, or use the BRK icon from within VenomIDE to achieve the same action: Stopping your application code.

Timed Loops: Every

The **Every** construct is similar to **Forever** except that the command is executed periodically, with a period specified in milliseconds. The following example prints the string once every second:

```
-->Every 1000 Print "And again", CR
And again
And again
And again
```

(user presses Ctrl-C)

```
Runtime error 2: Escape via CTRL-C
in the command line.
-->
```

Again, the command had to be interrupted with Ctrl-C.

If the code inside the **Every** construct takes longer than the given period then **Every** will not attempt to make up any lost time: that particular loop will just take longer; the next will be the normal length.

Loop Count: Index, Index0

In all looping constructs, the keyword **Index** expresses the 'loop count'. For example, the following prints the numbers 1 to 5.

```
-->Repeat 5 Print Index, CR
     1
     2
     3
     4
     5
-->
```

Note that the **Index** starts at 1 – there is another keyword called **Index0** that starts at 0.

Grouping Commands: [ ]

In all the above examples, only one command is repeated. If more than one command is to be repeated, they should be grouped into a 'block' of commands with the square bracket symbols `[` and `]`.

As far as the loop constructs are concerned, a block is just treated as a single command, so the following repeats the printing and 'toggles' the LED 4 times:

```
-->Repeat 4 [Print "Toggling the LED" , CR led.Toggle]
Toggling the LED
Toggling the LED
Toggling the LED
Toggling the LED
-->
```

If the square brackets were not included, the printing would have been done 4 times but the LED would only have been toggled once.

Making Decisions: If; Else

As well as repeating commands, it is also useful to be able to execute commands only if certain conditions are met. This is achieved using the `If` construct.

The following example uses a variable called a, which we need to define using the Becomes-equal-to symbol, `:=`. Variables are covered in greater detail later on. The `<` symbol means 'less than'; these conditions are explained fully in the next chapter, but it is sufficient to say that the condition is met if a is less than 30. If this is the case, then the LED is turned on.

```
a := 20
...
If a < 30 led.On
```

It is also possible to use `If` with `Else` so that one command is done if the condition is met, and another if it is not. For example:

```
If a < 30 led.On Else led.Off
```

Finally, there is an optional keyword `Then`, which may be used to visually separate the *condition* from the *statement* in an `If` construction.

```
If a < 30 Then led.On Else led.Off
```

`Then` is not often used, as Venom does not need it, and *indenting* your code will usually make the structure clear to other programmers.

Indentation

You will notice that all the above examples showed the whole of a looping or decision construct on one line.

This isn't how they are normally written. More usually an *indented* format is used. This helps you and others see the structure of the program more easily:

```
To dummy
  Forever
  [
    Repeat 5
    [
      If a < 30
        Print "Less"
      Else
      [
        Print "More"
        a := a - 1
      ]
    ]
  ]
End
```

Repeating Decisions: While & Do

The **While** construct repeats a command as long as a condition is met. Each time round the **While** loop the condition is re-tested.

```
While not_done    ;the condition
[
  do_something    ;the commands
  do_more       ; …
]
```

The **Do** … **While** construct is similar to **While**. However here the test is done at the end of the loop rather than at the start. This means that the commands inside the loop are always executed at least once.

```
Do
[
  do_something    ;the commands
  do_more       ; …
]
While not_done   ;the condition
```

The keywords **Index** and **Index0** are available in these loops, as with all loops.

Multiple choice: Select Case

The `Select Case` construct allows one of a number of different actions to be taken depending on the value of an integer *selection value*. In this example we assume a variable called 'choice' has been defined.

```
Select Case choice
Case 1
[
  Print "Choice 1"
]
Case 2
[
  Print "The second choice"
]
Case 10,11
[
  Print "A larger number"
]
Case Else
[
  Print "Default action"
]
```

The `Select` construct looks at the integer selection value, and then executes only the code associated with that particular `Case`. More than one case value may be associated with a bit of code (as with `Case 10,11` in the example above), and a default option may be specified with `Case Else`.

Waiting

Often it is useful to wait for certain events. The `Await` command may be used for this – it just waits for a condition to be met before carrying on.

```
Await my_button.Asserted
```

As the `Await` construct is waiting for a condition to be met while not actually running any other code it is usually only used to wait for external events, or for signals from other tasks in a multitasking application.

Finally the `Wait` command just waits for a given number of milliseconds. For example,

```
Wait 1000
```

just pauses execution for 1000 mS.

Breaking out of Loops: Break

Any loop may be exited prematurely using the **Break** command. This simply breaks out of the loop as soon as it is executed, and the code immediately after the loop is then run.

```
Forever
[
  Print Index
  If Index = 10
    Break
]
Print "Broken out!",CR
```

If loops are nested, **Break** will only break out of one level. To break out of more deeply nested loops see Try

SUMMARY

- A set of commands may be grouped into a single command block using the **[** and **]** symbols.
- **If** and **Else** may be used to make decisions and conditionally execute commands.
- **Select Case** … chooses one of many actions depending on a number.
- **Repeat** may be used to repeat commands a predetermined number of times.
- **Forever** repeats commands forever.
- **Every** repeat commands in a timed loop.
- **While** and **Do ... While** repeat commands as long as a condition is true.
- **Break** will break out of any loop.
- **Await** waits for a condition to become true before continuing.
- **Wait** may be used to pause for a number of milliseconds.

Variables and Expressions

A *variable* is a named bit of memory that can hold values that may change during the running of a program. The first time a new variable name is seen by Venom some memory is reserved to hold the value.

The value of a variable may be set using **:=** (spoken as 'becomes equal to').

In the following example, a variable called **counter** is created and set to the value 1.

```
-->counter := 1
-->
```

A variable's value may be changed at any time to any other value.

A variable's value may be examined using the **Print** command, for example:

```
-->Print counter, CR
     1
-->
```

Variable Names

It is good programming practice to use meaningful names for the variables in your programs. This makes it much easier for you to write the program in the first place, and is a great help anyone who tries to read your code later.

Consider the following two procedures, which both do exactly the same job. One is clearly easier to understand than the other.

```
To a (b,c)
  #Define f 3.14159
  Local d
  Local e
  Local g := b / 2
  d := f * (g^2)
  e := b * f * c
  Return d * 2 + e
End

To cylinder_surface_area (diameter , height)
  #Define Pi 3.14159
  Local circular_end
  Local curved_surface
  Local radius := diameter / 2
  circular_end := pi * (radius^2)
  curved_surface := diameter * pi * height
  Return circular_end * 2 + curved_surface
End
```

Note that both of these procedures compile down to exactly the same runtime code.

Variable names may contain up to 64 letters, digits and underscore _ characters. The name may not start with a digit.

Variable names may not be the same as any Venom keyword or object type. Some examples of both valid and invalid names are listed below:

Valid names	Invalid names	

`water_temperature`
`output_control_2`
`a`
	`word`	(WORD is a keyword)

	`var%`	(% is not allowed)
	`low byte`	(spaces are not allowed)

Style tider

Venom is not case-sensitive; you can write your code in any case you like. However we recommend our standard capitalisation style, which we have tried to use throughout this tutorial. VenomIDE can convert your files to our standard capitalisation using the Style Tidier: **Edit ▶ Style Tidier**.

Listing Names

You can list the names of all the variables that Venom has seen by using `List Word`.

```
-->List WORD
Procedures:
startup init main monitor_in illustrate_locals search
Integers:

Floats:

Strings:

Pointers:

Objects (inc. 'Nil'):
system serial net led clock
Undefined:
sense_in counter o ilst any_value
```

Note that some of the objects displayed will have been automatically created by the startup procedure (discussed later) regardless of whether they are required by the user's program. In general, the items are listed in the order they were first seen. The Undefined names are words that have been used in some way, but have not been assigned a value yet. After an application has been downloaded, but before it has been run, many of the names will be in this section.

Integers and Floating-Point Numbers

The numbers we have used so far have been whole numbers (or 'integers'). These numbers are adequate for many purposes, but they do have restrictions - integer values must be in the range **-2,147,483,648** to **2,147,483,647** and must be a whole number (not a fraction).

Venom can also use floating-point numbers (or 'floats'). For example our **`counter`** variable may be set to a floating-point value:

```
-->counter := 1.0
```

Floating-point numbers in Venom are calculated and stored to the IEEE single precision standard: a number range of around **±1.0E±38**, and a precision of around 7 digits.

Constants

Constants are values that stay the same throughout the runtime of a program. Constants have been used extensively in the examples so far: things like **`1`**, **`100`**, **`2.134`** and so on.

Numeric constants

Integer constants are numbers like these:

```
1
23
43435
```

They may have any value within the 32-bit integer range: -2,147,483,648 to 2,147,483,647.

Floating-point constants must start with a digit, but must also include either one decimal point and/or one e or E to indicate a exponent. These are all floating-point values in Venom:

```
1.234
1e2
2E4
2.34e12
```

Logical or Boolean constants

In venom the values **`True`** and **`False`** are keywords that may be used to represent the logical values of *true* and *false*.

`True` is an alias for the numeric value 1 and **`False`** is an alias for the numeric value 0.

```
-->Print True, False, CR
     1     0
-->
```

String constants

There are also string constants – these are bits of text that remain the same during the life of a program, for example:

```
"This is a string constant"
```

String constants always appear within double quotation marks.

(Actually, there is another way to define string constants, usually used when you need to define a large block of text - see *Embedded text* in the *Venom2 Help File*).

Named Constants

Just as it is very useful to give the variables in your program meaningful names, it is also useful to give many of the constants in your program names too.

There are several reasons for this: firstly, it helps with understanding the intention behind your code when you use a name:

```
Wait 60000
Wait ONE_MINUTE
```

Secondly, if the same constant appears throughout a program, and the value needs to be changed, it need only be changed in one place.

Named constants may be created with `#Define`:

```
#Define ONE_MINUTE 60000
#define two_minutes (ONE_MINUTE * 2)
```

Actually, `#Define` may be used to give any piece of program text a name. A named piece of text is called a 'macro'.

```
#Define LEDON led.on
```

Macros are dealt with in greater depth here.

Expressions

An expression is a bit of program code that calculates a result from one or more values using operators.

Examples of values are: `32`, `1.23`, `counter`.

Examples of operators are: `< > + - * /` and so on.

Arithmetic Operators

The standard arithmetic operations Add, Subtract, and Multiply are available. In Venom these are **+**, **-** and *****.

There are two forms of division in Venom: integer (**Div**) and floating point (**/**).

All symbols such as these are referred to as 'operators' since they operate on values. The following are examples:

```
-->Print 3 * 2, CR
     6
-->Print 3 + 2, CR
     5
-->Print 3 + 2 * 4 , CR
    11
-->
```

Note that the final command is calculating 2 times 4, giving 8, and then 8 plus 3. This is due to the 'precedence' of the operators, which determines in what order the operations should take place. Precedence will be discussed in detail in the next section.

If either of the numbers being added, subtracted or multiplied is a float, then the result will also be a float. For example:

```
-->Print 3.2 + 2, CR
    5.2000000
-->Print 6.7 * 5.8, CR
    38.860000
-->
```

This is called *promotion*, and happens automatically.

Division is slightly different – even if both numbers are integers, the result will always be a float.

```
-->Print 5 / 2, CR
     2.5000000
-->
```

If an integer result is required, the **Div** operator may be used – this always gives an integer, and also requires integer values to work with.

```
-->Print 5 Div 2, CR
     2
-->
```

The operator **Mod** calculates the *remainder* after the division of two integers.

```
-->Print 57 Mod 9, CR
     3
-->
```

The 'Unary minus' operator negates the value it is placed before. For example:

```
-->a:=5
-->Print - a , CR
    -5
-->
```

The operator, **Abs**, gives the absolute value of the following number. The absolute value is the magnitude or size of the value, and is always positive:

```
-->Print Abs -23, Abs 23, CR
    23    23
-->
```

Trig, Log and other functions

A useful set of trigonometric and exponential operators is also available:

```
Sin Cos Tan
Asin Acos Atan
Log Exp Sqrt
```

The trig functions operate in radians.

```
-->Print Sin 1.0
     0.841471-->
```

Exp gives 'e to the power of' a number. **Log** gives the natural logarithm of a number (that is Log_e). **Sqrt** gives the square root of a number.

Power operator

There is also a power operator, which raises the first operand to the power of the second: ^

```
Print 2^3
-->     8
```

This operator will give an integer result if both operands are integers. Note that it's precedence is the same as *****, **/** and **Div**.

Precedence

It was shown earlier that 3+2*4 is calculated as 11. This is because the multiplication is calculated first. The order in which operators are calculated is determined by their 'precedence' – the higher the precedence, the earlier they are calculated. When operators have the same precedence they are calculated in left to right order.

A full table of precedence for the operators discussed in this chapter (including those yet to be discussed) is given below. The operators with highest precedence are listed at the top – operators on the same line have equal precedence.

`( )`
`• As Int  As Float IsFalse`
`- Abs Inv ! Sin Cos Tan Asin Acos Atan Sqrt Exp Log ? TypeOf`
`* / Div Mod ^`
`+ -`
`> < >= <= = <>`
`And Or Eor AndAlso OrElse`

To change the order of calculation, parentheses (round brackets like these) may be used. For example:

```
-->Print (3 + 2) * 4, CR
    20
-->
```

This gives 20, since 3+2 is calculated first, giving 5, which is then multiplied by 4.

Adding redundant parentheses will produce exactly the same runtime code, but can sometimes add clarity to your source code.

Type Conversion Operators

Often it is useful to convert floats into integers or vice-versa. This can be done with the operators `As Int` and `As Float`. Their operation is simple – they convert the preceding number into an equivalent number of the type specified. For example:

```
-->Print 3 As Int, CR, 3.9347 As Int, CR
     3
     3
-->Print 3 As Float, CR, 3.9347 As Float, CR
    3.0000000
    3.9347000
-->
```

Note that `As Int` simply discards the fractional part of a number. There is no 'rounding up' to the nearest integer, and negative numbers round towards zero.

Relational Operators

It is often useful to make a comparison between numbers: 'Are these numbers equal?', 'Is this number bigger than that number?', and so on. This is done with relational operators.

Before discussing the operators themselves, it is important to appreciate the significance of their results. All relational operators return either 1 or 0; 1 indicates that the relationship was *true*, and 0 indicates that it was *false*.

The main relational operators are **=**, **<** and **>**. Of these, **=** tests whether two numbers are equal, **<** tests whether the first is less than the second number, and **>** tests whether the first is greater than the second number.

The following example illustrates their use (remember that the relationship is *true* if 1 is returned). Note that several numbers may be printed at once if separated by commas.

```
-->Print 3 = 3, 2 = 1, 1 < 2, 2 < 1 , 2 > 1, 1 > 2, CR
     1     0    1     0    1     0
-->
```

The remaining three relational operators (**<>**, **<=** and **>=**) are variations on the first three; **<>** tests whether two numbers are *not equal*, **<=** tests whether the first number is *less than or equal* to the second, and **>=** tests whether the first number is *greater than or equal* to the second. While the origins of **>=** and **<=** are obvious, **<>** is a slightly odd symbol for 'not equal' or 'different'.

There are some points worth noting about testing for equality (using **=** and **<>**).

Firstly, two numbers of different types (for example, integer and float) will *never* be regarded as equal. This is illustrated in the next example:

```
-->Print 3.0 = 3, 3.0 <> 3, 2.0 = 3, 2.0 <> 3, CR
      0    1     0    1
-->
```

Secondly, it is not usually a good idea to rely on a test for equality between two floats – even though they 'should' be equal, tiny errors that creep in due to the finite precision of the calculation may cause them not to be precisely equal.

Boolean Operators

In the same way that it is possible to perform calculations with numbers, there are also calculations that can be performed with true/false values (collectively termed 'Boolean', or logical, values). The operators that do this are **`AndAlso`**, **`OrElse`** and **`IsFalse`**.

These operators make it possible to describe complex conditions:

```
If (val > threshold AndAlso error_flag IsFalse) OrElse override
  do_something
```

These returned values may be used by themselves, but are more often used as the condition part of an **`If`**, **`While`**. **`Await`** or other similar statement.

The 'truth tables' of the Boolean operators are shown below:

```
False AndAlso False => False
False AndAlso True  => False
True  AndAlso False => False
True  AndAlso True  => True

False OrElse False => False
False OrElse True  => True
True  OrElse False => True
True  OrElse True  => True

False IsFalse => True
True  IsFalse => False
```

In Venom, `True` and `False` are keywords which represent the values `1` and `0` respectively.

Note that the Boolean operators, as well as all Venom constructs that take a condition (e.g. `If`, `While`, `Await`), will treat 0 as meaning `False`, but any non-zero number to mean `True`.

Lazy evaluation

Another property of the `AndAlso` and `OrElse` operators is that they don't actually evaluate the second (right hand) expression if the left hand expression determines the result. For example, if the first value given to `AndAlso` is false, there is no need to look at the second value - the result has to be false. This is called 'lazy' or 'short circuited' evaluation and can be useful for writing more efficient and clearer code.

Note that `IsFalse` is a 'postfix' operator - that is it comes *after* the expression it operates on.

Another look at Index

It was shown before that `Index` and `Index0` could be used in a loop to represent the number of times that the loop had repeated. However, only one value of `Index` is available at any time – if one loop is inside another, the `Index` value for the outer loop is not available. To solve this problem, the value of `Index` can be placed in a normal variable in the outside loop. This is shown in the example below, which prints a multiplication table.

```
Repeat 3
[
  line_number := Index
  Repeat 3
    Print Index * line_number
  Print CR
]
```

with the result being:

```
     1     2     3
     2     4     6
     3     6     9
```

Changing the type of a variable

Venom allows you to change the type of a variable at will. For example you can define the variable counter to be an integer first, and then a float later:

```
-->counter := 1
-->counter := 2.0123
```

It is also possible to change an object into a number, and vice-versa. Though this is sometimes useful it can be the source of some confusion when it is first encountered. For example if an analogue output channel is called 'level', then a short lapse of memory may cause you to type the second line, intending the output level to become 128.

```
-->Make level Analogue ($30)
-->level := 128
```

Instead the object is replaced by an integer. If you later try to send a message to it, then an error will be issued.

Help

The `Help` keyword can tell you what the type of a variable is:

```
-->Help level
It is an Integer
-->Make voltage Analogue ($31)
-->Help voltage
It is an Analogue object. Try PRINTing it for more info.
-->
```

TypeOf Operator

You can find the type of a variable within your program code using the TypeOf operator. This will return an integer that represents the current type of a variable.

```
-->Print TypeOf 12
     0
```

Or more usefully, inside a program you might use this to check if x is a floating point number:

```
If TypeOf x = TypeOf 1.0
[
  ...
]
```

Sets of Data

There are three pre-defined object types in Venom that provide storage for sets of data: `Array`, `Buffer` and `File`.

You can also create your own entirely new object types to hold sets of related data. This is dealt with later in Creating new classes.

Arrays are for holding fixed amounts of data, whereas Buffers can hold variable amounts of data. Arrays and Buffers also have other properties that are given in the table. Files are similar to buffers, but can potentially hold much more data, and will retain their contents when the controller is not powered. Access to files may be slower than buffers.

	Array	Buffer	File
Dynamic size	No	Yes	Yes
Contents initialised	Yes	No	No
Data can be modified	Yes	Yes	Yes
Mixed data types	No	Yes*	No
Text handling	Limited	Yes	No

Buffers, Arrays and Files can hold many kinds of data. All the data within a single object must be of the same type*. The data types that may be stored are:

- 8, 16 or 32-bit Integers
- Floating point numbers
- Arrays of string constants
- Buffers and files containing text
- Arrays of Pointers

**Note that there is also a special kind of Buffer which can hold mixed data of any type, including other objects.*

Array and Buffer will be fully described in the Objects section, but for now here are some examples of using them:

Constant Arrays

In your program file insert text like this and download it:

```
ARRAY some_data (Int , 10)
 1,
 2,
 3,
 4
End
```

On the command line type this:

```
-->Print some_data.Element (2)
    3-->
```

The **`Array`** we created was called **`some_data`**, and we indicated it should hold 32-bit-integer data, and that it would hold 10 of these integers. We then specified what the first four of them were, before **`End`** indicated the end of the definition of the **`Array`**.

We then read out one of these numbers using the **`Element`** message on the **`Array`**.

Note that Venom has a shortcut for **`.Element(n)`**, which is **`.(n)`**. We use this in the example below.

```
-->Print some_data.(2)
    3-->
```

Variable Arrays

The example above created a *constant* array - the numbers held by it would not change while the program was running.

You can create *variable* arrays where you can change the values held in the array, for example:

```
-->Make var_array Array(Int, 10, 1 , 2, 3)
-->var_array.(0) := 12
```

This makes an array of 10 32-bit numbers, and initialises them to 1, 2, 3, 3 ... , then we change the very first element to be 12.

Or you can use this:

```
-->var_array := some_data.Copy
```

This makes a variable array with contents initialised to those of the constant array we saw before.

Buffer

Buffers are different to arrays. Here we make a Buffer that takes 32-bit integers, put a couple of numbers into it, then print one of them:

```
-->Make buff Buffer(Int)
-->buff.Put(1)
-->buff.Put(2)
-->Print buff.(1)
     2-->
```

Files

Files are described fully in the Venom2 Help File.

SUMMARY

- Numbers may be stored in variables using the **`:=`** symbol.
- There are two types of number – integers (whole numbers) and floats, which are capable of holding 'real' numbers, i.e. those with a decimal point.

- The operators `+`, `-` and `*` are used to add, subtract and multiply numbers.
- Division using `/` always gives a floating point result and `Div` always gives an integer.
- The operators `=`, `<`, `>`, `<=`, `>=` and `<>` may be used to make comparisons between numbers giving a Boolean result (1 for true, or 0 for false).
- The operators `As Int` and `As Float` convert the preceding number into an integer or a float respectively.
- The operators `AndAlso`, `OrElse` and `IsFalse` may be used to manipulate logical (or Boolean) values.
- The order in which all of the above operators are calculated is determined by their precedence – high precedence operators are calculated before low ones.
- You can store sets of constant or variable data in Arrays, Buffers and Files.

Printing

The Print command has been introduced already for printing numbers. It actually has much more flexibility.

A Print command consists of the Print keyword followed by a 'print list', which is a list of items separated by commas. Each print item may be an expression, some text, or one of several special printing keywords.

(Note: there is also a method of printing very similar to C's printf() function: the PrintF message.)

Strings

It is often useful to include some text (a 'string' of letters) in the print list. You do this by enclosing it in double quotes. For example:

```
-->Print "The counter is ", counter, CR
The counter is      1
-->
```

Print Keywords

There are several special print keywords that may be included in a print list.

CR, *carriage return*, starts a new line.

BEEP will make the terminal beep. This is useful for attracting attention during debugging, when a terminal is attached, for example:

```
Print BEEP, "An error has occurred", CR
```

Print **CHR** is used for printing particular characters on the terminal screen. It is followed by a value: the ASCII code of the character to be printed. The following example displays the whole alphabet by printing characters 65 to 90. A full list of ASCII character codes is available.

Example

```
-->Repeat 26 Print CHR 65+Index0 Print CR
ABCDEFGHIJKLMNOPQRSTUVWXYZ
-->
```

Printing Integers

The colon operator (:) may be used to alter the way in which integers are printed. It is placed after the expression to be printed and is then followed by an integer value. This combination of colon and value is termed a 'format specifier'. In this case it specifies how many characters should be used to print the expression – the 'field width'. The following example prints the results from a couple of variables (assumed to be integers, and defined elsewhere) called *timeValue* and *random*.

```
-->Repeat 4 Print Index:2, timeValue, random:10, CR
 1 14082  14627625
 2 63173   2363283
 3 50987  47844170
 4 38904  37278678
-->
```

Note that *timeValue* is being printed in the default field width of 6 characters. If a number is too large to print in its allocated width, it will use as many characters as it needs.

If a negative field width is specified, then the number will be printed with zeros before it so that it always fills its width.

```
-->Print 10 : -4 , CR
0010
-->
```

Printing Floats

There may be up to three colon (:) format specifiers following a floating-point expression. The first specifies the total field width and operates as for integers.

If only one colon is used, a general floating point print format finds a sensible way of displaying the value.

```
-->print 1.2, 0.000000012, " ",1200000000.0,cr
          1.2      1.2e-08  1200000047.7
-->
```

The 7-digit precision in floats is showing in the third number.

If there are two colon format specifiers, the number of digits after the decimal point may be specified. The first colon specifies the total field width, and the second specifies the number of decimal places. Again, a number that does not fit will simply use as many characters as are needed.

```
-->Print 12.718281:15:5, CR, 12.718281:8:4, CR, 12.718281:3:3,
CR
       12.71828
 12.7183
12.718
-->
```

If there are three colon format specifiers, the number is printed in 'scientific' format – which is the format that is used normally when a number is either too large or too small to be printed as usual. The first number specifies the total field width as always; the second specifies the number of decimal places; and the third specifies that the 'E' format be used. The third value is not currently used. If the number is too wide, it will use as many characters as required.

```
-->Print 12.718281:15:5:0, CR, 12.718281:8:4:0, CR,
12.718281:3:3:4, CR
    1.27183E+01
1.2718E+01
1.272E+01
-->
```

Printing a Fragment of a String

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

```
-->Repeat 10 Print "[","abcdefghij":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 both to 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    ]
-->
```

SUMMARY

- There are a number of special keywords for printing, such as CR, CHR.
- Numbers may be formatted using the colon (**:**) operator. In its simplest form this sets the field width of the printed number.
- Strings may be partially printed to achieve special effects.

Procedures

Procedures were first introduced in the Getting Started section. This chapter explains more fully what procedures can do and how to create them.

A procedure is a set of commands that are grouped together and given a name. Some procedures will be created to perform low-level (i.e. detailed) aspects of an application, like turning outputs on and off and recording values. Other procedures will cover high-level aspects of the application, calling on lower level procedures to perform the detailed operation.

If meaningful names are given to procedures, then the higher-level procedures tend to read like an English description of the steps involved in solving a problem. For example, the top-level procedure of a Venom application, which controls a furnace's temperature cycle and logs the actual temperature, might look like the following. Don't worry about understanding the detail of the example.

```
To control_the_furnace
  initialise_the_variables
  Start temperature_log
  control_the_temperature
End
```

The lower-level procedures might look like this:

```
To initialise_the_variables
  Make logged_values Buffer
  rate_up := 1.435E-5
  rate_down := 1.435E-5
  minutes_hold := 120
End

To temperature_log
  Every 60 * 1000
  [
    logged_values . Put (temperature)
  ]
End

To control_the_temperature
  AutoDestruct
  Local timer1 := New Stopwatch
  Local timer2 := New Timer(1000 * 60 * minutes_hold)

  timer1 . Reset
  While temperature < final_temp_1
  [
    demand_temperature (timer1 . Time * rate_up)
  ]
  timer2 . Go
  Await timer2 . Done
  timer2.Period := 1000 * 60 * 10
  While temperature > final_temp_2
  [
    demand_temperature (timer2 . Time * rate_down)
  ]
End
```

These lower-level procedures call on other procedures lower than themselves, temperature and demand_temperature which, for compactness, are not listed here.

This division of processing into higher- and lower-level procedures is a large part of what *structured programming* is about. Structured programs tend to be quicker to develop, easier to understand and faster to debug.

Defining Simple Procedures

The **`To`** keyword starts the definition of a procedure and is always followed by the name of the procedure. The body of the procedure then follows as a number of commands. Finally, the procedure definition is finished with the **`End`** keyword.

The following procedure monitors a proximity sensor, **`sense_in`**, and when an object is detected (the sensor goes from False to True), it adds one to the variable counter.

```
To monitor_in
  Forever
  [
    Await sense_in.Asserted = False
    Await sense_in.Asserted = True
    counter := counter + 1
  ]
End
```

The best way to create your own procedures is to write them into your venom code file, then download the file using F7, etc.

We dealt with this earlier.

Procedure Names

Procedure names take exactly the same form as variable names. See Variables and Expressions.

Calling Procedures

A procedure is executed, or 'called', by using its name as if it were a command, for example by putting its name in another procedure:

```
To reset_then_monitor
  counter := 0
  monitor_in
End
```

Another example is given below.

```
To log_temperatures
  Forever
  [
    log_value.Put(measure_temperature)
  ]
End
```

Here we assume `measure_temperature` is a procedure that reads an analogue input and returns a calibrated temperature value; returning values from procedures is explained below. `Log_value` is a Buffer that accumulates data entries. See the second part of this manual for more information on Buffers.

Beware of using procedure calls in the place of the 'GOTO' commands of some early languages. If you try the following example, the code will stop within a few seconds. This is because each time a procedure is called, a small amount of 'stack' memory is taken. This memory is returned to the system when the procedure ends. If a procedure repeatedly calls itself, then small amounts of memory are continually taken but not returned.

```
To flash_led
  led.Toggle
  Wait 100
  flash_led ; Incorrect usage!
End
```

When a procedure calls itself, it is termed 'recursion'. This is sometimes useful; it will be dealt with later.

Comments

Comments may be inserted into your Venom code anywhere using the semi-colon (`;`) character. All text following the semi-colon, and before the next carriage-return, is treated as comment and ignored by the Venom compiler.

Comments in code are used to explain to someone else (and even to yourself) what the code is doing and why. This is necessary at both high and low levels, i.e. for detailed descriptions of what is going on, and also for the broad outlines of the application.

```
;This procedure reads the thermistor input,
;converts it to degrees C and returns it.
To read_temperature
  read_an ; Read the analogue to digital converter
  ...
End
```

Commenting code well is regarded as a very important part of good software engineering practice. Comments are essential for maintaining code, that is, when correcting errors (bugs) in the code, or when adding new functions.

It is widely recognised that after just a few months most people can't remember how their code works, or why they wrote it the way they did.

For particularly hard-to-solve problems it's a good idea to write down the thought processes behind a particular bit of code. Comments are the way to do this.

Procedures are not forgotten

If you are using a controller with a Battery-backed RAM then it won't lose any of the procedures that you have downloaded or typed in, even when it is turned off. Try entering a procedure and then turning the controller off. When you turn it on again, make sure that you answer 'N' to the Clear Memory question. Your procedure will still be in the controller if you type `List All`.

Note, however, that the values of all variables are lost over a power cycle. Your program should initialise these each time it runs.

No battery

If you are using a controller with no battery-backed RAM, then it will need the procedures loading again if the power goes off.

Protected Application Area

Your procedures should only be stored in RAM during software development. When your software development is complete you should 'Protect' your code in the controller's Protected Application Area (in Flash memory), where it is safer from accidental loss.

Backup your source code

You should also make sure that you have your source code (the code in your Venom program files) put somewhere safe - where it is safe from loss and where you can find it when you need it. You can't get it back out of the VM2!

Passing Information to Procedures: Parameters

To enable a procedure to accept information, certain variables – called parameters – can be set up to receive the information when the procedure is called. The parameter names are given after the procedure name. They must be enclosed in parentheses `()` and separated by commas or spaces. The values of the parameters will be set to the values given to the procedure when it is called. For example, the following procedure prints the results of Div and Mod on the two numbers given. The parameters are named **a** and **b**.

```
To div_and_mod(a,b)
  Print a Div b, a Mod b, CR
End
```

This procedure is called as before except that the values of the parameters must be given (again separated by commas or spaces and enclosed in brackets):

```
-->div_and_mod(10,3)
     3     1
-->
```

When called as above, the parameters a and b in the procedure will be set to 10 and 3 respectively.

Formally, parameters in Venom are *passed by value*, rather than *passed by reference*. If you want to pass by reference, you can use pointers. See Pointer Expressions.

Procedures that Return Information

Procedures that return information (sometimes called *functions*) use the Return command. Return is always followed by an expression. When Return is encountered, the expression is calculated and the resulting value is immediately passed back to the code that called the procedure.

For example the following procedure returns a calibrated temperature:

```
To measure_temperature
  Return (thermometer.Value - 12) / 3
End
```

Note that the expression will produce a float, and so a float is returned. The procedure's returned value may be displayed (and formatted) using the Print command:

```
-->Print measure_temperature:8:2, CR
   22.67
-->
```

Often procedures will both accept and return information. The following procedure takes two numbers as parameters and returns the larger of the two:

```
To greater(x,y)
  If x > y
    Return x
  Else
    Return y
End
```

It is called in exactly the same way as other procedures with parameters:

```
-->Print greater(1,2), greater(319,122), greater(117,980), CR
     2   319   980
-->
```

Exiting Procedures

A procedure is normally exited when the last statement has been executed. Control then returns to whatever called the procedure. However a procedure may be left before End by using Return. Return will exit a procedure immediately, as well as perform its other function of returning a value. If you don't need a value to be returned, just use Return 0.

```
To proc
  ...
  If x = 12
    Return 0
  ...
End
```

Local Variables

So far, apart from parameters, all the variables we have been dealing with have been 'global' – that is they are 'visible' from any procedure or the command line. Often it is desirable to have a set of variables that are only visible from one procedure. These are called *local* variables.

Local variables are set up using the Local command. The following procedure is a functionally identical version of `measure_temperature` shown before, but it uses a local variable called `int_temp` (short for 'integer temperature'):

```
To measure_temperature
  Local int_temp
  int_temp := thermometer.Value
  Return (int_temp - 12) / 3
End
```

Nothing except the procedure **measure_temperature** may use or even 'see' the local variable **int_temp**. If there is a global variable with the same name, it will be ignored inside the **measure_temperature** procedure. This feature is called *overriding*. Local names automatically *override* global names.

Local variables are very important to good programming. They allow programmers to be confident about which bits of code are accessing which data.

The Lifetime of Local Variables

Local variables are created anew every time a procedure is called, and are lost forever when the procedure finishes. They may be initialised to any value, either when they are created, or later.

The following procedure illustrates a variety of ways to declare and initialise local variables:

```
To illustrate_locals
  Local a
  Local b
  Local c,d,e,f
  Local g := 12, h := 13 * g + any_value
  a := 11
  b := 10
End
```

If you don't initialise a local variable when it is first defined, it is given the default value of integer zero.

Local variables must always be defined at the start of a procedure, before any other lines of code.

Recursion

Procedures may be called *recursively*. That is, a procedure may call itself, directly or indirectly. This technique is not often used in control applications, but is included here for completeness.

Recursion sometimes gives an elegant solution to some problems but you will probably never need to use it in Venom.

Trivial recursion

A rather trivial example of recursion is:

```
To recursive_procedure
  recursive_procedure
End
```

Local variables are on the stack

Whenever a procedure is called, it allocates some stack memory in which to store its return address and the values of its local variables (among other things). When a procedure is called recursively (or if it is called by several different tasks), each procedure call is termed an 'instance' of that procedure.

Even if there are many instances of a procedure, they will each have their own set of values for the local variables. If they don't affect any global variables or any external device then each instance of the procedure is entirely independent.

Example of using recursion

For example, the procedure below will find a given value in any Buffer object that contains a set of unique values in ascending order. It does this by considering the whole Buffer and then determining whether the value is in the upper or lower half. It then calls itself, this time giving half the range. It continues until a value is found, in which case all the procedures finish one after the other.

Note that when there are only two or less values to be searched, the procedure no longer needs to call itself. Recursive procedures must always have a way out like this, otherwise they will continue calling themselves until the stack is used up.

```
To search(buf,val,lo,hi)
  Local middle,result
  Print "Searching from ", lo, " to ", hi, CR
  middle := (lo + hi) Div 2
  If (hi - lo) < 2
  [
    If buf.Element(lo) = val
      Return lo
    If buf.Element(hi) = val
      Return hi
    Return 0
  ]
  If buf.Element(middle) < val
    result := search(buf,val,middle,hi)
  Else
    result := search(buf,val,lo,middle)
  Print "Returning from ", lo, " to ", hi, " with ", result, CR
  Return result
End
```

Here we call the search procedure on a buffer we've called *sorted_buffer*: Two Print commands show the recursive procedure calls working:

```
-->search(sorted_buffer,1225,0,36)
Searching from      0 to     36
Searching from     18 to     36
Searching from     27 to     36
Searching from     31 to     36
Searching from     33 to     36
Searching from     33 to     34
Returning from     33 to     36 with     34
Returning from     31 to     36 with     34
Returning from     27 to     36 with     34
Returning from     18 to     36 with     34
Returning from      0 to     36 with     34
-->
```

Listing Procedures

You cannot list the full source of a procedure back as the compiler has translated it into a completely different form. If you attempt to list a procedure it will give you a short summary of the compiled code:

```
-->list search
;To search(buf,val,lo,hi)
;  Local middle,result
;  No source list [248 bytes @$64000a18]
;End
-->
```

The exception to this is the startup procedure. This lists back its default text only – to allow you to see how it operates or to copy it so you can change it.

The master copy of your code should be in the files you create and download.

Deleting Procedures

Normally you won't have to delete any procedures - VenomIDE is set up to do this for you automatically - though you can change this setting it is normally best left as it is.

Delete

Rarely you might need to use the Delete command to delete an individual procedure. For example:

```
-->Delete monitor_in
-->
```

Predefined Procedures

Venom predefines three procedures when its memory is cleared. The three procedures are called `startup` and `init`. They make it easy for you to start programming your application. They are explained in detail later.

SUMMARY

- Procedures are defined with `To` and `End`.
- Procedures are retained while the power is off.
- Information may be sent to a procedure in parameters.
- A result may be returned from the procedure with `Return`.
- A procedure exits immediately on a `Return` command.
- Procedures may set up local variables with `Local`. These variables are private to each procedure and cannot be accessed from elsewhere.
- Delete may be used to delete a procedure from memory.

Objects

So far this manual has used objects without really explaining what they are. By seeing them used in context you will have picked up much of the basic information about them. This chapter will give a fuller definition of objects.

Object orientation allows 'bits of code' (programs) to be treated a bit like off-the-shelf electronic components.

In Venom, objects are used to represent actual devices in the real world, such as a heater connected to a digital I/O pin, or a thermometer connected to an analogue input pin. Objects do the job of *device drivers* in other languages.

In order to make things happen, like reading an analogue value or turning on a digital output, the object is sent messages.

In previous chapters we saw messages sent to objects in the commands:

```
led.on
```

and

```
Print thermometer.Value
```

What we did not cover was how to define these objects in the first place, or the range of messages that you could send them.

Creating Objects

Make

Objects may be created with the Make command.

The example below shows a Digital object being created to control a heater using one of your controller's digital channels.

```
Make heater Digital($2F, 1)
```

The Make command is followed by a variable name, in this case `heater`, and then by the type of object required, in this case `Digital`. Two parameters are also supplied:

1. '$2F' is the VM2 channel number that the Digital object will control.
2. '1' means make it an output rather than an input - some digital channels need to be told this when they are created.

Important Conceptual Note: *heater* is a variable that now *refers to,* the Digital object.

This concept will become important if you start to do more complex things with objects.

Other examples

This example shows the creation of an object to read the temperature sensor used in previous examples:

```
Make thermometer Analogue($30)
```

In this case `thermometer` refers to an Analogue object that reads channel $30.

It is advisable to use descriptive names for objects, even at the expense of more typing*, since the meaning of short names is easily forgotten (and may be a complete mystery to someone else). For example, some other objects could be created as follows:

```
Make display AlphaLCD (20,2,0)
Make buzzer Digital (129)
```

Part 2 of this manual contains detailed information about the various types of object and how they function.

**Note: VenomIDE2 has an autocompletion function that will complete a partially typed variable name from a list of names you have used before.*

When to create Objects

It is possible to create objects from the command line. This is fine when exploring the language, or when trying out something new. However, when it comes to writing a real application, most objects are normally created immediately after the controller is turned on, by convention in a procedure called `init`

There are several reasons for this:

- Objects are not retained over reset or power down, and so must be re-created before a program may use them
- It is good if most of the object definitions are in the same place in the program listing, for easy maintenance of the code
- All the memory that is going to be taken by objects will be taken early on, which makes programs easier to debug
- Objects should not be defined more than once, and this is easy to ensure if all the definitions are in one place

There are sometimes circumstances where you will want to create objects 'dynamically'. This requires special care and is covered in Creating Temporary Objects.

The Startup Procedure

There is always a procedure called *startup* in Venom. A default startup procedure is created by Venom. Startup tells the controller what to do at power-on.

The **`led`** object used in the examples in previous chapters is defined in the default startup procedure, along with several other useful objects. If you need to see the code of the default **`startup`** procedure, then use **`List`**:

```
-->List startup
 The text of the DEFAULT startup procedure:
 TO startup
   MAKE system OperatingSystem
   system.ErrorAction := NEW Digital($20).Asserted IsFalse
   MAKE serial SerialPort(115200,1,1)
   MAKE net I2Cbus
   MAKE led OnBoardLED
   MAKE clock RealTimeClock
   IF system.Runmode
   [ led.Flash($80)
     init
     main
     led.Flash(0)
   ]
 END
 -->
```

Notice **`startup`** calls two procedures, **`init`** and **`main`**.

You have to write init and main yourself. You can re-define startup, but it's usually better to leave it as it is.

The init procedure is the best place to put all your Make commands, as then they will be reliably executed at startup, when power is applied to the controller. The main procedure is the best place to put your main application code.

The startup procedure has a chapter all to itself, later on in this tutorial.

Using Objects

It has been shown that objects may be sent messages by placing a dot (**`.`**) after the object name, and before the message name. For example, the following two commands send 'On' and 'Off' to the Digital object named heater.

```
-->heater.On
-->heater.Off
-->
```

Similarly the command below can be used to read a thermometer connected to an analogue input object.

```
-->thermometer.Value
-->
```

However, nothing appears to happen because the 'result' was not used. To examine the result, the Print command could be used. For example, to print the value of thermometer, type the following:

```
-->Print thermometer.Value, CR
    46
-->
```

Thermometer.Value is simply an expression, the value of which may be assigned to variables, or used in further expressions, for example:

```
a := thermometer.Value
b := thermometer.Value * 10 / 2.546
```

Message Parameters

Some messages take parameters, just like procedures. An example of this is the Flash message to the led object. Try the following line:

```
-->led.Flash($A0)
-->
```

The message `Flash($A0)` sets the LED on your controller flashing around twice per second. (Many other flash patterns are possible - see here for more details)

Active Variables

Some messages are called 'active variables'. They may be both set and read, just like a normal variable. An example of this is the message `Asserted`, understood by `Digital` objects, among others. If `heater` is a digital object, then setting `heater.Asserted` to `True` or `False` will turn the heater on or off respectively:

```
-->heater.Asserted := True
-->
```

Reading `heater.Asserted` will return True or False depending on whether the heater is currently on or off:

```
-->Print heater.Asserted
    1-->
```

(Remember `True` has the value 1)

What Objects are available?

In Venom, there is a set of built-in object types. The exact definition of each object type, and of the messages it responds to, is given in the Venom2 Help File.

A less formal description of how to use a selection of these built-in objects is given in Part 2 of this tutorial.

User-defined Classes

Venom also allows you to define your own types of objects - see Creating new classes.

The I2C Bus

One particular object will be keep getting a mention, so it's worth introducing it at this point. This is the `I2CBus` object. The default startup procedure defines one for you called `net`. The I^{2}C bus is an industry-standard bus for communication at the 'chip level'. Literally it is the *Inter-Integrated-Circuit Bus*. The I^{2}C bus allows Venom controllers to connect to a variety of ICs that provide useful functions additional to those on the main controller, including:

- Digital I/O
- Analogue I/O
- LCDs
- Touchscreens
- Keypads

Removing Objects

Any object created with Make may be removed, if it is no longer required, by sending it a Die message:

```
Heater.Die
```

In general most applications will not need to use Die as there is rarely a need to remove objects you have defined.

However, Die may be useful during development, or in applications that use objects in a dynamic way, creating them when they are needed, and destroying them when they are no longer of required.

Trouble shooting

It is easy in Venom to change the type of a variable. This can sometimes cause confusion if you accidentally change an object into a number. See Changing the type of a variable for more information.

```
Make a Analogue ($30)
...
a := 129
```

Here we changed **a** from referring to an Analogue object into the integer value 129.

What was probably meant was this:

```
Make a Analogue ($30)
...
a.Value := 129
```

SUMMARY

- Objects may be created with the Make command.
- It is good to refer to objects with meaningful names.
- Messages are sent to objects by placing a dot after the object name and before the message name.
- Objects may be removed by sending them a Die message - but this is not usually needed.

The Startup Procedure

The startup procedure is one of the most important parts of a Venom application program. It determines what the Venom application does when it is first powered on.

The default startup procedure

There is a default startup procedure that Venom creates whenever memory is cleared. This procedure makes various objects that it is useful to have predefined, including the serial connection to allow programming. The default startup procedure is listed below. You can create a listing of it by typing `List startup` at the command line.

```
The text of the DEFAULT startup procedure:
TO startup
  MAKE system OperatingSystem
  system.ErrorAction := NEW Digital($20).Asserted IsFalse
  MAKE serial SerialPort(115200,1,1)
  MAKE net I2Cbus
  MAKE led OnBoardLED
  MAKE clock RealTimeClock
  IF system.Runmode
  [ led.Flash($80)
    init
    main
    led.Flash(0)
  ]
END
```

Notice that near the end of the startup procedure two procedures (called `init` and `main`) are called. Your application will need to define these.

`Init` is intended for all your `Make` statements and other initialisation code.

`Main` is the code that runs your application.

Try creating your own 'main' procedure in your venom code file and download it (you can hit F7):

```
To main
  Print "Hello world",CR
End
```

Now type run at the command line (or use F10)

```
-->Run
Hello world
-->
```

Run tells the controller to behave as if it had just been powered on with the Program/Run Switch in Run mode. Run is a simple way of testing the startup behaviour of your application.

You can create your own the init procedure in a similar way (don't forget to download it).

```
To init
  Print "Init is before main",CR
End
```

Now run does this:

```
-->Run
Init is before main
Hello world
-->
```

Notice, in the default startup, that init and main are only run if controller is in Run mode. In Program mode, only the basic objects are made.

The default startup procedure may be altered but it is usually best left the way it is.

Putting startup code in your code file

You might want to list out and then copy the default startup procedure to your code file so you (and VenomIDE) can see it. *Don't copy the text above as it may be out of date.*

Program mode

Up until this point in the tutorial you have been using Venom solely in Program mode (Program mode is used for developing applications).

Whenever you power up your controller in Program mode, you will see the Clear Memory message:

```
VM2 Control Computer running Venom2 at 72MHz
Version 2011 02 10
Copyright 2008-2011 Micro-Robotics Ltd.
Clear RAM?
```

You can reply to the question in two ways:

- **Y** (Yes) means go ahead and clear the memory. Everything in the controller's RAM will be removed, and you will have a 'clean' controller to start application development. Venom defines the default procedures `startup` and `init`, and then calls `startup`.
- **N** (No) means leave all the procedures you have defined in the controller's RAM, ready for you to continue developing. The startup procedure is run, though it may not do much, as the controller is in Program mode.

Run mode

Run mode is used when you have finished developing your code and want to run it for real in its intended environment. Run mode is entered when Venom powers up with the *Program/Run* switch in the *Run* position.

Copy to Flash

If your application code hasn't yet been copied to Flash (using the command **`Protect(1)`**) then Venom will ask at the terminal if you want to do this.

If you don't allow that then your application will not run. This is a safety feature, as too many Venom programs had been released into the field with the application code held only in battery backed SRAM, which later stopped working when they lost their program.

Run mode startup sequence

In Run mode, the startup procedure is called which then calls your init and main to run your application.

Run mode emulation

As mentioned before, Run mode may be *emulated* from Program mode by typing **`Run`** at the command line, or hitting **F10** from VenomIDE.

Don't let your application end

You may have noticed that the default startup procedure, and the altered init and main above, terminated by showing the command line prompt (`-->`).

The prompt is always issued when the main task runs out of things to do.

Most real applications, however, require the controller to carry on doing its job forever (or at least until power is removed). Thus your application code should normally enter some sort of infinite loop and never terminate.

If you do see a command line prompt when you run your final application, then it is likely that you haven't written your code correctly.

Example Init and Main

Here the procedures init and main have been redefined from their defaults to run a trivial application that rattles a relay 10 times a second. Startup has been left unaltered.

```
To init
 Make relay Digital(128)
End

To main
  Every 100
  [
    relay . Toggle
  ]
End
```

More on the LED

The LED on the controller may be used to indicate information to you, the developer, and later, to the end-user or person maintaining the equipment.

The behaviour of the LED may be altered by your application program, but it also has some default behaviour.

Program mode

In Program mode, the LED will be on continuously while at the Clear Memory prompt. Thus if the LED is seen to be on, it is likely that a) the controller has power, and b) it has been left in Program mode.

After you have responded to the Clear Memory prompt, the LED is turned off, though if you alter the startup procedure (not recommended) it may then do anything else.

Run mode

In Run mode, the LED is programmed (by the default startup) to flash approximately once every second. Your application can later alter its behaviour to be anything you like, to indicate problems or other information.

You are recommended to use the *'LED on continuously'* signal to indicate a controller that is stuck in Program mode and not use it for other purposes.

SUMMARY

- The procedure startup determines Venom's actions at power-on.
- Venom has three default procedures – startup, init and main –which are created when memory is cleared.
- Init and main should be redefined to suit your application. Startup is usually best left as it is.

- The Program mode switch determines whether Venom should run your application, or issue the Clear Memory message and command line prompt.
- The LED may be used to indicate the operational state of the controller.

Your Development Environment

Right at the start if this tutorial you may have seen how to create a Venom code file and download it into the VM2.

If you don't remember this then you can revisit it.

Running your application

You can run your application by typing **Run** at the command line.

```
-->Run
```

Alternatively you can click on the Run icon (▶), or hit the function key **F10** (both of these simply send the command **Run** to the command line).

Run at power on

When you application is finished and deployed in the field you will want the controller to run your program immediately, every time the power comes on.

To make it do this, you will first need to copy your application from RAM into the more secure Flash memory. This is usually done using the Protect command.

Then you will need to set **Run Mode** - by switching off the Program Mode switch. This switch usually located on the Application Board that the controller is plugged into.

If you put the controller into Run Mode with your application still in RAM the controller will refuse to run the application immediately, but will also give you an option at the terminal to copy the application to Flash.

But for now, while you are developing your code, keep your application in RAM and keep the program mode switch in the **Prog Mode** position. This makes developing your code much faster and more efficient.

Developing your code

You can now start adding new code to your file. Whenever you want to test what you have written, just download it (**F7**), and run it (**F10**).

Large projects

When your code gets too big for a single file (when it becomes hard to navigate around) then you can use VenomIDE's Project Manager to manage your project as a set of code files.

This has the added advantage that during code development you can selectively download only the code files that have changed (**F5** in VenomIDE), speeding up the edit-run-debug cycle.

See *VenomIDE Help* for how to create a new project.

Tips and Tricks

Here are a few useful shortcuts that we find useful when programming in Venom using VenomIDE.

Repeat or Edit commands

Sometimes you want to exercise or test a particular bit of code, and you don't want to have to type it in every time at the command line.

Venom remembers the last 20 commands that you sent it (until you clear its memory). You can recall these commands and edit them using the cursor arrow keys, **Home**, **End**, **Backspace** and **Delete**.

Hit **Enter** to send a command that has been recalled or edited.

Hit **Ctrl-C** to abort the command line.

Send any text to the terminal

If you select (i.e. highlight) any text in the VenomIDE editor you can send it to the terminal by hitting **F4**. Or, you can send a whole line to the terminal by moving the cursor to the line and then hitting **F4**.

Send a commented line to the terminal

It is often useful to list 'test scripts' in your program file - that is command lines that test sections of your code.

You can comment these lines out so that they aren't executed when the code downloads, but VenomIDE allows you to execute them easily: if you put the cursor on the commented line, and hit **F4**, then the line (minus the `;` comment character) is sent to the terminal.

```
To do_it(n)
...
End
;do_it(10) ;Test 'do_it': put cursor on this line and hit F4.
```

Note that currently the `;` *must be the first character on the line for this to work.*

Find the definition of a symbol

If you need to see a definition of a symbol in Venom, right click on the symbol and choose from the many options for finding information about that symbol.

If you place the cursor in a symbol and hit **F12** (or **Ctrl+?**) then the editor will navigate to the symbol's definition.

Find out what your program is doing

If you type **Ctrl-T** at the terminal at any time then Venom will send text to the terminal indicating exactly which part of your code is currently executing.

If there is more than one task running, then the code positions of all the tasks are listed (tasks will be discussed later).

Note that, by default, **Ctrl-T** is switched on only in Program Mode.

Multitasking

Until now, all Venom commands have been executed in sequence: one command has to end before the next can start. However, Venom is capable of executing several sequences at once. This is known as *multitasking*. It allows a single controller to handle several independent processes at the same time.

Multitasking can be hard to understand fully, so we will present a simple model for you to follow first. This model may well be enough to cover your needs, but if you need to use more complicated constructions, then these are presented later.

When to use Multitasking

Multitasking becomes necessary when the application requires two or more processes to be performed independently of each other. That is, when it is important not to hold up (or *block*) one process simply because the application program is still involved with another process.

Consider the common example of a controller that is controlling a machine that also has a user interface.

Without multitasking

Lets say the machine is an oven controller with a temperature sensor that needs polling every 100 milliseconds. The sensor is used to control the oven temperature by turning off the heating element if the temperature is higher than a set target. You could use code like this:

```
Every 100
[
  If temp_in . Value > target_temp
    oven . Off
  Else
    oven . On
]
```

The machine also has a user interface to allow the operator to set the target temperature. Generally, it is hard for the user interface code to also control the oven 10 times per second: you have to scan for keys at the keypad in a loop, and make sense of them, all while making sure that the program always called the oven control code at the correct times. It is possible, but the finished code is usually quite inflexible.

Here's one example of how it might be written:

```
Every 25 ;Scan rate for keypad.
[
  kpd . Update
  Select Case key_input . Key ; read a key
  Case 0
  [
    target_temp := target_temp+1
    Print To lcd, target_temp
  ]
  Case 1
  [
    target_temp := target_temp-1
    Print To lcd, target_temp
  ]

  If (Index0 And 3) = 0 ;Every 4th time round the loop…
  [
    ;control the oven while we are getting keys.
    If temp_in . Value > target_temp
      oven . Off
    Else
      oven . On  ]
]
```

You would have to make sure that none of the user interface code would ever wait for more than 100mS.

You would have to make sure that every user interface routine you wrote included the oven control code, for example if you extend the user interface to a set of menus. For very simple applications this approach is workable, but it 'blows up' when the control system and user interface get more complex.

With multitasking

If we use multitasking to solve this problem, we would create a task to control the oven, and a separate task to control the user interface.

The oven control task could have this code in it

```
Every 100
[
  If temp_in . Value > target_temp
    oven . Off
  Else
    oven . On
]
```

The user interface task could use code like this:

```
Every 30
[
  kpd . Update
  Select Case key_input . Key ;read a key
  Case 0
  [
    target_temp := target_temp+1
    Print To lcd, target_temp
  ]
  Case 1
  [
     target_temp := target_temp-1
     Print To lcd, target_temp
  ]
]
```

These two sets of code could be run simultaneously.

Now we can have the user interface looping every 30 mS, because it might suit the interface to run at that speed. We still have the oven control loop running at 100 mS.

It doesn't matter to the oven control task if the user interface task stops completely, nor vice versa.

The two tasks are now independent, and the code development for each of them may be considered separately, except where they explicitly interact.

How many Tasks can I use?

The Venom language allows you to use lots of tasks. That is, you may use more than it would ever be sensible to use!

Two useful rules of thumb are:

- If the solution to a problem can be solved elegantly without adding another task, then don't add a task
- If you find yourself using more than four tasks to solve a problem, then take a fresh look at your approach before proceeding

Starting Tasks

When a Venom application starts, there is just one task running – the one that executes the **`startup`** procedure. This is called the main *task* or the *command-line task*. Some simple applications may never need another task.

A new task may be started with the **`Start`** command.

This command takes a block of code, and runs that code in a new task. Often the code is just a single procedure, though any block could be used:

```
Start control_task   ;start a procedure as a task.

Start [Repeat 20 Print Index,CR] ;start a code block as a task.
```

Keep Tasks Simple

In general, your application programs will be easier to understand if your task's code blocks are each just a single procedure.

Starting all your tasks shortly after startup (in main, say) and keeping them running forever will make your code much easier to debug and maintain:

```
To main
  Start control_task_1
  Start control_task_2
  ; This last function is in the command-line task
  ;  so we don't need to start a new task.
  user_interface_task
End
```

The Prompt

You may notice that the prompt changes when there are other tasks running. This is just to let you know that there are tasks running in the background. Whenever more than one task is running, the prompt becomes a double arrow:

```
-->Start Forever []
==>
```

When you see a prompt like this you can carry on typing commands, but remember: there are other tasks running in the background still carrying out their instructions.

Stopping Tasks

Any task will stop naturally if it runs out of code to execute.

For example:

```
-->Start [Repeat 5 Print Index , CR]
==>     1
     2
     3
     4
     5  [User presses Enter here, to show the prompt]
-->
```

When the five numbers have been printed, the task runs out of code, and quietly disappears.

Notice that the **==>** prompt was displayed before the numbers printed by the new task. This is because the prompt was displayed before the new task had a chance to print the first number.

The third prompt, sent in response to the user typing Enter again, shows the task has gone.

In order to stop a task before it finishes its work, the Stop command may be used.

Stop

Stop needs to know which task you wish to stop. The Start command returns a 'task object', which may be used for this purpose, or you may use the task's ID number (see Listing Tasks later). The following example illustrates starting a procedure as a task and later stopping it.

```
-->mon_task := Start monitor_in
==>Stop mon_task
-->
```

The command Stop All will stop all tasks except the main (command-line) task.

The main task can’t be stopped using a Stop command. Ctrl-C will stop the main task.

Additionally, typing CTRL-C at an empty command line will stop all active tasks:

```
-->Start Forever[]
==>
[User typed CTRL-C here to stop the task:]
==>STOP ALL
-->
```

So, typing CTRL-C once will stop the main task, and typing it again will stop all the other tasks.

Try not to let tasks end

In general we recommend that you try to write your programs so that you start all the tasks you need near the beginning of your program, and that they all run forever.

If a task does need to end then it will probably have to clean up after itself, which can be difficult to think about.

Listing Tasks

The List Task command produces a list of all the tasks Venom is running, including details of where each task is.

For example:

```
==>List Task


Task ID: 0
waiting at the prompt.
___________________
Task ID: 1
in proc1 (working.vnm line 6)
in proc2 (working.vnm line 10)
in a task started from main (working.vnm line 14).
___________________
```

Ctrl-T

Perhaps even more usefully, if you type **Ctrl-T** at any time, then List Task is called, so you can find out what your application is doing at any time even if you don't have a command line.

The list of tasks will also tell you if any of your tasks are 'blocked' - i.e. waiting for another task to release a resource before it can continue.

Our Simple Multitasking Model

Even apparently simple multitasking systems can sometimes harbour complex problems if they are not written well.

If you follow the rules in our simple model, then you will be able to use multitasking in Venom without having to consider any of the more complicated things that can go wrong.

Only one task owns a resource

This means that major resources like the LCD, the Keypad, and the set of Digital I/O and so on, should each only be accessed by a single task. You should design your code around this idea.

For example, you might have one task that only controls the Digital I/O, and another task which only accesses the LCD and Keypad to make a user interface.

Tasks communicate via signals

If your tasks need to communicate with each other, you can use global variables to signal from one task to others. In previous examples, the variable target_temp was used as a signal from the user interface task to the control task. That is, the user interface task writes to target_temp, and the control task reads it. In this simple model, you should have only one task writing to a particular signal, though many may read it.

Don't call a procedure from more than one task

It is quite possible, and sometimes useful, for two or more tasks to be running the same procedure at the same time. However it is likely to break one of the preceding rules, so to be safe don't do it.

Don't use locking

Locking is a feature of some objects. It is used in more complicated multitasking systems where it is very useful. However, if you have obeyed the first rule, that only one task uses a particular resource, then you won't need to use locking.

Consider task latency

For every extra task you have running in Venom, your code may miss up to 2mS of run time. Thus if any part of your code needs to catch events shorter than 10mS, then you can't have more than 5 tasks active.

Start tasks at the beginning

All your tasks should be started soon after the initialisation phase of your application, preferably in a procedure called main.

Don't stop any tasks

If you stop a task (or allow it to end), the chances are you'll need to start another one again, which breaks the rule above.

That completes the rules for a pain-free multitasking application in Venom. It is possible to write much more powerful and sophisticated multi-tasking systems. This is covered later.

SUMMARY

- If you use our Simple Multitasking Model you should be able to create a robust multitasking application very easily.

Developing an Application

You have now been introduced to all the major parts of the Venom language except for the details of the 'object types'. We recommend that you now look through the second part of this manual (Part 2: Object Tutorial) and get familiar with objects by using them.

Then you will be ready to start writing your own application for VM2. There is a checklist for how to plan and complete your application in Appendix A: Development Checklist.

Advanced topics

This section of the manual continues with more advanced topics. You may not need to learn about these – glance over them and read any sections that are appropriate to your application.

Debugging

Debugging is the process every application programmer has to go through to remove bugs, or mistakes, in the program code.

The next pages present some of the tools available to help you find bugs.

Print

The simplest form of debugging available in Venom is Print. If there is a problem with the program you are working on, insert a line to print out the values of important variables, or use Print to show the order of execution of different parts of the program, or to find out exactly where an error is occurring.

Print output normally goes to your terminal. In some applications the main serial port is being used by the application. In this case the output may be redirected to another device.

Print output may be redirected using the **`To`** keyword or by using the **`Output`** message of the **`OperatingSystem`** object (called **system**). See Text Handlers and Redirection.

Useful places to redirect debug output are:

- To another of the controller's serial ports
- To an LCD display
- To a TextBuffer or file object, where it may be stored for later examination

Print is often all that is needed to find most bugs.

Commenting out

Another effective tool for finding bugs is 'commenting out' lines of code.

This means putting the comment character at the start of the line of code, so that it's not actually run. This way you can selectively remove parts of your program to isolate the bits that are going wrong.

```
To proc
  do_something
;  try_something_difficult
  do_other_thing
End
```

If you want to quickly comment out a whole block of code you can select the block by dragging the mouse over it, then hit **Ctrl**+; - i.e. hold the control key down while hitting the ; semi-colon key.

To un-comment a whole block use **Shift**+**Ctrl**+**;**

Finding errors in your source

Error listings in Venom2 refer to the file and line number the error was estimated to have occurred in. It is very easy to double click on error reports in the terminal and have the editor display the correct file and line of your program.

The lines that are active for error navigation are those that contain a file name and a line number in parentheses; e.g:

... `(main.vnm line 100)`....

Sometimes a range of lines is indicated - this is where Venom has not been able to pinpoint the source of a runtime error to a single line. When you double click on a range of lines then you are taken to the first line in the range.

Here's an example of a runtime error, seen when running an application:

```
Runtime error 5: Un-initialised variable: 'new_var'
  in read_port (mycode.vnm lines 21-24)
  in process_input (mycode.vnm line 30)
  in main (mycode.vnm line 39)
  in the command line.
```

This report shows where the original error occurred (but only gives a range of lines), and also lists the 'call history' that led to the error: which procedures were being called when the error happened. Double clicking on any of the lines with a file name and line number will take you to the correct point in your code file.

Narrowing down the error line

The runtime error reporter tracks down errors by looking for embedded line number information in the compiled code. A bit of code may contain lots of embedded line numbers, leading to an error being tracked down to a single line. However, the code may contain few embedded line numbers, leading to an error being tracked to within a range of lines.

You can improve the error tracking by temporarily embedding more line numbers in your using NOP statements. NOP stands for No Operation - i.e. it does nothing, but it does embed a line number in the code.

You can sprinkle NOP statements around your code to pin point the error.

No file name

If the error report doesn't contain a file name, but instead just lists a line number, then the line number refers to lines within the procedure. In this case error navigation won't work. This lack of file name usually occurs if the procedure wasn't downloaded with the IDE's download commands, but was instead sent as text into the terminal - maybe using **Edit ▸Paste** or **Download selection**.

Listing tasks

You can find out what each task in the system is doing at any time typing **Ctrl-T** (so long as **Ctrl-C** and **T** are enabled - which is the default setting).

Even if you only have one task in your system, **Ctrl-T** is a useful way of finding out what it is doing.

Read more

Help

The **Help** keyword will tell you what kind of thing the variable refers to. Many bugs are due to a variable referring to the something different to what you expected.

```
-->Help led
It is the OnBoardLED. Try PRINTing it for more info.
-->
```

Print may also be used to find out about the contents of variables. Printing an object will often tell you useful information about it.

SUMMARY

- Use Print statements to printout where your program is or the value of critical variables
- Comment out sections of code to find those that are causing a problem
- Double clicking on runtime error reports takes you to the source line in the editor
- Ctrl-T will list all the active tasks to the terminal allowing to you see what your program is doing at any time

Errors and Exceptions

This chapter discusses

- Runtime errors
- Handling runtime errors in controlled way
- Using exceptions to handle difficult coding problems more easily

Runtime Errors

Venom issues a runtime error whenever it fails to execute a command for any reason. When a runtime error occurs, the task in which the error occurred is stopped. An error report is sent to the designated error output device: the terminal, via the main serial port, by default.

Error reports

Runtime error reports will generally look like the example below.

```
Runtime error 5: Un-initialised variable: 'new_var'

  in read_port (mycode.vnm lines 21-24)
  in process_input (mycode.vnm line 30)
  in main (mycode.vnm line 39)
  in the command line.
```

The error text describes the error, and may give supporting information, in this case the name of the offending variable.

Then the report goes on to list the procedure the error occurred in, together with an estimated line number. It also gives the list of callers, i.e. the procedures that called the procedure that failed. Because Venom code is compiled, it is not possible for the error listing to locate the error exactly in all cases.

When a runtime error occurs, the task in which the error occurs will normally halt. If the error was in the main task then control returns to the command line. Unaffected tasks will carry on running.

Runtime errors causing program execution to halt like this is fine during the development of a program, but is unacceptable during operation.

Instead you can deal with runtime errors in two other ways:

1. Reset the controller when an error occurs
2. Trap the error

Reset on Error

Reset on error

Resetting the controller when any runtime error occurs is the default behaviour when the **Program Mode** switch input is 'off'. This behaviour is set up in the default startup procedure, by setting the 'system variable' ErrorAction.

Catching Errors

The Try/Catch construction allows you to *catch* or *handle* errors instead of just allowing the operating system to handle them - which involves stopping your program!

The Try/Catch construction typically has two parts: the *main code* and the *error handler*.

The main code follows `Try`.

The error handler follows `Catch`. It always starts by listing the name of a variable to hold the error number should an error occur.

```
Try
[
  main_code
]
Catch error_number
[
  error_handler
]
```

How it works

When the program reaches the Try command, the main code is executed. If no errors occur then the program jumps past the error handler and carries on as normal.

If there was an error, then the program immediately jumps to the error handler, where the error number may be tested to choose how to handle the error.

In the example below, the procedure will return the result of a division operation, or if there was a divide by zero error, it will print a warning (and return the value zero).

```
To try_divide(a,b)
```

```
    Local r := 0, error_code
    Try
    [
      r := a Div b ; try the division.
    ]
    Catch error_code
    [
      Print "caught error", error_code, CR
    ]
    Return r
  End
```

Passing on errors

If you want to distinguish between those errors you want to handle and those you want to pass on (either to the operating system, or to 'deeper' Try/Catch constructions), then you can use Exit to re-issue the error:

```
  #Define Div_ZERO_ERR 7
  To try_divide(a,b)
    Local r := 0, 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 to deeper error
  handlers.
    ]
    Return r
  End
```

Exceptions

Exceptions

Try is useful for handling errors - but it can also handle other exceptions. For example you may wish to jump right out of a set of nested loops. Exit will generate an exception (similar to a runtime error, but deliberate) that can be handled with Try.

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

Note that you don't have to use the Catch part of the construct - in this case if the code Exits, it just carries on after the Try block. However if you don't use Catch then you can't tell if the loop was exited because of Exit or a runtime error.

You can also use Try, Catch and Exit to jump out of deeply nested procedure calls. Exit always takes an integer value that may be used to distinguish an Exit from a runtime error, or between different Exit points:

```
To driver_code
  ...
  If unusual_event_a
    Exit 100
  ...
  If unusual_event_b
    Exit 101
  ...
End
```

later...

```
Try
[
  driver_code
]
Catch error_number
[
  If error_number = 100
    deal_with_unusual_event_a
  If error_number = 101
    deal_with_unusual_event_b
  Else
    Exit error_number ; pass on real errors.
]
```

Task end exceptions

When a task is commanded to Stop this is handled by a special runtime error. You can handle this error as an exception using Try and Catch, allowing your task to 'clean up' before it ends.

Here is some code that demonstrates how local objects and global locks can be cleaned up when a task ends, either naturally or if it is sent the Stop command.

```
#Define TASK_DEATH_ERROR 32 ; Error code.

To a_tidy_task
  Local error_number
  AutoDestruct
  Local buff := New Buffer(String)

  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
```

Runtime error codes

Note that you can find out the value of all the runtime error codes by typing **`Debug(13)`** at the command line.

Tidying up after exceptions

Because Try, Catch and Exit break normal program flow their use can result in parts of your application being left in unexpected states. For example objects might be left locked that should not be, or temporary objects might not be removed.

Using the restorative locking scheme can be used take care of any problems with locking.

AutoDestruct will handle problems with temporary objects.

Implicit locking

You may need to take care of locking even if you don't explicitly lock any objects in your code. Objects are sometimes locked by the system implicitly. This is usually in association with printing:

any object that is *printed to*, or is *printed*, is locked for the duration of the printing.

SUMMARY

- Try and Catch may be used to handle errors and exceptions.
- Exit may be used to generate an exception or jump out of nested loops.
- Exit may be used to simulate a runtime error.
- Take care with locking or temporary objects inside Try. See Tidying up after exceptions above.

Macros

Macros are pieces of program text that have been given a name. They are a very powerful tool, and can make your code easier to write and understand.

Things that may be useful to name using macros are

- Constant values
- Expressions
- Any text used in several different places in your program

Because a macro is defined in one place, you only have to make a change in one place to be sure that the change is reflected throughout your code.

Even though there are some special rules that need to be understood when using macros, they improve a program's readability so much that this is worth the extra effort.

Creating Macros

Macros are defined with the `#Define` construct.

For example:

```
#Define PI 3.14159
#Define clock_present net . Find(160)
#Define Age .Element(5) ;invent new message name
```

To use a macro, just use it's name in your code:

```
circumference := (2 * PI) * radius
```

This is the same as writing

```
circumference := (2 * 3.14159) * radius
```

Macros may also take parameters, for example:

```
#Define RGB(red,green,blue) (red  << 11 + green << 6 + blue) ;
Macro parameters

my_colour := RGB(12,2,19)
```

`RGB(12,2,19)` will evaluate to the value 24723 at compile time because of the constant folding built into the compiler.

Nesting Macros

Macros may be nested to any level. This means that a macro definition can include other macros. It doesn't matter which macro is defined first.

```
#Define hours (minutes * 60) ;a nested macro
#Define minutes (seconds * 60)
#Define seconds 1000
```

Note the use of () to make sure that, when the macro is used, the expression the macro may be embedded in is calculated with the correct precedence rules. Make sure there is a space between the end of the macro name and the **(**. *Putting a* **(** *immediately after the macro name indicates a macro that take parameters.*

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.
```

Constant Folding

If you define a macro where there is a lot of calculation of expressions then the compiler may be able to optimise the calculations so they are done at compile time rather than run time. This is called constant folding. For example the macro **hours** will be compiled down to the value **3,600,000** rather than the expression **60 * 60 * 1000**.

Redefining Macros

You can redefine macros using **#Define**, but if you do a warning will be issued if the text of the macro has changed in any way.

If you know that a particular macro will be redefined within your project, use **#ReDefine** instead. This won't issue a warning.

Removing Macros

If you want to remove the macro and use its name for something else then use **#UnDef <macroname>**

You can re-define the macro at any time.

Null Macros

You may define a macro to be nothing:

```
  #Define something
```

This will simply expand as nothing at all.

Macro Limitations

Current limitations that may be improved later

- Macros can only be one line long

SUMMARY

- `#Define` is used to define macros.
- Using macros to name constants and expressions makes for better programs.
- Macros can take parameters for more sophisticated expressions
- `Undef` is used to remove a macro - the name may be re-used for any other purpose
- `#Redefine` is used if you need to use a macro name for different purposes in the same project

Conditional Compilation

Conditional compilation is where the programmer can tell the compiler to include or not include parts of the program when it is compiled, or to provide different options within the code that are selected at compile time rather than run time.

This is often useful when there are different versions of the same basic application program that match different versions of the hardware, or where there are 'debug' options in the code that need to be turned on or off in a consistent way.

Conditional compilation is done using 'preprocessor' commands, which always start with a `#` symbol.

The simplest commands are

`#If` and `#EndIf`.

`#If` must be followed by a condition. The condition is an expression that the compiler can calculate immediately, and which must evaluate to an integer constant.

If the value of the expression is **not zero** then the lines of code between `#If` and `#EndIf` are passed to the compiler.

If the value is **zero** then the code is not passed to the compiler. It is as if they did not exist, or were commented out.

Example

In the example below, the line of debug code (which prints the values of some variables) is either included in the program or not based on the value of the macro DEBUG_FLAG.

```
#define DEBUG_FLAG True

To do_something
  x := a_function_of(y,z)
  #If DEBUG_FLAG
    Print "The value of x is: ", x, CR
    Print "The value of y is: ", y, CR
  #EndIf
  Return x
End
```

Other commands

There are two other conditional compilation commands: `#Else` and `#ELIF`.

`#Else` is used to include different code when the `#If` condition is zero:

```
#If CONDITION
  do_this
#Else
  do_that
#EndIf
```

#ELIF (short for 'Else If') is used to provide multiple clauses:

```
#If VALUE = 0
  this_code
#ELIF VALUE = 2
  that_code
#ELIF VALUE = 3
  the_other_code
#Else
  none_of_the_above
#EndIf
```

All these conditional compilation commands operate on blocks of lines in your program code. A line with one of these commands on it should not have any other code on it, though comments are OK.

Nesting

#If, etc, can be nested:

```
#If CONDITION_A
  #Define CONST_VAL 10
  #If CONDITION_B
    #Define MAX_VAL 34
  #EndIf
#EndIf
```

Optional parameters

Optional parameters

In Venom you can declare that some parameters to a procedure are optional - that is, you don't have to supply the optional parameters when you call the procedure.

Optional parameters are always at the end of the parameter list. When you call a procedure that has optional parameters you can only omit parameters from the end of the parameter list.

When an optional parameter is not supplied it is given the value *integer zero*, just like uninitialised Local variables.

Optional parameters are declared by putting `[]` around the optional parameters, as in the following examples.

Here two out of the three parameters are optional:

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

Here all the parameters are optional:

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

To call a procedure with optional parameters just treat it as if it took the number of parameters you wish to send as in the examples below:

```
proc1(1)
proc1(10,3)
proc1(10,3, )
proc2
proc2(1,2,3)
```

Processing optional parameters

Your procedure code will often need to know how many actual parameters have been passed to it - the Venom keyword `ParamCount` returns this number.

You can also access all the parameters to a procedure or method by using the Venom function `Parameter(n)`, which returns the value of the nth parameter, where the first parameter is `Parameter(1)`.

You can determine the data type of a parameter using the `TypeOf` operator.

See the Venom Help file for more information on all these Venom keywords.

Further Expressions

This chapter introduces some of the more advanced types of expression. Some of these subjects are covered in much more detail in the Venom2 Help File.

Initialising Global Variables

Venom will create a global variable the first time it sees its name. However, it doesn't assign the variable any value: you have to do that.

If you don't give a variable a value, then it has the value 'un-initialised' and it is an error to try to read it.

The procedure below is used to initialise a variable called **var**:

```
To initialise
  var := 1
End
```

However, **var** is still not assigned a value, as the procedure has not yet been called. Below, we've tried to read its value before it was assigned:

```
-->Print var
Runtime error 5: Un-initialised variable: 'var'
in the command line.
```

This is one of the most common runtime errors you will see. It is often good to make sure all the global variables you use are initialised in your init procedure, or elsewhere.

Using Hexadecimal and Binary numbers

In all of the examples so far, the numbers have been in decimal notation i.e. they are made up of the digits 0 to 9. In computing, other number bases are often used. Of these, Venom supports hexadecimal and binary. In hexadecimal (base 16), numbers consist of the digits 0 to 9 and the letters A to F (or a to f). To indicate that the number is hexadecimal, the **$** symbol should be used to 'introduce' the number. The example below prints the value of $3FFF.

```
-->Print $3FFF, CR
 16383
-->
```

Note that the number will still print as a decimal. Numbers can be printed in hexadecimal by using the **~** symbol. For example:

```
-->Print ~16383, CR
  3FFF
-->
```

A similar system is used for binary except that the symbol is **%** and values can be printed in binary using **~~**. For example:

```
-->Print ~~1253, CR, %100001, CR
10011100101
    33
-->
```

Characters and String Constants

Sometimes your program may have to deal with character information. To help with this you can express a *character constant* by using the ASCII character within single quotes:

```
-->an_integer := 'A'
-->Print an_integer,CR
-->    65
-->
```

String constants

String constants are used to hold strings of characters. These are usually expressed as text within double quotes, as in some of the `Print` commands we have seen already.

A string constant may be assigned to a variable:

```
-->str := "This is some text"
-->Print str,CR
This is some text
-->
```

Or passed as a parameter to a procedure:

```
-->ProcessText("Some text")
```

You may also find the length of a string constant:

```
-->Print str.Length , CR
    17
-->
```

Embedded text

There is another way to create string constants, which is called *embedded text*. Instead of using quote characters to delimit the string, the following delimiters are used:

```
<<<:this is the text>>>
```

There are several advantages to using embedded text:

1. Embedded text can spread over multiple lines
2. Embedded text text is literal, so no 'escape coding' is needed for special characters

Embedded text is mostly used to embed other languages within Venom code, e.g. HTML for web pages.

For example:

```
Print <<<:<h1>Title</h1>
  This is the body text<p>>>>
```

Notes

Note that all string constants are *constant*. You can print them, find their length, send them as parameters and refer to them with variables, but you can't change their contents. This is more than enough for many simple applications.

For more flexibility in text handling see String objects and Text Buffers.

Also see the sections on printing strings and Arrays of strings.

String constant concatenation

If two string constants appear one after another on the same or different lines, separated only by white space or comments then they will be *concatenated* - i.e. they are joined together as a single string constant.

This can be useful sometimes, for formatting your code into an easy-to-read form, but it can also lead to confusion.

For example the following array only has one (long) string in it, repeated five times, because all five quoted strings have been concatenated into one, and then the four empty elements of the array are filled with the last value supplied.

```
Array fruitnames (String, 5)
  "Apple"
  "Pear"
  "Banana"
  "Plum"
  "Apricot"
End
```

This is equivalent to

```
Array fruitnames (String, 5)
  "ApplePearBananaPlumApricot"
End
```

And printing it we see

```
-->Print fruitnames
ApplePearBananaPlumApricot
ApplePearBananaPlumApricot
ApplePearBananaPlumApricot
ApplePearBananaPlumApricot
ApplePearBananaPlumApricot
-->
```

To get the desired behaviour commas are needed between each string.

```
Array fruitnames (String, 5)
  "Apple",
  "Pear",
  "Banana",
  "Plum",
  "Apricot"
End
```

Escape sequences

Sometimes you might want to put the quote character, `"`, into a quoted string. This is obviously not possible in a straightforward way. Instead you have to use an escape sequence (or you might use embedded text).

An escape sequence is introduced with a `\`. To put a quote in a string use `\"`:

```
-->Print “There's a\” quote in here somewhere!”
There's a" quote in here somewhere!-->
```

The complete list of escape codes is:

Sequence	Yields the character	ASCII	
`\\`	Backslash: \	92	$5C
`\"`	Double quotation mark: "	34	$22
`\$hh`	Any ASCII character from hexadecimal	-	hh
`\a`	Causes a terminal to emit an audible alert	7	$07
`\b`	Backspace	8	$08
`\f`	Form feed	12	$0C
`\n`	New line	10	$0A
`\r`	Carriage Return	13	$0D
`\t`	Horizontal tab	9	$09
`\v`	Vertical tab	11	$0B
`\^hh`	Bitmap reference - hexadecimal number	-	hh

Strings on the command line

There is one more thing about string constants that you should know. Each time you use a string constant *on the command line*, then a small amount of memory is lost to the system until the next time the system restarts.

This doesn't happen when strings are used within procedures. It would only matter if your application relied on using large numbers of string constants on the command line, in which case you would need to restart Venom if memory got low. This would be a very unusual application of the Venom command line.

Bitwise Operators

Bitwise operators work at the level of individual binary bits within an integer.

The most commonly used bitwise operators are **`And`**, **`Or`**, **`Eor`** and **`Inv`**

`And`, **`Or`** and **`Eor`** each operate on two values.

And

`And` produces a binary '1' in the result only if there is a binary 1 in both the first value **and** the second value it is given, see the calculation set out here:

```
%100100101010101
And
%100110100101010
________________
%100100100000000
```

`And` is often used to reset bits within a binary number.

Or

`Or` produces a binary '1' in the result if there is a binary 1 in either the first **or** the second (or both) of the values it is given:

```
%100100101010101
Or
%100110100101010
________________
%100110101111111
```

`Or` is often used to set bits within a binary number.

Eor

`Eor` (for *Exclusive* Or) produces a binary '1' in the result if there is a binary 1 in either the first **or** the second value (**but not both**):

```
%100100101010101
Eor
%100110100101010
________________
%000010001111111
```

`Eor` is often used to optionally flip bits within a binary number.

Inv

`Inv` takes just one value and inverts each binary bit to give the result:

```
Inv
%100110100101010
________________
%011001011010101
```

An example of `And` and `Or` being used is

```
result := input_value And %10010101 Or %10 ; Reset some bits
and set others.
```

Shift operators

There are two more bitwise operators that shift bits within an integer to the left and right by a specified amount: **<<** and **>>**.

For example:

```
result := input_value << 5
```

or

```
result := input_value >> shift_amount
```

Memory Expressions

It is possible to access the memory of the your Venom-based controller directly using the **?** operator. This is not normally required. It is discussed in detail in the *Venom2 Help File*.

Pointer Expressions

Note to C programmers: Pointers in Venom are only used to 'pass by reference - never as pointers into memory. Everything you might use a pointer for in C you should use an object for in Venom. There may sometimes be an exception to this, in which case you can use the **?** *operator. This is discussed in the Venom2 Help File.*

Passing parameters by reference

When a command such as that shown below is executed, the value of variable alpha is set to the value of variable beta.

```
-->alpha := beta
-->
```

Occasionally it is necessary to get a reference (or 'pointer') to the variable instead of its actual value. For example, the following procedure was intended to take a variable and alter its value such that it lies between the two limits given. The procedure shown below is an example of its use.

```
To range(var,lo,hi)
  If var < lo
    var := lo
  If var > hi
    var := hi
End

-->num := 53
-->range(num,25,35)
-->Print num, CR
    53
-->
```

It can be seen that it has not worked – the value should have been changed to 35. The reason for its failure is that when the procedure is called, the value of num is given to the procedure. The procedure successfully ranges the value, but this has no bearing on the value of num. What is required is to give a *pointer* to num to the procedure. A pointer to a variable is obtained by placing a **@** symbol before it, as in **@num** – this is now a pointer to num. To 'follow' a pointer to its variable, the **!** symbol must be used. The correct version of the procedure is shown below.

```
To range(var_ptr,lo,hi)
  If ! var_ptr < lo
    ! var_ptr:= lo
  If ! var_ptr > hi
    ! var_ptr:= hi
End

-->num := 53
-->range(@num,25,35)
-->Print num, CR
    35
-->
```

Pointers to objects

It's not usually useful to take a pointer to an object.

This is because the variable name that refers to an object - like **dt** in the code below - is already a pointer, and it doesn't add anything useful to refer to that variable with a pointer.

```
Make dt DateTime ; 'dt' is like a pointer to the object
created.
```

Procedure Pointers

It is possible to obtain a pointer to a procedure. This can be useful when choosing between several courses of action.

Simple command languages may be created using this feature. The following example uses an Array of procedure pointers to act on commands coming over a serial network.

```
To turn_on
  led . On
End

To turn_off
  led . Off
End

To report
  Print led
End

Array procedure_table (@dummy,3)
  @ turn_on
  @ turn_off
  @ report
End

To run_protocol
  Forever
  [
    action := serial . Get – 'A'
    !procedure_table . Element(action)
  ]
End
```

The pointer expression

`!procedure_table.Element(action)`

Reads one of the elements out of the array depending on the value of `action`, and then uses this as a pointer. As the pointer is a procedure pointer, it calls one of the three procedures.

Parameters to procedure pointers

Procedure pointers may also be sent parameters. You have to help Venom decode this rather complicated construction by using parentheses.

Used as a statement

Additionally, if you are calling a procedure pointer as a Venom statement it is good practice to use square brackets, as shown below to stop the preceding expression (**proc_a** in this case) taking the opening parenthesis as a parameter list:

```
To procedure
  proc_a
  [(!procedure_table.(n)) (par1 , par2)]
End
```

If you didn't use the square brackets then the opening parenthesis would be seen as a parameter list to **proc_a**.

Used as an expression

However, if you wish to use the *value* returned by the procedure call, then the square brackets cannot be used. For example:

```
a := (!procedure_table.(n)) (par1 , par2)
```

Note that in both cases we have omitted name of the Element message. This is a shortcut allowed by Venom.

Further Objects

This chapter introduces some of the more sophisticated things you can do with objects.

Printing Objects

Just as numbers and strings may be printed, Print is also capable of printing objects. It has been shown already that the Print command can be used as shown below:

```
-->Print thermometer.Value, CR
    45
-->
```

However, the object itself may also be printed:

```
-->Print thermometer, CR
[Analogue: 45]
-->
```

It has a similar effect. The printing of the object is left up to the object itself – it is just instructed to print itself (it is sent a Print message internally). As a result, an object can print whatever information it deems to be useful. It uses the `[]` as a reminder to the programmer that it is not a simple number.

A particularly useful object to print is the I2C bus, usually called net. This lists all the devices it has connected to it.

```
-->Print net
Devices on the I2C network No.1:

Number   Channels    Device      Description
------   --------    ------      -----------

160     PCF8582/83...           RTC/EEPROM...
-->
```

In this case, an EEPROM was found.

When objects are printed, they may take account of any format specifiers sent (using the colon operator `:`). Each object will interpret print formatting in its own way. See the definition for each object in the Venom2 Help File.

Using Nil

There is a special type of object called **`Nil`**.

In short, it is an object that accepts any message (and ignores it).

`Nil` returns the value 0 (i.e. **`False`**) to any message that is sent to it, and ignores any text printed to it.

It can be useful to use **`Nil`** in the place of a real object or number to indicate 'nothing' or 'no value'. **`Nil`** may be tested for equality or inequality with anything else:

```
If x = Nil
  Print "x is Nil", CR
If x <> Nil
  Print "x is not Nil", CR
```

Object Expressions

When you create an object with Make, you are both creating an actual object in the Venom system, which may also affect the memory or hardware in some way, but you are also creating a variable that *refers* to that object.

```
-->Make str String(10000)
-->
```

Here, **`str`** is the variable that refers to the String object we created (which has no name of its own).

You can do all sorts of things to the *variable*, but that doesn't affect the *object*.

For example you can 'copy' the value of one variable into another:

```
-->str2 := str
```

This has not created another String, but *just another variable that refers to the same String*.

You can also 'lose' the object:

```
-->str := nil
```

Here, the original String still exists, taking up useful memory, but the variable **`str`** doesn't refer to it anymore and can't be used to access it. In this example we still happen to have a variable that refers to it (**`str2`**).

The above examples are not suggestions for how you should write your Venom code, but an attempt to illustrate how objects and variables are related.

Here are some examples of how objects may be used in a more sophisticated way.

Firstly, an object may be passed to a procedure as a parameter. For example, the procedure below will output ten pulses on any object that understands the message Pulse.

```
To pulse_ten(obj)
  Repeat 10
    obj.Pulse
End
```

Secondly, a procedure may create an object, modify it, and return it:

```
To give_me_a_string_filled_with(contents)
  Local str := New String(size)
  Print To str, contents
  Return str
End
```

In this case you should be careful to manage the objects created so as not to lose track of them, and thus 'leak' memory from your system.

Sending Messages to Expressions

Messages are only sent to objects, however, an object may be represented by an expression that is more complicated than a simple variable name. Some of these are:

- A procedure that returns an object as its result
- A message that returns an object
- Following a pointer

We will look at the most useful of these. The rest may be treated as exercises.

Sending messages to objects held in a Buffer

It is possible to create a special type of Buffer object that can hold any other kind of object - or any venom variable type. This is refered to as a Buffer of Any:

```
Make ba Buffer(Any)
```

You might choose to put some digital objects in it:

```
Repeat 16
  ba.Put New Digital(128+Index0)
```

Then you can send them any message acceptable to a Digital object - here sent to the object in the buffer's element number 5:

```
ba.(5).On
```

Creating Temporary Objects

Sometimes it will be necessary or useful to create an object on a temporary basis.

Temporary objects are often held in local variables. You can't use Make with a local variable, but you can use the keyword New:

```
Local workspace := New String(100)
```

It is necessary to remove these temporary objects, or else each time the procedure in which they are defined is called, a new set of temporary objects is created, but the old ones are not removed. This leads to a 'memory leak'. Eventually Venom will run out of memory, until it next restarts.

Usually its best to remove temporary objects when the procedure they were defined in terminates. However it can be very difficult to maintain code where every possible exit route from a procedure has to be covered.

AutoDestruct

AutoDestruct may be used to remove temporary objects no matter now a procedure terminates.

Any local variable that is declared *after* AutoDestruct is checked on exit from a procedure; if it contains an object, then the object is removed by sending a Die message.

AutoDestruct covers all normal procedure terminations - including End, Return, Exit and Try/ Catch. It doesn't cover the case when a task terminates on an uncaught runtime error or **Ctrl+C**.

Local variables that are defined *before* AutoDestruct are not checked.

```
To proc
  Local a := 10 ; Don't check this one
  AutoDestruct ; On exit, remove these:
  Local workspace := New String(100)
  Local temp := New Array(Int, 10)
  ...
End
```

Accessing dead objects

If you don’t keep track of the objects you are creating and destroying then it’s possible to try to access a dead object. Venom has an internal mechanism that detects this most of the time, but it’s better to get your software design right to begin with.

Here’s an example of how you might accidentally try to access a dead object:

```
-->Make b Buffer(String)
-->c := b
-->b.Die
-->c.Put(0)
Runtime error 25: Message to dead object
at $260408 in the command line.
-->
```

Here we kept a reference to the object, c. Even though we deleted the original reference, the other one could be used to try to access the object. Venom intervened with a runtime error to make sure the system didn't crash in an undefined manner.

Further Printing

This chapter introduces further facilities available from the Print command, and introduces the PrintF message.

Text Handlers and Redirection

Normally Print sends its output to the main serial port. It is possible to print text to a different object using Print To.

Changing the object to which text is printed is called *redirection*. The object to which the output is printed might be one of the types listed below, which are able to accept print.

Object Type	**Result**
AlphaLCD	The text is displayed on the alpha-numeric LCD
SerialPort	The text is sent directly to the serial port
Buffer(String)	The text is printed to the Buffer
File(String)	The text is printed to the file
GraphicsLCD.Window	The text is printed in the window
RealTimeClock	The text is interpreted to set the time and date
DateTime	The text is interpreted to set the time and date

For example, the following prints the value of counter on the LCD.

```
-->Print To lcd , counter , CR
-->
```

Note that the comma after the object being printed to is compulsory.

The object to which a plain Print (without 'To') normally sends print jobs may also be changed – see the Operating System message Output, in the Venom2 Help File.

Further Printing Keywords

As well as the printing keywords already introduced, there are a number of others. No single object understands all of these.

Keyword	**Effect**
BEEP	Emits a beep on the terminal
BS	Moves the cursor back one character
CENTRE	Changes the text justification mode of the graphics LCD to centred text
CHR n	Prints an ASCII character of the given number
CLS	Clears the screen
CR	Moves to the beginning of a new line
FONT	Changes the font that the text is printed in
GOTOXY(x,y)	Moves the cursor to the given location
HOME	Moves the cursor back to the top-left of the screen
LEFT	Changes the text justification mode of the graphics LCD to flush left text
RIGHT	Changes the text justification mode of the graphics LCD to flush right text

For more details on the use of a keyword on a particular device, refer to the object type in the next part of this manual, or refer to the Venom2 Help File.

How PRINT works

All the text handlers described above accept ‘extended’ text consisting of normal and extended characters. It is the task of the Print command to convert all the expressions in the print list into acceptable extended text. During this process it divides up the total print output into packets of text. Each 'packet' is a collection of up to 200 characters that are assembled before being passed on to the text handler as a *print job*. Short Print statements will cause single, less-than-full print jobs to be used, and very long ones will require many print jobs, one after the other, to carry the output.

Since it is important that no more than one task writes to an object at any time, the Print command locks the object being printed to while all the text is sent – only when it has finished will it unlock it.

When an object is asked to print itself, it may lock itself so that its contents are consistent during the print process.

PrintF

Venom2 also has a PrintF message that can be sent to any object that can accept print (i.e. can be PRINTed To). This works in a very similar way to the C function of the same name

In general Print is more useful at the command line - for printing small numbers of items, and PrintF is more useful for printing within your application - for variable items embedded in a line of text. Both may be used in either context.

Venom2 has extensions that add to C's printf, e.g. **%o** means print an object, **%b** is for binary.

In application code

The lines

```
serial.PrintF("The temperature is %iC on %2o at %3o\n",
temperature, clock, clock)
```

and

```
Print To serial, "The temperature is ", temperature, "C on ",
clock: 2, " at ", clock:3, CR\n"
```

both produce this output:

```
The temperature is 0C on 17-Feb-09 at 13:11:40
```

But the first is easier to write, understand and maintain.

At the command line

However, at the command line it's a lot easier to use

```
Print clock
```

than

```
PrintF("%o", clock)
```

See the Venom2 Help File for details of how PrintF is used.

Further Multitasking

This chapter discusses the more complicated aspects of multitasking. You only need to read this if you can't fit your application into our simple model, though you may want to read it out of interest.

Task Management Scheme

The Venom operating system uses a 'Round-Robin, Pre-emptive' task manager. 'Round-Robin' means that there is a simple list of tasks, and each is scheduled to run in list order, starting over again when it reaches the end of the list. There is no task-priority scheme.

'Pre-emptive' means that the Venom operating system takes charge of when task swaps happen – it's not under your control.

Atomic operations

An atomic operation is one that can't be split into more than one part by a task swap. Venom has some operations that are defined as atomic.

- Writing a variable
- Reading a variable
- Locking an object
- Semaphore object operations

The atomic nature of these operations is important, to allow tasks to share simple resources easily.

Processing Power and Task Latency

There are penalties for using a large number of tasks. Clearly, if Venom is handling many tasks at one time, it cannot run them as fast. Also, a significant amount of memory is required for each extra task running.

Since, at a very low level, the microprocessor in the controller cannot execute more than one of its instructions at a time, multitasking has to be done by rapidly switching between the tasks, executing a little bit of each task at a time. This has the appearance of all the tasks running at once. If all the tasks are using all the processing power the hardware has to offer, then each task will slow down proportionally to the number of tasks that are running. So if there were three tasks, each task would be running at one third its normal speed. Fortunately this is unusual as most real applications have tasks that do some work and then wait for a period of time or for an external event. A waiting task takes very little resource.

Unfortunately there is a system property that does suffer with every task added: Task Latency.

Venom will swap tasks roughly every 1mS, with maximum task duration of 2mS. Thus if there are, say, five tasks running, the maximum time that any one task will have to wait is 5 x 2mS = 10mS. This is known as the *latency* of the system. It is likely that most of the time no task will have to wait this long, but many applications are dependent on the *worst-case* delay so they don't miss important external events.

Task Objects

When you start a new task the **`Start`** command returns a *Task* object. This object is associated with the task and it allows you to monitor or control the task. For example you can send the task object the following messages:

`Done` - returns True if the task has finished (i.e. it is dead)

`Die` - this does the same as **`Stop taskobj`**

Task-local state

The task also has an active variable called **`State`**. This can contain any Venom value - usually an object - in fact it's best if it's a user-defined object.

Setting the task's **`State`** allows you to associate a whole set of values with a task - it is as if it has its own private set of global variables.

Note that a task's **`State`** is initialised to the value **`Nil`**.

Current task object

There is an object that represents the *current task*, and it is represented by the keyword **`Task`**. For example, you can set the **`State`** for the current task:

```
Task.State := New MyClass
```

Setting up a task's state

This is the best way for a new task to set up its State:

```
To task_code
  AutoDestruct
  Local state := New TaskState
  Task.State := state
  ;...
End
```

Note **`AutoDestruct`** is used to automatically remove the state object when the task dies.

You can detect when a newly started task has initialised it's state by comparing it with **`Nil`**:

```
To main
  tsk := Start task_code
  Await tsk.State <> Nil
  ;...
End
```

And this is a skeleton Class you might use for a task's state:

```
Class TaskState
  Value Int

  To Initialise
    Value := 10
  End
End
```

You can add any members and methods to the Class that you need for your application.

Sharing Resources

Every application has inputs, outputs and data storage of some kind. Each of these things is called a *resource*. For example, a keypad is a resource, and so is a display. Sometimes two or more resources are always used in conjunction with each other, and may usefully be considered a single resource. In this case, the keypad and display constitute the user interface resource.

Allocate Resources to Tasks?

Whenever two or more tasks have to share a resource you will need to be careful in your programming. In fact, it is sometimes so hard to design an application that will share a resource among tasks in real time that it is often easier to rewrite the application so that each major resource is only ever accessed by one task. In our earlier example this is illustrated by the user interface task, which is the only task that ever 'talks to' the display.

Easily-Shared Resources

Some kinds of resources are very easy to share. These are the ones that have a single value that you want to read, but not write to. Sharing these is just a matter of reading the value. An example of this is reading a Venom integer or float value from a global variable.

Signalling between Tasks

Because reading a variable won't cause resource-sharing problems, the simplest way for tasks to pass information to each other is by using global variables to signal each other. You just have to obey the rule:

Only one task owns a variable at any time.

That is, only one task may write to the variable, though any number may read it at any given time.

We saw the example of the value target_temp being used this way earlier.

Task-local state

Though is it simple to signal between two tasks using a global variable, it is not easy to do anything at all complex or sophisticated using global signals. It is better to use the task's **State** to do the signalling. Because the state may be a user-defined Class, you can create any signalling scheme you want to.

Synchronising Tasks

There are circumstances where you can have a global variable signalling two ways.

In these cases, the value of the variable performs the function of determining which task owns it. For example, the main task may signal another task to go and perform an operation. The other task can use the same signal to report that it has finished. In the example below, we assume each procedure is running in a separate task.

```
To main_task
  signal := True        ;signal sub-task to act
  do_something_else
  Await signal IsFalse ;wait for signal back
  carry_on
End

To sub_task
  Forever
  [
    Await signal       ;wait for signal before acting
    do_operation       ;act
    signal := False    ;signal back
  ]
End
```

Note that you can use a task's State to signal in similar ways.

Sharing other resources

The kinds of resources that are difficult to share are those that are written to, or otherwise affected, by more than one task.

The most common problem encountered with these sorts of resources is where a task might interfere with (or corrupt) another task's use of the resource.

The next most common problem is where an important task is held up while it waits for a slower task to finish using a shared resource.

Examples of where you will need to consider the implications of sharing a resource are:

- Sharing a display device – if you are not careful the display may be messed up
- Reading from input devices like a serial port or a keypad – one task may 'steal' keys or characters that another task was expecting
- Using data structures like Array and Buffer – the data set may become inconsistent
- Using networks and buses – you may mess things up on the bus, or hold up other traffic

Consider this simple example:

One task has a line of code that increments a global variable. A different task has a line that decrements the same variable.

```
count := count + 1 ;in task A ; In Task A

count := count – 1 ;in task B ; In Task B
```

Say count starts off with the value 4. It is possible that Task A could read count and add 1 to it. Then, just before it wrote 5 back into count, Task B could come in, read count, and subtract 1 from it, and write back the answer 3. Task A could then get back control and finish its job, writing 5 in count. '5' is, of course, the wrong answer. The wrong answer only happens when a read-then-write access to count is broken by a task swap that also needs to write count.

As this kind of problem only shows up intermittently, it is important, early in the software design process, to identify situations where it might arise and make sure it is dealt with before it shows up, rather than in the field.

Ask yourself the question: 'Are there any resources or variables in my system that more than one task has to affect?'

It is worth repeating: it is often easier to re-write an application so that none of its tasks have to share a resource than to try to make resource sharing work.

Sometimes you just have to share resources among tasks. Resource Locking is the mechanism that allows this to work. This is discussed in the next section.

Idling

In order to save electrical power, the Venom operating system will use a 'halt' instruction in the microcontroller's instruction set, to put the CPU into an idle mode whenever possible. It's worth knowing which kind of Venom instruction will cause the controller to idle, and those which keep it awake.

The most common examples are Wait, Await and Every.   See the appendix F: Optimisation for more details.

Local Variables and Tasks

It is possible for two tasks to call the same procedure at the same time. If all the variables altered by the procedure are local variables, there will be no problems, since not only are local variables local to a procedure, they are also local to the task as well. More accurately, they are local to each particular time a procedure is called (termed an 'instance' of the procedure).

Whenever a procedure is called, it allocates some *stack* memory in which to store the values of the local variables and so, even if there are many instances of a procedure, they will each have their own set of values for the local variables and will therefore be entirely independent.

You should beware of using global variables and objects in procedures you call this way, as you may run into the resource-sharing problems detailed above.

Task-local variables

Each task may be given it's own private set of 'global' variables by assigning an object to the task's **`State`**.

See here for more details.

Locking

Problems can occur if a number of tasks use the same object. For example, if a number of tasks were all writing to an LCD at the same time, the result would be unintelligible - or might even cause the system to crash.

To solve this type of problem, objects may be 'locked' and 'unlocked'. This means that one task can claim the object as its own ('locking' the object), and then it can do anything with it – other tasks that want to use the object must wait until its owner (the task) finishes with it (and 'unlocks' it). Clearly, an object may only be locked by one task at a time.

You may also use the locking mechanism to deal with the shared global variable problem seen before. This kind of statement:

```
count := count + 1
```

when used in multiple tasks is called a *critical area* of the code. This is discussed later.

Implicit Locking

Some Venom operations lock one or more objects in order to carry out their work.

The most common example is of this implicit locking is printing. When you print *to* an object (i.e. send text to it), the Venom Print Manager will always lock the object for the duration of the Print command. Thus you can be assured that the output from the Print command will never be interrupted by another task's print output.

Also, some objects will lock themselves while they are *being* printed, so that they remain in a stable state for the duration of the print operation.

Because of implicit locking you should be able to write most Venom applications without having to explicitly lock anything.

Locking Objects

Some object types have a lock mechanism built into them. Objects with no lock will still accept all the locking messages, but will ignore them. Locking behaviour is documented as part of each object's definition.

When you lock an object, if the object is not locked, or it is already owned by the current task, then the object is claimed for the current task and execution continues normally.

However, if the object has already been locked by another task, then the current task has to wait. While waiting, the lock is periodically tested until it becomes free. It is then claimed, as above, and execution continues normally.

When a task is waiting for a lock to become free, it does not use its whole 1mS task slot. Instead, it swaps out immediately, allowing other tasks to run.

Task lists show blocking

When you list out what all your tasks are doing using List Task or Ctrl-T, the listing will tell you if any of the tasks are *blocked, i.e*, waiting on a lock held by another task.

Here the user hit **Ctrl-T** while two tasks where running, one blocked by another:

```
Task ID: 0
in proc1 (working.vnm line 87)
in the command line.
___________________
Task ID: 2
Blocked by Task 0
in b.Lock
in proc2 (working.vnm lines 88-89)
in a task started from proc1 (working.vnm line 86).
___________________
```

Incremental Locking

The classic way to lock an object, and then unlock it is illustrated below:

```
Obj . Lock
Obj . … ;Use the object
Obj . Unlock
```

Locking and unlocking may be 'nested': if you lock an object twice, you will need to unlock it twice before it is available to other tasks:

```
Obj . Lock ;if it was free here…
…
Obj . Lock
Obj . … ; [Use the object]
Obj . Unlock
…
Obj . Unlock ;…it'll now be free here.
```

Restorative Locking

The above scheme works fine for most code, but consider what would happen if your code had to deal with an exception by using Exit while an object was still locked.

```
Try
[
  obj . Lock
  obj . ...  ;use the object
  if exception
    Exit 100
  obj . ...  ;use the object
  obj . Unlock
]
```

The object no longer gets unlocked as many times as it should. This problem can also occur when using Return or when responding to runtime errors with Try.

You can avoid situations like this with complicated programming, but there is another way of using locking that avoids the problem altogether.

This scheme uses the fact that Lock returns a result: the number of times it was locked before the Lock message.

```
N := obj . Lock
```

Additionally, Lock may be assigned a value, N, to lock an object to a given level (as if Lock had been called N times).

```
obj . Lock := N ;Lock obj N times
```

This allows the following scheme to be used:

```
To proc
  Local lock_state
  lock_state := obj . Lock ; Lock it and record pre-existing
lock level.
  obj . …   ;Use the object
  obj . Lock := lock_state ;restore pre-existing level.
End
```

This will work even if exceptions cause some lock-restoring commands to be missed.

You can also send the message `obj.Lock(0)` if you want to make sure the object is not locked by the current task at a particular point in your code. If the object is actually locked by another task then this silently does nothing.

This scheme is called *restorative locking*.

Non-Blocking Locking

With the Lock message, you have the risk that a task cannot lock an object immediately, and thus your task may have to wait (or be *blocked*) for an indeterminate time.

If this is not acceptable you can use the **TestLock** message. This tries to lock the object. If it succeeds then the object is locked and **TestLock** returns the new level of locking (which is non-zero). If it fails (because another task already owns the object) then **TestLock** returns zero.

If the task does manage to lock the object, then it may be unlocked with **Unlock**.

```
To proc
  If obj . TestLock    ;Try for the lock…
  [
    obj . …         ;Use the object
    obj . Unlock      ;restore old lock level.
  ]
End
```

To fit in with the restored locking scheme described above, TestLock returns the number of times the object has been locked. The example below shows how to use it.

```
To proc
  Local lock_state

  ;record lock level:
  lock_state := obj . TestLock

  ;Did we get the lock?
  If lock_state
  [
    obj . …   ;Use the object
   ;restore old lock level:
    obj . Lock := lock_state - 1
  ]
End
```

Notice we had to subtract 1 from the lock level to restore the lock, as TestLock returns the post-locking level, as zero is used as the return value for failure to secure the lock.

Lock Owner

Occasionally, most likely during development, it may be useful to find out which task has an object locked. The Owner message on any object will return the task-object that has object locked, or NIL if the object is not locked.

Deadlock

Deadlock is the 'fatal' tangling of tasks and locks illustrated by the following example:

In Task 1:

```
ObjA . Lock
...
ObjB . Lock
```

In Task 2:

```
ObjB . Lock
...
ObjA . Lock
```

If these bits of code are executed at roughly at the same time, then it's possible that Task 1 will lock objA and Task 2 will lock objB. From then on both tasks are stalled forever, as neither can ever lock the other object it needs, nor unlock the object the other task needs.

Deadlock will only show as an intermittent problem, so it's important to eliminate it at the software design stage.

If you construct your code correctly then deadlock can never happen. The trick is to make sure that any resources that are locked at the same time by a number of tasks are always locked in the *same sequence* in each task:

In Task 1:

```
ObjA . Lock
...
ObjB . Lock
```

In Task 2:

```
ObjA . Lock
...
ObjB . Lock
```

Task listing

You can see if deadlock has occurred by examining a task listing - it will show any blocked tasks. Deadlock is where two or more tasks are mutually blocking each other. Here task 0 is blocked by task 1 *and* task 1 is blocked by task 0 - so neither can proceed.

```
Task ID: 0
Blocked by Task 1
in proc1 (working.vnm line 87)
in the command line.
____________________
Task ID: 2
Blocked by Task 0
in b.Lock
in proc2 (working.vnm lines 88-89)
in a task started from proc1 (working.vnm line 86).
____________________
```

Implicit locking may be important

Remember to consider implicit locking when you are thinking about deadlock.

Ending Tasks

In general it makes a multitasking program simpler to think about if the tasks you use never end.

One of the complications of ending a task (one of the reasons it's easier if tasks never end) is that it is often necessary to clean up the resources that the task has used. This means removing any temporary objects it has created and unlocking any objects it has locked.

Another complication is that if a task ends then it's likely that you will need to start another one at some point, to perform the same function. Usually you will have to find some way of being sure the old task has really ended before you try to start the new one.

However it is sometimes useful to have tasks that end.

There are three different ways to end a task:

1. Let it end naturally
2. Signal to it to end from an external signal
3. Stop it from another task

We will consider all three below.

Let a task end naturally

This means the code just comes to a natural end after completing its job. Tasks that end naturally are the easiest to clean up after. You just write them as if they are normal bits of code (which also have to clean up after themselves).

Consider the procedure below, which we start as a task.

```
To task_1
  AutoDestruct
  Local str := New String(100)

  Serial.Lock
  do_something_with(str, serial)
  Serial.UnLock
End
...
Start task_1
...
```

Notice that AutoDestruct is used to clean up the temporary String object, and that though we lock serial, it is always unlocked before the task ends.

Signal a task to end

If a task would not end naturally - e.g. it usually runs in loop - then you can signal to it to end.

Consider the procedure/task below. Again, we use AutoDestruct to remove the temporary object, and the Unlock is always performed before the task ends.

```
To task_1
  AutoDestruct
  Local str := New String(100)

  Serial.Lock

  Every 100
  [
    do_something_with(str, serial)
    If task_stop_signal
      Break
  ]

  Serial.UnLock
End
```

Stop a task from another task

When a task is stopped using `Stop` or `Stop All`, the task has no control over the point at which it stops executing. This is very crude - it's a bit like a runtime error. It could leave objects controlled by the task in any state. In fact, `Stop` is processed just like a runtime error. Which means that you can trap the error using `Try`, and attempt to clean up.

Consider the code below.

```
#Define TASK_STOP_ERROR 32

To task_1
  Local err
  AutoDestruct
  Local str := New String(100)

  Try
  [
    Every 100
    [
      Serial.Lock
      do_something_with(str, serial)
      Serial.Unlock
    ]
  ]

  put_IO_in_safe_state
  Serial.Lock(0) ; Unlock fully
End
```

Here we've used Try/Catch to catch the task stop 'error'. Again, we can use AutoDestruct to remove any temporary objects at the end of the procedure/task.

But because we can't tell exactly where the task was interrupted, we can't know the 'lock state' of any objects we may have locked (explicitly or implicitly in Print, say). Luckily we can use the Restorative Locking scheme to make sure that an object is fully unlocked.

But also you may also have to make sure that any I/O you were controlling, or other systems, are put into a safe state before the task finally dies.

Critical Areas

Sometimes you will need to lock a whole area of code, not just access to a single object. An example of this was given above:

```
count := count + 1 ;in task A

count := count – 1 ;in task B
```

To control access to areas of code you can use an object just for its lock. The object to use is a Semaphore as this is intended for this job:

```
Make code_lock Semaphore
...
code_lock . Lock
count := count + 1 ;critical area in task A
code_lock . Unlock
...
code_lock . Lock
count := count – 1 ;critical area in task B
code_lock . Unlock
```

Semaphores

The Semaphore object is more than just a lock - though it's likely it will be used mostly just for its lock.

However, it also implements the classic *semaphore* function for resource allocation. It's not likely you will need to use this, but you can read more about its detailed functioning in the Venom2 Help File, and about the use of semaphores in computing in many online resources.

Internal Operation

Here are some further details of the internal operation of the Venom Task Manager. You should not need to understand these for most purposes.

Sharing highly contended resources

The venom task manager attempts to share access to highly contended resources in a fair manner.

Say there are three tasks all needing frequent access to a highly contended resource such as a file system or network connection.

All three tasks claim the resource by locking it, then use it for a time, and then release it for use by other tasks by unlocking it.

It is possible in this scenario for any one of the tasks to 'hog' the resource through accidents of timing.

Say a task claims the resource, then releases it for a while before quickly claiming it again, as in the code below:

```
To SendPackets
  Forever
  [
    network.Lock
    Network.Put(packet)
    network.Unlock
  ]
End
...
Start SendPackets
...
```

If there is no task swap in the time between the network being unlocked and then locked again, the other tasks will never get to lock the network - which is quite likely in this code because the lock is executed again only a few instructions after the unlock.

To get around this problem, if a task has a resource locked, and another task also tries to lock it, the Venom task manager will set a flag in the lock (called 'task waiting'). When the owner task releases the lock, if the task waiting flag is set then the task is forced to swap immediately, so preventing it from locking the resource again before other tasks have had a chance to use it.

Dead tasks holding locks

When a task comes to an end, or is stopped by another task, if you haven't coded the task well it may be holding one or more global objects in a locked state.

If a task dies leaving an object locked, and then another task tries to lock the object, then after a short time (less than N tasks x 512mS) the task/lock manager will detect that the live task is waiting to lock an object that can never be unlocked because its owner doesn't exist anymore. At this point it will silently unlock the object so other tasks may claim it - your program may seem to slow down for a fraction of a second. Though this doesn't usually cause a major problem it is much better to ensure that a task cleans up after itself.

It is also possible to have the system issue a runtime error (*Attempt to lock object held by dead task*) so you can detect this situation arising during development. See *OperatingSystem. Debug* in the Venom2 Help File.

Removing objects that other tasks are trying to use

When Die is sent to an object with a lock, if any other tasks were waiting to lock the object then these are allowed to lock it. When they are finished with the object it is then removed from the system.

If there is any attempt to use the object after this then the runtime error *Message to dead object* is issued.

The End

This is the end of the Venom2 *Language* Tutorial - you have been introduced to all the major parts of the Venom language.

If you haven't already done so, we recommend that you now look through the second part of this manual (Part 2: Object Tutorial) and get familiar with objects by using them.

Then you will be ready to start writing your own application for VM2. There is a checklist for how to plan and complete your application in Appendix A: Development Checklist.

Part 2:Object Tutorial

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 different functions. The first part of this tutorial used several types of object in the code examples.

The second part of the tutorial is a more intensive exploration of some of the more commonly used objects. They are grouped according to the kind of functions they perform: Input/Output, User interface, Data storage, Operating system…

The principals illustrated using these objects may be extended to cover all the other objects in Venom2.

For a complete description of every type of object see the Venom2 Help File.

A note to those familiar with C++, Java, etc:

The Venom system predefines many useful types of object. These have been written and tested by Micro-Robotics Ltd, and are supplied as part of the Venom language. This set of object types was created to handle the functions most commonly required in small to medium sized industrial control systems.

It is possible to create new object classes, though it's likely that these will be used more for data processing than low level device drivers, as they are written in Venom which is not well suited to this purpose.

We are happy to consider suggestions for new pre-defined objects.

Hardware dependence

Most of the objects in the Venom library interface with real input or output devices of one kind or another. Where possible this manual will not assume any particular hardware set-up.

Venom Channels

Before we go any further it's worth talking about *Channels* in Venom. A channel is a single Input/Output pin in your hardware system that has been given a number. This is purely a convenient way for Venom to refer to specific bits of hardware; it doesn't necessarily correspond to any numbers given to pins by the manufacturers of the ICs concerned.

For example, digital I/O channels on the main I2C bus are numbered from 128 to 255. Digital I/Os coming directly off the microcontroller are numbered from $10 upwards; the datasheet for the VM2 and the Application board have details of these.

Analogue inputs and outputs have a similar channel numbering. Where possible the channel numbers are detailed in this manual. If they are hardware-specific see the datasheet for your controller.

Some analogue devices supported by Venom2 don't use channel numbers - these will be described individually.

Digital

The Digital object type is designed for reading and writing digital input and output. It doesn't matter whether the I/O is located on the controller itself, on an IC connected to an I2C bus – Digital will handle them all.

Creating Digital objects

The following three lines create Digital objects.

```
Make VM2_ip Digital ($2E, 0)
Make VM2_op Digital ($2F, 1)
Make i2c_dig Digital (128)
```

The first two lines create digital input and digital output objects to control I/O pins on the VM2 controller.

The first parameter is the channel number on the VM2, and the second parameter gives the attributes of the digital channel - in this case just telling it to be a standard input or output.

The third line creates a digital object on a chip attached to the I2C bus. This doesn't need to be assigned input or output functionality as it can handle both implicitly.

Setting outputs

Now we can send messages to the objects to control them:

```
VM2_op.On    ;Turn it ON.
VM2_op.Off    ; Turn it OFF
```

Reading inputs

You can read the state of an input or an output with the Asserted message:

```
-->Print VM2_ip.Asserted,CR
    1
-->
```

The 1 (True) means the input was asserted - or ON. If it had been OFF then 0 would have been returned. There is no single standard for whether logic low and high mean on and off – it depends on the device you are dealing with. However it is most common for ICs to treat low as on.

Digitals on the direct VM2 channels may be defined with ON being high or low voltage.

Asserted may also be used to *set* the state of a digital output:

```
VM2_op.Asserted := True
```

Printing

Digital objects print '`ON `' or '`OFF`' depending on their state (always 3 characters).

Other messages

There are some other messages that Digitals understand like **`Toggle`**, which inverts the state of an output, and **`Pulse`**, which pulses the output.

Digital channel numbering

Channel numbers for digital channels on the controller itself are detailed in the controller's datasheet. They lie in the range $10 (16) to $7F (128).

Digital I/O channels on an I2C bus (using the PCF8574 IC) have well-defined channel numbers. These are listed in the table below.

Each PCF8574 will provide eight digital channels.

There are two types of PCF8574: the normal part and the A-suffix part PCF8574A.

These are identical except for the I2C address each responds to. Venom allocates different channel ranges to each type, and deals with the addressing transparently.

For example, a PCF8574 chip, connected to the main I2C bus (number 1), with it's address lines set to 000 (Low Low Low) will have digital channels 128 to 135. If the address were changed to 001 (Low Low High) then it would have digital channels 136 to 143.

Some other objects (Keypad, AlphaLCD) use these digital channel numbers as an easy reference to a particular PCF8574 they are using.

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

Similar Object Types

For pulsed digital signals see PulseCounter, PulseWidthOut, PulseWidthIn, and Shaft.

Analogue

The Analogue object type is designed for reading and writing analogue input and output. It doesn't matter whether the I/O port is located on the controller itself, on an IC connected to an I2C bus – Analogue will handle it.

Input

The following two lines of code make two analogue inputs:

```
Make an_1 analogue ($30)   ;analogue input on the VM2
Make an_2 analogue (256)   ;analogue input on I2C bus
```

To read the input use the Value message:

```
Print an_1.Value, CR
   173
-->
```

The number returned is an integer read from the Analogue to Digital Converter device (ADC). This represents the voltage applied to the actual input. In this case we will assume that the ADC has 12-bit resolution (i.e. a full scale from 0 – 4095 comprising 4096 steps) and that full scale is 3.3 Volts. Thus the reading in volts would be

```
-->Print an_1.Value * 3.3 / 4096 , CR
     0.844727
-->
```

Analogue channel numbering

Channel numbers for analogue I/O on the controller itself are detailed in the controller's datasheet.

For analogue I/O on the I2C bus, see the table below.

Channel	I or O	I2C Device	Addresses	I2Cbus
256 - 287	Input	PCF8591	000 - 111	1
288 - 295	Output	PCF8591	000 - 111	1
512 - 543	Input	PCF8591	000 - 111	2
544 - 551	Output	PCF8591	000 - 111	2

Accuracy and Resolution

The resolution of an analogue I/O device is not the same as its accuracy. The resolution limits the size of the smallest signal you can measure.

The overall accuracy is limited by the resolution, but also by many other parameters of the ADC or DAC. These include the device's offset error, full-scale error, linearity, temperature drift, input or output impedance, etc.

The resolution is sometimes expressed as bits - e.g. 12 bits. Or it may be expressed as the size of the LSBs - the smallest ananlogue signal that may be detected or generated.

If you need a measurement error smaller than 10 LSBs, in general you will need to use the device's data sheet and add up the sources of error.

Output

Some channels can be analogue outputs:

```
Make an_out Analogue ($14)    ;analogue I and O on the VM2
```

Writing to the output is done with Value yet again:

```
an_out.Value := 4095 Div 2 ; set the output at half the supply
rail
```

You can also play audio files using an analogue output - see Analogue in the Venom2 Help File.

Similar Object Types

Analogue values can also be represented in microcontroller systems using pulsed I/O.

See **`PulseCounter`**, **`PulseWidthOut`**, **`PulseWidthIn`**, **`FrequencyIn`**, and **`Shaft`**.

AlphaLCD

The AlphaLCD object can drive any alphanumeric LCD display that is controlled by the Hitachi HD44780 controller IC.

There are usually several ways to attach alphanumeric LCDs to your system: usually directly to the controller board, and also using a PCF8574 IC on an I2C bus. You can have as many LCDs as you like, though most applications will only need one.

The make command specifies the number of characters across the display, the number of lines on the display and the 'location', i.e. how you have connected it to the system.

```
Make lcd AlphaLCD (20 , 2 , 0)
```

This command initialises a 20 character by 2 line LCD attached directly to the VM2's parallel bus.

```
To print text to the display, use Print To.
Print To lcd, "some text"
```

The text will appear on the top line of the display starting at the left hand end.

AlphaLCD understands several Print keywords to modify the printed output:

Keyword	**Action**
CLS	Clears the display, setting the cursor to the top left
HOME	Puts the cursor at the top left
GOTOXY (X,Y)	Sets the cursor to the X – Y position specified
CR	Does a carriage return

GOTOXY takes two parameters in parenthesis. Remember to specify X (characters along the row) first. Note that the character positions and the lines are numbered from zero.

CR will move the cursor from one line down to the start of the next. If the cursor is on the bottom line, then the text on all the lines will scroll up, and the cursor will remain on the bottom line.

Location numbers

The table gives the location numbers to use for the different ways to connect LCDs.

Location	Number
VM2 parallel bus	0
PCF8574 on the I2C bus	Use one of the channel numbers as the location, e.g. 128.

Similar Object Types

See **`GraphicsLCD`** in the Venom2 Help File, for a more sophisticated user interface.

Keypad

The Keypad object class will drive several types of keypad.

Each type has been given a number. Most of the keypad circuits use one or two PCF8574 ICs on the I2C bus, which further drive matrix keypads of different sizes and shapes. However, you can also use a Touchscreen as a keypad - this is covered in the Venom2 Help File and in example code on our website.

The table below shows the type numbers to use for the different types of keypad.

Matrix Keypad	**PCF8574 Devices**	**Type Number**
4 by 4	1 needed	0
8 by 8	2 needed	1
12 by 4	2 needed	2

You can use these drivers to drive smaller keypads, for example at 4 x 4 driver can drive a 3 x 2 matrix.

The Make command for Keypad takes the type number, and then either one or two channel numbers, each of which specify a PCF8574 on the I2C bus.

An easy way to find out the channel numbers of any PCF8574s on an I2C Bus is to print the I2CBus object:

```
-->print net
Devices on the I2C network No.1:

Number    Channels    Device      Description
------    --------    ------      -----------

 124      240-247     PCF8574A    8 digital I/O lines
 126      248-255     PCF8574A    8 digital I/O lines
 162      PCF8582/83...           RTC/EEPROM...
-->
```

Here we can see that we have two PCF8574As on the I2C bus. We'll use both to make the keypad…

```
Make kpd Keypad (1 , 240 , 248)
```

This will drive an 8 by 8 keypad on two '8574A chips, on the I2C bus, near the top of the PCF8574A address range.

Getting Key presses

There are several ways to read the keypad. The simplest, but least flexible, uses the message Get:

```
-->Print kpd.Get, CR
     5
-->
```

Here the key decoded as '5' was pressed (key numbers always start from 0).

See the Venom2 Help File for how the keys are numbered on the matrix.

Get will de-bounce the keypad, and make sure that a long key-press is treated as only one action, by waiting for no keys to be pressed before it can look for a new key-press.

The disadvantage of using Get is that while no key is pressed, the message will wait, blocking the execution of any other code in the current task.

Another way to read Keypad that does not wait is to use the Asserted and GetLast messages:

```
Every 50
[
  If kpd.Asserted
  [
    Select Case kpd.GetLast , CR
      Case 0
      [...]
      Case 1
      [...]
      ;etc.

    While kpd.Asserted ; 'De-bounce' the keypad
      Wait 30
  ]
]
```

This code will wait for a key-press to be detected. Asserted scans the keypad and returns True if any key is pressed. GetLast reports the particular key found by Asserted.

The last line of text in the code example is used to make sure that the code only acts on individual key presses by waiting while the key is still being pressed. This line does cause code execution to halt if a key is held down, which might be a problem in some applications. There are ways around this, but it may be better to use a Keypad InputBuffer instead - which we deal with next.

Keypad InputBuffer

A Keypad InputBuffer is used to collect discrete key presses and buffer them up so that you can use them at a rate that suits you.

It is neater than using Get or Asserted/GetLast in many circumstances.

To turn on the Keypad's InputBuffer, use the following:

```
kpd.InputBuffer(10 , 25)
```

The parameters supplied are the *auto-repeat rate* and *auto-repeat-delay* for the keys.

You may omit these parameters if auto-repeat is not required.

Updating

Keypads with InputBuffers have to be 'updated' in order to do their work. This is achieved by sending the message Update to the Keypad. Update has to be sent both often and regularly for the InputBuffer to work well: every 30mS seems to be a good rate.

A typical use of a Keypad with an InputBuffer in a user interface might be

```
Every 30
[
  kpd.Update
  key_press := kpd.Key ; Key pressed?
  If key_press >= 0
  [
    ; select the action for each key.
    Select Case key_press
      Case 0[]
      Case 1[]
      Case Else[]

    ; Update the display if necessary
    update_display
  ]
]
```

Another way to update the Keypad is to start a task that calls Update regularly. Update has been written so that you don't have to worry about resource sharing.

```
Start Every 30 kpd.Update
...
Forever
[
  key_press := kpd.Get ; Wait for key to be pressed

  ; select the action for each key.
  Select Case key_press
    Case 0[]
    Case 1[]
    Case Else[]

  ; Update the display if necessary
  update_display

]
```

NumberReader

NumberReader allows you to set up a keypad to enter numbers into your application calculator-style. You tell it which keys represent the digits 0-9; minus sign, decimal point and so on, and it will assemble numeric input for you.

For visual feedback the digits of the number may be displayed on a display device (e.g. an LCD) while the number is being entered.

Creation

```
Make nr NumberReader
```

You don't have to supply any parameters if you want to use it for decimal numbers.

Conversion

The Conversion message tells the NumberReader which keys on your Keypad to use for its functions.

Since different keypads have different mappings between the identifiers on the faces of the keys and the logical key numbers, we need a way to specify to the NumberReader which keys are to be used for which purpose. The Conversion 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 the logical key numbers range from 0 - 15.)

Example

Here is an example of the use of Conversion. Note how the Delete function is disabled by being assigned to key ‘-1’:

```
#define Delete_KEY  -1
#define DECIMAL_KEY 3
#define MINUS_KEY   12
nr.Conversion  ;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
)
```

The first three parameters are the key numbers for the DECIMAL POINT, MINUS, and Delete functions.

1. MINUS is for entering negative numbers
2. DECIMAL is for entering a decimal point
3. Delete will delete the number you have entered one character at a time

The rest of the parameters are the logical key numbers for the decimal digit keys.

Redefining the keypad

`Conversion` may be re-sent at any time to change the functions of the keys.

The keypad

This conversion list above has been created for the this keypad:

```
 ; Logical Key numbers:
 ; 0    1    2    3
 ; 4    5    6    7
 ; 8    9    10   11
 ; 12   13   14   15

 ; Tile face legends:
 ; 1    2    3    F
 ; 4    5    6    E
 ; 7    8    9    D
 ; A    0    B    C

 ;Functions:
 ; 1    2    3    'Float' (dec point).
 ; 4    5    6    Enter
 ; 7    8    9    Delete
 ; -    0    B    Cancel
```

Reading Numbers

In the example below, we send key presses to the NumberReader as they come in. Every time we get a key we print the NumberReader to the LCD so the operator can see what's going on.

We only stop assembling characters when we get the ENTER key.

```
#define ENTER_KEY  7
#define CANCEL_KEY 15

To get_number
  Local k
  Forever
  [
    Print To nr, 100 ; we can set a default value
    nr.reset
    Forever
    [
      Print To lcd, HOME, nr ; visible feedback
      k := kpd.Get       ; get a key press
      Select Case k      ; decide what to do with it
        Case ENTER_KEY
           Break        ; exit loop
        Case CANCEL_KEY
           Print To nr, 0 ; reset number to 0
        Case Else
          nr.Put (k)     ; give key press to NumberReader
    ]
    Print nr.Value , CR  ; Print the value we got
  ]
End
```

You can use Keypad's InputBuffer.Key to get keys if you don't want your code to wait for keys inside Get.

Default Value

If you want the NumberReader to hold a default value at the start of number entry (to prompt a user to accept a default, say) then you can print to the NumberReader.

More

NumberReader has more features than we list here. Please see the Venom2 Help File.

OnBoardLED

The OnBoardLED object is used to control the LED on the controller board. The LED output is brought out on a connector pin so you can connect your own LED to it if the controller is not visible.

The behaviour of the LED at reset is determined by the Venom operating system:

- When the controller is waiting at the *Clear Memory* prompt in Program Mode, the LED is lit continuously.
- When the controller is in Run Mode the LED flashes at around once per second.

With this default behaviour, if the LED is unlit, then you can assume there is no power, or the controller is damaged. If the LED is lit continuously then the controller has been left in Program ModeMODE.

As soon as Venom starts running your code you can exert control over the LED. In general you should use the LED to signal the status of your application, in keeping with the default behaviour listed above. If the default behaviour is sufficient then you need not change it.

Messages

The LED behaves rather like a Digital object and will obey many of the same messages: On, Off, Asserted, Print and so on.

Flashing

The LED can be made to flash:

```
led.Flash($80)
```

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 examined from right - and ends when there are no more 1 bits to the left.

Example patterns are given below - but you can create your own.

Pattern in hexadecimal	**Pattern in binary**	**Description**
$80	%10000000	Short flash approximately once a second
$A0	%10100000	Two short flashes approximately once a second
$A8	%10101000	Three short flashes approximately once a second
$A9DDCA80	% 1010100111011101110010101 0000000	Signals "SOS" in Morse Code
$3AE77700	% 0011101011100111011101110 0000000	Signals "OK" in Morse Code

String Objects

We dealt with string constants earlier in this tutorial.

String constants are fixed strings of text. They always appear within double quotes, for example:

```
"This is a string constant"
```

You can create a variable that *refers to* a string constant:

```
str := "This is a string constant"
```

And you can re-assign the variable -

```
str := "new string"
```

But all you have done here is to point str at the first string constant, and later point it at the second string constant. The contents of the string constants is exactly the same throughout - they can not change unless you re-write the procedure or array in which they appear.

Variable text

String objects are similar to string constants, but you can *change* the text they contain. They are useful for manipulating text, or for creating variable text as the program runs.

String objects are created with Make or New - as with any object.

```
stro := New String(100)
```

When you create one you have to supply the maximum size of the text it is to hold - known as the *capacity* of the String. It can't hold any more text than this, though it can hold less.

When it is first created, a string object is empty - it contains no text.

You can use Print or Put to put text in it:

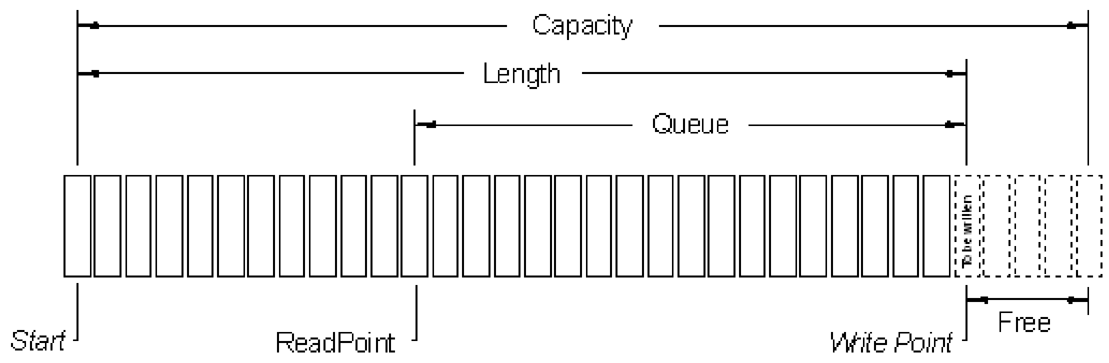
```
-->Print To stro, "Some text"
-->stro.Put("Some more text")
-->Print stro
Some textSome more text-->
```

Structure

The diagram below shows the structure of a string object.

Each of the boxes represents a single character in the string - such as 'A'.

New text is added using Print or Put at the *Write Point*.



The string has a length - which you can read using the Length message. This is the number of characters in the text.

It also has other properties that are available to read or write using other messages:

Get - gets one character from the ReadPoint and moves the ReadPoint forwards one character.

Queue - the number of characters left to Get

Free - the amount of empty space left in which to append more text in the string.

Element (or **.()**) - read or write any character in the string.

More printing

You can use colon format specifiers to print just a portion of the text within a String. If you use one colon, then you can print the left-most or right-most characters. Using two colons allow any segment of the buffer to be printed. This works in exactly the same way as printing string constants.

```
-->Repeat 5 Print stro: Index0 : 5,cr
Initi
nitia
itial
tial
ial t
```

Finding text

You can find the location of any sub-string in a String using the Find message.

```
-->position := stro.Find ("XXX")
-->Print position
    5-->
```

If the search string is not found then Find returns the value –1.

You can specify where the search is to start using an optional second parameter to Find.

```
position := stro.Find ("XXX" , start_pos)
```

The search is carried out from the start position towards the end of the String.

Find can also use another String object or a text buffer as the search text.

Buffer

A Buffer is a general data-storage object. Buffers are able to hold a collection of values, as opposed to the single values held by variables.

Buffers may be used to log data; hold varying amounts of data; form first-in-first-out (FIFO) queues; form 'circular buffers' and so on.

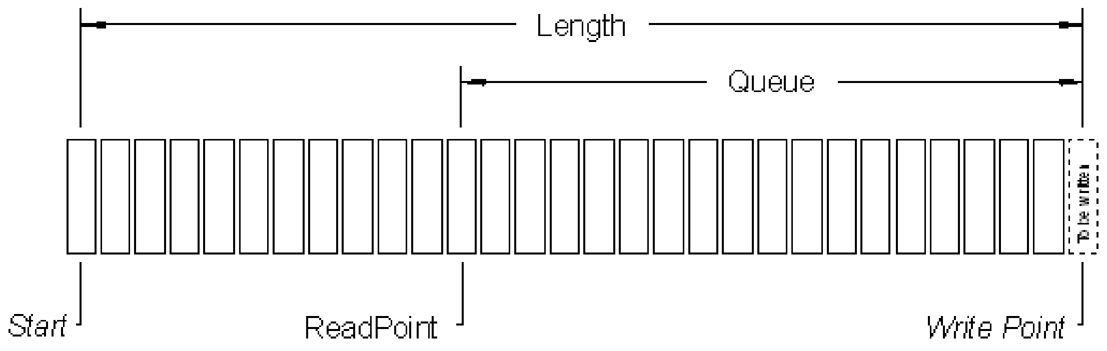
Diagram

The diagram below may make the operation of the buffer easier to envisage.

Data is written into a buffer at the write-point. When data is written, the buffer grows to the right of the diagram.

Data is read at the read-point. After each read, the read-point is moved one space to the right. The data that has been read out is not deleted; the read-point just moves on.

The read-point may be repositioned to any point in the buffer. Also, each 'element' of a buffer may be read or written-to 'randomly'. Any particular element may be accessed in any order.



Data types

Buffers can hold many different types of data. They can hold 8,16 and 32 bit integers, floating point numbers and also text. Buffers that hold text are referred to as 'text buffers', and sometimes operate in a different way to 'numeric buffers'.

Most buffers can only hold one type of data; in general you can't mix data types in a single buffer.

To create a buffer, you need to indicate the data it is to hold:

```
Make b8  Buffer(Int 8)   ;holds 8-bit integer data
Make b16 Buffer(Int 16)  ;holds 16-bit integer data
Make b32 Buffer(Int)  ;holds 32-bit integer data
Make bf  Buffer(Float) ;holds floating point data
Make bt  Buffer(Char)  ;holds text
```

Buffer of Any

However there is one buffer that can hold any type of data. We refer to it as a *Buffer of Any*.

```
Make ba  Buffer(Any) ;Buffer of Any
```

It can hold integers, floating point numbers, object references, and pointers to anything. Additionally, the type in each element can be different in the same Buffer.

The disadvantage is that it uses 8 bytes for each element stored.

Buffer of any may be used to build up complex data structures - e.g. arrays of string objects, or arrays of procedure pointers or user Class objects.

Filling a buffer

To put data into a buffer, you can use the message Put. This line puts five consecutive integers into the buffer, i.e. 3, 4, 5, 6 and 7.

```
Repeat 5 b8.Put (index0 + 3)
```

Printing a buffer

You can print a buffer to find out what's in it:

```
-->Print b8
     3
     4
     5
     6
     7
-->
```

Printing a buffer lists out it's numerical contents in a column. If you use colon formatting then the format is applied to each element as it is printed.

Reading a buffer

To read data out of the buffer you can use the message Get

```
-->Repeat 3 Print b8.Get
     3     4     5-->
```

Each Get reads the next data item in the buffer, starting from the beginning. Get does not remove items from the buffer, it just reads them in sequence. If you attempt to read past the write-point of a buffer (i.e. read data that isn't there) a runtime error will occur.

Flushing a buffer

You can remove the data that has been read by using the Flush message:

```
-->b8.flush
-->Print b8
     6
     7
-->
```

Here, **the elements between the start and the current read-point** have been removed, leaving the unread elements in the buffer.

Other Buffer messages

A Buffer may accept several other messages. These are listed below.

Message	**Action**
`Length`	Returns the total number of data items in the buffer
`Queue`	Returns the number of data items available to be read with Get
`ReadPoint`	Sets or returns the position of the read-point
`Reset`	Resets the read-point back to the start of the buffer
`Empty`	Removes all the data from a buffer
`Element (n)` or `(n)`	Accesses a single element of the buffer - see below
`GetLast`	This *removes* an element from the *end* of the buffer - so you can implement FILO, or stack-like structures.
`Remove()`	This removes a section of data from any point in the buffer: start, middle or end.

See the Venom2 Help File for details of all Buffer messages

Element

The Element message may be used to access any element of the buffer.

As Element is quite a long but frequently used message in Venom, an abbreviation may be used:

`b8.Element (n)` is equivalent to `b8.(n)`, i.e. you can just leave the message name out if the parentheses are there.

Here we look at the zeroth element of the buffer, then change it's value.

```
-->Print b8.(0)
     6-->
-->b8.(0):= 10
-->Print b8.(0)
    10-->
```

If you attempt to access an element that doesn't exist, a runtime error will occur.

How big can a buffer get?

There is no limit to the size of a buffer apart from the memory it takes.

A buffer takes as much RAM as it needs to hold its data. Of course this means it's possible for a buffer to use the entire RAM available. In this case Venom will issue the runtime error Ram Full.

Buffers take RAM in small blocks, so that they don't rely on the memory manager having large blocks of contiguous memory available.

If you need to keep an eye on how much RAM is available in your controller, then you can use the system message Free.

```
-->Print Free
 98276-->
```

This returns the number of bytes left in the 'heap'.

Text Buffers

Text buffers are sufficiently different to numeric buffers to warrant discussing separately. Text buffers do all the things mentioned above in the same way as numeric buffers, apart from Print. They also do many things that numeric buffers don't do.

This is how you create a text buffer:

```
Make tb Buffer(Char)
```

or

```
tb := New Buffer(Char)
```

A text buffer is similar to a numeric buffer with 8-bit integer elements. You can put and get 8-bit values, but these values are treated as ASCII when performing textual operations.

A text buffer is also similar to a String object.

Text buffers are more efficient than String objects in some situations, and less efficient in others. The main difference is that text buffers don't have a fixed capacity - instead they use chains of memory blocks to store the text. They tend to be used for larger amounts of text.

Printing to and from a Text Buffer

When you print a text buffer it prints out the text it holds. However, if you want to see anything, first you must put some text in it. One way to do this is to print to it:

```
-->Make tb Buffer(Char)
-->Print To tb , "Initial text"
-->Print tb
This is some text-->
```

When you print to a text buffer, the new text is appended on to any existing text in the buffer:

```
-->Print To tb, "more text"
-->Print tb
Initial textMore text-->
```

Selecting what to print

You can use colon format specifiers to print just a portion of the text within a text buffer. If you use one colon, then you can print the left-most or right-most characters. Using two colons allow any segment of the buffer to be printed. This works in exactly the same way as printing string constants.

```
-->Repeat 5 Print tb: Index0 : 5,cr
Initi
nitia
itial
tial
ial t
```

As with strings, you can also implement scrolling text with this feature.

Inserting text

As well as appending text to the end of a text buffer, you can also insert text anywhere within a text buffer using the message Insert.

```
-->Print tb
Initial textmore text-->
-->tb.Insert("XXX",5)
--> Print tb
InitiXXXal textmore text-->
```

You can also insert the contents of a text buffer into another text buffer.

Finding text

You can find the location of any sub-string in a text buffer using the Find message.

```
-->position := tb.Find (“XXX”)
-->Print position
    5-->
```

If the search string is not found then Find returns the value –1.

You can specify where the search is to start using an optional second parameter to Find.

```
position := tb.Find ("XXX" , start_pos)
```

The search is carried out from the start position towards the end of the buffer.

Find can also use another text buffer as the search string.

Array

Array is a data storage object intended for the following uses:

Storing fixed-size tables of constant data in the ROM (or rather, the Flash memory)

Storing tables of variable data in RAM

Creating Constant Arrays

Because they hold data that is constant during an application, Arrays of constant data are rather like procedures. They sit in your code file, just like procedures.

The following code creates an Array of 10 8-bit integers.

```
ARRAY ar8(Int 8,10)
  1,2,3,4,5,6,7,8,9,10
End
```

The parameter `Int 8` specifies that the Array will store 8-bit integers.

You can also create Arrays of 16 and 32 bit integers, floating point numbers, pointers, and string constants. The following lines indicate how each of these should start.

```
ARRAY ar16(Int 16, 10)

ARRAY ar32 (Int 32, 10)

ARRAY ar_float(Float, 10)

ARRAY ar_ptr(@dummy, 10)

ARRAY ar_str(String, 10)
```

Auto fill

When the contents of an Array are not fully defined, the un-specified elements are filled with the last defined value.

Array of pointers

In the case of Arrays of pointers, the type is indicated by supplying any pointer. I have used a dummy variable to make it clear that it doesn't matter what the pointer points to: you must use the `@` symbol, but you can have any name you like.

```
ARRAY ptrs (@dummy)
  @variable1,
  @var2
End
```

Array of strings

Arrays of string constants may not seem straightforward at first. Here's an example of their use.

```
-->ARRAY ar_str(String ,5)
02>"Karl"
03>"Clive"
04>"Della"
05>End
ARRAY Defined (69 bytes @ $26027A)

-->Print ar_str.(0)
Karl-->
```

Each element of the Array is a string constant. This is very different to a text buffer, which is rather like an Array of characters.

Printing

You can print the contents of an Array.

```
-->Print ar8
     1     2     3     4     5     6     7     8     9    10-->
```

You can read out the data using the Element message…

```
-->Print ar8.Element(4)
     5-->
```

or using the shorthand for Element: `.()`

```
-->Print ar8.(4)
     5-->
```

But you can't write to it.

RAM copies of Arrays

If you need to have an Array that you can write to, but that is initialised with constant data, then you can take a copy of a constant Array:

```
-->ar_copy := ar8.Copy
-->print ar_copy
     1     2     3     4     5     6     7     8     9    10-->
```

You can now write to the elements in ar_copy, like this

```
Ar_copy.(0) := 2
```

There are other, more usual ways to create arrays you can write to...

Variable Arrays

You can create Arrays of variable data that aren't copied from constant Array. This might be useful if you need to create Arrays dynamically for temporary data storage. Or it might be wasteful to use up ROM space with initialisation data this isn't needed.

The syntax is rather similar to the constant Array:

```
 Make a Array (Int 8 , 10 ,1,2,3,4,5)
```

Or

```
 a := New Array (Int 8 , 10 ,1,2,3,4,5)
```

Here the type and size are Int 8 and 10, just like the constant Arrays. The rest of the numbers are optional initialisers. Again, if there are fewer initialisers than Array elements then the last initialiser is used to fill the rest of the space.

Unlike the constant Array syntax, you may use variables in the parameter list for Make or New Array.

Because the initialisers are in the parameter list, it's not a good idea to use many of them: it will put lots of data on the *stack*. If you need a large amount of initialising data, use the constant Array syntax and take a copy.

RealTimeClock

RealTimeClock keeps track of the actual calendar date and time using the real time clock/ calendar module (RTC) built into the VM2.

This module will keep the correct time even when power is removed if the Lithium battery is fitted and holds charge.

It is possible to calibrate the RTC to around 30 seconds per year drift, assuming constant temperature. If the temperature is not constant then the time keeping will drift according to the clock crystal's F/T curve.

It is possible to adjust the RTC calibration according to a measured temperature, though this is unlikely to be worth the effort.

Creation

```
Make clock RealTimeClock
```

This line will make the RealTimeClock, but there should be no need to type it as it is already in the default startup procedure.

Clock not set

If the clock has not been set (or has lost it's setting) then the Time message will report zero seconds and the Valid message will return False.

Dividing up the time

You may need to extract elements of the date for use in your application code, for example you may want to know if it's a Friday. RealTimeClock does not allow you to do this directly, because of the problem of skewing – for example the seconds value might roll over from 59 to 00 between reading the seconds and the minutes.

You should use a DateTime object to split up a time value into its date and time constituent parts: year, month, day, hour, minute, second, and the day of the week.

Setting the Clock

You can set the time in the clock by setting its Time. One way to do this is to use a DateTime object to find the number of seconds since 1990:

```
Make now DateTime
now.year := 2012
now.month:= 4
...
Clock.Time := now.Time
```

Printing to the clock

You can also set the clock by printing to it, which may be easier on the command line:

```
-->Print to clock , "2012-4-6 9:48:12"
```

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`

Printing the Date and Time

If you want to print the current date and time you can simply Print the RealTimeClock object. It will print out it's current date and time in ISO format:

```
-->Print clock
2012-04-11 00:02:12-->
```

You can also specify exactly which parts or the date or time you want to printed, how you want to see each part formatted, by supplying a format string that contains special codes to format each part of the date or time.

For example you could use the following (note the use of the colon `:` to introduce the format string)

```
Print clock:"h:mmtt, ddo MMMM yyyy"
```

This would result in a date presented in this format:

```
3:15pm, 11th April 2012
```

The table below has a list of the special date/time format codes. Any other characters will appear in the output untranslated, such as the comma and spaces in the example above.

If you want to put literal text in the output, but it contains special format characters, then you can use backslash (`\`) as an escape code, or `<` and `>` as literal text delimiters.

Date/Time formatters

Code	Meaning
`a` or `aa`	*am* or *pm*

`d`	Day number 0-31
`dd`	Day number 00-31
`ddd`	Day, abbreviated name
`dddd`	Day, full name
`h`	Hour 1-12
`hh`	Hour 01-12
`H`	Hour 0-23
`HH`	Hour 00-23
`m`	Minute 0-59
`mm`	Minute 00-59
`M`	Month 1-12
`MM`	Month 01-12
`MMM`	Month, abbreviated name
`MMMM`	Month, full name
`o`	Day ordinal: the characters after the day number in 1st, 2nd, 3rd, 4th, etc.
`s`	Seconds 0-59
`ss`	Seconds 00-59
`y` or `yy`	Year 00-99
`yyyy`	Year as 4-digit number
`\`	Next character is literal (note that you will need to use two \ characters to actually enter a \ into a quoted string).
<*text***>**	Embed literal text between < and >

The most up-to-date list of special date/time formatting codes appears in the *Venom2 Help file*.

You can change the *locale* of the day, month and ordinal strings for different languages - see **`OperatingSystem.Debug`** in the *Venom2 Help file*.

'Venom Seconds'

RealTimeClock stores the time as the number of seconds that have elapsed since the base date: midnight on 1st January 1990. We call this version of time *Venom Seconds*.

The message Time returns this number, and allows it to be set:

```
-->clock.Time := 0
-->Print clock.Time
     4-->
```

There was a 4 second gap between the user setting the time and then reading it back.

Calibration

The clock's initial accuracy depends on the crystal oscillator circuit used by the clock IC. This is usually accurate to around a couple of seconds a day, depending somewhat on temperature.

It is possible to calibrate the clock using the Adjust message, with a precision of around 1ppm, which is equivalent to 30 seconds per year, assuming constant temperature. See the Venom2 Help File for more details.

Date Extent

The RealTimeClock can work with dates up until the year 2090, at which point a software upgrade will become necessary.

DateTime

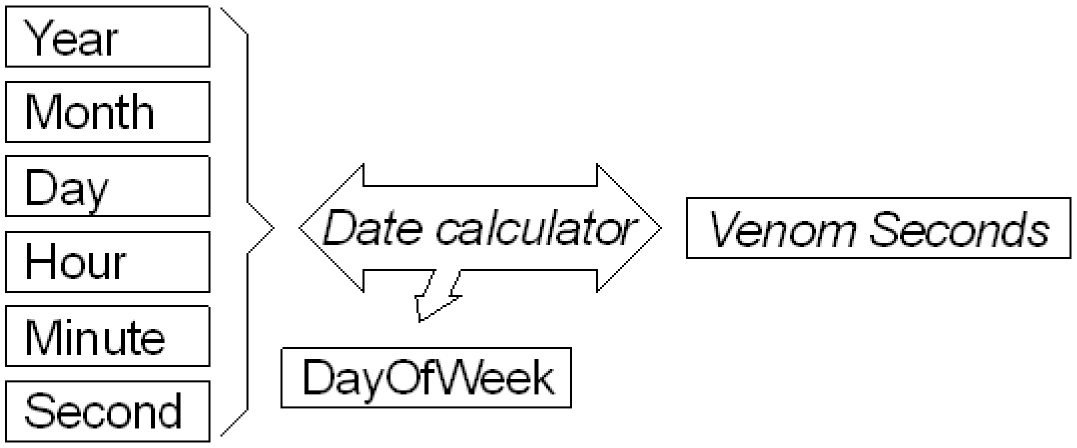
DateTime objects are primarily date calculators.

DateTime objects perform the complex calculations that translate between the calendar-date/ time-of-day that we are familiar with, and Venom Seconds.

The diagram illustrates how they work.

Each DateTime object holds values of both the calendar date and time, and also the time in Venom Seconds.

If the Venom Seconds value changes, then the date and time values will be updated to reflect it. If any one of the date or time values changes, then the Venom seconds value will be updated. The DayOfWeek value is purely a function of the date, and so cannot be altered directly.



Creation

You can make as many DateTime objects as you want, though each one uses a small amount of memory.

```
    Make when DateTime
```

Spurious Dates

It is possible to enter non-existent dates into a DateTime, for example 30th February.

A DateTime will normally Print a 'real' date: in this case the 2nd March in non-leap years. However, the Day and the Month elements of the DateTime will still hold the spurious date! You can elect to print the spurious date if you like. See the Venom2 Help File.

Of course, if a spurious date is set, and then Time is read, Time will always reflect real date, as there are no spurious values of Time.

Fixing

You can fix up spurious dates by sending the Update message.

```
when.Update
```

This is the equivalent of

```
when.Time := when.Time
```

Days of the Week

The DayOfWeek message returns the day of the week as a number.

Day	DayOfWeek
Sunday	0
Monday	1
Tuesday	2
Wednesday	3
Thursday	4
Friday	5
Saturday	6

The DayOfWeek value cannot be set as it is a function of the date.

Number ranges

The various elements of the date and time have number ranges associated with them, which you should obey else an error will be issued.

Element	**Range**
Year	1990 – 2089*
Month	1 – 12
Day	1 – 31
Hour	0 – 23
Minute	0 – 59
Second	0 – 59

**The Year value is four figures.*

Printing a DateTime

Printing a DateTime prints the date and time held in the object in ISO format, or with special formatting, just like printing the RealTimeClock.

Also see the *Venom2 Help File* for more on printing DateTime.

Assigning a date and time

When a DateTime is first created, its time is set to zero seconds. You can set its time and date in one of four ways:

1. Print to it, like the RealTimeClock
2. Set its Time in Venom seconds
3. Set its date and time elements individually
4. Call the Adjust message to set it like a digital watch.

The code below illustrates these

```
Print To when , "4-1-02 10:23:00"
…
when. Time := 378987780
…
when. Year := 2002
when. Month := 1
when. Day := 4
when. Hour := 10
when. Minute := 23
when. Second := 0
```

Printing to a DateTime should obey the same rules as printing to the RealTimeClock.

If you want to use a DateTime to extract the date and time in the RealTimeClock so that you can break it down into it's elements, you can put the clock's time into a DateTime first…

```
Make now DateTime
now.Time := clock.Time
```

Altering the date and time

Often you will want to allow the operator of your equipment to alter a date and time - for example to set or correct the time in the Real Time Clock.

There are many ways to do this, and one is to use the Adjust message on a DateTime object.

Adjust

Adjust allows you to implement ‘digital watch’ style methods to change the date in a DateTime 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 value.

To decrement a value, use a value of ‘-1’ instead 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.

Day value

Notice that there are two choices for nudging the day value - one that uses the range 1-31 and the other that only uses the number of days in the month currently held by the DateTime object.

See the Vemom2 Help File for more detailed information.

Timer

The Timer object is a millisecond countdown timer. You can give it a time period, set it going and test it to see if it has timed out.

Here we make a Timer with a default time period of 10 seconds.

```
Make t Timer (10000)
```

… and set it going…

```
t.Go
```

You can test whether a Timer has finished using Done. Done will return True when the Timer has finished timing.

```
...
Await t.Done
...
```

Other Messages

`Period` will set and read the Timer's time period in milliseconds.

`Time` will set and read the period remaining in milliseconds.

Printing

You can print a Timer in various formats to show how much time it has left. You should use colon formatting to get the format you want.

```
-->Print t:1 , CR
00:00:10
-->
```

:	Printed Format
:0	DD:HH:MM:SS
:1	HH:MM:SS
:2	MM:SS
:3	SS

Key
D is a day digit (range 0-24)
H is an hours digit (range 0-23)
M is a minutes digit (range 0-59)
S is a seconds digit (range 0-59)

Note: there are more formatting options for Timer - see the *Venom2 Help file*.

Stopwatch

Stopwatch is a millisecond up-counter. You can use it to time how long things have taken.

Stopwatch will start counting milliseconds as soon as it has been made.

```
 -->Make s stopwatch
 -->Print s. Time
   7395-->
```

You can reset to zero at any time with Reset.

Note that after around 24 days the Time returned by a stopwatch will overflow, and is not easily usable. The overflow will cause Time to go to the most negative integer value and count towards zero, from where it will carry on as normal.

Printing

Stopwatches print in the same ways as Timers.

SerialPort

SerialPort objects control serial communication ports. You may already have been using one of them as the default output device for the Print command.

The five ports can operate at standard baud rates up to 115,200 Baud, and higher for non-standard Baud rates.

Creating a SerialPort

Here we create a serial communication object on port 1:

```
Make serial SerialPort(115200,1,1)
```

The three parameters define the port's *baud rate*, *serial port number* and *handshaking method*.

The port numbers range from 1 to 5. Port 1 is the main serial port usually used to communicate with the terminal window in your development system.

The handshaking parameter value is shown in the table:

Value	**Handshaking**
0	NONE
1	Hardware
2	Software

The baud rate and handshaking can also be changed after the port has been created.

Messages

The main messages you need to know about are Put, Get and printing.

Get fetches a character from the serial input buffer. If there is no character in the buffer, Get will wait

```
character := serial.Get
```

Printing to the serial object sends the print output to the serial output buffer. Each character is taken in turn from this buffer, and transmitted. If there is no room in the buffer, then the print will wait. Serial is the default print output device.

```
-->Print "Fred"
Fred-->
```

Put sends a single character to the serial output buffer.

```
-->Serial.Put('A')
A-->
```

If your application should not wait for an indeterminate time for the input and output buffers, use the Free, Look and Queue messages.

More messages

SerialPort objects can take many more messages. Please see the Venom2 Help File for full details.

OperatingSystem

OperatingSystem is used to mop up quite a lot of general system functions that would otherwise clutter the Venom language.

It only makes sense to have one OperatingSystem object, and this is defined in the default startup procedure

```
Make system OperatingSystem
```

Shortcut

Because of a shortcut in Venom, some system messages can be sent without the `system.` in front of them - when seen by themselves they imply the system object. Thus the two lines below are equivalent, and will reset the controller.

```
-->system.Reset

-->Reset
```

The full list of messages that are implicitly sent to the system object is

```
Run
Reset
Debug
PrintF
Protect
```

Operating System Messages

Here some of the most useful operating system messages are described. The rest are documented in the Venom2 Help File.

ErrorAction

If set to the value 1, this message will restart the Venom system if a runtime error occurs.

```
System.ErrorAction := 1
```

This is essential for robust applications, but is just annoying during development, as you can't break out of a program without resetting the controller!

For this reason the default startup procedure sets ErrorAction to zero (0) if the Program Mode switch is on, i.e. you are developing code. It is usually best to leave the default startup procedure as it is.

RunMode

The **RunMode** message returns the 'run mode' state.

There are two versions of the message.

This version below returns the *soft* run mode state - this returns True if the Program Mode switch is set to Off, or if you used the **Run** message.

```
system.RunMode
```

This version returns the *hard* run mode state - it returns True only if the Program Mode switch is set to Off:

```
system.RunMode(1)
```

Debug

This covers a ragbag set of functions that may help when debugging the Venom system. There are very few things here that the average Venom application writer needs to know.

If you just type Debug, it will list out its capabilities. These are liable to change.

Free

This returns the amount of general-purpose heap memory left in the controller.

```
-->Print system.Free
 99466-->
```

It will also report on other areas of the controller's memory, as shown in the table:

`system.Free(0)`	Heap memory free
`system.Free(1)`	Largest free block in the heap

For example, to find the largest free block…

```
-->Print system.Free(1)
107982-->
```

Protect

Protect message allows you to take the code you have developed in RAM and copy it to the Protected Application Area (in Flash memory), where it is much safer from accidental erasure. Usually you won't need to do this until you have finalised your application code.

`Protect(1)` will copy your application code from RAM into the Protected Application Area.

It's best to use `Protect(0)` first - to make sure there is no application in flash - before using `Protect(1)`.

Mostly, Protect is typed at the command line, e.g.

```
-->Protect(0)
-->Protect(1)
```

Protect also has other options:

`Protect (0)`	Erases the Protected Application Area
`Protect (1)`	Copies the application from RAM into the Protected Application Area
`Protect (2, ...)`	May be used to create binary distribution files (.vex and/or .vos). *This has been superseded by Protect(4) for most purposes.*
`Protect (3)`	Looks for binary application and/or operating system files (.vex, .vos files) in the root directory of the Flash Filing System and uses it/them to update the VM2's firmware.
`Protect (4, ...)`	Create a Venom Firmware Update (.vfu) file, which combines your application and and the Venom OS into one distributable file. This may be used to program your units in production, or for remote firmware update. See `OperatingSystem.Protect` in the *Venom2 Help File* for more information.

Run

This will cause the controller to reset *as if* it were in Run Mode. This is useful when you are testing your application during development. You can leave the controller in program mode, and just type Run to exercise your application as if it were powering up in Run Mode.

Shortcuts

Note that the function key **F10**, and the Run icon will send 'Run' to the terminal - these are short cuts to achieve the same thing - running your application.

Reset

This immediately resets the controller. The controller will start in either program mode or run mode depending on the program mode switch. This reset is just the same as a power-on reset for the controller. However, other parts of the hardware system may not be reset fully if they rely on power-on to reset them.

Speed

You can control the clock speed on the VM2 using the Speed message. You can set the speed in increments of 8MHz from 16 to 72MHz. You might want to do it to make your controller use less power.

```
system.Speed := 16
```

When Speed is changed you may have to reset the speeds of other objects that had already been defined, like the serial ports or I2CBus, as their speeds were defined relative to the original VM2 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
```

PRINT

When you print the system message, a listing of useful system parameters is given.

```
-->Print system
Symbol table 48 bytes
7 Global variables
99478 Heap bytes free
-->
```

The format and content of this will change from time to time.

Creating new classes

Some programs are much easier to write and maintain if you can group together items of data that are logically connected into a single entity.

For example, you might want to hold data about a set of people, say their names, ages and heights.

One way of doing this is to create a variable for each attribute of each person. E.g.

```
Person_A_name   := "Fred Jones"
Person_A_age    := 35
Person_A_height:= 1.76
Person_B_name   := "Jim Smith"
Person_B_age    := 32
Person_B_height:= 1.78
```

However this very quickly gets out of hand if you want to add more attributes, or add more people.

Another way of doing it is to create three separate *arrays* of data:

```
Make names Array(String, 100)
Make ages Array(Int 8, 100)
Make heights Array(Float, 100)
```

This is much better than using variables because you can add new people easily, but it is still rather cumbersome:

- Every time you want to add a piece of data to the description of a person you have to add an additional array, and the code to manipulate it
- Accessing the set of data for any particular person becomes unwieldy as each item is an array expression.
- Every person has to be represented within a system of parallel arrays - you can't easily represent isolated individuals.

A better way of solving this problem is to encapsulate the data for each person within a single entity (an object), and then manipulate these objects - grouping them together into lists, or passing them around individually. Even better, you can also encapsulate the code that manipulates the data within the object.

Venom allows you to do all of this by defining your own new classes (or types) of object.

To solve the problem described above, we would define a new class of object called *Person*, and then create a new object of type *Person* for each real person we wanted to represent in our program.

Once you get used to using Classes you may find that you want to use them even in situations where you don't need more than one instance of the Class, because the *encapsulation of code and data makes your program much easier to write and maintain.*

Defining a Class

Each class has a list of data elements called *members*. Each of these members has a *name* and a *data type*. Data types might be *integer* or *floating point*, or other types.

For example a simple class definition might be

```
Class Person
  Age Int
  Height Float
End
```

There are many other member data types possible - including different sizes of integer, String, Array, Buffer, or 'Any' - we will look at these later.

Creating objects

User-defined objects are created similarly to other objects, so we can create a new object of type Person like this:

```
Make p Person
```

or like this:

```
p := New Person
q := New Person
```

Each object of type Person we create is called an *instance* of the class Person.

Accessing class members

We can access any member of the object by sending the object a message. Here we send the object `p` the message `Age`, to read the value of Age:

```
a := p.Age
```

or

```
Print p.Age
```

There is a quick way to print all the members of a user-defined object for debugging purposes:

```
-->Print p
Person (at $64000c64)
  Age = 0
  Height = 0.
```

Note that the memory address of the object is printed after its type as this address uniquely identifies the object.

We can also write to the members of **p** explicitly:

```
p.Age := 21
p.Height := 1.85
```

Initial values of members

When a new object is created, each member of the newly created object is set to the *default initial value* for its data type, which is zero (0 or 0.0) for all numeric types.

```
-->p := New Person
-->Print p
Person: (at $64000c64)
  Age = 0
  Height = 0.
```

Member types

Members of a class can have types such as Integer, Floating point number, **String**, **Array**, **Buffer** - all of the built-in Venom object types - or **Any** (*Any* means the member can hold a venom value of *any* datatype - e.g. any kind of object, number, string, etc.).

Note that when a member is an *object* of some kind, the member only holds a *reference* to the object, just as when you create an object with **Make** the global variable only holds a reference to the object created.

For example we might add a member called Name to the Person class. Name has the type *String*.

```
Class Person
  Age Int
  Height Float
  Name String
End
```

And we could create a new Person object and initialise **Name** with a string *constant* like this:

```
p := New Person
p.Name := "James"
```

Or we could create a new Person and initialise **Name** with a String *object* like this:

```
p := New Person
p.Name := New String(50)
```

Note that the String object will be empty until some text is put in it, like this:

```
Print To p.Name, "James"
```

or

```
p.Name.Put("James")
```

We can create members with types like **Array**, **Buffer**, **Class** or **Any**. For example

```
Class Person
  Age Int
  Height Float
  Name String
  TelephoneNumbers Array
  ListOfContacts Buffer
  Other Any
End
```

'New' Strings and Arrays

You can declare String or Array members of a Class as `New`, which means that a new String or Array object will be created and assigned to that member *automatically* when the object is created, and will be removed automatically when the object is removed. See the following example:

```
Class Person
  Age Int
  Height Float
  Name New String(50)
  List New Array(Int, 50)
End
```

Note: Only Arrays of numbers may be declared in this way - not arrays of strings or pointers.

Note:

1. You can't *overwrite* a New member (that is, you can't overwrite the *reference* to it); a New String or Array it is permanently associated with the object. However, you can empty the String or Array and write what data you like into it.
2. New String and Array members are removed automatically when your object dies.

Complete list of types

The complete list of member types you can specify are listed in the table:

Type specifier	**Type of data stored in the member**	**Default initial value**
`Int`	Integer (32-bits, signed)	0
`Int 32`	Integer (32-bits, signed)	0
`Int 16`	Integer (16-bits, unsigned)	0
`Int 8`	Integer (8-bits, unsigned)	0

Float	Floating point number (IEEE Single precision)	0.0
*Any built-in Venom object type**	Reference to an object	*Un-initialised*
Class	Reference to an object of a type defined by Class	*Un-initialised*
Any	Any Venom type (number, object, pointer, etc)	**Nil**
New String (capacity)	String	*Empty String*
New Array (type, length) (***type*** *must be numeric)*	Array	*Array filled with integer or floating point zeros.*

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

Removing objects from memory

If your program no longer needs an object you can remove it from the system (freeing up any memory it took) by sending it the message Die.

You may remember that you can use AutoDestruct to automatically send Die to objects that are held in Local variables when a procedure exits. You can also use **AutoDestruct** to automatically remove sub-objects held by an object when it dies. This can be very useful when you have a tree-like structure of objects that you need to remove. You can send Die to the 'trunk' of the tree, and all the 'branches' marked with **AutoDestruct** will also be removed. This feature is often used in conjunction with *Buffer of Any*, which will pass on the Die message to all the objects it contains.

```
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 by* the Class, then use **AutoDestruct**
- If the member object is *passed into* the Class from outside, then don't use **AutoDestruct**

- Note that **`New Strings`** and **`Arrays`** are automatically given the **`AutoDestruct`** attribute so you don't have to apply it explicitly.

For even more finely controlled behaviour, you can *override* the Class-default Die message with your own Die *method* - this is covered later.

Methods

So far the Person class we have created only has data - it doesn't have any procedures or *methods* that act on that data.

It is easy to add methods - they look just like normal procedures, but inside the Class definition.

```
Class Rectangle
  Width Int
  Height Int

  To Area
    Return Width * Height
  End

End

-->Make r Rectangle
-->r.Width := 10
-->r.Height := 20
-->Print r.Area, CR
    200
-->
```

Initialising objects

When you first create a user-defined object with New or Make the object is allocated some space in memory and each member of the object is reset to the *default value* for it's datatype.

Just like with other Venom objects, you can pass parameters to New or Make. But to process these parameters you have to give your class a special method called **`Initialise`**.

This **`Initialise`** method will be called by New or Make, and they will pass on their parameters so it can use them to initialise the new object.

For example we might do this:

```
Class Rectangle
  Width Int
  Height Int

  To Area
    Return Width * Height
  End

  To Initialise(Width, Height)
    If Width > 400
      Width := 400
    If Height > 200
       Height := 200
    This.Width := Width
    This.Height := Height
  End

End


--> Make r new Rectangle(100, 100)
```

Our Initialise method uses the parameters we pass to **`New`** to initialise the Rectangle's Width and Height. We also take the opportunity to check the values and limit them if they are too big.

Notice that we introduce a new Keyword, **`This`**. **`This`** is used here to tell the compiler that we are referring to the *class member*, not the *local variable* of the same name.

`(This` can also be used to refer to *the current instance* of the class - i.e. the object that the message was sent to).

Optional parameters

You can define an Initialise method to take optional parameters, and/or use variable *types* of parameters and check them using the **`TypeOf`** operator, to provide different ways to create a new object.

See the Venom2 Help file for more information about the Initialise method.

Indentation

Notice that the method code is indented (using space characters). This makes it visually clear that the method is inside the Class definition.

Member and method names

You can use any legal venom name for members and methods. Member and method names won't interfere with global, local or parameter names: a global variable, a local variable/

parameter, and a member/method could all have the same name but would still coexist happily.

When several things use the same name, you can to tell the compiler which of the three *name spaces* to use with the keywords **This** or **Global:**

```
Global.height := 5
This.height := 10
```

Accessing global variables

By default, you can't access global variables from inside a method of a Class. However you can *explicitly* specify a global variable by using the **Global** operator:

```
Print To Global.serial, "Hello", CR
```

Alternatively you can declare a *list* of globals that you want to use within the Class, at the start of each Class:

```
Class MyClass
  Global serial, clock, net
  ...
End
```

Optional parameters

Parameters to methods (or any Venom procedure) may be declared as *optional* by using **[]** to enclose the optional parameters.

See here for more information on optional parameters.

Active variables

A method can be an active variable. To turn a method into an active variable you have to use **:=** followed by a parameter name, after the method name and optional parameter list. For example, this can be used to implement the Element message for a two-dimensional array class **Array_2D**. Partial code for the Class's Element message, which is an active variable, is shown below; the keyword **Assignment** is used to detect when an active variable is being written to:

```
Class Array_2D
  ...
  To Element(x, y) := val
    If Assignment
    [
      ... := val
    ]
    Else
    [
      ...
      Return ...
    ]
  End
End
```

Using the array:

```
a2d := New Array_2D(Int, 20, 30)
a2d.(1, 2) := 3
y := a2d.(1, 2)
```

Inheritance

Often it is useful to re-use software that you have already developed. A property of Classes called *Inheritance* allows you to do that more easily.

Inheritance allows you to create a new, *derived*, class by inheriting the behaviour of an existing, *base*, class, and adding to or modifying that behaviour.

Consider the following class for implementing a button in a graphical user interface.

Looking at the code, you can see that this class is allowed to access the global variable called **LCD**; it has a rectangular extent defined by the members **Xpos**, **Ypos**, **Width** and **Height**; and it has a method to **Draw** itself, and a method to **Initialise** itself.

```
Class BasicButton
  Global LCD

  Xpos Int
  Ypos Int
  Width Int
  Height Int
  Label String

  To Draw
    LCD.textBox(Xpos, Ypos, Width, Height, 1)
    Print to LCD, Label
  End

  To Initialise(Label, x, y, w, h)
    Xpos := x
    Ypos := y
    Width := w
    Height := h
    This.Label := label
  End
End
```

We might want to create a new kind of button that can do all the things that the original button can do, but slightly differently. So we can define a new class that *inherits* the original class (note the colon operator that indicates inheritance). When a new class is declared, that inherits an existing class, all the members and methods of the base class are available in the derived class, so if we do nothing else, the new class's behaviour is exactly the same:

```
Class NewButton : BasicButton
End
```

To change its behaviour we can add a method; in this case we add a method with the same name as one in the base class:

```
Class NewButton : BasicButton
  Global lcd
  To Draw
    Lcd.textBox(Xpos, Ypos, Width, Height, $100)
    Print to Lcd, Label
  End
End
```

So the only behaviour we change here is how the button draws itself. Re-defining the Draw method *overrides* the original class's Draw method.

Overriding simply means declaring a method in the derived class which has the same name as a method in the base class, and which becomes the new default method of that name.

In this case, the new Draw method uses a different (constant) border style when drawing the button.

We can also add new members to a derived class:

```
Class NewButton2 : BasicButton
  Global lcd
  Border Int
  To Draw
    Lcd.textBox(Xpos, Ypos, Width, Height, Border)
    Print to Lcd, Label
  End
End
```

Here we've added a new member, **Border**, and redefined **Draw** so that this value is used to define a *variable* border style for the new class.

Typically, modifying the behaviour consists of any of the following:

1. Overriding existing methods
2. Adding new members
3. Adding new methods

We have covered points 1 & 2 above.

Adding new methods

To add a new method to a derived class you just have to declare it as normal. Make sure its name doesn't clash with any exisitng method or member name. For example, here we add the new method MoveTo:

```
Class NewButton2 : BasicButton
  Global lcd
  Border Int
  To Draw
    Lcd.textBox(Xpos, Ypos, Width, Height, Border)
    Print to Lcd, Label
  End
  To MoveTo(x,y)
    Xpos := x
    Ypos := y
  End
End
```

Where to 'put' behaviour

A lot of thought will often go into deciding which behaviour (methods and members) should be contained in which Class in a set of derived classes.

Overriding the Initialise method

Often the Initialise method is overridden in a derived class, especially if the derived class has new members which will need initialising.

One way to define a new Initialise is to copy the old one and add new initialisation code:

```
To Initialise(Label, x, y, w, h, b)
  Xpos := x
  Ypos := y
  Width := w
  Height := h
  This.Label := label
  Border := b
End
```

Another way is to call the Initialise method in the base class, and then initialise members declared in the new class:

```
To Initialise(Label, x, y, w, h, b)
  Base.Initialise(Label, x,y,w,h)
  Border := b
End
```

Note the use of **`Base`** to specify which version of Initialise we are calling (not the one we are in, but the one in BasicButton). If we didn't use this then we would be indicating that Initialise should call itself recursively.

Overriding members

Members may be overridden in the same way as methods, though this is much less common. When there is an overridden member, the class has two (or more) values associated with that member name. You can use **`Base`**, to access the base value(s).

Listing classes

The command **`List Class`** will list out all the classes you have defined, in an inheritance tree, allowing you to see how your class inheritances are organised.

Inheriting Venom types

It is not possible for a user-defined class to inherit a Venom type, such as **`String`**, **`Buffer`** or **`Array`**. However it is possible to emulate this using message redirection.

Accessibility

It is often useful to 'hide' a lot of the data and code used in a class so that they are not visible from outside the class. This allows a class to present a very tightly controlled 'interface' to the rest of the system, which makes the overall system much easier to debug and maintain.

This hiding of data and code is achieved by using the keywords **`Private`** and **`Protected`** in member and method definitions.

For example, in the class **`BasicButton`** we looked at before, we might decide to use the **`Private`** keyword make the **`Label`** member invisible outside the class:

```
Class BasicButton
  Global LCD

  Xpos Int
  Ypos Int
  Width Int
  Height Int
  Private Label String

  To Draw
    LCD.textBox(Xpos, Ypos, Width, Height, 1)
    Print to LCD, Label
  End

  To Initialise(Label, x, y, w, h)
    Xpos := x
    Ypos := y
    Width := w
    Height := h
    This.Label := label
  End
End
```

However, this will mean that classes inheriting **`BasicButton`** would not be able to access the **`Label`** member. If we want such derived classes to be able to access **`Label`** we can use **`Protected`** instead:

```
Class BasicButton
  Global LCD

  Xpos Int
  Ypos Int
  Width Int
  Height Int
  Protected Label String

  To Draw
    LCD.textBox(Xpos, Ypos, Width, Height, 1)
    Print to LCD, Label
  End

  To Initialise(Label, x, y, w, h)
    Xpos := x
    Ypos := y
    Width := w
    Height := h
    This.Label := label
  End
End
```

There is another keyword, **Public**, that is not often used because it is the default for all members and methods in Venom2.

Summary

Here is a summary of the properties of **Private**, **Public** and **Protected:**

- **Private**: the member or method is only visible in the class where it was defined.
- **Protected**: the member or method is only visible in the class where it was defined, and in derived classes.
- **Public**: the member or method is visible anywhere - you can send messages to an object to access this member or method. This is the default.

Class-default messages

In Venom, all user-defined objects have a common set of 'Class-default' messages, that they recognise. The *Die* message is one of these - the Class-default Die message removes the object from memory, and also passes on Die to any 'sub-objects' that have the **AutoDestruct** attribute).

Class-default messages may be overridden. For example you might override the Die message because you want the object to do some special tidying up before it dies. To illustrate this we might write the Person class like this:

```
Class Person

  YearOfBirth Int
  Height Float
  Name String

  To Die
    Global.global_list.Remove(This)
    Base.Die
  End

End
```

Notice that the Die method first removes the person from some global list (we won't go into the details of this) and then calls the base Die message to actually remove itself from memory. In this case there is no explicit base class, so the Class-default Die message is called.

Calling Class-default messages explicitly

Sometimes you may wish to call a class-default message explicitly, rather than as an implicit base class. For example, to find the class name of a user-defined object you can do this:

```
Print myObject.[Class]Name
```

From within a user-defined class you can also use these variants:

```
Print This.[Class]Name
Print Class.Name
```

Full list of Class-default messages

The full list of Class-default messages that are understood by all user-defined Classes is given in the table below.

All of these messages may be called explicitly, and some may also be called automatically by the system, as indicated in the table.

'Class-default' message	Function	Called internally...
`Address`	Return the memory address of the object's data block. This is intended only for debugging purposes.	
`Die`	Remove the object; also send Die	By `AutoDestruct`

	to all sub-objects that have the **AutoDestruct** attribute.	
Name	Return the object's class name.	
Print	Prints a user-defined object to the current output stream by attempting to print the value of each member. Sending this message explicitly is only meaningful inside a method called **Print**. If parameters are supplied then these are used for formatting in the same ways as the colon formatting specifiers in a print list.	By **Print <object>**
PrintF	Sending a PrintF message is one way to send text to an object. For this to work you must define an **AceptPrintJob** method in your class. See here for more information.	
Length	Returns the number of bytes needed to hold a *binary record* representation of the object.	

Special methods

There are some special methods that may be defined within a Class. The three we deal with here are used to print objects, and print *to* objects.

Printing objects

If you want to be able to print your object, as in **Print p**, then you have to define a method called **Print**. For example, this method might be defined as part of the Person class:

```
To Print
  Print Name, " is ", Height, "m tall"
End
```

The Person's **`Print`** method will be called if you make a Person object and then **`Print`** it:

```
-->Make p Person(1.78,"Ruth")
-->Print p, CR
Ruth is 1.78m tall
-->
```

If you define your Print method to take optional parameters, then the parameters will take the values of any 'colon' print formatting values used in Print. For example:

```
To Print([frmt])
  Select Case frmt
    Case 0
      Print Name
    Case Else
      Print Name, " is ", Height:1:1, "m tall"
End
```

```
-->Make p Person(1970,1.78,"Ruth")
-->Print p, CR
Ruth
-->Print p:1, CR
Ruth is 1.7m tall
```

Printing *to* objects, or sending PrintF

If you want to print *to* your object, or send a **`PrintF`** message to it, you must define a method called **`AcceptPrintJob`**.

You should *not* define a method called **`PrintF`**, as this will not work. For more information on sending print to your classes, look up *Print To Class* in the Venom2 Help File.

Classes as Records

One important use for user-defined classes is to define data templates for *records* in Files, EEPROMs or other storage media.

A record is a group of data items in a file (or other storage medium) where each data item within the record has a distinct meaning, and may have a different data type and a different size.

For example a file of records might contain the details of many different people, for example their age, name and height. Using a Person object to write, and later read back, the data one whole record at a time makes the process a lot easier than writing, and later reading, each piece of data individually.

As another example, an object stored as a single record in a file or EEPROM might be used as a convenient way to manage an application's non-volatile settings. Multiple records might be used to manage different sets of settings.

Put and Get

If you **Put** an object to a File or SafeData object, then the object will be written to the file in a defined format, so that when you **Get** an object of the same type from the file, an exact copy of the original object is re-created.

This example uses a file, but similar code will work for a SafeData object:

```
Define a record template:
```

```
Class Person
  Age Int
  Name New String(100)
  Height Float
End
```

```
Write a record to a file:
```

```
p := New Person(50,"Albert",1.7)
file.Put(p)
```

```
Read the data into a different object of the same class:
```

```
q := New Person
file.Reset ;'Rewind' the file.
file.Get(q)

Print q
Person:
  Age = 50
  Name = "Albert"
  Height = 1.7
```

Note: you may have to remove the objects p and q after you have used them; this is not shown here.

Permissible record member types

In general a class used for records should only contain members that are of type Int, Float, Array of Int or Float and String. If you include other member types they won't be stored in the record.

You may find it is easier to read records with classes that use New Strings and Arrays.

Data formats

Normally the data is written in a binary format. This is the best format to use when writing to small EEPROM devices.

However it is often useful to use a 'human-readable' format in files. Classes support a 'CSV' format (Comma-Separated-Values) and an INI file format. See below for more details of these.

Record length

Any object that is to be used as a record may be sent the Class-default message `Length`, which returns the number of bytes required to store the record in binary format. Length is a fixed value for all objects of a given Class so long as there are no String or non-New Array members in the Class. However if there are such variable-length data in the Class then `Length` is variable.

Reading strings

When you Get a binary record from storage, and the record contains a String, the String object in the template is first emptied before being filled from the string held in storage. If the data is too long for the String object (i.e. the null termination is not seen before the String object is full) then a runtime error is issued.

Record classes must be consistent

In order to be able to write and then read back data consistently, the template classes used for writing and reading back should be consistent with each other; ideally they will be the same class.

Different record classes in one file

You don't have to just use one type of record in a file. You can use any number of different types so long as you can predict which type to Get before you get it.

CSV Format

Writing in CSV format

To write records to a file in CSV format you have to print the 'template' object to the file like this:

```
Print To file, p:",", CR
```

The formatting expression `:","` specifies that the object is to be printed in CSV, using comma characters as the delimiter. The `CR` puts each CSV record on it's own line in the file.

You could equally use this if you want to use `#` as the delimiter:

```
Print To file, p:"#", CR
```

Reading CSV format

To read back data in CSV format you should use `Get`, but with an extra String parameter - which specifies that CSV format is to be used and supplies the actual delimiter too. For example:

```
file.Get(q, ",")
```

(Note: the first character in the string is taken to be the delimiter; the string can be any length but extra characters are ignored.)

INI file format

You can also read and write user-defined Class objects in the popular and human-readable 'INI file' format - for example:

```
[Person1]
Name="Jim"
Age=42
[Person2]
Name="Fred"
Age=56
```

When you print an object with the format descriptor **"INI"** (case sensitive!) then the object will print all it's members in INI file format:

```
Class Person
  Name New String(50)
  Age Int 8
End

p := New Person("Fred", 56)
Print to myfile, p:"INI"
```

prints to the object myfile:

```
Name="Fred"
Age=56
```

You have to add the section headers yourself, by printing them, e.g.:

```
Print to myfile, "[Person1],cr
```

To read an object back from a file you have to use Get, for example like this:

```
section := New String(50)
p := New Person("[no name]", 0)

myfile.Get(section) ; read the section header
n := myfile.Get(p,"INI") ; read the object data
```

Each member of the Person object p will be filled from the values found in the file.

Member names are not case sensitive.

Member values will be read until the end of the file, or a line beginning with **[**, is seen (the start of the next section header). The file is 'rewound' so that the **[** is the next character to be read from the file.

If a member value occurs more than once then it will be over-written with the last value seen.

Arrays are listed in comma-separated-value format, with a \ to indicate continuation on the next line.

Strings are always in double quotes and no escape characters are supported currently.

Lines starting with `;` or `#` are treated as comments and ignored.

`Get` will return the number of member values it read.

Data errors

If there are errors in the INI file these will normally be ignored by Get.

However, 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
```

Advanced topics

Sending messages to a derived class

By default, the compiler will resolve accesses to members or methods of the current class immediately (i.e. at compile time). However, you can force the compiler to delay resolving the access until run time by using the keyword `Derived`:

```
Derived.Message
```

This allows for a possible override of the message declared in a derived class.

For example

```
Class BaseClass

  To Method
    ...
  End

  To TestMethod
    Method ; Call the method in the *current* class.
    Derived.Method ; Call the method in the *derived* class.
  End

End

Class DerivedClass : BaseClass

  To Method
    ...
  End

End
```

Calling an overridden base method

There are some situations where you want to call the *base* version of a method that has been overridden. You can use the keyword **Base** for this:

```
Class myClass : baseclass
  To method
    ...
    Base.method
    ...
  End
End
```

Indirect message send

You can take a reference to a message (or member or method), that may be used to send a message indirectly later. Message references are actually 16-bit integers in Venom2

To take a message reference, use `@` followed by a dot and the message name. For example:

```
msgref := @.MyCallBackMethod
```

To send the message you have to use this syntax:

```
  <object>.!(<message ref>)(parameters)
```

For example

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

A larger example of this being used is listed below.

```
Class WindowClass
  Name String
  myButton Class
  To Action1(p1,p2)
    PrintF("We are in %s.action1, params: %i %i\n", Name, p1,
p2)
    Return "This is the Action 1 return value"
  End
  To Initialise(name)
    This.Name := name
    This.myButton := New ButtonClass("Button1", This, @.Action1
)
  End
End

Class ButtonClass
  Name String
  CB_obj Class ; Call-back object
  CB_msg Int ; Call-back message ref
  To OnClick
    Return
      CB_obj.!(CB_msg)(1,2) ; This is the call-back
  End
  To Initialise(name, CB_obj ,CB_msg)
    This.Name := name

    ;Set up the call-back 'pointer':
    This.CB_msg := CB_msg
    This.CB_obj := CB_obj
  End
End

To main
  w := New WindowClass("Window1")
  Print w.myButton.OnClick, CR
End
```

On running this program the result is

```
We are in Window1.action1, params: 1 2
This is the Action 1 return value
-->
```

Method prototypes/recursive methods

If you need a method in a class to be able to call itself - ie. recursion, you have to 'prototype' the method before it is seen or else the compiler will complain. For example, here the compiler will complain that it can't call Method because it has not been defined:

```
Class a
  To Method(n)
    If n
      Method(n-1)
  End
End
```

To get around this you can 'prototype' the method by defining an empty method of the same name first:

```
Class a
  To Method End ; Prototype

  To Method(n)
    If n
      Method(n-1)
  End
End
```

Message redirection

It is possible to arrange for a Class to *appear* to inherit a Venom pre-defined type, such as String, Buffer, Array or any other type (these base types can't be inherited directly in the normal manner as they are too different to user-defined Classes internally).

This can be done by *message redirection*, where a defined set of messages to a Class are passed on to a particular member (or members) of the Class. The messages to be redirected are listed, with dots, after the member declaration, as in the example below.

```
Class XYString
  XPos Int
  YPos Int
  str String
    .Put
    .Get
    .Empty
    .Print
    .AcceptPrintJob
    .Free

  To Initialise(x,y,size)
    XPos := x
    YPos := y
    str := New String(size)
  End
End

-->xys := New XYString(10,20,50)
-->print to xys, "Hello World"
-->print xys, CR
Hello Word
-->
```

It is possible to have more than one member set up to have messages redirected to it, but the sets of redirected messages should not overlap.

The type of the target member for redirected messages can be **Any**, so you can put any Venom object in it.

Objects that run in their own tasks

It is sometimes useful to have objects where each instance runs in a separate task. This example shows how this can be done.

```
Class WebServer
  id Int ; An identifier for this object

  To Thread
    ; dummy code:
    Print "Webserver: ", id,CR
    Await False ; wait here forever
  End

  To initialise(id)
    This.id := id
    Start Thread ; Run each instance in it's own task.
  End
End

To main
  Repeat 4
    New WebServer(Index0)
End
```

Inheritance Testing

Sometimes it is useful to check that an object that has been passed as a parameter, or fetched from a Buffer or Class member, is the correct type for the operation you are about to perform on it.

One way to do this is to check that the object either is, or is derived from a given base Class. The **`Is`** operator is used for this purpose.

```
If x Is MyClass
[
  x.Message
]
```

Interface testing

Another test you can perform is whether a user-defined object has a member or a method with a given name - using the Has operator:

```
If x Has MethodName
[
  x.MethodName
]
```

Note that **`Has`** won't work with pre-defined Venom types, such as **`Digital`**, **`String`**, etc.

Handling errors during Initialise

Sometimes you may want to handle errors that occur during a Class's Initialise method. Usually you should arrange your code so that Initialise doesn't generate errors - for example by checking parameters for valid ranges before you make a new object, or by converting out of range parameter values to valid values within the Initialise method.

However if you really do need to handle errors in Initialise you may encounter a problem in Venom: the Initialise method is called after the block for the new object has been allocated on the heap. If the error is trapped using Try and Catch outside of Initialise then a garbage block will be left on the heap. Here is one way to get around this problem:

```
To Initialise
  Autodestruct
  Local ref := This ; Autodestruct This on errors

  ; <code that may cause an error>

  ref := 0 ; Don't Autodestruct This on normal return
End
```

The End

This is the end of the Venom2 *Object* Tutorial - you have been introduced to some of the more commonly used objects that are built into the Venom language.

You may wish to read about the large number of other objects available, all detailed in the Venom2 Help File.

You are now ready to start writing your own application for VM2. There is a checklist for how to plan and complete your application in Appendix A: Development Checklist.

Appendices

A: 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 application board have the hardware interfaces that you require. See the datasheet for the controller. Often customers will buy the controller from Micro-Robotics Ltd, and make the application board themselves. However, Micro-Robotics can design and manufacture custom application boards.
2. Get familiar with the Venom language and basic Object Types by reading this manual and by trying out your ideas on your development system.
3. Also use the Venom2 Help File. This will be required for all serious applications.
4. Using prototype hardware, write key sections of your application program 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 using the real hardware.
7. Go through Appendix D 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 Flash. See the system message Protect.
10. Go into production with the application hardware and the application software.
    (If possible, fix the version of Venom2 Operating system/Language that you use with your application to avoid any compatibility issues)
    Note: You can load new versions of the Operating system and/or your finished application code into an 'empty' VM2 using a USB Connection. This is much quicker than other methods. See *Production Programming* in the Venom2 Help File.

B: How Do I ... ?

This ‘FAQ.’ section deals with how to achieve solutions to commonly encountered problems using the Venom2 language and object types.

Store Non-Volatile Data

The following objects can handle non-volatile data:

- FileSystem: stores data and text files, RAM Filing System, the Flash filing System or external memory cards (SD, SDHC). Files can hold very large amounts of data.

Manipulate Text

Use the text Buffer and String objects to manipulate text.

- Print To the buffer or String to append text
- Print all of the buffer or String, or any sub-section of it to extract text
- Use Put to append or insert text
- Use Find to search for occurrences of a sub-string.
- Use Element to access any character within the buffer or String
- Use the String or Buffer Value message to convert text to a number

See also Array and string constants.

Enter Numbers on a Numeric Keypad

Use the NumberReader object in conjunction with the Keypad object.

Deal with Calendar Dates

Use the DateTime object to

- Convert calendar dates to and from a linear seconds value
- Deal with leap years
- Find which day of the week it is on any date
- Print the date and time in a variety of formats
- Facilitate 'digital watch' style date/time entry

Create User Interfaces

Use the AlphaLCD or GraphicsLCD objects for displaying information and a Keypad and/or Touchscreen object for entering data, or for navigating menus.

Time events

- Use Wait, Every and Timer to have your application do things at the right time
- Use Stopwatch to time external events
- Use RealTimeClock to relate the controllers actions to real times and dates
- Use PulseWidthIn, PulseCounter, PulseWidthOut to measure and generate pulses (See the Venom2 Help File)

Talk to serial devices

- Use SerialPort for RS232 and RS485 communications
- Use I2CBus for I2C Bus devices
- Use the SPI object for devices on the SPI or Microwire buses
- Use OneWire for Dallas 1-Wire bus and iButtons

Generate Pulses

- Use the PulseWidthOut object.

Measure Pulses

- Use the PulseCounter object to count pulses
- Use the Shaft object to count quadrature shaft encoder edges
- Use PulseWidthIn to measure the pulse width

Measure Temperature

- Use a thermocouple amplifier to generate a 0-3.3 Volt signal and read this using one of the on-board 12-bit analogue inputs on the VM2.
- Use a precision thermistor bead and a resistor in a potential divider, feed the voltage into the on-board 12-bit analogue inputs (0.2°C, or better, accuracy is possible). We publish a linearisation function to convert ADC readings to temperature.
- Read a thermocouple directly using an external 16-or-more-bit external ADC on the I2C Bus or SPI Bus. A thermistor can be used to measure the cold junction

temperature for compensation.

Sleep with very low Power

It is possible to put the controller into 'Stop Mode' - where it will consume around 55 microamps. You can wake it from Stop Mode either a number of seconds into the future, or using a digital input channel. See **`RealTimeClock.Timeout`** in the Venom2 Help file.

Use Files

- Use the FileSystem object to create files in RAM.

C: 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 0.7µS to 1.5µS on the VM2.

A bit of code like a := a + 1 will take ~4.5 µS.

A simple message sent to an object takes something like 6uS.

Measuring Execution Times

The following code allows you to measure the execution time of 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
  Print (t * 1000 / n - c):10:4, " microseconds", 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 = 100000; 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. E.g.

```
measure_time(100000,2.42)
```

D: Robust Applications

This section details steps you should take to make your Venom application least likely to fail in the field.

Protecting the 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 Flash memory - which can hold it safe against loss due to processor crashes or battery failure.

See the system message Protect.

Protecting Against Errors

Runtime errors can stop a program from running correctly and cause it to halt forever. This is usually unacceptable for an embedded control application in the field.

To prevent errors from causing your program to halt, use Try/Catch 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 Try/Catch.

The default startup procedure defines a 'safe' setting for ErrorAction: it is set to restart on errors if the Program Mode switch is set to 'Run'.

Serial Break

Most applications should turn off the Ctrl-C Break function, as this could potentially halt an application. Ctrl-C Break is treated as a runtime error, so if ErrorAction is set, the Venom application will be restarted. To turn off Ctrl-C Break, use

```
Serial.Escape := False
```

Note that this will also turn off Ctrl-T task listing.

If you want to avoid your application halting on receiving a Ctrl-C character (perhaps due to noise) but want to keep Ctrl-T enabled, you can leave serial.Escape set to True, but make sure ErrorAction is set to 1 to at least ensure your application restarts on receiving a Ctrl-C character.

Memory

Memory leaks

A memory leak is where an incorrectly written program uses up the RAM memory in your controller.

It's hard to spot memory leaks through normal testing as, if the leak is a slow one, you might not notice it for a long time. However, in the field, your application may fail after a period because it eventually runs out of memory.

If you have ErrorAction set correctly (e.g. leave the default startup procedure unchanged) then your application will restart, which may lead to a good recovery.

However it's much better to check for memory leaks and fix them, before the application is released.

A typical memory leak looks like this:

```
To proc
  Make b Buffer(Int 8)
  ... ; further operations using b
  ...
End
```

If proc is called repeatedly, and the Buffer, b, is not killed when proc exits, then you have a memory leak: each time another buffer is made, memory is taken that is never given back to the system.

There are two different ways to avoid the problem in the example above:

1. Make the Buffer once only, say in your init procedure, and then use it wherever it is needed.
2. Create a temporary buffer object in the procedure and remove it after you have used it. See here for more details.

Garbage scanner

You can detect if you have a memory leak by using the 'Garbage scanner' built into Venom. This will detect any memory that has leaked from the system, and also tell you what the leaked blocks where, if possible. This can help you find the leak and solve it.

This a typical output showing lost buffers. The unknown blocks are also part of the buffers but the system wasn't able to identify them:

```
 36 bytes at $6400096C: Buffer?
264 bytes at $64000998: <Unknown>
 36 bytes at $64000AA8: Buffer?
264 bytes at $64000AD4: <Unknown>
```

To find out more about the Garbage Scanner and how to use it see the Venom2 Help File.

Code image validation

It is possible (though very unlikely) that when you download the Venom2 Language and Operating System into your VM2, that the code gets corrupted.

When we release a new version of Venom2 we include a system checksum to validate the code image. This is available using `OperatingSystem.Checksum` and `.Valid` messages.

To make sure your Venom2 operating system has been downloaded correctly you might include a line like this in your application code's `init` procedure:

```
To init
  If system.Valid
  [
    ... go to normal operation...
  ]
  Else
  [
    Print To display, "Corrupt OS!"
    ;... and any other actions you might need to take.
  ]
End
```

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 microcontroller.

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. However, the highest security applications may require extra watchdogs at the application code level. You can write these using Venom code.

SUMMARY

- Check your code for memory leaks.
- Always 'ROM' your application code.

- Something like the following lines should appear near the start of any Venom2 application released into the field. Some of these will have been taken care of by the default startup procedure – List startup to find out.

```
system.ErrorAction := 1       ;Restart on errors (taken care of
in the default startup procedure)
...
Serial.Escape := False ;Disable CTRL-C Break
```

E: 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.

Note: although character 13 is called CR in ASCII, Venom2 uses the character 10 for the Carriage Return character internally, to maintain consistency with as many other systems as possible.

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		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	~

F: 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 optimises 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 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 that involve only constants. In order to give the compiler the best chance of folding an expression, enclose the operations in parentheses:

```
5 * a * 4   ;will not be folded (the compiler's not that
clever!)
5 * 4 * a   ;might be…
a * 5 * 4   ;might be…
a * (5 * 4) ;definitely will be.
```

Power saving

The Venom operating system automatically uses the HALT instruction on the host processor, if there is a suitable one available. 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 HALT 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.NotAsserted []
```

However if you don't mind being up to 10mS late in the detection of the input you can save power by using something like

```
While dig.NotAsserted [Wait 10]
```

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.

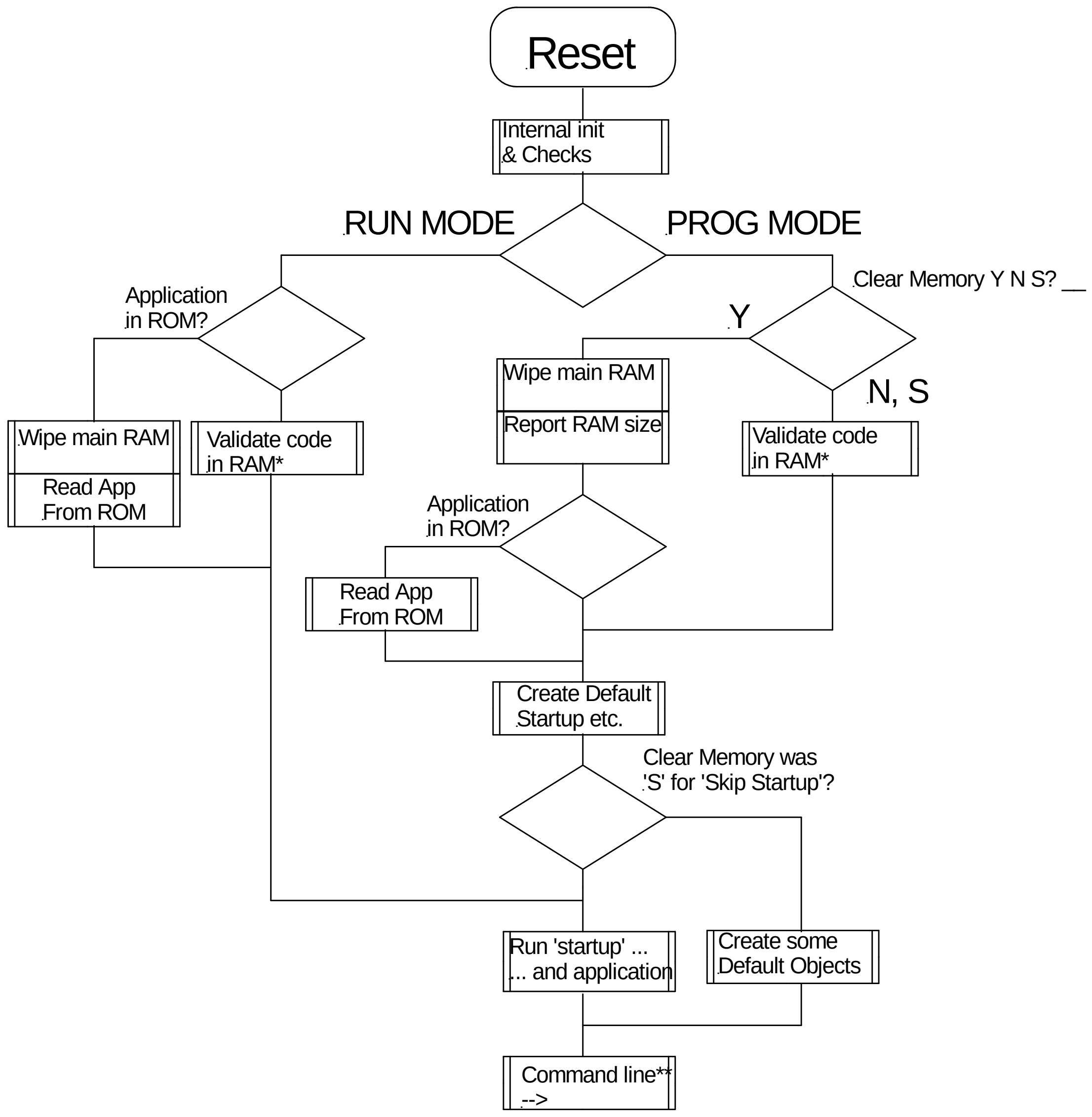
The operating system message **`system.Low`** will set all uninitialised IO pins to the state 'input pulled low' to make sure every uninitialised IO pin is pulled to a defined state.

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).

G: Startup Sequence

The diagram shows what happens when Venom2 starts.



*'Validate code in RAM' means check that the controller's heap-memory, global variables etc. contain valid data. If not, then reset them.

**All routes through the flow diagram end up at the command line, unless startup never returns – i.e. the application code loops forever. Application programs in general should never terminate to the command line.

Not using VenomIDE

This is the getting started guide for users who can't use the VenomIDE development tools for Windows, and shows you how to use a terminal emulator and text editor to program your Venom-based controller.

You might need to use this method if you have a MAC or UNIX computer, or if you are using a computer that doesn't have VenomIDE installed.

Unlike most application development systems, this one runs on the target hardware in real time. This has many advantages when it comes to learning the language and debugging.

To start learning about the Venom language you need to be able 'talk' to the controller running it. This is normally done over an RS232 serial link using your own personal computer - your computer will need to run a *terminal emulator* program. Suitable terminal emulators are often available for free.

What you will need

In order to start learning about Venom, you will need a minimum of:

- A Venom-based controller, such as VM2
- You may need an application board for the controller
- A suitable power supply
- An RS232 lead to connect the controller to your computer, or to a USB-Serial converter on your computer
- A PC running terminal emulation software

Micro-Robotics supplies starter kits containing all you need, except the personal computer and the terminal software.

Connecting it all together

The exact details of connecting the controller to a personal computer are given in the Getting Started Guide for the particular controller configuration you have. The Getting Started Guide will take you as far as seeing the Venom startup message:

```
VM2 Control Computer running Venom2 at 72MHz
Version 2011 02 10
Copyright 2008-2011 Micro-Robotics Ltd.
Clear RAM?
```

As this is the first time you have used the system, type in a Y. This tells the controller to clear its memory.

The cursor will be positioned just after an arrow: `-->`. This arrow is called the 'prompt' and means Venom is waiting for your instructions.

Simple Commands

Try pressing Carriage Return a few times. You will notice that Venom replies with a prompt on a new line. This is a quick way of checking that Venom is talking to you.

Now try typing the following (press Carriage Return at the end of the line). You type the text after the `-->` prompt.

```
-->Print "hello"
hello-->
```

Venom responds to the command by printing the string you gave it back to your terminal window.

Now try the command below. Don't forget to type the dot between the two words.

If you make a mistake in your typing, then you can use the Delete or Backspace key to remove the characters you have entered.

```
-->led.On
-->
```

To see the effect of this command you will need to be able to see the LED on the controller. The LED on the board will light up. If you repeat the command using the word Off instead of On, the LED will be turned off.

Objects

An object is a part of the Venom language that will control and monitor a device in response to a fixed set of messages. In the example above, LED was the object responsible for controlling the LED device on the controller. On was the message sent to the led object. The dot (.) tells Venom that a message follows. Objects will be covered in much greater detail later. For now it is enough to know what it looks like when an object is being used.

Incidentally, you don't have to type commands in exactly as our examples – when accepting commands, Venom is case-insensitive.

The Command Line

The *command line* is the text that you type in at the `-->` prompt. The term will be used frequently throughout this manual.

Errors

If you made any mistakes in the examples above, Venom probably issued an error message. In case you haven't seen an error message yet, type in led.Onf. You will see:

```
-->led.Onf
       ^^^
Syntax Error: Expected message name.
Command line not executed.
-->
```

Venom issued a Syntax Error message, meaning it didn't understand the command. The offending line is listed together with a pointer to where Venom thinks the error is (the ^^^ characters), and the reason Venom didn't like it.

Syntax errors like the one above will show up when your code is downloading. There is another type of error that can occur – *runtime errors*. These will be dealt with later.

Simple Procedures

The commands shown above were very simple. To perform more complicated tasks, commands may be grouped together into *procedures*. Try the following line, taking care to include the dots and spaces.

```
-->To blip led.On Wait 1000 led.Off End
 Procedure defined
-->
```

The keywords To and End tell Venom that the commands in-between should be treated as a single command (or procedure) called blip. Incidentally, the Wait 1000 command tells Venom to do nothing for 1000 milliseconds.

Try issuing blip as a command:

```
-->blip
-->
```

The LED should turn on for one second then turn off again. The new prompt will only appear once the procedure has finished.

Blip could also be issued as a command from within a procedure. The following procedure 'calls' blip once, waits for a second and then calls blip again. Try entering it and then typing double.

```
-->To double blip Wait 1000 blip End
```

It is not necessary to enter procedures on a single line. The blip procedure could have been entered as below, or in any form where the spaces are replaced by carriage returns.

```
-->To blip
02>led.On
03>Wait 1000
04>led.Off
05>End
Procedure Defined
-->
```

You will notice that the prompt is different during entry of the procedure. This tells you that Venom will not act on the commands you type immediately, and also lists the line numbers of the procedure.

Listing Procedures

Listing back of procedures is not fully supported. If you type List blip you will get a short summary of the procedure, somewhat like this:

```
-->List blip
;To blip
;  No source list [36 bytes @$260532]
;End-->
```

Editing Procedures

Simple procedures may be typed in at the command line as shown above. When procedures get larger it is useful to be able to edit them.

This is best done with a text editor. Suitable text editors are usually available for free.

Type the code of the procedure into your favourite text editor, and make sure it's what you want it to look like. Then Cut-and-Paste the text into the window of your terminal emulator. This is equivalent to typing in the procedure, but much faster.

Most Windows® programs allow the use of the shortcut keys Ctrl-C and Ctrl-V for Cut-and-Paste.

Any syntax errors in the code will be indicated as the text downloads, and you can go to the editor to correct them.

PROGRAM command

If you want to download one or more whole files full of procedures then it helps if you put two special commands around all your code. Put them on the first and last lines in the file if you can. Note that you can fill in the name of you file if you like (or just use any name you like).

```
PROGRAM "your_code.txt" ; On the first line of your file.
```

(all your procedures)

```
PROGRAM End
```

Now you can download the whole file or files using Cut & Paste, or by other means.

Using `PROGRAM...PROGRAM End` makes the download easier to understand, and will make error reports more meaningful: the file and line number of the error will be shown.

Help

You can download the Help Files for the Venom language from our website - they are available in Windows Help format and PDF.

Venom2 also has a simple on-board help system. This allows you to interrogate the runtime system. It may not always have the information you are looking for, but it can be useful. Try this:

```
-->Help led
It is the OnBoardLED. Try PRINTing it for more info.
-->Help put
'Put' is a message name.
-->
```

The second example is a useful way to check that a word you want to use is not already reserved by Venom.

In Venom, printing something will often give you information about it. System is a predefined object that represents the Venom system. For example:

```
-->Print system
Symbol table 61 bytes
9 Global variables
108880 of 110594 bytes free in heap (biggest block 108490)
NV RAM area 0 bytes (0 unused)
```

SUMMARY

- It is possible to program a VM2 controller without using **VenomIDE** - just using a terminal emulator and a text editor.
- You can issue commands on the command line and enter simple procedures.
- You can download your program files using Cut and Paste from a text editor to a terminal emulator.

What next?

You should now go on to read the next chapter of Venom language tutorial, Repeating and Deciding.

Index