DDFLink

User Manual and Reference Guide

Introduction

Features

The DDFLink package provides:

A set of *Mathematica* functions that allow you to:
- Receive real-time streaming tick data in *Mathematica*
- Define your own event handler functions to manage the streaming tick data in *Mathematica*
- Obtain instrument and exchange specifications
- Use Java table grids from *Mathematica* that display live streaming price updates
- *Mathematica* symbol browser and *Mathematica* expression browser

About

DDFLink
Version 1.0
Copyright © 2012 Andreas Lauschke Consulting

Table of Contents

Installation

Requirements

- DDFPlus API. Included in the set of package files.
- *Mathematica* 6 or later. Available from www.wolfram.com.
  - Current version is *Mathematica* 8.0.4.
- Java 7 or later. Available from www.java.com.
  - Current version is Java 7 update 4. Security baseline is Java 7 update 3.
- DDFLink 1.0 or later. Available from www.lauschkeconsulting.com.
  - Current version is DDFLink 1.0.

Mathematica 7 has Java 6 bundled with it, *Mathematica* 6 has Java 5 bundled with it, *Mathematica* 8 has Java 6 bundled with it.

Installation and Configuration

To use the *Mathematica* Package from *Mathematica*:

- Create a directory called DDFLink. It is recommended to place this directory in your user home directory. This is $HomeDirectory, not $UserBaseDirectory. Place all files from the DDFLink distribution (4 files) in the DDFLink directory.

- Edit the file ddflinkconfig.m in the DDFLink directory with a text editor to set the variables ddflinklocation, ddflinkconnectionmode, ddflinkusername, ddflinkpassword, ddflinkserver.
  - ddflinklocation is the location of the DDFLink directory from step 1 above.
  - ddflinkconnectionmode must be TCP if you use the TCP connection from ddfplus and UDP if you use the UDP connection from ddfplus. Default is TCP, which is the recommended mode for normal operation.
  - ddflinkusername is the username you set up with ddfplus (string).
  - ddflinkpassword is the password you set up with ddfplus (string).
  - ddflinkserver is the servername ddfplus provides you with (string).

- If you want to use a JLink.jar that is different from the default file (<MathematicaInstall>/SystemFiles/Links/JLink/JLink.jar), set the variable jlinklocation to the full-path location of the file JLink.jar.

- If you want to modify the Java runtime arguments used with DDFLink, set the variable commandline in the file ddflinkconfig.m to the string representation of the Java runtime arguments that you may want to modify from the default values, e. g. to specify a particular Java runtime you want to use (not the one that is included in the *Mathematica* distribution) or to set additional runtime options (memory settings, JIT-compilation, etc.). Assigning to commandline will automatically reinstall the Java runtime. The default is to not use a special command line and not to reinstall the Java runtime (i. e. the variable assignment is missing to in the file).

Look-and-Feel Configuration

DDFLink supports all standard and most third-party add-on/plug-in look-and-feels. To install a third-party look-and-feel, download the .jar file from the third-party supplier, place it in the DDFLink directory, identify the name of the entry point class, and place a line

laf="<full name space class name to entry point class>"

in a text file lookandfeel.m in the DDFLink directory. Example:

laf="com.jgoodies.looks.plastic.Plastic3DLookAndFeel"

This will enable the look-and-feel immediately for the *Mathematica* package. To also use the look-and-feel from Calc (through the OpenOffice plug-in/extension), add the .jar file to the OpenOffice class path, as described in steps 3 and 4 in the previous section.

If no file lookandfeel.m exists in the DDFLink directory or laf is assigned the string "default", DDFLink will use the system's default look-and-feel.

For a good overview of various free and commercial look-and-feels, visit
http://www.javootoo.com

The following Look-and-Feels have been tested to work with DDFLink:

- JGoodies Plastic3D
- JGoodies PlasticXP
- JGoodies Plastic
- JGoodies Windows
- Office2003 (Windows only)
- OfficeXP (Windows only)
- VisualStudio2005 (Windows only)
- Nimrod
- Fh
- Tiny
- Tonic
- Tonic Slim
- Infonode
- Napkin
- SquareNess
- EaSynth

which can be downloaded from javootoo.

The Alloy look-and-feel has also been tested to work with DDFLink, which can be obtained from http://lookandfeel.incors.com/.

Working in *Mathematica*

Getting Started

Loading the Package

To start using the link from *Mathematica*, you must first load the DDFLink package.

With *Mathematica* version 6 and above:

Get@ToFileName[{$HomeDirectory, "DDFLink"}, "DDFLink.m"]

With *Mathematica* version 7 and above:

Get@FileNameJoin[{$HomeDirectory, "DDFLink", "DDFLink.m"}]

If you installed DDFLink in a different location (not in the user home directory), you need to load the package file DDFLink.m from that location.

Setting the List of Symbols to Receive Live Tick Data for ("Watch List")

Next you have to set the "watch list", unless you don't want to receive live tick data with DDFLink. You do this with DDFSetSymbols:

DDFSetSymbols["DELL,YHOO,ESU0,QQQQ,E6M0"];

subscribes to the DELL and YHOO and ESU0 and QQQQ and E6M0 symbols. You will receive a message for every tick in these symbols now. What to do with these messages (most likely you will want to extract the market data from the XML expression) is done in the event handler functions, see below.

Note that you HAVE to set the symbol list / watch list if you want to receive live streaming market data. When the DDFLink package is loaded, no symbol list is defined. If you want to receive live streaming data, it is recommended to set the symbol list immediately after loading the package, and to then define the event handler functions (see next step).

Definining the Streaming Event Handler Functions

There are three types of streaming event handler functions: quote data, book quote data, and date/time data. You receive a quote data message whenever ddfplus sends a quote data message for a symbol in your watch list, you receive a book quote data message whenever ddfplus sends a book quote data message for a symbol in your watch list, and you receive a date/time message whenever ddfplus sends a time stamp associated with a tick messsage for a symbol in your watch list. DDFLink provides the functions DDFQuoteFunction, DDFBookQuoteFunction, and DDFDateFunctions to allow you to make definitions for these event handlers.

Simply define:

DDFQuoteFunction[Quote_]:=<*Mathematica* code to handle quote events>

DDFBookQuoteFunction[BookQuote_]:=<*Mathematica* code to handle book quote events>

DDFDateFunction[date_String]:=<*Mathematica* code to handle time stamp events>

in your *Mathematica* program or package.

NOTE:
Bear in mind that the *Mathematica* front-end is not available when JLink "owns" the kernel and interacts with the Java runtime. Whenever a Java event is received, JLink links the kernel with the Java runtime, blocking the *Mathematica* front-end. However, as soon as the event is received (microseconds/nanoseconds later), JLink releases the link and the kernel is "owned" by the *Mathematica* front-end again. It would thus be impossible to, for example, update live data in a *Mathematica* front-end (table, graphics, etc.) exactly in the moment of time when the event is received, as the *Mathematica* front-end is blocked during that instant. However, as soon as the event is fully received, JLink releases the link to the Java runtime and re-enables the link between the kernel and the *Mathematica* front-end. Any expression in the *Mathematica* front-end to be updated based on the new event data can NOW be updated, or other front-ends could be used, for example other Java front-ends (with JLink) or .Net front-ends (with NETLink). The appendix shows a few simple *Mathematica* functions to use Java or .Net windows for this purpose. For (very simple) tabular data the utility grid function DDFUtilityGrid included with DDFLink can be used for this purpose, as it is a Java window that is controlled from *Mathematica* and is thus available in the *Mathematica* event handler function when the *Mathematica* front-end is not available.

In practice, the "delay" is entirely negligible, as it only takes a few microseconds or nanoseconds for JLink to switch the links, but it is very important to note that while JLink links the kernel to the Java runtime, the *Mathematica* front-end is completely inaccessible from the kernel.

This affects only the link between the kernel and the *Mathematica* front-end, not the kernel itself. It would, for example, be possible to record all tick data and possibly write it to a file:

DDFQuoteFunction[Quote_]:=Write[stream,Quote];

for the quote event handler function, after a

stream=OpenWrite[<path/file location>];

statement was executed to open a file and a file stream for writing (or OpenAppend to append to a file).

To update expressions created with the *Mathematica* front-end (this will usually be a table or a chart or other *Mathematica* graphics expression) based on new live tick data during the execution of the event handler one could use Java or .Net windows that update live. The appendix of this user guide contains several such functions.

It is also possible to "poll" for this data periodically, for example with the *Mathematica* function AddPeriodical[<expr>,<time interval>], which allows the user to define a task that is to be executed periodically, with <time interval> permitting sub-second time intervals to be specified. However, as Periodicals in *Mathematica* use front-end sharing from the JLink package, the same issue as described above applies with Periodicals.

Another possibility to update an expression created with the *Mathematica* front-end based on new live tick data from the event handler is to use an infinite While loop that updates the expression whenever the number of received events has changed, by storing the value of the DDFLink function DDFQuoteQuote[] in the event handler function.

```
        DDFQuoteFunction[Quote_]:=(storequote=Quote;newquotecount=DDFQuoteCount[]);


        Module[{storequotecount=0},
                While[True,
                        Pause[0.01]; (* wait 10 ms *)
                        If[storequotecount<newquotecount,  (* check if a new quote event was received *)
                        Print@storequote;  (* execute code using the Mathematica front-end, possibly involving
                                        Dynamic[] variables *)
                        storequotecount=newquotecount];  (* store the new quote count *)
                ];
        ];
```

However, this method has the disadvantage that it is running permanently in the evaluating cell in the *Mathematica* front-end and doesn't allow the user the change the code in it while it is executing. It is generally much better to use *Mathematica* code to control Java or .Net windows in the event handler function, which can be changed any time (even when receiving live streaming data – once the event handler function has been updated it will take effect immediately).

DDFLink Contract and Exchange Look-Up Functions

The DDFLink functions DDFContractInfo and DDFExchangeInfo can be used to retrieve information about market symbols and obtain exchange information.

For example,

DDFContractInfo["ESH0"] returns information about the ESH0 contract (month symbol, root symbol, short symbol, symbol, symbol type, and expiration year).

DDFExchangeInfo["N"] returns current exchange information for the NYSE (Advancing Issues, Declining Issues, Advancing Shares, Declining Shares, New Highs, New Lows, Exchange Code)

These can be viewed in a Java Swing-based interactive data viewer that is included in DDFLink:

DDFShowContractInfo["ESH0"] creates a new interactive Swing window with the contract information for ESH0.

DDFShowWatchList[] creates a new interactive Swing window with the symbol list.

DDFShowExchangeInfo["N"] creates a new interactive Swing window with the current market information about the NYSE.

The DDFLink functions that return data in interactive Java Swing windows have "Show" in the function name (DDFShowWatchList, DDFShowContractInfo, and DDFShowExchangeInfo).

Note that for these look-up functions the symbol does not have to be in the watch list. Any valid symbol can be used. The watch list contains only the symbols for which the user subscribes to the live streaming data feed. This is unrelated to the look-up functions.

DDFLink Functions: Returning Mathematica Symbols or Performing Tasks

The DDFLink package provides the following functions:

DDFSetSymbols[s_String]	Sets the symbols you want to retrieve live market data for. ("watch list")
DDFGetSymbols[]	Gets the symbols you want to retrieve live market data for. ("watch list")
DDFTree[expr_]	An interactive *Mathematica* expression browser using expandable/collapsible tree nodes

DDFShowSymbols[]	An interactive *Mathematica* symbol browser/list for all *Mathematica* symbols, including user-defined symbols.
DDFShowUserSymbols[]	An interactive *Mathematica* symbol browser/list for user and package symbols
DDFConnectionMode[]	Returns the current connection mode with ddfplus (either TCP or UDP)
DDFContractInfo[]	Returns contract information for the given symbol
DDFExchangeInfo[]	Returns exchange information for the given exchange symbol
DDFQuoteCount[]	The total number of quote events received during the current session.
DDFBookQuoteCount[]	The total number of book quote events received during the current session.
DDFDateCount[]	The total number of time stamp messages received during the current session.
DDFUtilityGrid[]	A general utility grid that creates a new Java Swing window to display user-specified tabular data in a JTable (n rows, 2 columns).
DDFSetSymbols[]	A function to set (reset) the watch list (for real-time streaming quotes).
DDFShowContractInfo[]	Creates a new Java Swing window showing contract info for the given symbol.
DDFShowExchangeInfo[]	Creates a new Java Swing window showing current exchange information for the given exchange.
DDFShowWatchList[]	Creates a new Java Swing window showing the user's current watch list.

DDFLink Utility Grid

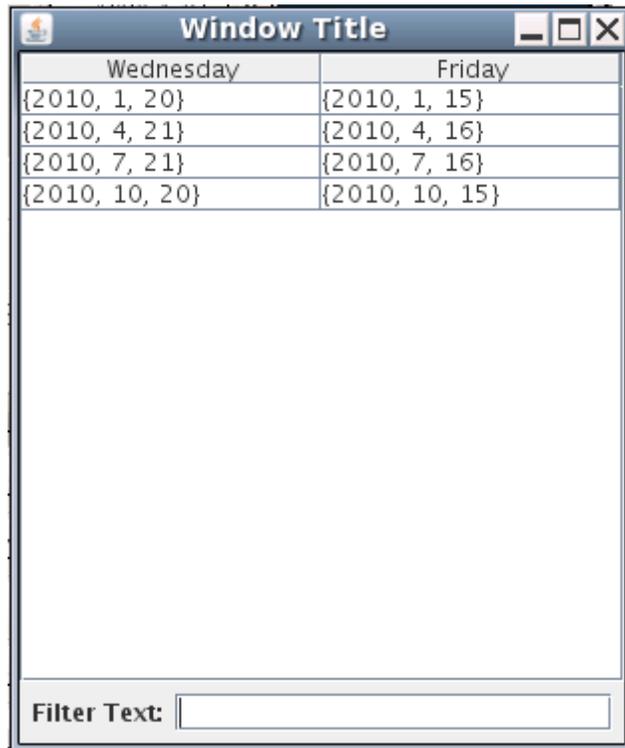DDFLink provides a simple general-purpose Java Swing-based data grid to view tabular data.

DDFUtilityGrid[<2-dim data array>,<window title>,<first column label>,<second column label>]

creates a new Java Swing window and displays the contents of the first argument (a 2-

dimensional array of data, consisting of 2 columns and an arbitrary number of rows), using window title and column labels. For example, the following creates a new Java Swing window showing the dates when the third Wednesday of the month and the third Friday of the month in 2010 are not in the same week:

```
DDFUtilityGrid[
    ToString@# & /@ # & /@
        Select[Table[{NestWhile[DaysPlus[#,1]&,{2010,i,15},DayOfWeek@#
        =!=Wednesday&],NestWhile[DaysPlus[#,1]&,{2010,i,15},DayOfWeek@#
        =!=Friday&]},{i,12}],DaysBetween[#[[1]],#[[2]]]<0&],
    "Window Title","Wednesday","Friday"
]
```


Wednesday	Friday
{2010, 1, 20}	{2010, 1, 15}
{2010, 4, 21}	{2010, 4, 16}
{2010, 7, 21}	{2010, 7, 16}
{2010, 10, 20}	{2010, 10, 15}

Filter Text:

The filter text field allows to select only the matching lines, using Perl 5 regular expression syntax. This happens "on the fly", directly as the user types the regex in the filter text field.

The return type of DDFUtilityGrid is a Java object (a JFrame), so an advanced JLink/Java programmer would be able to further manipulate this Java object, for example extracting data from it (the model object of the JFrame objects contains the table data) or combining it with other Java objects handled by JLink.

Interactive Tree Representation/Inspection of *Mathematica* Expressions

The function DDFTree[] creates a new window containing a tree of the expression using expandable/collapsible tree nodes. At every node the name of the head of the expression at that level is shown. The node can be expanded to display all its nodes or leaves. Only leaves (symbols that are atomic, i. e. AtomQ[] is True) can not be expanded anymore.
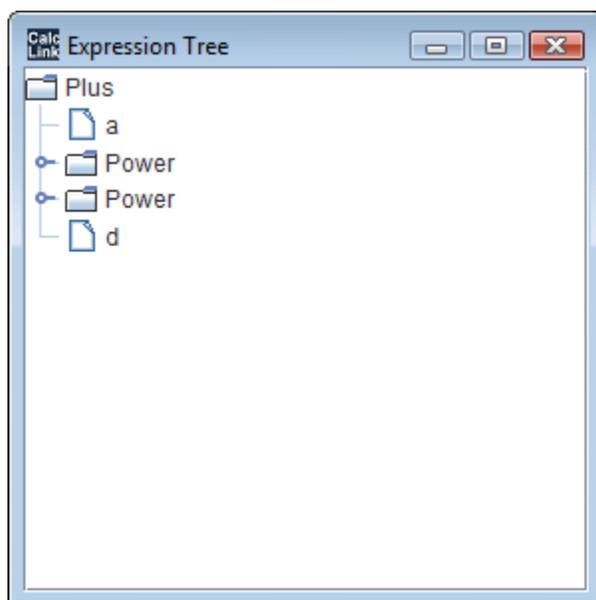
A very simple example involving only leaves under the root node:
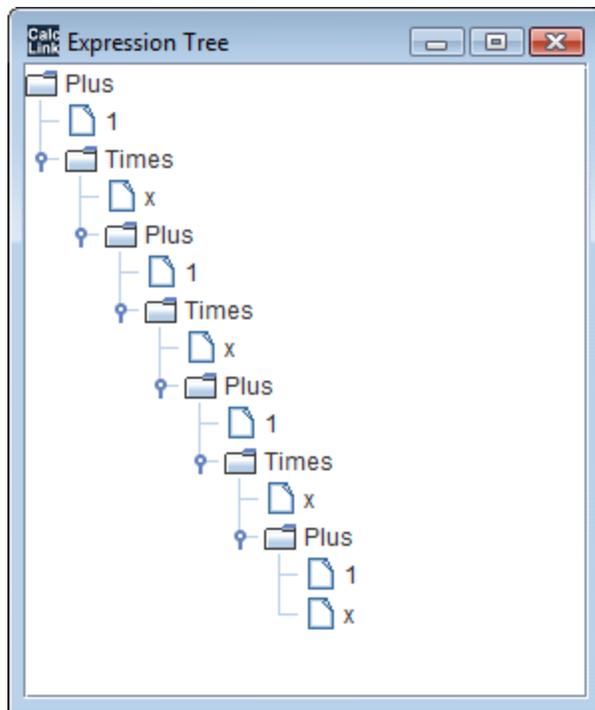
DDFTree[a + b + c + d + Pi]
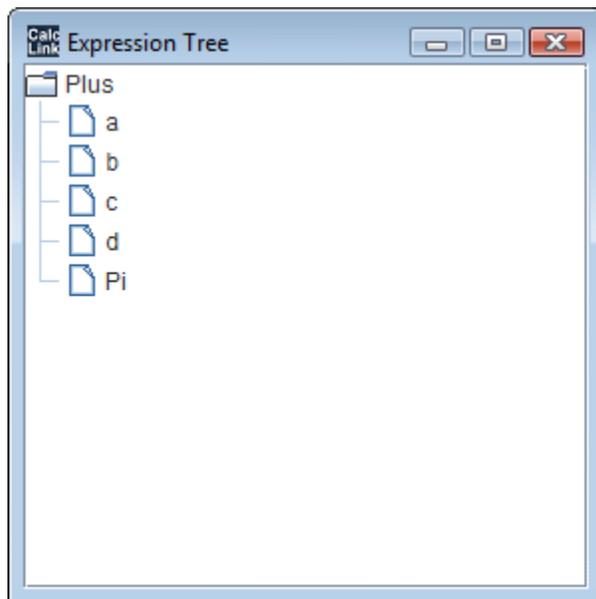
a and d are atomic, b^2 and c^3 are not:

DDFTree[a + b^2 + c^2 + d]

HornerForm[] creates an expression that has two nesting levels per order of the polynomial (minus 1).

DDFTree[HornerForm[1 + x + x^2 + x^3, x]]

Inspect the coefficients of the Taylor series visually:
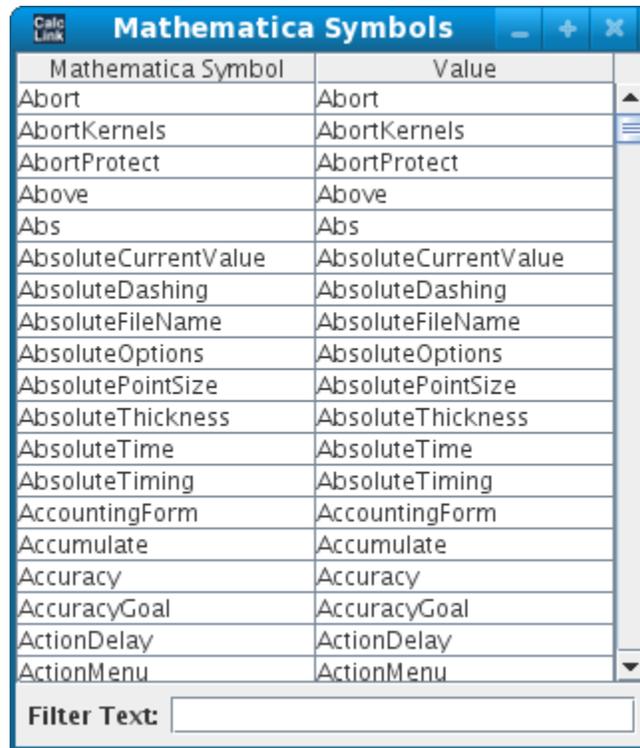
DDFTree[Series[Tan@x, {x, 0, 12}]]
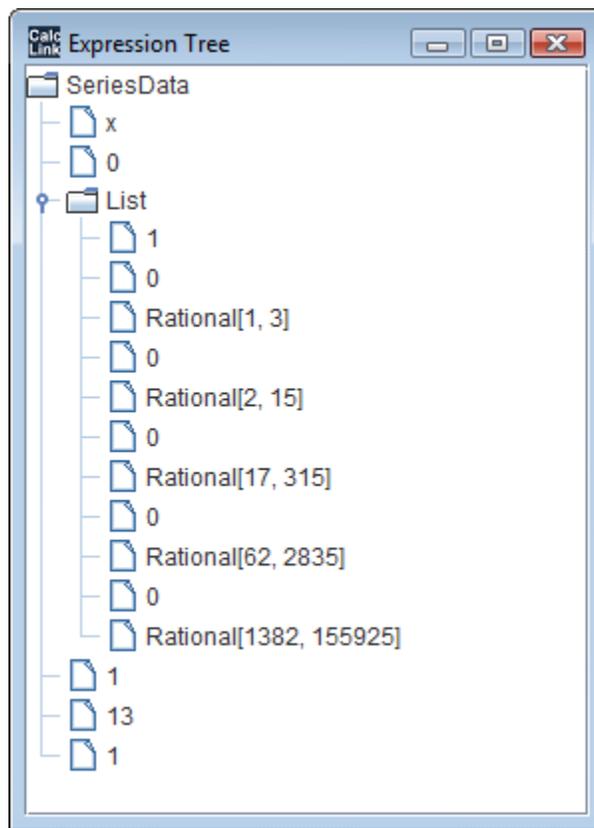
The real usefulness of DDFTree[] becomes evident when used on complex, deeply nested expression structures. Sometimes *Mathematica* expressions can be so complex that it is hard to understand the nested symbol structure, so DDFTree[] makes it possible to "zoom in" on a branch of interest, while leaving others collapsed.

DDFTree[Plot3D[Sin@x Sin@y, {x, -Pi, Pi}, {y, -Pi, Pi}]]

Expression Tree

```
Graphics3D
  GraphicsComplex
    List
    List
      List
        List
          EdgeForm
          GraphicsGroup
            List
              Polygon
              Polygon
              Polygon
        List
        List
        List
        List
      List
        List
          GrayLevel
          Line
        List
        List
        List
        List
    Rule
      VertexNormals
      List
  List
    Rule
    Rule
      BoxRatios
      List
    Rule
    Rule
    Rule
      PlotRangePadding
      List
```

While the *Mathematica* function TreeForm[] shows the expression in a "top-down" fashion in a possibly more "intuitive" and visually appealing form, the DDFLink function DDFTree[] allows for interactive and selective inspection of the branches/leaves of a complex nested expression.

The return type of DDFTree is a Java object, so an advanced JLink/Java programmer would be able to further manipulate this Java object, for example extracting data from it (the model

object of the JFrame objects contains the table data) or combining it with other Java objects handled by JLink.

Mathematica Symbol Browser

DDFLink provides two _Mathematica_ symbol browsers, one showing all symbols, and one showing all user and package symbols (the contents of the Global` context).

At the bottom of the window is a filter field, where you can enter the names of _Mathematica_ symbols (or parts thereof), and the window will automatically filter out the _Mathematica_ symbols that contain the strings or substrings you have entered. This shows a list of all _Mathematica_ symbols containing "Pi".

The *Mathematica* Symbol Browser uses Perl 5 regular expression pattern matching, so you can use \ (backspace character) as an escape character (e. g. "\$..." to see all *Mathematica* symbols beginning with a dollar sign).

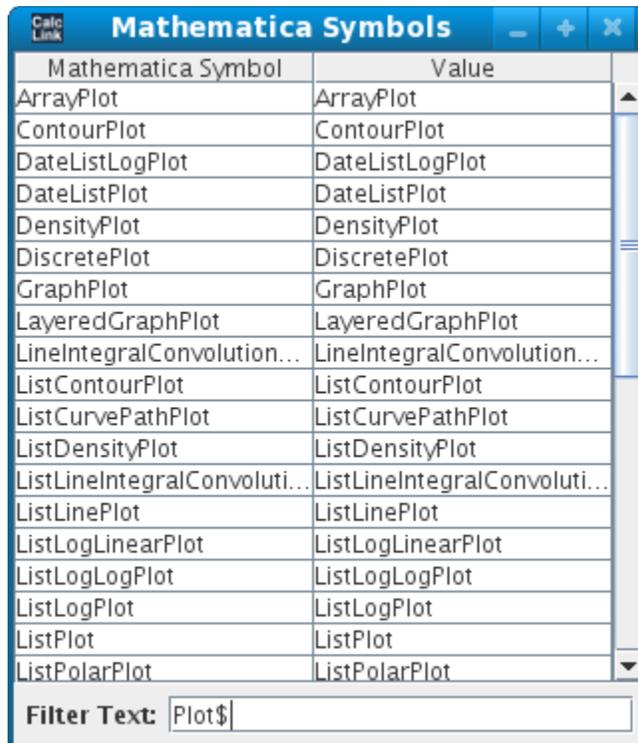The following shows all symbols that begin with "$Version":
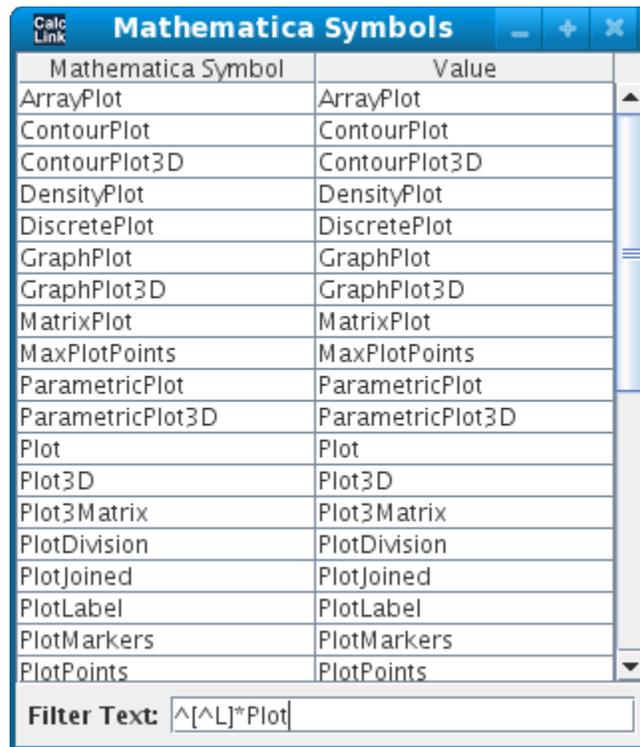
The symbols window also displays its contents on the tooltip, so longer strings in the output can be further inspected with the tooltip (or the window resized).
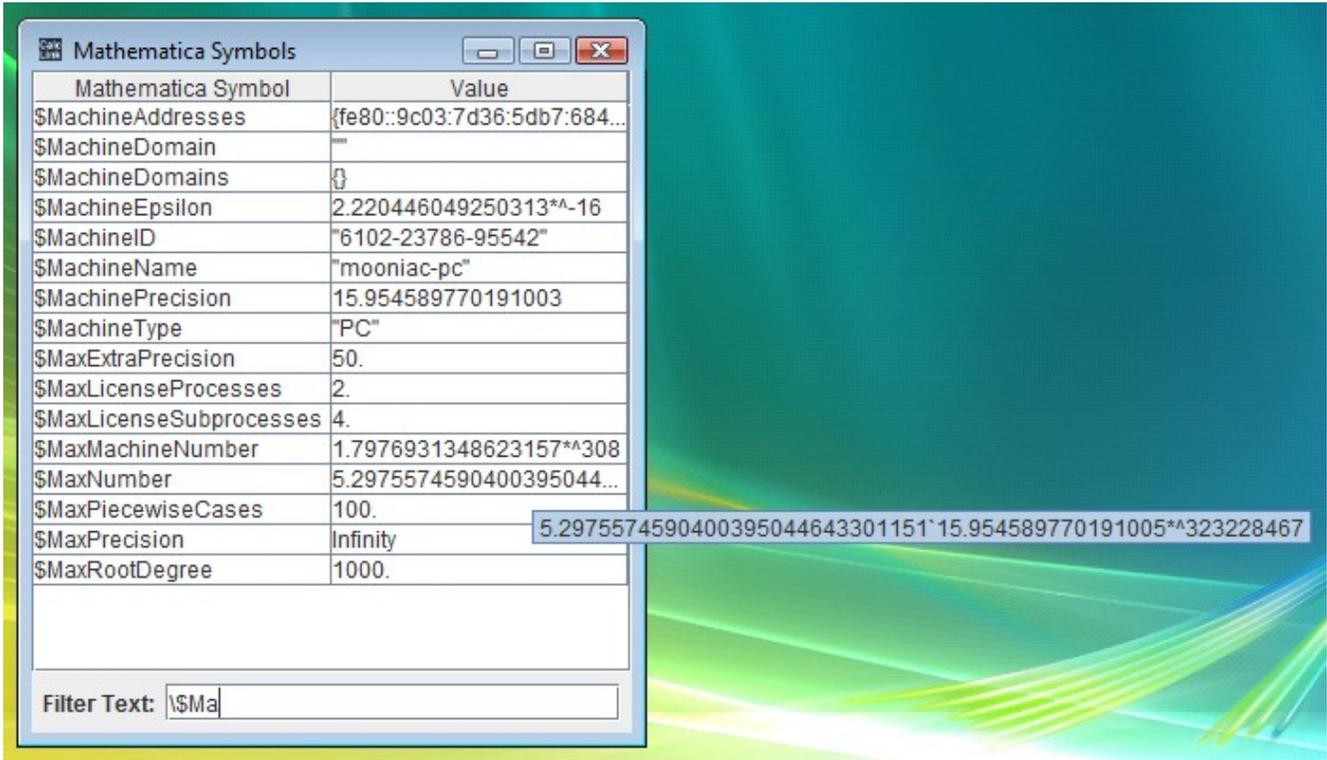
The following shows all symbols that end with "Plot":

The following shows all symbols that contain "Plot" but don't begin with "L":

The symbol browser includes your own symbols as well as the symbols of Mathematica packages you have loaded. For example, if you have made a symbol definition mysymbol=232, it will be included in the symbol list. (The list displayed contains everything included in Names["*"], which includes the symbols from loaded packages.)

The other *Mathematica* symbol browser only displays the contents of the Global` context. This includes all user-defined symbols (unless they were put in their own contexts).

You can launch as many *Mathematica* symbol browsers as you want and filter for symbols in them independently of one another. New symbols browsers HAVE to be launched after new symbols have been created (user-defined or from a *Mathematica* package that was loaded) to make them appear in the displayed list.

Once a symbol browser is displayed the symbol list contained therein can not be changed/updated anymore. This is not considered a drawback because when the user is done with a task in the symbol browser the user will most likely close the window (which also releases the memory space held for the list of over 3600 symbols along with their values in the Java heap), and a new one with an updated symbol list can always be popped up again with a simple click on the push-button "Show All Symbols" on the DDFLink control center pane.

The return type of DDFShowSymbols and DDFShowUserSymbols is a Java object, so an advanced JLink/Java programmer would be able to further manipulate this Java object, for

example extracting data from it (the model object of the JFrame objects contains the table data) or combining it with other Java objects handled by JLink.

DDFLink for Advanced JLink/Java Programming

DDFLink fully exposes its API to the advanced JLink and Java programmer. The link object used by DDFLink is called ddflink. You can inspect its Fields and Methods and constructor and events with the usual JLink functions for these purposes (Fields, Methods, Constructor, Events). That allows you to get to the "bare metal" of the interface if you choose so.

Appendix A

This appendix shows a few Java and .Net functions that can be used in the event handler functions to create new or update existing windows with *Mathematica* expressions (for example tables/grids, or graphics).

The "send" type functions described below should then be in the event handler functions (DDFQuoteFunction[], DDFBookQuoteFunction[], DDFDateFunction[])

Using Java

For the following *Mathematica*/Java functions to work you need the JLink package installed and running. However, with DDFLink this is already happening, DDFLink loads JLink right at the beginning. If you want to use the following functions without DDFLink, you first have to load the JLink package:

Needs["JLink`"];

Then you can use the following functions to create new Java windows and update their contents with new contents. The function javawindow[] creates a new Java window displaying the result of the *Mathematica* expression func passed as first argument. It returns two symbols which are pointers to the corresponding Java objects in the Java runtime, which must be stored in *Mathematica* symbols to be used afterwards with the function sendjavawindow[], or it's not possible to update the contents of that window (these effectively function as "pointers" or "references" or "handles" to the corresponding Java objects). If they are not stored in *Mathematica* symbols, there is no more reference to the Java window or its contents. Use it like

canvas=javawindow[Pi^2];

and when updating the contents of the window, pass them to the function sendjavawindow[]:

canvas=sendjavawindow[canvas, Pi^3];

When you are done and no longer need the Java window, you should dispose of the resources and remove the *Mathematica* references to it:

form@dispose[];
ReleaseJavaObject[canvas];

These are the function definitions for javawindow[] and sendjavawindow[]:

SetAttributes[javawindow, HoldFirst];
SetAttributes[sendjavawindow, HoldFirst];

javawindow[func_, feoption_: True, width_Integer: 300, height_Integer: 300] :=
  JavaBlock[
  frame = JavaNew["com.wolfram.jlink.MathFrame"];
  frame@setLayout[JavaNew["java.awt.BorderLayout"]];
  mathCanvas = JavaNew["com.wolfram.jlink.MathCanvas"];
  frame@add["Center", mathCanvas];
  frame@setSize[width, height];
  frame@layout[];
  mathCanvas@setImageType[MathCanvas`GRAPHICS];
  mathCanvas@setUsesFE[feoption];
  mathCanvas@setMathCommand[ToString[Unevaluated[func], InputForm]];
  JavaShow[frame];
  mathCanvas
  ];

sendjavawindow[mathCanvas_,func_,feoption_:True]:=
JavaBlock[
mathCanvas@setImageType[MathCanvas`GRAPHICS];
mathCanvas@setUsesFE[feoption];
mathCanvas@setMathCommand[ToString[Unevaluated[func],InputForm]];
mathCanvas@recompute[];
mathCanvas
];

Using .Net

You first have to load the NETLink package:

Needs["NETLink`"];

You should do this BEFORE loading the DDFLink package.

Then you can use the following functions to create new .Net windows and update their contents with new contents. The function netwindow[] creates a new .Net window displaying the result of the *Mathematica* expression func passed as first argument. It returns two symbols which are pointers to the corresponding .Net objects in the .Net runtime, which must be stored in *Mathematica* symbols to be used afterwards with the function sendnetwindow[], or it's not possible to update the contents of that window (these effectively function as "pointers" or "references" or "handles" to the corresponding .Net objects). If they are not

stored in *Mathematica* symbols, there is no more reference to the .Net window or its contents. Use it like

{form, box}=netwindow[Pi^2];

and when updating the contents of the window, pass them to the function sendnetwindow[]:

{form, box}=sendnetwindow[form, box, Pi^3];

When you are done and no longer need the .Net window, you should dispose of the resources and remove the *Mathematica* references to it:

closewindow[form];
ReleaseNETObject[form];
ReleaseNETObject[box];

These are the function definitions for netwindow[] and sendnetwindow[]:

```
Options[netwindow] = {windowsize -> {300, 300}, windowtitle -> "", windowcoords->{300,300},
    windowopacity -> 0.8, format -> Automatic};

netwindow[func_, opts___?OptionQ] :=
   NETBlock@Module[{form, box,icoords,size, title, opacity},
    {icoords, size, title, opacity} = {windowcoords, windowsize, windowtitle, windowopacity}
      /. Flatten[{opts}] /. Options[netwindow];
    form = NETNew["System.Windows.Forms.Form"];
    form@Width = size[[1]];
    form@Height = size[[2]];
    form@Text = title;
    form@Opacity = opacity;
    box = NETNew["Wolfram.NETLink.UI.MathPictureBox"];
    box@Parent = form;
    LoadNETType["System.Windows.Forms.DockStyle"];
    LoadNETType["System.Drawing.Color"];
    box@Dock = DockStyle`Fill;
    box@BackColor = Color`White;
    box@PictureType = format /. Flatten[{opts}] /. Options[netwindow] /. Automatic ->
"Automatic";
    ShowNETWindow[form];
    box@MathCommand = ToString[Unevaluated[func], InputForm];
    form@Location = NETNew["System.Drawing.Point", icoords[[1]], icoords[[2]]];
    KeepNETObject@form;
    KeepNETObject@box;
    {form, box}];

SetAttributes[sendnetwindow, HoldFirst];

sendnetwindow[form_?NETObjectQ, box_?NETObjectQ, func_, opts___?OptionQ] :=
  NETBlock@Module[{icoords,size, title, opacity},
```

```
{icoords, size, title, opacity} = {windowcoords, windowsize, windowtitle,
 windowopacity} /. Flatten[{opts}] /. Options[sendnetwindow];
form@Width = size[[1]];
form@Height = size[[2]];
form@Text = title;
form@Opacity = opacity;
box@MathCommand = ToString[Unevaluated[func], InputForm];
If[icoords != {300, 300},
    form@Location = NETNew["System.Drawing.Point",icoords[[1]],icoords[[2]]]];
KeepNETObject@form;
KeepNETObject@box;
{form, box}];

closewindow[x_]:=x@Dispose[];
```