Tutorials in Quantitative Methods for Psychology
2009, Vol. 5 (2), p. 59-67.

Using Mathematica within E-Prime

Denis Cousineau

Université de Montréal

When programming complex experiments (for example, involving the generation of stimuli online), the traditional experiment programming software are not well equipped. One solution is to give up entirely the use of such software in favor of a low-level programming language. Here we show how E-Prime can be connected to Mathematica so that the easiness and reliability of this software can be preserved while at the same time granting it the full computational power of a high-level programming language. As an example, we show how to generate noisy images with noise proportional to the rate of success of the participants with as few as 12 lines of codes in E-Prime.

Psychology experiments can be rather simple, being composed of a fixed number of trials, presenting a fixed set of stimuli and collecting a fixed set of responses. For such situations, many software exists that can program the experiment rapidly and easily (such as Superlab, ERTS, InQuisit, E-Prime, DirectRT, to name a few, Stahl, 2006). However, more sophisticated experiments are sometimes required which can for example (i) continue training until a performance criterion is reached, (ii) generate random stimuli, (iii) alter stimulus differently to adapt to the participant, (iv) interpret the participant's response and continue the experiment accordingly, etc. The possibilities are endless and we are only enumerating a few. All these possibilities can be implemented as long as a programming language is available. However, (a) very few experiment programming software offer the possibility to include lines of code within the experiment, (b) when they do, it is often a low-level programming language.

E-Prime is such a experiment programming software. Within E-Prime, it is possible to add customized code using the Visual-Basic programming language. Whereas verifying that a performance criterion is reached is fairly easy to program in Visual-Basic (point i above), generating random stimuli can be more difficult if a random number generator different from a uniform distribution is needed. As of altering stimuli, it can be near impossible to do with Visual-Basic as there is no linear algebra, no Fourier or wavelet transforms, no convolution sub-routines in this language (this is why it is called a "low-level" programming language). Such sub-routines (either procedures or functions) can be defined in E-Prime, but doing so takes times (the code for any such sub-routine can be many pages long, with debugging and testing a tedious process).

When one of those situations arise, there are two possible courses of action: (A) Give up the use of an experiment programming software and program everything with a low-level programming language such as C. Although advanced sub-routines are not part of the C language, they can be found in various references (e. g. Press, Flannery, Teukolsky and Vetterling, 1986). (B) Use a high-level programming environment as long as it is capable of presenting stimuli and reading responses with a very high timing accuracy. To our knowledge, only Matlab is capable of this if a special library is downloaded, the

Request for reprint should be addressed to Denis Cousineau, Département de psychologie, Université de Montréal, C. P. 6128, succ. Centre-ville, Montréal (Québec) H3C 3J7, CANADA, or using e-mail at Denis.Cousineau@Umontreal.CA. This research was supported in part by the Conseil pour la Recherche en Sciences Naturelles et en Génie du Canada.

Psychophysics toolbox (Brainard, 1997).

We believe that both solutions are too radical as they totally evacuate the experiment programming software. These software are robust, very easy to program and easy to connect to other devices such as response boxes, fMRI and EEG acquisition systems, etc. In what follow, we propose a third alternative: a mixed environment in which all the stimulus presentation and response collections are assumed by an experiment programming software and in which all the advanced computational capabilities are assumed by a high-level programming software.

The duo discussed in this article is E-Prime and Mathematica (E-Prime, 2004, Chen, Cui, & Zhang, 2005, Wolfram, 1996). Here, E-Prime is used as the primary program: It will start Mathematica, send requests and read responses, and finally, shut down Mathematica when the experiment is finished. In the first section, we show how E-Prime can start and end a Mathematica session. In the subsequent section, we will briefly explain the protocol used to exchange information between the two programs. However, all these technical details are not essential: Section 3 provides the necessary sub-routines to add to E-Prime. You can simple type them into E-Prime and use them to control Mathematica without advanced understanding of how they operate. Section 4 gives a complete example where additive noise is added to images in proportion to the accuracy of the identification responses.

1- Starting and ending a session with the Mathematical kernel

Mathematica is in fact composed of two programs: The Mathematica front-end (the file mathematica.exe) which is the user interface, and the Mathematica kernel (the file mathkernel.exe) which is responsible for actually performing the computations. When the user type "Enter" in the front-end, the expression in the current cell is packed and send to the kernel. The kernel processes it and returns packages ("packet" in the Mathematica idiom) containing the response(s) of the kernel. The sending and receiving of packets are managed by the MathLink protocol. Figure 1 summarizes this.

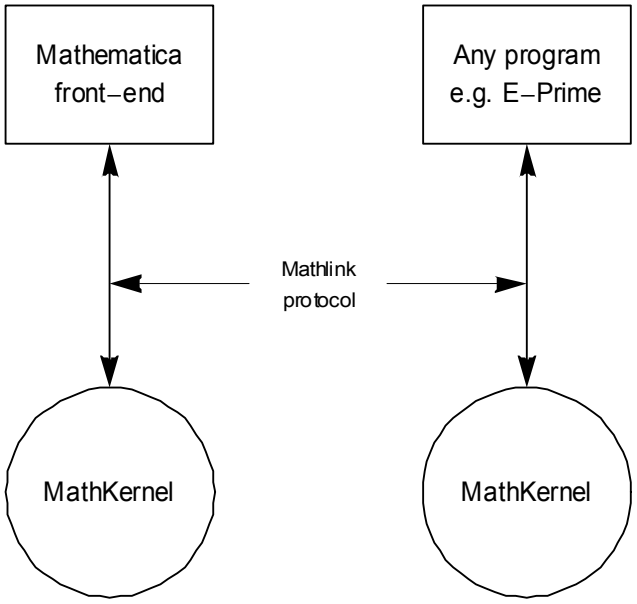

Figure 1. Relation between a front-end software and the Mathematica kernel

The MathLink protocol is implemented in a library composed of functions and subprograms. The latest version (Implementation 3) is available for both 32-byte and 64 byte processors (in doubt, the 32-byte version works on all machines). It is installed by Mathematica in the C:\Windows\System32 folder under the name ML32I3.dll (or ML64I3 for the 64-byte version). Here, the extension ".dll" stands for "dynamically-linked library". Such files contain functions and subroutines that can be used by any programs written in any language. They are already compiled and ready to use.

To start a kernel and be able to communicate with it, you need to use two functions provided in the MathLink library, MLInitialize and MLOpenString. The second will return a link which will identify the kernel with which E-Prime is interacting (many kernels can be opened simultaneously). The link is necessary for all interaction and to close the kernel. To start the kernel, a connecting string is required which specifies that a new kernel must be launched (the alternative would be to spy on an existing kernel) and what is the path and file name of the kernel on your computer. The instructions in E-Basic (Visual Basic for E-Prime) are:

```
Dim MLEnv as long, MLLink as long, MLErrNo as long
MLEnv = MLInitialize(0)
MLLink = MLOpenString(MLEnv,
"-linkmode launch -linkname \"C:\\\\Program
  files\\\\Wolfram Research\\\\Mathematica
  \\\\6.0\\\\MathKernel.exe -mathlink\" ",
MLErrNo)
```

The second command is all on a single line. Because both E-Prime and the MathLink library interpret the backslash as an escape character, it has to be doubled twice (hence quadrupled). Finally, the path must be enclosed in " so they must be preceded by the escape character (the backslash).

To close the kernel, the instructions are:

```
MLClose MLLink
MLDeinitialize MLEnv
```

2- The protocol to communicate with the Mathematical kernel

Before it can be sent to the kernel, an expression must be

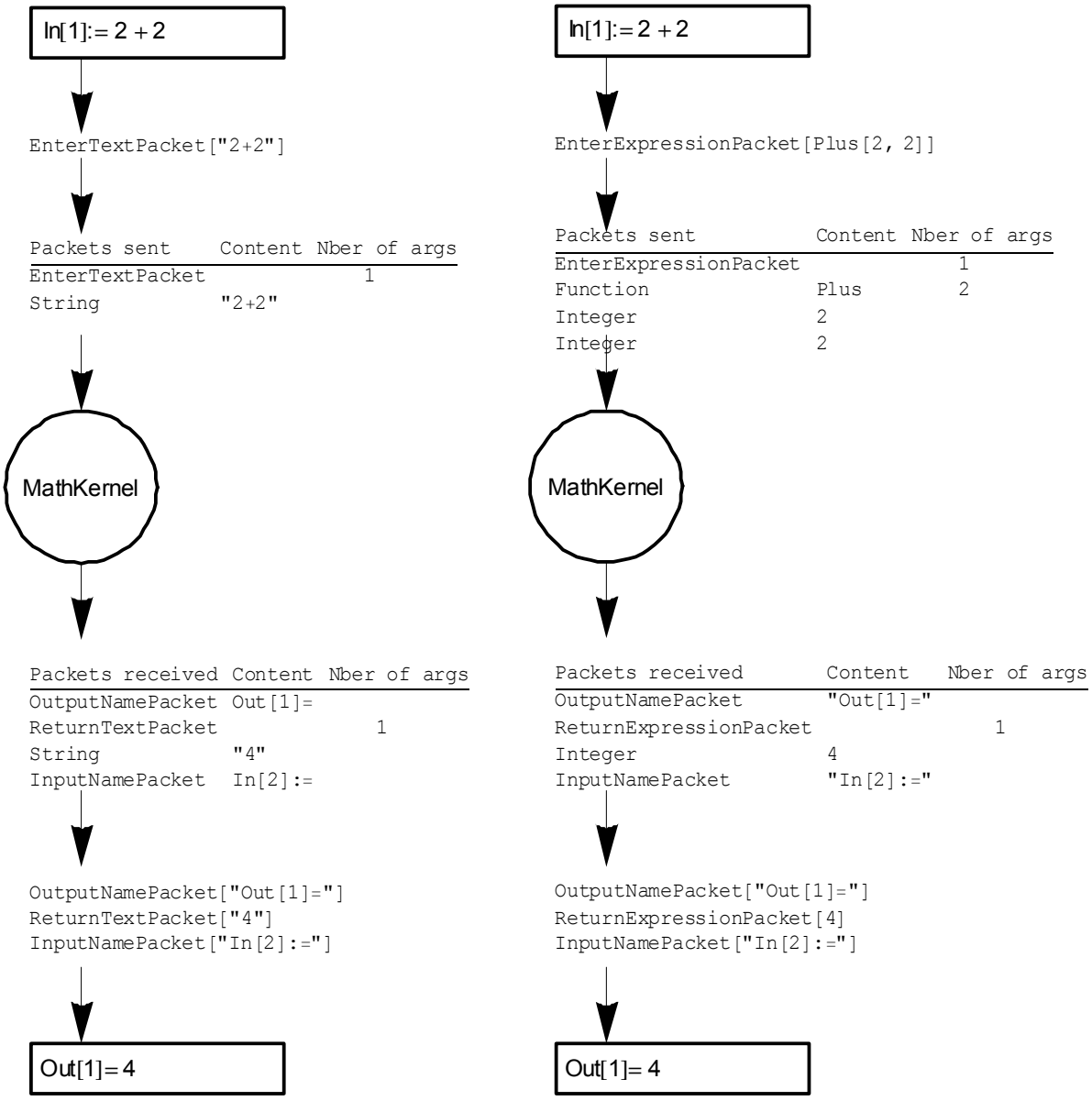

Figure 2. Examples of protocol in response to the input 2+2. Left: the protocol uses a text packet; right: the protocol uses an expression packet. The first step is to put the wrappers, then each element of the packet can be sent. Likewise, the kernel returns packets that include wrappers.

packaged, that is, enclosed in wrappers which identify what kind of package is being sent to the kernel. Since it is most convenient to send a string of text containing a Mathematica command, the wrapper is often **EnterTextPacket** followed by a Mathematica expression, e.g. "2+2" (here, **Enter** is to be understood as "Entered into the kernel").

Once received, the kernel returns the response, accordingly wrapped in a **ReturnTextPacket**, followed by a string containing the response, e.g. "4" (here, **Return** means "Returned from the kernel"). The kernel also returns identifications of the current output label (e.g. "Out[1]=") wrapped in **OutputNamePacket** as well as the next input label (e.g. "In[2]:=") wrapped in **InputNamePacket**. In addition, if an error occurred, an **ErrorMessagePacket** and a **TextPacket** will be issued giving the error name and the error description (e.g. Part::partd : Part specification <<1>> is longer than depth of object). Figure 2 left gives an example where the wrappers are shown.

If more flexibility is required, instructions can be sent under the form of an expression. Expressions detail explicitly the functions, the symbols, the integers, the reals and the strings being sent. For example, the expression 2+2 is in fact given by Plus[2,2] (use FullForm in Mathematica to know how to represent an expression explicitly). To do so, the expression is wrapped in an **EnterExpressionPacket** and the response is wrapped in a **ReturnExpressionPacket**. Figure 2 right gives and example.

Once the packets have been set, they must be sent using the appropriate function from the MathLink library, one function call per element in the packet.

Functions are sent using the function MLPutFunction followed by the name of the function (or wrapper, they are seen as function as well) and the number of arguments to the function. Data are sent using MLPutInteger, MLPutReal or MLPutString depending on the type of data. Finally, symbol names (variables and constants) are sent using MLPutSymbol. To signal the end of a packet, use MLEndPacket. Listing 1, top, indicates how to send a string containing 2+2 using the **EnterTextPacket** while Listing 2, bottom does the same using an expression. Those functions returns a non-zero value if an error occurred (but if the link opened properly and the packet is syntaxically correct, there should not be an error).

Table 1 gives the list of functions that are required to manage a link to a kernel. If the wrapper **EnterTextExpression** is used, the functions to put and get reals, integers and symbols are no longer useful since everything is sent and received as strings.

The kernel places the response packets in a waiting list where they can be fetched. The packets have a packet type and a packet content. For example, in response to Plus[2,2], the response packet will be of type Integer and its content will be the number 4. Response packets are wrapped in a **ReturnTextPacket**. This packet has a type of its own (see Table 2) but has no content of its own. It only signals that a string packets follows.

To know the packet type, use the function MLNextPacket. Packet types are identified by a unique number, listed in Table 2. As wrapper packets have no content, simply fetch the subsequent packet with MLGetNext or skip the remaining of the packet with MLNewPacket.

Table 1. Functions part of the MathLink library

Functions that send packet content to the kernel; returns 0 if no error

```
MLPutFunction(link, "function name",
  number of arguments)
MLPutString(link, "string content")
MLPutReal(link, realvalue)
MLPutInteger(link, integervalue)
MLPutSymbol(link, "symbol name")
```

Functions that returns the packet type (see Table 2)

`MLNextPacket(link)`	`get the beginning of a packet`
`MLGetNext(link)`	`get the next item in the packet`
`MLNewPacket(link)`	`skip the whole packet`

Functions that get the packet content; returns 0 if no error

```
MLGetFunction(link, stringvariable,
nbargs)
MLGetString(link, stringvariable)
MLGetReal(link, realvariable)
MLGetInteger(link, integervariable)
MLGetSymbol(link, stringvariable)
```

Listing1. Sending packets with E-Basic

Listing 1a: Sending an EnterTextPacket

```
Dim ErrNo As Long
ErrNo = MLPutFunction(MLLink,
"EnterTextPacket",1)
ErrNo = MLPutString(MLLink, "2+2")
ErrNo = MLEndPacket(MLLink)
```

Listing 1b: Sending an EnterExpressionPacket

```
Dim ErrNo As Long
ErrNo = MLPutFunction(MLLink,
"EnterExpressionPacket",1)
ErrNo = MLPutFunction(MLLink, Plus,2)
ErrNo = MLPutInteger(MLLink,2)
ErrNo = MLPutInteger(MLLink,2)
ErrNo = MLEndPacket(MLLink)
```

3- Integrating all this into E-Prime

All the previous considerations can be reduced to four operations: Opening and closing the kernel, sending Mathematica instructions and receiving responses. To achieve this within E-Prime, we programmed two sub-procedures and two functions:

`Sub MLStart`	starts the kernel in mathlink mode; you must verify that the path to the file MathKernel.exe is correct on your system.
`Sub MLEnd`	end and close the kernel.

Table 2: types of packet and the number identifying them (received by MLNextPacket or MLGetNext).

Wrappers		*Data*	
`ReturnExpressionPacket`	16	`Function`	70
`ReturnTextPacket`	4	`String`	34
`InputNamePacket`	8	`Integer`	43
`OutputNamePacket`	9	`Real`	42
`ErrorMessage`	5	`Symbol`	35
`TextPacket`		2	

`Function MLWrite(astring) As String` This is the function that you will use to send an expression as a string and get the result as a string as well.

`Function MLRead(n) As String` This function reads *n* packets from the kernel. You should not use this function, as MLWrite reads the returned packets automatically after the command has been sent.

As an example within E-Prime, you can use the following in an Inline object:

```
msgbox "The result of 2+2 is " & MLWrite("2+2")
```

Any command known to Mathematica can be send, e.g.

```
msgbox "The result of 1-e^(-i pi) is " &
MLWrite("1-E^(-I Pi)")
```

in which E is 2.7182, I is the square root of -1 and Pi is 3.1415.

Appendix 1 gives the code for all four subprograms. This code must be given to E-Prime in the "User" tab of the script page (visible using the menu View: Script).

The code given in the appendix, in addition to interact with the kernel, also send an echo of all the inputs and outputs processed by the kernel in the Ouptut window of E-Prime (near the bottom of the screen). In case of difficulties, this might be helpful for debugging.

4- A complete example

As an example, we show how to use MathKernel to create noisy images, with additive noise in proportion to the proportion of success achieved in the previous trials. The example is available on the journal's web site (don't forget to adapt the paths to your system in the MLStart subroutine).

At the beginning of the experiment, it is necessary to start the kernel so an Inline object is added which contains only the instruction`

```
MLStart
```

Likewise, at the end of the experiment, an Inline object contains

```
MLEnd
```

At the beginning of the experiment, we must inform the kernel of the locations where the original, unnoisy images are to be found. It is not possible to send backslashes with the MathLink protocol within E-Prime (the kernel, the protocol and E-Prime all believes that the backslashes are escape characters, resulting in uncontrollable interactions). A way around is to use the Mathematica function ToFileName which can be use to construct a complete path from separate folder names. Hence:

```
Dim useless as string
useless = MLWrite("mypath = ToFileName[{\"C:\",
      \"Documents and settings\",\"\yourusername\"];
")
```

in which \" represents literally the double-quote character, not the end of the E-Prime string. The result (here "Null" since the Mathematica instruction ends with a semi-comma) is useless so we discard it.

To keep count of the number of trials and the number of successes, we also set two counters at zero in the kernel before the block begins:

```
useless = MLWrite("nbtrial = 0; nbsuccess = 0;")
```

Finally, in order to add additive noise to an image, a random matrix is superposed to the image matrix. In Mathematica 7, the instructions for an image of 360 pixels by 360 pixels would be:

```
noise = RandomReal[NormalDistribution[0, nbsuccess
/ nbtrial],{360,360}];
image = Import[mypath<>"imageX.bmp", "GrayLevels"];
image2 = Image[image + noise];
Export[mypath<>"imageX2.bmp",image2];
```

In these Mathematica instructions, the letter X should be replaced by the current stimulus from E-Prime, which we can get with:

```
dim stim as string
stim = c.getattrib("StimulusIdentity")
```

as long as the attribute "StimulusIdentity" is defined in the list of stimuli in E-Prime. Therefore, the second line above is sent to the kernel with

```
useless = MLWrite("image = Import[mypath<>\"image"
+ stim + ".bmp,\"GrayLevels\"];")
```

in which the segment + stim + interrupts the command string, insert the variable stim then resume it.

As seen, it was possible to generate an experiment generating complex stimuli in E-Prime with as few as a dozen commands in four Inline objects by using the computational power of Mathematica.

The present example is deliberately very simple. To save lines of code, more complex operations could be preprogrammed with Mathematica functions and saved in a .m file (e.g. MyFunctions.m). This file could be loaded into the kernel with

```
useless = MLWrite("Get[mypath <> \"MyFunctions.m\"]
" )
```

if the file is located in the folder mypath.

The file MyFunctions.m might contain the following Mathematica function

```
AdditiveNoise[stim_,prop_]:=Module[{},
  noise=RandomReal[NormalDistribution[0,2
      prop+0.0001],{360,360}];
  image=Import[mypath<>"image"<>
      ToString[stim]<> ".bmp","GrayLevels"];
        image2=Image[image+noise];
  Export[mypath<>"image"<>ToString[stim]<>
"b.bmp",image2]
```

```
]
```

so that the randomly generated stimuli would be obtained with only one line of code

```
useless = MLWrite("AdditiveNoise[" + stim +
",nbsuccess/nbtrial]")
```

This demonstration is also available on the journal's web site. Finally, the function name AdditiveNoise could even be an attribute manipulated in E-Prime:

```
fct = c.getattrib("NoiseFunction")
```

so that

```
useless = MLWrite(fct + "[" + stim +
",nbsuccess/nbtrial]")
```

would generate a noisy image using a noise function selected by E-Prime. The possibilities becomes endless.

References

Press, W. H., Flannery, B. P., Teukolsky, S. A. & Vetterling, W. T. (1986). Numerical Recipes: The art of scientific computing. New York: Cambridge University Press.

Brainard, D. H. (1997). *The Psychophysics Toolbox*. Spatial Vision, 10, 433-436.

Chen, W., Cui, Y., Zhang, J. (2005). *Introduction to E-Prime and its application*. Psychological Science (China), 28, 1456-1458.

E-Prime (Version 1.1) [Computer program] (2004).

Wolfram, S. (1996). The Mathematica Book (third edition). New York: Cambridge University Press.

Manuscript received June 16, 2009

Appendix follows.

Appendix: Code that needs to be added to E-Prime to manage communications with the Mathematica Kernel

```
'+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
'                                    Global variables
'+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

' Global variables containing the connections environnement and link
Public MLEnv As Long, MLLink As Long, MLerrno As Long

' Global variables containing the input label and output label from the kernel, e.g. In[1]:=
Public MLcurrentinput As String, MLcurrentoutput As String


'+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
'                             Functions provided by MathLink
'+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

' All the following functions are part of the MathLink library,
' installed in C:\Windows\system32 by Mathematica
' Here, we use the third implementation (the most recent one) of the 32 bytes version
Declare Function MLInitialize Lib "ML32I3.dll" (ByVal p As Long) As Long
Declare Function MLOpenString Lib "ML32I3.dll" (ByVal env As Long, ByVal comm As String, ByRef errno As Long)
As Long
Declare Sub MLClose Lib "ML32I3.dll" (ByVal link As Long)
Declare Sub MLDeinitialize Lib "ML32I3.dll" (ByVal env As Long)

Declare Function MLPutFunction Lib "ML32I3.dll" (ByVal link As Long, ByVal funct As String, ByVal n As Long)
As Long
```

```
Declare Function MLPutSymbol Lib "ML32I3.dll" (ByVal link As Long, ByVal symb As String) As Long
Declare Function MLPutString Lib "ML32I3.dll" (ByVal link As Long, ByVal comm As String) As Long
Declare Function MLPutInteger Lib "ML32I3.dll" (ByVal link As Long, ByVal n As Integer) As Long
Declare Function MLPutDouble Lib "ML32I3.dll" (ByVal link As Long, ByVal x As Double) As Long

Declare Function MLGetString Lib "ML32I3.dll" (ByVal link As Long, ByRef funct As String) As Long
Declare Function MLGetFunction Lib "ML32I3.dll" (ByVal link As Long, ByRef funct As String, ByRef n As Long)
As Long
Declare Function MLGetInteger Lib "ML32I3.dll" (ByVal link As Long, ByRef n As Integer) As Long
Declare Function MLGetDouble Lib "ML32I3.dll" (ByVal link As Long, ByRef x As Double) As Long
Declare Function MLGetSymbol Lib "ML32I3.dll" (ByVal link As Long, ByRef symb As String) As Long

Declare Function MLEndPacket Lib "ML32I3.dll" (ByVal link As Long) As Long
Declare Function MLNextPacket Lib "ML32I3.dll" (ByVal Link As Long ) As Long
Declare Function MLGetNext Lib "ML32I3.dll" (ByVal link As Long) As Long
Declare Function MLNewPacket Lib "ML32I3.dll" (ByVal Link As Long ) As Long


'+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
'             Functions and sub-procedures for Kernel <-> E-Prime exchanges
'+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Function MLRead(nb As Integer) as String
  ' nb is the number of packets to read
  ' Do not use unless you know the kernel has issued a packet

  ' MLNextPacket and MLGetNext return one of the following:
  ' 16 (returned expression packet) 8 (inputname packet) 9 (outputname packet)
  ' 5 (error message packet) 2 (content of an error message)
  ' 70 (function), 34 (string), 43 (integer) 42 (real) 35 (symbol)

  Dim nbarg As Long, ans As Integer, x As Double
  Dim pkttype As Long, res As Long, tmpstr As String
  Dim errornam As String, errortxt As String, errordsc As String

  Do While nb >= 1
    pkttype = MLNextPacket(MLLink) 'Get the packet head type

    Select Case pkttype
      Case 16 'This signals the begining of a complex expression
        ' in fact, it should never be anything else but a string, but just in case...
        pkttype = MLGetNext(MLLink)
        Select case pkttype
          Case 34 'a string
            res = MLGetString(MLLink, tmpstr)
          Case 70 'a function name
            res = MLGetFunction(MLLink, tmpstr, nbarg)
            res = MLNewPacket(MLLink) 'lets skip the remaining of the packet...
            tmpstr= tmpstr+"[...(" & nbarg & " arguments)...]"
          Case 35 'a symbol
            res = MLGetSymbol(MLLink, tmpstr)
```

```
          Case 42 'a real number
            res = MLGetDouble(MLLink, x)
            tmpstr=cstr(x)
          Case 43 'an integer
            res = MLGetInteger(MLLink, ans)
            tmpstr=cstr(ans)
          Case Else
            MsgBox "ML warning: unknown packet type: " & pkttype
        End select
        Debug.print MLcurrentoutput & " " & tmpstr

      Case 8 ' input and output labels
        res = MLGetString(MLLink, MLcurrentinput)
      Case 9
        res = MLGetString(MLLink, MLcurrentoutput)

      ' treating error message sent by mathematica; always sent by three.
      Case 5
        res = MLGetSymbol(MLLink, errornam)    'Get the symbol of the error
        res = MLGetString(MLLink, errortxt)    'Get the name of the error
        res = MLGetFunction(MLLink, errordsc, nbarg) 'Skip the "TextPacket" wrappper
        res = MLGetString(MLLink, errordsc)          'Get the description of the error

        Debug.print errornam & "::" & errortxt & " " & errordsc
        nb = nb + 1 ' this was unexpected, so read one extra packet

     Case 2 ' a textpacket producted by Print[]
       res = MLGetString(MLLink,tmpstr)
             Debug.print "\t\t" & tmpstr
             nb = nb + 1 ' this was unexpected so read one extra packet

      Case Else
        MsgBox "ML warning: unknown packet head: " & pkttype

    End Select
    nb = nb - 1
  Loop

  MLRead = tmpstr  ' returns the answer given by the kernel

End Function


Function MLWrite(astr As String) as String
  'The expression is wrapped within "ToString" so that the result ought to be a string
  'The expression to evaluate is ToString[ToExpression[astr]]
  'The result of this function is the string returned by the kernel
  dim res as long

  res=MLPutFunction(MLLink, "EnterExpressionPacket",1)
  res=MLPutFunction(MLLink, "ToString",1)
```

```
  res=MLPutFunction(MLLink, "ToExpression",1)
  res=MLPutString(MLLink, astr)
  res=MLEndPacket (MLLink)
  debug.print MLcurrentinput & astr

  MLWrite = MLRead(3) 'reads the output name, the result, and the new input name

End Function


'+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
'                     Sub-procedures for starting/closing the kernel
'+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Sub MLLaunch()
  ' The mathematica connection string requires to quadruple the backslash in the path
  ' Adapt the path according to your system
  dim strmath as string, temp as String
  strmath="-LinkMode Launch -LinkName \"c:\\\\Program Files\\\\Wolfram
Research\\\\Mathematica\\\\7.0\\\\MathKernel.exe -mathlink\" "

  ' initialize then start the mathematica kernel in mode -mathlink
  MLenv = MLInitialize(0)
  MLLink = MLOpenString(MLenv, strmath, MLerrno)
  If MLerrno = 0 then
    Debug.print "MATHLINK::Mathematica started with success"
  else
    Debug.print "MATHLINK::Mathematica kernel has returned an error " & MLerrno
  end if

  ' reads the initialisation message
  temp = MLRead(1) 'it should return "In[1]:=" only
end sub


Sub MLEnd()
  ' close the link then deinitialize the environnement
  MLClose MLLink
  MLDeinitialize MLenv
  Debug.print "MATHLINK::Mathematica closed."
end sub
```