A *MathLink* Tutorial

Todd Gayley
Wolfram Research

MathLink is a library of functions that implement 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*. This use of *MathLink* is treated in the first chapter of this tutorial.

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. This requires a deeper understanding of *MathLink*, and is treated in the second chapter.

Each of these two chapters is designed to stand on its own, so there is some repetition. There are also topics that are relevant to all *MathLink* programmers that are treated more fully in one chapter than in the other. I strongly recommend that you read both, but keep in mind that some of the information may not apply to you, depending on how you plan to use *MathLink*.

This document is designed to supplement the information in the *MathLink Reference Guide*, which is the main documentation for *MathLink*. There is also some information on newer features of *MathLink* in the *Major New Feautres of Mathematica Version 2.2* document, which comes with Version 2.2, and is also available on MathSource. My intention here is to flesh out some details, provide useful code fragments, discuss some underdocumented features, and show how to accomplish some common tasks.

The information presented here refers to Version 2.2.2 of *MathLink* and later. Most of the information is also correct for earlier versions, but a few of the functions and features may not be present.

1. Calling External Programs from the *Mathematica* Kernel

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. The intent is that you should be able to take pre-existing 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 entry 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. This template file is then passed through a tool called mprep, which writes C code that manages most, possibly all, of the *MathLink*-related aspects of the program.

I want to emphasize how easy, even trivial, it is to perform these steps for many external functions. With just a little more effort you can handle unusual functions or more sophisticated communication. The *MathLink Reference Guide* is perhaps a little intimidating, but some of the information is not directly relevant for programmers who merely want to call external functions from *Mathematica*. I hope that this chapter will encapsulate much of the information you need in a concise form.

1.1 The Simplest Example: addtwo

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 more advanced techniques. Here is the C source file addtwo.c:

```
#include "mathlink.h"

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

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

Note that if you already had a C routine that took two `int`s and returned an `int`, all you would have to do to make it installable would be to insert the one-line `main` function (actually, for Windows users `main` is slightly more complicated, but it is still something that can simply be pasted into your own code). 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.

The details of building the executable from the addtwo.tm and addtwo.c source files differ from platform to platform. On Unix, you will usually use the mcc script that comes with *Mathematica*. You would use a line like

```
mcc addtwo.tm addtwo.c -o addtwo
```

The steps that mcc performs are as follows: (1) run mprep on the .tm file, to create a .tm.c file; and (2) compile and link all the source files, including the .tm.c file, specifying to the 'cc' compiler where to find the mathlink.h file and the *MathLink* library file (named libML.a on Unix machines). It is the .tm.c file that contains the mprep-generated C source. Normally, this file is deleted by mcc after it has been compiled, but if you want to see what it looks like you can prevent its deletion by specifying the -g command-line option to mcc. Advanced users of *MathLink* can learn a lot by studying this file. On Macintosh and Windows, the steps to build the program will be different, and you should consult the README file that comes with *MathLink*.

The mcc method is convenient for simple projects, but it has some drawbacks, one of which is that it is hard-coded to call the 'cc' compiler. You might want to skip mcc altogether and write your own makefile. In that case, you will be calling mprep yourself. Here's an example:

```
/math/Bin/MathLink/mprep addtwo.tm -o addtwo.tm.c
```

Note that mprep is not on your Unix path, so you will need to specify the full pathname. The MathLink library, libML.a, is also located in the math/Bin/MathLink directory, and the mathlink.h file is in math/Source/Includes.

To use the **`AddTwo`** function in *Mathematica*, you launch the external program with the **`Install`** function:

```
link = Install["addtwo"]

LinkObject[addtwo, 2, 2]
```

The function **`LinkPatterns`** shows what functions are defined by the external program associated with a given link:

```
LinkPatterns[link]

{AddTwo[i_Integer, j_Integer]}

AddTwo[3,4]

7
```

You may wonder, "How does the definition for **AddTwo** appear in *Mathematica*?" After all, the only thing we've done is start up the kernel, type **Install**, and suddenly *Mathematica* knows about a function called **AddTwo**. The answer is that the external program sends to *Mathematica* the definitions for the functions it exports when the link is first opened. Here's what such a definition looks like:

```
?AddTwo

Global`AddTwo

AddTwo[i_Integer, j_Integer] :=
  ExternalCall[LinkObject["addtwo", 2, 2], CallPacket[0, {i, j}]]
```

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, you might want to take a look at a .tm.c file and the *Mathematica* package Install.m, which resides in the StartUp subdirectory of the Packages directory.

1.2 Using :Evaluate: to Include Accessory *Mathematica* Code

It was mentioned earlier that when the external program is installed it sends code to *Mathematica* to set up the "*Mathematica* side" of the functions it exports. You can also specify arbitrary *Mathematica* code to be sent. You might have some accessory code that your functions need to have exist in *Mathematica*. A simple example is usage messages.

You can specify arbitrary *Mathematica* code to be sent to the kernel when your program is installed by using another feature of template files, the `:Evaluate:` line. Here's an example of specifying a usage message:

```
:Evaluate:       AddTwo::usage = "AddTwo[i, j] adds two integers."

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

Defining messages is a trivial example of the use of `:Evaluate:` lines. Another common use is to make your functions appear in a package context. The current behavior of **Install** is to cause all functions defined in installable programs to appear in the **Global`** context, not the current

Mathematica context (this behavior may be changed in a future version). This means that if you want the **AddTwo** function to appear in a package context, say **MyPackage`**, then you cannot do this:

```
BeginPackage["MyPackage`"];

Install["addtwo"]

EndPackage[]
```

The **AddTwo** function will still be put into the **Global`** context. The best way to handle this is to put the **BeginPackage** statement into an `:Evaluate:` line in the .tm file:

```
:Evaluate:       BeginPackage["MyPackage`"]
:Evaluate:       AddTwo::usage = "AddTwo[i, j] adds two integers."
:Evaluate:       Begin["Private`"]

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

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

Everything that follows an `:Evaluate:` up until the first blank line or line whose first character is not a space will be sent as a single unit. This means you need to have a separate `:Evaluate:` for each separate statement or definition. There is more discussion of the use of `:Evaluate:` in Section 1.9, Large Projects.

1.3 Putting and Getting Arguments Manually

Note that in writing the addtwo program and the template, we have not had to make a single *MathLink* call. With a little additional effort you 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 `int`s. 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(void) {

   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, you tell mprep to pass no arguments to `addtwo` when it is called. Similarly, by specifying `Manual` on the `:ReturnType:` line, you tell mprep to ignore any return value.

It is possible to use `Manual` on one of these lines and not the other. It is also possible to mix `Manual` with other types on the `:ArgumentTypes:` line. For example, if you want to have the first argument read automatically but get the second one yourself, you 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 you use `Manual` on the `:ArgumentTypes:` line, it must be the last type in the list. In effect, `Manual` means "I want to get all the remaining arguments from the link myself". You cannot specify

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

It is likely that the arguments you will be passing to your function are among the set handled automatically by mprep (integers, reals, lists of these, strings, and symbols). In this case it is quite convenient to have mprep take care of this part of the *MathLink* communication. However, I recommend that you manually return results to *Mathematica*. It only takes one line of code to send simple types back, and for any of the more advanced *MathLink* techniques described below, you will need to have control over what is sent back and when.

1.4 Passing Lists and Arrays

Another case where you need to use the `Manual` keyword is when you need to return a list to *Mathematica*. The *MathLink* sample program bitops demonstrates this. For our purposes only one of the functions defined in bitops.c is relevant: the function `complements`, which takes a list of integers and returns a list of the bitwise complements of the integers. Here is the template entry in bitops.tm:

```
:Begin:
:Function:       complements
```

```
:Pattern:        BitComplements[x_List]
:Arguments:      {x}
:ArgumentTypes:  {IntegerList}
:ReturnType:     Manual
:End:
```

There is a keyword `IntegerList` that can be used on the `:ArgumentTypes:` line, so you can have mprep get the list for you, but you cannot use it in the `:ReturnType:` line--you have to use `Manual` and put the result list yourself. Here is the C function:

```
void complements(int px[ ], long nx) {

    long    i;
    int     *cpx;

    cpx = (int *) malloc(nx);
    for(i = 0; i < nx; i++)
        cpx[i] = ~ px[i] ;
    MLPutIntegerList(stdlink, cpx, nx);
    free(cpx);
}
```

Note that we have specified only one argument, an `IntegerList`, to be passed to the external function, but the function itself is written to take an integer array followed by a long integer. Confusion over this is a source of many user errors. When the mprep-generated code reads the list of integers, it will determine the length of the list and pass this to your function. Sometimes users mistakenly believe that they must themselves pass the length of the list from *Mathematica*, so they erroneously write the `:Arguments:` and `:ArgumentTypes:` lines like this:

```
:Arguments: {x, Length[x]}
:ArgumentTypes: {IntegerList, Integer}
```

The long parameter that will receive the length of the list always comes immediately after the list itself in the arguments to your function. For example, if you need to receive a list of integers, a list of reals, and an integer, you would write the `:ArgumentTypes:` line like this:

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

and the function prototype would look like:

```
void func(int ilist[ ], long ilen, double rlist[ ], long rlen, int j);
```

To put the result list back to *Mathematica*, you can use `MLPutIntegerList` or `MLPutRealList`.

In addition to putting and getting lists of integers and doubles, *MathLink* has some new functions for putting and getting multidimensional arrays in a single step, for example `MLGetDoubleArray` and `MLPutDoubleArray`. Check the mathlink.h header file for the complete set. The easiest way to describe these functions is to show a sample program. The following is an example function that creates an identity matrix of size n:

```
void identity_matrix(int n) {

     long   dimensions[2];
     char   *heads[2] = {"List", "List"};
     long   depth = 2;
     int    *mat;
```

```
        int    i,j;

        mat = (int*) calloc(n * n, sizeof(int));

        for(i=0; i<n; i++)
           for(j=0; j<n; j++)
              if(i == j) mat[i + j * n] = 1;

        dimensions[0] = dimensions[1] = n;

        MLPutIntegerArray(stdlink, mat, dimensions, heads, depth);

        free(mat);
    }
```

The "`Array`" functions are similar to their "`List`" counterparts. In a `PutArray` function, instead of a `long` length parameter, you pass an array of `long`s giving the length in each dimension. The `heads` parameter is an array of `char*` that give the heads in each dimension (**`List`** in most cases). If the heads are **`List`** in each dimension, you can simply pass `NULL` in place of `heads`.

Here's a complete example showing the use of `MLGetDoubleArray` and `MLPutDoubleArray`. The function transposes a matrix of reals:

```
:Begin:
:Function:         transpose
:Pattern:          MyTranspose[l_?MatrixQ]
:Arguments:        {l}
:ArgumentTypes:    {Manual}
:ReturnType:       Manual
:End:

void transpose(void) {

   long   *dimensions;
   char   **heads;
   long   depth;
   double *data;
   int    i, j;
   double *tdata;          /* put the transposed array here */
   long   tdimensions[2];  /* reverse of dimensions */

   MLGetDoubleArray(stdlink, &data, &dimensions, &heads, &depth);

   tdata = (double*) malloc(sizeof(double)*dimensions[0]*dimensions[1]);

   for(i=0; i<dimensions[0]; i++)
      for(j=0; j<dimensions[1]; j++)
         tdata[i + j * dimensions[0]] = data[j + i * dimensions[1]];

   tdimensions[0] = dimensions[1];
   tdimensions[1] = dimensions[0];

   MLPutDoubleArray(stdlink, tdata, tdimensions, heads, 2);
```

```
        free(tdata);
        MLDisownDoubleArray(stdlink, data, dimensions, heads, depth);
    }
```

Note the call to `MLDisownDoubleArray`. Whenever you use `MLGet` to receive an object whose size cannot be known at compile time (e.g., a string, symbol, list, or array), *MathLink* reads the object into its own memory space and gives you only the address of the data. For example, in `MLGetString`, you pass the address of a `char*` (*i.e.*, a `char**`), and *MathLink* stuffs the address of the string it received into your `char*`. You'll note that you haven't had to allocate any memory yourself or worry about how big the data is. At this point, *MathLink* "owns" the data, and it is waiting for your permission to free the memory that it occupies, which you grant when you call the `MLDisown` functions. Between the time you call `MLGet` and `MLDisown`, you can only read the data--do not try to modify it in place. If you need to do that, allocate your own memory and copy the data into it (*e.g.*, using `strcpy`).

Note that you need to worry about calling `MLDisown` functions only if you call `MLGet` yourself. For strings, symbols and lists that mprep gets automatically for you, it takes care of calling the appropriate `Disown` functions after your function returns.

1.5 Passing Arbitrary Expressions

MathLink has functions for passing all native C types, along with single- and multidimensional arrays. There are times, though, when you need to send or receive expressions that do not fit neatly into C types. Your function might need to return a list of mixed integers and reals, or a list of lists that is not a matrix, or something even more complicated like **`Integrate[x^2, {x,0,1}]`**. How do you go about transferring expressions like these?

I will focus on returning such expressions from an external function. It is less likely that your function would want to receive such expressions. It is certainly possible to receive complex expressions, but what would you do with them? You'd have to write your own code to analyze them and extract the desired information. If you need to deal with complicated expressions in your external functions, you'd be better off writing some code on the *Mathematica* side that acts as a "wrapper" around your template functions, manipulating and decomposing the expressions into meaningful C-size chunks, and sending these instead. This type of chore is more easily programmed in the *Mathematica* language.

You send expressions over *MathLink* in a way that mimics their **`FullForm`** representation. There are *MathLink* functions for the necessary "atomic" types (integers, reals, strings, and symbols), and if you need to put a "composite" expression (something with a head and zero or more arguments), you use `MLPutFunction` to put the head and the number of arguments, then `MLPut` calls for each of the arguments in turn. For example, to put the **`Integrate`** expression above, you would use:

```
    MLPutFunction(stdlink, "Integrate", 2);
        MLPutFunction(stdlink, "Power", 2);
            MLPutSymbol(stdlink, "x");
            MLPutInteger(stdlink, 2);
        MLPutFunction(stdlink, "List", 3);
            MLPutSymbol(stdlink, "x");
```

```
        MLPutInteger(stdlink, 0);
        MLPutInteger(stdlink, 1);
```

Of course, if you want to return an expression like this from your function, you will need to declare a `Manual` return type in the .tm file.

A very common error is attempting to put more than one expression from the external function. An external function, just like any built-in function, cannot return two things. In the earlier examples, we sent complex expressions back to *Mathematica*, but always only one of them. Here is an example of this error:

```
void return_two(void) {

    int i, j;

    MLGetInteger(stdlink, &i);
    MLGetInteger(stdlink, &j);

    MLPutInteger(stdlink, i);
    MLPutInteger(stdlink, j);
}
```

The two integers returned need to be wrapped in a head of some sort so that they become part of a single expression. The put calls need to be written like this:

```
MLPutFunction(stdlink, "List", 2);
   MLPutInteger(stdlink, i);
   MLPutInteger(stdlink, j);
```

1.6 Requesting Evaluations by the Kernel

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 wrapped with the head **`ReturnPacket`**. You should read this **`ReturnPacket`** from the link whether you care what is in it or not.

As an example, let's go back to the `addtwo` function and say you 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, you want to show an error message in *Mathematica* and then return the symbol **`$Failed`** instead of the sum.

You can use `MLEvaluate` to trigger the message, but how do you get the definition of the message into *Mathematica* in the first place? You use an `:Evaluate:` line in your .tm file:

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

The `addtwo` function now looks like this:

```
void addtwo(void) {
```

```
        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`**). You need to drain this packet from the link, so you call `MLNextPacket` (which will return `RETURNPKT`) and then `MLNewPacket` to discard the contents. If you wanted to read the contents of the **`ReturnPacket`**, then you would replace `MLNewPacket` with an appropriate series of `MLGet` calls. As an example, let's say you wanted to have *Mathematica* compute a Bessel function for you. Here's how you would send the request and read the result:

```
MLEvaluate(stdlink, "BesselJ[0, 5.0]");
MLNextPacket(stdlink);             /* a RETURNPKT will be waiting */
MLGetDouble(stdlink, &my_double);  /* inside there will be a real */
```

`MLNextPacket`, `MLNewPacket`, and the `MLGet` functions are discussed in more detail in the second chapter of this tutorial, where they are used more extensively.

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 merely a convenience function whose code just creates the expression **`EvaluatePacket[ToExpression["the string"]]`** and sends this to *Mathematica*.

After *Mathematica* calls your 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 **`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 you need evaluated as a string (for example, if the code is known at compile time), you can use `MLEvaluate`. In some cases, though, it may be easiest to send it as an expression wrapped in **`EvaluatePacket`**. In the above example computing **`BesselJ`**, it is likely that the arguments to **`BesselJ`** will be variables in your own program, not constants embedded in a string. Rather than constructing a string and sending it with `MLEvaluate`, you might want to replace the `MLEvaluate` line with the following lines:

```
MLPutFunction(stdlink, "EvaluatePacket", 1);
  MLPutFunction(stdlink, "BesselJ", 2);
    MLPutInteger(stdlink, my_int);
    MLPutDouble(stdlink, my_double);
MLEndPacket(stdlink);
```

You read the resulting **ReturnPacket** in the same way as before.

1.7 Error Handling

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 check their return values, at least for the reading functions. If you continue to issue *MathLink* calls after an error has occurred, without clearing the error, things will no longer work as expected. Specifically, the link simply refuses to do anything until you clear the error. Checking for `MLGet` errors is handled for you by the code that mprep writes for any arguments that are read automatically. If you don't do any manual getting of arguments, then you don't have to worry about error checking. For any `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 you want to try to recover or not. If an `MLGet` call fails, the easiest thing to do is to simply abandon the external function call completely and return the symbol **$Failed**. It would be more informative to trigger some kind of diagnostic message. There is a *MathLink* function called `MLErrorMessage`, which 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];
      sprintf(err_msg, "%s\"%.76s\"%s",
           "Message[AddTwo::mlink,",
           MLErrorMessage(stdlink),
           "]");
      MLClearError(stdlink);
      MLNewPacket(stdlink);
      MLEvaluate(stdlink, err_msg);
      MLNextPacket(stdlink);
      MLNewPacket(stdlink);
      MLPutSymbol(stdlink, "$Failed");
      return;
}
```

Naturally, if you have more than one or two `MLGet` calls in your code, you would want to implement this as a function or macro. Upon detecting the error, the first thing you 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 of `MLEvaluate`. 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`."
```

Now let's see these error messages in action. Earlier, we introduced the **AddTwo::ovflw** error message, to be triggered when the two integers can be read from the link properly, but their sum is detected to have overflowed:

```
AddTwo[2000000000, 1000000000]

AddTwo::ovflw: The sum cannot fit into a C long type.

$Failed
```

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

```
AddTwo[5000000000, 1]

AddTwo::mlink:
   There has been a low-level MathLink error. The message is:
    machine integer overflow.

$Failed
```

1.8 Troubleshooting and Debugging

If you get either one of these two errors when you try **Install["progname"]**:

```
LinkOpen::linkf: LinkOpen[progname] failed.

LinkConnect::linkc: LinkObject[progname, 1, 1] is dead; attempt to connect failed
```

then either the program is not being found, or it is launching and then immediately crashing. If you **Install** a program that exists but is not properly *MathLink*-aware, then **Install** will hang until you abort it. **Install** does not interpret the string you give it, and in particular it does not search the directories on **$Path** (this behavior may change in the future). The directories it does search are dependent on factors outside of *Mathematica*, such as the operating system and shell. On Unix, for example, the path that is searched is the path inherited by shell processes launched by the kernel. You may need to give a complete pathname to the program. To make sure that your program is at least minimally able to run, simply launch it from the command line (under Unix) or by double-clicking it (Macintosh or Windows). You should get a "Listen on:" prompt, which you can dismiss, followed by a "Connect to:" prompt, which you also dismiss, at which point the program will exit.

If your program passes the above test, but otherwise behaves unexpectedly, then a few simple debugging techniques will likely pinpoint the error. If the program crashes because of something in your computational code, or if it exits because you are using *MathLink* calls incorrectly, you will probably see the following message:

```
LinkObject::linkd:
   LinkObject[progname, 18, 3] is closed; the connection is dead.
```

In most cases, there is a simple error in your *MathLink* code. Most of the *MathLink* functions return 0 to indicate that an error has occurred. Go back into your source and insert statements to check the return values of each *MathLink* function (start with the reading functions--`MLNextPacket`, `MLNewPacket`, and anything with `Get` in its name).

If you want to run your installable program with a debugger, you will generally need to launch it inside the debugger, and then establish a connection with *Mathematica* manually, rather than having *Mathematica* launch your program automatically. This issue is discussed in the *MathLink Reference Guide*, along with an example using the Unix gdb debugger. The details differ from platform to platform, but the concept is the same. One side of the link needs to open in Listen mode, and the other side then uses Connect mode to connect to that listening link. Which side does which is not important; in my example I reverse the roles in the *MathLink Reference Guide* example. Begin by launching the program in your debugger. You will get a "Listen On" prompt, to which you give an arbitrary link name (on Macintosh and Windows, these are arbitrary strings, like myLink; on Unix, they will be numbers, 5000 for example). Now, switch to *Mathematica* and type:

```
link = Install["name", LinkMode->Connect]
```

where **`name`** is the linkname you specified to the "Listen On" prompt. Use string quotes around the name, even if it is a number (you don't use string quotes earlier, when you reply to the Listen On prompt). Note that **`Install`** can take the same sort of arguments that **`LinkOpen`** takes. Here, we give a linkname as the first argument (when we want *Mathematica* to launch the program, this is just the filename), and specify link options as well.

1.9 Large Projects

The examples so far have all been single functions. They are a good model for the occasional numerical function that you want to incorporate into *Mathematica*. The potential for installable functions is much greater, though. You can create entire packages or sets of packages, implemented in one or more external programs, that effectively "graft" new capabilities onto the kernel. Some special issues arise when considering larger projects based on installable programs.

First, you will undoubtedly need to write some *Mathematica* code to go along with your C functions. I suggest writing "wrapper" functions in *Mathematica* that perform the handling of options, some processing of arguments and error checking, and other tasks that are more easily done in *Mathematica*. These are the functions that are visible to the user, and they then call private functions that are the ones named in templates and map directly to functions in the external program. You can develop very sophisticated interactions between the C and *Mathematica* code.

Through the use of `:Evaluate:` lines in your .tm file, you can embed your entire package code in the program source files, so that there is no separate .m file to be loaded. The advantage to this is convenience for users (they can just **`Install`** the program and be ready to go), but the disadvantage is that any modification of the package code requires that the program be recompiled. Chances are that your users will not be doing this, though, and during development you can keep the package code in a separate .m file.

The basic decision is whether you will have the package code embedded in the external program, so what the user types is **`Install["progname"]`**, or have a package (.m) file that calls **`Install`** within it, so what the user types is **`<<Packagename``**. The problem with the latter approach is that users need to either: (1) always give the program a predetermined name and always put it someplace it will automatically be found by **`Install`**, or (2) edit the .m file to

reflect what they choose to name the program and the pathname where they put it. The advantage to this approach is that it makes your program behave more like a seamless extension to the kernel. Specifically, the context-handling functions will work correctly with it, so that users use **`Get`** and **`Needs`** with your package name just like any other package name, and may even be unaware that an external *MathLink* program is involved. Having written significant programs that use both approaches, I recommend the second method, writing your package code in a .m file that calls **`Install`** internally.

You can embed C code in a .tm file, and it will be passed along unchanged by mprep. This means that you don't need a separate .c file, and this is convenient if your code is not long or complicated. In fact, all your code--templates, package code, and C code--can be included in one .tm file if desired. Here is a sample of the suggested structure of such a .tm file:

```
:Evaluate:   BeginPackage["MyPackage`"]
```

All of the package code is here, in :Evaluate: sections...

```
:Evaluate:   FirstFunction::usage = "FirstFunction does..."
```

etc....

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

The C code begins:

```
#include "mathlink.h"

void template_func1() { ...
```

etc....

Templates begin:

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

:Begin:
:Function:   template_func1
```

etc....

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

If you are writing a commercial-quality program, make sure that your external functions behave as if they were well-written built-in functions. This means, among other things, that they should be abortable, and they should return *Mathematica*-style messages for all errors or warning conditions.

1.10 Special Topics

■ 1.10.1 If You Don't Know the Length of the Result

Notice that you have to specify the number of arguments that will follow in every `MLPutFunction` call. Sometimes it is inconvenient to have to know ahead of time the number of arguments that you will send. For example, you might be running a loop, generating one element of the result list in every iteration, and you don't know ahead of time when the loop will end. There are a couple of tricks for getting around this problem.

One method doesn't involve *MathLink* at all--you just allocate enough local storage in your C program to hold all the elements, counting them as you place them in this storage, and when you are finished you put them on the link in the usual way. This is relatively easy, except you have to deal with the hassle of memory management in C. You may need to do a lot of allocating and reallocating memory to hold the result as it grows, and you need to be sure you free it all before returning.

It would be easier just to send the elements as they are generated. Then you would need to allocate only enough storage to hold a single element, reusing the same space for each successive element. One way to do this is to create a nested list wrapped in **`Flatten`**. If you think about it, you'll see that you never have to make any promises about the total number of elements that will appear in the final flattened list. Every sublist contains two elements: an integer (in this particular example) and another sublist. When all the integers you need have been sent, you send two empty lists (to fulfill the final promise of two arguments), which will be obliterated by the **`Flatten`**. The expression that is sent might look like **`Flatten[{1,{2,{3,{4,{{},{}}}}}}]`**, which evaluates to **`{1,2,3,4}`**. It's actually a bit more complicated, since if you send a list that is nested too deeply, you will hit *Mathematica*'s **`$RecursionLimit`**, and trigger an error. The way around this is to temporarily set **`$RecursionLimit`** to **`Infinity`**, which is best done by localizing its value in a **`Block`**. Thus, the actual *Mathematica* code you will send will look like:

```
Block[{$RecursionLimit = Infinity},
      Flatten[{1,{2,{3,{4,{{},{}}}}}}]
]
```

The sequence of *MathLink* calls to send this is straightforward:

```
MLPutFunction(stdlink, "Block", 2);
MLPutFunction(stdlink, "List", 1);
MLPutFunction(stdlink, "Set", 2);
MLPutSymbol(stdlink, "$RecursionLimit");
MLPutSymbol(stdlink, "Infinity");
MLPutFunction(stdlink, "Flatten", 1);
MLPutFunction(stdlink, "List", 2);
while(not_finished) {
   /* Here is the computation that generates the elements of the
      result. This would probably be the main computational section
      of your function. */
   i = generate_next_element();
   MLPutInteger(stdlink, i); /* or whatever the list elements are */
   MLPutFunction(stdlink, "List", 2);
```

```
}
MLPutFunction(stdlink, "List", 0);
MLPutFunction(stdlink, "List", 0);
```

This may look complicated, but it's just "boilerplate" code that can be pasted into your program where necessary.

If the elements of the result list are themselves lists, then **`Flatten`** will not work since it will flatten out the sublists as well. You can use **`Level`** instead in this case. **`Level[listOfLists, {-2}]`** extracts those expressions of depth 2 from the nested list, which is what you want if the elements of the outer list are simple lists. If they are matrices, use **`{-3}`** as the **`Level`** specification, since a matrix has depth 3.

This is an interesting example because what you send back to *Mathematica* is in effect a "program", the execution of which produces the results. Of course, every *Mathematica* expression is a "program" and vice-versa, but it is a conceptually useful mental leap here. There are lots of other programs you can send that will evaluate to the desired list (a method similar to the above could be based on **`Join`**), and you can even do something as fancy as sending back a program that itself reads from the link, collecting the elements until it reads an "END" marker. Once you start thinking in these terms, a wealth of sophisticated interactions become possible.

Another method for dealing with the problem of not knowing ahead of time how many elements will be in the result list involves the use of a "loopback link", and it is described in the next section. This method is the most elegant and probably the most desirable, except in cases where the speed of the *MathLink* transfer is the most important consideration.

■ 1.10.2 Loopback Links

Beginning in Version 2.2 of *Mathematica*, a new link mode was introduced--the "loopback" mode (joining Launch, Listen, and Connect). This link type is quite useful, but underused by *MathLink* programmers. Brief documentation for loopback links can be found in the *Major New Features of Mathematica Version 2.2* document.

A loopback link is a link that "points back" at you. You both write to and read from it. You can think of it as a U-shaped track onto which you can place expressions for storage and later examination or retrieval. If you think about it, you'll see that loopback links effectively give the C progammer a *Mathematica* expression "type".

There are a lot of interesting things you can do with loopback links, but I will focus on one application of great use in installable programs. This is to solve the problem discussed in the previous section: how to send an expression (like a list) back to *Mathematica* when you don't know in advance how many arguments it will have. The loopback link provides a very simple solution--as you generate the elements of the result list, put them on a loopback link, not the link back to the kernel, counting them as you go. Then when it comes time to send them to *Mathematica*, you know how many there are, and you can specify this when you use `MLPutFunction` to put the enclosing **`List`**.

The loopback link method for solving this problem has an additional advantage over the nested list method mentioned in the last section. It may be that during the generation of the result list you encounter an error condition or some other circumstance where you no longer want to send the list at all (you might want to send the symbol **`$Failed`** instead, and you might want to trigger an error message). If you are putting the result on a loopback link, you don't send anything to the kernel until the computation is finished, and you can decide at that time to send whatever you want.

You open a loopback link in the usual way, except you specify "loopback" as the linkmode. Let's look at a complete function that returns a list of integers to *Mathematica* by first placing them on a loopback link.

```
void foo(void) {

    int i, num_elements;
    char loopback_argv[3] = {"-linkmode", "loopback", NULL};
    MLINK loopback;

    loopback = MLOpen(2, loopback_argv);
    if(!loopback) {
        /* might want to issue a message as well */
        MLPutSymbol(stdlink, "$Failed");
        return;
    }

    num_elements = 0;
    while(some_test) {
        i = generate_next_element();
        MLPutInteger(loopback, i);
        num_elements++;
    }

    MLPutFunction(stdlink, "List", num_elements);
    for(i=1; i<=num_elements; i++) MLTransferExpression(stdlink, loopback);

    MLClose(loopback);
}
```

Note the use of `MLTransferExpression` to move the integers from the loopback link to the kernel link. This function is described in the *Major New Features of Version 2.2* document. It provides a very convenient means for moving expressions from one link to another, since you don't need to be concerned with the exact structure of each expression. The destination link is given first, the source link second.

The above method is similar to storing the integers in memory allocated and maintained by you in your C program. The advantage of using a loopback link is that you let *MathLink* deal with all the memory-management issues. There are no calls to `malloc`, `realloc` or `free`, or checks for writing past the end of your allocated storage. Memory management would not be difficult in the case of a list of integers, but if you were accumulating a list of strings or functions, it would be a big chore, with many possibilities for memory leaks and other bugs.

I cheated a bit, though, in not checking for errors in the `MLPut` calls onto the loopback link. An `MLPut` might trigger a memory allocation inside *MathLink*, which could conceivably fail. You don't really need to worry about this when writing to `stdlink`, the link to the kernel, because that link is being drained by the other side as you pour data into it. A local link will require enough memory to hold all the data at once, so you should check for errors in the `MLPut` calls if you are storing a lot of data.

■ 1.10.3 Making Your Function Abortable

If your function takes significant time to execute, you will want to make it abortable. That is, when the user types the usual abort key sequence (Control-C in Unix, Command-period on Macintosh, etc.), the function should terminate as quickly as possible and return something appropriate.

To understand how this is done, you need to know that a link actually contains two separate "channels" of communication. One channel is for the expressions being sent back and forth, and the other one is for urgent messages that need to be sent out of sequence with the flow of expressions. Examples are requests to interrupt or abort execution. This second channel is the one that is managed by the "`Message`" functions in *MathLink*: `MLPutMessage`, `MLGetMessage`, and a few others. Don't confuse these with the `MLErrorMessage` function (which returns a string describing an internal *MathLink* error), or the familiar *Mathematica* error messages.

Normally, programmers writing installable functions don't need to worry about the low-level details at all. Handling messages from the kernel is performed in code that mprep writes for you. All you need to know is that there is a global variable `MLAbort` in installable programs whose value will reflect whether or not the user has requested that the function be aborted. If you are running a time-consuming loop, you should periodically check the value of `MLAbort`. If it is non-zero, then you should bail out as quickly as possible.

What should your function return to *Mathematica* if the user aborts the evaluation? A quick answer might be the symbol **`$Aborted`**. Indeed, this is what the *MathLink Reference Guide* suggests, and this is what your function will return if you do not use `Manual` as the return type in the template, because then it is the mprep-generated code that takes care of sending the final answer to *Mathematica*. That code checks the variable `MLAbort`, and if it is set, **`$Aborted`** is sent no matter what your function returns.

However, sending **`$Aborted`** is probably not the ideal behavior. After all, when the user aborts a calculation that does not involve an external function, the entire evaluation aborts, and **`$Aborted`** is returned as the **`Out[]`** value. It is not the case that whatever function was executing at the time the abort was requested returns **`$Aborted`**. That is, if you evaluate **`f[g[x]]`**, and you abort during the execution of **`g[x]`**, you don't get **`f[$Aborted]`** as the result. Unfortunately, if **`g`** was an external program that returned **`$Aborted`**, this is what you would get. If you're writing a program that involves calling external functions, you don't want to worry that an expression deep inside your code is going to evaluate to **`$Aborted`** instead of something meaningful simply because the user tried to abort at an inopportune time!

While the external function is executing, *Mathematica* captures abort requests and sends them to the function as *MathLink* messages. As a *MathLink* programmer, if you want your functions to behave like built-in ones, it is your responsibility to "propagate" the abort request back to *Mathematica*'s normal abort-processing mechanisms, which are restored when the external function returns. You do this by returning not the symbol **`$Aborted`**, but rather the function **`Abort[]`**. If you return **`Abort[]`**, then *Mathematica* will halt the entire evaluation no matter how deep inside it, just like it does with programs written entirely in *Mathematica*.

The fact that the abort behavior of mprep-generated code is not ideal is another reason to routinely use the `Manual` return type and put the result to *Mathematica* yourself.

Earlier, I said that all you need to know is that there is a global variable `MLAbort` in installable programs. This isn't strictly true, however--there is one more detail you need to be aware of. On systems without preemptive multitasking (Macintosh and Windows), your function needs to yield the processor so that the kernel has a chance to actually send the abort message to you. You could write calls to an appropriate function depending on the platform (*e.g.*, on Macintosh, call `WaitNextEvent`), but there is an easier solution that will keep your code portable between systems. *MathLink* supports something called a yield function, which is discussed more fully in Section 2.5.2. For now, simply note that template programs define and install a yield function (it is written by mprep) that calls the appropriate function to yield the processor temporarily to other programs. Therefore, you can simply call the yield function periodically during your calculation.

Note that many *MathLink* functions call the yield function themselves internally (including the `Put`-type calls). Therefore, you don't need to worry about calling the yield function if you are making `MLPut` calls during your computation (for example, if you are putting pieces of the result on the link periodically during your calculation). You need to call the yield function only if your function doesn't make any `MLPut` calls before checking the value of `MLAbort`. You also don't need to do it if you are running under Unix (but you might want to, for portability reasons).

Here is a skeleton of a template program that performs a long calculation and periodically checks `MLAbort`. Note that `MLCallYieldFunction` is new in Version 2.2.2, and don't worry about what the arguments mean--just copy the code exactly as it appears below.

```
void long_function(void) {

   int result = 1;

   while(some_test && !MLAbort) {
      result = perform_computation(result);
      MLCallYieldFunction(MLYieldFunction(stdlink), stdlink,
                              (MLYieldParameters)0 );
   }

   if(MLAbort) {
      MLPutFunction(stdlink, "Abort", 0);
        /* Contrast with:  MLPutSymbol(stdlink, "$Aborted"); */
   } else {
      MLPutInteger(stdlink, result);
   }
}
```

What if you are sending elements of a result list to *Mathematica* as you go, so that when you detect an abort you have already sent a partial answer? You cannot "take back" what you've sent and send something else (like **`Abort[]`**) instead. However, if you simply call `MLEndPacket` in the middle of sending an expression (*i.e.*, at a place where `MLEndPacket` is illegal), *Mathematica* will get the symbol **`$Aborted`** by default.

In summary, a time-consuming function should periodically check the value of `MLAbort`. You might need to call the yield function periodically, to allow the kernel process to send the abort message to you. If you are putting the result back to *Mathematica* manually, you should send the function **`Abort[]`** and return. If you are not using `Manual` on the `:ReturnType:` line of the template file, then you should immediately return a value from your function (it can be garbage, since it will not be sent to *Mathematica* anyway). The mprep-generated code will send

`$Aborted` in its place. Finally, if you have already sent part of the result by the time you detect the abort, you should just call `MLEndPacket`, which will cause *Mathematica* to get the symbol **`$Aborted`** by default.

2. Calling the *Mathematica* Kernel from External Programs

Using *MathLink* to "install" external programs into *Mathematica* is very useful, but it only scratches the surface of what can be done. The real power of *MathLink* is that you can add the computational and programming services of the *Mathematica* kernel to your own programs. *MathLink* is not just a way to control the kernel, it is *the* way. When you use the standard "notebook front end" that ships with *Mathematica*, you are using *MathLink* in this way. There is no privileged communication between front end and kernel--everything takes place via the same open, documented set of *MathLink* functions that is available to all programmers.

Using *MathLink* to drive the kernel is more complex than writing installable functions because you have to write all the *MathLink* code yourself, and you will be having more complicated interaction with the kernel.

It is important to remember that you do not use the kernel as if it were a compiled library of mathematical routines. Rather, you are interacting with a separate program that has its own thread of execution. You will be running the kernel in so-called "mathlink mode", which means that all the kernel's input will come from your program, all its output will be directed to your program, and all communication will take place in the form of "packets". You have to know the proper way to send expressions to the kernel, what type of results to expect back, and how to read them off the link.

Two useful resources you might not be aware of are available on *MathSource*. The first is the Macintosh program Link Tutor, written by this author, which gives you a point and click interface for executing *MathLink* functions. You can execute *MathLink* functions one at a time in an interactive session and see the results. It is a good tool for learning what types of packets the kernel will send you under certain conditions, in what order, and what their contents are. The other resource is frontend.c, a small C program that implements a more or less complete terminal-like interface to the kernel. This is a good place to see code for reading out the contents of all of the packet types you might get from the kernel.

2.1 A Simple Program

Let's look at a simple example of a program that uses the kernel for computation. This program will launch the kernel, have it calculate the sum of two integers, then close it and quit. We will look at these specific programming techniques in detail later; for now note the general idea and how simple it is.

```
#include <stdio.h>
#include "mathlink.h"

int main(int argc, char * argv[ ]) {

    int     i, j, sum;
```

```
    MLINK   lp;
    int     pkt;
    MLEnvironment env;

    printf( "Enter two integers:\n\t" );
    scanf( "%d %d", &i, &j);

    env = MLInitialize(NULL);
    if(env == NULL) return 1;
    lp = MLOpen(argc, argv);
    if(lp == NULL) return 1;

    /* Send Plus[i, j] */
    MLPutFunction(lp, "Plus", 2);
      MLPutInteger(lp, i);
      MLPutInteger(lp, j);
    MLEndPacket(lp);

    /* skip any packets before the first ReturnPacket */
    while (MLNextPacket(lp) != RETURNPKT)  MLNewPacket(lp);

    /* inside the ReturnPacket we expect an integer */
    MLGetInteger(lp, &sum);

    printf( "sum = %d\n", sum);
    MLClose(lp);
    MLDeinitialize(env);
    return 0;
}
```

2.2 Opening a Link to the Kernel

■ 2.2.1 MLOpen

The *MathLink* function that opens a link is `MLOpen`. There are a lot of details about how links are opened, what protocols are used, and so forth, that are treated in the *MathLink Reference Guide*. I will just briefly discuss some of the main points. You can use `MLOpen` to launch the kernel directly from your program, and this is probably what you will most often want to do. `MLOpen` takes an `argc/argv` pair of arguments, like the `main` function of a C program. This is so you can pass the `argc` and `argv` originally passed to your program directly into `MLOpen`, allowing the user to specify arguments for the link when they launch your program. `MLOpen` will ignore command line arguments that do not make sense to it, so you don't have to worry about interference from arguments that you want your own `main` function to use. A typical Unix command line to launch a program that will itself launch a kernel might look like this:

```
myprog -linkname 'math -mathlink' -linkmode launch
```

Note that 'math -mathlink' must be quoted so that it is sent to `MLOpen` as a single argument (the -mathlink is an argument to math, so in effect there is a command line within the command line). Alternatively, you can just hard-code the `argv` array in your program like this:

```
int argc = 4;
char *argv[5] = {"-linkname",
                 "math -mathlink",
                 "-linkmode",
                 "launch",
                 NULL};
```

The advantage of allowing the user to specify the link options on the command line is that they might want to use a linkmode other than 'launch', or perhaps they will need to specify a different name for the kernel program than just 'math'. Of course, you can query the user for link arguments through prompts or a dialog box if you wish, instead of reading the command line.

The above example was typical for Unix. On the Macintosh, the link name will normally look a bit different, perhaps the following:

```
int argc = 4;
char *argv[5] = {"-linkname",
                 "'Hard Disk:Math 2.2:Mathematica Kernel' -mathlink",
                 "-linkmode",
                 "launch",
                 NULL};
```

Notice the very important inner set of single quotes around the pathname. Because Macintosh folder and file names can have spaces in them, it is important that the pathname be enclosed in single quotes so it is seen inside `MLOpen` as a single string, not separate chunks broken up by spaces.

In Windows, don't forget that C treats the '\' character specially in string constants. If you are embedding filenames into your code, make sure you use two consecutive '\' to indicate a directory separator, as in this example:

```
int argc = 4;
char *argv[5] = {"-linkname",
                 "c:\\wnmath\\math -mathlink",
                 "-linkmode",
                 "launch",
                 NULL};
```

■ 2.2.2 MLConnect

If `MLOpen` fails, it will return `NULL`. However, the fact that `MLOpen` returns non-`NULL` does not mean that the link is connected and functioning properly. There are a lot of things that could be wrong. For example, if you launch a program that knows nothing about *MathLink*, the `MLOpen` will still succeed. There is a difference between opening a link (which involves setting up your side) and connecting one (which verifies that the other side is alive and well).

If the link cannot be connected, then the first *MathLink* call you make that tries to read or write something will fail, or worse, hang indefinitely. Rather than put some special-case test on your first reading or writing function (which may be physically quite distant in your code from the `MLOpen` call), you might want to call `MLConnect` after `MLOpen`. `MLConnect` will try to connect the link without actually reading or writing anything, and it's a convenient, self-documenting way of ensuring that the link is functioning properly before proceeding with your program. `MLConnect` takes a link object as its argument, and returns non-zero to indicate a successful connection.

It's important to note that the `MLConnect` function will block until the connection succeeds or until it detects a fatal problem with the link. Thus, your program will hang during the startup time of the kernel (if you call `MLConnect` immediately after `MLOpen`). A more serious problem is that if the user mistakenly launches a program that is not *MathLink*-aware, `MLConnect` will block indefinitely. Dealing with blocking in *MathLink* functions is discussed more thoroughly later, but for now note that there are two strategies: installing a yield function or polling `MLReady`. The use of `MLReady` deserves special comment in the present context. Before the link is connected, `MLReady` has a special meaning: it tells whether the other side is ready to participate in a connection. In other words, it tells whether `MLConnect` will block or not. Thus, before you call `MLConnect`, you can repeatedly call `MLReady`, waiting for it to return `TRUE`, and perhaps bail out of the attempt to connect after some period elapses.

■ 2.2.3 Using Listen and Connect LinkModes

You do not have to launch the kernel in your program. If it is already running, users can establish a connection to your program manually. This is done using the Listen and Connect link modes. One side must open a link in Listen mode, and the other opens a link with Connect mode, specifying the listening link to connect to. For example, your program can open a listening link, announcing to the user what "name" is being broadcast (or letting the user pick a name), and then the user can manually connect to that link from *Mathematica*. For example, if you opened a link on a Macintosh with this `argv` array:

```
char *argv[5] = {"-linkname",
                 "myLink",
                 "-linkmode",
                 "listen",
                 NULL};
```

then the command in *Mathematica* to connect to that link would be the following:

```
LinkOpen["myLink", LinkMode->Connect]
```

At this point, the connection will be established so that expressions can be read and written on each end, but the kernel is still functioning in its normal interactive mode; it has not become a "slave" to your program (it is not yet in "mathlink mode"). To point the kernel's attention toward your program, you need to set the kernel's **`$ParentLink`** variable to be the link to your program:

```
$ParentLink = %;

(* or, just do it in one line:
       $ParentLink = LinkOpen["myLink", LinkMode->Connect]
*)
```

When you use the Launch linkmode, all this is taken care of for you.

As an experiment, some time when you are using the front end, type **`$ParentLink = Null`**. This will "unattach" the master/slave relationship between front end and kernel. On Macintosh or Windows, you will see the kernel's terminal-interface window appear in the background. Switch to it, and you will see that you can now interact with it as if you had launched it by itself, instead of from the front end. In the kernel window, type **`$ParentLink = First[Links[]]`** (this will point **`$ParentLink`** back at the link to the front end, which is still open). Switch back to the front end, and you should be able to continue with your session.

■ 2.2.4 MLInitialize and MLDeinitialize

In the sample program above there is a call to `MLInitialize` before the link is opened. Starting with Version 2.2.2, all correct *MathLink* programs must call `MLInitialize` before making any *MathLink* calls, and `MLDeinitialize` after closing all opened links. `MLInitialize` and `MLDeinitialize` have never been documented before Version 2.2.2, so it is likely that all existing *MathLink* programs do not call them. Does this mean that every existing *MathLink* program is suddenly broken when built with version 2.2.2 of the *MathLink* libraries? Technically yes, but in practice there will rarely be a problem. If you are the author of a *MathLink* program that you distribute in source code form, you should update the code to call these two functions. If you are currently writing a program, make sure it calls them (this change is backward compatible with older versions of *MathLink*).

Note that this is not a concern when writing "installable" functions (treated in the first chapter of these notes). With installable functions, the mprep tool writes most of the *MathLink* code, including calls to `MLInitialize` and `MLDeinitialize`.

Here is how to use them. Declare a variable of type `MLEnvironment` and assign it the return value from `MLInitialize`. Then pass this variable to `MLDeinitialize` before your program exits.

```
MLEnvironment  env;

...

env = MLInitialize(NULL);
if(env == NULL) clean_up_and_exit();
link = MLOpen(....);

...

MLClose(link);
MLDeinitialize(env);
return;
```

2.3 Sending Expressions to the Kernel

The things you send with *MathLink* are *Mathematica expressions*, not just strings or numbers or some other limited type. Since everything in *Mathematica* is an expression, you have its full power and expressiveness at your disposal. There are two classes of expressions in *Mathematica*: "atomic" expressions, which have no subparts (these are strings, symbols, and numbers), and "composite" expressions, which have a head and zero or more arguments. Composite expressions are things you would write with square brackets, such as **`f[]`**, **`h[x, y]`**, **`{1,2,3}`** (which is a shorthand for **`List[1,2,3]`**), **`2+2`** (which is a shorthand for **`Plus[2,2]`**), and **`Integrate[x^2, {x,0,1}]`**. There are `MLPut` functions for the necessary atomic types (`MLPutString`, `MLPutSymbol`, `MLPutInteger`, etc.), and for composite expressions you use `MLPutFunction`. You send expressions using these `MLPut` calls in a way that mirrors their `FullForm` representation in *Mathematica*. Thus, to send the expression **`{1.23, f[x], {5, "a string"}}`**, you would say:

```
MLPutFunction(link, "List", 3);
  MLPutReal(link, 1.23);
  MLPutFunction(link, "f", 1);
    MLPutSymbol(link, "x");
  MLPutFunction(link, "List", 2);
    MLPutInteger(link, 5);
    MLPutString(link, "a string");
```

If you aren't sure what sequence of calls is required for some expression, just launch *Mathematica*, type in **`FullForm[Hold[expression]]`**, and the output can be translated directly into the appropriate calls. Ignore the **`Hold`** that will be wrapped around the output--it is included merely to prevent the expression from evaluating (you want to see the **`FullForm`** of the original expression, not of what it evaluates to).

I mentioned earlier that when the kernel is in "mathlink mode", it sends all results in the form of packets, and expects all input in packets. (The use of the term "packet" here should not be confused with the concept of packets that might exist in some low-level communication protocol; TCP/IP packets, for example. The *MathLink* programmer need have no concern over such low-level issues.) *MathLink* packets are simply functions, "heads" that serve to convey to the receiving side of the link some information about what to do with the contents. When the kernel sends back the result of computing **`2+2`**, it sends back the answer wrapped in the do-nothing function **`ReturnPacket`**:

```
ReturnPacket[4]
```

The **`ReturnPacket`** wrapper tells your program that the content is the result of an evaluation.

■ 2.3.1 Packets for Sending Things to *Mathematica*

Everything that you send to the kernel should be wrapped in a packet head. There are three packet types for sending things to the kernel.

□ The *Mathematica* "main loop"

To appreciate the difference between the various packets, you need to understand the concept of *Mathematica*'s "main loop". When you use *Mathematica* in the usual way, each input string you type is fed through a main loop that begins with parsing the string into an expression, evaluating the expression, and finally turning the resulting expression back into a string for printing to the screen. An accounting of the steps in the main loop is given in Appendix A.7.3 of *The Mathematica Book*. The steps include application of the **`$PreRead`**, **`$Pre`**, **`$Post`** and **`$PrePrint`** functions, and most importantly, assigning the **`In`** and **`Out`** values.

The main loop is designed to implement the notion of an interactive "session", with a history of inputs and outputs recorded in the **`In`** and **`Out`** values. For your use of the kernel, such a notion may be superfluous. If you are just using it for computational services, you may have no reason to want a running history of previous inputs and outputs. In this case, you want to circumvent all the steps in the main loop except the actual evaluation of the expression. On the other hand, if you are creating your own front end that a user interacts with, you mght want to display the **`In`** and **`Out`** prompts, or at least provide a way to recall previous input and output.

The three packet types differ in the form of their contents, the form of the results returned the kernel, the number and type of packets you will get back, and whether the main loop will be run.

As an example, if the main loop is run, you will always get an **InputNamePacket,** and perhaps also an **OutputNamePacket**, with each evaluation.

☐ EvaluatePacket

The contents of an **EvaluatePacket** are an arbitrary *Mathematica* expression, which will be evaluated and the result sent back to you as an expression wrapped in **ReturnPacket**. The main loop is circumvented.

Here is how you would send **3+3**:

```
MLPutFunction(lp, "EvaluatePacket", 1);
  MLPutFunction(lp, "Plus", 2);
    MLPutInteger(lp, 3);
    MLPutInteger(lp, 3);
MLEndPacket(lp);
```

You may have seen or written *MathLink* code that did not explicitly use a packet head for sending things. In the past, if you left off a packet head, **EvaluatePacket** was assumed. Be aware that this behavior is no longer supported; always use a packet head and an explicit call to `MLEndPacket`. Note that there is no "`MLPutPacket`" for sending the packet head--since packets are just functions, `MLPutFunction` is used.

Building up complicated expressions with a series of these calls is straightforward, but it can be very tedious. Another way to send something to *Mathematica* is as an input string wrapped in the **ToExpression** function:

```
MLPutFunction(lp, "EvaluatePacket", 1);
  MLPutFunction(lp, "ToExpression", 1);
    MLPutString(lp, "3 + 3");
MLEndPacket(lp);
```

For sending **3+3** this isn't any easier, but for something like **Plot3D[Sin[x] Cos[y], {x,0,2Pi}, {y,0,2Pi}]** it saves a lot of code. You should use this method whenever it is more convenient to send code as a string (for example if you know it at compile time, or if you are reading the input from a file or keyboard).

Keep in mind that what you are sending to the kernel is *Mathematica* code in all its generality. Anything that's possible to type into a *Mathematica* session can be sent via *MathLink*. It may be a bit clumsier to create expressions with sequences of `MLPut` calls, but keep separate in your thinking the code you want the kernel to execute and the details of "assembling" that code in your C program.

Say you want not only to send the code as a string, but also to receive the result as a formatted string, exactly as it is displayed in a normal interactive *Mathematica* session. You would do this if you wanted to display the result to the user (i.e., with all the complicated line-breaking logic for having multi-line expressions formatted properly). You need merely ask yourself how you would write *Mathematica* code that would take a string, turn it into an expression, evaluate it, and then turn the result back to a string. That code is simply

```
ToString[ToExpression["the string"]]
```

The series of functions to assemble this expression and send it from your C program follows directly:

```
MLPutFunction(lp, "EvaluatePacket", 1);
  MLPutFunction(lp, "ToString", 1);
    MLPutFunction(lp, "ToExpression", 1);
      MLPutString(lp, my_string);
MLEndPacket(lp);
```

□ EnterTextPacket

The contents of an **EnterTextPacket** must be a string, which will be sent through the entire main loop, beginning with parsing as *Mathematica* input. The result of the evaluation will be sent back to you as a formatted string wrapped in a **ReturnTextPacket**. Since the main loop is run, you will also get an **InputNamePacket**, and possibly an **OutputNamePacket**. You will not get an **OutputNamePacket** if the calculation returns **Null** (because *Mathematica* doesn't give output prompts for **Null** return values). Note that the last thing you will get is the **InputNamePacket**, because it is the prompt for the next input, not the one you just sent. In other words, the signal that *Mathematica* is finished dealing with your last input is the arrival of an **InputNamePacket**. **EnterTextPacket** is not discussed in the *MathLink Reference Guide*, but causes the same behavior as using the **Enter** function, and this function is documented. That is, the following two fragments are equivalent:

```
MLPutFunction(lp, "EnterTextPacket", 1);
  MLPutString(lp, "2 + 2");
MLEndPacket(lp);

/* OBSOLETE... */
MLPutFunction(lp, "Enter", 1);
  MLPutString(lp, "2 + 2");
MLEndPacket(lp);
```

The use of **Enter** is now obsolete. Always use **EnterTextPacket** instead.

EnterTextPacket is what Wolfram Research's own front ends use for sending user input (which is nothing more than a string of characters when typed in) to the kernel. In future versions of *Mathematica* this may change, but the point is that if you want to implement an interface that is similar to what the standard front ends present (accept user input as a string and print out formatted output as a string, with prompts), you can send input to the kernel as a string wrapped in **EnterTextPacket**. It is possible to implement a primitive interface that looks very much like the kernel-only "terminal interface" with just a small number of lines of *MathLink* code by using **EnterTextPacket**, which is exactly what is done in frontend.c example program, available on *MathSource*.

□ EnterExpressionPacket

EnterExpressionPacket is like **EnterTextPacket** in that the main loop is run, except that the contents of an **EnterExpressionPacket** must be an expression, not a string to be parsed as code. Furthermore, the result is sent back to you as an expression wrapped in a **ReturnExpressionPacket**.

```
MLPutFunction(lp, "EnterExpressionPacket", 1);
  MLPutFunction(lp, "Plus", 2);
  MLPutInteger(lp, 2);
  MLPutInteger(lp, 2);
MLEndPacket(lp);
```

Actually, when you use **`EnterExpressionPacket`**, only a subset of the main loop is run. There are some steps at the beginning of the main loop that occur before the input string is parsed into an expression (application of the **`$PreRead`** function is an example). With **`EnterExpressionPacket`**, you in effect bypass the parsing step because what you send is already an expression. Similarly, there is a step at the end of the main loop that converts the result expression to a string for display on the screen (the application of the **`$PrePrint`** function). This step never occurs with **`EnterExpressionPacket`**, since what is sent back is the result still in the form of an expression.

■ Summary of Packet Types for Sending to the Kernel

There are only three packet types: **`EvaluatePacket`**, **`EnterTextPacket`**, and **`EnterExpressionPacket`**. These packet types differ in whether their contents are to be an expression or a string, whether their results are to be returned as an expression or a string, and whether they implement the kernel's so-called "main loop".

The one you choose will depend on the answers to the following questions:

Do you want the main input/output loop to be run?

Ask yourself whether you or your users will ever need to refer directly to previous input or output. If the answer is no, then use **`EvaluatePacket`** for sending things to the kernel, which bypasses the main loop. If the answer is yes, then use **`EnterTextPacket`** or **`EnterExpressionPacket`** (the "Enter" in their names conveys the property of running the main loop).

Do you want to send input as a string or as an expression?

As mentioned earlier, if you are letting users type input for the kernel, or if you know at compile time some code you want to send, it is easiest to send the code as a string. If you will be using an **`EvaluatePacket`** (based on your answer to the previous question), you can send a string as follows:

```
MLPutFunction(lp, "EvaluatePacket", 1);
  MLPutFunction(lp, "ToExpression", 1);
    MLPutString(lp, "some string of Mathematica code");
MLEndPacket(lp);
```

If you have decided that you want the main loop, you will be using **`EnterTextPacket`** or **`EnterExpressionPacket`**, and the choice between these is whether you want to send a string or an expression.

Do you want to receive output as a string or as an expression?

If you are using **`EvaluatePacket`** and want to get the result back as a string, use this:

```
MLPutFunction(lp, "EvaluatePacket", 1);
  MLPutFunction(lp, "ToString", 1);
    MLPutFunction(lp, "ToExpression", 1);
      MLPutString(lp, "some string of Mathematica code");
MLEndPacket(lp);
```

If you use **EnterTextPacket** the result will always be a string, and with **EnterExpressionPacket**, it will be an expression, although you can force the result expression into a string in the same way as with **EvaluatePacket**, by wrapping the input expression with **ToString** when you send it:

```
MLPutFunction(lp, "EnterExpressionPacket", 1);
  MLPutFunction(lp, "ToString", 1);
    MLPutFunction(lp, "Plus", 2);
      MLPutInteger(lp, 3);
      MLPutInteger(lp, 4);
MLEndPacket(lp);
```

2.4 Receiving Expressions from the Kernel

■ 2.4.1 Reading Packets: MLNextPacket and MLNewPacket

Everything the kernel sends to you will be wrapped in a packet. As mentioned earlier, packets are just functions from the kernel's perspective. Their purpose is to convey to you information about what is inside, whether you might be interested in it, and how to read it. You can read packets with `MLGetFunction` if you want to, but this would be inconvenient. Instead, *MathLink* has two special functions for manipulating incoming packets: `MLNextPacket` and `MLNewPacket`. Their names are confusingly similar, but their actions are quite different. Think of packets as boxes, and the kernel as sending you a stream of boxes on a conveyor belt. `MLNextPacket` opens a box, whereas `MLNewPacket` discards an already-opened box. Once you have opened a box with `MLNextPacket`, you must either read out its entire contents, or abandon it with `MLNewPacket`. It is an error to call `MLNextPacket` at any time other than when you are "between" boxes, either because you have completely emptied the last one or because you threw it away with `MLNewPacket`. `MLNewPacket` has only one effect--to abandon already-opened boxes. If you call it between boxes, it does nothing. That is important to remember because it means it is always safe to call `MLNewPacket` more than once. If you encounter a condition where you know you are not interested in any data that might be left in a packet, or you want to make sure that you are currently between packets, you can call `MLNewPacket` without worrying if it has already been called.

`MLNextPacket` returns one of a set of predefined integer constants to indicate the type of packet that was opened. These constants are defined in mathlink.h, and have names like `RETURNPKT`, `RETURNTEXTPKT`, `INPUTNAMEPKT`, etc. If you are implementing a sophisticated "front end" for the kernel, you will typically have a `switch` statement in your main loop that tests the value of `MLNextPacket` and branches to code appropriate for reading the contents of the various packet types. You will need to do this if you are allowing the user to enter arbitrary *Mathematica* code, so you need to be prepared to receive virtually any kind of packet type. For example, the user might execute the **InputString** function, which prompts for an input from the user. If this happens, your program will receive an **InputStringPacket** from the kernel, which is a signal that you need to display a dialog box or other prompt to get input and then send back a **TextPacket** with the reply. The **InputString** function is rarely used, of course, and most of the time you'd only be sending and receiving a handful of common packets, but if you are going to give the user a way to enter *Mathematica* code you need to be prepared for anything. An excellent reference for handling the entire set of packets is the frontend.c sample program, which can be found on *MathSource*.

For many uses, though, you are only interested in some limited interaction with the kernel (like using it strictly for computational services), where you know the types of things you will be sending, or at least where you know the types of results you are interested in (*e.g.*, you don't have to worry about displaying kernel messages, or the results of **Print** statements, or graphics). In these cases, you will simply be discarding most packets. Typically, you will only be interested in one packet type, the result of a computation, which will usually be a **ReturnPacket** (depending on how you sent the computation in the first place, as discussed in Section 2.3). This is what was done in the example program in Section 2.1. In this case, you want to implement the logic, "for every packet that is not a **ReturnPacket**, throw it away". That is coded as follows:

```
while (MLNextPacket(link) != RETURNPKT)
    MLNewPacket(link);
read_contents_of_ReturnPacket();
```

That little piece of code is responsible for more *MathLink* programming headaches than anything else! That's because it appears throughout the sample programs, so people copy it verbatim into their own programs without really being aware of its consequences. Specifically, they use it in situations where a **ReturnPacket** will not be coming. What happens then? Well, you drain off all packets waiting for a **ReturnPacket**, then call `MLNextPacket`, which will block forever, so your program hangs.

Therefore, it is extremely important to be sure that you actually will be getting a **ReturnPacket** before you use this code fragment! Typically, people encounter this error for one of two reasons: they get out of sync with the incoming packets, so that they have already discarded the **ReturnPacket** by the time the loop is entered; or, they send computations to the kernel wrapped in **EnterTextPacket** (or, equivalently, they wrap it in the function **Enter**), which causes results to come back in a **ReturnTextPacket**, not a **ReturnPacket**. The latter issue is discussed in Section 2.3; for now I'll assume you are sending an **EvaluatePacket** (if you will be getting a **ReturnTextPacket**, just substitute that for **ReturnPacket** in this discussion).

If your program is hanging unexpectedly, it is almost a certain bet that it is hanging in `MLNextPacket` because you are looping waiting for a packet type that will never arrive.

Another potential problem is that you have done something to cause a *MathLink* error before the loop is entered. In that case, `MLNextPacket` (like most *MathLink* functions) will return 0 until the error condition is cleared with `MLClearError`. A simple way to avoid this problem is to use the following packet-reading loop instead:

```
while ((pkt = MLNextPacket(link)) && pkt != RETURNPKT)
    MLNewPacket(link);
if(!pkt) {
    handle_error(link); /* including calling MLClearError(link) */
} else {
    read_contents_of_ReturnPacket();
}
```

Finally, it is often the case that you don't even care what is in the **ReturnPacket**. For example, if you send a definition to *Mathematica* like **f[x_] := x^2**, then you will get back a

`ReturnPacket` that contains the symbol **`Null`** (many users are not aware that an expression of this type returns something, because there is no **`Out`** line printed). Another example is if you read in a package with **`Get`**--the return value of **`Get`** is the symbol **`Null`**. If you don't want to read the **`ReturnPacket`**, throw it away with a final call to `MLNewPacket`:

```
MLPutFunction(link, "Get", 1);
MLPutString(link, "Statistics`NonlinearFit`");

while ((pkt = MLNextPacket(link)) && pkt != RETURNPKT)
    MLNewPacket(link);
MLNewPacket(link);  /* abandon the RETURNPKT */
```

The second most common *MathLink* error is to forget to drain off the **`ReturnPacket`**s from code that you send, especially "initialization" code you send before your real work begins. If you forget to run the "throw away everything up to and including the next **`ReturnPacket`**" loop, then later when you read a **`ReturnPacket`** that you expect to have the result of the first "real" computation, you will instead be getting a **`ReturnPacket`** from something sent earlier. Remember that everything you send will cause a **`ReturnPacket`** to be sent back, even if just contains the symbol **`Null`**.

■ 2.4.2 Packets *Mathematica* Might Send to You

There are quite a few packet types that *Mathematica* might send to your program. Generally, if you are not allowing a user to directly interact with *Mathematica* (so you as the programmer have control over what gets sent), you don't need to worry about many of them. You either won't get them, or you can just discard them because you are not interested in their contents. For some of them (*e.g.*, **`InputNamePacket`** and **`OutputNamePacket`**), whether or not you get them depends on how you send the input to *Mathematica* in the first place (see Section 2.3). Here is a brief discussion of the more important types. Again, I refer you to the frontend.c program for an example of how to handle every one of the packet types you might get from *Mathematica*.

□ InputNamePacket, OutputNamePacket

You will get these any time you send something wrapped in **`EnterTextPacket`** or **`EnterExpressionPacket`** (you may not always get an **`OutputNamePacket`**, as discussed earlier). Their contents are strings, something like **`"In[1]:="`** or **`"Out[12]="`**, which you can print directly to the screen if you want to show the prompts. If you don't want to show these to the user, you probably should use **`EvaluatePacket`** to send the original input, as explained in Section 2.3, so they won't even be generated.

□ ReturnPacket, ReturnTextPacket, ReturnExpressionPacket

These packets will contain the result of an evaluation. A **`ReturnTextPacket`** will contain a formatted string, ready to be printed directly to the screen (that is, it will have the appropriate line-breaking and padding so that exponents and fractions will be lined up). A **`ReturnTextPacket`** results from sending an **`EnterTextPacket`**. In contrast, a **`ReturnPacket`** or a **`ReturnExpressionPacket`** will contain an expression, which you will have to read with an appropriate series of `MLGet` calls. This is discussed in more detail below.

◻ MessagePacket

This packet signals the beginning of a warning or error message generated by the kernel. An example is the following:

```
Part::partd: Part specification x[[1]] is longer than depth of object.
```

A **`MessagePacket`** will contain two things: first, a symbol (to be read with `MLGetSymbol`) that is the name of the function (**`Part`** in the above example), then a string that is the "tag" of the message ("partd" above). The **`MessagePacket`** will be followed by a **`TextPacket`** that contains the text of the message.

◻ TextPacket

This is used for the output of a **`Print`** statement, the output of **`?Function`** or **`Information[Function]`**, and also the text of a message (see discussion of **`MessagePacket`**). The contents will always be a string (to be read with `MLGetString`, of course).

◻ DisplayPacket

This will hold PostScript code for a graphic, in the form of a string. There may be a series of these, each containing a piece of the total PostScript. A **`DisplayEndPacket`** will signal the last piece of PostScript. How to handle graphics is discussed elsewhere in this tutorial.

■ 2.4.3 Reading the Contents of a Packet

Once you have "opened" a packet with `MLNextPacket` and you have decided not to discard it, you need to read out the contents. This will require some appropriate sequence of `MLGet` calls. In many cases, the contents of the packet are something simple, like a string, which can be read with `MLGetString`. In the case of a **`ReturnPacket`**, though, the contents are an expression, and you may need to implement some expression-reading logic of your own. An example of this is the function `read_and_print_expression` from the factorinteger.c example program that is included with *MathLink*. The basic idea is to recursively descend into the expression, calling `MLGetType` for each new element to find out what `MLGet` call you will need to read it properly.

If you find yourself embarking on such a project, ask yourself if it is really necessary for you to receive the results as an arbitrary expression. If you are expecting some simple type of expression, like a number, that is meaningful in a C program, then fine. But for many applications, there isn't really a whole lot to do with the dismembered expression pieces you're going to get from the process. Often, what people really want is just a string form of the result, because they are only going to display it on the screen. If that's the case, send the computation in such a way that you will get back a formatted string as a result (see Section 2.3).

■ 2.4.4 The "Disown" Functions

MathLink has several functions with "`Disown`" in their name, for example `MLDisownString` and `MLDisownIntegerList`. Whenever you use `MLGet` to receive an object whose size cannot be known at compile time (*e.g.*, a string, symbol, list, or array), *MathLink* reads the object into its own memory space and gives you only the address of the data. For example, in `MLGetString`, you pass the address of a `char*` (i.e., a `char**`), and *MathLink* stuffs the address of the string it received into your `char*`. You don't have to allocate any memory yourself or worry about how big the data is. At this point, *MathLink* "owns" the data, and it is waiting for your permission to free the memory that it occupies, which you grant when you call the `MLDisown` functions. Between the time you call `MLGet` and `MLDisown`, you can only read the data--do not try to modify it in place. If you need to do that, allocate your own memory and copy the data into it (*e.g.*, using `strcpy`).

2.5 Blocking, Yield Functions, and All That

When you issue a `Get`-type call (including `MLNextPacket`), *MathLink* will block if there is nothing waiting to be read from the link. This will cause a problem if you need to do something (like service your user interface) without interruption. There are three solutions for handling this problem.

■ 2.5.1 MLReady and MLFlush

`MLReady` returns 0 to indicate that there is no data on the link waiting to be read, and 1 to indicate there is. In other words, it tells you whether a `Get`-type call will block. You can use it to check that there is data waiting before you call a reading function.

```
MLFlush(lp);
if(MLReady(lp)) {
     MLNextPacket(lp);
     ...handle the packet...
}
```

Note the call to `MLFlush` before `MLReady`. *MathLink* is buffered, meaning that if you call a `Put` function, the data is not necessarily sent right away, but might be held in a buffer instead. Any time you need to ensure that the data is sent immediately, you can call `MLFlush`. An example when you might need to do this is if you are sending something to the kernel that will trigger a side effect, like writing something to a file that your program will read right away.

In the normal flow of writing and reading the link, calls to `MLFlush` are generally unnecessary, since *MathLink* automatically flushes the link at appropriate times. Specifically, if you issue a `Get`-type call and there is nothing there, your side of the link will be flushed. `MLReady`, however, does not flush the link, so it can lie to you in that it is possible for `MLReady` to return 0, yet a `Get` call would not block. This will happen if your side has some data that when sent will trigger the other side to reply--the `Get` flushes the link and receives the reply right away. For this reason, you should always call `MLFlush` before `MLReady`.

■ 2.5.2 Let It Block

When *MathLink* is blocking, it calls what is known as a yield function. A yield function must have two features: (1) it allows other processes to get processor time (on operating systems like Macintosh and Windows that do not have preemptive multitasking); (2) it returns 0 or 1 to indicate whether *MathLink* should continue blocking or bail out of the read call.

There is a default yield function inside *MathLink*, the details of which differ from platform to platform. For example, on the Macintosh, it calls `WaitNextEvent` to allow other processes to get time. If it didn't yield to other processes, then the sending side would never get a chance to send anything and your read call would block forever!

You can install your own yield function if you wish, and this provides a solution to the blocking problem. You simply call back to your main event loop from inside your yield function. When the `Get` call returns, you process the result, send something else if you want, and then immediately issue another `Get` call. Your program can spend most of its life blocking inside a *MathLink* function, calling the main event loop to run the user interface or other periodic tasks. Here is a trivial skeleton of what such a yield function might look like. Note that the second argument to a yield function is of type `MLYieldParameters`. This is a reserved argument that is currently used only in *MathLink*'s own default yield function. You should simply ignore this argument.

```
int yield_function(MLINK link, MLYieldParameters yp) {
    one_pass_main_event_loop();
    return 0;  /* keep on blocking */
}
```

The function you use to install your own yield function is `MLSetYieldFunction`. The interface to this function has changed a bit to support some new *MathLink* platforms (Windows and Power Macintosh). To see how to call it, I recommend that you look at a .tm.c file generated by mprep (see the first chapter for details). The code generated by mprep installs its own yield function that works in just the way I've described above (template programs spend their lives blocking inside `MLNextPacket`, waiting for the kernel to call them). This will show you the proper way to call `MLSetYieldFunction` for your platform and version of *MathLink*.

Having said this, note that in Versions 2.2.x and earlier of *MathLink* on Unix, there is a bug that prevents *MathLink* from calling your yield function when it is blocking in an `MLGet`-type call, unless your program receives a Unix signal. Thus, you will need to set some sort of timer to periodically send yourself a signal, `SIGALRM` for example.

One final note: *MathLink* is not fully reentrant, in the sense that you cannot issue a call on a link while another call on that link is in progress (you *can* read or write to different links). Therefore, if you allow your user interface to run while inside the yield function, you must prevent users from doing anything that would trigger a call on the same link. Thus, before calling the main event loop, you might need to disable some menu choices or other features. You can then re-enable them before returning from the yield function.

■ 2.5.3 Write a Multithreaded Program

If your operating system and development environment allow you to write multithreaded programs, this is an ideal solution to the blocking problem. Most operating systems support multithreaded programming, including Macintosh, Windows NT, the upcoming Windows 4.0, and many flavors of Unix. Simply fork a thread in which the read occurs and let it block. You carry on other processing in other threads.

2.6 Graphics

When many people think of *Mathematica* graphics, they think of the PostScript code that is rendered into the image they see. It is important to remember that a *Mathematica* graphic is an expression, like everything else in *Mathematica*. It might look something like:

```
Graphics[{Line[{{0,0}, {1,1}}], Point[{.5,.5}], .... ]
```

The PostScript code is generated as a side-effect of the **`Display`** function, and is not an inherent part of the graphics object.

If you send a command that produces a graphic, say for example:

```
MLPutFunction(lp, "ToExpression", 1);
  MLPutString(lp, "Plot[x^2, {x,0,1}]");
```

You will get back a `ReturnPacket` containing the **`Graphics`** object, and also a series of `DisplayPackets` containing PostScript (the last one of which is a `DisplayEndPacket`). If you want to display the graphic you have two choices: either render the PostScript somehow, or convert the **`Graphics`** object into a form that you can render. This may seem obvious, but many programmers forget that handling the PostScript is not their only option.

It is likely that in the future the PostScipt interpreter that is built into the notebook front end will be spun off as a separate *MathLink* program that is callable by programmers. When this happens, it will be easy for *MathLink* programmers to render *Mathematica* PostScript in their own programs. Until then, though, dealing with PostScript is problematic unless your machine or environment supports PostScript rendering.

You might want to consider dealing directly with the *Mathematica* graphics expression instead of the PostScript. If you have a graphics library among your programming tools, you will probably find it is not difficult to convert most *Mathematica* graphics into the native functions of your library.

Here are some tips for handling PostScript on various platforms.

■ 2.6.1 Unix

Under Unix, standalone PostScript interpreters have always been part of the *Mathematica* distribution. You can use them in the same way as they are used by the non-notebook interface. What you will get is a separate window, not a part of your program, managed by the PostScript interpreter (*e.g.*, motifps, olps, etc.) To enable this behavior, read in the appropriate graphics initialization file. On most Unix systems, this will be Motif.m:

```
        MLPutFunction(link, "EvaluatePacket", 1);
        MLPutFunction(link, "Get", 1);
        MLPutString(link, "Motif.m");      /* NeXT.m on a NeXT */
        MLEndPacket(link);

       /* Now, read and discard packets up to, and including, the next
          ReturnPacket, which will be the return value of the "Get" function.
          It will contain the symbol Null, which is of no interest.  */
```

This imitates what happens when the kernel is run from the Unix command line. When you issue commands that trigger graphics, you will not get `DisplayPacket`s containing PostScript; rather, a window displaying the graphic will appear.

■ 2.6.2 Macintosh and Windows

There is currently no supported way to render *Mathematica* PostScript on these platforms. For Windows, see the source for the demo Visual Basic front end, which performs this feat.

It is possible to have your program create a skeleton notebook file and write the PostScript to that file. When the file is opened in the notebook front end, the graphics will be displayed. The frontend.c program demonstrates this.

3. Using Other Languages

3.1 C++

The *MathLink* library can be called directly from C++ exactly as it is called from C. You don't even need to think about it. However, there is a complication when writing installable functions, depending on what version of *MathLink* you have. The issue is that in some versions of *MathLink* the C code that is generated by mprep is K&R-style (for compatibility with older C compilers), not ANSI-style, and so it will not pass through a C++ compiler. Beginning with Version 2.2.3, mprep has the ability to generate .tm.c files that are legal C++ files (you can rename them .tm.cpp if you wish). On Macintosh and Windows, this is the default behavior. On Unix platforms, you need to specify a command-line argument to mprep to have it generate C++-compatible code. This behavior is undocumented and may change in the future, but for now the argument is -prototypes:

```
/math/Bin/MathLink/mprep -prototypes addtwo.tm -o addtwo.tm.c
```

If you have Version 2.2.2 or earlier, you must use a C compiler on the .tm.c file generated by mprep. You can still code your external functions in C++ and pass those source files through your C++ compiler. However, note that your functions are being called from the .tm.c file, and thus from C. C++ provides a mechanism whereby you can inform the compiler that certain functions will be called from C: the `extern "C"` declaration. This tells the C++ compiler not to perform the usual name-mangling.

In summary, if you are not making use of the template mechanism (that is, you will not have a .tm file in your project), you can call *MathLink* from C++ without worrying about language issues. If you are using a .tm file, and have Version 2.2.2 or later of the *MathLink* materials, you also don't need to worry about these issues since you can rename the .tm.c file .tm.cpp and pass it through the C++ compiler. On Unix, you will need to specify the -proto option to mprep to enable this. If you have an earlier version of *MathLink*, you will need to obey these two guidelines:
-- The .tm.c file must be compiled with a C compiler.
-- Every C++ function named in the .tm file (and which will therefore be called from C) needs to be declared `extern "C"`.

3.2 FORTRAN and Others

The *MathLink* library is written in C. To create programs that use *MathLink*, you need to call the functions in this library. For this reason, and others outlined below, *MathLink* is easiest to use from C. You can, however, use *MathLink* in conjunction with FORTRAN or other languages. This section refers specifically to FORTRAN, but much of the information is relevant to other languages as well.

As discussed earlier, there are two broad classes of uses of *MathLink*. The first and most common class of uses is to make external functions, written in some compiled language like C or FORTRAN, available within *Mathematica* as if they were built-in functions. We call such external functions "installable" since they use the **`Install`** mechanism to be made available within *Mathematica*, or alternatively "template-based" since they involve writing a template file. The second class of uses of *MathLink* is to allow your own programs to make use of *Mathematica* as a computational engine. It is your program that users interact with, and the services of *Mathematica* are used in the background. These two uses of *MathLink* present different issues and problems to the FORTRAN programmer, so they will be discussed separately.

■ 3.2.1 Calling External FORTRAN Functions from *Mathematica*

The mprep program writes C code for a very significant amount of the *MathLink*-related portions (perhaps all of it) of an installable program. It is convenient to make use of this template mechanism when you want to call external functions from *Mathematica*, no matter what language they are written in. This requires that you have a C compiler. You may not need to know C in any significant sense, because the C code that you write may only be a few lines, and some of that is "boilerplate" code that is the same for every program and can just be copied out of the *MathLink Reference Guide* or the sample programs supplied with *MathLink*.

You will be creating a C program that needs to call your external FORTRAN function. The exact details of how you prepare your FORTRAN routine to be called from C depends on details of your FORTRAN compiler, and perhaps also your C compiler. The difficulty of doing this depends on the types of parameters you need to pass from C to FORTRAN and back. If you only need to pass integers or real numbers, then it may be very simple. It is more complicated to work with strings and arrays, since their representations differ in the two languages. As a simple example, consider how you would modify the basic addtwo example. Everything about this example remains the same except the actual C code for the `addtwo` function, and the fact that there is now a separately compiled FORTRAN file containing code for the computation. Here's how the FORTRAN code might look:

```
subroutine addtwoF(i, j, k)
integer*4 i,j,k
k = i + j
return
end
```

Here's how the `addtwo` function might look:

```
int addtwo(int i, int j) {

    int result;

    addtwoF(&i, &j, &result);
    return result;
}
```

The `addtwo` C code is just a "wrapper" that prepares things for calling the `addtwoF` function. The & is the "address-of" operator in C, and it is needed because FORTRAN expects arguments to be passed by reference, not by value (as is the case with C). Thus, you need to pass to `addtwoF` not the values of the integers `i` and `j`, but the actual addresses where the values are stored. The FORTRAN code extracts the values from these addresses, adds them, then stuffs the result at the address of the result variable. On some systems, you may need to put an underscore at the end of the `addtwoF` function in the C source, calling it as `addtwoF_(&i, &j, &result);`

It is also possible that your FORTRAN compiler allows you to specify that parameters to a function will be passed by value. If this is the case, then you may be able to completely dispense with the C portion of the `addtwo` function because your FORTRAN code will be written in a way that it can be called exactly as if it were written in C. The call to `addtwo` is made from the code that mprep creates, and of course it writes the `addtwo` function call as if `addtwo` were written in C. If your FORTRAN compiler lets you write FORTRAN that adheres to C's calling conventions, then you may not need to write any C wrappers around your functions. An example is Absoft FORTRAN, which includes the `VALUE` keyword to specify that parameters will be passed by value. Here is what the `addtwo` code might look like in such a dialect of FORTRAN:

```
integer*4 function addtwo(i, j)
integer*4 i,j
VALUE i, j
addtwo = i + j
return
end
```

In this case, there is no `addtwo` written in C.

I want to emphasize that different FORTRAN compilers may use different calling conventions, and thus there are many issues that might need to be resolved. These include what order the arguments are passed in, whether they are on the stack or in registers, whether the return value is on the stack or in a register, etc.

Whichever of these two methods you choose, remember that you need a C compiler, and that the issue is not how to call C from FORTRAN, but rather how to call FORTRAN from C. You should consult your FORTRAN compiler's documentation for information on how to write and compile FORTRAN functions so that they can be called from C.

■ 3.2.2 *InterCall* TM

There is a commercial product (not from Wolfram Research) called *InterCall* that simplifies the process of calling external FORTRAN (or C) functions from within *Mathematica*. This method uses *MathLink* only indirectly. *InterCall* has many capabilities, and I make no attempt to describe them here. This discussion is not an endorsement of *InterCall*. For more information, consult *MathSource*, which has a lot of *InterCall*-related materials. Try sending the following email message to mathsource@wri.com:

find InterCall

You will get a return mailing of abstracts of items on MathSource pertaining to *InterCall*. Probably the most useful item is this one:

0202-587: InterCall(tm) Information Sheet and Abridged Manual (June 1992)
Author: Terry Robb
InterCall completely integrates the symbolic capabilities of
Mathematica with the numeric routines of any external library. You
can pass a Mathematica function, array, or any other expression as
an argument to any external routine and InterCall will send the
correct type of information to that external routine.

0011: Info.txt Plain-text information sheet (5 kilobytes)

0022: InterCall.tex TeX version of Abridged InterCall Manual (53 kilobytes)

0033: InterCall.ps PostScript version of Abridged InterCall Manual (180 kilobytes)

If you want, say, the Info.txt document, send the following message to mathsource@wri.com:

send 0202-587-0011

■ 3.2.3 Calling *Mathematica* from a FORTRAN Program

The other class of uses of *MathLink* is where you write the program that the user interacts with, and use the services of *Mathematica* in the background. This requires more in-depth use of the *MathLink* library, because you will be writing all the code yourself (opening and closing the link, putting and getting all expressions, checking for and handling errors, etc.) If you need to make extensive use of *MathLink* in such a program, it may be easiest to write in C (or at least write the *MathLink*-related portions of your program in C). However, since many FORTRAN compilers allow you to call external functions written in C, it is possible to make *MathLink* calls directly from a FORTRAN program. Unfortunately, there are differences in calling conventions and data representations that need to be overcome.

One approach is to write some "glue" code that acts as a wrapper around the *MathLink* functions and serves to translate back and forth between FORTRAN and C conventions. (That's what was done in the first example above, although it was done in reverse--the glue was so that FORTRAN could be called from C.) I suppose it would be possible to write the glue code in FORTRAN, depending on the capabilities of your FORTRAN compiler, but it would be easiest to write in C. Either way, if you know enough about C to write this glue, then you'll probably want to just do the whole project in C. Fortunately, someone else has already written a basic glue library, and it is available on *MathSource*, although it has not been updated for a while. I have not used it myself, but I have no reason to believe that it doesn't still work. To get it, send a message with the following body to mathsource@wri.com:

send 0205-434

This package provides a C source file that encapsulates a basic set of *MathLink* calls in such a way that they can be called from FORTRAN. You compile the file (with a C compiler, of course) to create an external library that provides a number of functions that you can call from FORTRAN, instead of directly using the functions in the *MathLink* library itself.