beginner

COLLABORATORS			
	TITLE : beginner		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 2, 2023	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

beginner

1.1 E Built-In Constants Variables and Functions

```
                E Built-In Constants, Variables and Functions
**********************************************
```

This chapter describes the constants, variables and functions which are built in to the E language. You can add more by using modules, but that's a more advanced topic (see Modules).

1.2 Built-In Constants

```
                Built-In Constants
==================
```

We've already met several built-in constants. Here's the complete list:

TRUE, FALSE
: The boolean constants. As numbers, TRUE is -1 and FALSE is zero.

NIL
: The bad pointer value. Several functions produce this value for a pointer if an error occurred. As a number, NIL is zero.

ALL
: Used with string and list functions to indicate that all the string or list is to be used. As a number, ALL is -1.

GADGETSIZE

```
     The minimum number of bytes required to hold all the data for one
     gadget.  See
               Intuition support functions
               .

OLDFILE,  NEWFILE
     Used with Open to open an old or new file.  See the 'AmigaDOS Manual'
     for more details.

STRLEN
     The length of the last string constant used.  Remember that a string
     constant is something between ' characters, so, for example, the
     following program prints the string s and then its length:

          PROC main()
            DEF s:PTR TO CHAR, len
            s:='12345678'
            len:=STRLEN
            WriteF(s)
            WriteF('\nis \d characters long\n', len)
          ENDPROC
```

1.3 Built-In Variables

```
                Built-In Variables
==================

   The following variables are built in to E and are called system
variables.  They are global so can be accessed from any procedure.

arg
     This is a string which contains the command line arguments passed
     your program when it was run (from the Shell or CLI).  For instance,
     if your program were called fred and you ran it like this:

          fred file.txt "a big file" another

     then arg would the string:

          file.txt "a big file" another

     If you have AmigaDOS 2.0 (or greater) you can use the system routine
     ReadArgs to parse the command line in a much more versatile way.
     There is a worked example on argument parsing in Part Three (see
     Argument Parsing).

wbmessage
     This contains NIL if your program was started from the Shell/CLI,
     otherwise it's a pointer to the Workbench message which contains
     information about the icons selected when you started the program
     from Workbench.  So, if you started the program from Workbench
     wbmessage will not be NIL and it will contain the Workbench
     arguments, but if you started the program from the Shell/CLI
```

wbmessage will be NIL and the argments will be in arg (or via ReadArgs). There is a worked example on argument parsing in Part Three (see Argument Parsing).

stdin, stdout, conout
The stdin and stdout variables contain the standard input and output filehandles. If your program was started from the Shell/CLI they will be filehandles on the Shell/CLI window (and conout will be NIL). However, if your program was started from Workbench these will both be NIL, and in this case the first call to WriteF will open an output CON: window and store the file handle for the window in stdout and conout. The file handle stored in conout will be closed using Close when the program terminates, so you can set up your own CON: window or file for use by the output functions and have it automatically closed. See
Input and output functions
.

stdrast
The raster port used by E built-in graphics functions such as Box and Plot. This can be changed so that these functions draw on different screens etc. See
Graphics functions
.

dosbase, execbase, gfxbase, intuitionbase
These are pointers to the appropriate library base, and are initialised by the E startup code, i.e., the Dos, Exec, Graphics and Intuition libraries are all opened by E so you don't need to do it yourself. These libraries are also automatically closed by E, so you shouldn't close them yourself. However, you must explicitly open and close all other Amiga system libraries that you want to use. The other library base variables are defined in the accompanying module (see Modules).

1.4 Built-In Functions

Built-In Functions
==========

There are many built-in functions in E. We've already seen a lot of string and list functions, and we've used WriteF for printing. The remaining functions are, generally, simplifications of complex Amiga system functions, or E versions of support functions found in languages like C and Pascal.

To understand the graphics and Intuition support functions completely you really need to get something like the 'Rom Kernel Reference Manual (Libraries)'. However, if you don't want to do anything too complicated you should be able to get by.

```
                Input and output functions

                Intuition support functions

                Graphics functions

                Maths and logic functions

                System support functions
```

1.5 Input and output functions

```
                Input and output functions
-------------------------

WriteF(string,param1,param2,...)
     Writes a string to the standard output and returns the number of
     characters written.  If place-holders are used in the string then the
     appropriate number of parameters must be supplied after the string in
     the order they are to be printed as part of the string.  So far we've
     only met the \d place-holder for decimal numbers.  The complete list
     is:

          Place-Holder  Parameter Type   Prints
          ------------------------------------------------
               \c          Number        Character
               \d          Number        Decimal number
               \h          Number        Hexadecimal number
               \s          String        String

     So to print a string you use the \s place-holder in the string and
     supply the string (i.e., a PTR TO CHAR) as a parameter.  Try the
     following program (remember \a prints an apostrophe character):

          PROC main()
            DEF s[30]:STRING
            StrCopy(s, 'Hello world', ALL)
            WriteF('The third element of s is "\c"\n', s[2])
            WriteF('or \d (decimal)\n',                s[2])
            WriteF('or \h (hexadecimal)\n',            s[2])
            WriteF('and s itself is \a\s\a\n',         s)
          ENDPROC

     This is the output it generates:

          The third element of s is "l"
          or 108 (decimal)
          or 6C (hexadecimal)
          and s itself is 'Hello world'

     You can control how the parameter is formatted in the \d, \h and
     \s fields using another collection of special character
     sequences before the place-holder and size specifiers after it.  If
     no size is specified the field will be as big as the data requires.
```

A fixed field size can be specified using [number] after the place-holder. For strings you can also use the size specifier (min,max) which specifies the minimum and maximum sizes of the field. By default the data is right justified in the field and the left part of the field is filled, if necessary, with spaces. The following sequences before the place-holder can change this:

Sequence	Meaning
\l	Left justify in field
\r	Right justify in field
\z	Set fill character to "0"

See how these formatting controls affect this example:

```
PROC main()
  DEF s[30]:STRING
  StrCopy(s, 'Hello world', ALL)
  WriteF('The third element of s is "\c"\n', s[2])
  WriteF('or \d[4] (decimal)\n',             s[2])
  WriteF('or \z\h[4] (hexadecimal)\n',       s[2])
  WriteF('\a\s[5]\a are the first five elements of s \n', s)
  WriteF('and s in a very big field  \a\s[20]\a\n',   s)
  WriteF('and s left justified in it \a\l\s[20]\a\n', s)
ENDPROC
```

Here's the output it should generate:

```
The third element of s is "l"
or  108 (decimal)
or 006C (hexadecimal)
'Hello' are the first five elements of s
and s in a very big field  '         Hello world'
and s left justified in it 'Hello world         '
```

WriteF uses the standard output, and this file handle is stored in the stdout variable. If your program is started from Workbench this variable will contain NIL. In this case, the first call to WriteF will open a special output window and put the file handle in the variables stdout and conout, as outlined above (see

Built-In Variables
).

PrintF(string,param1,param2,...)

PrintF works just like WriteF except it uses the more efficient, buffered output routines only available if your Amiga is using Kickstart version 37 or greater (i.e., AmigaDOS 2.04 and above).

StringF(e-string,string,arg1,arg2,...)

The same as WriteF except that the result is written to e-string instead of being printed. For example, the following code fragment sets s to 00123 is a (since the E-string is not long enough for the whole string):

```
DEF s[10]:STRING
```

```
StringF(s, '\z\d[5] is a number', 123)
```

Out(filehandle,char)
Outputs a single character, char, to the file or console window denoted by filehandle, and returns -1 to indicate success (so any other return value means an error occurred). For instance, filehandle could be stdout, in which case the character is written to the standard output. (You need to make sure stdout is not NIL, and you can do this by using a WriteF('') call.) In general, you obtain a filehandle using the Amiga system function Open from the dos.library (see String Handling and I-O).

Inp(filehandle)
Reads and returns a single character from filehandle. If -1 is returned then the end of the file (EOF) was reached, or there was an error.

ReadStr(filehandle,e-string)
Reads a whole string from filehandle and returns -1 if EOF was reached or an error occurred. Characters are read up to a linefeed or the size of the string, whichever is sooner. Therefore, the resulting string may be only a partial line. If -1 is returned then EOF was reached or an error occurred, and in either case the string so far is still valid. So, you still need to check the string even if -1 is returned. (This will most commonly happen with files that do not end with a linefeed.) The string will be empty (i.e., of zero length) if nothing more had been read from the file when the error or EOF happened.

This next little program reads continually from its input until an error occurs or the user types quit. It echoes the lines that it reads in uppercase. If you type a line longer than ten characters you'll see it reads it in more than one go. Because of the way normal console windows work, you need to type a return before a line gets read by the program (but this allows you to edit the line before the program sees it). If the program is started from Workbench then stdin would be NIL, so WriteF('') is used to force stdout to be valid, and in this case it will be a new console window which can be used to accept input! (To make the compiled program into a Workbench program you simply need to create a tool icon for it. A quick way of doing this is to copy an existing tool's icon.)

```
PROC main()
  DEF s[10]:STRING, fh
  WriteF('')
  fh:=IF stdin THEN stdin ELSE stdout
  WHILE ReadStr(fh, s)<>-1
    UpperStr(s)
  EXIT StrCmp(s, 'QUIT', ALL)
    WriteF('Read: \a\s\a\n', s)
  ENDWHILE
  WriteF('Finished\n')
ENDPROC
```

There are some worked examples in Part Three (see String Handling and I-O) which also show how to use ReadStr.

```
FileLength(string)
     Returns the length of the file named in string, or -1 if the file
     doesn't exist or an error occurred.  Notice that you don't need to
     Open the file or have a filehandle, you just supply the filename.
     There is a worked example in Part Three (see String Handling and I-O)
     which shows how to use this function.

SetStdIn(filehandle)
     Returns the value of stdin before setting it to filehandle.
     Therefore, the following code fragments are equivalent:

            oldstdin:=SetStdIn(newstdin)

            oldstdin:=stdin
            stdin:=newstdin

SetStdOut(filehandle)
     Returns the value of stdout before setting it to filehandle, and is
     otherwise just like SetStdIn.
```

1.6 Intuition support functions

```
Intuition support functions
---------------------------

   The functions in this section are simplified versions of Amiga system
functions (in the Intuition library, as the title suggests).  To make best
use of them you are probably going to need something like the 'Rom Kernel
Reference Manual (Libraries)', especially if you want to understand the
Amiga specific things like IDCMP and raster ports.

   The descriptions given here vary slightly in style from the previous
descriptions.  All function parameters can be expressions which represent
numbers or addresses, as appropriate.  Because many of the functions take
several parameters they have been named (fairly descriptively) so they can
be more easily referenced.

OpenW(x,y,wid,hgt,idcmp,wflgs,title,scrn,sflgs,gads,tags=NIL)
     Opens and returns a pointer to a window with the supplied properties.
     If for some reason the window could not be opened NIL is returned.

    x,  y
          The position on the screen where the window will appear.

    wid,  hgt
          The width and height of the window.

    idcmp,  wflgs
          The IDCMP and window specific flags.

    title
          The window title (a string) which appears on the title bar of
          the window.
```

scrn, sflgs
: The screen on which the window should open. If sflgs is 1 the window will be opened on Workbench, and scrn is ignored (so it can be NIL). If sflgs is $F (i.e., 15) the window will open on the custom screen pointed to by scrn (which must then be valid). See OpenS to see how to open a custom screen and get a screen pointer.

gads
: A pointer to a gadget list, or NIL if you don't want any gadgets. These are not the standard window gadgets, since they are specified using the window flags. A gadget list can be created using the Gadget function.

tags
: A tag-list of other options available under Kickstart version 37 or greater. This can normally be omitted since it defaults to NIL. See the 'Rom Kernel Reference Manual (Libraries)' for details about the available tags and their meanings.

There's not enough space to describe all the fine details about windows and IDCMP (see the 'Rom Kernel Reference Manual (Libraries)' for complete details), but a brief description in terms of flags might be useful. Here's a small table of common IDCMP flags:

IDCMP Flag	Value
IDCMP_NEWSIZE	$2
IDCMP_REFRESHWINDOW	$4
IDCMP_MOUSEBUTTONS	$8
IDCMP_MOUSEMOVE	$10
IDCMP_GADGETDOWN	$20
IDCMP_GADGETUP	$40
IDCMP_MENUPICK	$100
IDCMP_CLOSEWINDOW	$200
IDCMP_RAWKEY	$400
IDCMP_DISKINSERTED	$8000
IDCMP_DISKREMOVED	$10000

Here's a table of useful window flags:

Window Flag	Value
WFLG_SIZEGADGET	$1
WFLG_DRAGBAR	$2
WFLG_DEPTHGADGET	$4
WFLG_CLOSEGADGET	$8
WFLG_SIZEBRIGHT	$10
WFLG_SIZEBBOTTOM	$20
WFLG_SMART_REFRESH	0
WFLG_SIMPLE_REFRESH	$40
WFLG_SUPER_BITMAP	$80
WFLG_BACKDROP	$100
WFLG_REPORTMOUSE	$200
WFLG_GIMMEZEROZERO	$400
WFLG_BORDERLESS	$800

```
     WFLG_ACTIVATE        $1000
```

All these flags are defined in the module intuition/intuition, so if you use that module you can use the constants rather than having to write the less descriptive value (see Modules). Of course, you can always define your own constants for the values that you use.

You use the flags by OR-ing the ones you want together, in a similar way to using sets (see Sets). However, you should supply only IDCMP flags as part of the idcmp parameter, and you should supply only window flags as part of the wflgs parameter. So, to get IDCMP messages when a disk is inserted and when the close gadget is clicked you specify both of the flags IDCMP_DISKINSERTED and IDCMP_CLOSEWINDOW for the idcmp parameter, either by OR-ing the constants or (less readably) by using the calculated value $8200.

Some of the window flags require some of IDCMP flags to be used as well, if an effect is to be complete. For example, if you want your window to have a close gadget (a standard window gadget) you need to use WFLG_CLOSEGADGET as one of the window flags. If you want that gadget to be useful then you need to get an IDCMP message when the gadget is clicked. You therefore need to use IDCMP_CLOSEWINDOW as one of the IDCMP flags. So the full effect requires both a window and an IDCMP flag (a gadget is pretty useless if you can't tell when it's been clicked). The worked example in Part Three illustrates how to use these flags in this way (see Gadgets).

If you only want to output text to a window (and maybe do some input from a window), it may be better to use a console window. These provide a text based input and output window, and are opened using the Dos library function Open with the appropriate CON: file name. See the 'AmigaDOS Manual' for more details about console windows.

CloseW(winptr)

Closes the window which is pointed to by winptr. It's safe to give NIL for winptr, but in this case, of course, no window will be closed! The window pointer is usually a pointer returned by a matching call to OpenW. You must remember to close any windows you may have opened before terminating your program.

OpenS(wid,hgt,depth,scrnres,title,tags=NIL)

Opens and returns a pointer to a custom screen with the supplied properties. If for some reason the screen could not be opened NIL is returned.

wid, hgt

The width and height of the screen.

depth

The depth of the screen, i.e., the number of bit-planes. This can be a number in the range 1-8 for AGA machines, or 1-6 for pre-AGA machines. A screen with depth 3 will be able to show 2 to the power 3 (i.e., 8) different colours, since it will have 2 to the power 3 different pens (or colour registers) available. You can set the colours of pens using the SetColour function.

scrnres

The screen resolution flags.

title
The screen title (a string) which appears on the title bar of the screen.

tags
A tag-list of other options available under Kickstart version 37 or greater. See the 'Rom Kernel Reference Manual (Libraries)' for more details.

The screen resolution flags control the screen mode. The following (common) values are taken from the module graphics/view (see Modules). You can, if you want, define your own constants for the values that you use. Either way it's best to use descriptive constants rather than directly using the values.

Mode Flag	Value
V_LACE	$4
V_SUPERHIRES	$20
V_PFBA	$40
V_EXTRA_HALFBRITE	$80
V_DUALPF	$400
V_HAM	$800
V_HIRES	$8000

So, to get a hires, interlaced screen you specify both of the flags V_HIRES and V_LACE, either by OR-ing the constants or (less readably) by using calculated value $8004. There is a worked example using this function in Part Three (see Screens).

CloseS(scrnptr)
Closes the screen which is pointed to by scrnptr. It's safe to give NIL for scrnptr, but in this case, of course, no screen will be closed! The screen pointer is usually a pointer returned by a matching call to OpenS. You must remember to close any screens you may have opened before terminating your program. Also, you must close all windows that you opened on your screen before you can close the screen.

Gadget(buf,glist,id,flags,x,y,width,text)
Creates a new gadget with the supplied properties and returns a pointer to the next position in the (memory) buffer that can be used for a gadget.

buf
This is the memory buffer, i.e., a chunk of allocated memory. The best way of allocating this memory is to declare an array of size n*GADGETSIZE, where n is the number of gadgets which are going to be created. The first call to Gadget will use the array as the buffer, and subsequent calls use the result of the previous call as the buffer (since this function returns the next free position in the buffer).

glist
This is a pointer to the gadget list that is being created,

```
        i.e., the array used as the buffer.  When you create the first
        gadget in the list using an array a, this parameter should be
        NIL.  For all other gadgets in the list this parameter
        should be the array a.

    id
        A number which identifies the gadget.  It is best to give a
        unique number for each gadget; that way you can easily identify
        them.  This number is the only way you can identify which gadget
        has been clicked.

    flags
        The type of gadget to be created.  Zero represents a normal
        gadget, one a boolean gadget (a toggle) and three a boolean that
        starts selected.

    x,  y
        The position of the gadget, relative to the top, left-hand
        corner of the window.

    width
        The width of the gadget (in pixels, not characters).

    text
        The text (a string) which will centred in the gadget, so the
        width must be big enough to hold this text.

     Once a gadget list has been created by possibly several calls to this
     function the list can be passed as the gads parameter to OpenW.
     There is a worked example using this function in Part Three (see
     Gadgets).

Mouse()
     Returns the state of the mouse buttons (including the middle mouse
     button if you have a three-button mouse).  This is a set of flags,
     and the individual flag values are:

          Button Pressed   Value
          ----------------------
          Left              %001
          Right             %010
          Middle            %100

     So, if this function returns %001 you know the left button is being
     pressed, and if it returns %110 you know the middle and right buttons
     are both being pressed.

     This mouse function is not strictly the proper way to do things.  It
     is suggested you use this function only for small tests or demo-like
     programs.  LeftMouse and WaitLeftMouse can be used to do things in a
     friendly way, but are restricted to seeing when the left mouse button
     is pressed.  More generally, the proper way of getting mouse details
     is to use the appropriate IDCMP flags for your window, wait for
     events (using WaitIMessage, for example) and decode the received
     information.

MouseX(winptr)
```

Returns the x coordinate of the mouse pointer, relative to the window pointed to by winptr.

As above, this mouse function is not strictly the proper way to do things.

MouseY(winptr)
Returns the y coordinate of the mouse pointer, relative to the window pointed to by winptr.

As above, this mouse function is not strictly the proper way to do things.

LeftMouse(winptr)
Returns TRUE if left mouse button has been clicked in the window pointed to by winptr, and FALSE otherwise. In order for this to work sensibly the window must have the IDCMP flag IDCMP_MOUSEBUTTONS set (see above).

This function does things in a proper, Intuition-friendly manner and so is a good alternative to the Mouse function.

WaitIMessage(winptr)
This function waits for a message from Intuition for the window pointed to by winptr and returns the class of the message (which is an IDCMP flag). If you did not specify any IDCMP flags when the window was opened, or the specified messages could never happen (e.g., you asked only for gadget messages and you have no gadgets), then this function may wait forever. When you've got a message you can use the MsgXXX functions to get some more information about the message. See the 'Rom Kernel Reference Manual (Libraries)' for more details on Intuition and IDCMP. There is a worked example using this function in Part Three (see IDCMP Messages).

This function is basically equivalent to the following function, except that the MsgXXX functions can also access the message data held in the variables code, qual and iaddr.

```
PROC waitimessage(win:PTR TO window)
  DEF port,msg:PTR TO intuimessage,class,code,qual,iaddr
  port:=win.userport
  IF (msg:=GetMsg(port))=NIL
    REPEAT
      WaitPort(port)
    UNTIL (msg:=GetMsg(port))<>NIL
  ENDIF
  class:=msg.class
  code:=msg.code
  qual:=msg.qualifier
  iaddr:=msg.iaddress
  ReplyMsg(msg)
ENDPROC class
```

MsgCode()
Returns the code part of the message returned by WaitIMessage.

MsgIaddr()

```
     Returns the iaddr part of the message returned by WaitIMessage.
     There is a worked example using this function in Part Three (see
     IDCMP Messages).

MsgQualifier()
     Returns the qual part of the message returned by WaitIMessage.

WaitLeftMouse(winptr)
     This function waits for the left mouse button to be clicked in the
     window pointed to by winptr.  It is advisable to have the IDCMP
     flag IDCMP_MOUSEBUTTONS set for the window (see above).

     This function does things in a proper, Intuition-friendly manner and
     so is a good alternative to the Mouse function.
```

1.7 Graphics functions

```
                Graphics functions
------------------

   The functions in this section use the standard raster port, the address
of which is held in the variable stdrast.  Most of the time you don't need
to worry about this because the E functions which open windows and screens
set up this variable (see
                Intuition support functions
                ).  So, by default,
these functions affect the last window or screen opened.  When you close a
window or screen, stdrast becomes NIL and calls to these functions have no
effect.  There is a worked example using these functions in Part Three
(see Graphics).

   The descriptions in this section follow the same style as the previous
section.

Plot(x,y,pen=1)
     Plots a single point (x,y) in the specified pen colour.  The
     position is relative to the top, left-hand corner of the window or
     screen that is the current raster port (normally the last screen or
     window to be opened).  The range of pen values available depend on
     the screen setup, but are at best 0-255 on AGA machines and 0-31 on
     pre-AGA machines.  As a guide, the background colour is usually pen
     zero, and the main foreground colour is pen one (and this is the
     default pen).  You can set the colours of pens using the SetColour
     function.

Line(x1,y1,x2,y2,pen=1)
     Draws the line (x1,y1) to (x2,y2) in the specified pen colour.

Box(x1,y1,x2,y2,pen=1)
     Draws the (filled) box with vertices (x1,y1), (x2,y1),
     (x1,y2) and (x2,y2) in the
     specified pen colour.

Colour(fore-pen,back-pen=0)
```

Sets the foreground and background pen colours. As mentioned above, the background colour is normally pen zero and the main foreground is pen one. You can change these defaults with this function, and if you stick to having the background pen as pen zero then calling this function with one argument changes just the foreground pen.

TextF(x,y,format-string,arg1,arg2,...)
This works just like WriteF except the resulting string is drawn on the current raster port (usually the last window or screen to be opened) ,starting at point (x,y). Take care not to use any line-feed, carriage return, tab or escape characters in the string--they don't behave like they do in WriteF.

SetColour(scrnptr,pen,r,g,b)
Sets the colour of colour register pen for the screen pointed to by scrnptr to be the appropriate RGB value (i.e., red value r, green value g and blue value b). The pen can be anything up to 255, depending on the screen depth. Regardless of the chipset being used, r, g and b are taken from the range zero to 255, so 24-bit colours are always specified. In operation, though, the values are scaled to 12-bit colour for non-AGA machines.

SetStdRast(newrast)
Returns the value of stdrast before setting it to the new value. The following code fragments are equivalent:

```
oldstdrast:=SetStdRast(newstdrast)

oldstdrast:=stdrast
stdrast:=newstdrast
```

SetTopaz(size=8)
Sets the text font for the current raster port to Topaz at the specified size, which defaults to the standard size eight.

1.8 Maths and logic functions

Maths and logic functions

We've already seen the standard arithmetic operators. The addition, +, and subtraction, -, operators use full 32-bit integers, but, for efficiency, multiplication, *, and division, /, use restricted values. You can use * only to multiply 16-bit integers, and the result will be a 32-bit integer. Similarly, you can use / only to divide a 32-bit integer by a 16-bit integer, and the result will be a 16-bit integer. The restrictions do not affect most calculations, but if you really need to use all 32-bit integers (and you can cope with overflows, etc.) you can use the Mul and Div functions. Mul(a,b) corresponds to a*b, and Div(a,b) corresponds to a/b.

We've also met the logic operators AND and OR, which we know are really bit-wise operators. You can also use the functions And and Or to do

exactly the same as AND and OR (respectively). So, for instance, And(a,b) is the same as a AND b. The reason for these functions is because there are Not and Eor (bit-wise) functions, too (and there aren't operators for these). Not(a) swaps one and zero bits, so, for instance, Not(TRUE) is FALSE and Not(FALSE) is TRUE. Eor(a,b) is the exclusive version of Or and does almost the same, except that Eor(1,1) is 0 whereas Or(1,1) is 1 (and this extends to all the bits). So, basically, Eor tells you which bits are different, or, logically, if the truth values are different. Therefore, Eor(TRUE,TRUE) is FALSE and Eor(TRUE,FALSE) is TRUE.

There's a collection of other functions related to maths, logic or numbers in general:

Abs(expression)
Returns the absolute value of expression. The absolute value of a number is that number without any minus sign (i.e., its the size of a number, disregarding its sign). So, Abs(9) is 9, and Abs(-9) is also 9.

Sign(expression)
Returns the sign of expression, which is the value one if it is (strictly) positive, -1 if it is (strictly) negative and zero if it is zero.

Even(expression)
Returns TRUE if expression represents an even number, and FALSE otherwise. Obviously, a number is either odd or even!

Odd(expression)
Returns TRUE if expression represents an odd number, and FALSE otherwise.

Max(exp1, exp2)
Returns the maximum of exp1 and exp2.

Min(exp1, exp2)
Returns the minimum of exp1 and exp2.

Bounds(exp, minexp, maxexp)
Returns the value of exp bounded to the limits minexp (minimum bound) and maxexp (maximum bound). That is, if exp lies between the bounds then exp is returned, but if it is less than minexp then minexp is returned or if it is greater than maxexp then maxexp is returned. This is useful for, say, constraining a calculated value to be a valid (integer) percentage (i.e., a value between zero and one hundred).

The following code fragments are equivalent:

```
y:=Bounds(x, min, max)

y:=IF x<min THEN min ELSE IF x>max THEN max ELSE x
```

Mod(exp1,exp2)
Returns the 16-bit remainder (or modulus) of the division of the 32-bit exp1 by the 16-bit exp2 as the regular return value (see Multiple Return Values), and the 16-bit result of the division as the

first optional return value. For example, the first assignment in the following code sets a to 5 (since 26=(7*3)+5), b to 3, c to -5 and d to -3. It is important to notice that if exp1 is negative then the modulus will also be negative. This is because of the way integer division works: it simply discards fractional parts rather rounding.

```
a,b:=Mod(26,7)
c,d:=Mod(-26,7)
```

Rnd(expression)
Returns a random number in the range 0 to (n-1), where expression represents the value n. These numbers are pseudo-random, so although you appear to get a random value from each call, the sequence of numbers you get will probably be the same each time you run your program. Before you use Rnd for the first time in your program you should call it with a negative number. This decides the starting point for the pseudo-random numbers.

RndQ(expression)
Returns a random 32-bit value, based on the seed expression. This function is quicker than Rnd, but returns values in the 32-bit range, not a specified range. The seed value is used to select different sequences of pseudo-random numbers, and the first call to RndQ should use a large value for the seed.

Shl(exp1,exp2)
Returns the value represented by exp1 shifted exp2 bits to the left. For example, Shl(%0001110,2) is %0111000 and Shl(%0001011,3) is %1011000. Shifting a number one bit to the left is generally the same as multiplying it by two (although this isn't true when you shift large positive or large negative values). (The new bits shifted in at the right are always zeroes.)

Shr(exp1,exp2)
Returns the value represented by exp1 shifted exp2 bits to the right. For example, Shr(%0001110,2) is %0000011 and Shr(%1011010,3) is %0001011. For positive exp1, shifting one bit to the right is the same as dividing by two. (The new bits shifted in at the left are zeroes if exp1 is positive, and ones otherwise, hence preserving the sign of the expression.)

Long(addr), Int(addr), Char(addr)
Returns the LONG, INT or CHAR value at the address addr. These functions should be used only when setting up a pointer and dereferencing it in the normal way would make your program cluttered and less readable. Use of functions like these is often called peeking memory (especially in dialects of the BASIC language).

PutLong(addr,exp), PutInt(addr,exp), PutChar(addr,exp)
Writes the LONG, INT or CHAR value represented by exp to the address addr. Again, these functions should be used only when really necessary. Use of functions like these is often called poking memory.

1.9 System support functions

```
System support functions
------------------------

New(bytes)
     Returns a pointer to a newly allocated chunk of memory, which is
     bytes number of bytes.  If the memory could not be allocated NIL is
     returned.  The memory is initialised to zero in each byte, and taken
     from any available store (Fast or Chip memory, in that order of
     preference).  When you've finished with this memory you can use
     Dispose to free it for use elsewhere in your program.  You don't have
     to Dispose with memory you allocated with New because your program
     will automatically free it when it terminates.  This is not true for
     memory allocated using the normal Amiga system routines.

NewR(bytes)
     The same as New except that if the memory could not be allocated then
     the exception "MEM" is raised (and so, in this case, the function
     does not return).  See Exception Handling.

NewM(bytes,type)
     The same as NewR except that the type of memory (Fast or Chip) to
     be allocated can be specified using flags.  The flags are defined in
     the module exec/memory (see Amiga System Modules).  See the 'Rom
     Kernel Reference Manual (Libraries)' for details about the system
     function AllocMem which uses these flags in the same way.

     As useful example, here's a small program which allocates some
     cleared (i.e., zeroed) Chip memory.

          MODULE 'exec/memory'

          PROC main()
            DEF m
            m:=NewM(20, MEMF_CHIP OR MEMF_CLEAR)
            WriteF('Allocation succeeded, m = $\h\n', m)
          EXCEPT
            IF exception="NEW" THEN WriteF('Failed\n')
          ENDPROC

Dispose(address)
     Used to free memory allocated with New, NewR or NewM.  You should
     rarely need to use this function because the memory is automatically
     freed when the program terminates.

DisposeLink(complex)
     Used to free the memory allocated with String (see String functions)
     or List (see List functions).  Again, you should rarely need to use
     this function because the memory is automatically freed when the
     program terminates.

FastNew(bytes)
     The same as NewR except it uses a very fast, recycling method of
     allocating memory.  The memory allocated using FastNew is, as ever,
     deallocated automatically at the end of a program, and can be
```

deallocated before then using FastDispose. Note that only FastDispose can be used and that it differs slightly from the Dispose and DisposeLink functions (you have to specify the number of bytes again when deallocating).

FastDispose(address,bytes)
Used to free the memory allocated using FastNew. The bytes parameter must be the same as the bytes used when the memory was allocated with FastNew, but the benefit is much faster allocation and deallocation, and generally more efficient use of memory.

CleanUp(expression=0)
Terminates the program at this point, and does the normal things an E program does when it finishes. The value denoted by expression is returned as the error code for the program. It is the replacement for the AmigaDOS Exit routine which should never be used in an E program. This is the only safe way of terminating a program, other than reaching the (logical) end of the main procedure (which is by far the most common way!).

CtrlC()
Returns TRUE if control-C has been pressed since the last call, and FALSE otherwise. This is really sensible only for programs started from the Shell/CLI.

FreeStack()
Returns the current amount of free stack space for the program. Only complicated programs need worry about things like stack. Recursion is the main thing that eats a lot of stack space (see Recursion).

KickVersion(expression)
Returns TRUE if your Kickstart revision is at least that given by expression, and FALSE otherwise. For instance, KickVersion(37) checks whether you're running with Kickstart version 37 or greater (i.e., AmigaDOS 2.04 and above).