

A *MathLink* Program for Accessing Binary Files

I present a *MathLink* program that implements the package `FastBinaryFiles` for reading and writing binary files. This package is intended as a replacement for the standard package `Utilities`BinaryFiles``, which is slow and cannot be used on Windows machines. `FastBinaryFiles` is a good example of a non-trivial *MathLink* program, and it provides the context for a discussion of some *MathLink* programming issues.

Todd Gayley

Many users of *Mathematica* need to read or write binary files, these being files that are interpreted as nothing more than a sequence of raw bytes or groups of bytes. For example, one might want to read the bit sequence 01100001 as the number 97 instead of the ASCII encoding of the letter *a*. Typical examples are image and sound files, and files created by some laboratory and industrial data-collecting instruments. The standard *Mathematica* package `Utilities`BinaryFiles`` was created to simplify the process of reading and writing binary files. Unfortunately, there are two problems with this package. First, it can be very slow; so slow, in fact, as to render it practically unusable for large files or slower computers (some timings are given below). The second problem is that it does not work on DOS and Windows machines. This is because *Mathematica* is only capable of opening files in so-called text mode, not binary mode. On UNIX and Macintosh, the distinction is not important, but on DOS/Windows it is. In text mode, some characters and character combinations are assigned special meanings that prevent them from being read or written literally. For example, the byte pair with hex values 0D and 0A (the carriage return-linefeed combination) is read as a single character.

To overcome these difficulties, I present here an external *MathLink* program for handling binary files, and a package that communicates with it. The package is modestly named `FastBinaryFiles`, and it is intended to be a more or less exact replacement for the standard package `Utilities`BinaryFiles``. In particular, the documentation for that package serves for mine as well.

It is likely that a future release of *Mathematica* will be able to open files in binary mode. That will make it possible to handle binary files from within *Mathematica* on DOS and Windows machines. Even so, `FastBinaryFiles` will be considerably faster than is possible within *Mathematica*.

`FastBinaryFiles` is an excellent example of a non-trivial *MathLink* program and it provides a framework to examine

several topics in *MathLink* programming. The next section contains a brief description of how to use the package. The rest of the column is devoted to a discussion of *MathLink* and the development issues encountered in `FastBinaryFiles`.

The *MathLink* program that implements `FastBinaryFiles` works with *Mathematica* Version 2.2 or later (except on DOS machines, for which *MathLink* is not available). Windows users must have Version 2.2.2 or later, or Version 2.2.1 and the *MathLink* Developer's Kit for Windows. These may or may not be shipping at the time this column is published, but they can be ordered.

Using the Package

The electronic supplement contains the source code for the `FastBinaryFiles` program. For Macintosh and Windows users, there is also a ready-to-run executable. Everyone else will need to build the program from the source; this process is explained later. For the moment, I will assume you have a program named `binary` in a directory that is accessible to *Mathematica*.

The first step is to launch the program and prepare its functions for use in *Mathematica*. This is done with the `Install` function:

```
In[1]:= link = Install["binary"]
Out[1]= LinkObject[binary, 2, 2]
```

Note that there is no separate package file to be loaded. The `FastBinaryFiles`` context and all its definitions have been loaded automatically:

```
In[2]:= $Packages
Out[2]= {FastBinaryFiles`, Global`, System`}
```

Since `FastBinaryFiles` is intended to be an exact replacement for the standard package `Utilities`BinaryFiles``, I will not devote much space here to showing how to use the package. You can consult the documentation for `Utilities`BinaryFiles`` in the *Guide to Standard Mathematica Packages*

Todd Gayley received his Ph.D. in evolutionary biology from the University of Arizona in 1989. He began using *Mathematica* immediately after its introduction in 1988 for his own research in population genetics. He is currently a member of the vertical applications department at Wolfram Research.

for further discussion. All the functions and options in `Utilities`BinaryFiles`` are present in `FastBinaryFiles`.

Suppose you have a file named `testdata` containing 16-bit integer data that you want to read into a *Mathematica* list. `ReadListBinary` is the function for this task:

```
In[3]:= data = ReadListBinary["testdata", SignedInt16] // Short
Out[3]//Short= {17999, 21069, 0, 17626, <<8811>>, 258, -1022}
```

As you can see, the resulting list consists of integers between -32678 and 32767, in accordance with the `SignedInt16` specification.

All the reading and writing functions take either a filename or a stream as their first argument. If you do the reading or writing all at once, you can just give a filename. If you need to do consecutive reads or writes, you should open a stream to the file. For example, you could write the data back to disk as follows:

```
In[4]:= stream = OpenWriteBinary["outfile"]
Out[4]= LinkOutputStream["Homer:Toolbox:outfile", 2]
In[5]:= WriteBinary[stream, data];
```

Note the return value from `OpenWriteBinary`: an object with head `LinkOutputStream`. The heads `LinkOutputStream` and `LinkInputStream` are created by `FastBinaryFiles` to distinguish input and output streams maintained by the external program from *Mathematica*'s standard `OutputStream` and `InputStream` types. `FastBinaryFiles` modifies the kernel functions `Close`, `Streams`, `StreamPosition`, and `SetStreamPosition` so that they work with the new stream types:

```
In[6]:= Streams[]
Out[6]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
LinkOutputStream["Homer:Toolbox:outfile", 2]}
```

The following input resets the stream position of `stream` to the beginning of the file, so that further writes begin at that point:

```
In[7]:= SetStreamPosition[stream, 0];
```

When you are finished with a stream, you can close it:

```
In[8]:= Close[stream];
```

How much faster is `FastBinaryFiles` compared to the standard package `Utilities`BinaryFiles``? A lot faster, as is demonstrated by the timings shown in Table 1. You can see from the results that the `ReadListBinary` in `Utilities`BinaryFiles`` is particularly slow. This is a consequence of design decisions made in the development of its code, not an inherent limitation of the speed of *Mathematica*. A considerably faster way to read binary files from within *Mathematica* is to use the `ReadList` function with the `Byte` datatype:

```
ReadList["filename", Byte]
```

This method reads a file as a series of unsigned 8-bit integers, and is even faster than `FastBinaryFiles`. However, for any other type of data, you need to do some computation with the byte values in *Mathematica*, so the timings increase considerably. For example, to read a file of 16-bit signed integers, you can use the following:

```
Apply[If[#1 > 127, (#1 - 256) 256 + #2, (#1 256) + #2]&,
Partition[ReadList["filename", Byte], 2], {1}]
```

Timings for methods based on `ReadList` with the `Byte` type are given in the middle row of the table (there is no timing for the `Double` type using this method because I didn't want to bother writing the complex calculation required). The `ReadList` timings, like the `Utilities`BinaryFiles`` timings, are not relevant for Windows users, since as discussed earlier, they cannot use any *Mathematica*-only method for reading or writing binary files.

Function	Byte	SignedInt16	Double
<i>Utilities`BinaryFiles`</i>			
ReadListBinary	> 2 hrs.	> 2 hrs.	> 2 hrs.
WriteBinary	86	74	2128
ReadList with Byte type	3	58	—
<i>FastBinaryFiles`</i>			
ReadListBinary	21	13	8
WriteBinary	19	11	6

TABLE 1. Wall-clock timings for reading and writing a 40 Kb binary file in various formats. Times are in seconds, on a 68040-based Macintosh.

There are a few minor differences in functionality between `FastBinaryFiles` and `Utilities`BinaryFiles``. One of the most important is in the `ByteOrder` option, which controls whether integers are read and written with their high-order or low-order bytes first (the option values `MostSignificantByteFirst` and `LeastSignificantByteFirst`, respectively). `FastBinaryFiles` adds a new default value, `Automatic`, which automatically detects the appropriate byte order for the processor that is running the external program.

Another difference concerns the options for the `WriteBinary` function. In `Utilities`BinaryFiles``, you specify a function to perform the conversion of the data to byte form with the `ByteConversion` option. The default is the supplied function `ToBytes`. If you have a list of integers that you want written in the `Int32` form, which is not the default form for integers in the `ToBytes` function, you need to use a line like:

```
WriteBinary["filename", intList,
ByteConversion->(ToBytes[#, IntegerConvert -> Int32]&)]
```

To simplify the syntax, `FastBinaryFiles` allows you to use:

```
WriteBinary["filename", intList, IntegerConvert -> Int32]
```

or simply,

```
WriteBinary["filename", intList, Int32]"
```

In other words, all the options for `ToBytes` are also options for `WriteBinary`. In fact, my `WriteBinary` doesn't even use `ToBytes` for byte conversion (or any other function you might specify). Thus, you cannot write your own function to use with the `ByteConversion` option.

Another important difference is in the way filenames are interpreted. When you give a filename to a built-in function like `OpenRead`, *Mathematica* uses its notion of a current directory and a `$Path` to locate the file. With `FastBinaryFiles`, the external program will not share these notions, and may even be running on a different computer with a different file system or operating system. To cope with this issue, the `FastBinaryFiles` code first expands the filename string you specify into a full pathname, which is what is actually passed across the link to the external program. This expansion is done in a way that attempts to mimic the behavior of the built-in functions. For example, you'll notice that in `In[4]` the filename is specified only as "outfile", whereas a full pathname appears in the `LinkOutputStream` object (the current directory was "Homer:Toolbox").

Under most conditions you need pay no attention to this issue. However, if you need to alter or suppress this filename expansion, you can do this with the `FilenameConversion` option in `OpenReadBinary`, `OpenWriteBinary`, and `OpenAppendBinary`. The default value for this option is `Automatic`, but you can specify `Identity` to turn off conversion, in which case the string you specify will be sent unaltered to the external program. You

can also specify your own function, which must be written to take a string and return a string.

For example, say you are running the kernel on a Macintosh, but the external program is running on a UNIX machine. You try:

```
In[9]:= OpenReadBinary["usr4/tgayley/myfile"]
OpenReadBinary::openf:
  The external program was unable to open the file
  Homer:Toolbox:/usr4/tgayley/myfile for reading.
Out[9]= $Failed
```

As you can see, the `FastBinaryFiles` code, which is in the kernel, expands this as if it were a Macintosh filename. To force it to be passed along unmodified, you can use

```
In[10]:= OpenReadBinary["usr4/tgayley/myfile",
  FilenameConversion -> Identity]
Out[10]= LinkInputStream["usr4/tgayley/myfile", 3]
```

***MathLink* Basics**

MathLink is a protocol for sending and receiving *Mathematica* expressions. Its uses fall into two general categories. The easiest and most common application is to allow external functions written in other languages to be called from within

the *Mathematica* environment. If you have an algorithm that needs to be implemented in a compiled language for efficiency reasons, or if you have code that you don't want to rewrite in *Mathematica*, it is a relatively simple matter to incorporate the routines into *Mathematica*. The `FastBinaryFiles` program uses *MathLink* in this way.

The second use of *MathLink* is to allow your program, running in the foreground, to use the *Mathematica* kernel in the background as a computational engine. In effect, the program is a “front end” for the *Mathematica* kernel. I will say no more about this type of application, except to note that Wolfram Research's notebook front ends for *Mathematica* use *MathLink* in this way. There is no “privileged” communication between the notebook front end and the kernel. *Mathematica*'s developers use the same set of calls as is available to you in the *MathLink* library.

I will refer to external functions that are called from *Mathematica* as “installable” functions, since they use the `Install` mechanism to be incorporated into the *Mathematica* environment. WRI provides a tool, the program called `mprep`, to assist in making it as easy as possible to call external functions. The intent is that you should be able to take preexisting C language routines, and with as little effort as possible (ideally with no source code changes to the routines themselves), package them so they can be called from *Mathematica*. For each function you want to call from *Mathematica*, you write a “template” that specifies the name of the function, the arguments that the function needs to be passed and their types, and the type of argument it returns. These templates are put into a file with the `.tm` suffix, which is processed by `mprep` into a C language source file that contains most, if not all, of the *MathLink*-related portions of the final program. The C source for the routines themselves is compiled with this source file to produce the final program. The actual steps for processing and compiling the files depend on the platform, and will be discussed in more detail at the end of the column. The `Install` function is then used to launch the program and make its functions available in a *Mathematica* session, as demonstrated in the previous section.

Let's look at a trivial example of an installable program, the `addtwo` program that is supplied with *MathLink*. We will modify the program in several ways to demonstrate some of the *MathLink* techniques used in the `FastBinaryFiles` code. Here is the C source file `addtwo.c`:

```
#include "mathlink.h"

int addtwo(i, j)
    int i, j;
{
    return i+j;
}

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

Note that if we already had a C routine that took two ints and returned an int, all we would have to do to make it installable would be to insert the one-line `main` function. (For Windows users, `main` is slightly more complicated, but that is not relevant here). The `main` function is simply a “stub” that calls the real `main` function (named `MLMain`), which is written by `mprep`.

Here is the template file `addtwo.tm`:

```
:Begin:
:Function:      addtwo
:Pattern:      AddTwo[i_Integer, j_Integer]
:Arguments:    { i, j }
:ArgumentTypes: { Integer, Integer }
:ReturnType:   Integer
:End:
```

The `:Function:` line specifies the name of the C routine. The `:Pattern:` line shows how the routine will be called in *Mathematica*. The pattern given on this line will become the left-hand side of a function definition, exactly as you would type it if you were creating the entire function in *Mathematica*. The `:Arguments:` line specifies the expressions to be passed to the external program. These expressions don't have to be the same as the variable names on the `:Pattern:` line, although they often will be. You could, for example, put `{Abs[i], j^3}`. The point is that what you put on the `:Pattern:` line and the `:Arguments:` line is *Mathematica* code; it will be used verbatim in a definition that could be caricatured as follows:

```
AddTwo[i_Integer, j_Integer] :=
    SendToExternalProgramAndWaitForAnswer[{i, j}]
```

The `:ArgumentTypes:` and `:ReturnType:` lines contain special keywords used by `mprep` to create the appropriate `MLGet` and `MLPut` calls that transfer data across the link.

Note that in writing the C source program and the template, we have not had to make a single *MathLink* call. With a little additional effort we can take more control over the passing of arguments and return values. This would be necessary, for example, if the external function needed to receive or return expression types that are not among the set handled automatically by `mprep`, or if the function returned different types of results (such as an integer or the symbol `Failed`) in different situations.

As an example, we will modify the `addtwo` program so that it works for larger integers, up to the `long` integer size. In the template file, the keyword `Integer` on the `:ArgumentTypes:` and `:ReturnType:` lines causes `mprep` to create calls to `MLGetInteger` and `MLPutInteger`, which transfer C ints. Instead, we need to call `MLGetLongInteger` and `MLPutLongInteger`, so we change these two lines:

```
:ArgumentTypes: { Manual }
:ReturnType:    Manual
```

The keyword `Manual` on the `:ArgumentTypes:` line informs `mprep` that we will write our own calls to get the arguments, and similarly `Manual` on the `:ReturnType:` line indicates that we will put the result ourselves. Here's how the `addtwo` function looks now:

```

void addtwo()
{
    long i, j, sum;

    MLGetLongInteger(stdLink, &i);
    MLGetLongInteger(stdLink, &j);
    sum = i + j;
    MLPutLongInteger(stdLink, sum);
}

```

Note the change in the function's prototype. Remember that the actual call to the `addtwo` function is made from code that `mprep` writes, so its arguments and return value must match `mprep`'s assumptions, as determined from the `:ArgumentTypes:` and `:ReturnType:` lines of the template. By specifying `Manual` on the `:ArgumentTypes:` line, we tell `mprep` to pass no arguments to `addtwo` when it is called. Similarly, by specifying `Manual` on the `:ReturnType:` line, we tell `mprep` to ignore any return value.

It is possible to use `Manual` on just one of these lines. It is also possible to mix `Manual` with other types on the `:ArgumentTypes:` line. For example, if we want to have the first argument read automatically, but get the second one ourselves, we can write:

```
:ArgumentTypes: { Integer, Manual }
```

In this case, the `addtwo` function would be written to take one `int` argument, and inside it there would be one call to `MLGetInteger`. If we use `Manual` on the `:ArgumentTypes:` line, it must be the last type in the list. In effect, `Manual` means "We want to get all the remaining arguments from the link ourselves." We cannot specify

```
:ArgumentTypes: { Integer, Manual, Integer }
```

The external function can request evaluations by *Mathematica* between the time it is called and the time it returns its result. For example, you might want *Mathematica* to assist you in computing something, or you might want to trigger some side effect such as displaying an error message. The *MathLink* function `MLEvaluate` is designed for this purpose. `MLEvaluate` takes a string argument that will be interpreted by *Mathematica* as input. The result will be returned to your function as an expression (not a string) wrapped with the head `ReturnPacket`. You should read this `ReturnPacket` off the link whether you care what is in it or not.

As an example, say we want to detect an overflow when adding the two `long` integers (that is, a sum that is outside the range of a `long`). If an overflow occurs, we want to show an error message in *Mathematica* and then return the symbol `$Failed` instead of the sum.

We can use `MLEvaluate` to trigger the message, but how do we get the definition of the message into *Mathematica* in the first place? We need to use one more feature of template files, the `:Evaluate:` line. Any *Mathematica* code included in a section of the template file that begins with `:Evaluate:` will be evaluated when the external program is first installed. We can put the definition for the error message in this section. Thus, we add the following line to the beginning of the `addtwo.tm` file:

```
:Evaluate: AddTwo::ovflw =
    "The sum cannot fit into a C long type."
```

The `addtwo` function now looks like this:

```

void addtwo()
{
    long i, j, sum;

    MLGetLongInteger(stdLink, &i);
    MLGetLongInteger(stdLink, &j);
    sum = i + j;
    if(i>0 && j>0 && sum<0 || i<0 && j<0 && sum>0) {
        MLEvaluate(stdLink, "Message[AddTwo::ovflw]");
        MLNextPacket(stdLink);
        MLNewPacket(stdLink);
        MLPutSymbol(stdLink, "$Failed");
    } else {
        MLPutLongInteger(stdLink, sum);
    }
}

```

After the call to `MLEvaluate`, *Mathematica* will send back a `ReturnPacket` containing the return value of the `Message` function (which is simply the symbol `Null`). We need to drain this packet off the link, so we call `MLNextPacket` (which will return `RETURNPKT`) and then `MLNewPacket` to discard the contents. If we wanted to read the contents of the `ReturnPacket`, we would replace `MLNewPacket` with an appropriate series of `MLGet` calls. If we think of packets as boxes, then `MLNextPacket` opens a box, and `MLNewPacket` discards an already-opened box. It is an error to call `MLNextPacket` if there is an open box that hasn't been fully emptied (you need to read out the entire contents or abandon it with `MLNewPacket`).

Using `MLEvaluate` is not the only way the external function can send code to *Mathematica* for evaluation. Anything sent wrapped in the head `EvaluatePacket` will be treated in this way. In fact, `MLEvaluate` is just a convenience function whose code does nothing more than create the expression

```
EvaluatePacket[ToExpression["the string"]]
```

and send it to *Mathematica*. After *Mathematica* calls our external function, it reads from the link, expecting to find the final result. The head `EvaluatePacket` tells *Mathematica* "This is not the final answer. Evaluate this and return the result to me wrapped in a `ReturnPacket`. Keep waiting for the final answer." In this way, the external function can initiate dialogs of arbitrary length and complexity with the kernel before it returns.

If it is most convenient to send the code we need evaluated as a string (for example, if the code is known at compile time), we can use `MLEvaluate`. In some cases, though, it may be easiest to send it as an expression. To use this method in the above example, we would replace the `MLEvaluate` line with the following lines:

```
MLPutFunction(stdLink, "EvaluatePacket", 1);
MLPutFunction(stdLink, "Message", 1);
MLPutFunction(stdLink, "MessageName", 2);
MLPutSymbol(stdLink, "AddTwo");
MLPutString(stdLink, "ovflw");
```

In case this looks unfamiliar, note that

```
MessageName[AddTwo, "ovflw"]
```

is the FullForm representation of `AddTwo::ovflw`.

Our `addtwo` function is still missing an extremely important aspect of *MathLink* programming: error-checking. Most *MathLink* functions return 0 to indicate an error has occurred, and you should always check their return values. If you continue to issue *MathLink* calls after an error has occurred, without clearing the error, the link will probably die. Checking for `MLGet` errors is handled for you by the code that `mprep` writes for any arguments that are read automatically. For `MLGet` calls that you write yourself, it's up to you.

The exact series of steps you take after an error has been detected depends on whether or not you want to try to recover. If an `MLGet` call fails, the easiest thing to do is simply to abandon the external function call completely and return the symbol `$Failed`. It would be more informative to trigger some kind of diagnostic message. The *MathLink* function `MLErrorMessage` returns a string describing the current error and this string is a good candidate for use in an error message to be seen by the user. Here is a code fragment that demonstrates how to detect an error, issue a useful message, and then safely bail out of the function call. For each `MLGet`-type call in your code, you can wrap it with something like:

```
if( !MLGetLongInteger(stdLink, &i) ) {
    char err_msg[100];
    MLClearError(stdLink);
    MLNewPacket(stdLink);
    sprintf(err_msg, "%s\\%.76s\\%s",
            "Message[AddTwo::mLink, ",
            MLErrorMessage(stdLink,
            "]);");
    MLEvaluate(stdLink, err_msg);
    MLNextPacket(stdLink);
    MLNewPacket(stdLink);
    MLPutSymbol(stdLink, "$Failed");
    return;
}
```

Naturally, if we have more than one or two `MLGet` calls in your code, we would want to implement this as a function or macro. Upon detecting the error, the first thing we do is call `MLClearError` to attempt to remove the error condition, and then `MLNewPacket` to abandon the rest of the packet containing the original inputs to the function (in case it hasn't been completely read yet). The `sprintf` is used to construct a string of the form:

```
"Message[AddTwo::mLink, \"the text returned by MLErrorMessage\"]"
```

which is what is sent to `MLEvaluate`. The gyrations required to produce this string using `sprintf` are a bit clumsy; this is getting close to a case where it would be easiest to send the code as an expression rather than a string, as demonstrated earlier. The remaining lines are the same as in the previous example. The message triggered here, `AddTwo::mLink`, needs to be defined in an `:Evaluate:` line in the `addtwo.tm` file as follows:

```
:Evaluate: AddTwo::mLink =
    "There has been a low-level MathLink error.
    The message is: `1`."
```

Once the `addtwo.tm` and `addtwo.c` files have been processed and compiled to produce an executable file called `addtwo`, we can install it into a *Mathematica* session:

```
In[1]:= Link = Install["addtwo"]
Out[1]= LinkObject[addtwo, 3, 2]
```

The `Install` function launches the program and opens a link through which the external function can be called from *Mathematica*. The program sends to *Mathematica* the definitions for its functions specified in the template file along with whatever code is given on the `:Evaluate:` lines. Of course, the programmer never sees any of this process, because it is handled at one end by the code that `mprep` writes and at the other end by the `Install` code. Most programmers have no reason to care how this feat is performed, but you should know that all the code involved is accessible. If you are interested in *MathLink*, you might want to take a look at a C source file produced by `mprep` (a `.tm.c` file) and at the *Mathematica* package `Install.m`, which resides in the `Startup` subdirectory of the `Packages` directory.

Now, let's see our error messages in action. The `AddTwo::ovflw` error is triggered when the two integers can be read from the link properly, but their sum is detected to have overflowed:

```
In[2]:= AddTwo[2000000000, 1000000000]
AddTwo::ovflw: The sum cannot fit into a C long type.
Out[2]= $Failed
```

The `AddTwo::mLink` error is triggered whenever the arguments are not read properly by `MLGetLongInteger`, which will happen if either one is too large to fit into a C long type:

```
In[4]:= AddTwo[5000000000, 1]
AddTwo::mLink:
    There has been a low-level MathLink error. The message is:
    machine integer overflow.
Out[4]= $Failed
```

Implementation

The `FastBinaryFiles` program is an interesting example of extending the *Mathematica* kernel. The program involves a rather complicated interaction between the external C code,

the *Mathematica* package code, and the kernel. Many interesting programming and design issues arose in its development (very few of which I have the space to discuss here). The relative ease with which it was written is a testament to the design of *MathLink* and its tight integration into the kernel.

The basic design is as follows. The functions that are visible to the user (such as `ReadBinary` and `WriteBinary`) are written in *Mathematica*. They perform the handling of options, some processing of arguments and error-checking, and other tasks that are easiest done in *Mathematica*. They then call private functions (such as `templateread` and `templatewrite`), which are the ones that are named in templates and map directly to functions in the external program. Some error messages are issued by the *Mathematica* code, but most errors can only be detected inside the external functions. Such errors include out-of-memory situations, inability to open files, failed *MathLink* calls, and so on. These error messages are all issued with `MLEvaluate` calls inside the external functions, as demonstrated in the `addtwo` example above. The external program is also responsible for maintaining the list of open streams, which the user sees as `LinkInputStream` and `LinkOutputStream` objects.

All the code (C, templates, and *Mathematica* package code) is included in the file `binary.tm`. The basic structure of `binary.tm` is as follows:

```
:Evaluate:  BeginPackage["FastBinaryFiles`"]
```

...all of the package code is here...

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

the C code begins:

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

...and so on...

templates begin:

```
:Evaluate:  Begin["FastBinaryFiles`Private`"]
```

```
:Begin:
```

```
:Function:      templateread
```

...and so on...

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

Note that it is possible to include C code in a `.tm` file. It is simply passed through unchanged by the `mprep` preprocessor. The entire package code appears in `:Evaluate:` sections of `binary.tm`, so that, in effect, `FastBinaryFiles` “bootstraps” itself by supplying its own code at runtime. This organization of source files is probably not ideal. I chose it for convenience in distribution. A more typical method would be to have a separate `.m` file containing the package code, a `.tm` file containing the templates, and one or more C source files. The problem with embedding the package code in the C code is that even the smallest change in the *Mathematica* code requires a recompile. For this reason, it is best to keep the package code separate during development. The advantage is that it takes only one step, launching the program with `Install`, to

use it in *Mathematica*. If the package file were separate, you would need to read it in before installing the program. It would not work to embed the call to `Install` within the package file, since there is no way to know the name or location of the external program.

The C code for reading and writing binary types is relatively straightforward, though tedious. I make no claims about its efficiency or elegance, but it accounts for only a small part of the execution time. What consumes most of the time? Part of the time is spent passing the newly-read list of data through the evaluator once it gets to *Mathematica*. The largest block of time, though, is consumed by sending the data across the link. *MathLink* sends all data in a textual form. Roughly speaking, if you call `MLPutInteger` to send the number 12345, it is converted and sent as the 5-byte string “12345”. The other side of the link has to convert the string back to its representation as a number. Why does *MathLink* take this obviously inefficient approach? The reason is that the two programs on either side of the link may be running on different computers, which may have completely different binary representations of numbers.

The next version of *MathLink* will be able to detect when both sides of the link share the same binary representations of numbers, and will transfer numbers (and arrays of numbers) in binary form in that case. A 10 Kb array of ints would then be sent as an image of the 10 Kb chunk of memory. This change will drastically speed up *MathLink* communication between compatible computers, and of course between two programs running on the same computer. To take advantage of this improvement, programmers should always use the function calls that correspond to the highest level of structure in the data when writing or reading a link. For example, to send a multidimensional array of integers, call `MLPutIntegerArray` instead of putting the function `List` and then calling `MLPutIntegerList` for each sublist. The `Array` functions and other new features of *MathLink* are documented in the pamphlet *Major New Features in Version 2.2*, which you should already have.

`FastBinaryFiles` uses the `Array` functions instead of the `List` functions because there are more of them, corresponding to more native C data types. For example, there is no `MLPutLongIntegerList`, but there is an `MLPutLongIntegerArray`, and it can be used to send a list as a special case. For the various integer and real types handled by `FastBinaryFiles` (`Int8`, `SignedInt8`, `SignedInt16`, `Single`, `Double`, and so on) there are corresponding `MLPut` and `MLGet` functions that can be used, perhaps with a little typecasting. There is one exception: the `Int32` type. There is an `MLPutLongInteger` function, but there is no `MLPutUnsignedLongInteger` and there is no integer type larger than 32 bits into which an unsigned long could be cast. For sending the `Int32` type, we need to use the “textual interface” functions. As discussed earlier, *MathLink* converts all data into a textual representation before it is sent. You can also perform this conversion manually, and this is required when sending data types that are not handled directly by existing *MathLink* functions. Here is how the `addtwo` example would look if we wanted it to be able to return numbers up to the magnitude of an unsigned long (ignoring for now the fact that numbers larger than a long will not be generated by the sum).

Note that the standard C library function `sprintf` can be used to perform the integer-to-string conversion.

```
void addtwo(i, j)
    int i, j;
{
    unsigned long sum, len;
    char the_string[12];

    sum=i+j;
    len = sprintf(the_string, "%lu", sum);
    MLPutNext(stdLink, MLTKINT);
    MLPutSize(stdLink, len);
    MLPutData(stdLink, the_string, len);
}
```

As mentioned earlier, the external part of `FastBinaryFiles` implements two new data types, `LinkInputStream` and `LinkOutputStream`. Of course, every function you create in *Mathematica* can be considered a new data type. What is interesting about these two new “stream” types is that they are implemented entirely outside of the kernel. All activities with these objects happen in the external program: creation, destruction, validation, reading, writing, management of their stream position markers, and so on. To the user, these types become transparent extensions to the kernel. They are “black box” objects whose definitions cannot be found in any *Mathematica* package file (including the `FastBinaryFiles` package).

I’m sure many readers can think of other types, data structures, and functions that could profitably be implemented in external programs. Once you begin to think about extending the *Mathematica* kernel, the true potential of *MathLink* becomes evident.

How To Build the Program

A ready-to-run executable is supplied for Macintosh and Windows. These will also be present on *MathSource*, along with the source code. You build the executable in the normal way you build template-based external functions, the exact details of which differ from platform to platform (as documented in the *MathLink Reference Guide*, and in the `README` files that come with the *MathLink* materials for Macintosh and Windows). There is only one source file, `binary.tm`. In addition to general package documentation, there is compiler- and platform-specific information and advice in this file. If you want to build the program yourself for Macintosh or Windows, you should look at this information. Note that the C code is ANSI C, so you must use an (at least mostly) ANSI-compliant compiler. This may cause a problem with using the `mcc` script provided with *Mathematica* on UNIX platforms, since this script calls the `cc` compiler, which is often not ANSI-compliant. The test is to try the standard command for compiling *MathLink* template programs under UNIX:


```
mcc binary.tm -o binary
```

If you get compiler errors, then you will have to resort to another method. If you have an ANSI-compliant compiler (like the GNU C compiler `gcc`), you can modify the `mcc` script so that it calls that compiler rather than `cc`. Alternatively, you can skip `mcc` altogether, and perform the steps by hand or create your own makefile. The steps for any template-based program are as follows: First, run `mprep` on the `.tm` file to create a `.tm.c` file. Then compile and link all the source files, including the `.tm.c` file (in the present case, this is the only file), specifying to the compiler where to find the `mathLink.h` file and the *MathLink* library file (named `libML.a` on UNIX machines). Of course, you can name the executable whatever you want.

Acknowledgements

I would like to thank Shawn Sheridan for his patient explanations of the past, present, and future of *MathLink*.

Todd Gayley
Wolfram Research, Inc., 100 Trade Center Drive,
Champaign, IL 61820-7237 tgayley@wri.com

 The electronic supplement contains the programs `binary` for Macintosh and `binary.exe` for Windows, along with the single source file `binary.tm`. UNIX users will need to build the program from the source.