Wolfram Mathematica® Tutorial Collection

SYSTEMS INTERFACES AND DEPLOYMENT
# Contents

## Global Aspects of Mathematica Sessions
- The Main Loop .......................................................... 1
- Dialogs ........................................................................ 7
- Date and Time Functions .............................................. 10
- Memory Management ................................................. 17
- Global System Information ........................................... 20
- Mathematica Sessions .................................................. 23

## The Internals of Mathematica
- Why You Do Not Usually Need to Know about Internals .. 29
- Basic Internal Architecture ......................................... 33
- The Algorithms of Mathematica .................................... 35
- The Software Engineering of Mathematica .................... 36
- Testing and Verification ............................................... 38

## Security and Connectivity
- Mathematica Internet Connectivity ............................... 40
- Notebook Security ........................................................ 43

## MathLink and External Program Communication
- Introduction to MathLink .............................................. 49
- How MathLink Is Used ................................................ 50
- Installing Existing MathLink-Compatible Programs .......... 52
- Setting Up External Functions to Be Called from Mathematica ................................. 53
- Handling Lists, Arrays, and Other Expressions ............... 59
- Portability of MathLink Programs .................................. 72
- Using MathLink to Communicate between Mathematica Sessions ................................. 76
- Calling Subsidiary Mathematica Processes ....................... 80
- Two-Way Communication with External Programs .......... 85
- Running Programs on Remote Computers ....................... 88
- Running External Programs under a Debugger ................ 89
- Manipulating Expressions in External Programs .............. 90
- Error and Interrupt Handling ....................................... 95
- Running Mathematica from Within an External Program ............ 96
- MathLink Interface 3 ..................................................... 100
Global Aspects of Mathematica Sessions

The Main Loop

In any interactive session, Mathematica effectively operates in a loop. It waits for your input, processes the input, prints the result, then goes back to waiting for input again. As part of this “main loop”, Mathematica maintains and uses various global objects. You will often find it useful to work with these objects.

You should realize, however, that if you use Mathematica through a special front end, your front end may set up its own main loop, and what is said here may not apply.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>In[n]</td>
<td>the expression on the ( n^{\text{th}} ) input line</td>
</tr>
<tr>
<td>InString[n]</td>
<td>the textual form of the ( n^{\text{th}} ) input line</td>
</tr>
<tr>
<td>% n or Out[n]</td>
<td>the expression on the ( n^{\text{th}} ) output line</td>
</tr>
<tr>
<td>Out[{n_1,n_2,...}]</td>
<td>a list of output expressions</td>
</tr>
<tr>
<td>%% ... % (( n ) times) or Out[-n]</td>
<td>the expression on the ( n^{\text{th}} ) previous output line</td>
</tr>
<tr>
<td>MessageList[n]</td>
<td>a list of messages produced while processing the ( n^{\text{th}} ) line</td>
</tr>
<tr>
<td>$Line</td>
<td>the current line number (resettable)</td>
</tr>
</tbody>
</table>

Input and output expressions.

In a standard interactive session, there is a sequence of input and output lines. Mathematica stores the values of the expressions on these lines in \( \text{In}[n] \) and \( \text{Out}[n] \).

As indicated by the usual \( \text{In}[n] := \) prompt, the input expressions are stored with delayed assignments. This means that whenever you ask for \( \text{In}[n] \), the input expression will always be reevaluated in your current environment.
This assigns a value to \( x \).

\[
\text{In}[1]:= \quad x = 7
\]

\[
\text{Out}[1]= 7
\]

Now the value for \( x \) is used.

\[
\text{In}[2]:= \quad x - x^2 + 5 x - 1
\]

\[
\text{Out}[2]= -8
\]

This removes the value assigned to \( x \).

\[
\text{In}[3]:= \quad x = .
\]

This is reevaluated in your current environment, where there is no value assigned to \( x \).

\[
\text{In}[4]:= \quad \text{In}[2]
\]

\[
\text{Out}[4]= -1 + 6 x - x^2
\]

This gives the textual form of the second input line, appropriate for editing or other textual manipulation.

\[
\text{In}[5]:= \quad \text{InString}[2] // \text{InputForm}
\]

\[
\text{Out}[5]//\text{InputForm}= "\(x - \(x^2\) + \(5 x\) - 1\)"
\]

---

$\text{HistoryLength}$

the number of previous lines of input and output to keep

Specifying the length of session history to keep.

*Mathematica* by default stores all your input and output lines for the duration of the session. In a very long session, this may take up a large amount of computer memory. You can nevertheless get rid of the input and output lines by explicitly clearing the values of In and Out, using Unprotect [In, Out], followed by Clear [In, Out]. You can also tell *Mathematica* to keep only a limited number of lines of history by setting the global variable $\text{HistoryLength}$.

Note that at any point in a session, you can reset the line number counter $\text{Line}$, so that, for example, new lines are numbered so as to overwrite previous ones.
Global functions used in the main loop.

*Mathematica* provides a variety of “hooks” that allow you to insert functions to be applied to expressions at various stages in the main loop. Thus, for example, any function you assign as the value of the global variable $\$Pre$ will automatically be applied before evaluation to any expression you give as input.

For a particular input line, the standard main loop begins by getting a text string of input. Particularly if you need to deal with special characters, you may want to modify this text string before it is further processed by *Mathematica*. You can do this by assigning a function as the value of the global variable $\$PreRead$. This function will be applied to the text string, and the result will be used as the actual input string for the particular input line.

This tells *Mathematica* to replace listHead by {...} in every input string.

```
In[6]:= $PreRead = (ReplaceAll[#, "listHead" :> "List"] &)
Out[6]= #1 /. listHead :> List &
```

You can now enter lists as listHead expressions.

```
In[7]:= listHead[a, b, c]
Out[7]= {a, b, c}
```

You can remove the value for $\$PreRead$ like this, at least so long as your definition for $\$PreRead$ does not modify this very input string.

```
In[8]:= $PreRead =.
```

Once *Mathematica* has successfully read an input expression, it then evaluates this expression. Before doing the evaluation, *Mathematica* applies any function you have specified as the value

<table>
<thead>
<tr>
<th>Global function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$PreRead$</td>
<td>a function applied to each input string before being fed to <em>Mathematica</em></td>
</tr>
<tr>
<td>$$Pre$</td>
<td>a function applied to each input expression before evaluation</td>
</tr>
<tr>
<td>$$Post$</td>
<td>a function applied to each expression after evaluation</td>
</tr>
<tr>
<td>$$PrePrint$</td>
<td>a function applied after Out[n] is assigned, but before the result is printed</td>
</tr>
<tr>
<td>$$SyntaxHandler$</td>
<td>a function applied to any input line that yields a syntax error</td>
</tr>
</tbody>
</table>
of $\text{Pre}$, and after the evaluation, it applies any function specified as the value of $\text{Post}$. Note that unless the $\text{Pre}$ function holds its arguments unevaluated, the function will have exactly the same effect as $\text{Post}$.

$\text{Post}$ allows you to specify arbitrary “postprocessing” to be done on results obtained from *Mathematica*. Thus, for example, to make *Mathematica* get a numerical approximation to every result it generates, all you need do is to set $\text{Post} = \text{N}$.

This tells *Mathematica* to apply \text{N} to every result it generates.

\[
\text{In}[9]:= \text{Post} = \text{N}
\]
\[
\text{Out}[9]= \text{N}
\]

Now *Mathematica* gets a numerical approximation to anything you type in.

\[
\text{In}[10]:= \text{Sqrt}[7]
\]
\[
\text{Out}[10]= 2.64575
\]

This removes the postprocessing function you specified.

\[
\text{In}[11]:= \text{Post} = .
\]

As soon as *Mathematica* has generated a result, and applied any $\text{Post}$ function you have specified, it takes the result, and assigns it as the value of Out[$\text{Line}$]. The next step is for *Mathematica* to print the result. However, before doing this, it applies any function you have specified as the value of $\text{PrePrint}$.

This tells *Mathematica* to shorten all output to two lines.

\[
\text{In}[12]:= \text{PrePrint} = \text{Short}[\#, 2] &;
\]

Only a two-line version of the output is now shown.

\[
\text{In}[13]:= \text{Expand}[(x + y)^{40}]
\]
\[
\text{Out}[13]= x^{40} + 40 x^{39} y + 780 x^{38} y^2 + 9880 x^{37} y^3 + 91390 x^{36} y^4 + 658008 x^{35} y^5 + \ll 30 \gg + 91390 x^4 y^{36} + 9880 x^3 y^{37} + 780 x^2 y^{38} + 40 x y^{39} + y^{40}
\]

This removes the value you assigned to $\text{PrePrint}$.

\[
\text{In}[14]:= \text{PrePrint} = .
\]
There are various kinds of output generated in a typical *Mathematica* session. In general, each kind of output is sent to a definite output channel, as discussed in "Streams and Low-Level Input and Output". Associated with each output channel, there is a global variable which gives a list of the output streams to be included in that output channel.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Output</td>
<td>standard output and text generated by Print</td>
</tr>
<tr>
<td>$Echo</td>
<td>an echo of each input line (as stored in InString[n])</td>
</tr>
<tr>
<td>$Urgent</td>
<td>input prompts and other urgent output</td>
</tr>
<tr>
<td>$Messages</td>
<td>standard messages and output generated by Message</td>
</tr>
</tbody>
</table>

Output channels in a standard *Mathematica* session.

By modifying the list of streams in a given output channel, you can redirect or copy particular kinds of *Mathematica* output. Thus, for example, by opening an output stream to a file, and including that stream in the $Echo list, you can get each piece of input you give to *Mathematica* saved in a file.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streams[]</td>
<td>list of all open streams</td>
</tr>
<tr>
<td>Streams[&quot;name&quot;]</td>
<td>list of all open streams with the specified name</td>
</tr>
<tr>
<td>$Input</td>
<td>the name of the current input stream</td>
</tr>
</tbody>
</table>

Open streams in a *Mathematica* session.

The function Streams shows you all the input, output and other streams that are open at a particular point in a *Mathematica* session. The variable $Input gives the name of the current stream from which *Mathematica* input is being taken at a particular point. $Input is reset, for example, during the execution of a Get command.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MessagePrePrint</td>
<td>a function to be applied to expressions that are given in messages</td>
</tr>
<tr>
<td>$Language</td>
<td>list of default languages to use for messages</td>
</tr>
</tbody>
</table>

Parameters for messages.

There are various global parameters which determine the form of messages generated by *Mathematica*.
As discussed in "Messages", typical messages include a sequence of expressions which are combined with the text of the message through `StringForm`. `$MessagePrePrint` gives a function to be applied to the expressions before they are printed. The default for `$MessagePrePrint` uses `Short` for text formatting and a combination of `Short` and `Shallow` for typesetting.

As discussed in "International Messages", Mathematica allows you to specify the language in which you want messages to be produced. In a particular Mathematica session, you can assign a list of language names as the value of `$Language`.

<table>
<thead>
<tr>
<th>Exit[] or Quit[]</th>
<th>terminate your Mathematica session</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Epilog</td>
<td>a global variable to be evaluated before termination</td>
</tr>
</tbody>
</table>

Terminating Mathematica sessions.

Mathematica will continue in its main loop until you explicitly tell it to exit. Most Mathematica interfaces provide special ways to do this. Nevertheless, you can always do it by explicitly calling `Exit` or `Quit`.

Mathematica allows you to give a value to the global variable `$Epilog` to specify operations to perform just before Mathematica actually exits. In this way, you can for example make Mathematica always save certain objects before exiting.

<table>
<thead>
<tr>
<th>$IgnoreEOF</th>
<th>whether to ignore the end-of-file character</th>
</tr>
</thead>
</table>

A global variable that determines the treatment of end-of-file characters.

As discussed in "Special Characters: Strings and Characters", Mathematica usually does not treat special characters in a special way. There is one potential exception, however. With the default setting `$IgnoreEOF = False`, Mathematica recognizes end-of-file characters. If Mathematica receives an end-of-file character as the only thing on a particular input line in a standard interactive Mathematica session, then it will exit the session.

Exactly how you enter an end-of-file character depends on the computer system you are using. Under Unix, for example, you typically press `Ctrl+D`.

Note that if you use Mathematica in a “batch mode”, with all its input coming from a file, then it will automatically exit when it reaches the end of the file, regardless of the value of `$IgnoreEOF`. 
Dialogs

Within a standard interactive session, you can create "subsessions" or dialogues using the Mathematica command Dialog. Dialogs are often useful if you want to interact with Mathematica while it is in the middle of doing a calculation. As mentioned in "Tracing Evaluation", TraceDialog for example automatically calls Dialog at specified points in the evaluation of a particular expression. In addition, if you interrupt Mathematica during a computation, you can typically "inspect" its state using a dialog.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dialog[]</td>
<td>initiate a Mathematica dialog</td>
</tr>
<tr>
<td>Dialog[expr]</td>
<td>initiate a dialog with expr as the current value of %</td>
</tr>
<tr>
<td>Return[]</td>
<td>return from a dialog, taking the current value of % as the return value</td>
</tr>
<tr>
<td>Return[expr]</td>
<td>return from a dialog, taking expr as the return value</td>
</tr>
</tbody>
</table>

Initiating and returning from dialogs.

This initiates a dialog.

\[\text{In}[1]:= \text{Dialog[]}\]

You can do computations in a dialog just as you would in any Mathematica session.

\[\text{In}[2]:= 2^41\]
\[\text{Out}[2]= 2199023255552\]

You can use Return to exit a dialog.

\[\text{In}[3]:= \text{Return[]}\]
\[\text{Out}[3]= 2199023255552\]

When you exit a dialog, you can return a value for the dialog using Return[expr]. If you do not want to return a value, and you have set $\text{IgnoreEOF} = \text{False}$, then you can also exit a dialog simply by giving an end-of-file character, at least on systems with text-based interfaces.

To evaluate this expression, Mathematica initiates a dialog.

\[\text{In}[4]:= 1 + \text{Dialog[]}^2\]
The value \(a + b\) returned from the dialog is now inserted in the original expression.

\[
\text{In}[5]:= \text{Return}[a + b]
\]

\[
\text{Out}[5]= 1 + (a + b)^2
\]

In starting a dialog, you will often find it useful to have some “initial expression”. If you use \texttt{Dialog[expr]}, then \textit{Mathematica} will start a dialog, using \textit{expr} as the initial expression, accessible for example as the value of \%.  

This first starts a dialog with initial expression \(a^2\).

\[
\text{In}[6]:= \text{Map[Dialog, \{a^2, b + c\}]}
\]

\[
\text{Out}[6]= a^2
\]

\% is the initial expression in the dialog.

\[
\text{In}[7]:= \%^2 + 1
\]

\[
\text{Out}[7]= 1 + a^4
\]

This returns a value from the first dialog, and starts the second dialog, with initial expression \(b + c\).

\[
\text{In}[8]:= \text{Return[\%]}
\]

\[
\text{Out}[8]= b + c
\]

This returns a value from the second dialog. The final result is the original expression, with values from the two dialogs inserted.

\[
\text{In}[9]:= \text{Return[444]}
\]

\[
\text{Out}[9]= \{1 + a^4, 444\}
\]

Dialog effectively works by running a subsidiary version of the standard \textit{Mathematica} main loop. Each dialog you start effectively "inherits" various values from the overall main loop. Some of the values are, however, local to the dialog, so their original values are restored when you exit the dialog.

Thus, for example, dialogs inherit the current line number \texttt{$Line} when they start. This means that the lines in a dialog have numbers that follow the sequence used in the main loop. Nevertheless, the value of \texttt{$Line} is local to the dialog. As a result, when you exit the dialog, the value of \texttt{$Line} reverts to what it was in the main loop.
If you start a dialog on line 10 of your *Mathematica* session, then the first line of the dialog will be labeled In[11]. Successive lines of the dialog will be labeled In[12], In[13] and so on. Then, when you exit the dialog, the next line in your main loop will be labeled In[11]. At this point, you can still refer to results generated within the dialog as Out[11], Out[12] and so on. These results will be overwritten, however, when you reach lines In[12], In[13], and so on in the main loop.

In a standard *Mathematica* session, you can tell whether you are in a dialog by seeing whether your input and output lines are indented. If you call a dialog from within a dialog, you will get two levels of indentation. In general, the indentation you get inside \( d \) nested dialogs is determined by the output form of the object `DialogIndent[d]`. By defining the format for this object, you can specify how dialogs should be indicated in your *Mathematica* session.

<table>
<thead>
<tr>
<th>DialogSymbols: ( {x,y,...} )</th>
<th>symbols whose values should be treated as local to the dialog</th>
</tr>
</thead>
<tbody>
<tr>
<td>DialogSymbols: ( {x=x_0,y=y_0,...} )</td>
<td>symbols with initial values</td>
</tr>
<tr>
<td>DialogProlog: ( expr )</td>
<td>an expression to evaluate before starting the dialog</td>
</tr>
</tbody>
</table>

Options for `Dialog`.

Whatever setting you give for `DialogSymbols`, `Dialog` will always treat the values of `$Line`, `$Epilog` and `$MessageList` as local. Note that if you give a value for `$Epilog`, it will automatically be evaluated when you exit the dialog.

When you call `Dialog`, its first step is to localize the values of variables. Then it evaluates any expression you have set for the option `DialogProlog`. If you have given an explicit argument to the `Dialog` function, this is then evaluated next. Finally, the actual dialog is started.

When you exit the dialog, you can explicitly specify the return value using `Return[expr]`. If you do not do this, the return value will be taken to be the last value generated in the dialog.
Date and Time Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateList[]</td>
<td>give the current local date and time in the form {year, month, day, hour, minute, second}</td>
</tr>
<tr>
<td>DateList[TimeZone-&gt;z]</td>
<td>give the current date and time in time zone z</td>
</tr>
<tr>
<td>$TimeZone</td>
<td>give the time zone assumed by your computer system</td>
</tr>
</tbody>
</table>

Finding the date and time.

This gives the current date and time.

```
In[1]:= DateList[]
```

The Mathematica `DateList` function returns whatever your computer system gives as the current date and time. It assumes that any corrections for daylight saving time and so on have already been done by your computer system. In addition, it assumes that your computer system has been set for the appropriate time zone.

The variable `$TimeZone` returns the current time zone assumed by your computer system. The time zone is given as the number of hours which must be added to Greenwich Mean Time (GMT) to obtain the correct local time. Thus, for example, U.S. Eastern Standard Time (EST) corresponds to time zone -5. Note that daylight saving time corrections must be included in the time zone, so U.S. Eastern Daylight Time (EDT) corresponds to time zone -4.

This gives the current time zone assumed by your computer system.

```
In[2]:= $TimeZone
```

This gives the current date and time in time zone +9, the time zone for Japan.

```
In[3]:= DateList[TimeZone -> 9]
Out[3]= {2005, 4, 1, 10, 21, 29.579505}
```
### AbsoluteTime[]
- Total number of seconds since the beginning of January 1, 1900

### SessionTime[]
- Total number of seconds elapsed since the beginning of your current Mathematica session

### TimeUsed[]
- Total number of seconds of CPU time used in your current Mathematica session

### $TimeUnit$
- The minimum time interval recorded on your computer system

**Time functions.**

You should realize that on any computer system, there is a certain “granularity” in the times that can be measured. This granularity is given as the value of the global variable $TimeUnit$. Typically it is either about $\frac{1}{100}$ or $\frac{1}{1000}$ of a second.

### Pause [n]
- Pause for at least n seconds

**Pausing during a calculation.**

This gives various time functions.

\[
\text{In}[4]:= \{\text{AbsoluteTime[]}, \text{SessionTime[]}, \text{TimeUsed[]}\}
\]

\[
\text{Out}[4]= \{3.321285689607146 \times 10^8, 6.125768, 2.24\}
\]

This pauses for 10 seconds, then reevaluates the time functions. Note that TimeUsed[] is not affected by the pause.

\[
\text{In}[5]:= \text{Pause[10]; \{AbsoluteTime[], SessionTime[], TimeUsed\}}
\]

\[
\text{Out}[5]= \{3.321285699616089 \times 10^8, 16.134709, 2.24\}
\]

### AbsoluteTime [date]
- Convert from date to absolute time

### DateList [time]
- Convert from absolute time to date

**Converting between dates and absolute times.**

This sets d to be the current date.

\[
\text{In}[6]:= d = \text{DateList[]}
\]

\[
\]
This adds one month to the current date.

\[\text{In[7]}:= \text{DateList[]} + \{0, 1, 0, 0, 0, 0\}\]

\[\text{Out[7]}= \{2005, 4, 31, 19, 21, 39.629234\}\]

This gives the number of seconds in the additional month.

\[\text{In[8]}:= \text{AbsoluteTime[]} - \text{AbsoluteTime}[d]\]

\[\text{Out[8]}= 2.678400003320 \times 10^6\]

---

**DateList \["string"]**

- convert a date string to a date list

**DateList**

\[\{"string", \{"e_1", \"e_2\", \ldots\}\}\]

- give the date list obtained by extracting elements \"e_i\" from \"string\"

Converting from different date formats.

You can use `DateList\["string"\]` to convert a date string into a date list, as long as the date format is sufficiently unambiguous.

This attempts to interpret the string as a date.

\[\text{In[9]}:= \text{DateList["June 23, 1988 - 3:55 pm"]}\]

\[\text{Out[9]}= \{1988, 6, 23, 15, 55, 0.\}\]

For more control of the conversion, you can specify the order and type of date elements appearing in the string. The elements can be strings like "Year", "Quarter", "Month", "MonthName", "Day", "DayName", "Hour", "AMPM", "Minute", or "Second".

This extracts a date using the specified elements.

\[\text{In[10]}:= \text{DateList["3/5/2001", \{"Month", "Day", "Year\"\}]}\]

\[\text{Out[10]}= \{2001, 3, 5, 0, 0, 0\}\]

If the date element delimiters contain letters or digits, these must also be specified as part of the date elements.

This extracts a date containing a letter as a separator.

\[\text{In[11]}:= \text{DateList["Jun Y1988", \{"MonthName", \"Y\", "Year\"\}]}\]

\[\text{Out[11]}= \{1988, 6, 1, 0, 0, 0\}\]
DateString[]
give a string representing current local date and time

DateString[datespec, elems]
give elements elems of date and time given by datespec

Converting to different date formats.

DateString is used to give a nice string representation of a date and time. The exact output format can be specified from a long list of date elements, such as "DateTime", "DayName", "HourShort", etc.

This gives the current date and time in the default format.

In[12]:=DateString[]

This specifies a format for the given date.

In[13]:=DateString[{1988, 6, 23, 15, 55, 0}, {"MonthName", ",", "DayShort", ",", "Year", "," - ",", "Hour12Short", ",", "Minute", ",", "AMPMLowerCase"}]

DateString is used to give a string representing current local date and time.

DatePattern[elems]
string pattern matching a date with the given elements

Extracting dates from a string.

You can use DatePattern[elems] as a string pattern in string matching functions. The date elements are the same as used in DateList, although the default date element delimiters are restricted to the /, -, : or . characters. Other delimiters can be given explicitly in the list of date elements.

This extracts dates of the given format from a string.

In[14]:=StringCases["abc 12/5/2005 def 1/15/2002 ghi",
x : DatePattern[{"Month", "Day", "Year"}] \[\rightarrow\] DateList[x]]
Datelist::ambig: Warning: the interpretation of the string 12/5/2005 as a date is ambiguous. \[\rightarrow\]
Out[14]=\{\{2005, 12, 5, 0, 0, 0.\}, \{2002, 1, 15, 0, 0, 0.\}\}

This extracts dates with explicit delimiters.

In[15]:=StringCases["abc Mar 2002 def Aug 2005 ghi",
x : DatePattern[{"MonthName", ",", "Year"}] \[\rightarrow\] DateList[x]]
Out[15]=\{\{2002, 3, 1, 0, 0, 0.\}, \{2005, 8, 1, 0, 0, 0.\}\}
Plotting data with date coordinates.

**DateListPlot** can be used to plot data with date or time horizontal coordinates. Dates can be lists, strings, or absolute times as with **DateString**, and **AbsoluteTime**. A date specification **datespec** can be given to associate dates with data given as \( \{y_1, y_2, \ldots\} \). **DateListLogPlot** allows you to plot the data with a logarithmic vertical scale.

This gathers some financial time series data.

\[
\text{In[16]:= } \text{fd} = \text{FinancialData["GE", \{\{1980\}, \{2005\}\}]};
\]

This plots the financial data.

\[
\text{In[17]:= } \text{DateListPlot[fd]}
\]

This plots the same data on a logarithmic scale.

\[
\text{In[18]:= } \text{DateListLogPlot[fd]}
\]
This plots monthly data which does not contain explicit dates.

\[
\text{In}[19] := \text{DateListPlot}[[1, 2, 4, 5, 7, 6, 10, 8, 12, 10, 15], \{\{2008\}, \text{Automatic}, \text{"Month"}\}]
\]

\[
\text{Out}[19] = \text{Timings}
\]

Timing \([expr] \)  
- evaluate \(expr\), and return a list of the CPU time needed, together with the result obtained

AbsoluteTiming \([expr] \)  
- evaluate \(expr\), giving the absolute time taken

Timing \(Mathematica\) operations.

Timing allows you to measure the CPU time associated with the evaluation of a single \(Mathematica\) expression. Timing corresponds to the increase in TimeUsed. Note that only CPU time associated with the actual evaluation of the expression within the \(Mathematica\) kernel is included. The time needed to format the expression for output, and any time associated with external programs, is not included.

AbsoluteTiming allows you to measure absolute total elapsed time. You should realize, however, that the time reported for a particular calculation by both AbsoluteTiming and Timing depends on many factors.

First, the time depends in detail on the computer system you are using. It depends not only on instruction times, but also on memory caching, as well as on the details of the optimization done in compiling the parts of the internal code of \(Mathematica\) used in the calculation.

The time also depends on the precise state of your \(Mathematica\) session when the calculation was done. Many of the internal optimizations used by \(Mathematica\) depend on details of preceding calculations. For example, \(Mathematica\) often uses previous results it has obtained, and avoids unnecessarily reevaluating expressions. In addition, some \(Mathematica\) functions build internal tables when they are first called in a particular way, so that if they are called in that way again, they run much faster. For all of these kinds of reasons, it is often the case that a particular calculation may not take the same amount of time if you run it at different points in the same \(Mathematica\) session.
This gives the CPU time needed for the calculation. The semicolon causes the result of the calculation to be given as Null.

\[ \text{In}[20] := \text{Timing}[100000!;] \]
\[ \text{Out}[20] := \{0.49 \text{ Second, Null}\} \]

Now \textit{Mathematica} has built internal tables for factorial functions, and the calculation takes no measurable CPU time.

\[ \text{In}[21] := \text{Timing}[100000!;] \]
\[ \text{Out}[21] := \{0. \text{ Second, Null}\} \]

However, some absolute time does elapse.

\[ \text{In}[22] := \text{AbsoluteTiming}[100000!;] \]
\[ \text{Out}[22] := \{0.000102 \text{ Second, Null}\} \]

Note that the results you get from \text{Timing} are only accurate to the timing granularity $\text{$\$TimeUnit$}$ of your computer system. Thus, for example, a timing reported as $0$ could in fact be as much as $\text{$\$TimeUnit$}$.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{TimeConstrained[expr,t]} \quad \text{try to evaluate} \ expr, \ \text{aborting the calculation after} \ t \ \text{seconds} \\
\textbf{TimeConstrained[expr,t,failexpr]} \quad \text{return} \ failexpr \ \text{if the time constraint is not met} \\
\hline
\end{tabular}
\end{center}

Time-constrained calculation.

When you use \textit{Mathematica} interactively, it is quite common to try doing a calculation, but to abort the calculation if it seems to be taking too long. You can emulate this behavior inside a program by using \text{TimeConstrained}. \text{TimeConstrained} tries to evaluate a particular expression for a specified amount of time. If it does not succeed, then it aborts the evaluation, and returns either \text{$\$Aborted$}, or an expression you specify.

You can use \text{TimeConstrained}, for example, to have \textit{Mathematica} try a particular approach to a problem for a certain amount of time, and then to switch to another approach if the first one has not yet succeeded. You should realize however that \text{TimeConstrained} may overrun the time you specify if \textit{Mathematica} cannot be interrupted during a particular part of a calculation. In addition, you should realize that because different computer systems run at different speeds, programs that use \text{TimeConstrained} will often give different results on different systems.
Memory Management

<table>
<thead>
<tr>
<th>MemoryInUse[]</th>
<th>number of bytes of memory currently being used by Mathematica</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxMemoryUsed[]</td>
<td>maximum number of bytes of memory used by Mathematica in this session</td>
</tr>
</tbody>
</table>

Finding memory usage.

Particularly for symbolic computations, memory is usually the primary resource which limits the size of computations you can do. If a computation runs slowly, you can always potentially let it run longer. But if the computation generates intermediate expressions which simply cannot fit in the memory of your computer system, then you cannot proceed with the computation.

*Mathematica* is careful about the way it uses memory. Every time an intermediate expression you have generated is no longer needed, *Mathematica* immediately reclaims the memory allocated to it. This means that at any point in a session, *Mathematica* stores only those expressions that are actually needed; it does not keep unnecessary objects which have to be "garbage collected" later.

This gives the number of bytes of memory currently being used by *Mathematica*.

In[1]:= MemoryInUse[]
Out[1]= 947712

This generates a 10000-element list.

In[2]:= Range[10000] // Short
Out[2]= {1, 2, 3, 4, 5, 6, 7, 8, <<9985>>, 9994, 9995, 9996, 9997, 9998, 9999, 10000}

Additional memory is needed to store the list.

In[3]:= MemoryInUse[]
Out[3]= 989616

This list is kept because it is the value of Out[2]. If you clear Out[2], the list is no longer needed.

The memory in use goes down again.

In[5]:= MemoryInUse[]
Out[5]= 954 408

This shows the maximum memory needed at any point in the session.

In[6]:= MaxMemoryUsed[]
Out[6]= 1 467 536

One issue that often comes up is exactly how much memory Mathematica can actually use on a particular computer system. Usually there is a certain amount of memory available for all processes running on the computer at a particular time. Sometimes this amount of memory is equal to the physical number of bytes of RAM in the computer. Often, it includes a certain amount of "virtual memory", obtained by swapping data on and off a mass storage device.

When Mathematica runs, it needs space both for data and for code. The complete code of Mathematica is typically several megabytes in size. For any particular calculation, only a small fraction of this code is usually used. However, in trying to work out the total amount of space available for Mathematica data, you should not forget what is needed for Mathematica code. In addition, you must include the space that is taken up by other processes running in the computer. If there are fewer jobs running, you will usually find that your job can use more memory.

It is also worth realizing that the time needed to do a calculation can depend very greatly on how much physical memory you have. Although virtual memory allows you in principle to use large amounts of memory space, it is usually hundreds or even thousands of times slower to access than physical memory. As a result, if your calculation becomes so large that it needs to make use of virtual memory, it may run much more slowly.

MemoryConstrained[expr,b] try to evaluate expr, aborting if more than b additional bytes of memory are requested

MemoryConstrained[expr,b,failexpr] return failexpr if the memory constraint is not met

Memory-constrained computation.

MemoryConstrained works much like TimeConstrained. If more than the specified amount of memory is requested, MemoryConstrained attempts to abort your computation. As with TimeConstrained, there may be some overshoot in the actual amount of memory used before the computation is aborted.
Finding the size of expressions.

Although you may find `ByteCount` useful in estimating how large an expression of a particular kind you can handle, you should realize that the specific results given by `ByteCount` can differ substantially from one version of *Mathematica* to another.

Another important point is that `ByteCount` always gives you the *maximum* amount of memory needed to store a particular expression. Often *Mathematica* will actually use a much smaller amount of memory to store the expression. The main issue is how many of the subexpressions in the expression can be *shared*.

In an expression like \( f[1 + x, 1 + x] \), the two subexpressions \( 1 + x \) are identical, but they may or may not actually be stored in the same piece of computer memory. `ByteCount` gives you the number of bytes needed to store expressions with the assumption that no subexpressions are shared. You should realize that the sharing of subexpressions is often destroyed as soon as you use an operation like the `/.` operator.

Nevertheless, you can explicitly tell *Mathematica* to share subexpressions using the function `Share`. In this way, you can significantly reduce the actual amount of memory needed to store a particular expression.

### Table: Functions for Finding the Size of Expressions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ByteCount[expr]</code></td>
<td>the maximum number of bytes of memory needed to store <code>expr</code></td>
</tr>
<tr>
<td><code>LeafCount[expr]</code></td>
<td>the number of terminal nodes in the expression tree for <code>expr</code></td>
</tr>
</tbody>
</table>

Optimizing memory usage.

On most computer systems, the memory used by a running program is divided into two parts: memory explicitly allocated by the program, and "stack space". Every time an internal routine is called in the program, a certain amount of stack space is used to store parameters associated with the call. On many computer systems, the maximum amount of stack space that can be used by a program must be specified in advance. If the specified stack space limit is exceeded, the program usually just exits.
In *Mathematica*, one of the primary uses of stack space is in handling the calling of one *Mathematica* function by another. All such calls are explicitly recorded in the *Mathematica* Stack discussed in "The Evaluation Stack". You can control the size of this stack by setting the global parameter $\text{\$RecursionLimit}$. You should be sure that this parameter is set small enough that you do not run out of stack space on your particular computer system.

## Global System Information

In order to write the most general *Mathematica* programs you will sometimes need to find out global information about the setup under which your program is being run.

Thus, for example, to tell whether your program should be calling functions like NotebookWrite, you need to find out whether the program is being run in a *Mathematica* session that is using the notebook front end. You can do this by testing the global variable $\text{\$Notebooks}$.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{$Notebooks}$</td>
<td>whether a notebook front end is being used</td>
</tr>
</tbody>
</table>

Determining whether a notebook front end is being used.

*Mathematica* is usually used interactively, but it can also operate in a batch mode—say taking input from a file and writing output to a file. In such a case, a program cannot for example expect to get interactive input from the user.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{$BatchInput}$</td>
<td>whether input is being given in batch mode</td>
</tr>
<tr>
<td>$\text{$BatchOutput}$</td>
<td>whether output should be given in batch mode, without labeling, etc.</td>
</tr>
</tbody>
</table>

Variables specifying batch mode operation.

The *Mathematica* kernel is a process that runs under the operating system on your computer. Within *Mathematica* there are several global variables that allow you to find the characteristics of this process and its environment.
$CommandLine the original command line used to invoke the Mathematica kernel
$ParentLink the MathLink LinkObject specifying the program that invoked the kernel (or Null if the kernel was invoked directly)
$ProcessID the ID assigned to the Mathematica kernel process by the operating system
$ParentProcessID the ID of the process that invoked the Mathematica kernel
$UserName the login name of the user running the Mathematica kernel
Environment["var"] the value of a variable defined by the operating system

Variables associated with the Mathematica kernel process.

If you have a variable such as $x$ in a particular Mathematica session, you may or may not want that variable to be the same as an $x$ in another Mathematica session. In order to make it possible to maintain distinct objects in different sessions, Mathematica supports the variable $SessionID$, which uses information such as starting time, process ID and machine ID to try to give a different value for every single Mathematica session, whether it is run on the same computer or a different one.

$SessionID a number set up to be different for every Mathematica session

A unique number different for every Mathematica session.

Mathematica provides various global variables that allow you to tell which version of the kernel you are running. This is important if you write programs that make use of features that are, say, new in Version 6. You can then check $VersionNumber to find out if these features will be available.

$Version a string giving the complete version of Mathematica in use
$VersionNumber the Mathematica kernel version number (e.g. 6.0)
$ReleaseNumber the release number for your version of the Mathematica kernel on your particular computer system
$CreationDate the date, in DateList format, on which your particular Mathematica release was created

Variables specifying the version of Mathematica used.
Mathematica itself is set up to be as independent of the details of the particular computer system on which it is run as possible. However, if you want to access external aspects of your computer system, then you will often need to find out its characteristics.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$System</td>
<td>a full string describing the computer system in use</td>
</tr>
<tr>
<td>$SystemID</td>
<td>a short string specifying the computer system in use</td>
</tr>
<tr>
<td>$ProcessorType</td>
<td>the architecture of the processor in your computer system</td>
</tr>
<tr>
<td>$MachineType</td>
<td>the general type of your computer system</td>
</tr>
<tr>
<td>$ByteOrdering</td>
<td>the native byte ordering convention on your computer system</td>
</tr>
<tr>
<td>$OperatingSystem</td>
<td>the basic operating system in use</td>
</tr>
<tr>
<td>$SystemCharacterEncoding</td>
<td>the default raw character encoding used by your operating system</td>
</tr>
</tbody>
</table>

Variables specifying the characteristics of your computer system.

Mathematica uses the values of $SystemID to label directories that contain versions of files for different computer systems, as discussed in "Reading and Writing Mathematica Files: Files and Streams" and "Portability of MathLink Programs". Computer systems for which $SystemID is the same will normally be binary compatible.

$OperatingSystem has values such as "Windows" or "Unix". By testing $OperatingSystem you can determine whether a particular external program is likely to be available on your computer system.

This gives some characteristics of the computer system on which the input is evaluated.

```
In[1]:= {$System, $ProcessorType, $OperatingSystem}
Out[1]= {Linux x86 (32-bit), x86, Unix}
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MachineAddresses</td>
<td>the list of current IP addresses</td>
</tr>
<tr>
<td>$MachineName</td>
<td>the name of the computer on which Mathematica is running</td>
</tr>
<tr>
<td>$MachineDomains</td>
<td>the current network domains for the computer</td>
</tr>
<tr>
<td>$MachineID</td>
<td>the unique ID assigned by Mathematica to the computer</td>
</tr>
</tbody>
</table>

Variables identifying the computer on which Mathematica is running.
$LicenseID  
the ID for the license under which Mathematica is running

$LicenseExpirationDate  
the date on which the license expires

$NetworkLicense  
whether this is a network license

$LicenseServer  
the full name of the machine serving the license

$LicenseProcesses  
the number of Mathematica processes currently being run under the license

$MaxLicenseProcesses  
the maximum number of processes provided by the license

$PasswordFile  
password file used when the kernel was started

Variables associated with license management.

**Mathematica Sessions**

**Command-Line Options and Environment Variables**

- `-pwfile`  
  Mathematica password file
- `-pwpath`  
  path to search for a Mathematica password file
- `-run`  
  Mathematica input to run (kernel only)
- `-initfile`  
  Mathematica initialization file
- `-initpath`  
  path to search for initialization files
- `-noinit`  
  do not run initialization files
- `-mathlink`  
  communicate only via MathLink

Typical command-line options for Mathematica executables.

If the Mathematica front end is called with a notebook file as a command-line argument, then this notebook will be made the initial selected notebook. Otherwise, a new notebook will be created for this purpose.

Mathematica kernels and front ends can also take additional command-line options specific to particular window environments.
Mathematica will read the values of operating system environment variables, and will use these values in addition to any command-line options explicitly given.

**Initialization**

On startup, the Mathematica kernel does the following:

- Performs license management operations.
- Runs Mathematica commands specified in any `-run` options passed to the kernel executable.
- Runs the Mathematica commands in the systemwide initialization file `$BaseDirectory/Kernel/init.m`.
- Runs the Mathematica commands in the user-specific initialization file `$UserBaseDirectory/Kernel/init.m`.
- Loads `init.m` and `Kernel/init.m` files in Autoload directories.
- Begins running the main loop.

**The Main Loop**

All Mathematica sessions repeatedly execute the following main loop:

- Read in input.
- Apply `$PreRead` function, if defined, to the input string.
- Print syntax warnings if necessary.
Apply $SyntaxHandler function if there is a syntax error.

Assign InString[$n].

Assign In[$n].

Evaluate expression.

Apply $Pre function, if defined, to the input expression.

Assign In[$n].

Evaluate expression.

Apply $Post function, if defined.

Assign Out[$n], stripping off any formatting wrappers.

Apply $PrePrint function, if defined.

Assign MessageList[$n] and clear $MessageList.

Print expression, if it is not Null.

Increment $Line.

Clear any pending aborts.

Note that if you call Mathematica via MathLink from within an external program, then you must effectively create your own main loop, which will usually differ from the one described above.

**Messages**

During a Mathematica session messages can be generated either by explicit calls to Message, or in the course of executing other built-in functions.

<table>
<thead>
<tr>
<th>f::name::lang</th>
<th>a message in a specific language</th>
</tr>
</thead>
<tbody>
<tr>
<td>f::name</td>
<td>a message in a default language</td>
</tr>
<tr>
<td>General::name</td>
<td>a general message with a given name</td>
</tr>
</tbody>
</table>

Message names.
If no language is specified for a particular message, text for the message is sought in each of the languages specified by $Language. If $f::name is not defined, a definition for General::name is sought. If still no message is found, any value defined for $NewMessage is applied to $f and "$name".

Quiet[expr] evaluates expr while preventing messages from being printed during the evaluation. Off[message] prevents a specified message from ever being printed. Check allows you to determine whether particular messages were generated during the evaluation of an expression. $MessageList and MessageList[n] record all the messages that were generated during the evaluation of a particular line in a Mathematica session.

Messages are specified as strings to be used as the first argument of StringForm. $MessagePrePrint is applied to each expression to be spliced into the string.

---

**Termination**

<table>
<thead>
<tr>
<th>Exit[] or Quit[]</th>
<th>terminate Mathematica</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Epilog</td>
<td>symbol to evaluate before Mathematica exits</td>
</tr>
<tr>
<td>$IgnoreEOF</td>
<td>whether to exit an interactive Mathematica session when an end-of-file character is received</td>
</tr>
<tr>
<td>end.m</td>
<td>file to read when Mathematica terminates</td>
</tr>
</tbody>
</table>

**Mathematica termination.**

There are several ways to end a Mathematica session. If you are using Mathematica interactively, typing Exit[] or Quit[] on an input line will always terminate Mathematica.

If you are taking input for Mathematica from a file, Mathematica will exit when it reaches the end of the file. If you are using Mathematica interactively, it will still exit if it receives an end-of-file character (typically Ctrl+d). You can stop Mathematica from doing this by setting $IgnoreEOF = True.
Network License Management

<table>
<thead>
<tr>
<th>single-machine license</th>
<th>a process must always run on a specific machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>network license</td>
<td>a process can run on any machine on a network</td>
</tr>
</tbody>
</table>

Single-machine and network licenses.

Copies of *Mathematica* can be set up with either single-machine or network licenses. A network license is indicated by a line in the `mathpass` file starting with `!name`, where `name` is the name of the server machine for the network license.

Network licenses are controlled by the *Mathematica* license management program `mathlm`, which is run on the server machine. This program must be running whenever a *Mathematica* with a network license is being used. Typically you will want to set up your system so that `mathlm` is started whenever the system boots.

- Type `. \mathlm` directly on the command line
- Add `mathlm` as a Windows service

Ways to start the network license manager under Microsoft Windows.

- Type `. /mathlm` directly on the Unix command line
- Add a line to start `mathlm` in your central system startup script

Ways to start the network license manager on Macintosh and Unix systems.

When `mathlm` is not started directly from a command line, it normally sets itself up as a background process, and continues running until it is explicitly terminated. Note that if one `mathlm` process is running, any other `mathlm` processes you try to start will automatically exit immediately.
Some command-line options for `mathlm`.

For more detailed information on `mathlm`, see "System Administration for Network Licenses".

### -logfile `file`
write a log of license server actions to `file`

### -loglevel `n`
how verbose to make log entries (1 to 4)

### -logformat `string`
use a log format specified by `string`

### -language `name`
language to use for messages (default English)

### -pwfile `file`
use the specified `mathpass` file (default ./.mathpass)

### -timeout `n`
suspend authorization on stopped *Mathematica* jobs after `n` hours

### -restrict `file`
use the specified restriction file

### -mathid
print the MathID for the license server, and exit

### -foreground
run `mathlm` in the foreground, logging to stdout

### -install
install `mathlm` as a Windows service (Microsoft Windows only)

### -uninstall
uninstall `mathlm` as a Windows service (Microsoft Windows only)

Some command-line options for `monitorlm`.

### `monitorlm`
a program to monitor network license activity

### `monitorlm name`
monitor activity for license server `name`

Monitoring network license activity.

If `monitorlm` is run in an environment where a web browser can be started, it will automatically generate HTML output in the browser. Otherwise it will generate plain text.

### -file `file`
write output to a file

### -format `spec`
use the specified format (text, html or cgi)

### -template `file`
use the specified file as a template for the output

Some command-line options for `monitorlm`. 
The Internals of Mathematica

Why You Do Not Usually Need to Know about Internals

Most of the documentation provided for Mathematica is concerned with explaining what Mathematica does, not how it does it. But the purpose of this is to say at least a little about how Mathematica does what it does. "Some Notes on Internal Implementation" gives more details.

You should realize at the outset that while knowing about the internals of Mathematica may be of intellectual interest, it is usually much less important in practice than you might at first suppose.

Indeed, one of the main points of Mathematica is that it provides an environment where you can perform mathematical and other operations without having to think in detail about how these operations are actually carried out inside your computer.

Thus, for example, if you want to factor the polynomial \( x^{15} - 1 \), you can do this just by giving Mathematica the command `Factor[x^15 - 1]`; you do not have to know the fairly complicated details of how such a factorization is actually carried out by the internal code of Mathematica.

Indeed, in almost all practical uses of Mathematica, issues about how Mathematica works inside turn out to be largely irrelevant. For most purposes it suffices to view Mathematica simply as an abstract system which performs certain specified mathematical and other operations.

You might think that knowing how Mathematica works inside would be necessary in determining what answers it will give. But this is only very rarely the case. For the vast majority of the computations that Mathematica does are completely specified by the definitions of mathematical or other operations.

Thus, for example, \( 3^{40} \) will always be 12 157 665 459 056 928 801, regardless of how Mathematica internally computes this result.
There are some situations, however, where several different answers are all equally consistent
with the formal mathematical definitions. Thus, for example, in computing symbolic integrals,
there are often several different expressions which all yield the same derivative. Which of these
expressions is actually generated by `Integrate` can then depend on how `Integrate` works
inside.

Here is the answer generated by `Integrate`.

```
In[1]:= Integrate[1 / x + 1 / x^2, x]
Out[1]= -x + Log[x]
```

This is an equivalent expression that might have been generated if `Integrate` worked differ-
ently inside.

```
In[2]:= Together[Out[1]]
Out[2]= -1 + x Log[x]
```

In numerical computations, a similar phenomenon occurs. Thus, for example, `FindRoot` gives
you a root of a function. But if there are several roots, which root is actually returned depends
on the details of how `FindRoot` works inside.

This finds a particular root of $\cos(x) + \sin(x)$.

```
In[3]:= FindRoot[Cos[x] + Sin[x], {x, 10.5}]
Out[3]= {x -> 14.9226}
```

With a different starting point, a different root is found. Which root is found with each starting
point depends in detail on the internal algorithm used.

```
In[4]:= FindRoot[Cos[x] + Sin[x], {x, 10.8}]
```

The dependence on the details of internal algorithms can be more significant if you push approxi-
mate numerical computations to the limits of their validity.
Thus, for example, if you give \texttt{NIntegrate} a pathological integrand, whether it yields a meaningful answer or not can depend on the details of the internal algorithm that it uses.

\texttt{NIntegrate} knows that this result is unreliable, and can depend on the details of the internal algorithm, so it prints warning messages.

\texttt{In[5]:= \texttt{NIntegrate[Sin[1 / x], \{x, 0, 1\}]}}

\texttt{NIntegrate::slwcon: Numerical integration converging too slowly; suspect one of the following: singularity, value of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. \texttt{\$\_}}

\texttt{NIntegrate::ncvb: NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near \{x\} = \{0.0053386\}. NIntegrate obtained 0.5038782627066661` and 0.0011134563535424439` for the integral and error estimates. \texttt{\$\_}}

\texttt{Out[5]= 0.503878}

Traditional numerical computation systems have tended to follow the idea that all computations should yield results that at least nominally have the same precision. A consequence of this idea is that it is not sufficient just to look at a result to know whether it is accurate; you typically also have to analyze the internal algorithm by which the result was found. This fact has tended to make people believe that it is always important to know internal algorithms for numerical computations.

But with the approach that \textit{Mathematica} takes, this is rarely the case. For \textit{Mathematica} can usually use its arbitrary-precision numerical computation capabilities to give results where every digit that is generated follows the exact mathematical specification of the operation being performed.

Even though this is an approximate numerical computation, every digit is determined by the mathematical definition for $\pi$.

\texttt{In[6]:= \texttt{N[Pi, 30]}}

\texttt{Out[6]= 3.14159265358979323846264338328}

Once again, every digit here is determined by the mathematical definition for $\sin(x)$.

\texttt{In[7]:= \texttt{N[Sin[10 ^ 50], 20]}}

\texttt{Out[7]= -0.78967249342931008271}
If you use machine-precision numbers, Mathematica cannot give a reliable result, and the answer depends on the details of the internal algorithm used.

\[\text{In}[8]:= \text{Sin}[10.^{50}]\]
\[\text{Out}[8]= 0.669369\]

It is a general characteristic that whenever the results you get can be affected by the details of internal algorithms, you should not depend on these results. For if nothing else, different versions of Mathematica may exhibit differences in these results, either because the algorithms operate slightly differently on different computer systems, or because fundamentally different algorithms are used in versions released at different times.

This is the result for \(\sin(10^{50})\) on one type of computer.

\[\text{In}[9]:= \text{Sin}[10.^{50}]\]
\[\text{Out}[9]= 0.669369\]

Here is the same calculation on another type of computer.

\[\text{In}[10]:= \text{Sin}[10.^{50}]\]
\[\text{Out}[10]= 0.669369\]

And here is the result obtained in Mathematica Version 1.

\[\text{In}[11]:= \text{Sin}[10.^{50}]\]
\[\text{Out}[11]= 0.669369\]

Particularly in more advanced applications of Mathematica, it may sometimes seem worthwhile to try to analyze internal algorithms in order to predict which way of doing a given computation will be the most efficient. And there are indeed occasionally major improvements that you will be able to make in specific computations as a result of such analyses.

But most often the analyses will not be worthwhile. For the internals of Mathematica are quite complicated, and even given a basic description of the algorithm used for a particular purpose, it is usually extremely difficult to reach a reliable conclusion about how the detailed implementation of this algorithm will actually behave in particular circumstances.

A typical problem is that Mathematica has many internal optimizations, and the efficiency of a computation can be greatly affected by whether the details of the computation do or do not allow a given internal optimization to be used.
Basic Internal Architecture

<table>
<thead>
<tr>
<th>numbers</th>
<th>sequences of binary digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>strings</td>
<td>sequences of character code bytes or byte pairs</td>
</tr>
<tr>
<td>symbols</td>
<td>pointers to the central table of symbols</td>
</tr>
<tr>
<td>general expressions</td>
<td>sequences of pointers to the head and elements</td>
</tr>
</tbody>
</table>

**Internal representations used by Mathematica.**

When you type input into *Mathematica*, a data structure is created in the memory of your computer to represent the expression you have entered.

In general, different pieces of your expression will be stored at different places in memory. Thus, for example, for a list such as \( \{2, x, y + z\} \) the “backbone” of the list will be stored at one place, while each of the actual elements will be stored at a different place.

The backbone of the list then consists just of three “pointers” that specify the addresses in computer memory at which the actual expressions that form the elements of the list are to be found. These expressions then in turn contain pointers to their subexpressions. The chain of pointers ends when one reaches an object such as a number or a string, which is stored directly as a pattern of bits in computer memory.

Crucial to the operation of *Mathematica* is the notion of symbols such as \( x \). Whenever \( x \) appears in an expression, *Mathematica* represents it by a pointer. But the pointer is always to the same place in computer memory—an entry in a central table of all symbols defined in your *Mathematica* session.

This table is a repository of all information about each symbol. It contains a pointer to a string giving the symbol’s name, as well as pointers to expressions which give rules for evaluating the symbol.

- Recycle memory as soon as the data in it is no longer referenced.

The basic principle of *Mathematica* memory management.

Every piece of memory used by *Mathematica* maintains a count of how many pointers currently point to it. When this count drops to zero, *Mathematica* knows that the piece of memory is no longer being referenced, and immediately makes the piece of memory available for something new.
This strategy essentially ensures that no memory is ever wasted, and that any piece of memory that *Mathematica* uses is actually storing data that you need to access in your *Mathematica* session.

- Create an expression corresponding to the input you have given.
- Process the expression using all rules known for the objects in it.
- Generate output corresponding to the resulting expression.

The basic actions of *Mathematica*.

At the heart of *Mathematica* is a conceptually simple procedure known as the *evaluator* which takes every function that appears in an expression and evaluates that function.

When the function is one of the thousand or so that are built into *Mathematica*, what the evaluator does is to execute directly internal code in the *Mathematica* system. This code is set up to perform the operations corresponding to the function, and then to build a new expression representing the result.

- The built-in functions of *Mathematica* support universal computation.

The basic feature that makes *Mathematica* a self-contained system.

A crucial feature of the built-in functions in *Mathematica* is that they support *universal computation*. What this means is that out of these functions you can construct programs that perform absolutely any kinds of operations that are possible for a computer.

As it turns out, small subsets of *Mathematica*’s built-in functions would be quite sufficient to support universal computation. But having the whole collection of functions makes it in practice easier to construct the programs one needs.

The underlying point, however, is that because *Mathematica* supports universal computation you never have to modify its built-in functions: all you have to do to perform a particular task is to combine these functions in an appropriate way.

Universal computation is the basis for all standard computer languages. But many of these languages rely on the idea of *compilation*. If you use C or Fortran, for example, you first write your program, then you compile it to generate machine code that can actually be executed on your computer.

*Mathematica* does not require you to go through the compilation step: once you have input an expression, the functions in the expression can immediately be executed.
Often *Mathematica* will preprocess expressions that you enter, arranging things so that subsequent execution will be as efficient as possible. But such preprocessing never affects the results that are generated, and can rarely be seen explicitly.

**The Algorithms of *Mathematica***

The built-in functions of *Mathematica* implement a very large number of algorithms from computer science and mathematics. Some of these algorithms are fairly old, but the vast majority had to be created or at least modified specifically for *Mathematica*. Most of the more mathematical algorithms in *Mathematica* ultimately carry out operations which at least at some time in the past were performed by hand. In almost all cases, however, the algorithms use methods very different from those common in hand calculation.

Symbolic integration provides an example. In hand calculation, symbolic integration is typically done by a large number of tricks involving changes of variables and the like.

But in *Mathematica* symbolic integration is performed by a fairly small number of very systematic procedures. For indefinite integration, the idea of these procedures is to find the most general form of the integral, then to differentiate this and try to match up undetermined coefficients.

Often this procedure produces at an intermediate stage immensely complicated algebraic expressions, and sometimes very sophisticated kinds of mathematical functions. But the great advantage of the procedure is that it is completely systematic, and its operation requires no special cleverness of the kind that only a human could be expected to provide.

In having *Mathematica* do integrals, therefore, one can be confident that it will systematically get results, but one cannot expect that the way these results are derived will have much at all to do with the way they would be derived by hand.

The same is true with most of the mathematical algorithms in *Mathematica*. One striking feature is that even for operations that are simple to describe, the systematic algorithms to perform these operations in *Mathematica* involve fairly advanced mathematical or computational ideas.
Thus, for example, factoring a polynomial in \( x \) is first done modulo a prime such as 17 by finding the null space of a matrix obtained by reducing high powers of \( x \) modulo the prime and the original polynomial. Then factorization over the integers is achieved by “lifting” modulo successive powers of the prime using a collection of intricate theorems in algebra and analysis.

The use of powerful systematic algorithms is important in making the built-in functions in \emph{Mathematica} able to handle difficult and general cases. But for easy cases that may be fairly common in practice it is often possible to use simpler and more efficient algorithms.

As a result, built-in functions in \emph{Mathematica} often have large numbers of extra pieces that handle various kinds of special cases. These extra pieces can contribute greatly to the complexity of the internal code, often taking what would otherwise be a five-page algorithm and making it hundreds of pages long.

Most of the algorithms in \emph{Mathematica}, including all their special cases, were explicitly constructed by hand. But some algorithms were instead effectively created automatically by computer.

Many of the algorithms used for machine-precision numerical evaluation of mathematical functions are examples. The main parts of such algorithms are formulas which are as short as possible but which yield the best numerical approximations.

Most such formulas used in \emph{Mathematica} were actually derived by \emph{Mathematica} itself. Often many months of computation were required, but the result was a short formula that can be used to evaluate functions in an optimal way.

\section*{The Software Engineering of \emph{Mathematica}}

\emph{Mathematica} is one of the more complex software systems ever constructed. It is built from several million lines of source code, written in C/C++, Java and \emph{Mathematica}.

The C code in \emph{Mathematica} is actually written in a custom extension of C which supports certain memory management and object-oriented features. The \emph{Mathematica} code is optimized using \texttt{Share} and \texttt{DumpSave}.

In the \emph{Mathematica} kernel the breakdown of different parts of the code is roughly as follows: language and system: 30%; numerical computation: 20%; algebraic computation: 20%; graphics and kernel output: 30%.
Most of this code is fairly dense and algorithmic: those parts that are in effect simple procedures or tables use minimal code since they tend to be written at a higher level—often directly in Mathematica.

The source code for the kernel, save a fraction of a percent, is identical for all computer systems on which Mathematica runs.

For the front end, however, a significant amount of specialized code is needed to support each different type of user interface environment. The front end contains about 700,000 lines of system-independent C++ source code, of which roughly 200,000 lines are concerned with expression formatting. Then there are between 50,000 and 100,000 lines of specific code customized for each user interface environment.

Mathematica uses a client-server model of computing. The front end and kernel are connected via MathLink—the same system as is used to communicate with other programs. MathLink supports multiple transport layers, including one based upon TCP/IP and one using shared memory.

The front end and kernel are connected via three independent MathLink connections. One is used for user-initiated evaluations. A second is used by the front end to resolve the values of Dynamic expressions. The third is used by the kernel to notify the front end of Dynamic objects which should be invalidated.

Within the C code portion of the Mathematica kernel, modularity and consistency are achieved by having different parts communicate primarily by exchanging complete Mathematica expressions.

But it should be noted that even though different parts of the system are quite independent at the level of source code, they have many algorithmic interdependencies. Thus, for example, it is common for numerical functions to make extensive use of algebraic algorithms, or for graphics code to use fairly advanced mathematical algorithms embodied in quite different Mathematica functions.

Since the beginning of its development in 1986, the effort spent directly on creating the source code for Mathematica is about a thousand developer-years. In addition, a comparable or somewhat larger effort has been spent on testing and verification.
The source code of *Mathematica* has changed greatly since Version 1 was released. The total number of lines of code in the kernel grew from 150,000 in Version 1 to 350,000 in Version 2, 600,000 in Version 3, 800,000 in Version 4, 1.5 million in Version 5 and 2.5 million in Version 6. In addition, at every stage existing code has been revised—so that Version 6 has only a few percent of its code in common with Version 1.

Despite these changes in internal code, however, the user-level design of *Mathematica* has remained compatible from Version 1 on. Much functionality has been added, but programs created for *Mathematica* Version 1 will almost always run absolutely unchanged under Version 6.

## Testing and Verification

Every version of *Mathematica* is subjected to a large amount of testing before it is released. The vast majority of this testing is done by an automated system that is written in *Mathematica*.

The automated system feeds millions of pieces of input to *Mathematica*, and checks that the output obtained from them is correct. Often there is some subtlety in doing such checking: one must account for different behavior of randomized algorithms and for such issues as differences in machine-precision arithmetic on different computers.

The test inputs used by the automated system are obtained in several ways:

- For every *Mathematica* function, inputs are devised that exercise both common and extreme cases.
- Inputs are devised to exercise each feature of the internal code.
- All the examples in *Mathematica*’s documentation system are used, as well as those from many books about *Mathematica*.
- Tests are constructed from mathematical benchmarks and test sets from numerous websites.
- Standard numerical tables are optically scanned for test inputs.
- Formulas from standard mathematical tables are entered.
- Exercises from textbooks are entered.
- For pairs of functions such as \texttt{Integrate} and \texttt{D} or \texttt{Factor} and \texttt{Expand}, random expressions are generated and tested.

When tests are run, the automated testing system checks not only the results, but also side effects such as messages, as well as memory usage and speed.
There is also a special instrumented version of *Mathematica* which is set up to perform internal consistency tests. This version of *Mathematica* runs at a small fraction of the speed of the real *Mathematica*, but at every step it checks internal memory consistency, interruptibility, and so on.

The instrumented version of *Mathematica* also records which pieces of *Mathematica* source code have been accessed, allowing one to confirm that all of the various internal functions in *Mathematica* have been exercised by the tests given.

All standard *Mathematica* tests are routinely run on current versions of *Mathematica*, on each different computer system. Depending on the speed of the computer system, these tests take from a few hours to a few days of computer time.

Even with all this testing, however, it is inevitable in a system as complex as *Mathematica* that errors will remain.

The standards of correctness for *Mathematica* are certainly much higher than for typical mathematical proofs. But just as long proofs will inevitably contain errors that go undetected for many years, so also a complex software system such as *Mathematica* will contain errors that go undetected even after millions of people have used it.

Nevertheless, particularly after all the testing that has been done on it, the probability that you will actually discover an error in *Mathematica* in the course of your work is extremely low.

Doubtless there will be times when *Mathematica* does things you do not expect. But you should realize that the probabilities are such that it is vastly more likely that there is something wrong with your input to *Mathematica* or your understanding of what is happening than with the internal code of the *Mathematica* system itself.

If you do believe that you have found a genuine error in *Mathematica*, then you should contact Wolfram Research Technical Support, so that the error can be corrected in future versions.
Security and Connectivity

Mathematica Internet Connectivity

Introduction

Mathematica provides important functionality through accessing the internet. Most Mathematica functions that provide computable data operate by loading data over the internet. Some functions require real-time access to the internet; others update a local data repository by accessing the internet when required. Mathematica also requires internet access when you explicitly use Import to read from a URL, or when you use web services. The Mathematica documentation system also supports automatic updating via the internet.

When you call a data function like FinancialData, Mathematica gets the data it needs from the internet. When you click a link to a documentation notebook or call a data function like CountryData, Mathematica knows whether a newer version of the information is available on a Wolfram Research Paclet Server, and if so it will download and install the update automatically. In the case of smaller paclets like documentation notebooks, this is often so fast that you will not even notice it happening.

Mathematica acts like a web browser when it accesses the internet, so if you can browse the web from your computer, you should be able to use Mathematica's internet connectivity features, although in some cases additional configuration may be required.

Internet Connectivity Dialog

The Internet Connectivity dialog, accessed from the Help menu, allows you to configure a number of settings related to the paclet system, and Mathematica's use of the internet in general.

The Allow Mathematica to access the Internet checkbox can be turned off to prevent Mathematica from even attempting to use the internet. You will not be able to get load-on-demand updates to documentation, and some data collection functions will not operate.
The **Test Internet Connectivity** button is useful to see if *Mathematica* is properly configured for internet use. After clicking the button, you should see a dialog within a few seconds (perhaps slightly longer if it fails) reporting success or failure. If the test succeeds, then *Mathematica*’s internet functionality should work properly. If it fails, consult "Troubleshooting Connectivity Problems".

The **Proxy Settings** section allows you to specify a proxy server if necessary. In many cases, *Mathematica* is able to inherit the proxy settings configured globally for your system or browser. This is the default setting, and most users will leave it as is. If you know that you do not need to go through a proxy server to access the internet, you can click the **Direct connection to the Internet** checkbox. You can also manually configure proxy settings if necessary. Contact your system administrator for the values to use. Most users will only need to set the HTTP proxy.

The **Automatically check for documentation updates** checkbox and the **Automatically check for data updates** checkbox can be turned off to disable load-on-demand updates to documentation and data files. This will not interfere with the operation of *Mathematica*, except that you will not receive updates to the documentation or data paclets as they become available.

The **Update Local Indices from the Wolfram Research Server** button will cause *Mathematica* to read information from the Wolfram Research Paclet Server that describes the versions of the paclets that are available. *Mathematica* uses this information to decide whether an update is available to a given resource when you access that resource. *Mathematica* reads this information on a weekly basis automatically, but you can force an update using this button. You might want to do this to be sure you will get the absolute latest data from data collection functions like `CountryData`, `ChemicalData`, `AstronomicalData`, etc.

---

**Troubleshooting Connectivity Problems**

If you get error messages or dialogs that report internet connectivity problems while running *Mathematica*, the first thing to do is try the **Test Internet Connectivity** button in the **Internet Connectivity** dialog, accessible from the **Help** menu. If the test succeeds, then *Mathematica* is correctly configured for general internet use, and the problem probably lies elsewhere (such as trying to access an incorrect URL). If the test fails, try the following steps:
1. Test network connectivity by seeing if other programs on your computer can access the internet. For example, launch a web browser and see if it works. If it fails, then the network might be unavailable, or you might have connectivity problems beyond what can be configured in *Mathematica*.

2. Check proxy settings, as in "Proxy Settings".

3. Check firewall settings, as in "Firewall Settings".

**Proxy Settings**

Incorrect proxy settings are a common cause of problems with internet connectivity. Many users on a company network cannot directly access the internet, but instead must pass through a proxy server, which acts like a gateway to the internet. By default, *Mathematica* will attempt to use systemwide proxy settings, if your operating system has such settings. For example, on Windows, *Mathematica* will use the proxy settings configured for Internet Explorer. On Mac OS X, proxy settings are configured in the **Network Preferences** panel.

*Mathematica*'s proxy settings are configured in the **Internet Connectivity** dialog. The default setting, described in "Internet Connectivity Dialog", is **Use proxy settings from my system or browser**. If this does not work for you, try the **Direct connection to the Internet** choice. If that does not work, then contact your system administrator for proxy settings to enter into the text fields. If you can successfully surf the web with a web browser, you can find its proxy settings dialog and read the values it is using. Many users will only need to set the HTTP proxy.

If your system or browser is configured to get proxy settings from a configuration script, then *Mathematica* will not be able to use these settings, and you will have to manually configure its proxy settings.

If *Mathematica* is configured to **Use proxy settings from your system or browser**, and your browser functions but *Mathematica* cannot connect, see if your system proxy settings have a **Use same proxy server for all protocols** checkbox and try unchecking it. *Mathematica* will attempt to use a SOCKS proxy if one is set, and if your HTTP proxy does not also handle SOCKS traffic, **Use same proxy server for all protocols** is not a correct setting for your system. An incorrectly configured SOCKS proxy can cause very long timeouts, so if the **Test Network Connectivity** button fails after a minutes-long delay, an incorrect SOCKS proxy configuration is likely the problem. *Mathematica* does not require a SOCKS proxy, so the SOCKS **Host** field can be left blank, but if you supply a value, either manually or via the system setting **Use the same proxy server for all protocols**, it must be correct.
Firewall Settings

Because Mathematica acts like a web browser when it accesses the internet, most company firewalls will not interfere with it. Some users, however, have so-called "personal" firewalls on their machines (ZoneAlarm, Norton, etc., or the one built into Microsoft Windows). If configured with strict settings, these firewalls might interfere with Mathematica's attempts to use the internet.

These types of firewalls will generally display a dialog box warning you that a program is trying to use the internet and allow you to accept or reject it. If you see such a dialog, it might report that the program is the Mathematica kernel or the Java Runtime Environment that is bundled in the Mathematica layout. Configure the firewall to always allow such requests.

Further Information

If the information in this document is not sufficient to help you solve connectivity problems, consult the Wolfram Research Technical Support troubleshooting guide at http://support.wolfram.com/technotes/networkconnectivityissues.html

Notebook Security

Mathematica provides users with access to their computer's file system (Files), interprocess communication (MathLink Mathematica Functions), evaluation of data as code (Converting between Expressions and Strings), and the ability to run arbitrary external programs (Calling External Programs). While these features enable Mathematica users to create powerful programs that can perform truly useful tasks, they bring with them the potential for misuse.

The Mathematica notebook front end provides three mechanisms for evaluating code: Shift + Return evaluations, initialization cells, and dynamic content.

Note that this tutorial contains live controls, so if you change anything it will immediately change the settings on your system.
**Shift +Return Evaluations**

Because Shift +Return evaluations require user interaction to start them, *Mathematica* provides no safeguards against potentially malicious code that is evaluated using this mechanism. Users should ensure that they do not perform Shift +Return evaluations on code from untrusted sources. When writing their own code, users should take great care to ensure that the code does not have unintended consequences. For example, *Mathematica* will not provide a warning when the user evaluates a program to delete files from his or her computer.

**Initialization Cells**

Initialization cells provide users with a convenient way to evaluate startup code needed by a given notebook when the user first evaluates any input in that notebook. Since this code will be automatically evaluated, likely without the user ever seeing the initialization code, *Mathematica* will display an alert prompt asking the user to confirm his or her intent to run the initialization code. Users should not evaluate initialization code in a notebook that was obtained from an untrusted source unless the code has been determined to be safe.

- Clicking the **Yes** button will first evaluate all of the initialization cells in the notebook, then evaluate the selected cells.
- Clicking the **No** button will not evaluate any of the initialization cells, but will still evaluate the selected cells. Note that this may cause errors in the evaluations since they may rely on startup code that has not yet been evaluated.
- Clicking the **Cancel** button will cause neither the initialization cells nor the selected cells to be evaluated.
**Dynamic Content**

*Mathematica* 7 has amazing dynamic interactivity features. Notebooks containing interactive content can automatically evaluate code without any additional action by the user, aside from opening the notebook file. Many times this is exactly what the user will want, while other times the user should be alerted before this sort of automatic evaluation takes place.

When the user opens a notebook containing dynamic content, *Mathematica* will first determine whether the notebook's directory is trusted, untrusted, or neither.

- If the notebook's directory is trusted, the notebook will be allowed to automatically perform dynamic evaluations without alerting the user.
- If the notebook's directory is untrusted, the user will be alerted upon any attempt by the notebook to perform dynamic evaluations.

![Dynamic Content Warning]

- If the notebook's directory is neither trusted nor untrusted, the behavior will depend on the value of the `TrustByDefault` option explained as follows.

*Mathematica* provides some options which can be used to configure which notebooks will alert the user about automatic evaluations and which notebooks will not.

**TrustedPath**

The value of the `TrustedPath` option is a list of directories that are always trusted by *Mathematica*. Any notebook file located in any directory in `TrustedPath` is trusted by *Mathematica*. *Mathematica* will never display an alert when a trusted notebook is opened, and the notebook can automatically perform dynamic evaluations.
By default, the TrustedPath option value contains $InstallationDirectory, $BaseDirectory, and $UserBaseDirectory so that Mathematica installation files and additional installed applications will be able to display dynamic content without alerting the user.

Here are the directories on your computer that are currently trusted by Mathematica:

```
Dynamic[{Column[{ToFileName[CurrentValue[$FrontEnd, {"NotebookSecurityOptions", "TrustedPath"}]}],
/Applications/Mathematica.app/
/Library/Mathematica/
/home/usr2/larrya/Library/Mathematica/}
```

Edit TrustedPath ...

**UntrustedPath**

The value of the UntrustedPath option is a list of directories that are always untrusted by Mathematica. Any notebook file located in any directory in UntrustedPath is untrusted by Mathematica. Mathematica will always display an alert when an untrusted notebook is opened and attempts to perform dynamic evaluations.

By default, the UntrustedPath option value contains the user's desktop folder (where web browser downloads are likely to be stored), the user's configuration folder (where email attachments are likely to be stored), and the computer's temporary directory. If the user has configured his or her web browser or email program to save downloaded files in nonstandard locations, then the user is encouraged to add these locations to the UntrustedPath option value.

Here are the directories on your computer that are currently untrusted by Mathematica:

```
Dynamic[{Column[{ToFileName[CurrentValue[$FrontEnd, {"NotebookSecurityOptions", "UntrustedPath"}]}],
/home/usr2/larrya/Desktop/
/home/usr2/larrya/Downloads/
/home/usr2/larrya/Library/
/tmp/
/var/
/private/}
```

Edit UntrustedPath...
Nesting

Directories in `TrustedPath` and `UntrustedPath` can be nested. A notebook is trusted if the most deeply nested directory containing the notebook is trusted. Consider the following example:

- `FrontEnd`\`FileName[{$HomeDirectory, “Desktop”}]` is untrusted.
- `FrontEnd`\`FileName[{$HomeDirectory, “Desktop”, “SafeNotebooks”}]` is trusted.
- `FrontEnd`\`FileName[{$HomeDirectory, “Desktop”}, “SomeDownload.nb“]` would be untrusted because “Desktop” is untrusted.
- `FrontEnd`\`FileName[{$HomeDirectory, “Desktop”, “SafeNotebooks”, “MyNotebook.nb”}]` would be trusted because “SafeNotebooks” is trusted.

TrustByDefault

The `TrustByDefault` option determines whether `Mathematica` should display an alert when the user opens notebooks with dynamic content whose containing directories are neither trusted nor untrusted. Below are the possible values for the `TrustByDefault` option.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>A notebook which is not located in a directory in <code>UntrustedPath</code> is considered to be trusted and will not display an alert when opened.</td>
</tr>
<tr>
<td>False</td>
<td>A notebook which is not located in a directory in <code>TrustedPath</code> is considered to be untrusted and will display an alert when opened.</td>
</tr>
<tr>
<td>Automatic</td>
<td>A notebook which is not located in any directory in either <code>TrustedPath</code> or <code>UntrustedPath</code> will display an alert when opened only if the notebook contains unsafe dynamic content (see below).</td>
</tr>
</tbody>
</table>

Values for `TrustByDefault` option.

The current value of the `TrustByDefault` option is: Automatic

Unsafe Dynamic Content

Dynamic content is considered unsafe if it:

- uses File operations
- uses interprocess communication via `MathLink Mathematica` Functions
- uses `J/Link` or `.NET/Link`
- uses Low-Level Notebook Programming
- uses data as code by Converting between Expressions and Strings
- uses Namespace Management
- uses Options Management
- uses External Programs

**Changing Option Values**

`TrustedPath`, `UntrustedPath`, and `TrustByDefault` are options in the `NotebookSecurityOptions` category. They can be changed using the `Preferences...` dialog or the `Option Inspector...`.

Any attempt to change the value of the `TrustedPath`, `UntrustedPath`, or `TrustByDefault` options will cause *Mathematica* to prompt the user to confirm the change. *Mathematica* does this as a security precaution so that malicious code cannot change the value of these options without the user's knowledge.
**MathLink and External Program Communication**

**Introduction to MathLink**

In many cases, you will find it convenient to communicate with external programs at a high level, and to exchange structured data with them.

On almost all computer systems, Mathematica supports the MathLink communication standard, which allows higher-level communication between Mathematica and external programs. In order to use MathLink, an external program has to include some special source code and a MathLink library, which are usually distributed with Mathematica.

MathLink allows external programs both to call Mathematica, and to be called by Mathematica. "MathLink and External Program Communication" discusses some of the details of MathLink. By using MathLink, you can, for example, treat Mathematica essentially like a subroutine embedded inside an external program. Or you can create a front end that implements your own user interface, and communicates with the Mathematica kernel via MathLink.

You can also use MathLink to let Mathematica call individual functions inside an external program. As described in "MathLink and External Program Communication", you can set up a MathLink template file to specify how particular functions in Mathematica should call functions inside your external program. From the MathLink template file, you can generate source code to include in your program. Then when you start your program, the appropriate Mathematica definitions are automatically made, and when you call a particular Mathematica function, code in your external program is executed.

<table>
<thead>
<tr>
<th>Install[&quot;command&quot;]</th>
<th>start an external program and install Mathematica definitions to call functions it contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninstall[link]</td>
<td>terminate an external program and uninstall definitions for functions in it</td>
</tr>
</tbody>
</table>

Calling functions in external programs.
This starts the external program `simul`, and installs *Mathematica* definitions to call various functions in it.

```math
In[1]:= Install["simul"]
Out[1]= LinkObject[simul, 5, 4]
```

Here is a usage message for a function that was installed in *Mathematica* to call a function in the external program.

```math
In[2]:= ? srun

srun[{a, r, gamma}, x] performs a simulation with the specified parameters.
```

When you call this function, it executes code in the external program.

```math
In[3]:= srun[{3, 0, 7}, 5]
```

This terminates the `simul` program.

```math
In[4]:= Uninstall["simul"]
Out[4]= simul
```

You can use *MathLink* to communicate with many types of programs, including with *Mathematica* itself. There are versions of the *MathLink* library for a variety of common programming languages. The *J/Link* system provides a standard way to integrate *Mathematica* with Java, based on *MathLink*. With *J/Link* you can take any Java class, and immediately make its methods accessible as functions in *Mathematica*.

## How *MathLink* Is Used

*MathLink* provides a mechanism through which programs can interact with *Mathematica*.

- Calling functions in an external program from within *Mathematica*.
- Calling *Mathematica* from within an external program.
- Setting up alternative front ends to *Mathematica*.
- Exchanging data between *Mathematica* and external programs.
- Exchanging data between concurrent *Mathematica* processes.

Some typical uses of *MathLink*. 
MathLink provides a general interface for external programs to communicate with Mathematica. Many standard software systems now have MathLink compatibility either built in or available in add-on modules.

In addition, the MathLink Developer Kit bundled with most versions of Mathematica provides the tools you need to create your own MathLink-compatible programs.

Once you have a MathLink-compatible program, you can transparently establish a link between it and Mathematica.

The link can either be on a single computer, or it can be over a network, potentially with a different type of computer at each end.

- Implementing inner loops in a low-level language.
- Handling large volumes of data external to Mathematica.
- Sending Mathematica graphics or other data for special processing.
- Connecting to a system with an existing user interface.

A few uses of MathLink-compatible programs.

MathLink-compatible programs range from very simple to very complex. A minimal MathLink-compatible program is just a few lines long. But it is also possible to build very large and sophisticated MathLink-compatible programs. Indeed, the Mathematica notebook front end is one example of a sophisticated MathLink-compatible program.

MathLink is a mechanism for exchanging Mathematica expressions between programs.

The basic idea of MathLink.

Much of the power of MathLink comes from its use of Mathematica expressions. The basic idea is that MathLink provides a way to exchange Mathematica expressions between programs, and such expressions can represent absolutely any kind of data.

- An array of numbers.
- A collection of geometrical objects.
- A sequence of commands.
- A stream of text.
- Records in a database.
- The cells of a Mathematica notebook.

A few examples of data represented by Mathematica expressions in MathLink.
The MathLink library consists of a collection of routines that allow external programs to send and receive Mathematica expressions using the fundamental C data types.

The MathLink Developer Kit provides utilities for incorporating these routines into external programs. Utilities are included for a variety of languages, although here we discuss mainly the case of C.

An important feature of the MathLink library is that it is completely platform independent: it can transparently use any interprogram communication mechanism that exists on your computer system.

## Installing Existing MathLink-Compatible Programs

One of the most common uses of MathLink is to allow you to call functions in an external program from within Mathematica. Once the external program has been set up, all you need do to be able to use it is to “install” it in your current Mathematica session.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Install[&quot;prog&quot;]</td>
<td>install a MathLink-compatible external program</td>
</tr>
<tr>
<td>Uninstall[link]</td>
<td>uninstall the program</td>
</tr>
</tbody>
</table>

Setting up external programs with functions to be called from within Mathematica.

This installs a MathLink-compatible external program called bitprog.

```
In[1]:= Install["bitprog"]

Out[1]= LinkObject[./bitprog, 6, 5]
```

BitShift is one of the functions inside bitprog.

```
In[2]:= BitShift[111, 3]

```

You can use it just as you would a function within Mathematica.

```
In[3]:= Table[BitShift[111, i], {i, 3, 8}]

Out[3]= {13, 6, 3, 1, 0, 0}
```

When you have a package written in the Mathematica language a single version will run unchanged on any computer system. But external programs typically need to be compiled separately for every different type of computer.
*Mathematica* has a convention of keeping versions of external programs in directories that are named after the types of computers on which they will run. And assuming that this convention has been followed, `Install["prog"]` should always install the version of *prog* appropriate for the particular kind of computer that you are currently using.

<table>
<thead>
<tr>
<th>Install[&quot;name&quot;]</th>
<th>install a program found anywhere on $Path</th>
</tr>
</thead>
</table>

Using context names to specify programs to install.

When you ask to read in a *Mathematica* language file using `<< name`, *Mathematica* will automatically search all directories in the list `$Path` in order to find a file with the appropriate name. Similarly, if you use `Install["name"]` *Mathematica* will automatically search all directories in `$Path` in order to find an external program with the name `name.exe`. `Install["name"]` allows you to install programs that are stored in a central directory without explicitly having to specify their location.

### Setting Up External Functions to Be Called from *Mathematica*

If you have a function defined in an external program, then what you need to do in order to make it possible to call the function from within *Mathematica* is to add appropriate *MathLink* code that passes arguments to the function, and takes back the results it produces.

In simple cases, you can generate the necessary code just by giving an appropriate *MathLink* template for each external function.

```mathematica
:Begin:
:Function:      f
:Pattern:       f[x_Integer, y_Integer]
:Arguments:     {x, y}
:ArgumentTypes: {Integer, Integer}
:returnType:    Integer
:End:
```

A file `f.tm` containing a *MathLink* template for an external function `f`. 
The elements of a *MathLink* template.

Once you have constructed a *MathLink* template for a particular external function, you have to combine this template with the actual source code for the function. Assuming that the source code is written in the C programming language, you can do this just by adding a line to include the standard *MathLink* header file, and then inserting a small `main` program.

Include the standard *MathLink* header file.

```
#include "mathlink.h"
```

Here is the actual source code for the function $f$.

```
int f(int x, int y) {
    return x+y;
}
```

This sets up the external program to be ready to take requests from *Mathematica*.

```
int main(int argc, char *argv[]) {
    return MLMain(argc, argv);
}
```

A file `f.c` containing C source code.

Note that the form of `main` required on different systems may be slightly different. The release notes included in the *MathLink* Developer Kit on your particular computer system should give the appropriate form.

```
mprep
```

Typical external programs for processing *MathLink* source files.
MathLink templates are conventionally put in files with names of the form file.tm. Such files can also contain C source code, interspersed between templates for different functions.

Once you have set up the appropriate files, you then need to process the MathLink template information, and compile all of your source code. Typically you do this by running various external programs, but the details will depend on your computer system.

Under Unix, for example, the MathLink Developer Kit includes a program named mcc which will preprocess MathLink templates in any file whose name ends with .tm, and then call cc on the resulting C source code. mcc will pass command-line options and other files directly to cc.

This preprocesses f.tm, then compiles the resulting C source file together with the file f.c.

```
mcc -o f.exe f.tm f.c
```

This installs the binary in the current Mathematica session.

```
In[1]:= Install["f.exe"]
Out[1]= LinkObject[f.exe, 4, 4]
```

Now f[x, y] calls the external function f (int x, int y) and adds two integers together.

```
In[2]:= f[6, 9]
Out[2]= 15
```

The external program handles only machine integers, so this gives a peculiar result.

```
In[3]:= f[2^31 - 1, 5]
Out[3]= -2147483644
```

On Windows, the MathLink Developer Kit includes a program named mprep, which you have to call directly, giving as input all of the .tm files that you want to preprocess. mprep will generate C source code as output, which you can then feed to a C compiler.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Install[&quot;prog&quot;]</td>
<td>install an external program</td>
</tr>
<tr>
<td>Uninstall[link]</td>
<td>uninstall an external program</td>
</tr>
<tr>
<td>Links[&quot;prog&quot;]</td>
<td>show active links associated with &quot;prog&quot;</td>
</tr>
<tr>
<td>Links[]</td>
<td>show all active links</td>
</tr>
<tr>
<td>LinkPatterns[link]</td>
<td>show patterns that can be evaluated on a particular link</td>
</tr>
</tbody>
</table>

Handling links to external programs.
This finds the link to the \texttt{f.exe} program.

\begin{verbatim}
In[4]:= Links["f.exe"]
Out[4]= {LinkObject[./f.exe, 8, 6]}
\end{verbatim}

This shows the \textit{Mathematica} patterns that can be evaluated using the link.

\begin{verbatim}
In[5]:= LinkPatterns[%%[[1]]]
Out[5]= {f[x_Integer, y_Integer]}
\end{verbatim}

Install sets up the actual function \texttt{f} to execute an appropriate \texttt{ExternalCall} function.

\begin{verbatim}
In[6]:= ? f
Global`f
f[x_Integer, y_Integer] :=
    ExternalCall[LinkObject[./f.exe, 8, 6], CallPacket[0, {x, y}]]
\end{verbatim}

When a \textit{MathLink} template file is processed, two basic things are done. First, the \texttt{:Pattern:} and \texttt{:Arguments:} specifications are used to generate a \textit{Mathematica} definition that calls an external function via \textit{MathLink}. And second, the \texttt{:Function:}, \texttt{:ArgumentTypes:} and \texttt{:ReturnType:} specifications are used to generate C source code that calls your function within the external program.

\texttt{:Begin:}

This gives the name of the actual C function to call in the external program.

\texttt{:Function:} \quad \texttt{prog_add}

This gives the \textit{Mathematica} pattern for which a definition should be set up.

\texttt{:Pattern:} \quad \texttt{SkewAdd[x_Integer, y_Integer:1]}

The values of the two list elements are the actual arguments to be passed to the external function.

\texttt{:Arguments:} \quad \{x, If[x > 1, y, y + x - 2]\}

This specifies that the arguments should be passed as integers to the C function.

\texttt{:ArgumentTypes:} \quad \{\text{Integer, Integer}\}
This specifies that the return value from the C function will be an integer.

:ReturnType:    Integer

:End:

Both the :Pattern: and :Arguments: specifications in a MathLink template can be any Mathematica expressions. Whatever you give as the :Arguments: specification will be evaluated every time you call the external function. The result of the evaluation will be used as the list of arguments to pass to the function.

Sometimes you may want to set up Mathematica expressions that should be evaluated not when an external function is called, but instead only when the external function is first installed.

You can do this by inserting :Evaluate: specifications in your MathLink template. The expression you give after :Evaluate: can go on for several lines: it is assumed to end when there is first a blank line, or a line that does not begin with spaces or tabs.

This specifies that a usage message for SkewAdd should be set up when the external program is installed.

:Evaluate:    SkewAdd::usage = "SkewAdd\[x, y\] performs a skew addition in an external program."

When an external program is installed, the specifications in its MathLink template file are used in the order they were given. This means that any expressions given in :Evaluate: specifications that appear before :Begin: will have been evaluated before definitions for the external function are set up.

Here are Mathematica expressions to be evaluated before the definitions for external functions are set up.

:Evaluate:    BeginPackage["XPack`"]
:Evaluate:    XF1::usage = "XF1\[x, y\] is one external function."
:Evaluate:    XF2::usage = "XF2\[x\] is another external function."
:Evaluate:    Begin["`Private`"]
This specifies that the function XF1 in *Mathematica* should be set up to call the function \( f \) in the external C program.

```
:Begin:
:Function:       f
:Pattern:        XF1[x_Integer, y_Integer]
:Arguments:      {x, y}
:ArgumentTypes:  {Integer, Integer}
:ReturnType:     Integer
:End:
```

This specifies that XF2 in *Mathematica* should call g. Its argument and return value are taken to be approximate real numbers.

```
:Begin:
:Function:       g
:Pattern:        XF2[x_?NumberQ]
:Arguments:      {x}
:ArgumentTypes:  {Real}
:ReturnType:     Real
:End:
```

These *Mathematica* expressions are evaluated after the definitions for the external functions. They end the special context used for the definitions.

```
:Evaluate:  End[ ]
:Evaluate:  EndPackage[ ]
```

Here is the actual source code for the function \( f \). There is no need for the arguments of this function to have the same names as their *Mathematica* counterparts.

```c
int f(int i, int j) {
    return i + j;
}
```

Here is the actual source code for g. Numbers that you give in *Mathematica* will automatically be converted into C double types before being passed to g.

```c
double g(double x) {
    return x*x;
}
```
By using :Evaluate: specifications, you can evaluate *Mathematica* expressions when an external program is first installed. You can also execute code inside the external program at this time simply by inserting the code in main() before the call to MLMain(). This is sometimes useful if you need to initialize the external program before any functions in it are used.

```mathematica
MLEvaluateString [stdlib, "string"]
evaluate a string as *Mathematica* input
```

Executing a command in *Mathematica* from within an external program.

```mathematica
int diff(int i, int j) {

    This evaluates a *Mathematica* Print function if i < j.
    if (i < j) MLEvaluateString (stdlib, "Print["negative"]");
    return i - j;
}
```

This installs an external program containing the diff function defined above.

```mathematica
In[7]: Install["diffprog"]
Out[7]= LinkObject[./diffprog, 9, 7]
```

Calling diff causes Print to be executed.

```mathematica
In[8]:= diff[4, 7]
negative
Out[8]= -3
```

Note that any results generated in the evaluation requested by MLEvaluateString() are ignored. To make use of such results requires full two-way communication between *Mathematica* and external programs, as discussed in "Two-Way Communication with External Programs".

**Handling Lists, Arrays and Other Expressions**

*MathLink* allows you to exchange data of any type with external programs. For more common types of data, you simply need to give appropriate :ArgumentTypes: or :ReturnType: specifications in your *MathLink* template file.
<table>
<thead>
<tr>
<th>Mathematica specification</th>
<th>C specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>integer</td>
</tr>
<tr>
<td>Real</td>
<td>floating-point number</td>
</tr>
<tr>
<td>IntegerList</td>
<td>list of integers</td>
</tr>
<tr>
<td>RealList</td>
<td>list of floating-point numbers</td>
</tr>
<tr>
<td>String</td>
<td>character string</td>
</tr>
<tr>
<td>Symbol</td>
<td>symbol name</td>
</tr>
<tr>
<td>Manual</td>
<td>call MathLink routines directly</td>
</tr>
</tbody>
</table>

Basic type specifications.

Here is the MathLink template for a function that takes a list of integers as its argument.

```mathematica
:Begin:
:Function:       h
:Pattern:        h[a_List]
:Arguments:      {a}
:ArgumentTypes:  {IntegerList}
:return:        Integer
:End:
```

Here is the C source code for the function. Note the extra argument alen which is used to pass the length of the list.

```c
int h(int *a, long alen) {
    int i, tot=0;
    for(i=0; i<alen; i++)
        tot += a[i];
    return tot;
}
```

This installs an external program containing the specifications for the function h.

```
In[1]:= Install["hprog"]
Out[1]= LinkObject[./hprog, 11, 8]
```
This calls the external code.

```mathematica
In[2]:= h[{3, 5, 6}]
Out[2]= 14
```

This does not match the pattern `h[a_List]` so does not call the external code.

```mathematica
In[3]:= h[67]
Out[3]= h[67]
```

The pattern is matched, but the elements in the list are of the wrong type for the external code, so $Failed is returned.

```mathematica
In[4]:= h[{a, b, c}]
```

You can mix basic types of arguments in any way you want. Whenever you use `IntegerList` or `RealList`, however, you have to include an extra argument in your C program to represent the length of the list.

Here is an `:ArgumentTypes:` specification.

```mathematica
:ArgumentTypes: {IntegerList, RealList, Integer}
```

Here is a possible corresponding C function declaration.

```c
void f(int *a, long alen, double *b, long blen, int c)
```

Note that when a list is passed to a C program by `MathLink` its first element is assumed to be at position 0, as is standard in C, rather than at position 1, as is standard in `Mathematica`.

In addition, following C standards, character strings specified by `String` are passed as `char *` objects, terminated by \0 null bytes. "Portability of `MathLink` Programs" discusses how to handle special characters.
MLPutInteger32 (stdlink, int i)  
put a single integer

MLPutReal64 (stdlink, double x)  
put a single floating-point number

MLPutInteger32List (stdlink, int* a, int n)  
put a list of \( n \) integers starting from location \( a \)

MLPutReal64List (stdlink, double* a, int n)  
put a list of \( n \) floating-point numbers starting from location \( a \)

MLPutInteger32Array (stdlink, int* a, int* dims, NULL, int d)  
put an array of integers to form a depth \( d \) list with dimensions \( \text{dims} \)

MLPutReal64Array (stdlink, double* a, int* dims, NULL, int d)  
put an array of floating-point numbers

MLPutString (stdlink, char* s)  
put a character string

MLPutSymbol (stdlink, char* s)  
put a character string as a symbol name

MLPutFunction (stdlink, char* s, int n)  
begin putting a function with head \( s \) and \( n \) arguments

MathLink functions for sending data to Mathematica.

When you use a MathLink template file, what mprep and mcc actually do is to create a C program that includes explicit calls to MathLink library functions. If you want to see an example of how to use the MathLink library functions directly, you can look at the source code of this program. Note when you use mcc, you typically need to give a –g option, otherwise the source code that is generated is automatically deleted.

If your external function just returns a single integer or floating-point number, then you can specify this just by giving Integer or Real as the :ReturnType: in your MathLink template file. But because of the way memory allocation and deallocation work in C, you cannot directly give :ReturnType: specifications such as IntegerList or RealList. And instead, to return such structures, you must explicitly call MathLink library functions within your C program, and give Manual as the :ReturnType: specification.
Here is the *MathLink* template for a function that takes an integer as an argument, and returns a list of the digits in its binary representation using explicit *MathLink* functions.

```plaintext
:Begin:
:Function:       bits
:Pattern:        bits[i_Integer]
:Arguments:      {i}
:ArgumentTypes:  {Integer}
:ReturnType:     Manual
:End:
```

The return type of the function is declared as `void`.

```c
void bits(int i) {
    int a[32], k;

    This puts values into the C array `a`.
    for(k=0; k<32; k++) {
        a[k] = i%2;
        i >>= 1;
        if (i==0) break;
    }

    if (k<32) k++;

    This sends `k` elements of the array `a` back to *Mathematica*.
    MLPutInteger32List(stdlink, a, k);
    return ;
}
```

This installs the program containing the external function `bits`.

```
In[5]:= Install["bitsprog"]
```

The external function now returns a list of bits.

```
In[6]:= bits[14]
Out[6]= {0, 1, 1, 1}
```
If you declare an array in C as `int a[n][n][n]`, then you can use `MLPutInteger32Array()` to send it to Mathematica as a depth 3 list.

```c
int a[8][16][100];

int dims[] = {8, 16, 100};

MLPutInteger32Array(stdlink, a, dims, NULL, 3);
```

You can use MathLink functions to create absolutely any Mathematica expression. The basic idea is to call a sequence of MathLink functions that correspond directly to the FullForm representation of the Mathematica expression.

```c
MLPutFunction(stdlink, "Plus", 2);

MLPutInteger32(stdlink, 77);

MLPutSymbol(stdlink, "x");
```

In general, you first call `MLPutFunction()`, giving the head of the Mathematica function you want to create, and the number of arguments it has. Then you call other MathLink functions to fill in each of these arguments in turn. "Expressions" discusses the general structure of Mathematica expressions and the notion of heads.

```c
MLPutFunction(stdlink, "List", 2);

MLPutInteger32List(stdlink, r, 10);
```
The second element of the main list is itself a list with 2 elements.
MLPutFunction(stdlink, "List", 2);

The first element of this sublist is a floating-point number.
MLPutReal64(stdlink, 4.5);

The second element is an integer.
MLPutInteger32(stdlink, 11);

MLPutInteger32Array() and MLPutReal64Array() allow you to send arrays which are laid out in memory in the one-dimensional way that C pre-allocates them. But if you create arrays during the execution of a C program, it is more common to set them up as nested collections of pointers. You can send such arrays to Mathematica by using a sequence of MLPutFunction() calls, ending with an MLPutInteger32List() call.

This declares a to be a nested list of lists of lists of integers.
int ***a;

This creates a Mathematica list with n1 elements.
MLPutFunction(stdlink, "List", n1);
for (i=0; i<n1; i++) {

This creates a sublist with n2 elements.
    MLPutFunction(stdlink, "List", n2);
    for (j=0; j<n2; j++) {

This writes out lists of integers.
        MLPutInteger32List(stdlink, a[i][j], n3);
    }
}

It is important to realize that any expression you create using MathLink functions will be evaluated as soon as it is sent to Mathematica. This means, for example, that if you wanted to transpose an array that you were sending back to Mathematica, all you would need to do is to wrap
a Transpose around the expression representing the array. You can then do this simply by calling
\texttt{MLPutFunction(\textit{stdlibk}, "Transpose", 1);} just before you start creating the expression that represents the array.

The idea of postprocessing data that you send back to Mathematica has many uses. One example is as a way of sending lists whose length you do not know in advance.

This creates a list in Mathematica by explicitly appending successive elements.

\begin{verbatim}
In[7]:= t = {}; Do[t = Append[t, i^2], {i, 5}]; t
Out[7]= {1, 4, 9, 16, 25}
\end{verbatim}

This creates a list in which each successive element is in a nested sublist.

\begin{verbatim}
In[8]:= t = {}; Do[t = {t, i^2}, {i, 5}]; t
Out[8]= {{{{}, 1}, {4}, 9}, 16}, 25}
\end{verbatim}

\texttt{Flatten} flattens out the list.

\begin{verbatim}
In[9]:= Flatten[t]
Out[9]= {1, 4, 9, 16, 25}
\end{verbatim}

\texttt{Sequence} automatically flattens itself.

\begin{verbatim}
In[10]:= {Sequence[1, Sequence[4, Sequence[]]]}
Out[10]= {1, 4}
\end{verbatim}

In order to call \texttt{MLPutInteger32List()}, you need to know the length of the list you want to send. But by creating a sequence of nested \texttt{Sequence} objects, you can avoid having to know the length of your whole list in advance.

This sets up the \texttt{List} around your result.

\begin{verbatim}
MLPutFunction(\textit{stdlibk}, "List", 1);
while( \textit{condition} ) {
    /* generate an element */
    Create the next level \texttt{Sequence} object.
    MLPutFunction(\textit{stdlibk}, "Sequence", 2);
\end{verbatim}
Put the element.

    MLPutInteger32(stdlink,  i );

}

This closes off your last Sequence object.

    MLPutFunction(stdlink, "Sequence", 0);

Basic functions for explicitly getting data from Mathematica.

MathLink provides functions like MLPutInteger32() to send data from an external program into Mathematica. MathLink also provides functions like MLGetInteger32() that allow you to get data from Mathematica into an external program.

The list that you give for :ArgumentTypes: in a MathLink template can end with Manual, indicating that after other arguments have been received, you will call MathLink functions to get additional expressions.

    :Begin:
    :Function:       f

    The function f in Mathematica takes 3 arguments.

    :Pattern:       f[i_Integer, x_Real, y_Real]

    All these arguments are passed directly to the external program.

    :Arguments:       {i, x, y}

    Only the first argument is sent directly to the external function.

    :ArgumentTypes:       {Integer, Manual}

    :ReturnType:       Real
    :End:

    The external function only takes one explicit argument.

double f(int i) {

    MLGetInteger32 (stdlink,int*i)  get an integer, storing it at address i
    MLGetReal64  (stdlink,double*x) get a floating-point number, storing it at address x
This declares the variables \( x \) and \( y \).

```c
double x, y;
```

MLGetReal64() explicitly gets data from the link.

```c
MLGetReal64(stdlink, &x);
MLGetReal64(stdlink, &y);
```

```c
return i+x+y;
```

}\)

MathLink functions such as MLGetInteger32 \((\text{link}, \pi)\) work much like standard C library functions such as \texttt{scanf} \((\text{fp}, "%d", \pi)\). The first argument specifies the link from which to get data. The last argument gives the address at which the data that is obtained should be stored.

---

Getting a function via MathLink.

\begin{verbatim}
:Begin:
:Function:        f

The function \texttt{f} in Mathematica takes a list of integers as an argument.

:Pattern:        f[a:{___Integer}]

The list is passed directly to the external program.

:Arguments:      \{a\}

The argument is to be retrieved manually by the external program.

:ArgumentTypes:  \{Manual\}

:ReturnType:     \texttt{Integer}
:End:
\end{verbatim}

The external function takes no explicit arguments.

```c
int f(void) {
```
This declares local variables.

    int n, i;
    int a[MAX];

This checks that the function being sent is a list, and stores how many elements it has in n.

    MLCheckFunction(stdlink, "List", &n);

This gets each element in the list, storing it in a[i].

    for (i=0; i<n; i++)
        MLGetInteger32(stdlink, a+i);

In simple cases, it is usually possible to ensure on the Mathematica side that the data you send to an external program has the structure that is expected. But in general the return value from MLCheckFunction() will be MLSUCCESS only if the data consists of a function with the name you specify.

Note that if you want to get a nested collection of lists or other objects, you can do this by making an appropriate sequence of calls to MLCheckFunction().

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLGetInteger32List</td>
<td>get a list of integers, allocating the memory needed to store it</td>
</tr>
<tr>
<td>MLGetReal64List</td>
<td>get a list of floating-point numbers</td>
</tr>
<tr>
<td>MLReleaseInteger32List</td>
<td>release the memory associated with a list of integers</td>
</tr>
<tr>
<td>MLReleaseReal64List</td>
<td>release the memory associated with a list of floating-point numbers</td>
</tr>
</tbody>
</table>

Getting lists of numbers.

When an external program gets data from Mathematica, it must set up a place to store the data. If the data consists of a single integer, as in MLGetInteger32 (stdlink, & n), then it suffices just to have declared this integer using int n.

But when the data consists of a list of integers of potentially any length, memory must be allocated to store this list at the time when the external program is actually called.
MLGetInteger32List (stdcall, &a, &n) will automatically do this allocation, setting a to be a pointer to the result. Note that memory allocated by functions like MLGetInteger32List() is always in a special reserved area, so you cannot modify or free it directly.

Here is an external program that will be sent a list of integers.

```c
int f(void) {

    // This declares local variables. a is an array of integers.
    int n;
    int *a;

    // This gets a list of integers, making a be a pointer to the result.
    MLGetInteger32List(stdlink, &a, &n);

    // This releases the memory used to store the list of integers.
    MLReleaseInteger32List(stdlink, a, n);

    ...
}
```

If you use IntegerList as an :ArgumentTypes: specification, then MathLink will automatically release the memory used for the list after your external function exits. But if you get a list of integers explicitly using MLGetInteger32List(), then you must not forget to release the memory used to store the list after you have finished with it.

---

**Getting arrays of numbers.**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLGetInteger32Array</td>
<td>get an array of integers of any depth</td>
</tr>
<tr>
<td>MLGetReal64Array</td>
<td>get an array of floating-point numbers of any depth</td>
</tr>
<tr>
<td>MLReleaseInteger32Array</td>
<td>release memory associated with an integer array</td>
</tr>
<tr>
<td>MLReleaseRealArray</td>
<td>release memory associated with a floating-point array</td>
</tr>
</tbody>
</table>
MLGetInteger32List() extracts a one-dimensional array of integers from a single Mathematica list. MLGetInteger32Array() extracts an array of integers from a collection of lists or other Mathematica functions nested to any depth.

The name of the Mathematica function at level $i$ in the structure is stored as a string in $\text{heads}[i]$. The size of the structure at level $i$ is stored in $\text{dims}[i]$, while the total depth is stored in $d$.

If you pass a list of complex numbers to your external program, then MLGetReal64Array() will create a two-dimensional array containing a sequence of pairs of real and imaginary parts. In this case, $\text{heads}[0]$ will be "List" while $\text{heads}[1]$ will be "Complex".

Note that you can conveniently exchange arbitrary-precision numbers with external programs by converting them to lists of digits in Mathematica using IntegerDigits and RealDigits.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLGetString (stdin, char**s)</td>
<td>get a character string</td>
</tr>
<tr>
<td>MLGetSymbol (stdin, char**s)</td>
<td>get a symbol name</td>
</tr>
<tr>
<td>MLReleaseString (stdin, char*s)</td>
<td>release memory associated with a character string</td>
</tr>
<tr>
<td>MLReleaseSymbol (stdin, char*s)</td>
<td>release memory associated with a symbol name</td>
</tr>
</tbody>
</table>

Getting character strings and symbol names.

If you use String as an :ArgumentTypes: specification, then MathLink will automatically release the memory that is used to store the string after your function exits. This means that if you want to continue to refer to the string, you must allocate memory for it, and explicitly copy each character in it.

If you get a string using MLGetString(), however, then MathLink will not automatically release the memory used for the string when your function exits. As a result, you can continue referring to the string. Be careful not to modify the contents of the string by writing to the memory that is returned by MLGetString(). When you no longer need the string, you must nevertheless explicitly call MLReleaseString() in order to release the memory associated with it.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLGetFunction (stdin, char**s, int*n)</td>
<td>begin getting a function, storing the name of the head in $s$ and the number of arguments in $n$</td>
</tr>
<tr>
<td>MLReleaseSymbol (stdin, char*s)</td>
<td>release memory associated with a function name</td>
</tr>
</tbody>
</table>

Getting an arbitrary function.
If you know what function to expect in your external program, then it is usually simpler to call MLCheckFunction(). But if you do not know what function to expect, you have no choice but to call MLGetFunction(). If you do this, you need to be sure to call MLReleaseSymbol() to release the memory associated with the name of the function that is found by MLGetFunction().

**Portability of MathLink Programs**

The Mathematica side of a MathLink connection is set up to work exactly the same on all computer systems. But inevitably there are differences between external programs on different computer systems.

For a start, different computer systems almost always require different executable binaries. When you call Install["prog"], therefore, you must be sure that *prog* corresponds to a program that can be executed on your particular computer system.

---

**Table: Installing programs on different computer systems.**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Install[&quot;file&quot;]</code></td>
<td>try to execute <code>file</code> directly</td>
</tr>
<tr>
<td><code>Install[&quot;file&quot;,LinkProtocol-&gt;&quot;type&quot;]</code></td>
<td>use the specified protocol for low-level data transport</td>
</tr>
<tr>
<td><code>$SystemID</code></td>
<td>identify the type of computer system being used</td>
</tr>
<tr>
<td><code>Install[&quot;dir&quot;]</code></td>
<td>try to execute a file with a name of the form <code>dir / $SystemID / dir</code></td>
</tr>
</tbody>
</table>

---

Even though the executable binary of an external program is inevitably different on different computer systems, it can still be the case that the source code in a language such as C from which this binary is obtained can be essentially the same.
But to achieve portability in your C source code there are several points that you need to watch.

For a start, you should never make use of extra features of the C language or C runtime libraries that happen to be provided on a particular system, but are not part of standard C. In addition, you should try to avoid dealing with segmented or otherwise special memory models.

The include file `mathlink.h` contains standard C prototypes for all the functions in the `MathLink` library.

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLPutInteger32()</td>
<td>MLGetInteger32()</td>
<td>integer corresponding to C type <code>int</code>, that is, 32 bits</td>
</tr>
<tr>
<td>MLPutInteger16()</td>
<td>MLGetInteger16()</td>
<td>integer of type <code>short</code>, that is, 16 bits</td>
</tr>
<tr>
<td>MLPutInteger64()</td>
<td>MLGetInteger64()</td>
<td>64-bit integer</td>
</tr>
<tr>
<td>MLPutReal164()</td>
<td>MLGetReal164()</td>
<td>IEEE double-precision real number, corresponding to the C-language type <code>double</code></td>
</tr>
<tr>
<td>MLPutReal32()</td>
<td>MLGetReal32()</td>
<td>IEEE single-precision real number, corresponding to the C-language type <code>float</code></td>
</tr>
<tr>
<td>MLPutReal128()</td>
<td>MLGetReal128()</td>
<td>IEEE quad-precision real number</td>
</tr>
</tbody>
</table>

`MathLink` functions that use specific C types.

If you are going to call `MathLink` library functions in a portable way, it is essential that you use the same types as they do.

If your programs correctly match the argument types for the `MathLink` library functions, you do not have to worry about C type differences between computer systems. `MathLink` automatically converts the C types to the appropriate sizes for each platform. `MathLink` also swaps bytes as needed to correctly transfer numbers across platforms, and it converts between floating-point number formats with the smallest possible loss of precision.
### Manipulating general strings.

In simple C programs, it is typical to use strings that contain only ordinary ASCII characters. But in *Mathematica* it is possible to have strings containing all sorts of special characters. These characters are specified within *Mathematica* using Unicode character codes, as discussed in "Raw Character Encodings".

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLPutString (stdlink, char* s)</td>
<td>put a null-terminated C character string</td>
</tr>
<tr>
<td>MLPutUnicodeString (stdlink, unsigned short* s, int n)</td>
<td>put a string encoded in terms of 16-bit UCS-2 Unicode characters</td>
</tr>
<tr>
<td>MLPutByteString (stdlink, unsigned char* s, int n)</td>
<td>put a string containing only 8-bit character codes</td>
</tr>
<tr>
<td>MLPutUTF8String (stdlink, const unsigned char* s, int n)</td>
<td>put a string of UTF-8 encoded Unicode characters</td>
</tr>
<tr>
<td>MLPutUTF16String (stdlink, const unsigned short* s, int n)</td>
<td>put a string of UTF-16 encoded Unicode characters</td>
</tr>
<tr>
<td>MLPutUTF32String (stdlink, const unsigned int* s, int n)</td>
<td>put a string of UTF-32 encoded Unicode characters</td>
</tr>
<tr>
<td>MLGetString (stdlink, char** s)</td>
<td>get a null-terminated C character string</td>
</tr>
<tr>
<td>MLGetUnicodeString (stdlink, unsigned short** s, long* n)</td>
<td>get a string encoded in terms of 16-bit UCS-2 Unicode characters</td>
</tr>
<tr>
<td>MLGetByteString (stdlink, unsigned char** s, long* n, long spec)</td>
<td>get a string containing only 8-bit character codes, using spec as the code for all 16-bit characters</td>
</tr>
<tr>
<td>MLGetUTF8String (stdlink, const unsigned char** s, int* m, int* n)</td>
<td>get a string of UTF-8 encoded Unicode characters</td>
</tr>
<tr>
<td>MLGetUTF16String (stdlink, const unsigned short** s, int* m, int* n)</td>
<td>get a string of UTF-16 encoded Unicode characters</td>
</tr>
<tr>
<td>MLGetUTF32String (stdlink, const unsigned int** s, int* n)</td>
<td>get a string of UTF-32 encoded Unicode characters</td>
</tr>
</tbody>
</table>
C language char* strings typically use only 8 bits to store the code for each character. UCS-2 encoded strings, however, require 16 bits. As a result, the functions MLPutUnicodeString() and MLGetUnicodeString() work with arrays of unsigned short integers. The same is true of UTF-16 encoded strings and the corresponding functions MLPutUTF16String() and MLGetUTF16String().

UTF-32 encoded strings require 32 bits for each character, and the corresponding functions MLPutUTF32String() and MLGetUTF32String() work with arrays of unsigned int integers.

If you know that your program will not have to handle special characters, then you may find it convenient to use MLPutByteString() and MLGetByteString(). These functions represent all characters directly using 8-bit character codes. If a special character is sent from Mathematica, then it will be converted by MLGetByteString() to a fixed code that you specify.

- main() may need to be different on different computer systems

A point to watch in creating portable MathLink programs.

Computer systems and compilers that have C runtime libraries based on the Unix model allow MathLink programs to have a main program of the form main(argc, argv) which simply calls MLMain(argc, argv).

Some computer systems or compilers may however require main programs of a different form. You should realize that you can do whatever initialization you want inside main() before calling MLMain(). Once you have called MLMain(), however, your program will effectively go into an infinite loop, responding to requests from Mathematica until the link to it is closed.
Using *MathLink* to Communicate between *Mathematica* Sessions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkCreate[&quot;name&quot;]</td>
<td>create a link for another program to connect to</td>
</tr>
<tr>
<td>LinkConnect[&quot;name&quot;]</td>
<td>connect to a link created by another program</td>
</tr>
<tr>
<td>LinkClose[link]</td>
<td>close a <em>MathLink</em> connection</td>
</tr>
<tr>
<td>LinkWrite[link, expr]</td>
<td>write an expression to a <em>MathLink</em> connection</td>
</tr>
<tr>
<td>LinkRead[link]</td>
<td>read an expression from a <em>MathLink</em> connection</td>
</tr>
<tr>
<td>LinkRead[link, Hold]</td>
<td>read an expression and immediately wrap it with Hold</td>
</tr>
<tr>
<td>LinkReadyQ[link]</td>
<td>find out whether there is data ready to be read from a link</td>
</tr>
<tr>
<td>LinkReadyQ[link, t]</td>
<td>wait for up to t seconds to see if an expression becomes ready to read</td>
</tr>
<tr>
<td>LinkReadyQ[{link1, link2, ...}]</td>
<td>find out whether there is data ready to be read from one of the links</td>
</tr>
<tr>
<td>LinkReadyQ[{link1, link2, ...}, t]</td>
<td>wait for up to t seconds to see if an expression becomes ready to read</td>
</tr>
</tbody>
</table>

*MathLink* connections between *Mathematica* sessions.

### Session A

This starts up a link on port number 8000.

```mathematica
In[1]:= link = LinkCreate["8000", LinkProtocol -> "TCPIP"]
```

```mathematica
Out[1]= LinkObject[8000@frog.wolfram.com,4470@frog.wolfram.com, 17, 5]
```

### Session B

This connects to the link on port 8000.

```mathematica
In[2]:= Link = LinkConnect["8000", LinkProtocol -> "TCPIP"]
```

```mathematica
Out[2]= LinkObject[8000@frog.wolfram.com, 11, 4]
```

### Session A

This evaluates 15! and writes it to the link.

```mathematica
In[3]:= LinkWrite[link, 15!]
```
This reads from the link, getting the 15! that was sent.

\[\text{In}[4] := \text{LinkRead}[\text{Link}]\]

\[\text{Out}[4] = 1307674368000\]

This writes data back on the link.

\[\text{In}[5] := \text{LinkWrite}[\text{link}, \text{N}[\%^6]]\]

And this reads the data written in session B.

\[\text{In}[6] := \text{LinkRead}[\text{link}]\]

\[\text{Out}[6] = 5.00032 \times 10^{72}\]

One use of \textit{MathLink} connections between \textit{Mathematica} sessions is simply as a way to transfer data without using intermediate files.

Another use is as a way to dispatch different parts of a computation to different sessions.

This writes the expression 2 + 2 without evaluating it.

\[\text{In}[7] := \text{LinkWrite}[\text{link}, \text{Unevaluated}[2 + 2]]\]

This reads the expression from the link, immediately wrapping it in \texttt{Hold}.

\[\text{In}[8] := \text{LinkRead}[\text{Link, Hold}]\]

\[\text{Out}[8] = \text{Hold}[2 + 2]\]

This evaluates the expression.

\[\text{In}[9] := \text{ReleaseHold}[\%]\]

\[\text{Out}[9] = 4\]
When you call `LinkWrite`, it writes an expression to the `MathLink` connection and immediately returns. But when you call `LinkRead`, it will not return until it has read a complete expression from the `MathLink` connection.

You can tell whether anything is ready to be read by calling `LinkReadyQ[link]`. If `LinkReadyQ` returns `True`, then you can safely call `LinkRead` and expect immediately to start reading an expression. But if `LinkReadyQ` returns `False`, then `LinkRead` would block until an expression for it to read had been written by a `LinkWrite` in your other `Mathematica` session.

---

**Session A**

There is nothing waiting to be read on the link, so if `LinkRead` were to be called, it would block.

```
In[10]:= LinkReadyQ[link]
Out[10]= False
```

---

**Session B**

This writes an expression to the link.

```
In[11]:= LinkWrite[Link, x + y]
```

---

**Session A**

Now there is an expression waiting to be read on the link.

```
In[12]:= LinkReadyQ[link]
Out[12]= True
```

`LinkRead` can thus be called without fear of blocking.

```
In[13]:= LinkRead[link]
Out[13]= x + y
```

`LinkReadyQ` can take a list of link objects, evaluating each link in parallel to determine if there is data to read. As in the case of a single link, a second argument specifies a time out period, causing `LinkReadyQ` to wait until one of the links is ready to use.
Ways to set up MathLink links over TCP/IP.

MathLink can use whatever mechanism for interprogram communication your computer system supports. In setting up connections between concurrent Mathematica sessions, a common mechanism is internet TCP ports.

Most computer systems have a few thousand possible numbered ports, some of which are typically allocated to standard system services.

You can use any of the unallocated ports for MathLink connections.

Session on frog.wolfram.com

This finds an unallocated port on frog.wolfram.com.

\begin{verbatim}
In[14]:= link = LinkCreate[LinkProtocol -> "TCP/IP"]
Out[14]= LinkObject["2981@frog.wolfram.com,2982@frog.wolfram.com", 5, 5]
\end{verbatim}

Session on toad.wolfram.com

This connects to the port on frog.wolfram.com.

\begin{verbatim}
In[15]:= link = LinkConnect["2981@frog.wolfram.com,2982@frog.wolfram.com", LinkProtocol -> "TCP/IP"]
Out[15]= LinkObject["2981@frog.wolfram.com,2982@frog.wolfram.com", 5, 5]
\end{verbatim}

This sends the current machine name over the link.

\begin{verbatim}
In[16]:= LinkWrite[link, $MachineName]
\end{verbatim}
Session on frog.wolfram.com

This reads the expression written on toad.

\[\text{In}[17]:= \text{LinkRead}[\text{link}]\]
\[\text{Out}[17]= \text{toad}\]

By using internet ports for *MathLink* connections, you can easily transfer data between *Mathematica* sessions on different machines. All that is needed is that an internet connection exists between the machines.

Note that because *MathLink* is completely system independent, the computers at each end of a *MathLink* connection do not have to be of the same type. *MathLink* nevertheless notices when they are, and optimizes data transmission in this case.

### Calling Subsidiary *Mathematica* Processes

**LinkLaunch["prog"]**

Start an external program and open a connection to it.

Connecting to a subsidiary program via *MathLink*.

This starts a subsidiary *Mathematica* process on the computer system used here.

\[\text{In}[1]:= \text{link} = \text{LinkLaunch}["\text{math -mathlink}\"]\]
\[\text{Out}[1]= \text{LinkObject[math -mathlink, 4, 4]}\]

Here is a packet representing the first input prompt from the subsidiary *Mathematica* process.

\[\text{In}[2]:= \text{LinkRead}[\text{link}]\]
\[\text{Out}[2]= \text{InputNamePacket[In}[1]:= ]\]

This writes a packet representing text to enter in the subsidiary *Mathematica* process.

\[\text{In}[3]:= \text{LinkWrite}[\text{link}, \text{EnterTextPacket["10"]}]\]

Here is a packet representing the output prompt from the subsidiary *Mathematica* process.

\[\text{In}[4]:= \text{LinkRead}[\text{link}]\]
\[\text{Out}[4]= \text{OutputNamePacket[Out}[1]= ]\]
And here is the actual result from the computation.

\begin{verbatim}
In[5]:= LinkRead[link]
\end{verbatim}

The basic way that the various different objects involved in a Mathematica session are kept organized is by using MathLink packets. A MathLink packet is simply an expression with a definite head that indicates its role or meaning.

<table>
<thead>
<tr>
<th>Basic packets used in Mathematica sessions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnterTextPacket[&quot;input&quot;]</td>
</tr>
<tr>
<td>ReturnTextPacket[&quot;output&quot;]</td>
</tr>
<tr>
<td>InputNamePacket[&quot;name&quot;]</td>
</tr>
<tr>
<td>OutputNamePacket[&quot;name&quot;]</td>
</tr>
</tbody>
</table>

The fact that LinkRead returns an InputNamePacket indicates that the subsidiary Mathematica is now ready for new input.

\begin{verbatim}
In[6]:= LinkRead[link]
Out[6]= InputNamePacket[In[2]:= ]
\end{verbatim}

This enters two Print commands as input.

\begin{verbatim}
In[7]:= LinkWrite[link, EnterTextPacket["Print[a]; Print[b];"]]
\end{verbatim}

Here is the text from the first Print.

\begin{verbatim}
In[8]:= LinkRead[link]
Out[8]= TextPacket[a]
\end{verbatim}

And here is the text from the second Print.

\begin{verbatim}
In[9]:= LinkRead[link]
\end{verbatim}

No output line is generated, so the new packet is an InputNamePacket.

\begin{verbatim}
In[10]:= LinkRead[link]
Out[10]= InputNamePacket[In[3]:= ]
\end{verbatim}
Some additional packets generated in *Mathematica* sessions.

If you enter input to *Mathematica* using `EnterTextPacket["input"]`, then *Mathematica* will automatically generate a string version of your output, and will respond with `ReturnTextPacket["output"]`. But if you instead enter input using `EnterExpressionPacket[expr]` then *Mathematica* will respond with `ReturnExpressionPacket[expr]` and will not turn your output into a string.

Packets for representing input and output lines using expressions.

This enters an expression into the subsidiary *Mathematica* session without evaluating it.

\[
\text{In[11]} := \text{LinkWrite[link, Unevaluated[EnterExpressionPacket[Factor[x^6 - 1]]]]}
\]

Here are the next 3 packets that come back from the subsidiary *Mathematica* session.

\[
\text{In[12]} := \text{Table[LinkRead[link], \{3\}]}
\]

\[
\text{Out[12]} = \{\text{OutputNamePacket[Out[3]}]=, \text{ReturnExpressionPacket[(-1 + x) (1 + x) (1 - x - x^2) (1 + x + x^2)], InputNamePacket[In[4]}]=\}
\]

InputNamePacket and OutputNamePacket packets are often convenient for making it possible to tell the current state of a subsidiary *Mathematica* session. But you can suppress the generation of these packets by calling the subsidiary *Mathematica* session with a string such as "math -mathlink -batchoutput".

Even if you suppress the explicit generation of InputNamePacket and OutputNamePacket packets, *Mathematica* will still process any input that you give with `EnterTextPacket` or `EnterExpressionPacket` as if you were entering an input line. This means for example that *Mathematica* will call `$Pre` and `$Post`, and will assign values to `In[$Line]` and `Out[$Line]`. 
EvaluatePacket[expr]  
an expression to be sent purely for evaluation
ReturnPacket[expr]  
an expression returned from an evaluation

Evaluating expressions without explicit input and output lines.

This sends an EvaluatePacket. The Unevaluated prevents evaluation before the packet is sent.

\[\text{In}[13]:= \text{LinkWrite}[\text{link}, \text{Unevaluated}[\text{EvaluatePacket}[10!]]]\]

The result is a pure ReturnPacket.

\[\text{In}[14]:= \text{LinkRead}[\text{link}]\]
\[\text{Out}[14]= \text{ReturnPacket}[3628800]\]

This sends an EvaluatePacket requesting evaluation of Print[x].

\[\text{In}[15]:= \text{LinkWrite}[\text{link}, \text{Unevaluated}[\text{EvaluatePacket}[\text{Print}[x]]]]\]

The first packet to come back is a TextPacket representing text generated by the Print.

\[\text{In}[16]:= \text{LinkRead}[\text{link}]\]
\[\text{Out}[16]= \text{TextPacket}[x]\]

After that, the actual result of the Print is returned.

\[\text{In}[17]:= \text{LinkRead}[\text{link}]\]
\[\text{Out}[17]= \text{ReturnPacket}[\text{Null}]\]

In most cases, it is reasonable to assume that sending an EvaluatePacket to Mathematica will simply cause Mathematica to do a computation and to return various other packets, ending with a ReturnPacket. However, if the computation involves a function like Input, then Mathematica will have to request additional input before it can proceed with the computation.

This sends a packet whose evaluation involves an Input function.

\[\text{In}[18]:= \text{LinkWrite}[\text{link}, \text{Unevaluated}[\text{EvaluatePacket}[2 + \text{Input}["data ="]]]]\]

What comes back is an InputPacket which indicates that further input is required.

\[\text{In}[19]:= \text{LinkRead}[\text{link}]\]
\[\text{Out}[19]= \text{InputPacket}[\text{data =}]\]
There is nothing more to be read on the link at this point.

```mathematica
In[20]:= LinkReadyQ[link]
Out[20]= False
```

This enters more input.

```mathematica
In[21]:= LinkWrite[link, EnterTextPacket["x + y"]]
```

Now the `Input` function can be evaluated, and a `ReturnPacket` is generated.

```mathematica
In[22]:= LinkRead[link]
Out[22]= ReturnPacket[2 + x + y]
```

| LinkInterrupt[link] | send an interrupt to a *MathLink*-compatible program |

Interrupting a *MathLink*-compatible program.

This sends a very time-consuming calculation to the subsidiary process.

```mathematica
In[23]:= LinkWrite[link, EnterTextPacket["FactorInteger[2^777-1]"]]
```

The calculation is still going on.

```mathematica
In[24]:= LinkReadyQ[link]
Out[24]= False
```

This sends an interrupt.

```mathematica
In[25]:= LinkInterrupt[link]
```

Now the subsidiary process has stopped, and is sending back an interrupt menu.

```mathematica
In[26]:= LinkRead[link]
Out[26]= MenuPacket[1, Interrupt> ]
```

This closes the link.

```mathematica
In[27]:= LinkClose[link]
```
Two-Way Communication with External Programs

When you install a MathLink-compatible external program using Install, the program is set up to behave somewhat like a simplified Mathematica kernel. Every time you call a function in the external program, a CallPacket is sent to the program, and the program responds by sending back a result wrapped in a ReturnPacket.

This installs an external program, returning the LinkObject used for the connection to that program.

\[
\text{In[1]} := \text{link} = \text{Install["bitsprog"]}
\]
\[
\text{Out[1]} = \text{LinkObject[bitsprog, 4, 4]}
\]

The function ExternalCall sends a CallPacket to the external program.

\[
\text{In[2]} := \text{?bits}
\]
\[
\text{Global`bits}
\]
\[
\text{bits[i_Integer]} :=
\]
\[
\text{ExternalCall[LinkObject[bitsprog, 4, 4], \text{CallPacket[0, \{i\}]}}]
\]

You can send the CallPacket explicitly using LinkWrite. The first argument of the CallPacket specifies which function in the external program to call.

\[
\text{In[3]} := \text{LinkWrite[link, \text{CallPacket[0, \{67\}]}}]
\]

Here is the response to the CallPacket from the external program.

\[
\text{In[4]} := \text{LinkRead[link]}
\]
\[
\text{Out[4]} = \{1, 1, 0, 0, 0, 0, 1\}
\]

If you use Install several times on a single external program, Mathematica will open several MathLink connections to the program. Each connection will however always correspond to a unique LinkObject.

<table>
<thead>
<tr>
<th>$\text{CurrentLink}</th>
<th>\text{the MathLink connection to the external program currently being run}</th>
</tr>
</thead>
</table>

Identifying different instances of a single external program.

:Begin:
:Function: addto
This gives $CurrentLink as an argument to addto.

:Pattern: addto($CurrentLink, n_Integer)

:Arguments: {n}
:ArgumentTypes: {Integer}
:ReturnType: Integer
:End:

This zeros the global variable counter every time the program is started.

```plaintext
int counter = 0;
```

```plaintext
int addto(int n) {
    counter += n;
    return counter;
}
```

This installs one instance of the external program containing addto.

```
In[5]:= ct1 = Install["addtoprog"]
```

This installs another instance.

```
In[6]:= ct2 = Install["addtoprog"]
Out[6]= LinkObject[addtoprog, 6, 6]
```

This adds 10 to the counter in the first instance of the external program.

```
In[7]:= addto[ct1, 10]
Out[7]= 10
```

This adds 15 to the counter in the second instance of the external program.

```
In[8]:= addto[ct2, 15]
Out[8]= 15
```

This operates on the first instance of the program again.

```
In[9]:= addto[ct1, 20]
Out[9]= 30
```
If an external program maintains information about its state then you can use different instances of the program to represent different states. $CurrentLink then provides a way to refer to each instance of the program.

The value of $CurrentLink is temporarily set every time a particular instance of the program is called, as well as when each instance of the program is first installed.

<table>
<thead>
<tr>
<th>MLEvaluateString (stdlib,&quot;string&quot;)</th>
<th>send input to Mathematica but return no results</th>
</tr>
</thead>
</table>

Sending a string for evaluation by Mathematica.

The two-way nature of MathLink connections allows you not only to have Mathematica call an external program, but also to have that external program call back to Mathematica.

In the simplest case, you can use the MathLink function MLEvaluateString() to send a string to Mathematica. Mathematica will evaluate this string, producing whatever effects the string specifies, but it will not return any results from the evaluation back to the external program.

To get results back you need explicitly to send an EvaluatePacket to Mathematica, and then read the contents of the ReturnPacket that comes back.

```c
This starts an EvaluatePacket.
MLEvaluateString (stdlib,"string")
```

```c
This constructs the expression Factorial[7] or 7!.
MLEvaluateString (stdlib, "Factorial", 1);
MLEvaluateInteger32 (stdlib, 7);
```

```c
This specifies that the packet you are constructing is finished.
MLEvaluateString (stdlib);
```

```c
This checks the ReturnPacket that comes back.
MLEvaluateString (stdlib, "ReturnPacket", &n);
```

```c
This extracts the integer result for 7! from the packet.
MLEvaluateInteger32 (stdlib, &ans);
```
Sending a packet to *Mathematica*.

When you can send *Mathematica* an `EvaluatePacket[input]`, it may in general produce many packets in response, but the final packet should be `ReturnPacket[output]`. "Manipulating Expressions in External Programs" will discuss how to handle sequences of packets and expressions whose structure you do not know in advance.

## Running Programs on Remote Computers

*MathLink* allows you to call an external program from within *Mathematica* even when that program is running on a remote computer. Typically, you need to start the program directly from the operating system on the remote computer. But then you can connect to it using commands within your *Mathematica* session.

This starts the program `fprog` and tells it to create a new link.

`fprog -linkcreate -linkprotocol TCPIP`

The program responds with the specification of the link it has created.

`Link created on: 2976@toad.wolfram.com, 2977@toad.wolfram.com`

This connects to the link that has been created.

`Install[LinkConnect["2976@toad.wolfram.com, 2977@toad.wolfram.com", LinkProtocol -> "TCPIP"]]

This now executes code in the external program on toad.wolfram.com.

`f[16]`

The result is `561243`.
External programs that are created using `mcc` or `mprep` always contain the code that is needed to set up *MathLink* connections. If you start such programs directly from your operating system, they will prompt you to specify what kind of connection you want. Alternatively, if your operating system supports it, you can also give this information as a command-line argument to the external program.

```
prog -linkcreate -linkprotocol TCPIP
Install[LinkConnect[
  "port1@host, port2@host",
  LinkProtocol->"TCP/IP"]]
```

Running an external program on a remote computer.

## Running External Programs under a Debugger

*MathLink* allows you to run external programs under whatever debugger is provided in your software environment.

*MathLink*-compatible programs are typically set up to take arguments, usually on the command line, which specify what *MathLink* connections they should use.

```
In debugger: run -linkcreate -linkprotocol TCPIP
In Mathematica: Install[
  LinkConnect["port",LinkProtocol->"TCP/IP"]]
```

Running an external program under a debugger.

Note that in order to get a version of an external program that can be run under a debugger, you need to compile the program so that the output is suitable for use with your debugger. Unix compilers commonly use `-g` as a command-line argument for producing a debuggable program. See your compiler documentation for specific information on the steps you should take.

Unix debugger

```
Set a breakpoint in the C function `f`.
break f
Breakpoint set: f: line 1
```
Start the external program.

```
run -linkcreate -linkprotocol TCPIP
```

The program responds with what port it is listening on.

```
Link created on: 2981@frog.wolfram.com, 2982@frog.wolfram.com
```

---

**Mathematica session**

This connects to the program running under the debugger.

```
In[1]:= Install[LinkConnect[  
    "2981@frog.wolfram.com, 2982@frog.wolfram.com", LinkProtocol -> "TCPIP"]]
```

```
Out[1]= LinkObject[2981@frog.wolfram.com, 2982@frog.wolfram.com, 1, 1]
```

This calls a function which executes code in the external program.

```
In[2]:= f[16]
```

---

**Unix debugger**

The external program stops at the breakpoint.

```
Breakpoint: f(16)
```

This tells the debugger to continue.

```
continue
```

---

**Mathematica session**

Now \( f \) returns.

```
Out[3]= 561243
```

---

**Manipulating Expressions in External Programs**

*Mathematica* expressions provide a very general way to handle all kinds of data, and you may sometimes want to use such expressions inside your external programs. A language like C, however, offers no direct way to store general *Mathematica* expressions. But it is nevertheless
possible to do this by using the *loopback links* provided by the *MathLink* library. A loopback link is a local *MathLink* connection inside your external program, to which you can write expressions that can later be read back.

```c
MLINK MLLoopbackOpen (stdenv, int*errno)
void MLClose (MLINK link)
int MLTransferExpression (MLINK dest, MLINK src)
```

**Functions for manipulating loopback links.**

This opens a loopback link.

```c
... 
ml = MLLoopbackOpen(stdenv, &errno);
```

This puts the expression `Power [x, 3]` onto the loopback link.

```c
MLPutFunction(ml, "Power", 2); 
MLPutSymbol(ml, "x");
MLPutInteger32(ml, 3);
... 
```

This gets the expression back from the loopback link.

```c
MLGetFunction(ml, &head, &n);
MLGetSymbol(ml, &sname);
MLGetInteger32(ml, &k);
... 
```

This closes the loopback link again.

```c
MLClose(ml);
```

You can use `MLTransferExpression()` to take an expression that you get via `stdlink` from *Mathematica*, and save it in a local loopback link for later processing.

You can also use `MLTransferExpression()` to take an expression that you have built up on a local loopback link, and transfer it back to *Mathematica* via `stdlink`.

This puts `21!` onto a local loopback link.

```c
... 
MLPutFunction(ml, "Factorial", 1);
MLPutInteger32(ml, 21);
```
This sends the head\texttt{FactorInteger} to \textit{Mathematica}.
\begin{verbatim}
MLPutFunction(stdlink, "FactorInteger", 1);
\end{verbatim}

This transfers the \texttt{21!} from the loopback link to \texttt{stdlink}.
\begin{verbatim}
MLTransferExpression(stdlink, ml);
\end{verbatim}

You can put any sequence of expressions onto a loopback link. Usually you get the expressions off the link in the same order as you put them on.

And once you have got an expression off the link it is usually no longer saved. But by using\texttt{MLCreateMark()} you can mark a particular position in a sequence of expressions on a link, forcing \textit{MathLink} to save every expression after the mark so that you can go back to it later.

\begin{verbatim}
MLMARK MLCreateMark (MLINK link)
create a mark at the current position in a sequence of expressions on a link
MLSeekMark (MLINK link, MLMARK mark, int n)
go back to a position \(n\) expressions after the specified mark on a link
MLDestroyMark (MLINK link, MLMARK mark)
destroy a mark in a link
\end{verbatim}

Setting up marks in \textit{MathLink} links.

This puts the integer 45 onto a loopback link.
\begin{verbatim}
MLPutInteger32(ml, 45);
\end{verbatim}

This puts 33 onto the link.
\begin{verbatim}
MLPutInteger32(ml, 33);
\end{verbatim}

And this puts 76.
\begin{verbatim}
MLPutInteger32(ml, 76);
\end{verbatim}

This will read 45 from the link. The 45 will no longer be saved.
\begin{verbatim}
MLGetInteger32(ml, &i);
\end{verbatim}
This creates a mark at the current position on the link.

mark = MLCreateMark(ml);

This will now read 33.
MLGetInteger32(ml, &i);

And this will read 76.
MLGetInteger32(ml, &i);

This goes back to the position of the mark.
MLSeekMark(ml, mark, 0);

Now this will read 33 again.
MLGetInteger32(ml, &i);

It is important to destroy marks when you have finished with them, so no unnecessary expressions will be saved.

MLDestroyMark(ml, mark);

The way the *MathLink* library is implemented, it is very efficient to open and close loopback links, and to create and destroy marks in them. The only point to remember is that as soon as you create a mark on a particular link, *MathLink* will save subsequent expressions that are put on that link, and will go on doing this until the mark is destroyed.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int MLGetNext (MLINK link)</td>
<td>find the type of the next object on a link</td>
</tr>
<tr>
<td>int MLGetArgCount (MLINK link, int*n)</td>
<td>store in <em>n</em> the number of arguments for a function on a link</td>
</tr>
<tr>
<td>int MLGetSymbol (MLINK link, char**name)</td>
<td>get the name of a symbol</td>
</tr>
<tr>
<td>int MLGetInteger32 (MLINK link, int*i)</td>
<td>get a machine integer</td>
</tr>
<tr>
<td>int MLGetReal64 (MLINK link, double*x)</td>
<td>get a machine floating-point number</td>
</tr>
<tr>
<td>int MLGetString (MLINK link, char**string)</td>
<td>get a character string</td>
</tr>
</tbody>
</table>

Functions for getting pieces of expressions from a link.
Constants returned by `MLGetNext()`.

```c
switch(MLGetNext(ml)) {

  This reads a composite function.
  case MLTKFUNC:
    MLGetArgCount(ml, &n);
    recurse for head
    for (i = 0; i < n; i++) {
      recurse for each argument
    }
  }

  This reads a single symbol.
  case MLTKSYM:
    MLGetSymbol(ml, &name);
    ...

  This reads a machine integer.
  case MLTKINT:
    MLGetInteger32(ml, &i);
    ...
}
```

By using `MLGetNext()` it is straightforward to write programs that can read any expression. The way *MathLink* works, the head and arguments of a function appear as successive expressions on the link, which you read one after another.

Note that if you know that the head of a function will be a symbol, then you can use `MLGetFunction()` instead of `MLGetNext()`. In this case, however, you still need to call `MLReleaseSymbol()` to disown the memory used to store the symbol name.
Functions for putting pieces of expressions onto a link.

MLPutNext() specifies types of expressions using constants such as MLTKFUNC from the mathlink.h header file—just like MLGetNext().

**Error and Interrupt Handling**

When you are putting and getting data via MathLink various kinds of errors can occur. Whenever any error occurs, MathLink goes into a completely inactive state, and all MathLink functions you call will return 0 immediately.

Handling errors in MathLink programs.

When you do complicated operations, it is often convenient to check for errors only at the end. If you find that an error occurred, you must then call MLClearError() to activate MathLink again.
Clearing out the remains of a packet.

After an error, it is common to want to discard the remainder of the packet or expression that you are currently processing. You can do this using `MLNewPacket()`.

In some cases, you may want to set it up so that if an error occurs while you are processing particular data, you can then later go back and reprocess the data in a different way. You can do this by calling `MLCreateMark()` to create a mark before you first process the data, and then calling `MLSeekMark()` to seek back to the mark if you need to reprocess the data. You should not forget to call `MLDestroyMark()` when you have finally finished with the data—otherwise `MathLink` will continue to store it.

Abort an external program.

If you interrupt Mathematica while it is in the middle of executing an external function, it will typically give you the opportunity to try to abort the external function. If you choose to do this, what will happen is that the global variable `MLAbort` will be set to 1 inside your external program.

`MathLink` cannot automatically back out of an external function call that has been made. So if you have a function that can take a long time, you should explicitly check `MLAbort` every so often, returning from the function if you find that the variable has been set.

## Running Mathematica from Within an External Program

To run Mathematica from within an external program requires making use of many general features of `MathLink`. The first issue is how to establish a `MathLink` connection to Mathematica.

When you use `MathLink` templates to create external programs that can be called from `Mathematica`, source code to establish a `MathLink` connection is automatically generated, and all you have to do in your external program is to call `MLMain(argc, argv)`. But in general you need to call several functions to establish a `MathLink` connection.
Opening and closing MathLink connections.

Include the standard MathLink header file.

```
#include "mathlink.h"
```

```
int main(int argc, char *argv[]) {
    MLENV env;
    MLINK link;
    int errno;

    This initializes MathLink library functions.
    env = MLInitialize();

    This opens a MathLink connection, using the same arguments as were passed to the main program.
    link = MLOpenArgcArgv(env, argc, argv, &errno);

    This activates the connection, waiting for the other program to respond.
    MLActivate(link);
    ...
}
```
Often the `argv` that you pass to `MLOpenArgcArgv()` will come directly from the `argv` that is passed to `main()` when your whole program is started.

The elements in the `argv` array are character strings which mirror the arguments and options used in the *Mathematica* functions `LinkLaunch`, `LinkCreate` and `LinkConnect`.  

```
"-linklaunch"          operate like `LinkLaunch["name"]`
"-linkcreate"          operate like `LinkCreate["name"]`
"-linkconnect"         operate like `LinkConnect["name"]`
"-linkname", "name"    give the name to use
"-linkprotocol", "protocol"  give the link protocol to use (TCPIP, Pipes, etc.)
```

Possible elements of the `argv` array passed to `MLOpenArgcArgv()`.

As an alternative to `MLOpenArgcArgv()` you can use `MLOpenString()`, which takes parameters concatenated into a single character string with spaces in between.

Once you have successfully opened a *MathLink* connection to the *Mathematica* kernel, you can then use standard *MathLink* functions to exchange data with it.

```
int MLEndPacket (MLINK link)  indicate the end of a packet
int MLNextPacket (MLINK link) find the head of the next packet
int MLNewPacket (MLINK link)  skip to the end of the current packet
```

Functions often used in communicating with the *Mathematica* kernel.

Once you have sent all the pieces of a packet using `MLPutFunction()` etc., *MathLink* requires you to call `MLEndPacket()` to ensure synchronization and consistency.

One of the main issues in writing an external program which communicates directly with the *Mathematica* kernel is handling all the various kinds of packets that the kernel can generate.

The function `MLNextPacket()` finds the head of the next packet that comes from the kernel, and returns a constant that indicates the type of the packet.
Mathematica packet | constant | result from a computation
---|---|---
ReturnPacket[expr] | RETURNPKT | textual form of a result
ReturnTextPacket["string"] | RETURNTEXTPKT | name of an input line
InputNamePacket["name"] | INPUTNAMEPKT | name of an output line
OutputNamePacket["name"] | OUTPUTNAMEPKT | textual output from functions like Print
TextPacket["string"] | TEXTPKT | name of a message generated by Mathematica
MessagePacket[symb,"tag"] | MESSAGEPKT | request for a response to an Input function
InputPacket["prompt","tag"] | INPUTPKT | request for a call to an external function
CallPacket[i,list] | CALLPKT |

Some packets recognized by MLNextPacket().

This keeps on reading data from a link, discarding it until an error or a ReturnPacket is found.

```c
while ((p = MLNextPacket(link)) && p != RETURNPKT)
    MLNewPacket(link);
```

If you want to write a complete front end to Mathematica, you will need to handle all of the possible types of packets that the kernel can generate. Typically you can do this by setting up an appropriate switch on the value returned by MLNextPacket().

The MathLink Developer Kit contains sample source code for several simple but complete front ends.

```
int MLReady (MLINK link)
    test whether there is data waiting to be read on a link

int MLReadyParallel (MENV e, MLINK *links, int n, mlt imeval t)
    test in parallel whether there is data to be read from a list of links

int MLFlush (MLINK link)
    flush out buffers containing data waiting to be sent on a link
```

Flow of data on links.

One feature of more sophisticated external programs such as front ends is that they may need to perform operations while they are waiting for data to be sent to them by Mathematica. When you call a standard MathLink library function such as MLNextPacket() your program will normally block until all the data needed by this function is available.

You can avoid blocking by repeatedly calling MLReady(), and only calling functions like MLNextPacket() when MLReady() no longer returns 0. MLReady() is the analog of the Mathematica function LinkReadyQ.
Note that *MathLink* sometimes buffers the data that you tell it to send. To make sure that all necessary data has been sent you should call `MLFlush()`. Only after doing this does it make sense to call `MLReady()` and wait for data to be sent back.

**MathLink Interface 3**

The library now fully supports the Unicode character encoding forms UTF-8, UTF-16, and UTF-32. Use the following new API functions to put or get Unicode characters to or from a link.

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLPutUTF8String()</td>
<td>MLGetUTF8String()</td>
</tr>
<tr>
<td>MLPutUTF16String()</td>
<td>MLGetUTF16String()</td>
</tr>
<tr>
<td>MLPutUTF32String()</td>
<td>MLGetUTF32String()</td>
</tr>
<tr>
<td>MLPutUTF8Symbol()</td>
<td>MLGetUTF8Symbol()</td>
</tr>
<tr>
<td>MLPutUTF16Symbol()</td>
<td>MLGetUTF16Symbol()</td>
</tr>
<tr>
<td>MLPutUTF32Symbol()</td>
<td>MLGetUTF32Symbol()</td>
</tr>
<tr>
<td>MLReleaseUTF8String()</td>
<td>MLReleaseUTF8Symbol()</td>
</tr>
<tr>
<td>MLReleaseUTF16String()</td>
<td>MLReleaseUTF16Symbol()</td>
</tr>
<tr>
<td>MLReleaseUTF32String()</td>
<td>MLReleaseUTF32Symbol()</td>
</tr>
</tbody>
</table>

The *MathLink* library header file `mathlink.h` no longer contains obsolete platform support sections such as those defined by `MACINTOSH_MATHLINK` or `OS2_MATHLINK`. `MACINTOSH_MATHLINK` definitions referred to Mac-OS 9 and earlier. `DARWIN_MATHLINK` contains all platform-specific definitions for Mac OS X.

All uses of special alternative names for common C types have been removed from the API. The *MathLink* header file `mathlink.h` still contains versions of the API functions with these types for use with Interface 2 and older programs.

<table>
<thead>
<tr>
<th>Previous <em>MathLink</em> type</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>uchar_ct</td>
<td>unsigned char</td>
</tr>
<tr>
<td>ucharp_ct</td>
<td>unsigned char *</td>
</tr>
<tr>
<td>ucharpp_ct</td>
<td>unsigned char **</td>
</tr>
<tr>
<td>ucharppp_ct</td>
<td>unsigned char ***</td>
</tr>
<tr>
<td>ushort_ct</td>
<td>unsigned short</td>
</tr>
<tr>
<td>ushortp_ct</td>
<td>unsigned short *</td>
</tr>
<tr>
<td>ushortpp_ct</td>
<td>unsigned short **</td>
</tr>
<tr>
<td>ushortppp_ct</td>
<td>unsigned short ***</td>
</tr>
<tr>
<td>uint_ct</td>
<td>unsigned int</td>
</tr>
<tr>
<td>uintp_ct</td>
<td>unsigned int *</td>
</tr>
<tr>
<td>uintpp_ct</td>
<td>unsigned int **</td>
</tr>
</tbody>
</table>
The memory allocator/deallocator functions passed to the library using MLSetAllocParameter() now must be thread-safe.

API functions that previously took a MLParametersPointer type as an argument or returned a MLParametersPointer type now instead take or return a char * type.

API functions that take as an argument or return a mlapi_result type now take or return type int.

API functions that take as an argument or return a mlapi_error type now take or return type int.

API functions that take as an argument or return a mlapi_token type now take or return type int.

API functions that take as an argument or return a mlapi_packet type now take or return type int.

API functions that take as an argument or return a MLPointer type now take or return void *.
The MLOpen* functions previously took type `long *` for the error variable but now take type `int *`.

The header file `mathlink.h` now contains several new error definitions related to the Unicode character encoding forms.

<table>
<thead>
<tr>
<th>Error code</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLEPDATABAD</td>
<td><code>MathLink</code> encountered invalid character data in given character encoding</td>
</tr>
<tr>
<td>MLEPSCONVERT</td>
<td>Unable to convert from given character encoding to <code>MathLink</code> encoding</td>
</tr>
<tr>
<td>MLEGSCONVERT</td>
<td>Unable to convert from <code>MathLink</code> encoding to requested character encoding</td>
</tr>
</tbody>
</table>

`MLPutMessage()` and `MLGetMessage()` now use types `int` and `int *` respectively instead of the `dev_message` and `dev_message *` types.

`MLSeekMark()` and `MLSeekToMark()` now use type `int` rather than type `long` for the expression index.

The functions in the following table took `long` types for some arguments; they now take `int`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLGetRawData()</td>
<td>MLGetData()</td>
</tr>
<tr>
<td>MLGetArgCount()</td>
<td>MLGetRawArgCount()</td>
</tr>
<tr>
<td>MLBytesToGet()</td>
<td>MLRawBytesToGet()</td>
</tr>
<tr>
<td>MExpressionsToGet()</td>
<td>MLTakeLast()</td>
</tr>
<tr>
<td>MLPutRawSize()</td>
<td>MLPutRawData()</td>
</tr>
<tr>
<td>MLPutArgCount()</td>
<td>MLPutComposite()</td>
</tr>
<tr>
<td>MLBytesToPut()</td>
<td></td>
</tr>
</tbody>
</table>

`MLGetReal()` is now an actual API function rather than a `#define` alias to `MLGetDouble()`. `MLGetReal()` still has the same functionality as `MLGetDouble()`.

`MLActivate()` is now an actual API function rather than a `#define` alias to `MLConnect()`. `MLActivate()` still has the same functionality as `MLConnect()`.

The functions in column one listed below are now obsolete. New programs should use the functions listed in column two for replacement functionality.

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLCheckFunction()</td>
<td>MLTestHead()</td>
</tr>
<tr>
<td>MLCheckFunctionWithArg()</td>
<td>MLTestHead()</td>
</tr>
<tr>
<td>MLGetShortInteger()</td>
<td>MLGetInteger16()</td>
</tr>
<tr>
<td>MLGetInteger()</td>
<td>MLGetInteger32()</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MLGetLongInteger()</td>
<td>MLGetInteger64() for 64-bit integers or MLGetInteger32() for 32-bit integers</td>
</tr>
<tr>
<td>MLGetFloat()</td>
<td>MLGetReal32()</td>
</tr>
<tr>
<td>MLGetDouble()</td>
<td>MLGetReal64()</td>
</tr>
<tr>
<td>MLGetLongDouble()</td>
<td>MLGetReal128()</td>
</tr>
<tr>
<td>MLGetShortIntegerArrayData()</td>
<td>MLGetInteger16ArrayData()</td>
</tr>
<tr>
<td>MLGetIntegerArrayData()</td>
<td>MLGetInteger32ArrayData()</td>
</tr>
<tr>
<td>MLGetLongIntegerArrayData()</td>
<td>MLGetInteger64ArrayData() for 64-bit integers or MLGetInteger32ArrayData() for 32-bit integers</td>
</tr>
<tr>
<td>MLGetFloatArrayData()</td>
<td>MLGetReal32ArrayData()</td>
</tr>
<tr>
<td>MLGetDoubleArrayData()</td>
<td>MLGetReal64ArrayData()</td>
</tr>
<tr>
<td>MLGetLongDoubleArrayData()</td>
<td>MLGetReal128ArrayData()</td>
</tr>
<tr>
<td>MLGetShortIntegerArray()</td>
<td>MLGetInteger16Array()</td>
</tr>
<tr>
<td>MLGetInteger()</td>
<td>MLGetInteger32Array()</td>
</tr>
<tr>
<td>MLGetLongInteger()</td>
<td>MLGetInteger64Array() for 64-bit integers or MLGetInteger32Array() for 32-bit integers</td>
</tr>
<tr>
<td>MLGetFloatArray()</td>
<td>MLGetReal32Array()</td>
</tr>
<tr>
<td>MLGetDoubleArray()</td>
<td>MLGetReal64Array()</td>
</tr>
<tr>
<td>MLGetLongDoubleArray()</td>
<td>MLGetReal128Array()</td>
</tr>
<tr>
<td>MLDisownShortIntegerArray()</td>
<td>MLReleaseInteger16Array()</td>
</tr>
<tr>
<td>MLDisownIntegerArray()</td>
<td>MLReleaseInteger32Array()</td>
</tr>
<tr>
<td>MLDisownLongIntegerArray()</td>
<td>MLReleaseInteger64Array() for 64-bit integers or MLReleaseInteger32Array() for 32-bit integers</td>
</tr>
<tr>
<td>MLDisownFloatArray()</td>
<td>MLReleaseReal32Array()</td>
</tr>
<tr>
<td>MLDisownDoubleArray()</td>
<td>MLReleaseReal64Array()</td>
</tr>
<tr>
<td>MLDisownLongDoubleArray()</td>
<td>MLReleaseReal128Array()</td>
</tr>
<tr>
<td>MLGetIntegerList()</td>
<td>MLGetInteger32List()</td>
</tr>
<tr>
<td>MLGetRealList()</td>
<td>MLGetReal64List()</td>
</tr>
<tr>
<td>MLDisownIntegerList()</td>
<td>MLReleaseInteger32List()</td>
</tr>
<tr>
<td>MLDisownRealList()</td>
<td>MLReleaseReal64List()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLPutShortInteger()</td>
<td>MLPutInteger16()</td>
</tr>
<tr>
<td>MLPutInteger()</td>
<td>MLPutInteger32()</td>
</tr>
<tr>
<td>MLPutLongInteger()</td>
<td>MLPutInteger64() for 64-bit integers or MLPutInteger32() for 32-bit integers</td>
</tr>
<tr>
<td>MLPutFloat()</td>
<td>MLPutReal32()</td>
</tr>
<tr>
<td>MLPutDouble()</td>
<td>MLPutReal64()</td>
</tr>
<tr>
<td>MLPutLongDouble()</td>
<td>MLPutReal128()</td>
</tr>
<tr>
<td>MLPutShortIntegerArrayData()</td>
<td>MLPutInteger16ArrayData()</td>
</tr>
<tr>
<td>MLPutIntegerArrayData()</td>
<td>MLPutInteger32ArrayData()</td>
</tr>
<tr>
<td>MLPutLongIntegerArrayData()</td>
<td>MLPutInteger64ArrayData() for 64-bit integers or MLPutInteger32ArrayData() for 32-bit integers</td>
</tr>
<tr>
<td>MLPutFloatArrayData()</td>
<td>MLPutReal32ArrayData()</td>
</tr>
<tr>
<td>MLPutDoubleArrayData()</td>
<td>MLPutReal64ArrayData()</td>
</tr>
</tbody>
</table>
Interface 3 changes the default link protocol for linkmode Listen and linkmode Connect links. By default the MathLink library will create "SharedMemory" links for linkmode Listen and linkmode Connect links on all platforms.