

\textbf{MathLink Mode}

\textit{MathLink} is a protocol for sending and receiving \textit{Mathematica} expressions over an interprocess communication link. The \textit{Mathematica} front end and kernel communicate using \textit{MathLink}.\textit{MathLink mode} is a special mode of operation used by the \textit{Mathematica} kernel when it is being controlled by a front end. This article exposes the workings of \textit{MathLink} mode via some innovative uses of \textit{MathLink}.

David B. Wagner

\textit{MathLink} is a protocol for sending and receiving \textit{Mathematica} expressions over an interprocess communication link. \textit{MathLink} has a C-language API (application programmer's interface) as well as a high-level interface in \textit{Mathematica}, which the kernel translates into the lower-level interface. \textit{MathLink} generally is used in one of two ways: to enable the \textit{Mathematica} kernel to call functions written in compiled languages such as C, or to enable a program written in one of those languages to control the kernel. In fact, \textit{MathLink} is the mechanism used by the \textit{Mathematica} front end to communicate with the kernel.

The purpose of \textit{MathLink} is to hide the details of the network, and to ensure that the data sent by one party is interpreted correctly by the other, even if they are running on different computer platforms. The universal data exchange format provided by \textit{MathLink} is the expression. Thus, if one program sends the expression $f[2,5]$, a receiving program on a different computer will interpret it correctly even if the two computers use different byte orders, floating-point formats, and character string encodings (the symbol $f$ is sent and received as a character string).

In terms of the OSI seven-layer network architecture model, \textit{MathLink} is a level 6 (presentation layer) protocol. Other layers in the OSI model include level 1 (physical), level 4 (transport, typically TCP), and level 7 (application). The reason I mention \textit{MathLink}'s place in the OSI model is that, from the standpoint of that model, \textit{MathLink} is almost never used directly. A higher-level \textit{application layer} protocol is almost always used on top of it. These higher-level protocols specify certain allowable \textit{packet} types (which are simply \textit{Mathematica} expressions with special heads) and the proper response to each type of packet.

David B. Wagner is the president and founder of Principia Consulting, a training and consulting firm specializing in \textit{Mathematica} and other technical software packages. He is the author of Power Programming with \textit{Mathematica} (McGraw-Hill 1996).

One example of an application-layer protocol is the \textit{installable function} protocol, which defines a way for the kernel to call functions inside of an external program. The kernel sends to the external program an expression of the form \texttt{CallPacket[n, argumentlist]}, where $n$ is an integer index identifying which function in the external program is to be called (there may be several). It is the external program's responsibility to call the specified function with the given arguments and return the result wrapped in the head \texttt{ReturnPacket}. The installable function protocol also specifies how the external program communicates the definitions of the functions it defines to the kernel at installation time. Fortunately, the \textit{MathLink} Developer's Kit provides a tool called \texttt{mprep} that frees the programmer from having to worry about these details. An overview of how to write installable functions using \texttt{mprep} can be found in [Gayley 1994a] or [Wagner 1996, chap.11].

Another example of an application-layer protocol used on top of \textit{MathLink} is \textit{MathLink mode}, which is the topic of this article. \textit{MathLink} mode is the protocol used between the \textit{Mathematica} front end and the kernel. It can also be used to write a custom front end (like \textit{MathLink} for Microsoft Word, which enables word-processing documents to contain live \textit{Mathematica} calculations) or to embed the kernel's computational power in another application (as in \textit{MathLink} for Microsoft Excel, which allows spreadsheet cells to contain calls to \textit{Mathematica} functions). Although the C-language API for \textit{MathLink} is quite well-documented, \textit{MathLink} mode is not. This article will expose the details of \textit{MathLink} mode via some innovative uses of \textit{MathLink}.

You can find an authoritative, although somewhat abbreviated, introduction to \textit{MathLink} mode in [Gayley 1994b]. In contrast, this article will present methods that can be used to reverse-engineer the operation of the protocol. There are several reasons for this approach. First, there is no guarantee that the description contained herein is exhaustive. Second, the official \textit{MathLink} documentation usually can't keep up with changes to the protocol – the \textit{MathLink Reference}
The High-Level MathLink Interface

We will be using the high-level Mathematica interface to MathLink extensively in our attempts to reverse-engineer the MathLink mode protocol. In a nutshell, there are four Mathematica functions that perform most of the work: LinkOpen and LinkClose are used to open and close a MathLink connection; LinkRead and LinkWrite are used to read expressions from and write expressions to an open link. Other functions will be introduced as needed.

There are two ways to set up a MathLink connection: parent-child and peer-to-peer. Parent-child is used when one program launches the other. For example, when the Mathematica front end launches the kernel, the front end assumes the role of the parent and the kernel assumes the role of the child. However, in order to connect two running programs, it is necessary to use a peer-to-peer connection. A peer-to-peer connection is established when one side creates a link and listens for a connection, and the other side connects to the existing link. It doesn't matter which side does which; as the name implies, once the connection is established there is no distinction drawn between the two peers.

To use a peer-to-peer connection, the two parties must agree in advance on a name for the link. Using the native interprocess communication mechanisms on systems such as MacOS and Windows, the link name can be an arbitrary string. However, if the underlying protocol is TCP/IP, as it would be on a UNIX system, the link name must be a string containing a TCP port number, such as "5000". Since this form of a link name will work on any system that MathLink supports, it is what we shall use. (Note that certain port numbers, particularly very low ones, may be restricted on some operating systems.)

Below, the local kernel listens on a link called "5000" and the remote kernel connects to that link. If the two parties are running on different computers, the link name specified by the connecting party (the remote kernel, in this case) should be of the form "port@host", where host is the network name or address of the party that creates the link (the local kernel, in this case).

Below, the local kernel listens on a link called "5000" and the remote kernel connects to that link. If the two parties are running on different computers, the link name specified by the connecting party (the remote kernel, in this case) should be of the form "port@host", where host is the network name or address of the party that creates the link (the local kernel, in this case).

Below, the local kernel listens on a link called "5000" and the remote kernel connects to that link. If the two parties are running on different computers, the link name specified by the connecting party (the remote kernel, in this case) should be of the form "port@host", where host is the network name or address of the party that creates the link (the local kernel, in this case).
The above evaluation will “hang” until the one shown below is submitted to the remote kernel. Although subsequent \texttt{LinkWrite} are non-blocking operations, the initial call to \texttt{LinkWrite} by either kernel will block that kernel until a \texttt{LinkRead} is performed at the other end. I know of no way to avoid this behavior.

\begin{verbatim}
(Remote) RemoteIn[8]:=      
    LinkRead[linktolocal]    
(Remote) RemoteOut[9]:=    
    hello there
\end{verbatim}

The \texttt{LinkRead} call will always block the kernel if there is nothing to read. This can be avoided by using \texttt{LinkReadyQ} to see if there is data available before calling \texttt{LinkRead}. \texttt{LinkReadyQ} returns \texttt{False} if there is nothing to read:

\begin{verbatim}
(Local) LocalIn[7]:=       
    LinkReadyQ[linktoremote]    
(Local) LocalOut[7]:=    
    False
\end{verbatim}

If the remote kernel now sends an expression, \texttt{LinkReadyQ} in the local kernel will return \texttt{True}.

\begin{verbatim}
(Remote) RemoteIn[7]:=      
    LinkWrite[linktolocal, 42]    
(Local) LocalIn[8]:=       
    LinkReadyQ[linktoremote]    
(Local) LocalOut[8]:=      
    True
\end{verbatim}

After reading the expression off the link, \texttt{LinkReadyQ} again returns \texttt{False} (assuming that the remote kernel hasn’t sent anything else).

\begin{verbatim}
(Local) LocalIn[9]:=       
    LinkRead[linktoremote]    
(Local) LocalOut[9]:=      
    42
\end{verbatim}

\begin{verbatim}
(Local) LocalIn[10]:=      
    LinkReadyQ[linktoremote]    
(Local) LocalOut[10]:=      
    False
\end{verbatim}

Note that \texttt{LinkWrite} evaluates its arguments, so extra precautions must be taken to send an expression that is intended for evaluation at the other end of the link. For example, suppose that a symbol named \texttt{x} has a different value in each kernel, and one side sends the expression \texttt{x^2} to the other side:

\begin{verbatim}
(Local) LocalIn[11]:=      
    x = 2;    
    LinkWrite[linktoremote, x^2]
\end{verbatim}

The expression \texttt{x^2} was evaluated by the local kernel before being sent across the link. There is a function called \texttt{LinkWriteHold} that is identical to \texttt{LinkWrite} save that it expects its second argument to be wrapped in a \texttt{Hold} head (which it strips off before sending). However, \texttt{LinkWriteHold} will become obsolete in Version 3.0. The preferred way to achieve this effect, which also works in Version 2.2, is to wrap the second argument to \texttt{LinkWrite} in \texttt{Unevaluated}:

\begin{verbatim}
(Local) LocalIn[13]:=      
    LinkWrite[linktoremote, Unevaluated[x^2]]
\end{verbatim}

\begin{verbatim}
(Remote) RemoteIn[10]:=      
    LinkRead[linktolocal]    
(Remote) RemoteOut[10]:=      
    9
\end{verbatim}

Also, on occasion it is desirable to read an expression from a link without allowing the expression to evaluate. The function \texttt{LinkReadHeld} accomplishes this.

\begin{verbatim}
(Local) LocalIn[14]:=      
    LinkWrite[linktoremote, y^2]
\end{verbatim}

\begin{verbatim}
(Remote) RemoteIn[11]:=      
    y = 4;    
    LinkReadHeld[linktolocal]    
(Remote) RemoteOut[12]:=      
    Hold[y^2]
\end{verbatim}

\begin{verbatim}
(Remote) RemoteIn[13]:=      
    ReleaseHold[x]    
(Remote) RemoteOut[13]:=      
    16
\end{verbatim}

\texttt{LinkReadHeld}, too, is about to become obsolete. Although it has no alternative in Version 2.2, its functionality is achieved in Version 3.0 by calling \texttt{LinkRead[link, Hold]}. The optional second argument to \texttt{LinkRead} is a symbol that is used as a head for the expression read from the link. This mechanism is more general than \texttt{LinkReadHeld} because the supplied head can be anything, not just \texttt{Hold}.

The \texttt{Links} function returns a list of all open links in a \texttt{Mathematica} session. Here is the result of evaluating \texttt{Links[]} in the local kernel:

\begin{verbatim}
(Local) LocalIn[15]:=      
    Links[]
\end{verbatim}

\begin{verbatim}
(Local) LocalOut[15]:=      
    {LinkObject["Mathematica", 1, 1],    
     LinkObject[5000, 2, 2]}
\end{verbatim}
The second link in this list is the one we opened explicitly. The first link is a link to the Mathematica front end that is controlling the local kernel, which was opened automatically by the kernel when it was launched by the front end.

**Master-Slave Operation**

We can define our own application-level protocol for controlling a remote kernel. A simple way to do it is to put the remote kernel into a loop, continuously reading an expression from the link and writing it back (evaluating it along the way):

```plaintext
<Remote>Remote[n][14] =
  While[True,
    command = LinkRead[linktolocal];
    If[command === $Failed, Break[],
      LinkWrite[linktolocal, command] ]
]
```

The remote kernel appears to hang. Now the local kernel can write expressions to the link and read back the answers, without any manual intervention at the other end. To make things even easier, we can define the following function for use by the local kernel.

```plaintext
<Local>Local[n][16] =
  SetAttributes[RemoteEvaluate, HoldFirst];
<Local>Local[n][17] =
  RemoteEvaluate[expr_] :=
    ( LinkWrite[linktoremote, Unevaluated[expr]];
      LinkRead[linktoremote]
    )
```

Here are a few examples of using `RemoteEvaluate`.

```plaintext
<Local>Local[n][18] =
  RemoteEvaluate[Sqrt[x^2 + y^2]]
<Local>Local[n][19] =
  5
<Local>Local[n][20] =
  RemoteEvaluate[Print["hello world"]]"
```

The second example illustrates a problem with our approach. The return value of the `Print` command is `Null`; the actual printed output appeared in the remote kernel’s window! Here are some other problematic cases:

```plaintext
<Local>Local[n][21] =
  RemoteEvaluate[1/0]
<Local>Local[n][22] =
  ComplexInfinity
```

In this case, an error message appeared in the remote kernel’s window – and there is no way for the local kernel to tell that an error message was generated. The next case is disastrous:

```plaintext
<Local>Local[n][23] =
  RemoteEvaluate[Break[]]
```

The `Break[]` command caused the remote kernel to break out of the `While` loop. Now the local kernel is stuck, waiting for a reply that will never come. The safest way to recover from this situation is to send something manually from the remote kernel to the local kernel.

```plaintext
<Remote>Remote[n][15] =
  LinkWrite[linktolocal, "better luck next time"]
```

This string is returned from the local kernel’s last call to `RemoteEvaluate`.

```plaintext
<Local>Local[n][24] =
  "better luck next time"
```

Clearly, a better alternative to this simple protocol is needed. That alternative is **MathLink** mode. To a first approximation, **MathLink** mode is similar to the protocol that we just defined: the kernel sits in a loop, receiving input from the link, evaluating the input, and sending back results. However, the kernel’s only interaction with the outside world is through its link to the front end. This means that not only results, but also error messages, printed output, and so forth, are sent across the link.

**$ParentLink**

The kernel decides whether or not it is in **MathLink** mode according to the value of the system-defined symbol `$ParentLink`.

```plaintext
<Local>Local[n][22] =
  $ParentLink
<Local>Local[n][23] =
  LinkObject[Mathematica, 1, 1]
```

As long as `$ParentLink` contains a valid link object, the kernel will be in **MathLink** mode. On the other hand, if `$ParentLink` is `Null`, the kernel is not in **MathLink** mode. In that case, the kernel opens a console window on the screen and takes input directly from the keyboard. You can verify this behavior by evaluating the following input in any front end window:

```plaintext
<Local>Local[n][23] =
  $ParentLink = Null
```

The front end window hangs; a console window opens and an input prompt from the local kernel (`Local[n][24]:=`) appears in it. As soon as the kernel finishes evaluating `$ParentLink = Null`, it leaves **MathLink** mode. Now that the kernel is no longer in **MathLink** mode, it prints the `In` and `Out` labels to the console.
You can continue to work with the kernel via the console window for as long as you like. When you are ready to reconnect it to the front end, simply evaluate this expression in the console window:

\$\text{ParentLink} = \text{First[Links[]]}

The following will appear in the front end window just beneath the input that originally disconnected the kernel.

\$\text{LocalOut[24]} = \text{LinkObject[ Mathematica, 1, 1]}

Note that the `Out` number of this result does not match the `In` number of the input that appears above it, because other calculations have intervened. This output actually is the result of the last input to the kernel's console window (the assignment of `First[Links[]]` to `\text{ParentLink}`). At this point, you can continue working with the kernel via the front end, as though the interruption had never occurred.

**Stimulus-Response Testing**

The *Mathematica* kernel is a very trusting program. It will consider any program that is at the other end of `\text{ParentLink}` to be its front end. This means, in particular, that we might gain some insight into *MathLink* mode by setting `\text{ParentLink}` to a link that we control directly. (The idea of doing this is not new; an example appears in the *MathLink Reference Guide*.)

For example, let us set the remote kernel's `\text{ParentLink}` to its link to the local kernel:

\$\text{Remote} = \text{linkToLocal}

The remote kernel appears to hang. This is because it is no longer paying attention to its former input window; it is paying attention only to its `\text{ParentLink}`. If we attempt to read an expression from the link in the local kernel, we will discover that the remote kernel has sent something:

\$\text{LocalIn[25]} = \text{LinkRead[linkToRemote]}

\$\text{LocalOut[25]} = \text{LinkObject[ Mathematica, 1, 1]}

The `\text{ResumePacket}` is the kernel's way of telling a front end, "I'm in your hands now." The `\text{ResumePacket}` contains the previous value of `\text{ParentLink}` (in this case, the link to the remote kernel's *Mathematica* front end), presumably so that a well-behaved front end can restore that value when it is finished using the kernel.

Now we can engage in some simple stimulus-response testing of the remote kernel, by writing expressions to the link and seeing what comes back. It's important that the local kernel not attempt to read from the link unless there's something there to be read (or else the local kernel would hang), so we define the following utility function:

\$\text{Local[26]} = \text{LinkRead[link] :=}

\$\text{While[LinkReadyQ[link],}

\$\text{Print[InputForm[LinkRead[link]]]}

It turns out that the remote kernel has sent a few more packets:

\$\text{Local[27]} = \text{LinkDrain[linkToRemote]}

\$\text{OutputNamePacket["RemoteOut[16] = "]}

\$\text{ReturnTextPacket["LinkObject[5000, 2, 2]" ]}

\$\text{InputNamePacket["RemoteIn[17] = "]}

The `\text{OutputNamePacket}` and `\text{InputNamePacket}` are self-explanatory. The `\text{ReturnTextPacket}` contains the string form of the result of the last evaluation (`\text{ParentLink} = \text{linkToLocal}`).

Let us verify that output from `Print` and `Message` commands is indeed being sent across the link:

\$\text{Local[28]} = \text{LinkWrite[linkToRemote, Unevaluated[Print["hello world"]]]}

\$\text{Local[29]} = \text{LinkDrain[linkToRemote]}

\$\text{TextPacket["hello world"]}

\$\text{TextPacket["\n"]}

\$\text{ReturnPacket[Null]}

The printed text is indeed returned, wrapped in the `\text{TextPacket}` head. The `\text{ReturnPacket[Null]}` is the actual result of the `Print` command. But where are the `\text{OutputNamePacket}` and `\text{InputNamePacket}`, and why is the result returned in a `ReturnPacket` rather than a `\text{ReturnTextPacket}`? Perhaps this is a consequence of the fact that the result was `\text{Null}`. Let's try a computation with a non-null result:

\$\text{Local[30]} = \text{LinkWrite[linkToRemote, Unevaluated[1/0]]}

\$\text{Local[31]} = \text{LinkDrain[linkToRemote]}

\$\text{MessagePacket[Power, "\text{\text{infty}}"]}

\$\text{TextPacket["}

\$ \text{1/nPower::infty: Infinite expression -\n
encountered. In}

\$\text{ReturnPacket[DirectedInfinity[]]}

This result doesn't answer our questions, although it does illustrate how error messages are conveyed across the link.

The reason we didn't see an `\text{InputNamePacket}` or `\text{OutputNamePacket}` is that we aren't sending the correct type of packet to the kernel. There are many different kinds of packet heads; we've seen only a few of them. Here's a partial list. (Some of these packets are defined by the *Mathematica* front end and are not part of the basic *Mathematica* mode protocol.)
Wiretapping

The definitive implementation of proper front end behavior is, of course, the *Mathematica* front end itself. Although we don’t have access to the *Mathematica* front end’s source code, we have something almost as good: We can monitor its communication with the kernel in real time using another *MathLink* program.

If you are using Version 3.0, be sure to use the front end’s menu commands to set the default input and output formats to *InputForm* and *OutputForm* before proceeding. Otherwise, what you see will look very different than what is presented here.

The idea pursued here is to take a pair of links—one connected to a *Mathematica* front end, and one connected to a *Mathematica* kernel—and connect each of them to a third, *MathLink*-aware process. This third process will simply pass data back and forth between the links without interpreting any of it. However, all data will be printed as it is passed through. The function `Wiretap` will monitor a pair of links in this way:

```mathematica
Wiretap[link1_LinkObject, link2_LinkObject] :=
  While[LinkConnectedQ[link1] && LinkConnectedQ[link2],
    LinkEcho[link1, link2, "--> ";]
    LinkEcho[link2, link1, "<-- ";];
```

`Wiretap` relies on another function, `LinkEcho`, which does the real work. We have made `LinkEcho` a separate function so that it will be simple to change the way the data is formatted.

```mathematica
LinkEcho[src_, dest_, leader_] :=
  Module[{msg},
    If[LinkConnectedQ[src] && LinkReadyQ[src],
      msg = LinkReadHe[dest, src, msg];
    If[msg == $Failed, Break[]];
    Print[leader, InputForm / HoldForm 00 msg];
    If[LinkConnectedQ[dest], LinkWriteHe[dest, msg]]; ];
  ];
```

`LinkEcho` checks the source link to see if it is connected (`LinkConnectedQ[src]`) and if there is data waiting to be read (`LinkReadyQ[src]`). If so, it reads an expression from the link without evaluating it. If `LinkReadHe` returns `$Failed`, then there is some problem with the link, so `LinkEcho` executes a `Return[]` command, causing the `While` loop in `Wiretap` to exit. (We need not be concerned that a `$Failed` symbol being passed across the link will trigger this action, because the value returned by `LinkReadHe` in that case would be of the form `Hold[packethead[$Failed]]`.) Otherwise, `LinkEcho` prints the held expression, preceded by the leader string, and then writes the held expression to the destination link (dest). `Wiretap` alternately calls `LinkEcho` with `link1` as the source and `link2` as the destination, and vice versa.
All we need to do now is call \texttt{Wiretap} with a link to a \textit{Mathematica} front end and a link to a \textit{Mathematica} kernel. By an amazing stroke of good fortune (well, not really), the local kernel already possesses one of each!

\begin{verbatim}
Links[]
\end{verbatim}

Before calling \texttt{Wiretap}, we must put the remote kernel back into \textit{MathLink} mode. Enter the following in the remote kernel's console window:

\texttt{\$ParentLink = linkToLocal}

We know from experience that there are now several packets pending on the link from the remote kernel. The present situation is analogous to an open electrical circuit, which we close by passing the two links to \texttt{Wiretap}.

\begin{verbatim}
Wiretap @@ Links[]
\end{verbatim}

Observe that the previous evaluation was entered into the local kernel's window, yet the input and output have been labeled by the remote kernel. The local kernel's \textit{Mathematica} front end has been tricked into thinking that the packets being sent by the remote kernel really are from the local kernel. (In Macintosh Version 2.2, which was used to prepare this article, the \textit{Mathematica} front end seems to have a bug that causes the \texttt{In} number displayed by the front end to be off by one.)

While the local kernel is running the \texttt{Wiretap} function, typing your system's interrupt keyboard combination (such as Control-C on UNIX, Command-. on MacOS, Alt-. on Windows) in the local kernel's front end window will cause the \texttt{Wiretap} function to abort. The reason is that a \textit{MathLink} link actually carries two channels of information: the normal channel, which we are accessing with \texttt{LinkRead} and \texttt{LinkWrite}, and a high-priority channel for interrupt messages and the like. The high-priority channel can be written to using the \texttt{LinkInterrupt} function, but you cannot receive from it using \textit{Mathematica} functions (it can be accessed only by using the C-language API). Typing the interrupt keyboard combination causes the front end to send a \textit{MathLink} interrupt message, which would be caught by the local kernel and would abort the execution of \texttt{Wiretap}.

From this point forward, the input to the local kernel's \textit{Mathematica} front end will be passed transparently to the remote kernel. Here's proof:

\begin{verbatim}
x * y
\end{verbatim}

\begin{verbatim}
-> Enter["x * y"]
\end{verbatim}

\begin{verbatim}
<< OutputNamePacket["RemoteOut[18] = "]
\end{verbatim}

\begin{verbatim}
<< ReturnTextPacket["7"]
\end{verbatim}

\begin{verbatim}
<< InputNamePacket["RemoteIn[19] := "]
\end{verbatim}

\begin{verbatim}
\end{verbatim}

We can see that the standard way of interacting with a kernel in \textit{MathLink} mode is to send the input, in string form, wrapped inside an \texttt{Enter} head. (Note that \texttt{Enter} is now obsolete; the preferred head is \texttt{EnterTextPacket}.) The result is returned as a string inside a \texttt{ReturnTextPacket}. However, if the result is \texttt{Null}, no \texttt{ReturnTextPacket} will be sent:

\begin{verbatim}
Print["a null result"]
\end{verbatim}

\begin{verbatim}
-> Enter["Print["a null result"]"]
\end{verbatim}

\begin{verbatim}
<< TextPacket["a null result"]
\end{verbatim}

\begin{verbatim}
a null result
\end{verbatim}

\begin{verbatim}
<< TextPacket["\n"]
\end{verbatim}

\begin{verbatim}
<< InputNamePacket["RemoteIn[20] := "]
\end{verbatim}

\begin{verbatim}
RemoteIn[20] := 
\end{verbatim}

On the other hand, if the input contains more than one expression, the result of each expression is returned in its own \texttt{ReturnTextPacket}, preceded by an \texttt{OutputNamePacket}.

\begin{verbatim}
\texttt{x}
\end{verbatim}

\begin{verbatim}
\texttt{y}
\end{verbatim}

\begin{verbatim}
-> Enter["x \texttt{\&} y"]
\end{verbatim}

\begin{verbatim}
<< OutputNamePacket["RemoteOut[20] = "]
\end{verbatim}

\begin{verbatim}
<< ReturnTextPacket["9"]
\end{verbatim}

\begin{verbatim}
<< OutputNamePacket["RemoteOut[21] = "]
\end{verbatim}

\begin{verbatim}
<< ReturnTextPacket["4"]
\end{verbatim}

\begin{verbatim}
<< InputNamePacket["RemoteIn[22] := "]
\end{verbatim}

\begin{verbatim}
RemoteIn[22] := 4
\end{verbatim}

(The fact that the \textit{Mathematica} front end didn't print the first of the two results is probably a bug. When a kernel is being run directly from the front end, both results are printed.)

The conclusion that can be drawn is that, when the kernel is being used in \textit{MathLink} mode, the only sure sign that a computation initiated by \texttt{Enter} (or \texttt{EnterTextPacket}) is completely finished is the receipt of an \texttt{InputNamePacket}. Observe what happens when the kernel sends PostScript to the front end. (If the remote kernel was not started from the \textit{Mathematica} front end, you will need to evaluate the expression \texttt{Display = $Output} before executing the following \texttt{Plot} command, or else the kernel will not know where to send the PostScript.)
Plot[x^2, {x, 0, 1}, Ticks -> None]

--> Enter["Plot[x^2, {x, 0, 1}, Ticks -> None"]]

\[\text{~DisplayPacket["~]~}
\]

\[\text{~DisplayPacket["\%!\%\%Creator: ~Mathematica\%!\%\%AspectRatio: 0.61803\%!\%\%Graphics\%!\%\%Courier\%!\%\%Graphics\%!\%\%}~}
\]

\[\text{~End of Graphics\%!\%\%GraphicsEnd\%!\%\%}~}

\[\text{~DisplayEndPacket["~]~}
\]

\[\text{~OutputNamePacket["\%!\%\%RemoteOut[22]= ~]~}
\]

\[\text{~ReturnTextPacket["-Graphics-~]~}
\]

\[\text{~InputNamePacket["\%!\%\%RemoteIn[23]= ~]~}
\]

\text{Local RemoteOut[22]=}

\text{-Graphics-}

The PostScript is sent in a series of DisplayPacket (of which there may be many for complicated graphics). The PostScript is terminated by a single DisplayEndPacket. From the position of the graphic above, we deduce that the Mathematica front end actually begins to render a graphic before all of the PostScript has arrived.

For convenience, in the future you may wish to use the following definition of LinkEcho. It differs from our first definition only in that it prints the contents of a DisplayPacket as the string "<postscript>".

\text{LinkEcho[src_, dest_, leader_] := ~}

\text{Module[{msg}, ~}

\text{\quad If[LinkConnectedQ[src] \&\& LinkReadyQ[src], ~}

\text{\quad \quad msg = LinkReadHead[src]; ~}

\text{\quad \quad If[msg == "$Failed", Break[]]; ~}

\text{\quad \quad If[msg[[1, 0]] == "DisplayPacket", ~}

\text{\quad \quad \quad Print[leader, "DisplayPacket<postscript>"], ~}

\text{\quad \quad \quad Print[leader, InputForm \hspace{1em} \& HoldForm \hspace{1em} \& msg]; ~}

\text{\quad \quad If[LinkConnectedQ[dest], ~}

\text{\quad \quad \quad LinkWriteHead[dest, msg]; ]; ]; ]}

Certain Mathematica commands require input from the user. One of these is the Input command.

\text{Local RemoteOut[24]=}

\text{1 + Input["Enter your favorite number"]}

--> Enter["1 + Input[\"Enter your favorite number\"]"]

\text{InputPacket["Enter your favorite number"]}

The Input command causes the kernel to send an InputPacket to the front end containing the prompt string. In response, the front end displays a dialog box like this:

![Local Kernel Input](image)

The front end sends the user's input back to the kernel in another Enter (or EnterTextPacket) packet. The result of the calculation comes back in a ReturnTextPacket, as usual.

--> Enter["42"]

\text{OutputNamePacket["\%!\%\%RemoteOut[23]= ~]~}

\[\text{~ReturnTextPacket["43"]~}

\[\text{~InputNamePacket["\%!\%\%RemoteIn[24]= ~]~}

\text{Local RemoteOut[23]=}

\text{43}

The InputString command is slightly different from the Input command; you may wish to monitor a call to InputString to see the difference.

Another command that requires input from the user is the Interrupt command. The Interrupt command causes the kernel to send a MenuPacket to the front end, which implies that the user is to be presented with certain choices. The first element of the MenuPacket identifies which "menu" is to be displayed. The second element is a text prompt suitable for text-based front ends.

\text{Local RemoteOut[25]=}

\text{1 + Interrupt[]}

--> Enter["1 + Interrupt[]"]

\[\text{~MenuPacket[1, "Interrupt"]~}

In response to this MenuPacket, the Mathematica front end displays another dialog box.

![Local Kernel Interrupt](image)
If you click the Abort button, the following packets will be exchanged:

```plaintext
--> "a"
<-- OutputPacket["RemoteOut[24] = "a"]
<-- ReturnPacket["Aborted"]
<-- InputPacket["RemoteIn[25] := "a"]
```

Here is what happens if you click the Enter Dialog button instead:

```plaintext
(Local) RemoteIn[26] :=
  1 + Interrupt[1]
  --> Enter["1 + Interrupt"]
  <-- MenuPacket[1, "Interrupt"]
  --> "a"
  <-- BeginDialogPacket[1]
  <-- InputPacket["RemoteIn[26] := "a"]
```

By now it is clear that the kernel expects a character string response to a MenuPacket. The Mathematica front end is programmed with all the valid responses to any given menu and it presents a dialog box with a button for each possible response; unfortunately, we don’t have this knowledge. We’ll return to the problem of how to respond to an arbitrary MenuPacket in a moment. First, we will investigate the behavior of the kernel within a dialog.

A dialog is a subsidiary Mathematica session that is begun while in the midst of another computation. The kernel sends a BeginDialogPacket to the front end to inform it that a dialog is beginning. This is necessary because there are other ways to enter a dialog besides requesting it from the Mathematica front end’s interrupt dialog box. A dialog may be entered directly using the Dialog command, or it may result from the use of the TraceDialog function, to name just two examples.

You can evaluate any Mathematica expression within a dialog. For example, the Stack function shows all of the functions on the kernel’s evaluation stack, which can be helpful if you are trying to debug a computation:

```plaintext
(Local Dialog) RemoteIn[27] :=
  Stack[]
  --> Enter["Stack[]"]
  <-- OutputPacket["RemoteOut[26] = "]
  <-- ReturnPacket["Plus, Interrupt"]
  <-- InputPacket["RemoteIn[27] := "]
```

You can even enter another dialog from within a dialog!

```plaintext
(Local Dialog) RemoteIn[28] :=
  Dialog[argument]
  --> Enter["Dialog[argument]"]
  <-- BeginDialogPacket[2]
  <-- OutputPacket["RemoteOut[28] = "]
  <-- ReturnPacket["argument"]
  <-- InputPacket["RemoteIn[29] := "]
```

The kernel has sent a BeginDialogPacket[2] packet, indicating that a second-level dialog has been entered. The Mathematica front end uses the information contained in a BeginDialogPacket to alter the In and Out labels that it places on cells; if you are writing your own front end, you may choose simply to ignore such packets. Also note that the kernel has sent the argument to the Dialog function back to the front end in a ReturnPacket. In other words, at the beginning of the dialog, you can refer to the argument to Dialog (if there was one) using %:

```plaintext
(Local Dialog Level 2) RemoteIn[30] :=
  1 + %
  --> Enter["1 + %"]
  <-- OutputPacket["RemoteOut[29] = "]
  <-- ReturnPacket["1 + argument"]
  <-- InputPacket["RemoteIn[30] := "]
```

A dialog is exited using the Return function. The kernel informs the front end of this by sending an EndDialogPacket.

```plaintext
(Local) Return[somevalue]
  --> Enter["Return[somevalue"]
  <-- EndDialogPacket[2]
  <-- OutputPacket["RemoteOut[27] = "]
  <-- ReturnPacket["somevalue"]
  <-- InputPacket["RemoteIn[28] := "]
```

Any argument supplied to Return is returned from the original call to Dialog. However, the kernel sets $Line to the value it had when Dialog was called, as though the dialog never happened. The Mathematica front end plays along with the charade, moving the cell insertion point to a position just after the cell group containing the most recent call to Dialog (in fact, the last three packets shown above actually appear a few cells back in the notebook window).

We are still inside of the dialog that we entered from the Interrupt menu.

```plaintext
(Local) Return[]
  --> Enter["Return[]"]
  <-- EndDialogPacket[1]
```
Once again, the front end jumps backward in the notebook file and prints the following:

```plaintext
<- ReturnPacket["1 + Null"]
<- InputPacket["RemoteIn[26] := "]
(Local) RemoteOut[25] =
1 + Null
```

Although I can appreciate what the Mathematica front end is trying to do, I find this behavior to be self-defeating: If the purpose of the notebook interface is to keep a record of a Mathematica session, then the nonlinear placement of output is at odds with this goal.

Let us return to the topic of the MenuPacket and how to respond to it. There is a way to find out the set of valid responses to any MenuPacket, but to demonstrate it we have to end the wiretapping session and reestablish the local kernel's direct control over the remote kernel. We will simply close the remote kernel's link to the local kernel and, at the same time, open another link back to the local kernel.

```plaintext
(LinkClose[linkToLocal]
$ParentLink =
linkToLocal = LinkOpen["5001", LinkMode -> Listen]
-> Enter["LinkClose[linkToLocal]n$ParentLink =n \n linkToLocal = LinkOpen["5001", LinkMode ->\n Listen]"]
<- SuspendPacket[Null]
LinkObject:::Link:
LinkObject[5000, 2, 2]
is closed; the connection is dead.
```

The error message is caused by LinkEcho's call to LinkReadHold, which returns $Failed, causing Wiretap to exit from the While loop and return Null. The local kernel resumes responding to commands entered in its front end window (note the labels on the cells):

```plaintext
(Local) LocalOut[29]=
x + y
(Local) LocalOut[39]=
2 + y
```

Next, reestablish the connection with the remote kernel.

```plaintext
(linktoremote = LinkOpen["5001", LinkMode -> Connect]
(Local) LocalOut[40]=
LinkObject[5001, 3, 2]
```

We are now ready to demonstrate the menu handling technique. Initiate a menu action:

```plaintext
LocalIn[41]=
LinkWrite[linktoremote,
EnterPacket["1 + Interrupt[])"]
```

As noted earlier, the second element of the MenuPacket is a string that is suitable for use as a prompt to the user. To obtain the information necessary to present a more user-friendly interface, simply send the string "?" to the kernel.

```plaintext
(Local) LocalIn[43]=
LinkWrite[linktoremote, "??"]
```

You can print the contents of the TextPacket directly to an ANSI-compatible output stream and it will format nicely. \n is an ASCII tab character; \r is an ASCII return character, which is used by the MacOs instead of ASCII newline. Users of Windows or UNIX systems will see \n in place of \r. Then, simply send the user's response back to the kernel; if it isn't one of the allowed responses, the kernel will repeat the MenuPacket and the TextPacket. In this case, we choose to allow the computation to terminate normally.

```plaintext
(Local) LocalIn[45]=
LinkWrite[linktoremote, "c"]
```

Miscellaneous Packet Types

While we have the remote kernel in stimulus-response mode, we will demonstrate a few more features of MathLink mode.

There are three different packets for requesting evaluations from the kernel. The first, EnterTextPacket, has been demonstrated. The other two are EnterExpressionPacket and EvaluatePacket. Both of these are used to send a Mathematica expression — not a string containing an expression — to the kernel for evaluation. The main difference between the three types of packets is the amount of processing that their contents undergo.

```plaintext
```

VOLUME 6, ISSUE 3 53
There are several functions that affect the global evaluation process in Mathematica, namely, the functions assigned to the symbols $\$PreRead$, $\$Pre$, $\$Post$, and $\$PrePrint$. The $\$PreRead$ function is applied to every input string before it is parsed. $\$Pre$ is applied to each input after parsing, but before evaluation. $\$Post$ is applied to each result after evaluation, but before the result is assigned to Out[n]. Finally, $\$PrePrint$ is applied to the result after the assignment to Out[n] but before printing. Unless the function assigned to $\$Pre$ holds its argument, the same effect can be achieved by assigning the function to $\$Post$.

When you request an evaluation from the kernel using EnterTextPacket, the evaluation goes through the entire input-output procedure outlined above. To demonstrate, we'll assign functions to each of the special symbols in the remote kernel.

\begin{verbatim}
LinkWrite[LinkRemote, Unevaluated[
  $\$PreRead = Function[s,
    Print["$\$PreRead", InputForm[s], "]"]; s;
  $\$Pre = Function[e,
    Print["$\$Pre", HoldForm[InputForm[e]], "]"]; e,
    HoldFirst];
  $\$Post = Function[e,
    Print["$\$Post", e, "]"]; e;
  $\$PrePrint = Function[e,
    Print["$\$PrePrint", e, "]"]; e;]]
\end{verbatim}

\begin{verbatim}
LinkWrite[LinkRemote, EnterTextPacket["x+y"]]
\end{verbatim}

\begin{verbatim}
LinkWrite[LinkRemote, EnterTextPacket["x*y"]]
\end{verbatim}

\begin{verbatim}
LinkWrite[LinkRemote, EnterTextPacket["x+y"]]
\end{verbatim}

\begin{verbatim}
$\$PreRead is not applied to the content of an EnterExpressionPacket, since it already is in the form of an expression. For the same reason, a syntax error cannot occur. The result of an EnterExpressionPacket is a ReturnExpressionPacket, which again contains an expression, rather than a string. (If the result is Null, no ReturnExpressionPacket is sent.) If you want to receive a string result, simply wrap the content of the EnterExpressionPacket in the ToString head.

EnterExpressionPacket doesn't seem to be used by the Version 2.2 front end, but the Version 3.0 front end uses it extensively.

EvaluatePacket is entirely different from the Enter packets. The content of an EvaluatePacket is evaluated by the kernel, and the result is returned, but no other processing is done.

Because the content of an EnterTextPacket is a string, it can contain syntax errors. When a syntax error is encountered, the kernel sends back a SyntaxPacket containing the byte offset within the input string at which the parser gave up. The kernel does not increment $\$Line$, although it still returns an $\$InputNamePacket$. Note that $\$PreRead$ is applied before parsing takes place.
LinkWrite[linkToRemote, 
Unevaluated[EvaluatePacket[Rationalize[x]]]]

LinkDrain[linkToRemote]
ReturnPacket[7]

In other words, the head EvaluatePacket is simply ignored; the effect is the same as if no packet head at all is used. However, it's probably a good idea to use it when called for. Let's clear the input-output processing functions.

LinkWrite[linkToRemote, 
Unevaluated[EvaluatePacket[ 
Clear[$PreRead, $Pre, $Post, $PrePrint]]]]

LinkDrain[linkToRemote]
ReturnPacket[Null]

Note that there is always a ReturnPacket in response to an EvaluatePacket, even when the result is Null.

The EvaluatePacket→ReturnPacket exchange does not affect $Line, nor does it cause any assignments to In or Out. Here is proof:

LinkWrite[linkToRemote, EnterTextPacket["1 + %"]]

LinkDrain[linkToRemote]

OutputPacket["RemoteOut[31]= "]
ReturnPacket["8."]
InputPacket["RemoteIn[32]= "]

EvaluatePacket is therefore useful when your front end needs to perform some calculation without disturbing the state of the user's session. The MathLink front end uses this strategy for several purposes; we'll see an example in the next section.

To summarize, there are three request/reply packet forms: EnterTextPacket/ReturnTextPacket, EnterExpressionPacket/ReturnExpressionPacket, and EvaluatePacket/ReturnPacket. The first two run the input through the kernel's sundry input-output processing, which you can remember by thinking of the prefix Enter as "entered at the keyboard." The third alternative performs no input-output processing.

Intercepting Kernel Initialization

Although the MathLink traffic analyzed in this section is not technically part of MathLink mode, these examples should give you some ideas about the kinds of user interface features that can be implemented using MathLink mode.

The kernel behaves slightly differently when the link controlling it is the link to the Mathematica front end that launched the kernel. This is because the Mathematica front end sends some initialization code over a link when the link is first opened. Since the kernel sends this code only once, subsequently tying the link to a different kernel (as we did with the Wiretap function) does not initialize the second kernel properly.

It is possible to intercept the MathLink front end's initialization of the kernel, by judiciously configuring the front end's kernel connection before the connection is opened. The idea is to make the front end connect to a named link rather than to launch a kernel program. The procedure presented here uses a single Mathematica front end to control several kernel connections. It is not something the front end was designed to do, as evidenced by the fact that I managed to hang the front end or kernel several times while writing this article. The steps presented below worked for me using Mathematica Version 2.2.2 for Macintosh. If you follow them to the letter, they should work for you, too. If you are using a different version of Mathematica, you may need to experiment a bit. In case of difficulty, try interacting with the local kernel through its own console window, or using a different Mathematica front end for each kernel connection.

Kernel connection settings are configured via a dialog box in the front end. This dialog box is available in Version 2.2 for Macintosh by choosing Kernels and Tasks from the Action menu, or, in the Windows Enhanced Version 2.2, by choosing Kernels from the Options menu. Consult the User's Guide for your version of Mathematica if you need help locating the dialog box.

Bring up the dialog box and click on the button to create a new kernel. Configure this connection so that, instead of running an executable file, it connects to an existing link. Here is how this configuration is specified on the Macintosh:

We have configured a kernel called Wiretap that will cause the front end to connect to port 5005 when the kernel is launched. Click the OK button, but do not start this kernel yet.

Next, open a link on port 5005 in the local kernel.

linktofe = LinkOpen["5005", LinkMode -> Listen]
LinkObject[5005, 4, 3]
If you are running the local kernel from the same front end that will connect to the wiretap kernel, do not attempt to make the connection yet. If you do, the front end will hang, waiting for some response to the connection attempt. Unfortunately, you won’t be able to type anything into the local kernel’s window to send that response – a classic deadlock situation.

When the front end connects, it waits for the first InputNamePacket from the kernel before sending anything, to ensure that it doesn’t send the code onto a link before the kernel is ready to receive it. The local kernel will send this packet to get things started. After sending the initialization code, the front end will then wait for the next InputNamePacket. The remote kernel needs to receive the initialization code and reply with the InputNamePacket. Therefore, to avoid a deadlock, the local kernel needs to send the first InputNamePacket and call Wiretap in a single input. Evaluate the following input in the local kernel; then open the Kernels dialog box and launch the wiretap kernel, which will connect to the link that the local kernel calls linktofe.

```
(Local) LinkWrite[Linktofe, InputNamePacket["jump start!"]]; Wiretap[Linktofe, LinkToRemote]

-> Enter["\n\n(FE) saved$Line = $Line; $Line = \nInfinity;\n\n\nGeneral::nomathlink = "MathLink
..."

"RemoteOut[32] := "]
```

As before, the remote kernel is now being controlled by the front end, but all MathLink traffic is filtered through the local kernel. A huge amount of output (shown in a highly abbreviated form, above) appears in the local kernel’s window. This output consists of initialization code that the Mathematica front end sends to the kernel when the connection is first set up. Rather than analyze this code here, we’ll comment on the relevant portions as we see its effects, below. (If your remote kernel was originally started by a Mathematica front end, a Messages window containing some error messages will appear. This is a consequence of the fact that the initialization code is being sent to the remote kernel for the second time, and some of the symbols it sets up are protected.)

Unless specified otherwise, the inputs in the remainder of this section should be evaluated in a front end window whose evaluator is set to the wiretap kernel that we configured above. (The local kernel’s front end window will not accept any input while it is waiting for a return from the Wiretap function.) Here’s an example of a simple evaluation in the wiretap kernel’s window:

```
(Wiretap) RemoteIn[32]:= x / y
(Wiretap) RemoteOut[32]:=
   3
   4
```

The input above was processed by the remote kernel. A final bit of private communication between the front end and the remote kernel appears in the local kernel’s window, and beneath that, the MathLink traffic for this evaluation:

```
-> Enter["x / y"]
<< InputFormPacket["3/4"]
<< OutputNamePacket["RemoteOut[32] := "]
<< ReturnTextPacket["3\n\n\n"]
<< InputNamePacket["RemoteIn[33] := "]
```

There’s a new kind of packet here which we haven’t seen before: the InputFormPacket. As its name suggests, this packet contains the InputForm of the result of the evaluation. The string is stored by the front end in a Mathematica notebook, but is hidden from the user. Having the InputForm of each result in the notebook allows the front end to implement the Copy Output From Above command.

How is the InputForm packet generated? An examination of the initialization code reveals that the front end creates a pure function that wraps the InputForm of its argument in the InputFormPacket head, and sends the resulting expression to $ParentLink. The pure function is then assigned to the symbol System`Private`$SystemPrint. This symbol is undocumented, but its purpose is fairly obvious. Even more interesting is the fact that the initialization code creates down-values for LinkWrite that cause the unsetting and resetting of $SystemPrint when SuspendPackets and ResumePackets are sent. (If the remote kernel was initially launched directly, rather than from a Mathematica front end, you won’t see the first two down-values in the following output.)

```
(Wiretap) RemoteIn[33]:= DownValues[LinkWrite]
(Wiretap) RemoteOut[33]:=

{
Literal[LinkWrite[LinkObject[Mathematica, 1, 1],
SuspendPacket[___, ___]] :>
Null ;
(FE) saved$SystemPrint =
System`Private`$SystemPrint;
System`Private`$SystemPrint = ; False),
Literal[LinkWrite[LinkObject[Mathematica, 1, 1],
ResumePacket[___, ___]] :>
Null ;
(System`Private`$SystemPrint =
(FE) saved$SystemPrint; False),
Literal[LinkWrite[LinkObject[5001, 3, 2],
SuspendPacket[___, ___]] :>
Null ;
(FE) saved$SystemPrint =
System`Private`$SystemPrint;
System`Private`$SystemPrint = ; False),
Literal[LinkWrite[LinkObject[5001, 3, 2],
ResumePacket[___, ___]] :>
Null ;
(System`Private`$SystemPrint =
(FE) saved$SystemPrint; False)}
```

56 THE MATHEMATICA JOURNAL © 1998 Miller Freeman Publications
Thus, when you set $ParentLink$ to a link that was not initialized by the front end, InputForm packets are suppressed; when you return control to that link, the generation of these packets is re-enabled. This is one of the reasons we went to the trouble of intercepting the initialization process.

Here are another trick that the Mathematica front end uses. When you type part of a command name, for example, Factor, and then choose the Complete Selection menu command, a menu of possible commands that begin with Factor pops up on the screen. Here is what is sent between the front end and the kernel:

```
\[ \text{-> EvaluatePacket[ToString[ToExpression["\!\(\text{\textbackslash Factor}\)\!\(\text{\textbackslash FactorComplete}\)\!\(\text{\textbackslash Factorial}\)\!\(\text{\textbackslash Factorial2}\)\!\(\text{\textbackslash FactorInteger}\)\!\(\text{\textbackslash FactorList}\)\!\(\text{\textbackslash FactorSquareFree}\)\!\(\text{\textbackslash FactorSquareFreeList}\)\!\(\text{\textbackslash FactorTerms}\)\!\(\text{\textbackslash FactorTermsList}\)\)]\]}}\]
\[  \text{\textbackslash ReturnPacket[\"Null\"]}\]
```

The expression that was evaluated in this particular case is a call to a function that was defined during the initialization process, whose purpose seems to be to print the list of strings that need to appear in the command completion menu. Note that the front end wrapped the contents of the EvaluatePacket in ToString, to ensure that a ReturnPacket would be generated.

The Mathematica front end uses EvaluatePacket for a variety of other purposes, such as the Make Template command and the Function Browser (on those versions of the front end that support it). The beauty of this technique is that it allows information to flow from the kernel to the front end without disturbing the state of the user's session (for example, the symbol % still refers to the result of the user's previous calculation).

When you have finished experimenting with this setup, terminate the wiretap by opening the front end's Kernels dialog box, selecting the Wiretap kernel, and clicking the Quit/Disconnect Kernel button. The front end will close the corresponding link. The Wiretap function in the local kernel will print the usual error message before exiting:

```
\[ \text{LinkObject::linkid:\!\(\text{LinkObject[5005, 4, 3]}
\  \text{is closed; the connection is dead.}\}}\]
\[ \text{LinkClose[linktoremote]}\]
```

Conclusions

A solid understanding of MathLink mode is a prerequisite for programmers wishing to write applications that control the Mathematica kernel. Although official documentation on this topic is incomplete, the interaction between the Mathematica front end and the kernel provides a rich source of examples. This article developed a tool for probing this interaction and used this tool to illustrate some of the more interesting aspects of MathLink mode. This technique will never become obsolete, as is inevitably the case with printed documentation.

References


David B. Wagner
Principia Consulting
3841 Orion Court, Boulder, CO 80304-1024
dbwagner@princon.com

The electronic supplement contains the notebooks Local and Remote.