



Control-C Version 2p06

Programming Language

Detailed description

Contents

Overview.....	4
Control-C and System software version control.....	4
Approach.....	4
Feedback.....	4
What is Control-C.....	5
The Syntax Diagram.....	5
Program Structure.....	6
Constants.....	8
User Defined Constants.....	8
BUILTIN Constants.....	8
Variables.....	11
Variable Format.....	11
Variable Names.....	11
Arrays.....	11
User Defined Local Variables.....	12
User Defined Global Variables.....	12
Pre-Defined Global Variables.....	13
Accessing Global Variables.....	29
Assignment Order.....	29
Broadcast Variables.....	30
Functions.....	31
main().....	31
Error Handler.....	32
errorh().....	32
Interrupts.....	32
Interrupt Control.....	32
Selecting interrupt input polarity.....	32
Enabling Interrupts.....	33
int0h() - int9h().....	33
uarth().....	33
msggh().....	33
Encoder Position Capture Interrupt.....	34
Encoder capture polarity.....	34
Enabling Encoder Capture.....	34
Registration Window.....	34
Input[0] interrupt filtering.....	34
Statement.....	35
Assignment operators.....	35
putch.....	35
puts.....	36
if.....	36
delay.....	36
beep.....	37
return.....	37
while.....	37
for.....	38
cam - electronic cam - (queued command).....	39
setuart.....	40
break.....	40
do.....	40
va - vector absolute - (queued command).....	41
vr - vector relative - (queued command).....	41
am - absolute move - (queued command).....	42
rm - relative move - (queued command).....	42
vm - velocity mode - (queued command).....	43
tm - torque mode - (queued command).....	44
pm - position mode - (queued command).....	45
output.....	46
status.....	46
fram.....	47
ints.....	47

gr - gear lock mode - (queued command).....	48
pulse.....	50
spi.....	50
dot.....	50
line.....	51
rectangle.....	51
block.....	52
circle.....	52
disc.....	53
cursor.....	53
edges.....	54
key.....	55
sputch.....	56
sputs.....	56
clear.....	56
bitmap.....	57
vartos.....	57
String.....	58
Expression.....	59
Factor.....	60
! (not).....	60
- (negate).....	60
~ (ones complement).....	60
getch.....	60
charin.....	61
getmsg.....	61
msgin.....	62
(EXPRESSION).....	62
CONSTANT.....	62
DEFINED.....	62
FNAME.....	62
GLOBAL.....	63
LOCAL.....	63
PREDEFINED.....	63
input.....	64
output.....	65
status.....	65
fram.....	66
ints.....	66
sin.....	66
asin.....	67
cos.....	67
acos.....	67
tan.....	67
atan.....	68
sqrt.....	68
varnum.....	68
beep.....	69
keypressed.....	69
getkey.....	69
xpos.....	69
ypos.....	70
stovar.....	70
edges.....	71
Appendix 1 - The Motion Command Queue.....	72
Motion Commands And The Motion Command Queue.....	72
Move Synchronisation.....	72
Appendix 2 - Error Code Definitions.....	73
Appendix 3 - Glossary Of Terms.....	76

Overview

This document provides a detailed technical description of Control-C the programming language. It is an embedded feature of the Control-C Integrated Development Environment and is used for programming the E-node range of network embedded industrial control modules.

This document is not intended to teach the reader about 'C' the language nor how to write good programs. There are many good books available on these subjects. If the reader is new to programming please read one of these before reading this document.

A free copy of Control-C together with installation instructions can be downloaded from www.etrol.co.uk .

Control-C and System software version control

Control-C and the E-node system software are regularly updated with improvements and corrections. Each update has a unique version number in the form XpXX added to the file name, e.g.

ControlC2p00.EXE	The main Control-C program (windows executable file)
Enode2p00.BIN	The E-node system software (binary file)

► *For correct operation the Control-C executable file and the E-node system software file must both have the same version number and they must both reside in the same directory.*

Approach

At Etrol Ltd we like to 'Keep It Simple' or 'KIS'. You will find this theme runs through everything we do and is reflected here in this document.

At the heart of Control-C lies the Syntax Diagram. This document details the exact layout requirements for any Control-C program. We urge any reader to take the time required to understand and interpret the Syntax Diagram. Both the language compiler and this document follow its content.

Feedback

If you find any errors, omissions, or have any suggestions to improve this document please email us at feedback@etrol.co.uk.

What is Control-C

Control-C (the language) is a programming language created by Etrol Ltd to program the E-nodes, its range of network embedded industrial control modules. It is based on the industry standard language 'C' as defined by 'Kernighan & Ritchie' but simplified to make it easier to use and remember. It is also enhanced to provide seamless communications and has a suit of commands for the provision of dedicated functions, e.g. motion commands.

Control-C (the language) forms an integral part of the Control-C Integrated Development Environment (IDE). This is a personal computer based program running under Windows XP/Vista providing every facility required for the programming and set-up of E-node systems. A separate document is available detailing the IDE.

The Main features of Control-C are:-

- **Simplicity** - All the power and flexibility of standard 'C' without the complexity
- **Speed** - Fully compiled for fast compact code
- **No pointers**
- **Only One variable type**
- **Easy to learn** - With a huge body of existing educational texts
- **Embedded communications** - All commands fully network enabled
- **One language for all Node types** - No matter what function the node performs
- **Dedicated commands** - For Motion Control, I/O read/write and more
- **Interrupts** - Supporting I/O and Communications state changes

The Syntax Diagram

The Syntax Diagram is a graphical document that exactly describes the flow and construction rules for any Control-C program. The compiler will stop with an error if the user program deviates from it in any way.

The Syntax Diagram is available on www.etrol.co.uk. Its version number is tied to that of Control-C and it forms a vital companion to this document which generally follows its content from beginning to end.

The Syntax Diagram is hierarchical in that it firstly details the topmost level of the program structure. It then steps down through the various levels to the lowest. This detailed description will generally follow the structure of the Syntax Diagram as you work through the document.

▶ *The Syntax Diagram may at first sight seem complex and to be contrary to Etrol's philosophy of 'Keeping It Simple - KIS'. Please persevere, it is a simple document at heart and when understood it will become your constant companion and all you need for reference when writing programs.*

▶ *As you work through this document there will be many examples to assist in the understanding of the Syntax Diagram.*

Throughout the rest of this document the 'Syntax Diagram' will be referred to as the 'SD'.

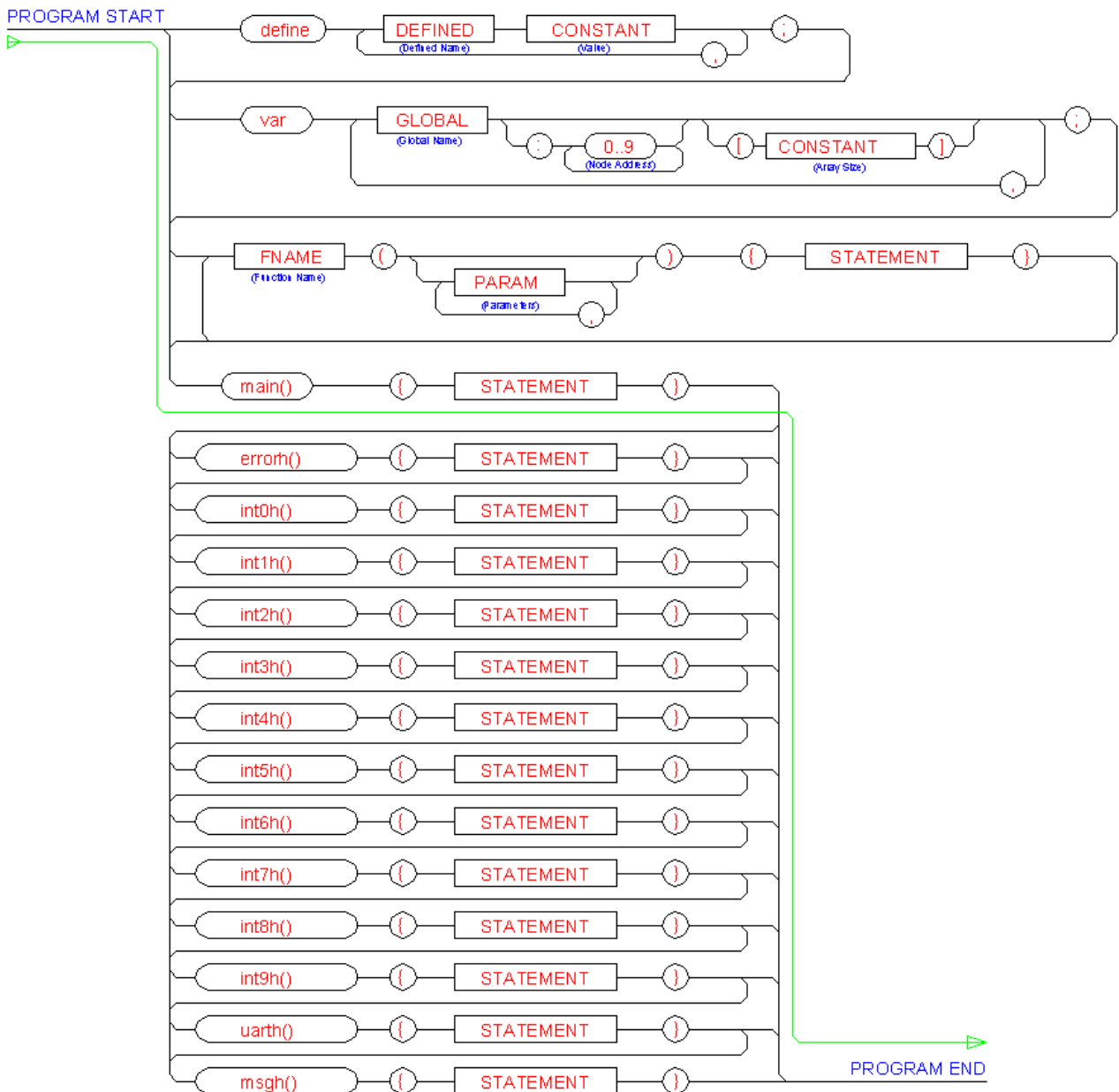
Program Structure

Here is an example of the simplest possible program that will compile correctly and run successfully on an E-node.

Example:

```
main() // this is the start point when the program is run
{
    // the main program goes here
}
```

This program does nothing but it can help to understand the SD (Syntax Diagram). From 'PROGRAM START' follow the line until you reach 'main()'. The box style indicates that 'main()' is a mandatory key word that must be a part of the program and typed exactly as shown. Moving on you then come to '{' which is also mandatory. Then comes 'STATEMENT' in an oblong box. This shows that you need to refer to the 'STATEMENT' section where the possible combinations of a statement content are illustrated. The first route that can be taken is a simple horizontal line indicating that a statement can in fact contain nothing, as in this case. Next comes '}' and then finally 'PROGRAM END'.



Here is a second example containing a constant definition, a global variable definition a subroutine and the main program that calls the subroutine.

```

define
  a 100;           // constant definition

var
  b;              // global variable definition

my_subroutine()  // a subroutine
{
  b = a;         // b = 100
}

main()         // this is the start point when the program is run
{
  my_subroutine(); // call the subroutine
}

```

The path through the SD for the above is:



Constants

Constants come in Two forms, 'User Defined' constants' and 'BUILTIN' constants. All constants have an associated numerical value. Wherever the compiler encounters a constant in the program its numerical value is substituted.

User Defined Constants

User defined constants are declared at the top of the program using the keyword 'define' followed by the required definitions.

Example:

```
define
  a 100,           // a is replaced by 100
  b -2,           // b is replaced by -2
  c 1.8,          // c is replaced by 1.8
  d 0x5.6,        // d is replaced by 5.6
  e 0xFFEE,       // e is replaced by 65518
  f TRUE,         // f is replaced by 1
  g 'A';          // g is replaced by 65 (from ASCII table)
```

- ▶ Constant names must conform to the syntax outlined in the 'IDENT' section of the SD.
- ▶ Constant values must conform to the syntax outlined in the 'CONSTANT' section of the SD.
- ▶ Constants are case sensitive where 'top' and 'TOP' are two different names.

BUILTIN Constants

BUILTIN constants are a list of pre-defined values provided for manipulating and testing the integer term of variables. Details are given in the 'BUILTIN' section of the SD. Their meaning, value and suggested use is given below.

<u>Name</u>	<u>Hex Value</u>	<u>Meaning</u>	<u>Use</u>
INACTIVE	0x00000000	inactive	testing 'activity' variable
IDLE	0x00000001	move idle	ditto
PACC	0x00000002	+ve acceleration	ditto
PVEL	0x00000003	+ve max velocity	ditto
PDEC	0x00000004	+ve deceleration	ditto
NACC	0x00000005	-ve acceleration	ditto
NVEL	0x00000006	-ve max velocity	ditto
NDEC	0x00000007	-ve deceleration	ditto
PTOR	0x00000008	+ve torque	ditto
NTOR	0x00000009	-ve torque	ditto
PMOD	0x0000000A	position mode	ditto
PLIM	0x0000000B	at +ve position limit	ditto
NLIM	0x0000000C	at -ve position limit	ditto
GRLK	0x0000000D	gear locked	ditto
BRK	0x0000000E	braked	ditto
HALTED	0x0000000F	halted	ditto
CAMMING	0x00000010	in cam mode	ditto
RATCHED	0x00000011	in ratchet mode	ditto
TRUE	0x00000001	true	general use
FALSE	0x00000000	false	ditto
ON	0x00000001	on	general and I/O testing
OFF	0x00000000	off	ditto
POS	0x00000001	positive	setting the 'ratchet' variable

<u>Name</u>	<u>Hex Value</u>	<u>Meaning</u>	<u>Use</u>
NEG	0x00000002	negative	setting the 'ratchet' variable
RS232	0x00000001	RS232	selecting uart mode in 'setuart' statement
RS485	0x00000000	RS485	selecting uart mode in 'setuart' statement
ALL	0x00000033	ALL	select all keys in the 'key' statement
VEL	0x00000001	velocity	testing 'mode' variable
POSIT	0x00000002	position	ditto
ABS	0x00000003	absolute	ditto
REL	0x00000004	relative	ditto
TOR	0x00000005	torque	ditto
GEAR	0x00000006	gear	ditto
VA	0x00000007	vector absolute	ditto
VR	0x00000008	vector relative	ditto
CAM	0x0000000B	cam	ditto
INT0_POL	0x00000000	interrupt 0 polarity	use with 'ints[]' statement
INT1_POL	0x00000001	interrupt 1 polarity	ditto
INT2_POL	0x00000002	interrupt 2 polarity	ditto
INT3_POL	0x00000003	interrupt 3 polarity	ditto
INT4_POL	0x00000004	interrupt 4 polarity	ditto
INT5_POL	0x00000005	interrupt 5 polarity	ditto
INT6_POL	0x00000006	interrupt 6 polarity	ditto
INT7_POL	0x00000007	interrupt 7 polarity	ditto
INT8_POL	0x00000008	interrupt 8 polarity	ditto
INT9_POL	0x00000009	interrupt 9 polarity	ditto
INT0_EN	0x0000000A	interrupt 0 enable	ditto
INT1_EN	0x0000000B	interrupt 1 enable	ditto
INT2_EN	0x0000000C	interrupt 2 enable	ditto
INT3_EN	0x0000000D	interrupt 3 enable	ditto
INT4_EN	0x0000000E	interrupt 4 enable	ditto
INT5_EN	0x0000000F	interrupt 5 enable	ditto
INT6_EN	0x00000010	interrupt 6 enable	ditto
INT7_EN	0x00000011	interrupt 7 enable	ditto
INT8_EN	0x00000012	interrupt 8 enable	ditto
INT9_EN	0x00000013	interrupt 9 enable	ditto
UARTH_EN	0x00000014	uart int enable	ditto
MSGH_EN	0x00000015	msg int enable	ditto
GLOBAL_EN	0x00000016	global int enable	ditto
START_MOVE	0x00000000	start move	use with 'status[]' statement
ENABLE_DAC0	0x00000001	enable dac0	ditto
MARK	0x00000002	encoder mark	ditto
CAPTURED	0x00000003	encoder pos captured	ditto
CAP_POL	0x00000004	capture edge polarity	ditto
TIMED_OUT	0x00000005	time-out occurred	ditto
AUTOLOAD	0x00000006	Auto load from queue	ditto
AUTORUN	0x00000007	motion auto run	ditto
SHOW_MODE	0x00000008	show mode on display	ditto
HALT_MOVE	0x0000000B	halt move	ditto
RUN_PROG	0x0000000C	run user program	ditto
LINK	0x0000000E	link moves	ditto
PROG_RUNNING	0x00000010	program running?	ditto
RESET	0x00000011	reset user program	ditto
ZERO_ENC	0x00000012	zero encoder	ditto
ADC0_UNIPOLAR	0x00000013	adc0 unipolar output	ditto
ADC1_UNIPOLAR	0x00000014	adc1 unipolar output	ditto
ADC2_UNIPOLAR	0x00000015	adc2 unipolar output	ditto
ADC3_UNIPOLAR	0x00000016	adc3 unipolar output	ditto
ADC4_UNIPOLAR	0x00000017	adc4 unipolar output	ditto
ADC5_UNIPOLAR	0x00000018	adc5 unipolar output	ditto

<u>Name</u>	<u>Hex Value</u>	<u>Meaning</u>	<u>Use</u>
ADC6_UNIPOLAR	0x00000019	adc6 unipolar output	ditto
ADC7_UNIPOLAR	0x0000001A	adc7 unipolar output	use with 'status[]' statement
MARK_POL	0x0000001C	mark input polarity	ditto
INVERT_ENC0	0x0000001D	invert encoder 0	ditto
INVERT_DAC0	0x0000001E	invert dac0	ditto
OP_DISPLAY	0x00000001	option display	setting 'options' variable
OP_RS485	0x00000002	option rs485	ditto
OP_ETHERNET	0x00000004	option Ethernet	ditto
OP_DAC0	0x00000008	option dac0	ditto
OP_DAC1	0x00000010	option dac1	ditto
OP_DAC2	0x00000020	option dac2	ditto
OP_DAC3	0x00000040	option dac3	ditto
OP_ENCODER	0x00000800	option encoder	ditto
OP_DO_0_7	0x00001000	option digital out 0-7	ditto
OP_DO_8_15	0x00002000	option digital out 8-15	ditto
OP_DO_16_23	0x00004000	option digital out 16-23	ditto
OP_DO_24_31	0x00008000	option digital out 24-31	ditto
OP_DI_0_7	0x00010000	option digital in 0-7	ditto
OP_DI_8_15	0x00020000	option digital in 8-15	ditto
OP_DI_16_23	0x00040000	option digital in 16-23	ditto
OP_DI_24_31	0x00080000	option digital in 24-31	ditto
OP_ADC0	0x00100000	option adc0	ditto
OP_ADC1	0x00200000	option adc1	ditto
OP_ADC2	0x00400000	option adc2	ditto
OP_ADC3	0x00800000	option adc3	ditto
OP_CONTROL	0x10000000	option control loop	ditto
OP_UART	0x20000000	option uart	ditto
OP_VAXIS	0x40000000	option virtual axis	ditto
ARIEL9	0x00000000	font ariel 9 point	setting 'font' variable
ARIEL18	0x00000001	font ariel 18 point	ditto
MONO5_8	0x00000002	font monospace 5 by 8	ditto
MONO8_8	0x00000003	font monospace 8 by 8	ditto
MSFONT5_8	0x00000004	font microsoft 5 by 8	ditto
MSFONT7_8	0x00000005	font microsoft 7 by 8	ditto
NARROW10	0x00000006	font narrow 10 point	ditto
NARROW20	0x00000007	font narrow 20 point	ditto
TIMES9	0x00000008	font times 9 point	ditto
TIMES13	0x00000009	font times 13 point	ditto
CON	0x00000001	cursor on	use with 'cursor()' statement
COFF	0x00000000	cursor off	ditto
CBLOCK	0x00000008	block cursor	ditto
CBLINK	0x00000004	blink cursor	ditto
WHITE	0x000000FF	colour white	setting colour on 'screen'
BLACK	0x00000000	colour black	ditto
RED	0x000000F9	colour red	ditto
YELLOW	0x000000FB	colour yellow	ditto
GREEN	0x000000FA	colour green	ditto
BLUE	0x000000FC	colour blue	ditto
ORANGE	0x000000BE	colour orange	ditto
GREY	0x000000F7	colour grey	ditto
MAGENTA	0x000000FD	colour magenta	ditto
CYAN	0x000000FE	colour cyan	ditto
RISING	0x00000001	rising edge	use with 'edges()' statement
FALLING	0x00000000	falling edge	ditto
BOTH	0x00000002	both edges	ditto

Variables

In Control-C there is only one variable type, a 48 bit fixed point value consisting of a 32 bit integer term and a 16 bit decimal term. There are Three different forms, 'User Defined Locals', 'User Defined Globals' and 'Pre-Defined Globals'.

▶ *The E-node embedded network scheme allows full sharing of global variables across the network. This is a very powerful and useful feature implemented in a simple way. Please read the following paragraphs carefully to ensure a full understanding of how variable access is correctly performed. Failure to do so could result in programming errors that display unexpected results and are difficult to track down.*

Variable Format

Format: 32 bit integer . 16 bit decimal

Range +/- 2,147,483,648.000000

Resolution 0.000015

Following is a series of values illustrating how they are stored:

Value	decimal format	Hexadecimal format (with imaginary decimal point)
10	10.0	0x0000000A.0000
-10	-10.0	0xFFFFFFFF6.0000
0.1	0.1	0x00000000.119A
-0.1	-0.1	0xFFFFFFFF.EE66
'A'	65	0x00000041.0000
TRUE	1	0x00000001.0000
FALSE	0	0x00000000.0000

▶ *When a variable is tested in a program to determine if it is 'true', for example in an 'if' statement, the result will be false if the variable is zero and true if a bit is set in either the integer or the decimal term*

Variable Names

Variable names can be a maximum of 20 characters in length. The first character must be a letter or '_' and the remainder can be a letter, '_' or a number. Examples are:

Name	Status
a	legal
_a	legal
a9	legal
aaaaaaaaaaaaaaaaaaaaa	illegal - too long
9a	illegal - leading number not allowed

▶ *Variable names are case sensitive, 'pop' and 'POP' are two different names*

Arrays

Arrays are one dimensional and consist of the standard variable type. They can be declared in both global and local forms.

- ▶ *Global arrays have a maximum size of 4000.*
- ▶ *Local arrays have a maximum size of 100.*

User Defined Local Variables

User defined local variables are individual or arrays of variables declared at the start of a subroutine using the key word 'var'. They are only visible within the subroutine in which they are declared and cannot be accessed by other subroutines or programs. All names used must be unique within that subroutine and cannot be the same as any global variable name. As the example below shows, 'main()' is also considered to be a subroutine for the purposes of local variable usage.

Example:

```
subroutine_1 ()
{
  var z;    // declare local variable
  z = 9;    // assign value
}

main ()
{
  var y;    // declare local variable within main
  // Program starts here!
}
```

► *It is legal to use same name for local variables in different subroutines, however this practice is not recommended because it can lead to confusion.*

User Defined Global Variables

User defined global variables are individual or arrays of variables declared at the top of the user program directly after any constant definitions. They can be accessed from anywhere within the program and by other programs running elsewhere on the network. The key word 'var' is used followed by the number of definitions required.

► *Important: Global variables always have an associated node address. During declaration the address is optionally provided by entering a colon followed by the numerical address. If no address is declared the address given at the top of the file window will be used by default. When a running program accesses any global variable its associated node address is automatically used to determine its location on the network. (see below for more detail).*

Example:

```
var
topx,          // variable resides on default node address
topy,          // ditto
topz,          // ditto
topw:9,        // variable resides on node address = 9
array[10],     // array resides on default node address
n2array:2[10]; // array resides on node address = 2
```

Pre-Defined Global Variables

Pre-defined global variables are a group of variables tied directly to the workings of the E-node. Every E-node holds a complete set that can be accessed using the node address. Some Pre-defined global variables will have no meaning depending on the type of node, e.g. the variable 'font' is only meaningful on 'screen' type nodes.

Following is a complete list of pre-defined global variables giving the name, its function, units if applicable, numerical range and accessibility.

software_version

Function: The current software version number
Units: none
Range: none
Access: read only
Node type: all

address

Function: The network address for the node
Units: none
Range: 0 to 999 (decimal term not used)
Access: read only (assigned during node configuration)
Node type: all

serial_number

Function: The factory programmed serial number for the node. Built from the day, month, year (20xx) and number that day. e.g.

day	month	year	number	=	hexadecimal
10	5	8	21	=	0x0A050815.0000 (decimal term unused)

Units: none
Range: none
Access: read only
Node type: all

node_type

Function: A numerical value for the Node type (factory programmed)

value	Node type
0	CORE (gateway)
1	AXIS
2	PNP-IO
3	NPN-IO
4	ADC
5	DAC
6	COMMS (gateway)
7	COMBI
8	SCREEN (gateway)

Units: none
Range: none
Access: read only
Node type: all

group_addr_hi

Function: The high group address for a gateway node (user configured)
Units: none
Range: 0 to 999 (decimal term not used)
Access: read only
Node type: gateway only (assigned during node configuration)

group_addr_lo

Function: The low group address for a gateway node (user configured)
Units: none
Range: 0 to 999 (decimal term not used)
Access: read only
Node type: gateway only (assigned during node configuration)

power_on_run

Function: If true will run the user program on power up (user configured)
Units: none
Range: 0 or 1 (decimal term not used)
Access: read only
Node type: all (assigned during node configuration)

error

Function: The last reported error
Units: none
Range: 0 to 65535 (decimal term not used)
Access: read / write
Node type: all

line_with_error

Function: Estimated program line where error occurred
Units: none
Range: full (decimal term not used)
Access: read only
Node type: all

node_with_error

Function: The address of the node reporting the error
Units: none
Range: 0 to 65535 (decimal term not used)
Access: read only
Node type: all

status

Function: Special case - variable is used as an array of bits for controlling and reporting the status of the node.
Units: none
Range: full (decimal term not used)
Access: write using `status[xx] = avalue`
read using `avar = status[xx]`
tested using `status[xx]==y`
Node type: all

There are a number of defined BUILTIN constants for use with setting and testing the status variable. Their functions are as follows:

status [START_MOVE] = 0;	current move is stopped
status [START_MOVE] = 1;	current move is started (if autorun == 0)
status [ENABLE_DAC0] = 0;	DAC0 output is disabled
status [ENABLE_DAC0] = 1;	DAC0 output is enabled
status [MARK] = 0;	clear encoder mark pulse seen flag
status [MARK] == 1	true if encoder mark pulse seen
status [CAPTURED] = 0;	clear encoder position captured flag
status [CAPTURED] == 1	true if encoder position captured
status [CAP_POL] = 0;	encoder capture input polarity = falling
status [CAP_POL] = 1;	encoder capture input polarity = rising
status [TIMED_OUT] = 0;	clear timed out flag for 'getch()'
status [TIMED_OUT] == 1	true if timed out
status [AUTOLOAD] = 0;	motion commands do not auto load from command queue
status [AUTOLOAD] = 1;	motion commands auto load from command queue
status [AUTORUN] = 0;	motion commands do not automatically run when loaded
status [AUTORUN] = 1;	motion commands automatically run when loaded
status [SHOW_MODE] = 0;	E-node ESD shows program status (if 'display' is 0)
status [SHOW_MODE] = 1;	E-node ESD shows motion command mode (if 'display' is 0)
status [HALT_MOVE] = 0;	current motion command is not halted
status [HALT_MOVE] = 1;	current motion command is halted
status [RUN_PROG] = 0;	user program is paused
status [RUN_PROG] = 1;	user program is run
status [LINK] = 0;	motion commands are not linked
status [LINK] = 1;	motion commands are linked
status [PROG_RUNNING] == 0	true if user program not running
status [PROG_RUNNING] == 1	true if user program running
status [RESET] = 1;	reset user program
status [ZERO_ENC] = 1;	zero encoder value
status [ADC0_UNIPOLAR] = 0;	adc output range -10 to 10 volts
status [ADC0_UNIPOLAR] = 1;	adc output range 0 to 10 volts
▶ same for other ADCx_UNIPOLAR options	
status [MARK_POL] = 0;	encoder mark polarity = falling
status [MARK_POL] = 1;	encoder mark polarity = rising
status [INVERT_ENC0] = 0;	encoder count not inverted
status [INVERT_ENC0] = 1;	encoder count inverted
status [INVERT_DAC0] = 0;	Dac0 output signal not inverted
status [INVERT_DAC0] = 1;	Dac0 output signal inverted

activity

Function: Reports the current activity of motion commands
Units: none
Range: full (decimal term not used)
Access: read only
Node type: Any motion capable node.

There are a number of defined BUILTIN constants for use with testing the 'activity' variable.

mode

Function: Reports the current mode of motion commands
Units: none
Range: full (decimal term not used)
Access: read only
Node type: Any motion capable node.

There are a number of defined BUILTIN constants for use with testing the 'mode' variable.

demand

Function: demand position for the internal move profile generator.
Writes only accepted when in 'am', 'rm' or 'pm' modes. Otherwise ignored.
Units: encoder counts
Range: full (decimal term not used)
Access: read / write
Node type: Any motion capable node.

ferr

Function: positional following error for current motion command
Units: encoder counts
Range: full (decimal term not used)
Access: read only
Node type: Any motion capable node.

vel

Function: maximum velocity for current motion command
Units: encoder counts/control loop period (2 millisecs)
Range: +/- 4000
Access: read / write
Node type: Any motion capable node.

accel

Function: maximum acceleration for current motion command
Units: encoder counts/control loop period² (2 millisecs)
Range: 0.000015 to 160
Access: read / write
Node type: Any motion capable node.

decel

Function: maximum deceleration for current motion command
Units: encoder counts/control loop period² (2 millisecs)
Range: 0.000015 to 160
Access: read / write
Node type: Any motion capable node.

ratchet

Function: controls the ratchet operation for 'position', 'cam' and 'gear' mode motion commands

`ratchet=0;` ratchet disabled
`ratchet=1;` +ve movement allowed
`ratchet=2;` -ve movement allowed

Units: none
Range: 0 to 2
Access: read / write
Node type: Any motion capable node.

offset

Function: offset to demand for relative moves. Motion commands only.

Units: encoder counts
Range: full (decimal term not used)
Access: read / write
Node type: Any motion capable node.

prop_gain

Function: proportional gain for the PID control loop. Motion commands only.

Units: none
Range: 0 to 10000
Access: read / write
Node type: Any motion capable node.

diff_gain

Function: differential gain for the PID control loop. Motion commands only.

Units: none
Range: 0 to 10000
Access: read / write
Node type: Any motion capable node.

integ_gain

Function: integral gain for the PID control loop. Motion commands only.

Units: none
Range: 0 to 10000
Access: read / write
Node type: Any motion capable node.

enc0_vel

Function: the velocity of encoder0

Units: encoder counts/control loop period (2 millisecs)
Range: full (decimal term not used)
Access: read only
Node type: Any motion capable node.

torque

Function: torque demand for torque mode motion command

Units: % of full dac output, 100 = 10v, -100 = -10v
Range: +/- 100
Access: read / write
Node type: Any motion capable node.

time

Function: the time since last reset
Units: seconds
Range: full
Access: read / write
Node type: all

enc0

Function: encoder0 reading
Units: encoder counts
Range: full (decimal term not used)
Access: read only
Node type: Any motion capable node.

enc1

Function: encoder1 value. Used in gear locking modes.
Usually seeded automatically with a broadcast master encoder value.
Units: encoder counts
Range: full (decimal term not used)
Access: read / write
Node type: all

inputs

Function: value of digital inputs. The individual bits of the integer term hold the input state.
input[0] starts at bit 0 of integer term.
Units: none
Range: none (decimal term not used)
Access: read only
Node type: any node with digital I/O

outputs

Function: value of digital outputs. The individual bits of the integer term hold the output state.
output[0] starts at bit 0 of integer term.
Units: none
Range: none (decimal term not used)
Access: read / write
Node type: any node with digital I/O

accum_limit

Function: accumulator limit value. Part of the integral term in the PID control loop.
Limits the effect of the integral term.
Units: none
Range: 0 to 32767
Access: read / write
Node type: Any motion capable node.

ferr_limit

Function: following error limit. If the following error of a move exceeds this limit + or -
a following error will occur. If set to zero following errors are ignored.
Units: encoder counts
Range: full (decimal term not used)
Access: read / write
Node type: Any motion capable node.

demand_vel

Function: reports the instantaneous velocity generated by the profile generator during profiled move commands
Units: encoder counts / control loop period
Range: full (decimal term not used)
Access: read only
Node type: Any motion capable node.

break

Function: Special system variable for use with break points during program development.
Units: none
Range: none
Access: none (system use only)
Node type: all

pos_limit

Function: positive software position limit for motion commands. The motor will stop if an attempt is made to drive it past this point. Ignored if set to zero.
Units: encoder counts
Range: full (decimal term not used)
Access: read / write
Node type: Any motion capable node.

neg_limit

Function: negative software position limit for motion commands. The motor will stop if an attempt is made to drive it past this point. Ignored if set to zero.
Units: encoder counts
Range: full (decimal term not used)
Access: read / write
Node type: Any motion capable node.

dac0_clip

Function: Clipping voltage for dac0 output. Acts symmetrically. A value of 5 will clip the output to a +/- 5v range
Units: volts
Range: 0 to 10.5
Access: read / write
Node type: any node with a dac0 output

dac0_offset

Function: Centre offset voltage for dac0 output.
Units: volts
Range: +/- 5v
Access: read / write
Node type: any node with a dac0 output

dac0

Function: voltage for dac0 output.
If the node is performing control loop operations this value is constantly overridden by the control loop output voltage.
Units: volts
Range: +/- 10.5v
Access: read / write
Node type: any node with a dac0 output

dac1 to dac7

Function: voltage for dac1 to dac7 output.
Units: volts
Range: +/- 10.5v
Access: read / write
Node type: any node with a dac1 to dac7 output

master_teeth

Function: Number of master gear teeth for gear locking motion commands
Units: gear teeth
Range: +/- 30000 (decimal term not used)
Access: read / write
Node type: Any motion capable node.

slave_teeth

Function: Number of slave gear teeth for gear locking motion commands
Units: gear teeth
Range: +/- 30000 (decimal term not used)
Access: read / write
Node type: Any motion capable node.

timeout

Function: Time the 'getch()' command will wait for a character before timing out.
Timing out is disabled if 'timeout' set to zero.
▶ *Timing out is performed by the node calling the 'getch()' command, therefore 'timeout' must be set on that node.*
Units: seconds
Range: full
Access: read / write
Node type: all

mgpos

Function: Master encoder (enc1) position for registered gear locking motion commands
Units: encoder counts
Range: full (decimal term not used)
Access: read / write
Node type: Any motion capable node.

sgpos

Function: Slave encoder (enc0) position for registered gear locking motion commands
Units: encoder counts
Range: full (decimal term not used)
Access: read / write
Node type: Any motion capable node.

cappos

Function: enc0 position captured from an interrupt on input[0]
Units: encoder counts
Range: full (decimal term not used)
Access: read / write
Node type: Any motion capable node.

display

Function: Allows the user program to use the Eight segment display (ESD) on the front of the E-node.

Units: none

Range: first 8 bits of integer term only. (decimal term not used)

Access: read / write

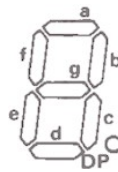
Node type: all E-nodes with an Eight segment display (ESD)

▶ If 'display' is zero the ESD will show normal operation.

▶ Displaying errors takes precedence over user program use of the ESD. Therefore if the user program wishes to use the ESD whilst in the error handler (errorh()) it must first clear the 'error' variable.

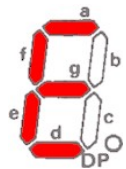
The 'display' variable uses the first 8 bits of the integer term to control the 8 segments of the ESD. If the bit is 1 the segment is ON and if the bit is 0 the segment is OFF.

bit:	7	6	5	4	3	2	1	0
segment:	a	b	c	d	e	f	g	dp



Example:

```
display = 0x9E // 10011110
```



rx_bcv0

Function: rx_bcv0 stands for 'receive broadcast variable zero'.

Holds the number of the variable that will receive the value of the broadcast variable zero.

▶ Use 'varnum()' to get the variable number.

Units: none

Range: full (decimal term not used)

Access: read / write

Node type: all

Example:

```
var myvarname;  
  
main() // varnum() gives the number of myvarname  
{  
    rx_bcv0 = varnum(myvarname);  
}
```

In this example another node on the network has been set up to regularly broadcast a variable value (using 'tx_bcv0'). On receipt the value is placed in 'myvarname'.

▶ This is a very powerful feature that can be used for many different data sharing needs. In another typical example the value of a master encoder is broadcast and other node/s use this value for gear locking commands.

rx_bcv1

Function: rx_bcv0 stands for 'receive broadcast variable one'. Holds the number of the variable that will receive the value of the broadcast variable one.
▶ Use 'varnum()' to get the variable number.

Units: none
Range: full (decimal term not used)
Access: read / write
Node type: all

tx_bcv0

Function: tx_bcv0 stands for 'transmit broadcast variable zero'. Holds the number of the variable that will be broadcast over the network.
▶ Broadcasting is performed once every 'bcv_interval' but only if the value of the variable has changed since the last broadcast.
▶ Use 'varnum()' to get the variable number.

Units: none
Range: full (decimal term not used)
Access: read / write
Node type: all

Example:

```
main ()          // varnum() gives the number of myvarname
{
    tx_bcv0 = varnum(enc0);
}
```

In this example the node will broadcast the value of 'enc0' every bcv_interval if that value has changed since the last broadcast.

tx_bcv1

Function: tx_bcv1 stands for 'transmit broadcast variable one'. Holds the number of the variable that will be broadcast over the network.
▶ Broadcasting is performed once every 'bcv_interval' but only if the value of the variable has changed since the last broadcast.
▶ Use 'varnum()' to get the variable number.

Units: none
Range: full (decimal term not used)
Access: read / write
Node type: all

▶ There is a maximum of two broadcast variables for any group of nodes on the network. Gateway nodes that connect groups can be programmed to block or pass broadcast variables. In this way groups may be set up to either share broadcasts with other groups or have their own dedicated broadcasts. Care must be taken when setting up broadcasts. For example if Two or more nodes are programmed to broadcast 'tx_bcv0' for the same group or sharing groups then overwriting of the receiving variable will occur. This may prove to be a difficult software fault to find.

bcv_interval

Function: The number of control loop counts between broadcasts
Units: control loop counts (one count = 2 milli seconds)
Range: 1 to 10000 counts (2 milli seconds to 20 seconds) (decimal term not used)
Access: read / write
Node type: all

▶ When option virtual axis has been selected bcv_interval should be set to 1 to match the control loop.

ints

Function: Special variable used for the set-up and control of the 12 user interrupts 'int0h()' through 'int9h()', 'uart()' and 'msgH()'.

Units: none

Range: none (decimal term not used)

Access: only using the 'ints[]' command

Node type: all

► 'ints' is used as an array of bit values. A range of 'BUILTIN' constants is provided. They are used to set-up the input edge polarity and enable the interrupts both individually and globally. Please refer to the section on interrupts for further information.

ipf0 to ipf9

Function: Stands for 'input filter x', where 'x' is the digital input number. Digital inputs can have filtering applied to block noise or unwanted signal pulse widths. This variable holds the number of digital I/O loop periods that an input must remain at a stable level before a change in that level will be reported. Digital I/O is reported every 500 microseconds, therefore an 'ipf' value of 6 will be required to generate a 3 millisecond filter window.

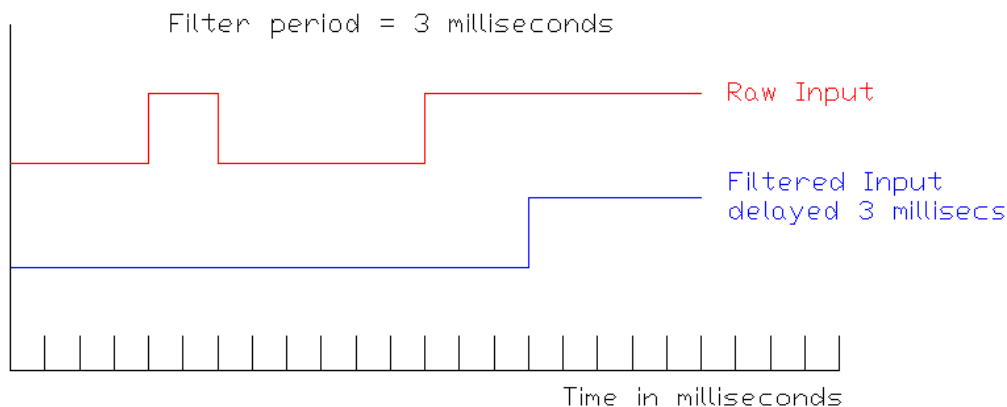
► Filtering is disabled if the value is set to zero.

Units: digital I/O loop counts

Range: 0 to 1000 (decimal term not used)

Access: read / write

Node type: all nodes with digital inputs



► Digital input filtering delays the input signal by the filter period. Therefore any action taken on changes to inputs including user interrupts will also be delayed by the filter period.

► Note: Very fast signal glitches less than 1 microsecond are very unlikely to get past the standard opto isolation barrier present on all inputs.

► Note: High speed position capture interrupts on input[0] are processed independently of normal I/O reporting and are NOT effected by digital input filtering. However they do have an input signal blanking feature see 'input0_blank'.

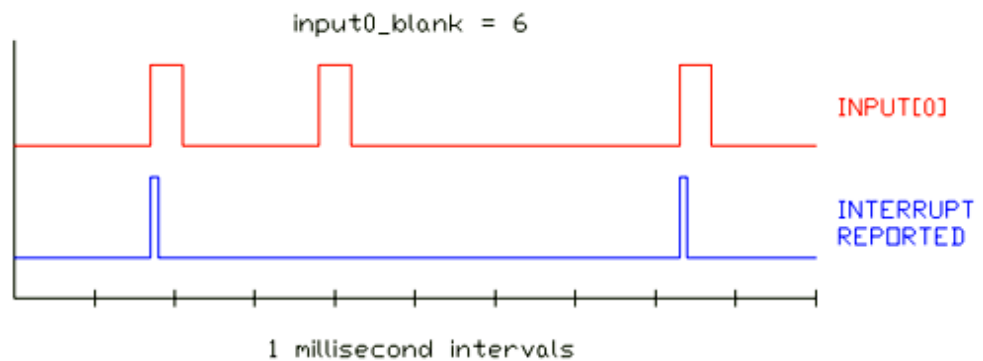
input0_blank

Function: The number of digital I/O loop periods that 'input[0]' must remain stable for, before a valid change will generate an encoder position capture interrupt. The digital I/O loop period is 500 microseconds therefore a value of 4 will generate a 2 millisecond blank period.

▶ This feature only effects position capture interrupt processing and is completely independent from normal digital I/O processing

▶ Note: This feature does not delay the reporting as with normal digital I/O filtering.

Units: milliseconds
Range: 0 to 400 (decimal term not used)
Access: read / write
Node type: all



options

Function: This variable is reserved for use with the 'CORE' node when setting hardware options. It is also used with any node for setting the virtual axis option 'OP_VAXIS'. It is organised as an array of bits. A range of pre-defined 'BUILTIN' constants are provided for use with the 'options' variable.

Units: none
Range: none (integer term only used)
Access: read / write ('CORE' node only)
Node type: 'CORE' only

▶ Options are enabled by 'or-ing' the 'options' variable with a value

```
options |= OP_UART | OP_CONTROL; // enable uart and control
// options
```

▶ Options are disabled by 'and-ing' the 'options' variable with a value.

```
options &= ~(OP_UART | OP_CONTROL); // disable uart and control
// options
```

queue

Function: Motion commands are queued in a motion command queue on a first in, first out basis. This variable reports the length of the queue.
Reading 'queue' returns the current queue length (the number of queued commands).
Writing a value smaller than the current 'queue' length will shorten the queue to the new value by removing commands from the front of the queue.

Units: none
Range: none (integer term only used)
Access: read / write
Node type: Any motion capable node.

▶ *Care must be taken when shortening the motion command queue. The length of the queue may vary dynamically depending on the settings in the 'status' variable. Please read the chapter on motion command queuing?*

adc0 to adc7

Function: The values recorded from the analogue to digital input channels 0 to 7.
Units: volts
Range: +/- 10.5v
Access: read only
Node type: any node with adc input

beep_frequency

Function: The frequency used to produce a beep sound on the 'SCREEN' node when 'beep()' is called without parameters. Calling 'beep()' with parameters temporarily overrides 'beep_frequency' for that call only.

Units: hertz
Range: 1000 to 5000 (integer term only used)
Access: read / write
Node type: 'SCREEN'

beep_duration

Function: The duration used to produce a beep sound on the 'SCREEN' node when 'beep()' is called without parameters. Calling 'beep()' with parameters temporarily overrides 'beep_duration' for that call only.

Units: milliseconds
Range: 1 to 10000 (integer term only used)
Access: read / write
Node type: 'SCREEN'

wheel

Function: The value read from the wheel encoder if connected to a 'SCREEN' node. The wheel encoder provides a rotary digital encoder input to the 'SCREEN' node enabling potentiometer like input for the user.

Units: counts

Range: full (integer term only used)

Access: read / write

Node type: 'SCREEN'

▶ *A wheel encoder generates 256 counts/revolution. However the software is only designed to track low speed inputs and will potentially miss encoder counts if the wheel is turned faster than 2 revs/second.*

fc colour

Function: The foreground colour used when writing text and some graphics to the screen of a 'SCREEN' node.

Units: none

Range: 0 to 255 (integer term only used)

Access: read / write

Node type: 'SCREEN'

▶ *Please refer to the E-nodes Technical Description document for full details of the colour range.*

▶ *A limited range of basic colours for use with fc colour is provided in 'BUILTIN' constants.*

bc colour

Function: The background colour used when writing text and some graphics to the screen of a 'SCREEN' node.

Units: none

Range: 0 to 255 (integer term only used)

Access: read / write

Node type: 'SCREEN'

▶ *Please refer to the E-nodes Technical Description document for full details of the colour range.*

▶ *A limited range of basic colours for use with bc colour is provided in 'BUILTIN' constants.*

tc colour

Function: Transparency colour. Colour bit mapped graphical images can be written to the screen of 'SCREEN' nodes. Any one of the 256 colours that can be in the image may be selected to be transparent by setting 'tc colour' to that number. During writing, any pixel with the colour 'tc colour' is not written leaving the original pixel intact. Using this method images can be written with, for example the background set to 'tc colour', giving the effect of superimposing one image on another.

Units: none

Range: 0 to 255 (integer term only used)

Access: read / write

Node type: 'SCREEN'

▶ *A value of -1 turns off transparency mode.*

keycolour

Function: Defines the colours used when touch keys are placed on the screen of 'SCREEN' nodes. The first 3 bytes are used to define the 3 colour used for a key, 'highlight colour', 'lowlight colour' and 'body colour'. Each can be any one of the 256 available colours.

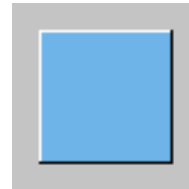
Units: none
Range: first 3 bytes only (integer term only used)
Access: read / write
Node type: 'SCREEN'

byte	3	2	1	0
	n/a	highlight	lowlight	body

highlight = top and left border

lowlight = bottom and right border

body = centre block



Key on grey background

► While a key is being touched the lowlight and highlight border colours are swapped to give the impression of the key being depressed.

touched

Function: Reports the touched state of the 'SCREEN' node touch screen.
'TRUE' if being touched.
'FALSE' if not being touched.

Units: none
Range: 0 or 1 (integer term only used)
Access: read only
Node type: 'SCREEN'

touchx

Function: Reports the 'x' coordinate for any touch of the 'SCREEN' node touch screen.
Units: none
Range: 0 to 319 (integer term only used)
Access: read only
Node type: 'SCREEN'

► Note: The top left hand corner is $[0,0] = [x,y]$

touchy

Function: Reports the 'y' coordinate for any touch of the 'SCREEN' node touch screen.
Units: none
Range: 0 to 239 (integer term only used)
Access: read only
Node type: 'SCREEN'

► Note: The top left hand corner is $[0,0] = [x,y]$

font

Function: Holds a number representing the font used when writing text to the screen of 'SCREEN' nodes. There are a range of 'BUILTIN' constants for use with 'font'.
Units: none
Range: none (integer term only used)
Access: read / write
Node type: 'SCREEN'

brightness

Function: Sets the screen back light brightness for 'SCREEN' nodes in normal operation. 0 = off, 100 = full brightness.
Units: none
Range: 0 to 100 (integer term only used)
Access: read / write
Node type: 'SCREEN'

sleep_brightness

Function: Sets the screen back light brightness for 'SCREEN' nodes in sleep mode. 0 = off, 100 = full brightness.
Units: none
Range: 0 to 100 (integer term only used)
Access: read / write
Node type: 'SCREEN'

brightness_timeout

Function: Sets the the time out in seconds from when the last touch was made to the screen entering sleep mode and reducing the brightness level to 'sleep_brightness'. Touching the screen exits sleep mode and returns the brightness level to the value given in 'brightness'.

▶ if *brightness_timeout* = 0 the time out feature is disabled.

Units: seconds
Range: full, +ve only (integer term only used)
Access: read / write
Node type: 'SCREEN'

cap_window_hi

Function: encoder capture registration window most +ve value

▶ if *cap_window_hi* = 0 and *cap_window_lo* = 0 the registration window is disabled.

Units: encoder counts
Range: full (decimal term not used)
Access: read / write
Node type: Any motion capable node.

cap_window_lo

Function: encoder capture registration window most -ve value

▶ if *cap_window_hi* = 0 and *cap_window_lo* = 0 the registration window is disabled.

Units: encoder counts
Range: full (decimal term not used)
Access: read / write
Node type: Any motion capable node.

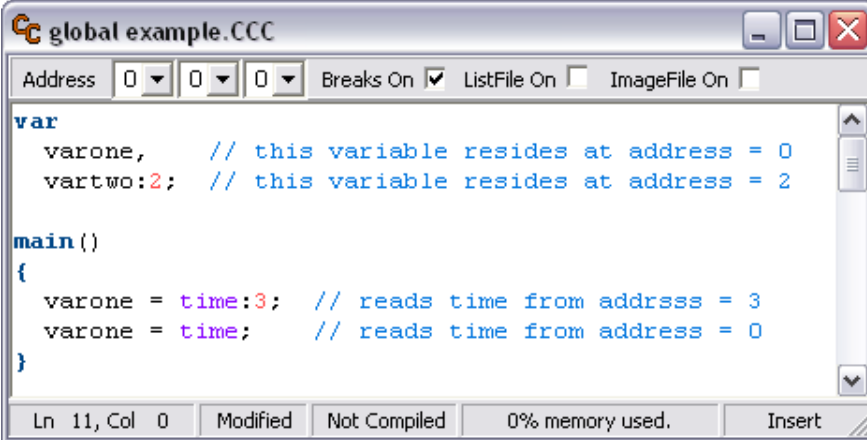
Accessing Global Variables

► *This is a very important subject. Correct understanding and use of accessing global variables can have a big impact on the simplicity and operation of user programs.*

All global variables (both pre-defined and user-defined) can be accessed by any program running on any node on the E-net network. To achieve this all global variables have an associated network address.

Pre-defined globals automatically take the address of the node on which they reside. To access the variable the programmer types the name followed by a colon and the address of the node on which the variable resides. Alternatively the colon and address can be omitted and the compiler will assume the address given at the top of the program file window.

User-defined globals are assigned an address when the user program is compiled. The programmer informs the compiler of the address to use by following the variable name with a colon followed by the address. Alternatively this step can be omitted and the compiler will assume the address given at the top of the program file window.



```
global example.CCC
Address 0 0 0 Breaks On ListFile On ImageFile On
var
varone, // this variable resides at address = 0
var two:2; // this variable resides at address = 2
main()
{
varone = time:3; // reads time from address = 3
varone = time; // reads time from address = 0
}
```

Assignment Order

When a program is compiled it starts at the top of the file window and works down through the program. As user-defined global variables are encountered, those with the same address are grouped together. Within each group they are each assigned a unique variable number on a first found, first assigned basis. The first found would be assigned a variable number 'x', the next 'x+1' and so on. This gives rule one for assignment:-

► **RULE ONE:** *The order in which global variables are declared is VERY IMPORTANT.*

To access a global variable declared in one program from another, the accessing program must know two things. Firstly the address and secondly the variable number. The address is given directly as described above. The variable number is given by the compiler according to the assignment order. This gives rule Two for assignment.

► **RULE TWO:** *For a program to access a global variable in another program the variable concerned MUST BE DECLARED IN THE SAME ORDER IN BOTH PROGRAMS.*

► *If rule Two is not followed the variable numbers will be different for the different programs. When a access is made from one program to another the wrong variable will be accessed.*

```

prog1.CCC
var
prog1a:1, // address = 1
prog1b:1, // address = 1

prog2a:2, // address = 2
prog2b:2[10]; // address = 2

prog2.CCC
var
prog2a:2, // address = 2
prog2b:2[10]; // address = 2

```

Global (and arrays) declarations over two programs

If you wish to access a global variable or number of global variables from another program, group them together at the very start of the variable declaration process. For example if you have 5 variables in a program declare all 5 at the start of every program from which you wish to access them. It is also recommended that all addresses are explicitly given using the colon + address process. This will help to avoid confusion.

In the example above prog1.CCC can access the two declared variables of prog2.CCC by simply typing in the full name and address, e.g. prog2a:2. However prog2.CCC has no knowledge of the variables declared in prog1.CCC and cannot access them.

► *Following these guidelines provides the programmer with the ability to access information from anywhere on the network by simply typing in the name and address of the required variable.*

Broadcast Variables

Using broadcast variables is a very efficient way of sharing a limited quantity of data between nodes in a group on the E-Net network. Groups can be linked together using gateway nodes that can be programmed to pass or block broadcast variables. A maximum of two broadcast variables can be set up for regular transmission across all nodes in the group or linked groups. Any node within the group or linked groups can be set up to receive.

When a node is set up to broadcast a variable it will do so once for every 'bcv_interval'. This variable holds the number of control loop counts between broadcasts in the range 1 to 10000 or 2 milliseconds to 20 seconds.

► *If broadcast variables are being used with motion control it is essential that the 'bcv-interval' be set to 1 to ensure correct update of encoder positional data.*

If the value of the variable has not changed since the last broadcast it is not sent. This keeps receiving nodes up to date with changed data but saves communications bandwidth when the data is unchanged.

To configure a node for transmission the pre-defined variable 'tx_bcv0' or 'tx_bxv1' needs to be assigned the number of the variable to be transmitted (see pre-defined variables above).

To configure a node to receive a transmission the pre-defined variable 'rx_bcv0' or 'rx_bxv1' needs to be assigned the number of the variable to receive the data. (see pre-defined variables above).

► *An example for the use of broadcast variables would be the regular transmission of a command variable to a large number of nodes. All 48 bits of the variable could be used as control flags or the variable could contain a control code.*

► *Another regular use for broadcast variables is the sharing of a master encoder value across several nodes. The receiving nodes can then perform gear locking or remote position related moves.*

Functions

A function is a sub-routine that may be called from elsewhere in the program. It may have parameters passed to it and it may return a single value when it finishes. Parameters will be temporarily stored as local variables and can only be accessed from within the function. To take full advantage of functions they should be written to perform well defined specific operations. This enables the programmer to build a library of self contained functions that may be repeatedly used in many different programs.

When the compiler encounters a function it creates a global return variable with the same name as the function. When a return is made this variable is seeded with the return value. This variable is initialised to zero on reset but is not initialised when the function is entered. The value of this variable is visible in the variable watch window in the IDE.

Example:

```
x_squared(i)           // parameter i passed
{
  return (i*i);        // i*i returned
}

main()
{
  var z;
  z = x_squared(10); // call function, 100 returned
}
```

- ▶ *Note: Functions must be declared before they can be called.*
- ▶ *Note: Parameters and return values can only be single variables. Arrays are not allowed.*
- ▶ *Note: Function name declaration must follow the same rules as variable names*

main()

This is a special form of function. It must be present in all programs and it is the point from which the program starts execution. Program execution finishes at the end of 'main()'. No parameters can be passed to it and no return value is allowed. Local variables declared at the beginning of 'main()' are accessible only from within 'main()'.

Error Handler

errorh()

This is the user error handler sub routine. It is optional. If an operational error occurs and the user error handler is not present the node will fill the 'error', 'line_with_error' and 'node_with_error' variables. It then enters the default error handler and stops program execution at the point of the error. Displays an error code on its Eight Segment Display (if present) and broadcasts the error condition across the network causing all other programs to stop execution. This is the default condition.

The programmer can override the default error handler by writing an 'errorh()' sub routine. This code will be executed before entering the default error handler and if the 'error' variable is set to zero before 'errorh()' exits the default error handler will take no action.

Example:

```
errorh ()
{
    if(error == 13) // over ride error handler if error == 13
    {
        puts("End stop reached");
        error = 0;
    };
}
```

Interrupts

The E-node range supports 12 different interrupts. 10 activated from digital I/O, 1 activated by character receipt on the uart and 1 activated by a message receipt from another node. Parameters cannot be passed to interrupts and return values are not allowed. The main program is temporarily suspended whilst interrupts handlers are active.

▶ *Interrupts are acknowledged according to their priority:*

```
int0h    =    Highest priority
msgH     =    Lowest priority
```

▶ *Only a single interrupt can be active at any one time.*

▶ *Pending interrupts are acknowledged according to their priority.*

▶ *If a particular interrupt is called for a second time whilst that same interrupt is either pending or currently running the second call will be ignored and discarded.*

Interrupt Control

Interrupts are controlled by the setting and clearing of bit fields in the 'ints' variable. A range of 'BUILTIN' constants are provided for this purpose. For input interrupts the edge type (either rising or falling) can be selected. Individual interrupts must be enabled for them to become active and the global enable bit 'GLOBAL_EN' must also be set. Clearing the global enable bit disables all interrupts regardless of the individual setting status.

Selecting interrupt input polarity

```
Polarity bit = 1 selects RISING EDGE
Polarity bit = 0 selects FALLING EDGE
```

Example:

```
ints[INT0_POL] = 1; // select rising edge
```

Enabling Interrupts

Global enabling is achieved using the 'GLOBAL_EN' constant. Setting the bit enables and clearing the bit disables.

Example:

```
ints[GLOBAL_EN] = 1; // enable all individually enabled interrupts
ints[GLOBAL_EN] = 0; // disable all interrupts
```

Individual interrupts must be enabled in addition to the global enable. Use the 'INT0_EN' to 'INT9_EN', 'UARTH_EN' and 'MSGH_EN' constants. Setting the bit enables and clearing the bit disables.

Example:

```
ints[INT0_EN] = 1; // enable interrupt 0
ints[INT1_EN] = 0; // disable interrupt 1
ints[UARTH_EN] = 1; // enable uart interrupt
```

int0h() - int9h()

These are the optional pre-defined interrupt handlers for digital inputs 0 through 9. They can be called using either rising or falling edges occurring on the inputs (see Interrupt Control above)

uarth()

This is the optional pre-defined interrupt handler for character receipt on the uart.

Example:

```
uarth()
{
    if(charin())
    {
        //user code
    };
}
```

msggh()

This is the optional pre-defined interrupt handler for message receipt. If enabled this interrupt will be executed when a message is received from another node and placed in the message buffer.

- ▶ Messages are sent from node to node using the 'msg' statement.
- ▶ The format of a single message is the same as a standard variable.

Encoder Position Capture Interrupt

Control-C supports fast encoder position capture for motion capable nodes. A rising or falling edge on input[0] can be selected to initiate the capture, the result of which is placed in the pre-defined variable 'cappos'. To prevent false triggering or exclude unwanted input[0] triggers a registration window feature is provided.

Encoder capture polarity

To select either rising or falling edge polarity for the encoder capture use the 'status[]' command as follows:-

```
status [CAP_POL] = RISING; // encoder capture edge polarity = rising
status [CAP_POL] = FALLING; // encoder capture edge polarity = falling
```

Enabling Encoder Capture

The encoder position capture function is entirely independent of the other interrupt functions on the E-Nodes. To enable a capture use the 'status[]' command as follows.

```
status [CAPTURED] = 0; // clear captured flag and enable a new capture
```

This clears the captured flag and enables another capture to take place. When a new capture is performed the captured value is placed in the pre-defined variable 'cappos' and the captured flag is set to 1. No further captures will take place until the captured flag is set to 0.

To determine if a capture has taken place test the captured flag as follows:-

```
if (status [CAPTURED])
{
    // do this
};

// or use this

while (!status [CAPTURED]) {};
```

Registration Window

To prevent unwanted triggers a registration window facility is provided. The pre-defined variables 'cap_window_hi' and 'cap_window_lo' can contain values representing the most +ve and most -ve encoder values. If the capture interrupt occurs when the encoder position is within these bounds the position will be captured and reported. If outside these bounds the interrupt will be ignored.

► If cap_window_hi and cap_window_lo are both set to 0 the registration window facility is disabled.

Input[0] interrupt filtering

Encoder capture interrupts generated by input[0] are vary fast acting. This interrupt is processed separately from the normal interrupts and is not influenced by the digital filtering facility. However to help prevent noise and false triggers occurring a blanking facility is provided. Please refer to the definition for 'input0_blank' given in the pre-defined variables section.

Statement

Assignment operators

Control-C has a range of standard assignment operators. Following is a set of examples to illustrate their use.

Example:

```
main ()
{
    var t;

    t++;           // t = t + 1   ( auto increment )
    t--;           // t = t - 1   ( auto decrement )
    t = 1;         // t = 1       ( assignment )
    t &= 0xF;      // t = t & 0xF ( & is bitwise 'AND' )
    t |= 0xF;      // t = t | 0xF ( | is bitwise 'OR' )
    t *= 5;        // t = t * 5   ( multiplication )
    t /= 5;        // t = t / 5   ( division )
    t += 5;        // t = t + 5   ( addition )
    t -= 5;        // t = t - 5   ( subtraction )
    t ^= 0xF;      // t = t ^ 0xF ( ^ is bitwise 'XOR' )
    t %= 6;        // t = t % 6   ( % is 'MOD' )
    t <<= 2;       // t = t << 2 ( << is bitwise 'SHIFT LEFT' )
    t >>= 2;       // t = t >> 2 ( >> is bitwise 'SHIFT RIGHT' )
}
```

putc

putc(EXPRESSION1 : EXPRESSION2)

Outputs a character to the uart where:

EXPRESSION1 = a character
EXPRESSION2 = optional node address (default address used if absent).

Example:

```
putc('A':4);     // send 'A' to uart on address 4
putc('A');       // send 'A' to uart of default address
```

puts

puts(STRING : EXPRESSION1)

Outputs a string to the uart where:

STRING = ASCII text string or a variable or constant, with possible formatting
EXPRESSION1 = optional node address (default address used if absent).

► *If STRING is a variable or a constant it will automatically be converted to its string equivalent.*

Example:

```
puts ("hello":4); // send "hello" to uart on address 4
puts (time); // send the value of time converted to a string
// to the uart of the default address
puts (%3.2,time); // send the value of time converted to a string
// with the format xxx.xx
// to the uart of the default address
```

if

if(EXPRESSION) STATEMENT else STATEMENT

Standard 'C' construct. If the expression evaluates to true the first optional statement is executed else the second optional statement is executed.

Example:

```
if(z) time = 0 else time = 1 ;
```

delay

delay(EXPRESSION)

Delay causes the program to halt execution for a given time period where EXPRESSION is that time period in seconds. The smallest possible delay is 500 microseconds.

Example:

```
delay(10); // delay 10 seconds
```

► *During the delay the following operations will continue:*

- 1. Any motion commands in the queue will be executed.*
- 2. User interrupts will continue to be serviced.*
- 3. Key presses on 'SCREEN' nodes will be entered into the key buffer.*
- 4. The error handler will still be entered in response to errors.*

beep

beep(EXPRESSION, EXPRESSION)

Causes a beep to be emitted on 'SCREEN' nodes where the first expression is the frequency in the range 1000 hertz to 5000 hertz and the second expression is the duration in milliseconds in the range 0 to 10000.

Example:

```
beep(); // issue a default beep
```

► If both expressions are omitted the default values given in 'beep_frequency' and 'beep_duration' will be used.

return

return EXPRESSION

Causes immediate return from within a function/sub-routine. If the optional expression is used the final value of the expression is placed in the return variable (with the same name as the calling function) and passed out to the calling statement.

Example:

```
f()
{
  if(time>10)
  {
    return TRUE;
  }
  else
  {
    return FALSE;
  };

  // the above could also be written thus

  if(time>10) return TRUE else return FALSE;
}
```

while

while(EXPRESSION) STATEMENT

While the expression is greater than zero the following statement will be executed. The expression is always evaluated before executing the statement.

Example: The while loop exits when time is greater than 10 seconds.

```
var notdone;
notdone = TRUE;

while(notdone)
{
  if(time>10) notdone = FALSE;
};
```

for

for(STATEMENT1; EXPRESSION1; STATEMENT2) STATEMENT3

The 'for' loop is a common feature of 'C' programs. For clarity it may be re-stated as:

for(initialisation; condition; increment) statement

The initialisation term is generally an assignment statement that is used to initialise the loop control variable. The condition term is an expression that determines when the loop exits. The increment term defines how the loop control variable changes each time the loop is repeated. The 'for' loop continues to execute as long as the condition is greater than zero. The condition is tested before the statement is executed and the increment is performed after the statement is executed.

Example:

```
main ()
{
    var i,a[10];
    for(i=0;i<10;i++) a[i] = 0; // fill array with 0
}
```

cam - electronic cam - (queued command)

cam(GLOBAL[start], GLOBAL[finish], EXPRESSION)

This command provides an electronic cam profile motion command. It is used in conjunction with a pre-defined global array of demand positions. GLOBAL is the array name. [start] is the start point in the array and [finish] is the end point in the array. The expression gives the number of control loop cycles between increments through the array.

- ▶ During execution the 'mode' variable will report 'CAM'.
- ▶ 'cam' will only work on nodes with motion control capability and is ignored by those that don't.
- ▶ 'cam' is a motion command and will be stored in the motion command queue to await execution in the correct order.
- ▶ The array used for 'cam' MUST be global and MUST reside on the node that executes the command. The declared address for the array will automatically determine the address to which the command will be sent for execution.
- ▶ 'cam' is a queued command. When a 'cam' command reaches the head of the queue it can take one of several actions. In all cases when the command has loaded it will automatically run if 'status[AUTORUN]' is set. Otherwise it will wait for the start command 'status[START_MOVE]=1'. The cases are:-
 1. If no motion command is currently running it will automatically load.
 2. If an 'am', 'rm', 'va', 'vr' or 'cam' command is currently running it will wait for the command to complete and then load.
 3. If an 'vm', 'tm', 'pm' or 'gr' command is currently running it will stop the command and then load.
- ▶ NOTE: If status[AUTOLOAD] is not set no commands are loaded from the command queue.

When 'cam' is first executed the node will position the motor at the position given by '[start]' and then increment through the array until it reaches '[finish]'. For each increment it will wait the given number of control loop cycles. If this number is greater than 1 it will perform linear interpolation between the current and next position in the array.

'cam' uses 'position mode' to control the motor. This is a point to point control method with no velocity profiling. Individual increments are velocity limited to the value given in 'vel' and subject to ratcheting if enabled. Please refer to the description of 'pm' mode for more detail.

The following example shows how to generate a sinusoidal cam profile in two ways. The first does not use linear interpolation. The second uses linear interpolation to reduce the array size. With the second method the sinusoid will actually be made up from a series of sort straight moves and the programmer must determine if this degradation in smoothness is worth the reduction in array size. In many cases the inertia of the motor and load effectively smooth the profile to an acceptable level.

Example:

```
var a[1000];

main()
{
  var i;
  for(i=0;i<1000;i++) a[i] = 200 * sin(i*0.36);
  cam(a[0],a[999],1); // without linear interpolation
  for(i=0;i<100;i++) a[i] = 200 * sin(i*3.6);
  cam(a[0],a[99],10); // with linear interpolation
}
```

setuart

setuart(EXP1, EXP2, EXP3, EXP4, EXP5: EXP6)

Resets and configures the uart where:

EXP1 = Baud rate (maximum = 115200)
EXP2 = Parity, 'o' or 'O' = odd, 'n' or 'N' = none, 'e' or 'E' = even.
EXP3 = Number of data bits, 7 or 8.
EXP4 = Number of stop bits, 1 or 2.
EXP5 = 'RS232' or 'RS485' format (optional - RS232 is default if missing).
EXP6 = Destination node address (optional - default used if missing)

Example:

```
setuart(1200, 'E', 8, 1, RS485:0) // 1200 baud, even parity
// 8 data bits, 1 stop bit
// rs485 format on address = 0
```

break

When encountered 'break' forces the termination of any of the loop statements, 'for', 'while' and 'do'. Execution continues with the next statement after the loop. If loops are nested only the loop containing the executed 'break' is terminated. Outer loops will not be effected.

Example:

```
main()
{
    var i;
    for(i=0;i<100;i++)
    {
        if(time > 15) break;
    }
}
```

do

do STATEMENT while (EXPRESSION)

The statement is executed while the expression is greater than zero. The expression is always evaluated after executing the statement.

Example: The do loop exits when time is greater than 10 seconds.

```
var notdone;
notdone = TRUE;
do
{
    if(time>10) notdone = FALSE;
}
while (notdone);
```

va - vector absolute - (queued command)

va(EXPRESSION1: EXPRESSION2, EXPRESSION3: EXPRESSION4)

Generates a vector absolute move along 'x' and 'y' axes where:

EXPRESSION1 = absolute position for the 'x' axis
EXPRESSION2 = node address for the 'x' axis
EXPRESSION3 = absolute position for the 'y' axis
EXPRESSION4 = node address for the 'y' axis

A vector move is one where the two associated axes start and finish the move at the same time and preserve the 'x' axis maximum velocity and accelerations along the vector path.

Example:

```
va(1000:1,1000:2); // node 1 and 2 move to absolute 1000
```

- ▶ During execution the 'mode' variable will report 'VA'.
- ▶ Vector moves can be 'linked'. If linking is on, only the first and last vectors in a series of vectors use acceleration and deceleration respectively. The inner vectors remain at maximum velocity and thus are seamlessly linked.
- ▶ Motion commands are ignored on nodes without motion capability. For vector commands both the 'x' and 'y' addresses MUST be motion capable. If both are not the command is simply ignored. If only one is motion capable then the move sequence will stop whilst one waits infinitely for the other.
- ▶ 'va' is a queued command. When issued, the 'va' command will enter the motion command queues on the nominated node addresses. When the 'x' axis command reaches the head of its queue it will automatically wait for the 'y' axis command to reach the head of its queue. At this point the 'x' axis command can take one of several actions. In all cases when the command has loaded it will automatically run if 'status[AUTORUN]' is set. Otherwise it will wait for the start command 'status[START_MOVE]=1'. The cases are:-
 1. If no motion command is currently running it will automatically load.
 2. If a 'am', 'rm', 'va', 'vr' or 'cam' command is currently running it will wait for the command to complete and then load.
 3. If a 'vm', 'tm', 'pm' or 'gr' command is currently running it will stop the command and then load.
- ▶ NOTE: If status[AUTOLOAD] is not set no commands are loaded from the command queue.

vr - vector relative - (queued command)

vr(EXPRESSION1: EXPRESSION2, EXPRESSION3: EXPRESSION4)

Generates a vector relative move along 'x' and 'y' axes where:

EXPRESSION1 = relative position for the 'x' axis
EXPRESSION2 = node address for the 'x' axis
EXPRESSION3 = relative position for the 'y' axis
EXPRESSION4 = node address for the 'y' axis

- ▶ During execution the 'mode' variable will report 'VR'.

Same as 'va' in all other respects.

am - absolute move - (queued command)

am(EXPRESSION1: EXPRESSION2, (optionally more))

Generates one or more absolute position moves on the addressed nodes, where:

EXPRESSION1 = absolute position (overwrites 'demand')
EXPRESSION2 = node address (optional - default used if missing)

When executed the one or more absolute positions are sent to the motion command queues of their respective nodes. They are then performed in the order determined by their queue position. If the programmer wishes for the moves to take place simultaneously they must ensure the queues are empty before issuing the command.

'am' moves are velocity profiled and obey 'accel', 'vel' and 'decel' values on the addressed nodes.

Example:

```
am(1000:1,1000:2); // node 1 and 2 move to absolute 1000
```

- ▶ During execution the 'mode' variable will report 'ABS'
- ▶ When in absolute move mode changing the 'demand' variable will immediately change the target position but changing the 'offset' will have no effect.
- ▶ 'am' is a queued command. When a 'am' command reaches the head of the queue it can take one of several actions. In all cases when the command has loaded it will automatically run if 'status[AUTORUN]' is set. Otherwise it will wait for the start command 'status[START_MOVE]=1'. The cases are:-
 1. If no motion command is currently running it will automatically load.
 2. If a 'am', 'rm', 'va', 'vr' or 'cam' command is currently running it will wait for the command to complete and then load.
 3. If a 'vm', 'tm', 'pm' or 'gr' command is currently running it will stop the command and then load.
- ▶ NOTE: If status[AUTOLOAD] is not set no commands are loaded from the command queue.

rm - relative move - (queued command)

rm(EXPRESSION1: EXPRESSION2, (optionally more))

Generates one or more relative position moves on the addressed nodes, where:

EXPRESSION1 = relative position (overwrites the 'offset' variable' and updates 'demand')
EXPRESSION2 = node address (optional - default used if missing)

- ▶ During execution the 'mode' variable will report 'REL'.
- ▶ When in relative move mode changing the 'demand' variable will immediately change the target position and change the 'offset' to the correct value relative to the start position. Changing the 'offset' will change the 'demand' position to reflect the new offset value.
- ▶ 'rm' is a queued command. When a 'rm' command reaches the head of the queue it can take one of several actions. In all cases when the command has loaded it will automatically run if 'status[AUTORUN]' is set. Otherwise it will wait for the start command 'status[START_MOVE]=1'. The cases are:-
 1. If no motion command is currently running it will automatically load.
 2. If a 'am', 'rm', 'va', 'vr' or 'cam' command is currently running it will wait for the command to complete and then load.
 3. If a 'vm', 'tm', 'pm' or 'gr' command is currently running it will stop the command and then load.

Same as 'am' in all other respects.

vm - velocity mode - (queued command)

vm(EXPRESSION1: EXPRESSION2, (optionally more))

Enters velocity controlled move mode on one or more addressed nodes, where:

EXPRESSION1 = velocity (overwrites the 'vel' variable)
EXPRESSION2 = node address (optional - default used if missing)

When executed the one or more velocity commands are sent to the motion command queues of their respective nodes. They are then performed in the order determined by their queue position. If the programmer wishes for the moves to take place simultaneously they must ensure the queues are empty before issuing the command.

'vm' moves are velocity profiled and obey 'accel', 'vel' and 'decel' values.

Example:

```
vm (20:1,30:2) // set velmode on node 1 & 2
```

- ▶ *During execution the 'mode' variable will report 'VEL'.*
- ▶ *When in velocity mode the maximum velocity may be varied directly by changing the 'vel' variable and the 'demand' variable is updated with the latest absolute position.*
- ▶ *'vm' is a queued command. When a 'vm' command reaches the head of the queue it can take one of several actions. In all cases when the command has loaded it will automatically run if 'status[AUTORUN]' is set. Otherwise it will wait for the start command 'status[START_MOVE]=1'. The cases are:-*
 1. *If no motion command is currently running it will automatically load.*
 2. *If a 'am', 'rm', 'va', 'vr' or 'cam' command is currently running it will wait for the command to complete and then load.*
 3. *if a 'tm', 'pm' or 'gr' command is currently running it will stop the command and then load.*
- ▶ *NOTE: If status[AUTOLOAD] is not set no commands are loaded from the command queue.*

tm - torque mode - (queued command)

tm(EXPRESSION1: EXPRESSION2, (optionally more))

Enters torque controlled move mode on one or more addressed nodes, where:

EXPRESSION1 = torque (-100 to 100) (overwrites the 'torque' variable)
EXPRESSION2 = node address (optional - default used if missing)

When executed the one or more torque commands are sent to the motion command queues of their respective nodes. They are then performed in the order determined by their queue position. If the programmer wishes for the moves to take place simultaneously they must ensure the queues are empty before issuing the command.

Torque mode assumes the node is controlling a motor/amplifier set up for torque mode operation. In torque mode the torque value given is translated into a voltage and output on the DAC0 analogue drive output.

torque value	DAC0 output
100	10v
0	0v
-100	-10v

Example:

```
tm(20:1,80:2) // set torque mode on node 1 & 2. 2 and 8 volts  
// respectively
```

- ▶ During execution the 'mode' variable will report 'TOR'.
- ▶ When in torque mode the demand position is updated with the latest positional value from 'enc0'.
- ▶ 'tm' is a queued command. When a 'tm' command reaches the head of the queue it can take one of several actions. In all cases when the command has loaded it will automatically run if 'status[AUTORUN]' is set. Otherwise it will wait for the start command 'status[START_MOVE]=1'. The cases are:-
 1. If no motion command is currently running it will automatically load.
 2. If a 'am', 'rm', 'va', 'vr' or 'cam' command is currently running it will wait for the command to complete and then load.
 3. if a 'vm', 'pm' or 'gr' command is currently running it will stop the command and then load.
- ▶ NOTE: If status[AUTOLOAD] is not set no commands are loaded from the command queue.

pm - position mode - (queued command)

pm(EXPRESSION1: EXPRESSION2, (optionally more))

Enters position controlled move mode on one or more addressed nodes, where:

EXPRESSION1 = absolute demand position (overwrites the 'demand' variable)
EXPRESSION2 = node address (optional - default used if missing)

When executed the one or more position commands are sent to the motion command queues of their respective nodes. They are then performed in the order determined by their queue position. If the programmer wishes for the moves to take place simultaneously they must ensure the queues are empty before issuing the command.

Position controlled moves are simple point to point moves. There is no velocity profiling and within the limits of the system, maximum power will be used for acceleration and deceleration whilst moving to the new absolute position. The maximum velocity set by the 'vel' value is observed and the move may also be limited by ratcheting if enabled.

Example:

```
pm(2000:1,8000:2) // immediate move to 2000 on node 1  
                // and 8000 on node 2
```

▶ During execution the 'mode' variable will report 'POSIT'.

▶ 'pm' is a queued command. When a 'pm' command reaches the head of the queue it can take one of several actions. In all cases when the command has loaded it will automatically run if 'status[AUTORUN]' is set. Otherwise it will wait for the start command 'status[START_MOVE]=1'. The cases are:-

1. If no motion command is currently running it will automatically load.
2. If 'pm' or 'gr' command is currently running it will immediately load.
3. If a 'am', 'rm', 'va', 'vr' or 'cam' command is currently running it will wait for the command to complete and then load.
4. if a 'vm' or 'tm', command is currently running it will stop the command and then load.

▶ NOTE: If status[AUTOLOAD] is not set no commands are loaded from the command queue.

output

output[EXPRESSION1: EXPRESSION2 (optional)] = EXPRESSION3

Writes a value to a given digital output port on a given node, where:

EXPRESSION1 = digital output number
EXPRESSION2 = node address (optional - default used if missing)
EXPRESSION3 = value (decimal term ignored)(see below)

▶ If the value of the integer term of EXPRESSION3 is non zero the output will be set and the output voltage will either float or be driven to 24v depending on the output type. If the value is zero the output will be cleared the output voltage will either float or be driven to 0v depending on the output type.

▶ The voltage outputs for 1 and 0 states vary dependant on the type of node

NPN type

Output = 1 = Output voltage floating (weak pull up to 24v)
Output = 0 = Output voltage is driven low (0v)

PNP type

Output = 1 = Output voltage is driven high (24v)
Output = 0 = Output voltage floating (weak pull down to 0v)

Example:

```
output[0]=1; // output 0 is set
output[0]=0; // output 0 is cleared
output[0]=-11; // output 0 is set
output[0]=10; // output 0 is set
```

status

status[EXPRESSION1: EXPRESSION2 (optional)] = EXPRESSION3

Writes a value to a given status bit on a given node, where:

EXPRESSION1 = status bit number
EXPRESSION2 = node address (optional - default used if missing)
EXPRESSION3 = value (decimal term ignored)(see below)

Example:

```
status[ENABLE_DAC0] = TRUE; // enable dac0
status[AUTORUN] = TRUE; // enable auto run for motion commands
```

▶ If the value of the integer term of EXPRESSION3 is non zero the 'status' bit will be set. If the value is zero the 'status' bit will be cleared.

▶ The variable 'status' controls many features on the E-node. Please refer to description of the variable for detailed information on the individual bit functions.

fram

`fram[EXPRESSION1: EXPRESSION2 (optional)] = EXPRESSION3`

E-nodes are equipped with non volatile memory called 'Ferroelectric Ram ' or fram. It is partitioned to resemble an array of variables and provides the means to permanently save variable values. Calling 'fram' writes a value to a given fram array location on a given node, where:

EXPRESSION1 = fram array location
EXPRESSION2 = node address for the fram store (optional - default used if missing)
EXPRESSION3 = value to be saved.

Example:

```
fram[0] = time; // save the current time to fram location 0
```

► *The minimum size of the fram' store is 1000 variables. Future designs of the E-node may have greater capacities. Please refer to E-node specifications for updated information.*

ints

`ints[EXPRESSION1: EXPRESSION2 (optional)] = EXPRESSION3`

Writes a value to a given 'ints' variable bit on a given node, where:

EXPRESSION1 = ints bit number (see 'BUILTIN')
EXPRESSION2 = node address (optional - default used if missing)
EXPRESSION3 = value (decimal term ignored)(see below)

Example:

```
ints[INT0_POL] = RISING; // set interrupt 0 polarity to rising  
ints[INT0_EN] = TRUE; // enable interrupt 0  
ints[GLOBAL_EN] = TRUE; // global interrupts enable
```

► *If the value of the integer term of EXPRESSION3 is non zero the 'ints' bit will be set. If the value is zero the 'ints' bit will be cleared.*

► *The variable 'ints' controls user software interrupts on the E-node. Please refer to the chapter on interrupts for further detail.*

gr - gear lock mode - (queued command)

gr(EXPRESSION1, EXPRESSION2, EXPRESSION3: EXPRESSION4 (optional))

Enters electronic gear locked move mode on the addressed node, where:

EXPRESSION1 = master_teeth
EXPRESSION2 = slave_teeth
EXPRESSION3 = Engagement mode
EXPRESSION4 = Node address (optional - default used if missing)

When executed the gear lock command is sent to the motion command queues of the addressed node. It will then be performed in the order determined by its queue position.

When the gear lock command is executed from the queue the node will control the position of its associated motor and encoder input (enc0) to that of encoder input 1 (enc1) multiplied by the given gear ration where:

gear_ratio = master_teeth / slave_teeth
demand position = enc1 * gear_ratio

It may be described as a slave shaft (enc0), geared to a master shaft (enc1).

For gear locking to work the node must be set up to receive regular updates for the value of the master encoder (enc1). This is accomplished by sharing the master encoder value over the E-net using 'broadcast variables'. The node reading the master encoder issues the following command:

```
tx_bcv0 = varnum(enc0); // issued by the master node
```

This causes the master encoder value to be broadcast every 'bcv_interval'. The controlling node issues the following command to receive the broadcasts:

```
rx_bcv0 = varnum(enc1); // issued by the slave node
```

This causes the controlling node to place the broadcast variable value 0 into the variable 'enc1'.

► *I most cases it is advisable to set the broadcast interval variable 'bcv_interval' to 1 control loop to ensure the encoder value is updated once every control loop cycle.*

Gear locking can be engaged in several different modes as specified by the 'Engagement mode' parameter. Variables 'mgpos', master gear position and 'sgpos', slave gear position are used for some of these. 'mgpos' and 'sgpos' are simple variables that must be initialised (if required) by the programmer prior to the execution of gear lock mode .

Mode	Operation
0	Normal gear locking takes place immediately using the current 'enc0' and 'enc1' positions.
1	Normal gear locking takes place when 'enc1' becomes GREATER THAN OR EQUAL TO 'mgpos'.
2	Normal gear locking takes place when 'enc1' becomes LESS THAN OR EQUAL TO 'mgpos'.
3	Registered gear locking takes place immediately. 'mgpos' and 'sgpos' are used for the initial engagement positions providing exact registration.

- 4 Registered gear locking takes place immediately after 'enc1 becomes GREATER THAN OR EQUAL TO 'mgpos'. 'mgpos and 'sgpos are then used for the initial engagement positions to provide exact registration
- 5 Registered gear locking takes place immediately after 'enc1 becomes LESS THAN OR EQUAL TO 'mgpos'. 'mgpos and 'sgpos are then used for the initial engagement positions to provide exact registration

During gear locked operation both 'master_teeth' and 'slave_teeth' can be varied at will to change the active gear ratio. The slave direction may also be changed by making either 'master_teeth' or 'slave_teeth' negative. Finally if either 'master_teeth' or 'slave_teeth' are set to zero the slave axis will halt (dis-engage) at the position the change was made. Thus once in gear lock mode the programmer can vary the gear ratio, change direction, dis-engage and re-engage the drive at will.

► *Note: For smooth operation of REGISTERED locking it is recommended that the programmer ensures that both the master and slave are very close to the positions in 'mgpos' and 'sgpos' before engagement. If this is not the case the slave will jump to the correct position, having calculated that position from 'mgpos', 'sgpos', 'enc0', 'enc1' and the gear ratio.*

► *Note: Gear locking uses position mode for controlling the motor position. Position controlled moves are simple point to point moves. There is no velocity profiling and within the limits of the system, maximum power will be used for acceleration and deceleration whilst moving to the new absolute position. A maximum velocity set by the 'vel' variable is observed and the move may also be limited by ratcheting if enabled.*

► *Note: If a 'pm' command is issued during a gear locked move gear locking will be exited immediately and the slave will move directly to the demanded position using position mode. This feature may be used to exit gear lock mode and stop the motor at an exact position.*

► *During execution the 'mode' variable will report 'GEAR' unless dis-engaged when it will report 'BRK' for 'braked'.*

► *'gr' is a queued command. When a 'gr' command reaches the head of the queue it can take one of several actions. In all cases when the command has loaded it will automatically run if 'status[AUTORUN]' is set. Otherwise it will wait for the start command 'status[START_MOVE]=1'. The cases are:-*

1. *If no motion command is currently running it will automatically load.*
2. *If 'pm' or 'gr' command is currently running it will immediately load.*
3. *If a 'am', 'rm', 'va', 'vr' or 'cam' command is currently running it will wait for the command to complete and then load.*
4. *if a 'vm' or 'tm', command is currently running it will stop the command and then load.*

► *NOTE: If status[AUTOLOAD] is not set no commands are loaded from the command queue.*

Example:

Registered gear locking when 'enc1' >= 'mgpos'

```
mgpos = 1000;
sgpos = 2000;
gr(100,200,4); // ratio = 0.5, reg mode 4, default node address
```

pulse

`pulse(EXPRESSION1: EXPRESSION2 (optional) , EXPRESSION3)`

Generates an immediate pulse on the given output on the given node address, where:

EXPRESSION1 = output number
EXPRESSION2 = node address (optional - default used if missing)
EXPRESSION3 = pulse width in milliseconds

A pulse is generated by immediately toggling the output, then waiting waiting the required pulse width time period, then toggling the output again. Thus the initial state of the output determines whether the pulse is high going or low going.

Example:

```
pulse(0,10); // 10 millisecond pulse on output 0, default address
```

spi

`spi(GLOBAL[])`

The hardware interface known as the 'Serial Peripheral Interface' can be made available for specific user applications. The 'spi' command provides a means to both write and read data to an 'spi' peripheral device. 'spi' uses an array containing all the command and data information required. For more information please contact Etrol Ltd.

dot

`dot(EXPRESSION1, EXPRESSION2)`

Sets a single pixel on the screen of a 'SCREEN' node to the colour given by 'fcolour', where:

EXPRESSION1 = x co-ordinate
EXPRESSION2 = y co-ordinate

Example:

```
fcolour = RED; // fore ground colour = RED  
dot(0,0); // top left hand pixel = RED
```

▶ Co-ordinates start at the origin in the top left hand corner of the screen. 'x' is positive left to right. 'y' is positive top to bottom. An error is generated if the values are outside the size of the screen.

▶ This command has to be initiated on a 'SCREEN' node. There is no associated address.

line

line(EXPRESSION1, EXPRESSION2, EXPRESSION3, EXPRESSION4)

Generates a line on the screen of a 'SCREEN' node using the colour given by 'fcolour', where:

EXPRESSION1 = from x co-ordinate
EXPRESSION2 = from y co-ordinate
EXPRESSION3 = to x co-ordinate
EXPRESSION4 = to y co-ordinate

Example:

```
line(20,20,100,100); // line from 20,20 to 100,100
```

▶ Co-ordinates start at the origin in the top left hand corner of the screen. 'x' is positive left to right. 'y' is positive top to bottom. An error is generated if the values are outside the size of the screen.

▶ This command has to be initiated on a 'SCREEN' node. There is no associated address.

rectangle

rectangle(EXPRESSION1, EXPRESSION2, EXPRESSION3, EXPRESSION4)

Generates a rectangle on the screen of a 'SCREEN' node using the colour given by 'fcolour', where:

EXPRESSION1 = from x co-ordinate
EXPRESSION2 = from y co-ordinate
EXPRESSION3 = to x co-ordinate
EXPRESSION4 = to y co-ordinate

Example:

```
rectangle(20,20,100,100); // rectangle from 20,20 to 100,100
```

▶ Co-ordinates start at the origin in the top left hand corner of the screen. 'x' is positive left to right. 'y' is positive top to bottom. An error is generated if the values are outside the size of the screen.

▶ This command has to be initiated on a 'SCREEN' node. There is no associated address.

block

`block(EXPRESSION1, EXPRESSION2, EXPRESSION3, EXPRESSION4)`

Generates a solid rectangular block on the screen of a 'SCREEN' node using the colour given by 'bcolour', where:

EXPRESSION1 = from x co-ordinate
EXPRESSION2 = from y co-ordinate
EXPRESSION3 = to x co-ordinate
EXPRESSION4 = to y co-ordinate

Example:

```
block(20,20,100,100); // solid block from 20,20 to 100,100
```

▶ *Co-ordinates start at the origin in the top left hand corner of the screen. 'x' is positive left to right. 'y' is positive top to bottom. An error is generated if the values are outside the size of the screen.*

▶ *This command has to be initiated on a 'SCREEN' node. There is no associated address.*

circle

`circle(EXPRESSION1, EXPRESSION2, EXPRESSION3)`

Generates a circle on the screen of a 'SCREEN' node using the colour given by 'fcolour', where:

EXPRESSION1 = centre x co-ordinate
EXPRESSION2 = centre y co-ordinate
EXPRESSION3 = radius in pixels

Example:

```
circle(100,100,10); // circle centre at 100,100, radius 10
```

▶ *Co-ordinates start at the origin in the top left hand corner of the screen. 'x' is positive left to right. 'y' is positive top to bottom. An error is generated if the values are outside the size of the screen.*

▶ *This command has to be initiated on a 'SCREEN' node. There is no associated address.*

disc

disc(EXPRESSION1, EXPRESSION2, EXPRESSION3)

Generates a solid disc on the screen of a 'SCREEN' node using the colour given by 'fcolour, where:

EXPRESSION1 = centre x co-ordinate
EXPRESSION2 = centre y co-ordinate
EXPRESSION3 = radius in pixels

Example:

```
disc(100,100,10); // disc centre at 100,100, radius 10
```

▶ Co-ordinates start at the origin in the top left hand corner of the screen. 'x' is positive left to right. 'y' is positive top to bottom. An error is generated if the values are outside the size of the screen.

▶ This command has to be initiated on a 'SCREEN' node. There is no associated address.

cursor

cursor(EXPRESSION1, EXPRESSION2 (optional))

Controls the operation of the text cursor on the screen of a 'SCREEN' node, where:

EXPRESSION1 = cursor x co-ordinate (for two expression command)
required cursor setup (for one expression command)
EXPRESSION2 = cursor y co-ordinate (for two expression command) (optional)

'BUILTIN' constants are provided for use with cursor setup. These values can be 'OR-ED' together for a single setup command.

Example:

```
cursor(COFF); // cursor is off and setup cleared  
cursor(CON | CBLINK); // cursor on with blink  
cursor(20,20); // place cursor at 20,20
```

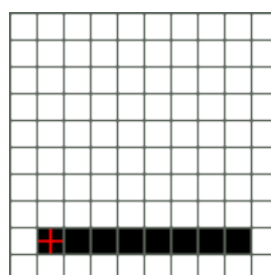
▶ The cursor size is automatically set to match the current font size.

▶ Co-ordinates start at the origin in the top left hand corner of the screen. 'x' is positive left to right. 'y' is positive top to bottom. An error is generated if the values are outside the size of the screen.

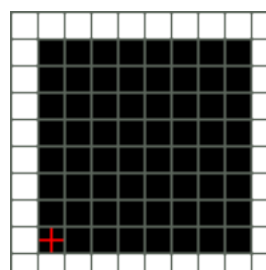
▶ This command has to be initiated on a 'SCREEN' node. There is no associated address.

▶ The 'x' and 'y' co-ordinates refer to the leftmost pixel along the bottom of the cursor.

Example: x, y position for:-



Line cursor



Block cursor

edges

edges(EXPRESSION1: EXPRESSION2 (optional), EXPRESSION3)

Initialises and starts the counting of digital input edges, where:

EXPRESSION1 = input number
EXPRESSION2 = node address (optional - default used if missing)
EXPRESSION3 = edge type, RISING, FALLING or BOTH.

When 'edges' is executed the edge count for the given input is cleared and the given edge type is selected. From that time onward the chosen edge types are counted. The current edge count is returned when 'edges' is called as a factor. 'BUILTIN' constants are provided for use with 'edges'.

Example:

```
var inedges;  
edges (1:1,BOTH);           // count both edge types, input[1] node 1  
delay (10);                 // wait 10 seconds  
inedges = edges (1:1);     // read the edge count
```

► *The edge count is updated every 500 microseconds. Therefore edge counting can only reliably record edges with a time period greater than 500 microseconds between edge changes.*

key

key(EXPRESSION1, EXPRESSION2)

or

key(EXPRESSION1, EXPRESSION2, EXPRESSION3, EXPRESSION4, EXPRESSION5,
EXPRESSION6, EXPRESSION7, EXPRESSION8, EXPRESSION9)

Used to control and setup touch keys on the touch screen of 'SCREEN' nodes.

Use the command with Two expressions for the control of keys, where:

EXPRESSION1 = key number or 'ALL'
EXPRESSION2 = required state, 'ON' or 'OFF'.

When a key is 'ON' the key will respond when touched and report the touched key number to the key buffer. When a key is 'OFF' the key will not respond to a touch and no report is made.

Use the command with Nine expressions to create new keys, where:

EXPRESSION1 = key number (reported when the key is touched) - Range 0 to 49
EXPRESSION2 = x co-ordinate (for key top left corner) - Range 0 to 319
EXPRESSION3 = y co-ordinate (for key top left corner) - Range 0 to 239
EXPRESSION4 = x co-ordinate (for key bottom right corner) - Range 0 to 319
EXPRESSION5 = y co-ordinate (for key bottom right corner) - Range 0 to 239
EXPRESSION6 = key text string (optional, null if missing)
EXPRESSION7 = key text x offset in pixels (optional, zero if missing)
EXPRESSION8 = key text y offset in pixels (optional, zero if missing)
EXPRESSION9 = visible flag, 1 = key is visible, 0 = key is invisible (optional, 1 if missing)

Expression1 specifies a key number. Every active key must have a unique key number. This value is placed in the key buffer when the key is touched.

The next four expressions specify the key size in pixels using x,y, co-ordinates for the top left and bottom right corners.

Expression6 is an optional key text string. When the key is written to the screen this text is placed centrally (if possible) on the key. Some fonts have variable character widths which makes it impossible to exactly centralise the text on the key surface (see Expression 7 and 8 below).

Expressions 7 & 8 are optional x and y offsets in pixels. These values are used to offset the key text from the calculated centralised position for fine correction of the text position.

Expression9 (optional) specifies the key visibility. If '1' the key is visible and responds in the normal way when touched - the key appears depressed and a beep is sounded. if '0' the key is invisible. When touched there is no visual change but the beep is sounded and the key number is reported to the key buffer.

Example:

```
key(0,0,0,100,50,"Hello"); // key number 0 in top right corner
// 100 by 50 with text = "hello"
key(1,101,0,200,50,"Press"); // another to the left
key(1,OFF); // turn key 1 off
key(1,ON); // turn key 1 on
key(ALL,OFF); // turn all keys off
```

► **Please Note: An error will result if the 'key number' or 'x' or 'y' co-ordinates are outside of the specified ranges.**

► Please refer to the definition for the variable 'keycolour' for information on how to set key colours.

- ▶ To completely remove a key from the screen the programmer must first make that key inactive and then overwrite the key area with the required visual.
- ▶ If key text is not provided it is the programmer's choice as to what is placed on the key surface. He may either place text using the 'cursor' and 'sputs' statements or he may place a bitmap image using the 'bitmap' statement.
- ▶ If a key is made invisible by setting Expression9 to zero the programmer can use it behind any feature he wishes. This method can provide hidden options or trick effects.
- ▶ Please refer to 'keypressed' and 'getkey' for additional information on using keys.
- ▶ When a key is touched only the border changes to give the effect of depression. The programmer is free to draw at any time new features on the key body.
- ▶ When a key is touched the beep() command is called to emit a beep from the screen.

sputch

sputch(EXPRESSION)

Places a single character to the screen of a 'SCREEN' node, where:

EXPRESSION = number representing an ASCII character

The character is placed at the current cursor position using the current 'font', 'fcolour' and 'bcolour' variable settings.

Example:

```
sputch ('A'); // put 'A' to the screen
sputch (65); // also puts 'A' to the screen 65 = ASCII 'A'
```

sputs

sputs(STRING)

Places a string to the screen of a 'SCREEN' node at the current cursor position using the current 'font', 'fcolour' and 'bcolour' variable settings., where:

STRING = ASCII text string or a variable or constant, with possible formatting

- ▶ Please refer to 'keypressed' and 'getkey' for additional information on using keys.

Example:

```
sputs ("hello"); // place "hello" on the screen
sputs (time); // the value of time converted to a string
// with default format placed on the screen
sputs (%3.2,time); // the value of time converted to a string
// with the format xxx.xx placed on the screen
```

clear

clear()

Clears the entire screen of a 'SCREEN' node by filling the screen with the 'bcolour' variable setting.

Example:

```
clear (); // clear screen using background colour 'bcolour'
```

bitmap

`bitmap(EXPRESSION1, EXPRESSION2, EXPRESSION3)`

Writes a bitmap from the bitmap store to the screen of a 'SCREEN' node, where:

EXPRESSION1 = bitmap store number
EXPRESSION2 = x co-ordinate (top left corner)
EXPRESSION3 = y co-ordinate (top left corner)

The bitmap must reside in the 'SCREEN' bitmap store. Bitmaps are stored in the bitmap store using the 'Screen Bitmap Manager' in the IDE.

Example:

```
bitmap(1,15,30); // put bitmap number 1 to screen at 15,30
```

► Please refer to the 'tcolour' variable definition for details on transparency.

vartos

`vartos(% EXPRESSION1 (optional), EXPRESSION2, EXPRESSION3)`

Converts a given variable or number with optional formatting into a string and places it in the given global array with a terminating null value, where:

EXPRESSION1 = format specifier (must be preceded by '%')
EXPRESSION2 = variable or number to convert to a string
EXPRESSION3 = global array start number (use with `varnum()`)

This command is provided for easy conversion of standard variable values into formatted strings. If provided the format specifier takes the form of a decimal number where the integer term specifies the number of places before the decimal point and the decimal term specifies the number of decimal places.

Example:

```
var a[10];  
main()  
{  
    vartos(%5.2,time,varnum(a)); // format time xxxxx.xx to array 'a'  
}
```

Using the above example the following 'time' values would be output thus:

time	output array values (ASCII values)
0.1	[32, 32, 32, 32, 48, 49, 48, 0]
15.32	[32, 32, 32, 49, 53, 51, 52, 0]

If the format specifier is omitted the default output will be the integer term only with no leading spaces.

► The output array must be declared with sufficient capacity to hold the output string plus the terminating null.

► Be certain to use the 'varnum()' function to correctly specify the start of the output array

String

Where 'string' is used in a statement it can take one of two forms, either:

% EXPRESSION1(optional), EXPRESSION2

or

" ASCII char string "

The first form is used when a variable value or number (EXPRESSION2) is given with an optional format specifier (% EXPRESSION1). When this form is encountered in a statement the variable value or number is automatically converted to a string. If the format specifier is given the integer term specifies the number of places before the decimal point and the decimal term specifies the number of places after the decimal point.

▶ *If the format specifier integer term is too small to display the full integer value it will automatically be extended.*

format specifier	output format
% 1.0	1 integer space and 1 decimal space
% 5	5 integer spaces only
% 4.2	4 integer space and 2 decimal spaces

▶ *If the format specifier is omitted the default format is %1. This will always display the integer term only with no leading spaces.*

Using the second form the required string is enclosed between double quotes. The string is used exactly as given.

Expression

An expression can consist of several 'factors' modified by operators. The precedence in the order of evaluation for these is defined in the syntax diagram by the 'P-LEVELS'.

operator	precedence
* / %	highest
+ -	
<< >>	
< <= > >=	
== !=	
&	
^	
&&	
	lowest

Where:

*	Multiply
/	Divide
%	Mod
+	Addition
-	Subtraction
<<	Left shift
>>	Right shift
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	equal (result is TRUE if equal, FALSE if not)
!=	Not equal (result is TRUE if Not equal, FALSE if equal)
&	Bitwise AND
^	Bitwise Exclusive OR
	Bitwise OR
&&	Logical AND (result is TRUE if left AND right operands are TRUE)
	Logical OR (result is TRUE if left OR right operands are TRUE)

Factor

! (not)

! FACTOR

Returns '1' if the FACTOR was zero and zero if the FACTOR was not zero.

Example: `while (!input[0]) {};`

- (negate)

- FACTOR

Returns the negated value of the FACTOR

Example: `master_teeth = -master_teeth;`

~ (ones complement)

~ FACTOR

Returns the ones complement (bitwise invert) of the FACTOR.

Example: `master_teeth = ~master_teeth;`

getch

getch(EXPRESSION (optional))

Returns a character from the uart buffer on a given node address, where:

EXPRESSION = node address (optional - default used if missing)

If there is no character in the uart buffer 'getch' will wait for one. If no character has arrived and the time out period (as given in variable 'timeout') has passed 'getch' will exit returning a null (0) and report the time out in the 'status' variable.

Example:

```
main ()
{
  var ch;
  timeout = 10; // time out period = 10 secs
  ch = getch(); // get any char from uart buffer
  if(status[TIMED_OUT]==0)
  {
    // process the character
  }
}
```

▶ Whilst waiting for a character program execution will be held up but enabled interrupts and queued motion commands will still function.

▶ If an error occurs whilst waiting for a character 'getch' will exit returning a null (0) and the program will enter the error handler routine.

charin

charin(EXPRESSION (optional))

Checks the uart buffer on a given node address for the presence of a character, where:

EXPRESSION = node address (optional - default used if missing)

If there is a character in the uart buffer 'charin' will return '1' else it will return '0'.

Example:

```
main()
{
  var ch;
  timeout = 10;      // time out period = 10 secs
  if(charin())
  {
    ch = getch();   // get a char from uart
  }
}
```

getmsg

getmsg(EXPRESSION (optional))

Returns a message from the message buffer on a given node address, where:

EXPRESSION = node address (optional - default used if missing)

▶ *A message takes the form of a standard variable.*

If there is no message in the message buffer 'getmsg' will wait for one. If no message has arrived and the time out period (as given in variable 'timeout') has passed 'getmsg' will exit returning a null (0) and report the time out in the 'status' variable.

Example:

```
main()
{
  var mess;
  timeout = 10;      // time out period = 10 secs
  mess = getmsg();   // get a message from the message buffer
  if(status[TIMED_OUT]==0)
  {
    // process the message
  }
}
```

▶ *Whilst waiting for a message program execution will be held up but enabled interrupts and queued motion commands will still function.*

▶ *If an error occurs whilst waiting for a message 'getmsg' will exit returning a null (0) and the program will enter the error handler routine.*

msgin

msgin(EXPRESSION (optional))

Checks the message buffer on a given node address for the presence of a message, where:

EXPRESSION = node address (optional - default used if missing)

If there is a message in the message buffer 'msgin' will return '1' else it will return '0'.

Example:

```
main()
{
  var mess;
  timeout = 10;           // time out period = 10 secs
  if(msgin())
  {
    mess = getmsg();     // get a message
  }
}
```

(EXPRESSION)

Returns the result of the evaluated EXPRESSION.

CONSTANT

Returns the value of the constant. Please refer to the section on constants for further detail.

DEFINED

Returns the value of a user defined constant. Please refer to the section on user defines constants for further detail.

FNAME

fname(EXPRESSIONS (optional))

Calls a function (sub-routine) with optional parameters and returns the return value from that function if one is provided.

The function must have been previously defined and the optional parameter list must match exactly.

Example:

```
double(x)           // define the function
{
  return 2*x;
}

main()
{
  var z;
  z = double(5);    // call the function
}
```

GLOBAL

GLOBAL: 0..9 (optional) [EXPRESSION] (optional) ++ or -- (optional)

Returns the value of the given global variable or global array element with optional post increment or decrement, where:

GLOBAL	=	defined variable name
:0..9	=	node address (optional)(default used if omitted)
[EXPRESSION]	=	array element (if the variable is an array)
++	=	auto increment variable by 1 after returning the value (optional)
--	=	auto decrement variable by 1 after returning the value (optional)

Example:

```
var z, a[8];

main()
{
  a[0] = 10;
  z = a[0]++;
}
```

In the example above the final values are 'z' equals 10 and 'a[0]' equals 11.

LOCAL

Same as GLOBAL above but using local variable.

PREDEFINED

Same as GLOBAL above but using PREDEFINED variables.

input

input[EXPRESSION1: EXPRESSION2 (optional)]

Returns the value of a given digital input on a given node address, where:

EXPRESSION1 = input number
EXPRESSION2 = node address (optional - default used if missing)

The returned value will be '1' or '0' .

► *The voltage inputs for '1' and '0' states vary dependant on the type of node*

NPN type

'0' = Input is driven low (0v)
'1' = Input floating (weak pull up to 24v)

PNP type

'0' = Input floating (weak pull down to 0v)
'1' = Input is driven high (24v)

Example:

```
main()
{
    var z;
    if(input[0])
    {
        z = time;
    }
}
```

output

output[EXPRESSION1: EXPRESSION2 (optional)]

Returns the value of a given digital output on a given node address, where:

EXPRESSION1 = output number
EXPRESSION2 = node address (optional - default used if missing)

The returned value will be '1' or '0'.

► *The voltage outputs for '1' and '0' states vary dependant on the type of node*

NPN type

'0' = Input is driven low (0v)
'1' = Input floating (weak pull up to 24v)

PNP type

'0' = Input floating (weak pull down to 0v)
'1' = Input is driven high (24v)

Example:

```
main()
{
    var z;
    if(output[0])
    {
        z = time;
    }
}
```

status

status[EXPRESSION1: EXPRESSION2 (optional)]

Returns the value of a given 'status' bit from a given node address, where:

EXPRESSION1 = status bit
EXPRESSION2 = node address (optional - default used if missing)

The returned value will be '1' or '0'. A range of 'BUILTIN' bit values are provided for use with 'status'. Please refer to the description of the 'status' variable for detailed information.

Example:

```
if(status[ENABLE_DAC0]) // if dac0 enabled
{
    am(1000);           // perform the move
}
```

fram

fram[EXPRESSION1: EXPRESSION2 (optional)]

Returns the value of a given fram array location from a given node address, where:

EXPRESSION1 = fram array location
EXPRESSION2 = node address (optional - default used if missing)

Example:

```
var z;  
z = fram[0]; // z = value at fram location 0
```

► The minimum size of the fram' store is 1000 variables. Future designs of the E-node may have greater capacities. Please refer to E-node specifications for updated information.

ints

ints[EXPRESSION1: EXPRESSION2 (optional)]

Returns the value of a given 'ints' bit from a given node address, where:

EXPRESSION1 = 'ints' bit number
EXPRESSION2 = node address (optional - default used if missing)

The returned value will be '1' or '0'. A range of 'BUILTIN' bit values are provided for use with 'ints'. Please refer to the description of the interrupts for detailed information.

Example:

```
if(ints[GLOBAL_EN]) // if global interrupts enabled  
{  
  putchar('G'); // send the Go message  
}
```

sin

sin(EXPRESSION)

Returns the sine of the EXPRESSION, where:

EXPRESSION = value to be converted in degrees.

Example:

```
var z;  
z = sin(230); // sine of 230 degrees
```

asin

asin(EXPRESSION)

Returns the arcsine of the EXPRESSION in degrees, where:

EXPRESSION = value to be converted.

Example:

```
var z;  
z = asin(0.2); // arcsine of 0.2
```

cos

cos(EXPRESSION)

Returns the cosine of the EXPRESSION, where:

EXPRESSION = value to be converted in degrees.

Example:

```
var z;  
z = cosin(230); // cosine of 230 degrees
```

acos

acos(EXPRESSION)

Returns the arccosine of the EXPRESSION in degrees, where:

EXPRESSION = value to be converted.

Example:

```
var z;  
z = acos(0.2); // arccosine of 0.2
```

tan

tan(EXPRESSION)

Returns the tangent of the EXPRESSION, where:

EXPRESSION = value to be converted in degrees.

Example:

```
var z;  
z = tan(45); // tangent of 45
```

atan

atan(EXPRESSION)

Returns the arctangent of the EXPRESSION in degrees, where:

EXPRESSION = value to be converted.

Example:

```
var z;  
z = atan(1); // arctangent of 1
```

sqrt

sqrt(EXPRESSION)

Returns the square root of the EXPRESSION, where:

EXPRESSION = value to be converted.

Example:

```
var z;  
z = sqrt(100); // square root of 100
```

varnum

varnum(GLOBAL/PREDEFINED: 0..9 (optional))

Returns the numerical identifier of GLOBAL and PREDEFINED variable names on a given node address, where:

GLOBAL/PREDEFINED = variable name
0..9 = node address (optional - default used if missing)

All GLOBAL and PREDEFINED variables are assigned a unique number by the compiler. The commands 'rx_bcv0', 'rx_bcv1', 'tx_bcv0', 'tx_bcv1' and 'vartos' need to know this number for correct function.

Example:

```
var myvarname;  
  
main() // varnum() gives the number of myvarname  
{  
  rx_bcv0 = varnum(myvarname);  
}
```

beep

beep()

Returns '1' if a beep is currently being emitted by a 'SCREEN' type node and '0' if not. Will always return '0' for non 'SCREEN' nodes.

► Used to prevent multiple beep commands from overriding each other.

Example:

```
if (beep ()) delay (1); // if beep sounding wait 1 sec
```

keypressed

keypressed()

Returns '1' if there is a key value in the key buffer of a 'SCREEN' node and '0' if not.

Example:

```
if (keypressed ()) // if key pressed on screen
{
    putch ('K'); // send message on uart
}
```

getkey

getkey()

Returns the key number of the first key in the key buffer. If the buffer is empty 'getkey' returns '-1'.

Example:

```
var keyis;
if (keypressed ()) // if key pressed on screen
{
    keyis = getkey (); // get the key number
}
```

xpos

xpos()

Returns the current 'x' axis position of the cursor on 'SCREEN' nodes.

Example:

```
var x,y;
x = xpos (); y = ypos ();
block (x,y,160,y-20);
```

► See 'Cursor()' definition for the location of the 'x' position within the cursor

ypos

ypos()

Returns the current 'y' axis position of the cursor on 'SCREEN' nodes.

Example:

```
var x, y;  
x = xpos(); y = ypos();  
block(x, y, 160, y-20);
```

► See 'Cursor()' definition for the location of the 'y' position within the cursor

stovar

stovar(EXPRESSION)

Returns the numerical value of an ASCII string representation of a number in a given global array, where:

EXPRESSION = global array start number (use with varnum())

The GLOBAL array containing the string representation of the number must have been previously defined, filled with the desired values and terminated with a null (0). Acceptable formats are:

number	string representation
1234	['1', '2', '3', '4', 0]
0.1	['0', '.', '1', 0]
.1	['.', '1', 0]
-0.1	['-', '0', '.', '1', 0]
-1	['-', '1', 0]
+1	['+', '1', 0]

Example:

```
var x, a[10];  
  
main()  
{  
  a[0] = '1';  
  a[1] = '.';  
  a[2] = '2';  
  a[3] = 0; // null terminator  
  x = stovar(varnum(a));  
}
```

When the above example has completed execution 'x' equals 1.2

edges

edges(EXPRESSION1: EXPRESSION2 (optional))

Returns the number of edges recorded for a given digital input on a given node address since the 'edges' statement was issued to set up the count, where:

EXPRESSION1 = input number
EXPRESSION2 = node address (optional - default used if missing)

Example:

```
main()
{
  var edgenum;
  edges(1,BOTH); // record both edges on input 1 from this time
  delay(10); // wait for 10 seconds
  edgenum = edges(1); // read the number of edges recorded
}
```

Appendix 1 - The Motion Command Queue

Motion Commands And The Motion Command Queue

Control-C provides a range of commands that produce changes in position of controlled motors. Examples are 'am()', 'cam()', 'gr()', etc. These are termed Motion Commands.

When a running program encounters a motion command it first determines on which node the command should be run and then sends the command to the 'Motion Command Queue' on that node. The program is then free to continue execution and is not held up by multiple motion commands.

If a motion command is sent to a node that is not capable of controlling a motor the command will be discarded and not placed in the queue.

The size of motion commands varies but generally about 20 commands can be held in the command queue. If the command queue overflows an error will result.

Queued motion commands are executed in the order they are received. When a command is executed from the queue the queue length is reduced by one and reported in the variable 'queue'.

When a motion command reaches the head of the queue and another motion command is currently running the action taken will vary depending on the types of the commands. For example if both commands are absolute move types ('am()') the second command will wait for the first command to fully complete before it executes. However there are types that will take control over the current command, stop it running and then immediately execute itself. Please refer to the detailed descriptions of the individual motion commands for more information.

► *Motion commands can be prevented from automatically loading from the command queue by clearing the status 'AUTOLOAD' bit. i.e. status[AUTOLOAD]=0.*

► *Motion commands can be removed from the head of the queue by altering the 'queue' variable. Please refer to the 'queue' variable description for further information.*

Move Synchronisation

A node can potentially receive motion commands from any programmed node on the network. These are queued and executed in the order of receipt. It is the programmers responsibility to ensure the correct synchronisation of multiple moves across multiple node. This is best achieved by knowing the current states of the nodes concerned and the precise timing of motion command issue.

It is worth noting that motion commands can run automatically when 'status[AUTORUN]==1' but if set to '0' they will not auto run. In this case when a motion command is loaded from the queue it will simply wait for the start command, 'status[START_MOVE]=1'.

Here are two simple ways to ensure move synchronisation:

1. Ensure the queues are empty on all concerned nodes. Issue a multiple move command with 'AUTORUN' set on all nodes. The E-nodes are very fast and will start all the moves within 1 or 2 milliseconds.
2. Ensure the queues are empty on all concerned nodes. Clear 'AUTORUN' on all concerned nodes. Issue the multiple move command. Read all nodes 'modes' to check they have loaded and are ready to run. Issue multiple 'status[START_MOVE]=1' commands.

► *Some motion commands have automatic synchronisation. Please refer to the detailed description of motion commands for further information.*

Appendix 2 - Error Code Definitions

All E-nodes have a pre-defined 'error' variable. If an error occurs the software will attempt to run the user error handler, 'errorh()'. If none has been provided the error is recorded in the 'error' variable and the error, the node address with the error and the program line number at which the error occurred are broadcast to all nodes.

► *If the program was compiled without breaks the 'line_with_error' variable will always report '0'*

A standard node that did not report the error will show a steady 'E' on the eight segment display. The node reporting the error displays 'E' followed by the error code on a repeated basis. The error code definitions are as follows:-

Error Code	Definition
0	No Node Errors
1	Error Value Exceeded
2	Serial Flash User Programme is Blank
3	Serial Flash System Programme is Blank
4	System Software Checksum Failure. Reprogramme System Software
5	User Program Checksum Failure. Reprogramme User Program
6	Serial Flash Programming Error
7	Serial Flash Erase Error
8	Fram Write Error
9	Following Error Exceeded
10	Serial Flash read returned blank result
11	Incorrect Mode Number
12	Software Pos Limit
13	Software Neg Limit
14	USB RxBuf Overflow
15	USB TxBuf Overflow
16	CAN RxBuf Overflow
17	not used
18	not used
19	not used
20	not used
21	not used
22	not used
23	CAN Bus Error
24	CAN BusOff Error
25	CAN Buf Error
26	Attempted Ethernet Access
27	ACK Receive Timeout
28	Data 256 Used Local
29	COM Buf Overflow
30	UART TxBuf Overflow
31	UART RxBuf Overflow
32	Fatal USP Op
33	POKE Stack Error
34	VAR Num To Large
35	Node Address Range Error
36	Max Vel Limit
37	Mult Overflow
38	Div By Zero
39	Max Accel Limit
40	Max Decel Limit
41	Max Prop Gain Limit
42	Max Diff Gain Limit
43	Max Integ Gail Limit
44	Max Torque Demand Limit
45	Max Accum Limit Limit
46	Max Following Error Limit Limit

Error Code	Definition
47	Max Dac Clip Limit
48	Max Dac Offset Limit
49	Max Dac Limit
50	Max Slave Teeth Limit
51	Max Master Teeth Limit
52	Varnum To Large
53	Parity Error
54	Baud Error
55	Databits Error
56	Stops Error
57	Fast Multiply Overflow
58	Max Ipf Limit
59	Max Ratchet Limit
60	Enable Num Error
61	MP Limit Vel Error
62	MN Limit Vel Error
63	Bit Num Error
64	Output Pulse Num Error
65	MoveQ Overflow
66	Link ON Error
67	Link OFF Error
68	Stramp ON Error
69	Stramp OFF Error
70	Response Timeout
71	Wrong Response
72	Unknown Command
73	Zero Encoder Whilst Active
74	Message Buf Overflow
75	Shift Error
76	Polarity Num Error
77	Max Input0 Blank Limit
78	Com Buf Underflow
79	MoveQ Underflow
80	CAM Interval Error
81	CAM Varnum Error
82	GEAR Mode Error
83	Max MoveQ Limit
84	Comms Retry Failure
85	Same Address Error
86	Chord Angle Error
87	Residual Angle Error
88	ARC Angle Error
89	Writes Blocked
90	Moves Blocked
91	Array Range Error
92	Cam Array Range Error
93	Illegal SPI Write
94	SPI Write array is to small
95	Status bit size error
96	Ethernet RX buffer overflow
97	Fram Read Error
98	Watchdog caused reset
99	Wrong Port Type specified,Only RS232 and RS485 allowed
100	Key String Error
101	Illegal Option write
102	OP_DI_24_31 and OP_RS485 both selected
103	Crystal Clock/Phase_lock loss caused last reset
104	ARCSINE input range error
105	ARCCOS input range error
106	Negative sqrt argument
107	Unknown MoveQ command

Error Code	Definition
108	Max node address exceeded
109	Bitmap Store is Empty
110	Pol Num Error
111	not in use
112	Maximum Error number limit exceeded
113	Maximum Screen Colour value limit exceeded
114	Screen Co-ordinate range exceeded
115	Key Number range exceeded
116	Key Return Value range exceeded
117	Key Size error
118	Font number range error
119	X,Y, position is outside the viewport
120	not in use
121	Bitmap number error
122	Invalid Bitmap number
123	Maximum bcv_interval exceeded
124	CAN BCV buffer overflow
125	ETHERNET BCV buffer overflow
126	NULL terminator not found
127	stovar format failure
128	I/O number error. Range = 0 to 9
129	EdgeType error. Range = 0 to 2

Appendix 3 - Glossary Of Terms

ADC	Analogue to digital converter
DAC	Digital to Analogue converter
ESD	Eight Segment Display
FLASH	Fast Electrically Erasable Read/Write Memory
FRAM	Ferro Electric Ram
IDE	Integrated Development Environment
PID	Proportional Integral Differential
SD	Syntax Diagram