# Venom-SC

## A Tutorial Guide

Venom-SC
A Tutorial Guide
Written by Karl Lam

First Edition
>    First printed 2002 (Electronic)
>    Second Impression 2003
Document 44281 Version 2003-01-21

# Contents

# ABOUT THIS MANUAL

This tutorial manual will take you through learning the Venom-SC language in a step-by-step fashion. No previous knowledge of programming is assumed. However, if you do know C, Pascal or other high-level languages, then you may be able to skim-read the Venom Language Tutorial or use the Quick Reference card and the Venom-SC reference books.

Much of the usefulness of Venom is in its library of *Objects*, so you will need to read Part 2 to find out how and when to use them.

This manual is divided into several parts.

## Part 1

You are taken step by step through the Venom-SC programming language – which will be familiar to Pascal, C, or Visual Basic programmers.

More sophisticated concepts are introduced in the later parts of this section.

## Part 2

In the second part of this manual the most commonly used *objects* are described.

## The Appendices

Finally, a number of appendices contain information on very specific points.

**NOTE Although this manual reflects the most current information possible, you should read the Venom-SC Release Note for information that may not have been available prior to our documentation being finalised. In particular, Venom-SC is being continually improved. The Release Note will document the added features and improved functioning available in any particular version of Venom-SC over that described in this manual.**

**The Release Notes can be found online at www.microrobotics.co.uk**

# Part 1:
## Language Tutorial

# GETTING STARTED

This chapter introduces you to the Venom-SC[1] development system.

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.

This tutorial will try not to assume any particular hardware configuration, though as this is a 'real world' system that may not always be possible.

To start learning about Venom you need to be able 'talk' to 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[2].

Venom does not need any extra software to be installed on your PC, though we may develop PC tools in the future.

## What you will need

In order to start learning about Venom, you will need a minimum of:

- A Venom-based controller
- You may need an application board for the controller
- A suitable power supply
- An RS232 lead to connect the controller to your computer
- A PC running terminal emulation software

Micro-Robotics supplies evaluation 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:

---

[1] Venom-SC is a *semi-compiled* version of the Venom language. It runs much faster than the original, and has been improved in many ways, while keeping the simplicity and originality that made it popular. We will usually refer to it as 'Venom'.

[2] A terminal emulator is a program that reads characters you type in at your personal computer's keyboard, and sends them out of the serial port. It also prints the characters that come back onto your personal computer's screen. The free terminal emulator provided in Windows is called *HyperTerminal*[©]. The free version of *HyperTerminal*[©] is useful, but also has annoying habits.

```
VM-1 Control Computer running Venom-SC
Version 2003 04 03
Copyright 2000-2003 Micro-Robotics Ltd.
Clear memory: Y/N/S ?
```

As this is the first time you have used the system, type in a `Y`. This tells the controller to clear its memory. The controller checks its memory size:

```
RAM Cleared - 128K Bytes RAM found.
-->
```

The cursor will be positioned just after the `-->` 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). The bits in **boldface** 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 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 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 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[3].

## 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 always 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. Commands may be grouped together into procedures that perform more complicated functions. Try the following line, taking care to include the dots and spaces.

---

[3] Though Venom is not case-sensitive there is a convention for the capitalisation of Venom code: all variable names are lower case; all language keywords are UPPER CASE. *Messages* and *object types* are 'capitalised' (TheInitialLetterOfEachWordIsUpperCase). We will use this convention throughout.

```
-->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 currently supported[4]. If you type `LIST blip` you will get a short summary of the procedure, somewhat like this:

---

[4] We may implement listing back of Venom code, or we may improve the development tools to obviate the need for it.

```
-->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. Simple text editors are provided with your PC, though there are usually better ones available, sometimes for free[5].

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. If Ctrl-C/V don't work in HyperTerminal[©], then enable Windows Keys in its Properties.*

Any syntax errors in the code will be indicated as the text downloads, and you can go to the editor to correct them.

This process will be most efficient if you run the terminal (and the controller) at a high baud rate.

## Help

Venom-SC 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.

---

[5] Contact us for a list of suitable text editors.

In Venom, printing something will often give you information about it. System[6] is a predefined object that represents the Venom system.

```
-->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)
```

*A hyper-linked help system based on the Venom-SC manuals will be constructed in the future.*

## SUMMARY

- You have seen how to talk to Venom, issue commands, build simple procedures and edit them.

- The exact meaning of the commands has not been covered.

---

[6] System, like led, other objects and two dummy procedures, are pre-defined by Venom-SC when memory is cleared. This is discussed in *The Startup Procedure*.

# OUR WORKED EXAMPLE

We are going to use a worked example to illustrate some of the concepts in this tutorial, and to provide a skeleton application that you might be able to use as the basis of your own application. The Venom code for this application may be found on the CD or on our website **www.microrobotics.co.uk**

Our example is an oven controller with the following requirements:

- Target and actual temperatures shown on a display
- Target temperature adjusted via a user interface
- Oven may be turned on and off from a user interface
- Oven temperature must be ramped up and down slowly

The user interface chosen is a 2-line by 20 character alphanumeric LCD and a 4 by 4 matrix keypad as these are readily available items.

The keypad is placed under the LCD so that its top 4 keys become 'soft keys' – that is, their function is indicated by what's written on the bottom line of the LCD.
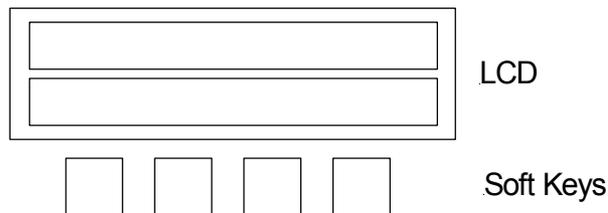


The temperature of the oven will be measured from a temperature sensor connected to an analogue input. The heating element of the oven will be controlled by a digital output connected to a relay. The relay must not 'chatter' – it must not switch more often than necessary.

There are many ways of using Venom to solve the elements of this problem, and we may show more than one to illustrate particular features of the language.

## A Template

You may be able to use the worked example as a template to base your application around.

# 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 below.

## 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 ad infinitum.

```
-->FOREVER PRINT "And again", CR
And again
And again
And again
```

… (and so on forever) …

## Escaping out of Loops: *Ctrl-C*

Whenever Venom is executing a command, it may be asked to stop by pressing Ctrl-C[7].  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)

---

[7] Ctrl-C means while holding down the *Ctrl* key, press *C*.

```
Run-time error 2: Escape via CTRL-C
at $2601CE in the command line (line~1?)
-->
```

*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.

## 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)

```
Run-time error 2: Escape via CTRL-C
at $2601CE in the command line (line~1?)
-->
```

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.

## Loop Count: INDEX; INDEX0

In all looping constructs, the keyword `INDEX` expresses the number of times the loop has executed. 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`, which starts at 0.

14

## 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 constructs are concerned, a block is just treated as a single command, so the following repeats the printing and 'toggles'[8] 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 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* code will usually make the structure clear to programmers.

---

[8] Toggle means change an Off to an On and vice-versa.

## Indentation

You will notice that all the above examples showed the whole of a looping or decision construct on one line.  This is because they are being typed in at the command line.  When such constructs are part of a procedure, it is useful to list them in an indented format:

```
TO dummy
  FOREVER
  [ REPEAT 5
    [ IF a < 30
        PRINT "Less"
      ELSE
      [ PRINT "More"
        a := a - 1
      ]
    ]
  ]
END
```

## Repeating Decisions: WHILE; DO; UNTIL

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
```

In addition to `WHILE` and `DO … WHILE`, Venom has the complementary constructs `UNTIL` and `DO … UNTIL` which will loop *until* the condition is met.

*The keywords* `INDEX` *and* `INDEX0` *are available in all loops.*

16

## Waiting

Often it is useful to wait for certain events. The `AWAIT` command may be used for this – it just waits[9] for a condition to be met before carrying on.

```
AWAIT led . 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 check up on external events, or other tasks in a multitasking application.*

Finally the `WAIT` command just waits[10] for a given number of milliseconds. For example,

```
WAIT 1000
```

just pauses execution for 1000 mS.

## Choosing actions: SELECT

The `SELECT CASE` construct allows one of a number of different actions to be taken depending on the value of a variable. *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 value of the integer it is given, and then executes only the code associated with that particular `CASE`. More than one case

---

[9] In fact `AWAIT` will swap tasks while waiting for the condition to become true so as not to hog processor power.

[10] Like `AWAIT`, `WAIT` will swap tasks while waiting.

value may be associated with a bit of code (as with `CASE 10,11`), and a default option may be specified with `CASE ELSE`.

## 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.

## 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` … can choose from among a number of actions.

- `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.

- `UNTIL` and `DO … UNTIL` repeat a command or block until 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.

18

# VARIABLES AND EXPRESSIONS

## Variables

A variable is a named memory location where values are kept. The name and location are created the first time a new name is seen by Venom. The value may be set using the `:=` symbol (spoken as 'becomes equal to').

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

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

A variable's value may be changed at any time to any other value using `:=` .

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 in your programs. This makes it much easier for you to write the program in the first place, and is a great help to you or anyone else who has to maintain your code later.

Consider the following two procedures – what is the first one doing? What is the second one doing?

```
TO a (b,c)                 TO cylinder_surface_area (diameter , height)
  LOCAL d                    LOCAL circ_ends
  LOCAL e                    LOCAL curved_surface
  LOCAL f := 3.14159         LOCAL pi := 3.14159
  LOCAL g := b / 2.0         LOCAL radius := diameter / 2.0
  d := g * g * f             circ_ends := radius * radius * pi
  e := b * f * c             curved_surface := diameter * pi * height
  RETURN d + e               RETURN circ_ends + curved_surface
END                        END
```

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, object type or message name. Some examples of both valid and invalid names are listed below:

| Valid names | Invalid names | |
| --- | --- | --- |
| water_temperature<br>output_control_2<br>a | WORD | (WORD is a keyword) |
| | var% | (% is not allowed) |
| | low byte | (spaces are not allowed) |

Venom is not case-sensitive, though there is a convention for the capitalisation of Venom code:  all variable names are lower case; all language keywords are upper case.  Messages and object types are 'capitalised' (the initial letter of each word is in upper case).

```
count - a variable name
WHILE - a language Keyword
Off - a message name
OnBoardLED  - an object type.
```

## 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:

Regions:

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

20

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 used in this manual so far have been whole numbers (or 'integers'). These numbers are adequate for many situations, 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'). `Counter` may be set to a floating-point value:

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

Floating-point numbers in Venom are calculated and stored to IEEE standard single precision: a number range of around $\pm1.0E\pm38$, 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.

Integer constants are formed from the decimal[11] digits 0123456789. They may have any value within the integer range -2,147,483,648 to 2,147,483,647.

Floating-point constants are also formed from the decimal digits. They 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 read as floating-point values:

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

There are also string constants – bits of text that keep the same value:

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

String constants always appear within double quotation marks.

---

[11] Venom also supports Hexadecimal and Binary integer constants – see Using Hexadecimal and Binary numbers on page 75.

## 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 120000
```

Actually, #DEFINE may be used to give any piece of program text a name. A named piece of text is called a 'macro'[12].

```
#DEFINE LEDON led.on
```

## 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:

---

[12] There are special rules for using macros. For now, all you need to know is that you should make sure you define the macro before you ask the compiler to use it. The compiler will tell you if you get this wrong.

```
-->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 in 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 always positive. If the following number is negative, then it is multiplied by –1:

```
-->PRINT ABS -23, CR
    23
-->
```

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[13].

```
-->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.

## 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.

---

[13] A radian is ~57.3˚. There are $2\pi$ radians in a complete circle.

```
 ( )
● AS INT  AS FLOAT
- ABS NOT ! SIN COS TAN ASIN ACOS ATAN SQRT EXP LOG ?
* / DIV MOD
+ -
> < >= <= = <>
AND OR EOR
```

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 brackets will often make an expression clearer even if they are not strictly necessary.

## 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 (minus one) or 0 (zero); -1 indicates that the relationship was *true*, and 0 indicates that it was *false*.  Venom has two keywords TRUE and FALSE which are the same as -1 and 0 respectively.

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 different, <= tests whether the first number is less than or equal to the second, and >= tests whether the first number

25

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 a not 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.

## 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 loses the fractional part of a number. There is no 'rounding up' to the nearest integer, and negative numbers round towards zero.

## 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' values). These are useful in conditions where it is necessary to make more than one comparison. These operators are `AND`, `OR`, `EOR` and `NOT`. In addition, the keywords `TRUE` and `FALSE` may be used to represent the numbers -1 and 0 respectively. The next four examples illustrate the use of the operators.

`AND` will give `TRUE` only if both of the values are true.

```
-->PRINT TRUE AND TRUE, TRUE AND FALSE, FALSE AND TRUE,
FALSE AND FALSE, CR
    -1     0     0     0
-->
```

OR gives TRUE if one or other or both of the values is true.

```
-->PRINT TRUE OR TRUE, TRUE OR FALSE, FALSE OR TRUE, FALSE
OR FALSE, CR
    -1    -1    -1     0
-->
```

EOR returns TRUE if one and only one of the values is true.

```
-->PRINT TRUE EOR TRUE, TRUE EOR FALSE, FALSE EOR TRUE,
FALSE EOR FALSE, CR
     0    -1    -1     0
-->
```

The NOT operator simply gives the opposite of the value after it.

```
-->PRINT NOT TRUE, NOT FALSE, CR
     0    -1
-->
```

Though TRUE and FALSE are used in the examples above, they may be replaced by any expression that gives a Boolean value i.e. one containing relational operators.  Since AND, OR and EOR have equal precedence, it is often necessary to use brackets to get the correct meaning:

```
-->PRINT 1 < 2 AND (3 < 4 OR 6 < 5), CR
     -1
-->
```

## Bug Alert

Watch out if you use AND, OR, EOR or NOT on numbers that are neither TRUE (-1) nor FALSE (0).  The operations are carried out in a bit-wise fashion, and so can give misleading results.  For example:

```
a := 1
b := 2
IF a PRINT "a is true"
IF b PRINT "b is true"
IF a AND b PRINT "a AND b is true"
```

Remember, IF will take anything non-zero as to mean 'true', but because AND does a bit-wise operation, the expression a AND b may well end up as zero.

See *Bit-wise Operators* on page 77 for more information.

## Another look at Decisions

The `IF` command introduced in the preceding chapter made decisions on the basis of 'conditions'. It can now be seen that a condition is simply an expression that gives a Boolean value – if the expression is true, the action is taken. For example, the following command will only print `"Yes"` if the temperature is less than 300 and/or the value of counter is between 100 and 200 (exclusive).

```
-->IF thermometer.Value<300 OR (counter>100 AND
counter<200) PRINT "Yes", CR
```

Note that all the decision-making constructs will take any number that is non-zero to mean `TRUE`. This can be useful at times, but beware of the pitfalls – see above.

## Another look at INDEX

It was shown before that `INDEX` and `INDEX0` could be used in a repeated command or block to represent the number of times that the command had executed. However, only one value of `INDEX` is available – if one loop is inside another, the `INDEX` value for the outer loop is not available. To solve this problem, the value of `INDEX` should be placed in a normal variable in the outside loop. This is shown in the example below, which prints a multiplication table.

```
REPEAT 3
[
  out_index := INDEX
  REPEAT 3
    PRINT INDEX * out_index
  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:

28

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

It is also possible to change an object into a number, and vice-versa[14]. 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[15] level Analogue (64)
-->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.

## Sets of Data

There are two major objects that provide storage for a set of data: `Array` and `Buffer`.

`Array` is for holding fixed amounts of data, whereas `Buffer` is for accumulating data. They also have other properties that are given in the table.

|  | Array | Buffer |
|---|---|---|
| Dynamic size | No | Yes |
| Contents initialised | Yes | No |
| Data can be modified | Yes[16] | Yes |
| Mixed data types | No | No |
| String handling | Limited | Yes[17] |

`Buffers` and `Arrays` can hold many kinds of data. Currently all the data must be of the same type. The types available are:

- 8, 16 or 32-bit Integers
- Floating point numbers
- Arrays of string constants

---

[14] Changing an object directly into a number may be prevented in future.
[15] The Keyword MAKE is used to create an analogue object here. MAKE will be described in detail later.
[16] Arrays can hold constant data in ROM, or variable data in RAM.
[17] Buffers of text are used for most text handling. See *Text Buffers* on page 120.

- Text Buffers
- Arrays of Pointers

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

```
-->ARRAY some_data (32 , 10) 1 , 2, 3, 4 END
-->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 the `END` command indicated the end of the definition of the `Array`[18].

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-->
```

Buffers are similar to Arrays:

```
-->MAKE buff Buffer (32)
-->buff. Put (1)
-->buff. Put (2)
-->PRINT buff . (1)
    2-->
```

Here we made a `Buffer` that took 32-bit integers, and put a couple of numbers into it, then printed one of them.

## SUMMARY

- Numbers may be stored in variables using the `:=` symbol.

- There are two types of number – integers, which are 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.

---

[18] Note that Arrays of constant data are not created like other objects; they are more akin to procedures in the way they are defined. This is because they will eventually be stored in ROM (like procedures) rather than RAM (like all other objects).

30

- 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 `AND`, `OR`, `EOR` and `NOT` may be used to manipulate 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.

# 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.

## 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` causes the cursor on the terminal to move to the beginning of the next line. If the cursor is already on the bottom line, the text on the screen is moved up one line (or 'scrolled').

`BEEP` causes the terminal to beep. This is useful for attracting attention, for example:

```
-->PRINT BEEP, "An error has occurred", CR
An error has occurred
-->
```

`PRINT CHR` is used for printing special characters on the terminal screen. It is followed by a value, which is 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 appears in the Appendices.

```
-->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[19].  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 procedures.

```
-->REPEAT 4 PRINT INDEX:2, timer, random:10, CR
 1 14082  14627625
 2 63173   2363283
 3 50987  47844170
 4 38904  37278678
-->
```

Note that `Timer` 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.

---

[19] Other *types* of format parameter (e.g. strings) may be used to achieve unusual formatting of integers.  These are not defined here as they are of more specialist interest.  See **:** in the *Language Reference* for more information.

```
-->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.

34

```
-->REPEAT 10 PRINT "[" , "abcdefghij" :INDEX0-5:10 , "]" ,
CR
[     abcde]
[    abcdef]
[   abcdefg]
[  abcdefgh]
[ abcdefghi]
[abcdefghij]
[bcdefghij ]
[cdefghij  ]
[defghij   ]
[efghij    ]
-->
```

This is probably a useful place to mention that *string constants* understand one message: `Length`.  This returns the number of characters in the string.

```
-->PRINT "12345" . Length
     5-->
```

## 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 *Getting Started* on page 6. 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 the low-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
  timer . Reset
  WHILE temperature < final_temp_1
  [ demand_temperature (timer . Time * rate_up)]
  timer . Reset
  AWAIT timer . Time > 1000 * 60 * minutes_hold
  WHILE temperature > final_temp_2
  [ demand_temperature (timer . 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 command 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 command.

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
```

When this is typed into the terminal (it is not necessary to get the indentation correct), the terminal screen will show the following:

```
-->TO monitor_in
02>  FOREVER
03>    [ AWAIT sense_in.Asserted = FALSE
04>      AWAIT sense_in.Asserted = TRUE
05>      counter := counter + 1
06>    ]
07>END
Procedure Defined
-->
```

Note the prompt changing from `-->` to `02>` indicating that Venom knows it is reading a procedure definition. The number before the `>` is the procedure line number.

`Procedure defined` is printed when a procedure has been successfully compiled.

Pressing Ctrl-C during the entry of a procedure will get you back to the command line, abandoning any text you have typed.

*If you do this you may get a message informing you the compiler could not clean up after itself properly – this is normal and may be ignored.*

In the rest of this manual, only the procedures themselves will be shown, without the line numbers or command line prompt.

## Procedure Names

Procedure names take exactly the same form as variable names[20].  See *Variables and Expressions* on page 19.

## Calling Procedures

A procedure is executed, or 'called', by typing its name as if it were a command, or by including its name in another procedure:

```
TO reset_then_monitor
  counter := 0
  monitor_in
END
```

Another example is given below.

---

[20] In fact procedures are just another type of variable.

```
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, Venom will run out of memory very quickly. This is because each time a procedure is called, a small amount of 'stack'[21] 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 repeat_led_toggle
  led . Toggle
  repeat_led_toggle
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[22] 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.

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 six 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.

---

[21] *The Stack* is a large block of memory used by procedures to remember where they were called from.
[22] *Code* is a general term for any type of program text.

## Procedures are not forgotten

Because Venom runs on hardware with a non-volatile memory (sometimes referred to as 'Battery-backed RAM'), it does not lose any of the procedures that you have entered, 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.

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

## 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 by reference. If you want to pass by reference, you can use pointers. See *Pointer* Expressions on page 78.

## Procedures that Return Information

Procedures that return information use the `RETURN` command. `RETURN` is always followed by an expression. When `RETURN` is encountered, the expression is calculated and the resulting value is passed back to where the procedure was called.

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
-->
```

## Leaving Procedures

A procedure is normally left when the last statement has been executed. Control then returns to whatever called the procedure. However a procedure may be left before the end using `RETURN`. `RETURN` will leave 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`.

## 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`, 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 variable `int_temp`. If there is a global variable with the same name in the system, it will be ignored inside the `measure_temperature` procedure, and the local version used instead.

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[23].

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 feature is not often used in control applications, but it sometimes allows an elegant solution to a problem. A rather trivial example of recursion is:

---

[23] This may change to an 'un-initialised' value in future, to aid debugging.

```
TO recursive_procedure
  recursive_procedure
END
```

Whenever a procedure is called, it allocates some stack memory in which to store 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[24] then each instance of the procedure is entirely independent.

For example, the procedure below will find a given value in a `Buffer` object containing sorted values. It does this by considering the whole `Buffer` and then determining whether the number 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. The two `PRINT` commands clarify the calls.

```
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
```

---

[24] This is one reason recursion is not often used in control applications. Affecting external devices is mostly what they are about!

```
-->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
-->
```

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.

## 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 [242 bytes @$26077A]
;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 the file you create in your text editor.

## Deleting Procedures

The DELETE command is used to delete individual procedures.  For example:

```
-->DELETE monitor_in
-->
```

When a procedure or variable is deleted, its definition is removed from memory, and it may no longer be used, but its name is still stored (and will appear in the 'Undefined' category of LIST WORD).  This is because other procedures that refer to the deleted item must still be able to display its name in their listing. However, the memory required to store the name is very small.

## Predefined Procedures

Venom predefines three procedures when its memory is cleared. The three procedures are called `startup`, `init` and `main`. 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 is left immediately when `RETURN` is used.
- 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.

# YOUR DEVELOPMENT ENVIRONMENT

Application code in Venom should be written using your favourite text editor. When you want to download your program to the Venom system, Cut-and-Paste either the whole program, or just the procedures you have changed, into your terminal window.

If you don't want to see hundreds of lines of your code flying by while you download it, you can use `PROGRAM START` ... `PROGRAM END` to give you a summary of the download process. You can group any number of procedures between these two commands. This is how some code in your editor might look:

```
PROGRAM START

  TO checked_temperature
    LOCAL reading
    reading := temperature
    IF reading = -1
      reading := 25
    RETURN reading
  END

  TO weather
    hum_reading := humidity
    temp := checked_temperature
    PRINT hum_reading, temp, CR
  END

PROGRAM END
```

When these are downloading they look like this:

```
-->PROGRAM START
Loading procedure checked_temperature... Procedure Defined
Loading procedure weather... Procedure Defined
Loading finished.
0 error(s)
```

Another advantage of `PROGRAM` is that syntax errors are easier to spot. Here, I have deliberately put an error in one of the procedures:

```
-->PROGRAM START
Loading procedure checked_temperature...
03>    reading := temperature..value
                                  ^
Syntax Error at line 3: Expected message name.
1 Syntax Error(s): procedure not defined.
Loading procedure weather... Procedure Defined
Loading finished.
1 error(s)
-->
```

## Getting a Command Line

Sometimes when you are downloading programs an END or PROGRAM END may get missed. If this happens inside PROGRAM then you will not see anything you type at the command line. If it happens outside PROGRAM then the prompt may indicate procedure line numbers:

```
20>  [User presses Carriage Return repeatedly]
21>
22>
```

In either case, Ctrl-C will get you back to a command line:

```
22>  [User presses Ctrl-C]
Input Aborted
-->
```

## SUMMARY

- Use a text editor to write your application code.

- Use Cut-and-Paste to send the text to your terminal.

- You don't have to send the whole application every time.

- Use PROGRAM and PROGRAM END to improve the readability of the download.

- Use Ctrl-C to get back to a command line.

# 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 very much 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

Objects are created with the MAKE command. The example below shows an object being created to represent a heater controlled by one of your controller's digital channels.

```
MAKE heater Digital(32)
```
The MAKE command is followed by the name of the object to be created, in this case heater, and then by the *type* of object, in this case Digital. The '32' means that the object heater should represent 'whatever is connected to digital channel 32'[25].

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

This example shows the creation of the object representing the temperature sensor used in the examples:

---

[25] The datasheet for the controller will tell you where each channel is.

```
MAKE thermometer Analogue(47)
```
In this case `thermometer` is the object representing a sensor connected to analogue channel 47.

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 a complete mystery to another user). For example, some other objects in the oven control system could be defined as follows:
```
MAKE display AlphaLCD (20,2,0)
MAKE buzzer Digital (129)
```

## When to create Objects

In the examples given, objects have been created 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, the objects are normally created immediately after the controller is turned on, preferably in the `init`[26] procedure.

There are several reasons for this:

- Objects are not retained over power failure, and so must be re-created before the program may use them

- It is good if all the object definitions are in the same place in the program listing, for easy code maintenance

- 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*, page 84.

## The Startup Procedure

There is always a procedure called `startup` in Venom. Even if you don't create one, a default one will be 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`:

---

[26] It is advisable to write a procedure called `init` to do all the initialising of objects and variables needed by your application. The section on *startup* explains this more fully.

```
-->LIST startup
The text of the DEFAULT startup procedure:
TO startup
  MAKE system OperatingSystem
  ErrorAction := UserSwitch(1) + 1
  MAKE serial AsynchronousSerial(38400,1)
  IF NOT UserSwitch(2)
    serial . Baud := 9600
  MAKE net I2Cbus
  MAKE led OnBoardLED
  MAKE clock RealTimeClock
  IF Runmode
  [ led.flash(10)
    init
    main
  ]
END
-->
```

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

These are empty procedures that you are free to re-define how you wish. You can re-define `startup` itself, 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 they will be reliably executed at `startup`, when power is applied to the controller.

*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 ask `thermometer` to return its reading.

```
-->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:

50

```
-->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 / 2.5460000
```

## 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 (13)
-->
```

`Led.Flash(13)` sets the LED on your controller flashing the Morse Code sequence 'SOS'. See `OnBoardLED` in the *Venom-SC Object Reference* 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-->
```

## What Objects are available?

In Venom, there is a fixed set of object types, and each type of object responds to a fixed set of messages. The exact definition of each object, and of the messages it responds to, is given in the *Venom-SC Object Reference*. A less formal description of how to use a selection of objects is given in *Part 2* of this tutorial.

## 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 useful ICs to provide useful functions not on the main controller, including:

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

## Deleting Objects

Any object created with `MAKE` may be removed, if it is no longer required, with the `DELETE` command followed by the name of the object to delete. For example, to remove the `heater` object:

```
-->DELETE heater
-->
```

In general most applications will not need to use `DELETE` as there is rarely a need to remove objects you have defined. However `DELETE` may be useful during development, and in those rare applications that use objects in a dynamic way, creating them and destroying them as needed.

## 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*, page 28, for more information.

## SUMMARY

- Objects are created with the MAKE command.
- It is good to use meaningful names for objects.
- Messages are sent to objects by placing a dot after the object and before the message name.
- Objects may be removed with the `DELETE` command.

# 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[27].

```
TO startup
  MAKE system OperatingSystem
  ErrorAction := UserSwitch(1) + 1
  MAKE serial AsynchronousSerial(38400,1)
  IF NOT UserSwitch(2)
    serial . Baud := 9600
  MAKE net I2Cbus
  MAKE led OnBoardLED
  MAKE clock RealTimeClock
  IF Runmode
  [ led.flash(10)
    init
    main
  ]
END
```

Notice that near the end of the `startup` procedure two procedures (called `init` and `main`) are called. These are also created by default by Venom when memory is cleared, but they are empty. This allows you to re-define them for your application.

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

`Main` is intended to call the code that actually does your application.

Try altering `main`:

---

[27] `Startup` may change with new language releases – list it if you need to know its exact form.

```
-->TO main
02>PRINT "Hello world",CR
03>END
Procedure Defined
-->
```

Now type `run` at the command line:

```
-->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 alter the `init` procedure in a similar way to `main`.

```
-->TO init
02>PRINT "Init is before main",CR
03>END
Procedure Defined
-->
```

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.

## Program mode

Up until this point in the tutorial you have been using Venom solely in *Program mode*. Program mode is usually only available via a switch on the controller, or some other hardware control. This is to stop controllers that are running application code accidentally going into Program mode.

When Venom first starts, it looks at the Program/Run mode switch to see what mode it should be in. Also, operating the Program mode switch will restart Venom in the new mode.

Program mode is used while developing applications. Whenever you power up your controller in Program mode, you will see the `Clear Memory` message.

```
VM-1 Control Computer running Venom-SC
Version 2003 04 03
Copyright 2000-2003 Micro-Robotics Ltd.
Clear memory: Y/N/S ?
```

You can reply to the question in one of three 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`, `init` and `main`, 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. Your `startup` procedure is run, though it may not do much, as the controller is in Program mode.

`S` (*Skip*) Means skip your `startup` and instead run Venom's default `startup`. This is for those rare occasions when your `startup` does not allow you to get to a command line prompt 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.

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

As mentioned before, Run mode may be emulated from Program mode by using `Run`[28] at anytime.

## Don't let your application end

You may have noticed that the default startup procedure, and the altered `init` and `main` above, ended with the command line prompt (`-->`) being issued. The

---

[28] Actually `Run` is short for System.Run, an operating system message call.

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 the application code would normally enter some sort of infinite loop, and never come to an end. 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[29].

## 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 is 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, you know 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.

---

[29] There are exceptions to this, for example some applications might want to use the command line as part of a protocol over a network, rather than having a stand-alone application.

**Run mode**

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

## SUMMARY

- The `startup` procedure 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.

- The LED may be used to indicate the operational state of the controller.

# 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 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, which also has a user interface.

### Without multitasking

Lets say the machine is an oven with a temperature sensor that needs polling[30] every 100 milliseconds. The sensor is used to control the oven temperature by turning off the oven 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[31], but the finished code is usually quite inflexible.

---

[30] *Polling* something means looking at it to see if any action is needed.
[31] Some mission-critical multitasking systems are written this way.

Here's one example of how it might be written[32]:

```
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.

---

[32] See *Keypad* on page 110 for details of how to use this object.

Oven control task:

```
EVERY 100
[
  IF temp_in . Value > target_temp
    oven . Off
  ELSE
    oven . On
]
```

User interface task:

```
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 can 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 viewed separately for the most part.

## 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

60

## Starting Tasks

When a Venom application starts, there is just one task running – the one that executes the `startup` procedure. This is called 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 the block of code it is given and runs that as a new task. Often the block of 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 one uses 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 Carriage Return here to see the prompt]
-->
```

When the five numbers have been printed, the task runs out of code, and quietly disappears[33].

Notice that the `==>` prompt was displayed before the numbers indicating the new task. This is because the prompt was printed before the new task had finished (in fact before it had a chance to print the first number).

The third prompt, seen in response to the user typing another Carriage Return, shows the task has gone.

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

*In general, it is bad practice to use STOP as part of your application code. If you need a task to end, it's better to signal to the task so that it can terminate of its own accord. STOP should mainly be seen as a development tool.*

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 reference number (see *Listing Tasks* below). 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 from a command in the code, only by typing CTRL-C. Additionally, typing CTRL-C at an empty command line will stop all active tasks:

---

[33] Actually it's only quiet as seen from your point of view. Under the hood, a lot is happening, which is one reason not to start and stop tasks without a good reason.

```
-->START FOREVER[]
==>[User presses Carriage Return here to see the prompt]
==>[User presses CTRL-C  here to stop the task]STOP  ALL
-->
```

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

## Listing Tasks

The `LIST  TASK` command produces a list of all the tasks Venom is running, including details of where each task is.

```
-->START FOREVER []
==>LIST TASK
TASK 0:
in the command line.
[BC_LIST @ $2610FA]
_____
TASK 1:
in proc1 (line 1)
in proc2 (line 1)
in a task started from code at $26057C, (old command
line?).
[BC_INC @ $260602]

_____
==>
```

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.

## Our Simple Multitasking Model

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

If you follow the rules in our 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 the example above, 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 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 any messages that require locking

*Locking* is a feature of some objects. It is used in more complicated multitasking systems. Most of the messages you are likely to use don't require you to explicitly lock an object first. Messages that require explicit locking don't fit into our simple model, so don't use them. These messages are well documented, and tend to be the lower-level ones.

### Consider task latency

For every extra task you have running in Venom, your code may miss 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.

## 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.

This section of the manual continues – glance over it and read the sections that are appropriate to your application.

## Robust Applications

When you have your application code performing roughly to your specification, read the appendix *D: .*

This provides a checklist to create a robust Venom application.

## SUMMARY

- [None]

# DEBUGGING

The simplest form of debugging available in Venom is `PRINT`. If there is a problem with the program you are working on, insert a `PRINT` 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 terminal connection is being used as part of the application. In these cases the output may be redirected to another device.

Debug output may be redirected using the `TO` keyword (see *Text Handlers and Redirection,* page 85).

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 held for later examination

`PRINT` is usually all that is needed to catch most bugs.

## Finding Procedure Lines

Runtime error listings refer to line numbers. One way to find the line numbers for a procedure is to download the procedure. The prompt will indicate line numbers.

## SUMMARY

- [None]

# ERRORS AND EXCEPTIONS

This chapter discusses runtime errors, and introduces the four keywords related to defining special blocks of commands that may be 'broken out of' in exceptional circumstances.

Most applications will never have to use them, but if you have to deal with error conditions or exceptions then these keywords can be very useful[34].

## REGION and EXIT

The `REGION` command defines a block of code from which you can exit in an unusual manner. Instead of having to complete any loops, or return from any procedures called, you can just use `EXIT` from inside the `REGION` to jump straight back to where the `REGION` started. This is used to deal with exceptional circumstances that can't be easily dealt with by the normal flow of the code. In fact it is best to use `REGION` and `EXIT` only in exceptional circumstances, as their use is complicated.

When using `REGION`, the keyword should be followed by a name for the region and then the command or, more likely, a block of commands. `EXIT` specifies the name of the region to exit. When `EXIT` is called, the next command *after* the `REGION` construct is run immediately.

```
TO log_values_into_the_buffer
  REGION log_data
  [
    FOREVER
    [
      data_store . Put (sensor_reading)
      IF exit_button
        EXIT log_data
    ]
  ]
END
```

Often it is useful to determine whether the region finished normally or whether `EXIT` was used. `REGION` returns a value, which is `TRUE` if the region was exited prematurely, and `FALSE` if it finished normally. The following example tests `REGION` to see which case occurred. Notice that `THEN` is used to improve readability.

---

[34] They are comparable to 'longjmp' in C.

```
TO log_values_into_the_buffer
  IF REGION log_data
  [
    FOREVER
    [
      data_store . Put (sensor_reading)
      IF exit_button
        EXIT log_data
    ]
  ]
  THEN
    PRINT TO display , "User Escape"
  ELSE
    PRINT TO display , "Logging done"
END
```

## Nesting REGION and EXIT

EXIT doesn't have to be called from the procedure that defined the region – it may be called from any point in the code that is 'inside' the region, i.e. from any code that the REGION code calls.

Also, regions may be nested, and any level of exit is possible, just by naming the region you need to exit from. You can even nest regions of the same name. Here, you will exit to the most recently entered region of that name.

## Runtime Errors

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

### Error reports

Runtime error reports will generally be like the example below.

```
Run-time error 5: Un-initialised variable: target_temp
at $2600E0 in jim (line 1)
in karl (line 1)
in the command line (line~1?)
```

The error number is just for reference. The error text details the kind of error that occurred, and may give supporting information, in this case the name of the offending variable.

The address in memory of the code that caused the error[35] is listed in case it is needed.

Then the report goes on to list the procedure the error occurred in, together with a rough line number[36]. It also gives the list of callers, i.e. the procedures that called the code that failed. Because Venom code is compiled, it is not possible for the error listing to be right every time when it reports the line number of an error.

If the error was in the main task, then a command line prompt is issued. Any unaffected tasks will carry on running.

Dealing with runtime errors like this is fine during the development of a program, but is unacceptable during operation.

## Reset on Error

There are several ways to make Venom do something more useful when a runtime error occurs. The first is to CATCH the error (described below). The second is to reset the controller when an error occurs. This may be achieved simply by using the system message ErrorAction.

```
System . ErrorAction := 1
```

This is described more fully on page 147.

## Catching Errors

The CATCH command operates similarly to REGION except that the block of commands is not left explicitly with EXIT, but only if an error occurs.

Whereas REGION returns TRUE or FALSE, a CATCH command returns the number of the runtime error that occurred, or zero if no errors occurred. Each error message has an error number associated with it. A full list is provided in the *Venom-SC Language Reference*.

CATCH should be followed by either the error number of the error to be caught or the keyword ALL, if all errors are to be caught. Note that, because of its nature, the 'RAM Full' error cannot be caught. The following example shows CATCH being used to guard against a 'Division by zero' error.

---

[35] Sometimes this will be the address of the next code along.
[36] To find line numbers within a procedure download it – the prompt indicates line numbers.

```
TO divide(a,b)
  LOCAL result
  IF CATCH 6
    result := a DIV b
  THEN
    [ PRINT "Division by zero - returning 0", CR
      result := 0
    ]
  RETURN result
END
```

In this example, the line after CATCH is always executed, but may generate an error. The line after THEN is only executed if an error occurred. RETURN is always executed.

Sometimes the use of CATCH and REGION can be confusing. If you use a variable to hold the value returned by CATCH or REGION then it becomes more readable:

```
TO divide(a,b)
  LOCAL result, error
  error := CATCH 36 [result := a DIV b]
  IF error
    [ PRINT "Division by zero - returning 0", CR
      result := 0
    ]
  RETURN result
END
```

Often it is useful to be able to handle some of the errors that can occur, but still inform the user about others. The THROW command can emulate any error. The following example procedure deals with division by zero, but passes other errors back to the Venom runtime error handler.

```
TO safe_divide (a,b)
  LOCAL result,err_num
  err_num := CATCH ALL
    [ result := a / b
    ]
  SELECT CASE err_num
    CASE 6
    [ PRINT "Division by zero - returning 0", CR
      result := 0
    ]
    CASE ELSE
    [
      THROW err_num ;Let Venom deal with the rest!
    ]

    RETURN result
END
```

*Usage Note:*
*In general you should not* PRINT *the value of a procedure that uses* CATCH *or*
REGION. *You are not likely to want to do this.*

## SUMMARY

- REGION may be used to define a critical area of code.

- EXIT may be used to leave a region, even from within loops or procedure calls.

- CATCH is similar to REGION: errors cause program execution to drop back to the CATCH.

- The THROW command may be used to emulate a runtime error.

# MACROS

Macros are pieces of program text that have been given a name. They can make your code easier to 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 the 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[37].

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.

```
#DEFINE <name> <text of macro> ;optional comment
```

For example:

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

Macros should always be defined before the compiler sees any program text that uses the macro. If you get this wrong, the compiler will tell you to start again.

One way to make this process easy is to download all your macros before the rest of the code. Once the compiler knows a name is a macro, it will remember it until memory is cleared.

## Listing Macros

Macros may be listed out:

---

[37] It's usual to re-compile all the code to make sure the changed macro has been used throughout.

```
LIST DEFINE ; lists all macros
LIST <name> ; lists out the given macro
LIST WORD   ; lists out all symbols by type, including
macros.
```

## 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 is reserved for macros that take parameters.*

## 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 60 * 60 * 1000.

## Redefining Macros

You can redefine macros, but if you do a warning will be issued. Any code compiled before the redefinition will use the original definition of the macro.

## Removing Macros

If you want to re-use a macro's name for a normal global variable then you have to clear the controller's memory. You can't use DELETE <name> as this will simply compile as DELETE <macro text>, which isn't what you want at all!

## Null Macros

You may define a macro to be nothing:
```
#DEFINE something
```
This will simply expand as nothing at all.

## Macro Limitations

You must define a macro name *before* you first use it

Macros can only be one line long

### Current limitations that may be improved later

Macros can't take parameters

#DEFINE is parsed as a Venom statement but can only occur 'on the command line' (i.e. not inside a procedure definition)

A macro name is a global name, which means that if there is a local variable name the same as the macro name, the compiler will see the local variable rather than the macro

## SUMMARY

- Using macros to name constants and expressions makes for better programs.
- #DEFINE is used to define macros.
- Macros must be defined before they are used.

# FURTHER EXPRESSIONS

This chapter introduces some of the more advanced types of expression. All of these subjects are covered in much more detail in the *Venom-SC Language Reference*.

## 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.

```
-->TO proc var := 1 END
Procedure Defined
```

In the example above, the variable `var` was created, but not assigned a value as the procedure has not been called. Below, we've tried to read its value:

```
-->PRINT var
Run-time error 5: Un-initialised variable: 'var'
near bytecode 'LDG' at $26014E
in the command line (line~1?)
```

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.

## 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 immediately before 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 the values can be printed 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 there is a way of expressing an integer value as its ASCII character, using single quotes:

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

*String constants* are used to hold strings of characters.  These are defined in 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
-->
```

You may also find the length of a string constant:

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

Note that these string constants are *constant*.  You can only print them, find their length, send them as parameters and assign them to variables.  This is more than enough for most applications. For more flexibility in text handling see *Text Buffers* on page 120.  Also see the sections on printing strings and Arrays of strings.

### Escape code sequences

Sometimes you might want to put the quote character `"` into a string.  This is obviously not possible in a straightforward way.  Instead you have to use an *escape code sequence*.

In Venom-SC 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:

- \\ gives a single \

- \" gives "

- \$hh gives any ASCII character in hexadecimal[38]

- \^hh insert sprite number $hh as a character[39]

- \ with any other character is ignored completely

### 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.

## Bit-wise Operators

The operators AND, OR, EOR and NOT have already been introduced for comparing Boolean values. They may also be used for manipulating numbers bit by bit. For all the operators, the value of each bit in the result depends on the one or two corresponding bits in the given numbers.

```
-->PRINT ~~ %1010 AND %1100, CR
  1000
-->
```

When laid out as shown below, it can be seen that each bit in the result is 1 only if both of the bits in the given numbers is 1:

```
AND 1010
    1100
    1000
```

Similarly, OR works by placing a 1 in the result if one or more of the bits in the given numbers is 1.

---

[38] \$00 will insert a zero character into the string. However this will serve to terminate the string at that point as far as Venom is concerned.

[39] This feature is only used by the GraphicsLCD object.

```
  -->PRINT ~~ %1010 OR %1100, CR
    1110
  -->
OR  1010
    1100
    ────
    1110
```

EOR is similar to OR except that a 1 is returned if one and only one of the bits is 1.

```
  -->PRINT ~~ %1010 EOR %1100, CR
    0110
  -->
EOR 1010
    1100
    ────
    0110
```

Finally, NOT simply replaces 0s with 1s and vice versa.

```
  -->PRINT ~~NOT %1010, CR
  11111111111111111111111111110101
  -->
NOT 1010
    ────
    0101
```

The reason that so many ones are printed is that Venom calculates numbers using 32 bits. When all the left-most bits are 0s, they are not printed, but when NOT is used, they become 1s, and so are printed.

## Memory Expressions

It is possible to access the memory of the your Venom-based controller directly using the ? symbol. This is not normally required since Venom already makes use of most of the available features. It is discussed in detail in the *Venom-SC Language Reference*.

## Pointer Expressions

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
-->
```

## 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[40]. 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
! (procedure_table . Element(serial . Get – 'A'))
END
```

The pointer expression

```
! (procedure_table . Element(…))
```

calls one of the three procedures depending on the value of `serial . Get`: A, B or C will call one of the three procedures via pointers in the table.

## Parameters to procedure pointers

Procedure pointers may also be sent parameters. You have to help Venom decode this rather complicated construction with parentheses. The square brackets may also be necessary, to force Venom to treat this construction correctly.

```
[(!procedure_table. (n)) (par1 , par2)]
```

*Note the name of the `Element` message has been omitted. This is a shortcut allowed by Venom.*

---

[40] `SELECT CASE` may also be used to select one of many varied actions.

# 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. As a result, an object can print whatever information it deems to be useful. The `[]` are used as a reminder that `thermometer` is not a simple number.

A particularly useful object to print is the $I^2C$ 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, a realtime clock/calendar 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 *Venom-SC Object Reference*.

## Using NIL

There is a special type of object called `NIL`. In short, it is an object that accepts any message (and ignores it); returns `0`, or `FALSE`, to anything that asks; and ignores any text printed to it.

It is often useful to use in the place of a real object or number. `NIL` may be tested for equality with anything else:

```
var:= 0
IF var = NIL PRINT "it was Nil"
```

## Object Expressions

When you create an object with `MAKE`, you are both creating an actual *object* in the memory of the Venom system, which may also affect the hardware in some way, but you are also creating a *label* for that object (called a global variable name).

```
-->MAKE relay Digital (128)
-->
```

Here, `relay` is the label, or global variable name, for the `Digital` object we created.

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

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

```
-->Relay2 := relay
```

This *has not* created another object, but just another label for the *same object*.


You can also 'lose' the object:

```
-->Relay := nil
```

Here, the original object has lost its original label, but still exists in some sort of limbo. `Relay` can't be used to access it. In this example we still happen to have a label for it (`Relay2`).

*The above examples are not suggestions for how you should write your Venom code, but an attempt to illustrate how objects and their labels 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 a `Digital` type object.

```
TO pulse_ten(obj)
  REPEAT 10
    obj.Pulse
END
```

Secondly, a procedure may return an object. However, it is potentially dangerous to create an object within a procedure and then return it, since it is very important

82

that an object is deleted when it is no longer used so as not to tie up memory unnecessarily[41].

## Sending Messages to Expressions

Messages are only sent to objects, however, an object may be represented by an expression that is more complicated than just its label. 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.

## Dot Chaining

Some of the examples in this manual will include sending messages to objects that are returned by messages. An example of this may be seen when using the `I2Cbus` object. One way of writing a routine to send a byte to a device on the bus would be:

```
TO send_value (address , value)
  net . On
  net . Put (address)
  net . Put (value)
  net . Off
END
```

However, many of the `I2Cbus` messages return the `I2Cbus` object as their result. Thus the code may also be written:

```
TO send_value
  net
    . On
    . Put (address)
    . Put (value)
    . Off
END
```

*Each of the messages has been shown on a new line for clarity.*

This is called 'Dot-Chaining' and is sometimes a convenient way of expressing something[42].

---

[41] There are exceptions to this: some objects don't actually take any memory at all, and so may be created at will with no adverse effects.

[42] In this case it is also runs slightly faster.

## Objects in Arrays and Buffers

*Currently you may not put objects in Arrays or Buffers, though you may fill an* `Array` *with* pointers *to objects.*

## Creating Temporary Objects

Sometimes it will be necessary 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 temp := NEW Digital (128)
```

## Killing Objects

It is vital to remove those temporary objects that consume resources[43] after they have been used: sending a `Die` message to the object does this. If this was not done, there would be another object created each time the procedure was called, and memory would soon be filled.

The object and the Venom variable name that refers to it are not the same thing. If you kill an object, you don't kill the name, so you might accidentally try to use the dead object. Venom has an internal mechanism that will prevent this most of the time, but really this is a problem that should be avoided by designing your application software well.

The keyword `DELETE` will kill an object, *and* delete the reference to it. However that still doesn't solve the problem of accidentally using dead objects:

```
-->MAKE b Buffer("")
-->c := b
-->DELETE b
-->c.Put(0)
Run-time error 25: Message to dead object
at $260408 in the command line (line~1?)
-->
```

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 had to intervene to make sure the system didn't crash. Good software design is still necessary.

---

[43] Note that some objects don't use any resource – the whole of the object is held in the Venom variable. Digitals tend to be like this. You don't need to kill these objects, though it does no harm if you do. You can make as many of them as you like, and it won't matter.

# FURTHER PRINTING

This chapter introduces further facilities available with the widely used `PRINT` command.

## Text Handlers and Redirection

Normally `PRINT` sends its output to the serial object.  It is possible to print text to a different object using `PRINT TO`.  The object to which the output is printed might be one of the types listed below, which are able to accept *print jobs*.

| Type | Result |
|------|--------|
| `AlphaLCD` | The text is displayed on the alpha-numeric LCD |
| `AsynchronousSerial` | The text is sent directly to the serial port |
| `GraphicsLCD.Window` | The text is printed in the window |
| `TextBuffer` | The text is added to the contents of the buffer |
| `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 `PRINT` normally sends print jobs may also be changed – see the Operating System message `Output`, in the *Venom-SC Object Reference*. The concept of changing the object to which text is printed is called 'redirection'.

## 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 *Venom-SC Object Reference*.

## How PRINT works

All the text handlers described above only accept text consisting of normal and *extended* characters[44]. It is the task of the PRINT command to convert all the expressions in the print list into acceptable text. During this process it divides up the total print output into packets of text. Each 'packet' is a collection of about 100 characters that are assembled before being passed on to the text handler as a *print job*. Short PRINT statements will cause less-than-full print jobs to be used, and very long ones will require many print jobs to carry the output.

Since it is important that no more than one task writes to a specific object at a 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.

---

[44] Extended characters are special characters put into a print job to represent the special print keywords. This is done using a special Escape Code character sequence.

# 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 at 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[45] – it's not under your control[46].

## Atomic operations

An atomic operation is one that can't be split into more than one part by a task swap. Venom has several operations that are defined as atomic.

- Writing a variable
- Reading a variable
- All messages to objects except where documented

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 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 micro-processor 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

---

[45] Actually, beneath the Venom level, the task system is 'co-operative' – that is the time to swap is decided on by the task currently running. However, each part of the Venom system obeys the rule that it will swap tasks before it has reached its maximum allowed duration (~2mS), and usually by 1mS on average.

[46] You can ask for an explicit task swap with the SWAP keyword, but this is almost never needed.

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[47].

## 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 worked 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 going ahead and reading the value you want to read. An example of this is *reading* a Venom integer or float.

---

[47] The Venom language has been designed so that it uses objects to deal with many of the external events that have strict timing requirements. This relaxes some of the requirements on the task system.

## 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.*

That is, only one task may write to the variable, though any number may read it.

We saw the example of the value `target_temp` being used this way earlier.

## 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 := 1          ;signal sub-task to do its stuff
  do_something_else
  WHILE signal = 1     ;wait for signal back
    WAIT 2
  carry_on
END

TO sub_task
  FOREVER
  [
    WHILE signal = 0  ;wait for signal to act
      WAIT 2
    do_operation      ;act
    signal := 0       ;signal back
  ]
END
```

*The WAIT 2 commands make sure the two tasks don't hog processor power while they are waiting for the signal. We could equally have used WHILE … SWAP, or AWAIT signal = …*

## Sharing other resources

The kinds of resources that are difficult to share are those that are affected by more than one task. The most common problem encountered with these sorts of resources is that you want each task to use the resource in such a way that it

doesn't mess up any other task's usage of the resource. The next most common problem is that you don't want to hold up an important task while it waits for a slower task to stop using the 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 – you may read keys or characters 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. Another task has a line that decrements the same variable.

```
count := count + 1 ;in task A
...
count := count – 1 ;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 happens 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 `SLEEP` instruction in the micro-controller's instruction set, to shut down the CPU whenever possible. It's worth knowing which kind of instruction will cause the controller to idle, and those which keep it awake.

The most common examples are `WAIT` 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.

# 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.

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 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 above. The statement `count := count + 1` and its converse are called *critical areas* of the code. This is discussed below.

### Implicit Locking

Some Venom objects implicitly use another object to carry out their work. For example, a `Digital` object may use an I$^2$C Bus object, if it is an I$^2$C digital channel.

In these cases, the client object (`Digital`) will always lock the I$^2$C bus while using it[48].

Another case of implicit locking is printing. When you print to an object, 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 locking mechanism built into them. Objects with no lock will still accept all the locking messages, but just ignore them. Locking behaviour is documented as part of each object's definition.

---

[48] Actually, it doesn't need to fully lock the bus: `Digital` just makes sure the bus is free and then uses it quickly before a task swap is required.

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 continually 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.

## 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.
```

## Restored 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.

```
obj . Lock
...
REGION reg
[
  obj . Lock
  obj . ...   ;use the object
  if exception EXIT reg
  obj . ...   ;use the object
  obj . Unlock
]
...
obj . UnLock
```

The object no longer gets unlocked as many times as it should. This problem can also occur when using RETURN or CATCH.

You can avoid situations like this with care, 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 take a parameter, 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 ;record 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 send the message obj . Lock(0) if you want to make sure the object is completely unlocked at a particular point in your code.

## 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 a non-zero number. 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 new_lock_state

  ;record new lock level:
  new_lock_state:= obj . TestLock

  ;Did we get the lock?
  IF new_lock_state
  [
    obj . … ;Use the object
   ;restore old lock level:
    obj . Lock (new_lock_state - 1)
  ]
END
```

*Notice we had to subtract 1 from the lock level to restore the lock, as TestLock returns the new lock 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.

*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
```

Remember to consider implicit locking (see page 92) when you are thinking about deadlock.

## Ending Tasks

In general, you should try to write your applications so that tasks never end.

If you do need a task to end, don't use STOP or STOP ALL. Instead use a signal to the task to tell it to end naturally[49].

---

[49] The exception to this is if you need your code to deal with an emergency situation where you need to know other tasks aren't active. Go ahead and use STOP ALL.

```
TO a_task
EVERY 100
[
  do_something
  IF task_stop_signal
    BREAK
]
END
```

## 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.  At the time of printing this tutorial we haven't implemented a special object that will fill this role, but that doesn't really matter; you can use any object with a lock to do the job:

```
MAKE code_lock Buffer("")
…
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
```

## Internal Operation

Here are some details of the internal operation of the Venom Task Manager.

### Task Death

When a task dies, Venom automatically makes sure that any objects the task held locked are unlocked.  It does this by scanning all the global variables and the stack of the dying task and sending each object it finds a special message.  In a large application this may take a few milliseconds.

## Object Death and Lock

When an object is killed, the Venom Task Manager checks all the other tasks to make sure that no other task was waiting to lock the (moribund) object. If there were tasks waiting, then these are forced to halt with an error.

# Part 2:
## Object Tutorial

# INTRODUCTION

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 Venom-SC.

For a complete description of every type of object see the *Venom-SC Object Reference*.

*A note to those familiar with C++, Java, etc:*
*All the object types in the Venom system are predefined. That is they 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. We are open to suggestions for new objects.*
*You can't create your own objects currently, though in the future we may provide the facility to extend the language with objects that have been written for Venom in C or assembler.*

## 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. Where this is not possible the examples given will be for the VM-1 control computer from Micro-Robotics Ltd, and to a lesser extent, its associated application boards and I/O cards.

## Venom Channels

Before we go any further it's worth talking about *Channels* in Venom. A channel is a single Input/Output port 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 ports by the manufacturers of the ICs concerned.

For example, digital I/O channels on the main I$^2$C bus are numbered from 128 to 255. Digital I/Os coming directly off the micro-controller are numbered from 1 upwards; the datasheet for the particular controller you are using will 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.

# DIGITAL

The `Digital` object type is designed for reading and writing digital input and output. It doesn't matter whether the I/O port is located on the controller itself, on an IC connected to an $I^2C$ bus – `Digital` will handle them all.

`Digital` can handle single digital I/O ports, or several I/O ports grouped together.

`Digital`'s `MAKE` takes either one or two parameters. With one parameter, it creates a single digital channel, on the channel specified. With two parameters, a 'grouped' `Digital` will be created using all the channels from the first to the last channel given.

## Single Channel Digitals

The following two lines create single channel digital objects.

```
MAKE cpu_dig Digital (3)
MAKE i2c_dig Digital (128)
```

The first line creates a `Digital` object to control an I/O pin on a VM-1 controller. The second line creates a `Digital` object on an IC attached to the $I^2C$ bus.

When these digitals are first made they are configured as *inputs*. The object will become an output the first time it is written to[50]:

```
cpu_dig . On    ;becomes an output – pulling low.
cpu_dig . Off   ; pull high now.
```
*Note that On is usually a logic LOW.*

If you want to turn a `Digital` back into an input, then you can use

```
cpu_dig . Output := FALSE
```

Now you can read its state with the `Asserted` message:

```
-->PRINT cpu_dig . Asserted,CR
    -1
-->
```

The -1 (`TRUE`) means the input was *asserted*, i.e. it was at the logic low level. If it had been logic high, then 0 would have been returned. There is no standard for

---

[50] Though most digital ports are bi-directional, i.e. selectable to be input or output, some ports can only be one or the other, depending on the actual hardware. Furthermore, Digital I/O on the $I^2C$ Bus's PCF8574 chips is 'psuedo bi-directional' – they have a weak current source to pull them high, but pull low strongly: On is Low, Off is High, and input is achieved by setting the output High, but reading the true input level.

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*.

Asserted may also be used to *set* the state of a digital output:

```
cpu_dig . Asserted := TRUE
```

There are some other messages that `Digitals` understand like `Toggle`, which inverts the state of an output, and `Pulse`, which pulses the output.

## Grouped Digitals

This command creates a grouped `Digital` object with 8 channels.

```
MAKE i2c_dig Digital (128,135)
```

Grouped digitals are mainly used as ports of some kind, and so are usually controlled with the `Value` message.  `Value` sets and reads the *binary value* of the bits on the `Digital` port, where a logic High is a '1' and a logic Low is a '0'.  The lowest channel number is always the least significant bit.  When the value of a grouped digital is set, all the outputs will change state at the same time, subject to the capability of the hardware.

### Example

A grouped `Digital` object may be used to implement a simple stepper motor controller:

```
ARRAY stepper_table () ;table of output states for the
stepper.
%0001
%0010
%0100
%1000
END


TO init
MAKE step_output Digital (128,131) ; 4 chans
END


TO STEP(n)
REPEAT n
[
   step_output . Value := NOT stepper_table . (INDEX0)
]
END
```

This code energises each of the four phases of the stepper motor in turn to produce the stepping motion. An eight-entry `Array` could be used to produce 'half steps'.

Not all digital channels that are numerically near each other may be grouped together. On the I$^2$C bus, only those channels that are on a single I$^2$C device may be grouped. See your controller's data sheet for digital I/Os on the controller itself.

Grouped and single channel `Digital`s may be sent the same set of messages, and they have the same results, except for *reading* `Asserted` for groups. For this case see the *Venom-SC Object Reference.*


## Digital channel numbering

Channel numbers for digital channels on the controller itself are detailed in the controller's datasheet.

Digital I/O channels on an I$^2$C 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 PCF8574**A**.

These are identical except for the I$^2$C 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 I$^2$C 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 |

### Printing

`Digital` objects print 'ON ' or 'OFF' depending on their state (always 3 characters).

When a multiple-bit `Digital` is printed, it prints the value of the `Value` active variable in the given field width.

## Similar Object Types

For pulsed digital signals see PulseCounter, PulseWidthOut, PulseWidthIn, FrequencyIn, 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 $I^2C$ bus – `Analogue` will handle it.

### Input

The following two lines of code make two analogue inputs:

```
MAKE an_1 analogue (40)      ;analogue input on the CPU
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 10-bit resolution (i.e. a full scale from 0 – 1023 comprising 1024 steps) and that full scale is 5 Volts.  Thus the reading in volts would be

```
-->PRINT an_1 . Value * 5.0 / 1024 , CR
     0.844727
-->
```

### Output

Some channels can be analogue outputs:

```
MAKE an_out Analogue (47)    ;analogue I/O on the CPU
```

In this case, this channel on a VM-1 can be an input or an output.  It's slightly confusing that when the channel is an input it has 10-bit resolution, but when it is an output is has 8-bit resolution (i.e. a range of 0 – 255).  Writing to the output is done with `Value` yet again.

```
an_out . Value := 127 ; set the output at ~2.5V
```

### 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.  If you

need a measurement error smaller than ten times the device's resolution, in general you will need to use the device's data sheet and add up the sources of error.

**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 I$^2$C 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 |

# Similar Object Types

Analogue values can also be represented in micro-controller 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 a VM-1.

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 |
|---|---|
| Direct on a VM-1 | 0 |
| PCF8574 on the I$^2$C bus | Use one of the channel numbers[51] as the location, e.g. 128. |

## Similar Object Types

See GraphicsLCD, which can provide a more sophisticated user interface.

---

[51] See *Venom Channels* on page 100 and *Digital channel numbering* on page 104.

# KEYPAD

The `Keypad` object class will drive several types of keypad circuitry.

Each type has been given a number. Currently all the keypad circuits use one or two PCF8574 ICs on the I²C bus, which further drive matrix keypads of different sizes and shapes.

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 |

*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 one or two channel numbers,[52] each of which specify a PCF8574 on the I²C bus.

*An easy way to find out the channel numbers of any PCF8574s on an I²Cbus is to print it.*

```
-->make net2 i2cBus(2)
-->print net2
Devices on the I2C network No.2:

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

  124     496-503     PCF8574A    8 digital I/O lines
  126     504-511     PCF8574A    8 digital I/O lines
  162     PCF8582/83...           RTC/EEPROM...
-->
```

*Here we can see that we have two PCF8574As on the second I²C bus. We'll use both to make the `keypad`...*

```
MAKE kpd Keypad (1 , 496 , 504)
```

This will drive an 8 by 8 keypad on two '8574A chips, on the second I²C bus, near the top of the PCF8574A address range.

---

[52] See *Venom Channels* on page 100 and *Digital channel numbering* on page 104.

There are several ways to read the keypad.  The first 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 *Venom-SC Object Reference Manual* 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, preventing 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
      …
      ;etc.
      WHILE kpd . Asserted 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 in the code example is used to make sure that the code only acts on individual key presses by halting while the keypad is still being operated.  This line does cause the code to wait 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` object instead.  See the next section.

# 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.

You can have more than one `InputBuffer` active at the same time. This might be useful if you need to use key presses in more than one task.

### Sub-type

`InputBuffer` is a 'sub-type' of `Keypad`. This means that you don't use a `MAKE` command to get an `InputBuffer`: you ask a `Keypad` for one. It's done like this because an `InputBuffer` object is intimately connected to the 'host' `Keypad` it operates with.

To create the simplest form of `InputBuffer`, use the following:

```
keys_in := 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. In future you will be able to specify the buffer length with a third parameter to 'queue up' key presses. The default buffer length is 1 key. Any key presses detected while the buffer is full are ignored.

### Updating an InputBuffer

InputBuffers have to be 'updated' in order to do their work. This is achieved by sending the message `Update` on the host `Keypad`. `Update` has to be sent both often and regularly for the `InputBuffer` to work well: every 30mS seems to be a good rate.

112

```
EVERY 30
[
    kpd . Update
    key_press := keys_in . Key ; Key pressed?
    IF key_press >= 0
    [
        SELECT CASE key_press
        … select the action for each key.
    ]
    IF key_press >= 0 ;Only update display if key pressed
    [
        … update the 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
```

## Getting Key Presses

To get keys from the `InputBuffer` you can use either `Get` or `Key`.

`InputBuffer.Key` will return any key in the buffer, or –1 if there is no key
press.

`InputBuffer.Get` will loop until there is a key to return.

# NUMBERREADER

`NumberReader` allows you to set up a keypad to enter numbers into your application calculator-style.  You tell it which keys represent the numbers, minus sign, decimal point and so on, and it will assemble numeric input for you.

## Creation

```
MAKE nrd NumberReader(@kpd,0,6)
```

`Kpd` is the `Keypad` you are reading from, and 0 is the type of `NumberReader` – in this case, *decimal*.  The third parameter is the *width*.  It sets the maximum number of digits allowed in the number being assembled.

## Conversion

The `Conversion` message tells the `NumberReader` which keys on your `Keypad` to use for its functions.

```
nrd . Conversion(-1,11,3,15,12,  13,0,1,2,4,5,6,8,9,10)
```

The first five parameters are the key numbers for the ENTER, DECIMAL POINT, MINUS, DELETE and CANCEL functions.

*Decimal Point* will make the number floating point.  It will only respond if the number doesn't have a decimal point already.

- *Minus* is for entering negative numbers.

- *Delete* will delete the number you have entered one character at a time.

- *Cancel* will cause the number to reset to its default value, or zero if you haven't set up any other default.

- *Enter is only needed if you use* `NumberReader.Get`*, which we don't cover here.*

The rest of the parameters are the keys numbers for the digits 0 to 9.

This conversion list above has been created for the this keypad:

114

```
;Actual Legend on key
;1  2  3  F
;4  5  6  E
;7  8  9  D
;A  0  B  C

;Logical Key number
;0  1  2  3
;4  5  6  7
;8  9  10 11
;12 13 14 15

;Our Function; X is 'not used'.
;1   2  3  -
;4   5  6  X
;7   8  9  .
;C   0  X  Del
```

*If you don't want the* NumberReader *to perform one of the functions, use the*
*value -1 in the appropriate key position.*

## Reading Numbers

There are several ways to read a number, but the easiest to understand uses Put.
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.

```
TO main
  LOCAL key_pressed
  PRINT TO nrd,"100.0"
  PRINT TO lcd, HOME,nrd
  FOREVER
  [
    key_pressed := kpd.Get
    nrd . Put (key_pressed)
    PRINT TO lcd, HOME,nrd
    IF key_pressed = 7 BREAK
  ]
  PRINT "The value is: ", nrd.value,cr
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 *Venom-SC Object Reference*.

# ONBOARDLED

The `OnBoardLED` object is used to control the LED on the controller board. The LED output is usually 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 startup is determined by the Venom runtime system. As soon as Venom starts running your code you have control over the LED.

In general you should use the LED to signal the status of your application. The startup procedure sets the default status in RUN MODE to be a flash 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 on continuously then it is waiting at the `Clear Memory` prompt in PROGRAM MODE. If the LED is flashing then it is running the application.

## Messages

The LED behaves rather like a `Digital` object and will obey many of the same messages: `On`, `Off`, `Asserted`, `Value`, `PRINT` and so on.

## Flashing

The LED can be made to flash:

```
led . Flash(10)
```

| Flash Number | Description |
|:---:|---|
| 0 | Off |
| 1 | On |
| 2 - 9 | --Not used yet-- |
| 10 | Short flash approximately once a second |
| 11 | Two short flashes approximately once a second |
| 12 | Three short flashes approximately once a second |
| 13 | Signals "SOS" in Morse Code |
| 14 | Signals "OK" in Morse Code |
| 15 | --Not used yet-- |

# 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[53].

## Diagram

The diagram below may make the operation of the buffer easier to envisage.
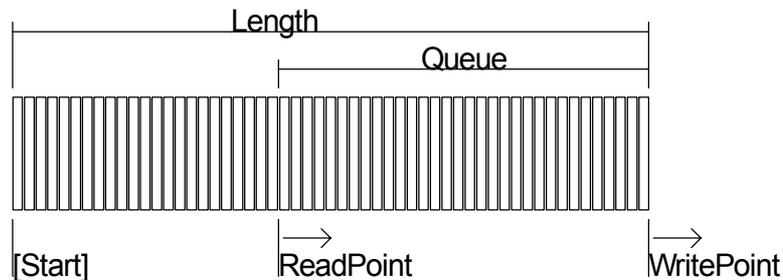


Data is written into a buffer at the write-point.  When data is written, the buffer grows to the right of the diagram.

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'.

Each buffer can only hold one type of data; you can't mix data types in one buffer.

To create a buffer, you need to indicate the data it is to hold:

---

[53] See ARRAY for fixed-size and 'constant' data storage.

```
MAKE b8 Buffer(8) ;holds 8-bit integer data
MAKE b16 Buffer(16) ;holds 16-bit integer data
MAKE b32 Buffer(32) ;holds 32-bit integer data
MAKE bf Buffer(1.0);holds floating point data
MAKE bt Buffer("")   ;holds text⁵⁴
```

**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:

---

[54] Putting text between these quotes would initialise the new buffer with that text.

```
-->b8.flush
-->PRINT b8
      6
      7
-->
```

Here, the elements up to the read-point have been removed, leaving the unread elements in the buffer.

### Random access

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.

### 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 |

## 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.

120

A text buffer is very 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.

## 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 bt Buffer("")
-->PRINT TO bt , "This is some text"
-->PRINT bt
This is some text-->
```

You can also initialise the text in a buffer at creation:

```
-->MAKE bt Buffer("Initial text")
-->PRINT bt
Initial text-->
```

When you print to a text buffer, the new text is appended on to any existing text in the buffer:

```
-->PRINT TO bt, "more text"
-->PRINT bt
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 – see page 34.

```
-->REPEAT 5 PRINT bt: INDEX0 : 5,cr
Initi
nitia
itial
tial
ial t
```

As with string constants, 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 bt
Initial textmore text-->
-->bt.Insert("XXX",5)
--> PRINT bt
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 := bt . 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 := bt . 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.

## 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'[55].

---

[55] The *heap* is the memory made available in the controller for general purpose use.

# 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

Storing tables of *non-volatile variable* data in the battery-backed RAM.

## Creating Constant Arrays

*Because it holds data that is constant during an application, an* `Array` *of constant data is rather like a procedure. It is created on the command line with its own special syntax rather than with* `MAKE` *or* `NEW`*.*

The following code creates an `Array` of 10 8-bit integers.

```
-->ARRAY ar8(8,10)
02>1,2,3,4,5,6,7,8,9,10
03>END
ARRAY Defined (22 bytes @ $26025E)
```

The integer parameter '8' is a prototype, indicating to the `Array` that it will be storing integers, specifically 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(16,10) … END
-->ARRAY ar32 (32,10) … END
-->ARRAY ar_float(1.0,10) … END
-->ARRAY ar_ptr(@dummy,10) … END
-->ARRAY ar_str("",10) … END
```

### 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 prototype looks like, as you would expect, a pointer. I have used a dummy variable to make it clear that it doesn't matter what the prototype points to. You must use the '@' symbol, but you can have any name you like.

## Array of strings

Arrays of string constants may not seem straightforward at first. Here's an example of their use.

```
-->ARRAY ar_str("",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
```

124

## 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 (8 , 10 ,1,2,3,4,5)
```
OR
```
a := NEW Array (8 , 10 ,1,2,3,4,5)
```

Here the prototype and size are 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.

### Non-Volatile Variable Arrays

Sometimes it is useful to have data in an application that is variable, but does not lose its value each time the application powers down. Non-volatile Arrays are one way to store this kind of data.

Non-volatile Arrays are always created by copying another `Array`, usually a constant `Array`. This special `Copy` is indicated by the '1' parameter.

$$\textbf{obj . Copy (1)} \implies \texttt{Array}$$

Here, instead of a complete copy being taken, just the information on the size of the `Array` and the kind of data it holds is copied.

The copy is created in a special part of the controller's RAM called the NV RAM area. This area of RAM is not initialised by the system when it powers up, so the data is not destroyed.

Because it is possible for non-volatile data in the RAM to be corrupt, special messages are available to check the data integrity.

`Checksum` will return a 16-bit checksum of all the data in the `Array`, and put a copy of it in the `Array` header.

`Valid` will returns TRUE if the checksum is correct.

Reset will copy data into the non-volatile Array, from its parent Array, to initialise it.

Note that in order for the new non-volatile Array to be located at the same address each time the controller resets, the Array.Copy(1) messages and any other functions that use the NV-RAM (like the RAM filing system) should occur *once only, and in the same order*. The best way to achieve this is to put them in the init routine, and then always run your code by typing 'Run' at the command line, or by powering-on in RUN MODE.

```
TO init
   Free (2) := 10000 ;enough NV-RAM for my n.v. Arrays
   nv_array1 := array1 . Copy(1)
   nv_array2 := array2 . Copy(1)
   nv_array3 := array3 . Copy(1)
   …
END
```

If there is not enough NV_RAM to create the copy, then a 'Resource Error' is given.

*See* OperatingSystem.*Free for more information.*

126

# REALTIMECLOCK

`RealTimeClock` keeps track of dates and times in the real world using the PCF8583 Real-time clock/calendar IC on an I$^2$C bus.

## 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.

## 'Venom Seconds'

`RealTimeClock` stores the time as the number of seconds that have elapsed since the base date: midnight on 1$^{st}$ January 1990[56]. 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 setting the time and reading it back.*

## Printing the Date and Time

It may be more useful to see this time translated into a date and time in the normal calendar form. You can `PRINT` the clock to get this.

```
-->PRINT clock
01-Jan-90 00:02:12-->
```

The clock can print itself in formats to satisfy most tastes…

| Format | Example |
|--------|---------|
| : 0 | 04-Jan-02 09:48:12 |
| : 1 | 04-Jan-02 09:48 |
| : 2 | 04-Jan-02 |
| : 3 | 09:48:12 |
| : 4 | 09:48:12 am |
| : 5 | 04-01-02 |
| : 6 | 01-04-02 |
| : 7 | Fri |
| : 8 | Jan |

---

[56] Roughly when the original *Venom* language was born.

## 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 := 2002
now.month:= 4
…
Clock.Time := now.Time
```

You can also set the clock by printing *to* it, which may be easier:

```
-->PRINT to clock , "4-1-2002 9:48:12"
```

Printing to the clock must obey these rules:

- The date comes first

- You must use ':' to delimit the time

- You must provide all six elements of the date, using numbers

The clock can accept just about any separator for the date.  It can also work with a 'short' year, e.g. "4-1-02".

## 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` 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.

## Clock not set

If the clock IC is not present, or the clock has not been set (or has lost it's setting) then it will report Time as zero seconds, and print the time and date with '-' or '=' characters instead of numbers so that an invalid time and date are never displayed.

## Accuracy

The clock's 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.

**Date Extent**

The `RealTimeClock` can work with dates up until the year 2090, at which point a software upgrade will become necessary[57].

**Alarm**

The real time clock IC has an alarm output that may be set to 'go off' at any time up to a year ahead. The message `SleepUntil` controls the alarm setting. When `SleepUntil` is called, the alarm output is turned off immediately and will turn on again at the specified time. If you connect the alarm output to a suitable circuit[58] then your application electronics will be able to 'sleep' consuming zero power until a specified time in the future. The messages `On` and `Off` control the state of the alarm output directly.

```
; Turn the alarm output on to hold PSU on:
clock . On
; Sleep for a minute:
clock . SleepUntil(clock.Time + 60)
WAIT 1000          ;wait for power off.
PRINT "Can't sleep!" ;should never see this message.
```

Don't set the alarm to less than two seconds, or more than a year, into the future.

The alarm state is not affected by anything other than `SleepUntil`, `On` and `Off`. When a clock IC is first plugged in, the alarm output will pulse at 1Hz until it is accessed by one of the above messages.

---

[57] Put this date in your diary now! ☺
[58] See the datasheet for your controller for suitable circuits

# DATETIME

`DateTime` is primarily a date calculator.

`DateTime` performs the complex calculations that translate between the calendar-date/time-of-day that we are familiar with, and Venom Seconds[59].

The diagram illustrates how it works.
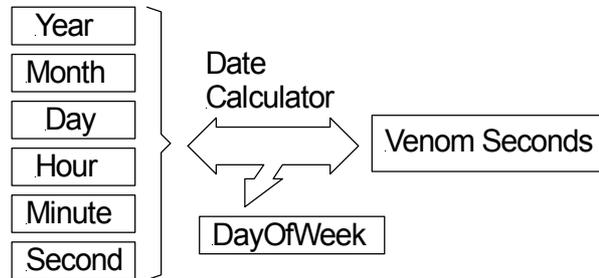


If the Venom seconds value changes, then the date and time values will be updated to reflect it. If any one of the date or time values changes, then the Venom seconds value will be updated. The DayOfWeek value is purely a function of the date, and so cannot be altered directly.

### Creation

You can make as many `DateTime objects` as you want, though each one uses a small amount of memory.

```
MAKE when DateTime
```

### 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 `Nudge`[60] message to set it like a digital watch.

The code below illustrates these

---

[59] The number of seconds since the beginning of 1990.
[60] A full treatment of the Nudge message is given in the Object Reference Manual.

```
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. See Page 128.

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
```

### Printing a DateTime

Printing a DateTime uses exactly the same formats as the RealTimeClock. See page 127.

### 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 message is always four figures.

### 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.

## 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 Venom-SC Object Reference.

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`.


You can fix up spurious dates by sending the `Update` message.

```
   when . Update
```
This is the equivalent of
```
  when . Time := when . Time
```

# 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.

## 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 (0-24) |
| H is an hours digit (0-23) |
| M is a minutes digit (0-59) |
| S is a seconds digit (0-59) |

133

# 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.

134

# ASYNCHRONOUSSERIAL

`AsynchronousSerial` objects control serial communication ports. Much of the time you will be using it by default as it is the default output device for `PRINT`.

Handshaking is optional: hardware ('RTS & CTS'), software (XON / XOFF) or none.

`AsynchronousSerial` objects interface to serial communication ports. The two hardware ports can operate at standard rates up to 38400 baud, or higher non-standard rates.

No 'software' serial ports have been implemented to date.

## Creation

Here we create a serial communication object on port 1:

```
MAKE serial AsynchronousSerial(38400,1,1)
```
*An object like this is created by the default startup routine at either 38400 or 9600 baud, depending on the state of the User Switch.*

## 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[61].

---

[61] `AsynchronousSerial` takes many other messages – see the Venom-SC Object Reference 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
```

Because of a shortcut in Venom, any message seen as a command by itself will be sent to `system`. Thus the two lines below are equivalent, and will reset the controller.

```
-->system  . Reset
-->Reset
```

## Operating System Messages

Here some of the most useful operating system messages are described. The rest are documented in the *Venom-SC Object Reference*.

### 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.

### 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.

### 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.

**Protect**

This deals with protecting your application program in Flash memory, also known as 'ROMing'.

In the VM-1, application code is stored in the same flash as the Venom system code.

*The following applies to the VM-1, but may not necessarily apply to all Venom-based controllers. See the Datasheet for your controller.*

When you have fully developed your application, you should copy it to flash. The table below details the `Protect` message.

| Code you type | Action |
|---|---|
| Protect (0) | Erases the Application Area of the Flash |
| Protect (1) | Programs the application code into the Flash |
| Protect (2) | Report the Application Area usage |
| Protect (4) | Returns the device ID code, or 'Silicon Signature' of the flash device |

*Later more options may become available in the* `Protect` *message.*

In order to program or erase the flash, the write-enable link (LK3) must be fitted to the VM-1. If for any reason the operation failed, the `Protect` message will return zero. You can check you have write-access to the flash:

```
-->PRINT ~protect(4)
    1A4-->
```

This would be zero if a valid, write-enabled flash was not detected.

Typing 'Y' at the `Clear Memory` startup banner will *not* erase applications in flash memory.

Once you have programmed the flash memory, you may want to remove the write-enable link to prevent accidental erasure of your code.

*You may copy the resulting combined Venom/Application flash device as many times as you like, so long as it is only used in equipment manufactured by Micro-Robotics Ltd, or manufactured under licence of Micro-Robotics Ltd. The licence to use the Venom Language and Operating System is included in the purchase of the controller.*

*If you have a Micro-Robotics product with a flash programmer in it (such as the Application Board 2 (5805) then you may use that to make copies of your combined application.*

## Free

This returns the amount of general-purpose *heap* memory left in the controller.

```
-->PRINT Free
 99466-->
```

It will also report on other areas of the controller's memory.

| | |
|---|---|
| Free (0) | Heap memory free |
| Free (1) | Largest free block in the heap |
| Free (2) | Total NV RAM[62] size |
| Free (3) | NV RAM free |

The size of the NV RAM area may be set using

```
 Free (2) := SIZE
```

If you change this size Venom has to reset itself, as it is a major upset internally. See *Appendix D:* for how and when to use this.

## 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.

## UserSwitch

On the VM-1, a switch is used to select the baud rate the controller uses when it starts in Program Mode. This switch is not limited to that function, and may be read with the `UserSwitch` message. If the switch is ON it returns `TRUE`.

## RunMode

This returns `TRUE` if the controller is in Run Mode, whether because the Program Mode switch was off, or because you used the `Run` message.

---

[62] NV RAM is an area of the controller's non-volatile RAM that is partitioned off for use by a RAM filing system or other systems.

**ErrorAction**

If set to the value 1, this message will restart the Venom system if a runtime error occurs.

```
ErrorAction := 1
```

This is essential to add for robust applications, but is just annoying during development. For this reason the default startup procedure sets `ErrorAction` to one (1) if the *Program Mode switch* is off, i.e. you are running your application for real.

Note that *Ctrl-C break* is considered a runtime error, so you won't be able to break into your application when `ErrorAction` is set.

Later, more flags may be added to this value to provide a more sophisticated response.

# 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. Obtain the Venom-SC *Language Reference* and *Object Reference*. These will be required for most serious applications.

4. Using development hardware, write key sections of your application to make sure that they are viable.

5. Design and build the application hardware in conjunction with the controller's datasheet and example circuits.

6. Write the complete application program.

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` on page 137. You may make copies of this Flash device so long as they are used in Micro-Robotics Ltd products, or products licensed by Micro-Robotics Ltd.

10. Go into production with the application hardware and the application Flash.

    *You don't need to buy an expensive Flash programmer to make copies of your master flash device – your development kit can do this – much faster than many programmers.*

# B: How Do I … ?

This 'F.A.Q.' section deals with how to achieve solutions to commonly encountered problems using the Venom-SC language and object types.

## Store Non-Volatile Data

The following objects can handle non-volatile data:

- SafeData: useful for storing parameters or calibration data for your application. Very safe as it's stored in EEPROM. Validation using checksum.

- FileSystem: stores data and text files, only in non-volatile main RAM currently. Not so safe as a battery failure or processor crash can erase it.

- `Array`: Store data either in the Flash (i.e. not alterable except at application creation time) or in the NV RAM (non-volatile main RAM). Not so safe as a battery failure or processor crash can erase it. Validation using checksum.

## Manipulate Text

Use the text `Buffer` object to manipulate text.

- `PRINT TO` the buffer to append text

- `PRINT` all of the buffer, 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

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 the `Keypad` object for entering data, or for navigating menus. It is also possible to drive touch screens.

## Time events

- Use `Stopwatch` and `Timer` for millisecond timing of events and sequences.
- Use `WAIT`, `AWAIT`, `EVERY` for providing timing in your application.

## Talk to serial devices

- Use `AsynchronousSerial` for RS232 and RS485 communications
- Use I2CBus for I$^2$C Bus devices
- Use SerialIO 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 0-5Volt signal and read this using one of the on-board 10-bit analogue inputs on the VM-1.
- Use a precision thermistor bead and a resistor in a potential divider, feed the voltage into the on-board 10-bit analogue inputs (0.2°C accuracy). We can supply a linearisation function to convert ADC readings to temperature.

144

- Use an external 12-or-more-bit analogue to digital converter for more accuracy. Suitable devices will interface to the I2CBus and SerialIO objects.

## Sleep with Zero Power

- Use the `RealTimeClock` alarm signal to control your application's power supply (see circuit available from Micro-Robotics Ltd). Use the `On`, `Off` and `SleepUntil` messages to control the alarm signal. You may need an override switch to turn power on if the alarm is not set, or to power up early.

## Use Files

- Use the `FileSystem` object to create files in RAM.

145

# C: SPEED OF EXECUTION

Venom-SC 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 7µS to 15µS on the VM-1. A bit of code like `a := a + 1` will take ~50 µS.

## Measuring Execution Times

The following code allows you to measure the execution time of bit of Venom code.

```
TO measure_time(n,c)
LOCAL stop_watch,t
  stop_watch := NEW StopWatch
  stop_watch.Reset
  REPEAT n
  [ ;commands to be timed
  ]
  t := stop_watch.Time AS FLOAT
  PRINT (t / n - c):10:4, " milliseconds", CR
  stop_watch.Die
END
```

The parameter `n` is the number of times the loop is repeated - increasing it increases the accuracy of the result. The parameter `c` is a constant adjustment that is used to take into account the time taken to execute the `REPEAT` command.

Firstly, the procedure should be run with `n` = 1000; `c` = 0 and the `REPEAT` command empty. This will then print the value to use for `c`.

Then put the code under test into the `REPEAT`, choose a value of `n`, and use the value of `c` you just found.

# D: ROBUST APPLICATIONS

## 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 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 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 Escape function, as this could potentially halt an application. Ctrl-C Escape is treated as a runtime error, so if ErrorAction is set, the Venom application will be restarted. To turn off Ctrl-C Escape, use

```
Serial.Escape := FALSE
```

## Heap size

The memory used by the system is mostly held in an area called the *heap*. There is also another area that is chiefly used by the RAM filing system called the non-volatile area or *NV RAM*. You may choose the size of this area. If you don't have a RAM filing system, or other non-volatile storage objects, then a sensible size is zero as the NV RAM subtracts from the space available to the heap. It is possible for the partition between the heap and the NV RAM to be moved accidentally[63]. To guard against this it is wise to include a line early in your init procedure to explicitly set the NV RAM size:

---

[63] The memory manager will always preserve a minimum amount of heap memory, to ensure it's able to start up. Currently this is ~20K.

```
Free (2) := 0 ; or what ever size you need.
```

It's best to use a constant value for the size: the controller has to reset itself if the partition moves. See `OperatingSystem.Free` in the *Venom-SC Object Reference* for more background information.

## 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 in ROM or **R**ead **O**nly **M**emory. That is, your application code should be programmed into some more permanent device, like a Flash memory. Once the application is in Flash memory, and the write-enable link is removed, then the application can't be accidentally erased.

See the system message `Protect` on page 137.

## Watchdogs

A watchdog is a hardware device that has control of the reset input to the controller. If the program does not 'kick' the watchdog every so often, then the watchdog will reset the controller. This is to halt and restart a crashed micro-controller.

In the VM-1 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. A system message will be implemented later to take care of this.

## SUMMARY

- Always 'ROM' your application code.

- Something like the following lines should appear near the start of any Venom-SC application released into the field. Some of these will have been taken care of by the default `startup` procedure – `LIST startup` to find out.

```
ErrorAction := 1        ;Restart on errors
Free (2) := 0           ;Ensure adequate heap
Serial.Escape := FALSE  ;Disable Escape
```

# 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.

| 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 | ~ |
| 31 | 1F | | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | DEL |

# 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 Venom-SC 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[64].

```
a := 5 * 4
a := 20
```

When Venom compiles these lines of code, it is able to notice the possible optimisation, and compiles as if the second line had been written.

Constant folding is performed on most operations. In order to ensure folding happens, enclose the operations in parentheses:

```
5 * a * 4   ;will not be folded (the compiler's not that
clever!)
5 * 4 * a   ;might be…
a * 5 * 4   ;might be…
a * (5 * 4) ;definitely will be.
```

## Power saving

The Venom operating system automatically uses the SLEEP 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 SLEEP instruction.

In order to make best use of this, make your tasks wait if they can do so without compromising the responsiveness of your code.

For example you could wait for a digital input like this:

---

[64] *Maintainable* means it is understandable by someone else, or at a later date.

```
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.

# G: CALLING FOREIGN CODE

You can call routines compiled from C or Assembler from Venom using the CALL keyword.

CALL takes any number of parameters, but the first one must be the address of the code to call.

```
CALL (address_of_my_code)
```

Your called code should expect to receive one parameter of equivalent to a void* in C. This is a pointer to the Venom stack containing all the parameters to CALL, except that the address parameter has been changed to indicate the number of parameters sent to CALL. This first parameter position is also the place to put the return result. Note the Venom stack grows *upwards*, i.e. it starts at low memory.

```
CALL (address , A , B , C)
```

Would put the following on the venom stack

```
    C
    B
    A
→ 4 ;N Params - Put return result here.
```

Each Venom value on the Venom stack is 8 bytes wide. The first 4 bytes are the value (MSB first on a 'big-endian' processor). The other 4 bytes hold the type and write-protection information (there is also some reserved space).

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

For example, say you wanted to call a C routine that multiplies two numbers and sends the result back. See below for the definitions used.

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

Here are some definitions used with Venom variables. There is the possibility that these will change slowly over time, so check they are up to date if you need to use them.

They will be made available on our website.

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

#define VENOM_TYPE_INT 0
```

## Task Swap Timing

Internally the Venom task manager relies on each task electing to 'swap out'. So as not to contravene the task manager's rules your foreign code should not take more than 2mS to complete.

An operating system call to the task manager to swap tasks will be made available later so your code can take as long as it needs to complete.

# H: STARTUP SEQUENCE

The diagram shows what happens when Venom-SC starts.

```
                          ╭─────────╮
                          │  Reset  │
                          ╰─────────╯
                               │
                        ┌──────────────┐
                        │ Internal init│
                        │ & Checks     │
                        └──────────────┘
                               │
        RUN MODE          ◇ ◇ ◇ ◇ ◇      PROG MODE
                  ◇─────────────────────────◇
                 /                            \                Clear Memory Y N S? __
    Application ◇                              ◇ ◇ ◇
    in ROM?    / \                       Y   /       \
              /   \          ┌────────────┐ /         \  N, S
   ┌──────────┐ ┌──────────┐ │Wipe main RAM│           ┌──────────┐
   │Wipe main │ │Validate  │ ├────────────┤            │Validate  │
   │RAM       │ │code      │ │Report RAM  │            │code      │
   ├──────────┤ │in RAM*   │ │size        │            │in RAM*   │
   │Read App  │ └──────────┘ └────────────┘            └──────────┘
   │From ROM  │                    │
   └──────────┘         Application◇
                        in ROM?  / \
               ┌────────────┐   /   \
               │Read App    │◇◇◇
               │From ROM    │
               └────────────┘
                        │
               ┌────────────┐
               │Create Default│
               │Startup etc. │
               └────────────┘
                        │         Clear Memory was
                        ◇         'S' for 'Skip Startup'?
                       / \
   ┌──────────────┐  ┌────────────┐
   │Run 'startup' │  │Create some │
   │... and       │  │Default     │
   │application   │  │Objects     │
   └──────────────┘  └────────────┘
                 │
         ┌──────────────┐
         │Command line**│
         │-->           │
         └──────────────┘
```
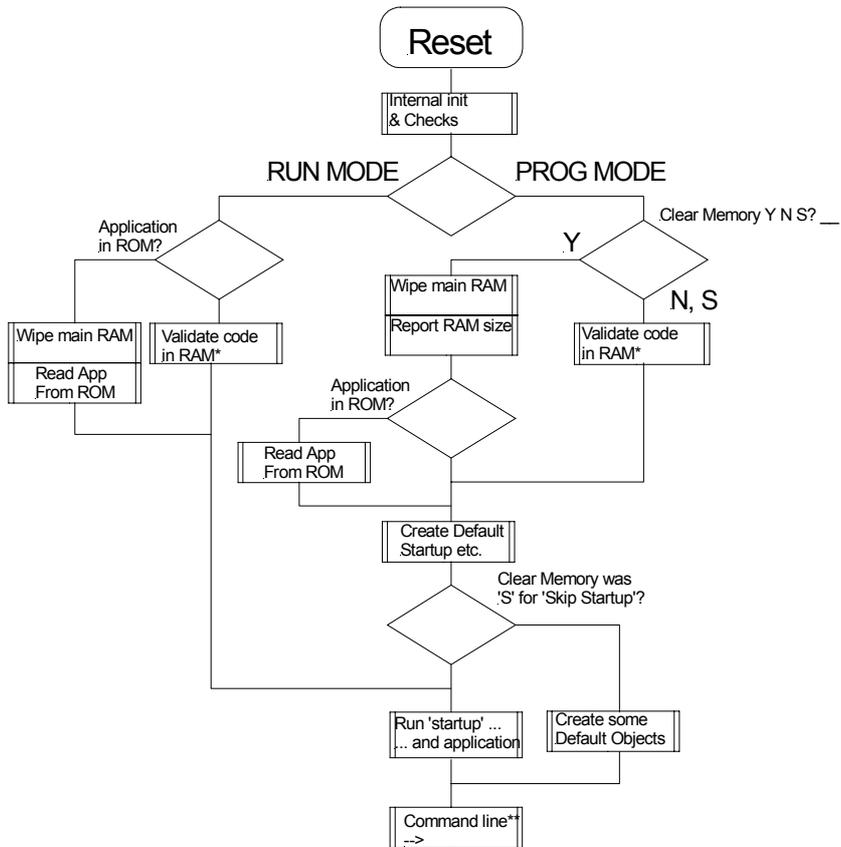
*'Validate code in RAM' means check that the controller's heap-memory, global variables etc. contain valid data.  If not, then reset them.

**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.

154

# Index