MathLink®
Reference Guide

Mathematica® Version 2.2

Second edition
First printing

796.3.9.1993

■ What Is *MathLink*?

MathLink is a *communication protocol* for *Mathematica*; that is, a way of sending data and commands back and forth between *Mathematica* and other programs. *MathLink* is easy to use for anyone who is familiar with *Mathematica* and another programming language. Your version of the *MathLink* library is most easily used from C programs. Future versions will include additional support for other languages.

■ What Can I Use It For?

- To call custom code from *Mathematica*
- To call *Mathematica* as a subprogram from another program
- To call *Mathematica* from *Mathematica*
- To develop an application program that communicates with *Mathematica*

The programs that you link to *Mathematica* can range from very simple routines to perform a particular kind of calculation to sophisticated front ends for *Mathematica* to applications that can act on data generated by *Mathematica* or can use *Mathematica* to perform advanced calculations or to create a graphic representation of the external application's data.

This guide gives you an introduction to *MathLink* along with a handy reference for the *MathLink* functions you can use in your programs. Try the examples to see how *Mathematica* can work hand in hand with an outside program.

Before you read this manual to learn about *MathLink* for the first time, you ought to be familiar with the C language. If C is new to you, or you need to refresh your knowledge of it, refer to a basic C text, such as Kernighan and Ritchie's *The C Programming Language*. You will also need to understand the general structure of *Mathematica* expressions; for this you can refer to Sections 2.1 and A.1 of *Mathematica: A System for Doing Mathematics by Computer*, Second Edition.

■ Where Can I Find More Information?

There is more information on new and experimental features of *MathLink* for Version 2.2 of *Mathematica* in the technical report *Major New Features in Mathematica Version 2.2*. This document is included in all copies of *Mathematica* and all *Mathematica* 2.2 upgrades. You should keep it handy if you want to create advanced *MathLink* applications.

Major New Features in Mathematica Version 2.2 can also be found in the form of a text file in the `Documents` subdirectory or folder in your *Mathematica* distribution.

In the C header file `mathlink.h`, you will find helpful comments along with declarations of the *MathLink* library functions, packet types, data types, error codes, and device information selectors supported for the current version of *MathLink*.

Note that many of the functions declared in `mathlink.h` are for *MathLink*'s internal use and therefore are not documented in this guide or in *Major New Features in Mathematica Version 2.2*.

If you have *Mathematica* 2.2 for Macintosh systems, you will find important information about using *MathLink* in the `README` file on the *MathLink* disk in your *Mathematica* distribution.

Table of Contents

1 Introduction

MathLink is a general mechanism for exchanging mathematical expressions between *Mathematica* and other programs. *Mathematica* supports *MathLink* communication through many built-in functions and a library of *MathLink* routines that can be used in external C language programs (or any program that can call a C function library).

You can use *MathLink* to call an external program and send data and commands to it from inside *Mathematica*, or to call *Mathematica* from an external program. You can also use *MathLink* to connect *Mathematica* to a front end program that handles the user interface, or you can use *MathLink* to exchange expressions between two concurrent *Mathematica* sessions.

One way to use *MathLink* is to take functions defined in an external program and install them into *Mathematica*, where you can use them in expressions just as if they were *Mathematica* functions. It is very easy to use *MathLink* in this way, because the communication details are taken care of by *Mathematica*'s `Install` function and the compiling tools provided with *MathLink*. Chapter 2 tells you how to use these tools to link C programs with *Mathematica*.

For more general applications of *MathLink*, you might want to learn some of the details of how *MathLink* works. You can write external programs that call *MathLink* library functions to initialize *MathLink* connections, to send and receive data, and to close the connections. You can also use corresponding commands for performing these operations from *Mathematica*. This reference guide explains the usage of these functions. Several examples are included to show *MathLink* functions in a working context, and listings of *MathLink* functions are included for easy reference in Chapters 10 and 11.

Chapter 3 is an overview of how data are sent back and forth through *MathLink*. It explains the general sequence of C function calls that put expressions to *MathLink* or get expressions from *MathLink*.

Chapter 4 introduces you to the basic high-level functions you can use in your programs to open links and conduct *MathLink* dialogs with *Mathematica*. A few very simple example programs are included.

Chapters 5 and 6 describe two ways to have *Mathematica* talk over *MathLink*. Chapter 5 explains how *Mathematica* behaves when it is accessed in *MathLink* mode from another program. Chapter 6 talks about the specific *MathLink* operations that you can perform manually from inside *Mathematica*.

MathLink can use various data transport systems: on Unix systems, it can use pipes or TCP, and on Macintosh systems, it can use TCP or PPC (program-to-program communication, a System 7 feature). A process can be linked with another process on the same or a different machine. You can use *MathLink* not only between a main program and a subprogram, but between processes that are running independently as peers. Chapter 7 explains the choices and gives examples that use the different link modes.

Chapter 8 covers the functions for checking *MathLink* error conditions. It also discusses how you can make an external program respond to an interruption request from *Mathematica*. These issues are important in all but the simplest communication tasks.

Chapter 9 rounds out the description of *MathLink* with some routines that represent *MathLink* expression elements as text rather than as various C types. You will be able to use *MathLink* for many purposes without these functions, but you will need them to deal with numbers that cannot be represented as C-type numbers.

Several examples in this guide are based on example programs provided on magnetic media with *Mathematica* or with the *MathLink* Developer's Kit. Try some of these examples as you go through this guide. Also refer to the README files on your distribution media for specific information on running the examples on your system.

Future versions of *MathLink* will provide support for other programming languages in addition to C. However, the current version of the *MathLink* library can be used with other languages, provided that your programs call *MathLink* library functions via an appropriate interface. See the Appendix on page 71 for some discussion of the issues involved in using *MathLink* with a language other than C.

2 Using External Functions from Inside *Mathematica*

Mathematica provides a simple way to use *MathLink* to call functions within an external program and receive results back. When you build an external program and link it up with *Mathematica* as described in this chapter, your external functions become functions defined in *Mathematica*, which you can use in expressions just like any other *Mathematica* function.

■ How to Install Functions from an External Program

To use functions from an external C program in *Mathematica*, you should compile the program using the special tools provided with *Mathematica* or the *MathLink* Developer's Kit, then use `Install["`*program*`"]` inside *Mathematica*. Here is a step-by-step summary of the procedure.

- Write the external function or functions as a C program.
- Write a *MathLink* template file associating each external function with a *Mathematica* pattern.
- Process and compile with `mcc` (Unix versions only) or `mprep`.
- Use `Install["`*name*`"]` in *Mathematica*, specifying the name of your compiled external program.
- Access the functions from *Mathematica* by the names given in the template file.
- When you are done using the installed functions, remove them by using `Uninstall[`*link*`]`, specifying the `LinkObject` expression that was returned by `Install`.

■ An Illustrative Example

The best way to understand what you need to do in each step of an external program installation is to see how it is done for a simple example. This section will illustrate the procedure by referring to an external program for adding two integers.

Writing the external functions

Begin by writing a C program consisting of one or more functions that you want to install, along with the function `main`, which should include the function call `MLMain(`*argc*`, `*argv*`)`. The *argc*/*argv* pair usually should be the argument list values passed to `main` by the operating environment. (The argument list may specify optional parameters for the *MathLink* connection, but in many cases, no arguments need to be supplied.)

Here is the program file for the `addtwo` external function.

```
#include "mathlink.h"

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

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

addtwo.c.

`MLMain` is a function that will be generated for you when you run the *MathLink* compiling tool, `mcc` (provided with Unix versions only), or the template file processor `mprep` (provided with Unix and Macintosh versions). `MLMain` takes care of opening the *MathLink* connection and sending and receiving data.

The argument list passed to `MLMain` may include certain parameters that tell *MathLink* what kind of connection to use. To learn what these connection parameters can be, see page 37. For most purposes, you can let the argument list be empty.

The C source file should include the *MathLink* header file `mathlink.h`.

The template file

A *MathLink* template file specifies the correspondence of function names and argument types between your external functions and their *Mathematica* forms. A template file matches specific *Mathematica* patterns with the external functions and arguments that should be used in evaluating expressions that fit those patterns. The template filename should end with the extension `.tm`.

The set of specifications for a single external function is called a template entry. A template file should have one template entry for each external function being installed.

The following table lists the kinds of specifications that can appear in a template file.

`:Begin:`	beginning of template entry
`:Function:` *f*	source code name of the external function
`:Pattern: f[`*x*`_`*type*`, `*y*`_`*type*`, ...]`	*Mathematica* pattern for which the external function is to be used
`:Arguments: {`*x*`, `*y*`, ...}`	*Mathematica* list of the arguments to be passed to the external function
`:ArgumentTypes: {`$mtype_1$`, `$mtype_2$`, ...}`	list of the data types of the arguments
`:ReturnType:` *mtype*	data type of the return value from the external function
`:End:`	end of template entry
`:Evaluate:` *expr*	*Mathematica* input to be evaluated when the external program is started
`::` *text*	comment

Template file keywords.

Here is a template file identifying the external `addtwo` function with the *Mathematica* object `AddTwo` when it is applied to two integers.

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

`addtwo.tm`.

The `:Begin:` and `:End:` lines delimit the template entry for a single external function. Here is what each line in the template entry means.

`:Function: addtwo` gives the name of the function in the external program.

`:Pattern: AddTwo[i_Integer, j_Integer]` gives the *Mathematica* pattern for the expressions that should be evaluated by a call of the external function. The external function will not be called for expressions that fail to match this pattern completely. For example, with the `addtwo` program installed, *Mathematica* would return the expression `AddTwo[4, Pi]` unchanged, since the symbolic second argument does not fit the pattern.

`:Arguments: {i, j}` lists the *Mathematica* objects that are to be passed as arguments to the external function. In this case, they correspond to blanks in the `:Pattern:` line. Generally, however, the arguments can be any *Mathematica* expressions that evaluate to the appropriate types.

`:ArgumentTypes: {Integer, Integer}` specifies the data types that the external function will expect to see in the call packet that is sent by *Mathematica*. If *Mathematica* sends an argument of the wrong type and it cannot be converted to the correct type, the external function will return `$Failed`. You should be able to avoid this by requiring suitable types in the `:Pattern:` line. The number of argument types must match the number of arguments.

`:ReturnType:` is the type of the result that will be sent back to *Mathematica*. It must correspond to the type returned by the function in your program, or it should be set to `Manual` if the external function explicitly calls *MathLink* library routines to send the return expression to *Mathematica*.

The following table lists the type names that can be used in a template's `:ArgumentTypes:` or `:ReturnType:` line, and shows what kinds of *Mathematica* data and C types they correspond to.

Template type	*Example in Mathematica*	*Corresponding C type(s)*
`Integer`	`37`	`int`
`Real`	`37.19`	`double`
`IntegerList`	`{1, -4, 6}`	`int *` (values) and `long` (list length)
`RealList`	`{-0.4, 2.7, 6.3, 0.}`	`double *` and `long`
`String`	`"hello"`	`char *`
`Symbol`	`x`	`char *`
`Manual`		none

Note: The `IntegerList` and `RealList` types are not allowed as return types; use the `Manual` return type to return lists.

Data types to use as `:ArgumentTypes:` or `:ReturnType:` in template entries.

Note: The template file and the C source file can be combined into one file with the extension `.tm`. The template file processor will turn the template entries into C code and leave your C code unchanged.

Compiling and installing the external functions

To build an installable external program, you should process the template file into a C module that takes care of the *MathLink* details, then compile this module and link it with your C functions and the *MathLink* library. *Mathematica* gives you tools to make this process easy.

A program called `mprep` takes the template file as input and produces a C file that gives the *MathLink* interface between your external functions and *Mathematica*. It is then a simple matter to compile the external program, following the usual procedure for your compiler.

On Unix systems, you can run a compiling script `mcc`, which will call `mprep` for you and then finish the compilation by calling `cc` with appropriate arguments.

To compile the `addtwo` program on a Unix system, you would type

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

to use the C source file `addtwo.c` and the template file `addtwo.tm`. The `-o` indicates that the following argument, `addtwo`, should be used as the name of the executable output file.

On Macintosh systems, there is no `mcc`, but there are versions of `mprep` for the Think C and MPW compilers. You would first process your template file with the appropriate version of `mprep`, then run your compiler on the output source file along with the C source file(s) that contain your function definitions. See the `README` file in your *MathLink* distribution for more information.

Installing and using the external functions in *Mathematica*

Inside *Mathematica*, use `Install["`*name*`"]` to install an external program. The `Install` mechanism opens a link to the external program and creates definitions in your *Mathematica* session for the functions that will be evaluated externally. These definitions use a *Mathematica* function named `ExternalCall` to carry out the communication details.

The output from `Install` is a `LinkObject` link identifier. If you assign this link object to a variable, you can conveniently close the link later. For example, the external functions installed via the link `addlink` can be removed by typing `Uninstall[addlink]`.

Installing the "addtwo" program.

Connect to and install the external program.

```
In[1]:= addlink = Install["addtwo"]

Out[1]= LinkObject[addtwo, 1, 1]

In[2]:= AddTwo[3,4]

Out[2]= 7
```

The function `AddTwo` was defined when the external program was installed.

```
In[3]:= ?AddTwo
Global`AddTwo

AddTwo[i_Integer, j_Integer] :=
  ExternalCall[LinkObject["addtwo", 1, 1],
   CallPacket[0, {i, j}]]
```

Notice that the definition is used for integer arguments only.

```
In[3]:= AddTwo[2.9, 5]

Out[3]= AddTwo[2.9, 5]
```

`Uninstall` removes all definitions for external functions and closes the link.

In[4] := **Uninstall[addlink]**

■ Using Lists as Arguments and Return Values

You might want to pass a list of numbers as an argument to an external function, or receive a list as a result. For list arguments, you can use the types `IntegerList` and `RealList` in your template file's `:ArgumentTypes:`. Lists as return values have to be handled differently, by specifying a `Manual` type and putting explicit *MathLink* library calls into your external function to send the list. The example in this section uses lists both as arguments and as return values. It also introduces the use of `:Evaluate:` specifications to define usage messages for an installed function.

The external program for the following example defines two functions: a function `bitand` that performs a bitwise "and" operation on two integers to produce an integer, and a function `complements` that takes a *list* of integers and returns a list containing the bitwise complement of each integer in the argument list.

```
#include "mathlink.h"

int  bitand(x, y)
    int     x, y;
{
    return(x & y);
}

void complements(px, nx)
    int     px[ ];
    long    nx;
{
    int    i;
    for(i = 0; i < nx; i++)
        px[i] = ~ px[i] ;
    MLPutIntegerList(stdlink, px, nx);
}

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

bitops.c.

Notice that the `complements` function produces a list of integers as the result. A list has to be sent back to *Mathematica* manually, that is, by writing the appropriate *MathLink* call into the program. In this case, the *MathLink* call is `MLPutIntegerList(stdlink, px, nx)` where `px` is the name of the integer array and `nx` is the number of integers to be put. The argument `stdlink` is a global link variable identifying the link to *Mathematica*. Other *MathLink* functions for manually writing various types of data to a link are described in the section "Put and Get Functions" in Chapter 4.

The first element of an array in C has index 0. Note that the same element in *Mathematica* form is referred to as part 1 of the list.

Here is the template file for this external program. It comprises two template entries.

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

:Evaluate: BitAnd::usage = "BitAnd[x, y] gives the bitwise conjunction of
    two integers x and y."

:Begin:
:Function: complements
:Pattern: BitComplements[x_List]
:Arguments: {x}
:ArgumentTypes: {IntegerList}
:ReturnType: Manual
:End:

:Evaluate:
    BitComplements::usage =
        "BitComplements[{x1,x2,...}] generates a list of the
        bitwise complements of the integers xi."
```

`bitops.tm`.

The template file contains two template entries. One entry associates the external function `bitand` with the *Mathematica* function `BitAnd`, and the other associates the external function `complements` with the *Mathematica* function `BitComplements`. Along with the template entries, there are also two `:Evaluate:` specifications. An `:Evaluate:` specification calls for the given *Mathematica* input to be evaluated when the external program is installed. In this template file, the `:Evaluate:` expressions set usage messages for the functions that are being installed.

An `:Evaluate:` specification can extend over several lines, as long as all lines after the first are indented and no blank lines intervene.

`:Evaluate:` expressions are evaluated in the order they are given. Those that come before a template entry are evaluated before the external function definition is made, and those that come after a template entry are evaluated after the function is defined.

Notice that `BitComplements` is given a return type of `Manual`, because the external function `complements` uses an explicit *MathLink* call to return the result. `Manual` tells the *MathLink* template file processor not to try to send the C function's return value (which in this case is void) back to *Mathematica*.

After compiling and installing the `bitops` program, you will be able to use two new functions in *Mathematica*: `BitAnd` and `BitComplements`.

Installing the "bitops" program.

Install the program.

```
In[1]:= bitlink = Install["bitops"];
```

The operator `^^` is used to enter numbers in a nondecimal base.

```
In[2]:= BitAnd[2^^1011010111, 2^^1101101011]

Out[2]= 579

In[3]:= BaseForm[%, 2]

Out[3]//BaseForm= 1001000011
                           2
```

On this machine, the bitwise complement of an integer x is equal to $-1 - x$.

```
In[4]:= BitComplements[{0, 1, 2}]

Out[4]//BaseForm= {-1, -2, -3}

In[5]:= LinkPatterns[bitlink]
```

`LinkPatterns` gives a list of the patterns defined by this installed program.

```
Out[5]= {BitAnd[x_Integer, y_Integer],
 BitComplements[x_List]}

In[6]:= ?BitAnd
BitAnd[x, y] gives the bitwise conjunction
of two integers x and y.

In[6]:= ?BitComplements
BitComplements[{x1,x2,...}] generates a list
of the bitwise complements of the integers xi.
```

■ How to Debug an Installable Program

The previous examples in this chapter have used what is called a *parent-child connection*, in which *Mathematica*, as the parent process, starts up an external process and then connects to it via *MathLink*. It is also possible for *Mathematica* to connect to an external process that is already running independently of *Mathematica*. This kind of connection is called a *peer-to-peer connection*. If you want to run your external program inside an interactive debugger, you will likely prefer to use a peer-to-peer connection to link *Mathematica* to the external process.

The general method is to start *Mathematica* and the debugger separately, and to use the appropriate *link modes* to establish a peer-to-peer connection between *Mathematica* and your external program. To do this, first use `LinkOpen["`*name*`", LinkMode->Listen]` in *Mathematica*, where *name* should be an appropriate port name for your data transport protocol. (You may omit *name* to let `LinkOpen` choose a valid port name for you.) Then use your debugger to start your external program, supplying the parameters `-linkmode connect -linkname` *name* as arguments to the external program. The *name* parameter should match the link name returned by `LinkOpen`. *MathLink*'s connection modes are discussed further in Chapter 7.

You may use whatever debugger you are accustomed to in your C programming environment; the following table lists some commonly used debugging tools for various programming environments.

Unix systems	`gdb`, `dbx`
Macintosh MPW	SourceBug, SADE
Macintosh Think C	Think C's integrated debugger

Commonly used debuggers.

The following example illustrates how `gdb` would be used to debug an installed external program on a Unix system. Other debuggers in other environments can be used similarly.

To rebuild the `addtwo` program (see page 4) on a Unix system with debugging information and without deleting the C code generated, use the `-g` option to `mcc`.

Unix session.

```
unix% mcc -g addtwo.c addtwo.tm -o addtwo
```

Now you may run `addtwo` from within the debugger, set breakpoints, examine program variables, and generally observe and affect the execution of your code.

Mathematica session on Unix host "moose".

Create a listening port.

```
In[1]:= addlink = LinkOpen["5000", LinkMode->Listen]

Out[1]= LinkObject[5000@moose, 1, 1]
```

Unix session.

Start a debugger, targeting the `addtwo` program.

```
unix% gdb addtwo
...
```

Set a breakpoint at your C function.

```
(gdb) break addtwo
Breakpoint 1 at 0x4160: file addtwo.c, line 3.
```

Run the program with arguments specifying the link mode and port name.

```
(gdb) run -linkmode connect -linkname 5000
Starting program: /users/shawn/mlsamples/addtwo
```

Mathematica session.

Install by referring to the link object returned by `LinkOpen`.

```
In[2]:= Install[addlink];
```

`AddTwo[3, 4]` causes your C function to be called.

```
In[3]:= AddTwo[3, 4]
```

Unix session.

The breakpoint was reached and the program halted.

```
Bpt 1, addtwo (i=3, j=4) (addtwo.c line 4)
4               return i+j;
```

You may now examine program variables, etc.

```
(gdb) print i+j
$1 = 7
...
```

The debugger command `cont` causes execution to continue.

```
(gdb) cont
Continuing.
```

Mathematica session.

```
Out[3]= 7
```

This example used a TCP port named `5000`; you should use an appropriate port number or name on your system. If you omit the name argument when you open the listening link, `LinkOpen` will generally choose a valid port name for you.

Requests for *Mathematica* Evaluations within an External Function

`MLEvaluate(stdlink, "`*text*`")`	evaluate the specified text as *Mathematica* input from within an external function

Sending requests back to *Mathematica*. The `stdlink` argument specifies the link to *Mathematica* in installable programs that use the standard `MLMain` routine. If you customize a program to use a different link, you may replace `stdlink` with another value.

An external function can send an evaluation request to *Mathematica* while it is in the process of handling a call from *Mathematica*. The function `MLEvaluate` can be used in installable programs to send such a request. `MLEvaluate(stdlink, "`*text*`")` sends *text* as an input string to *Mathematica*. *Mathematica* returns an expression wrapped with the head `ReturnPacket`. This expression should be read by using the "`MLGet`" functions described in Chapter 4.

An alternative way to send an expression to *Mathematica* for evaluation is to use `MLPut` functions to build up and send the expresssion wrapped in an `EvaluatePacket` head. The basic `MLPut` functions are listed in Chapter 4.

Do not confuse `MLEvaluate` with the `:Evaluate:` specification that may appear in a template file. `:Evaluate:` requests in the *MathLink* template file perform evaluations in *Mathematica* at the time the external program is installed, and do not generate `ReturnPacket` results.

Note on Memory Allocation

Note that pointers returned from your C functions or passed to `MLPut` functions are not freed by *MathLink*. So, for example, if one of your functions returns a symbol or string, the pointer returned will not be freed. If you place the return value in static storage or reuse dynamically allocated storage, then this is exactly the behavior you need. If, however, you need to dynamically allocate storage that needs to be reclaimed, you should specify a `Manual` return type, allocate the needed storage, call the necessary `MLPut` functions, deallocate the storage, and return `void`.

Advanced Topic: Using Multiple Instances of an External Program

For some applications, you might want to use more than one instance of an external program at the same time. A textbook example of such an application is a histogram-plotting program that defines functions to add values to a histogram and to plot the histogram. If you want to work with several histograms at the same time, you need to install several instances of the program and to be able to tell the functions which instance you are working with.

In order to make it possible to call a function in a particular installed copy of an external program, include the argument `ThisLink` in the `:Pattern:` line of the external function's template entry. During the installation of an external program, `ThisLink` is bound to the *MathLink* link object created for that installation. To use the installed function, you will give the specific link object as an argument in place of `ThisLink`.

The following example shows how to use this technique to define an external "counter".

```
:Evaluate:      BeginPackage["counter`"]

:Begin:
:Function:      AddToCounter
:Pattern:       AddToCounter[ThisLink, n_Integer]
:Arguments:     {n}
:ArgumentTypes: {Integer}
:ReturnType:    Integer
:End:

:Evaluate:      AddToCounter::usage = "AddToCounter[ck, n] adds n to the counter
                ck and returns the accumulated value."

:Evaluate:        EndPackage[ ]


int counter = 0;

int   AddToCounter(n)
    int n;
{
    counter += n;
    return counter;
}

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

counter.tm.

Note that for this example, the template entry and the external function's C code are combined in a single file. When this file is processed by `mcc` or `mprep`, the template specifications are first converted into C code to set up the external function interface, while the C code that is already there is left as is. The resulting C source file is compiled to create the installable program.

Assuming you have compiled the program and given it the name `counter`, you can install multiple copies in *Mathematica* as illustrated in the following sample session.

Installing two copies of the "counter" program.

Install the first instance of the "counter".

```
In[1]:= cnt1 = Install["counter"];
```

Install another one.

```
In[2]:= cnt2 = Install["counter"];
```

When called with a link object as a first argument, `AddToCounter` sends a message to the corresponding instance of the external program.

```
In[3]:= AddToCounter[cnt1, 11]

Out[3]= 11

In[4]:= AddToCounter[cnt2, 12]

Out[4]= 12

In[5]:= AddToCounter[cnt1, 21]

Out[5]= 32
```

The full information for `AddToCounter` shows that there are two definitions, specific to the two different links.

```
In[6]:= ??AddToCounter
AddToCounter::usage = AddToCounter[ck, n] adds n to
the counter ck and returns the accumulated value.

AddToCounter[LinkObject["counter", 1, 1], n_Integer] :=
  ExternalCall[LinkObject["counter", 1, 1],
   CallPacket[0, {n}]]

AddToCounter[LinkObject["counter", 2, 2], n_Integer] :=
  ExternalCall[LinkObject["counter", 2, 2],
   CallPacket[0, {n}]]
```

Install Command Summary

Here is a summary of the built-in functions that are used in connection with installable programs, as discussed in this chapter.

`Install["`*command*`"]`	open and install an external program that contains functions set up to be called via *MathLink*
`Install[`*link*`]`	install a previously opened link
`Uninstall[`*link*`]`	close the link to an external program and remove the rules for functions defined in it
`LinkPatterns[`*link*`]`	give the list of all patterns to call functions in the external program

Mathematica functions for handling installable programs.

Further Information in This Guide

These examples give some good beginning guidelines for designing external programs that can be called from *Mathematica*. If you want to use *MathLink* in a wider range of applications or want to become familiar with the *MathLink* function calls you can use in an external program, refer to the later chapters of this guide.

Most of the examples in this chapter have used a default connection mode by which the external program was launched as a subprogram from *Mathematica*. It is also possible to use `Install` to connect to an external program that has been started independently. This peer-to-peer method is particularly useful for debugging the external program, as shown on page 10. Chapter 7 has a more general discussion of how different connection modes are used.

3 How Expressions Are Sent over *MathLink*

■ Expressions and Packets

All data sent via *MathLink* are in the form of *Mathematica* expressions. Any *Mathematica* expression can be sent through *MathLink*.

Expressions sent over *MathLink* may convey many different kinds of information. For example, an expression sent from *Mathematica* to an external program might be a result of a calculation, a warning message, or graphics display data.

To make it easy for an external program to know what kind of information it is getting, *Mathematica* can generate output in the form of "packets". A packet is an expression that contains a particular kind of information, wrapped with a head that identifies what type of information is enclosed. *Mathematica* uses packets automatically when you run it in "*MathLink* mode", which is the usual mode for running *Mathematica* as a subprogram from an external program or front end. An external program that receives packet output from *Mathematica* can read the packet type first and then dispatch to a function written to handle that type of packet. *MathLink* mode and packet types are discussed in more detail in Chapter 5.

■ How an External Program Reads or Writes an Expression

To an external program, *Mathematica* expressions sent over *MathLink* are sequences of data elements arranged in a specific order. There are five types of data element, representing integers, real numbers, symbols, strings, and *composite expressions,* respectively.

A composite expression is an expression that is built from one or more subexpressions. A composite expression can be written *head*[arg_1, arg_2, ..., arg_n], with a head expression *head* and zero or more argument expressions arg_i.

Composite expressions can also be referred to as *nonatomic expressions.* Integers, real numbers, symbols, and strings are *atomic expressions,* or *atoms.*

An atomic expression can be read or written with a single *MathLink* function call. A composite expression generally takes several function calls to read or write.

When a composite expression is written to or read from *MathLink*, the number of arguments is established first, so that the receiving program knows how many parts will follow. Then the head of the expression is written or read, followed by each of the arguments in turn.

For example, the expression `b^2 - 4 a c` could be put onto a link by the following sequence of C program statements.

```
MLPutFunction(alink, "Plus", 2);  /* Composite, 2 arguments:          */
                                  /*    Head: Symbol Plus             */
MLPutFunction(alink, "Power", 2); /*    Arg1: Composite, 2 arguments: */
                                  /*       Head: Symbol Power         */
MLPutSymbol(alink, "b");          /*       Arg1: Symbol b             */
MLPutInteger(alink, 2);           /*       Arg2: Integer 2            */

MLPutFunction(alink, "Times", 3); /*    Arg2: Composite, 3 arguments: */
                                  /*       Head: Symbol Times         */
MLPutInteger(alink, -4);          /*       Arg1: Integer -4           */
MLPutSymbol(alink, "a");          /*       Arg2: Symbol a             */
MLPutSymbol(alink, "c");          /*       Arg3: Symbol c             */
```

A sequence of C statements for sending the expression `b^2 - 4 a c`.

As this example shows, the parts of a composite expression can themselves be composite expressions. Each part is completely described, including all of its subexpressions, before the next part starts.

Complex numbers of the form *a* `+` *b* `I` and rational numbers of the form *n*`/`*d* are represented in *MathLink* as `Complex[`*a*`,` *b*`]` and `Rational[`*n*`,` *d*`]` respectively.

A basic set of *MathLink* functions for putting and getting data is described in Chapter 4.

■ *MathLink* Functions Built into *Mathematica* and the *MathLink* Library

The *MathLink* functions built into *Mathematica* and included in the *MathLink* library implement *MathLink* communication over various transport systems. The Unix version supports communication via pipes or TCP; Macintosh versions support TCP or PPC (program-to-program communication, a feature of System 7). *MathLink* can be used to exchange data between *Mathematica* and external programs, between a *Mathematica* kernel and a front end, or between one kernel and another. The *MathLink* library functions provide an interface between the elements of *Mathematica* expressions and external programming data types.

The *MathLink* library has put and get routines for the atomic data types that make up expressions: integers, real numbers, symbols, and strings. Each of these types is represented in your programs by a suitable native C type. There are also put and get functions to handle composite expressions, and functions that transfer a list of integers or a list of real numbers with a single call. All of these are described in Chapter 4.

The *MathLink* library also supplies routines for representing any element of data as a string of ASCII text (see Chapter 9). These are especially useful in communicating numbers whose size or precision is too great to be represented using native C numeric types.

Depending on how you use *MathLink*, you might have little or no direct use for the *MathLink* functions that are built into *Mathematica*. If you use *MathLink* to install external functions, you might only be interested in the functions `Install`, `LinkPatterns`, and `Uninstall`, which are described in Chapter 2.

If you call *Mathematica* as a subprogram, using *MathLink* mode, in which *Mathematica* automatically reads expressions from the external program and writes results back, you do not need to use any of the built-in *MathLink* functions explicitly.

If you want to carry out *MathLink* operations manually from within *Mathematica*, however, you will need to be familiar with several functions that open and close links, write to and read from a link, and control link operation in various ways. You can refer to Chapter 6 to learn about these functions.

4 Using *MathLink* in Your C Programs

■ Basic Pieces of C Programming for *MathLink*

This chapter presents the basic building blocks for using *MathLink* in C, which are as follows.

- The *MathLink* header file and the link variable declarations
- The *MathLink* library function for opening a link
- The *MathLink* library functions for putting data to or getting data from a link
- The *MathLink* library functions for checking the type of incoming data elements
- The *MathLink* library functions for concluding an outgoing expression or handling a new incoming expression
- The *MathLink* library function for closing a link

Here is a sample program that uses all of these elements.

Sample program.

Include the *MathLink* header file.

Declare a link variable.

Open a link.

Put data on the link.

Terminate the outgoing expression.

Check incoming expressions.

Check the type of the current data element.

Get data from the link.

Close the link.

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

int main(argc, argv)
    int argc; char* argv[ ];
{
    MLINK   alink;
    char *response;

    alink = MLOpen(argc, argv);
    if(alink == NULL) return 1;

    MLPutSymbol(alink, "$Version");
    MLEndPacket(alink);

    while (MLNextPacket(alink) != RETURNPKT)
        MLNewPacket(alink);

    if (MLGetType(alink) == MLTKSTR) {

        MLGetString(alink, &response);
        printf( "%s\n", response);
    } else {
        printf("Error--output is not a string.\n");
    }
    MLPutFunction(alink, "Exit", 0);
    MLEndPacket(alink);
    MLClose(alink);
    return 0;
}
```

This chapter does not cover all of the functions in the *MathLink* library, but describes a basic set that will be enough for many simple applications. More *MathLink* functions are described in Chapters 8 and 9.

■ *MathLink* Header File

`mathlink.h`	header file to include in all C source files that use *MathLink* functions

The *MathLink* header file.

Include the header file `mathlink.h` in any source file that uses functions from the *MathLink* library. This header file can be found in the *Mathematica* `Source/Includes` subdirectory in Unix versions, or in the MPW or Think C folder in the *MathLink* Developer's Kit for Macintosh computers.

■ Link Variable Declarations

`MLINK` *link*;	declare a variable that will hold a pointer to a link data structure

Link variable declaration.

Every time a program opens a link, the connection function will return an object of type `MLINK`. An `MLINK` variable is a pointer to the link data structure that is created to manage communication over a *MathLink* connection. Every time you access the connection, you have to identify the link by passing the `MLINK` object as the first argument to a *MathLink* function.

■ Opening a Link

`MLOpen(`*argc*`,` *argv*`)`	open a *MathLink* connection and return an `MLINK`-type link pointer

Opening a link from a C program.

`MLOpen` is a general function for opening a *MathLink* connection. It takes the familiar command-line argument list types as arguments: the first argument is an argument count, and the second is a null-terminated array of strings. These arguments can be the same values that are passed to `main` by the operating system. This allows you to specify connection parameters in the command line when you start up your program.

The connection parameters are discussed in Chapter 7. In many cases, you will need to specify only one parameter or none at all; for instance, no parameters need to be supplied in an installable program that will be launched from inside *Mathematica*. External programs that launch *Mathematica* can do so by passing the command-line arguments `-linkname '`*mathcommand*`'` to `MLOpen`, where *mathcommand* is the appropriate command string for starting a *Mathematica* kernel on your system. (For Unix versions this is usually `'math -mathlink'`; for Macintosh versions consult the README file in the *MathLink* Developer's Kit.)

In programs written to be used with `Install`, you do not need to call `MLOpen`; just include the function call `MLMain(`*argc*`,` *argv*`)` and it will call `MLOpen` for you.

Besides opening a link, `MLMain` sets things up so that the installed functions are automatically called when the corresponding patterns are evaluated, and so that the results are passed back to *Mathematica*.

The `MLMain` function is automatically written when you process a *MathLink* template file with `mcc` or `mprep`.

■ Put and Get Functions

The *MathLink* library has a large number of functions for writing data to or reading it from a link. The following table shows a basic set of these functions. These include puts and gets of each of *Mathematica*'s atomic data types and of composite expressions.

Functions for putting and getting numbers, symbols, and strings	
`MLPutInteger(`*link*`, `*inum*`)`	`MLGetInteger(`*link*`, &`*inum*`)`
`MLPutReal(`*link*`, `*rnum*`)`	`MLGetReal(`*link*`, &`*rnum*`)`
`MLPutSymbol(`*link*`, `*string*`)`	`MLGetSymbol(`*link*`, &`*string*`)` `MLDisownSymbol(`*link*`, `*string*`)`
`MLPutString(`*link*`, `*string*`)`	`MLGetString(`*link*`, &`*string*`)` `MLDisownString(`*link*`, `*string*`)`
Functions for putting and getting composite expressions	
`MLPutFunction(`*link*`, `*string*`, `*count*`)`	`MLGetFunction(`*link*`, &`*string*`, &`*count*`)` `MLDisownSymbol(`*link*`, `*string*`)`
`MLPutNext(`*link*`, MLTKFUNC)` `MLPutArgCount(`*link*`, `*count*`)`	`MLGetNext(`*link*`)` or `MLGetType(`*link*`)` `MLGetArgCount(`*link*`, &`*count*`)`
Argument types for the above functions	
`MLINK` *link*; `long` *count*; `char *`*string*; `int` *inum*; `double` *rnum*;	
`MLTKFUNC` is an integer constant defined in `mathlink.h`.	

Basic set of C functions for putting and getting data via *MathLink*. The appropriate argument types are as indicated at the bottom of the table. Note that some `MLGet` functions must be followed by `MLDisown` functions to allow *MathLink* to reuse the memory in which the received data were stored.

The functions in the first section of this table handle the atomic data types. These data types represent expressions that cannot be broken down into any smaller expressions.

The functions in the second section of the table are used to handle composite expressions, which are expressions that can be written as a head expression followed by a sequence of parts in square brackets. In most cases, you will work with composite expressions whose heads are simply symbols; for example, `f[a, b, c]` is a composite expression with head `f`. You can use `MLPutFunction` or `MLGetFunction` to put or get the head and argument count for this kind of expression, then continue with the argument expressions, one after another.

The functions `MLPutNext`, `MLPutArgCount`, `MLGetNext`, `MLGetType`, and `MLGetArgCount` are used for composite expressions whose heads are not symbols. For more information, see the section "Putting and Getting Composite Expressions" on page 24.

`MLDisown` functions should be used in tandem with some of the `MLGet` functions in order to manage memory properly. After your program has finished looking at a character string returned by `MLGetString`, `MLGetSymbol`, or `MLGetFunction`, it should call `MLDisownString` or `MLDisownSymbol` with the string as the second argument. This is discussed further in the following section, "Disowning Strings and Arrays".

MathLink also provides functions for putting or getting a list of integers or a list of real numbers with a single function call.

Functions for putting and getting lists of numbers	
`MLPutIntegerList(`*link*`, `*iarray*`, `*count*`)`	`MLGetIntegerList(`*link*`, &`*iarray*`, &`*count*`)` `MLDisownIntegerList(`*link*`, `*iarray*`, `*count*`)`
`MLPutRealList(`*link*`, `*rarray*`, `*count*`)`	`MLGetRealList(`*link*`, &`*rarray*`, &`*count*`)` `MLDisownRealList(`*link*`, `*rarray*`, `*count*`)`

Argument types for the above functions

```
MLINK link;
long count;
int *iarray;
double *rarray;
```

Putting and getting lists of numbers. Arguments should have the types indicated at the bottom of the table.

If your program uses the integer or real list input/output functions, remember that element 0 of a C array corresponds to part 1 in the *Mathematica* list representation.

MathLink's input/output functions can be expected to coerce data from one type to another when necessary and possible. That is, integers can be read as floating-point numbers and the other way around. Also, any data type can be read as a string.

■ Disowning Strings and Arrays

The functions `MLGetString`, `MLGetSymbol`, `MLGetFunction`, `MLGetIntegerList`, and `MLGetRealList` store received data in an area reserved for *MathLink*'s use and return pointers into this memory area. Your program should not write to this memory; it should simply examine the data or copy them to another location. Then it should call the corresponding `MLDisown` function, as listed in the tables in the previous section.

The `MLDisown` functions tell *MathLink* that you are done looking at the data returned by a previous `MLGet` function, so that *MathLink* can reuse the memory it allocated for the data.

Putting and Getting Composite Expressions

To represent an expression that is nonatomic, which means that it can be written *head*`[...]`, usually with a sequence of arguments within the brackets, *MathLink* uses the composite expression data type. After this type is specified, the argument count is given; then comes the head of the expression, followed by the arguments (if any), one after another.

Usually, the head of an expression is a symbol. In this case, you can put or get the data type, the argument count and the head of the expression with the single call `MLPutFunction(`*link*`, `*symbol*`, `*count*`)` or `MLGetFunction(`*link*`, &`*symbol*`, &`*count*`)`. For example, `MLPutFunction(alink, "Plus", 2)` tells *MathLink* that a function of two arguments will follow and that the head of the function is `Plus`. You would follow this by putting the argument parts. The expression `a + b`, whose full form is `Plus[a, b]`, could be sent as follows.

```
MLPutFunction(alink, "Plus", 2);

MLPutSymbol(alink, "a");
MLPutSymbol(alink, "b");
```

Sending the expression `a + b` from a C program.

The same expression could be read from a link by a sequence similar to the following.

```
MLGetFunction(alink, &fname, &nargs);
process_name(fname);
MLDisownSymbol(alink, fname);

MLGetSymbol(alink, &sym);
process_symbol(sym);
MLDisownSymbol(alink, sym);

MLGetSymbol(alink, &sym);
process_symbol(sym);
MLDisownSymbol(alink, sym);
```

Reading the expression `a + b` in a C program.

Note that when you use `MLGetFunction` to get a function name, you may read or copy the function name string returned by `MLGetFunction`, but you should not write to it, and when you are done referencing the string, you should call `MLDisownSymbol` to disown it.

For expressions whose heads are not symbols, but instead are themselves composite expressions, you need to use a sequence of calls in place of `MLPutFunction` or `MLGetFunction`.

To write such a composite expression to a link, use `MLPutNext(`*link*`, MLTKFUNC)`, then `MLPutArgCount(`*link*`, `*count*`)`, and then the appropriate sequence of put calls to put the head of the expression, followed by the arguments.

For example, the head of `Derivative[1][f]` is `Derivative[1]`, and this head applies to the single argument `f`. `Derivative[1][f]` would be sent by the following sequence of calls.

```
MLPutNext(alink, MLTKFUNC);      /* MLTKFUNC is the composite data type. */
MLPutArgCount(alink, 1);         /* Head and 1 argument to follow. */

MLPutFunction(alink, "Derivative", 1);  /* These two lines put the head, */
MLPutInteger(alink, 1);                 /*   namely, Derivative[1]. */

MLPutSymbol(alink, "f");                /* This line puts the argument f. */
```

Sending an expression whose head is not a symbol.

Similarly, when receiving such an expression from a link, you would first read the data type with `MLGetNext(`*link*`)` or `MLGetType(`*link*`)`—either of which would return the value `MLTKFUNC`—then you would use `MLGetArgCount(`*link*`, &`*count*`)`, and then further `MLGet` calls to get the expression's head and arguments.

`MLGetNext` and `MLGetType` are generally useful for checking the type of any incoming data element; this is discussed further on page 26. `MLPutNext`, `MLGetNext`, and `MLGetType` are also used for handling atomic data in terms of a general text representation; see Chapter 9.

■ Alternative C Types for Numbers

The input/output functions mentioned earlier for integers and real numbers represent these numbers as C types `int` and `double`, respectively. Alternatively, you may choose to use the C types `short` or `long` for integers, and you may use `float` or `long double` for real numbers.

To put or get a number using one of these alternative types, simply use the `MLPut` or `MLGet` function from the following list with the appropriate type designation in place of `Integer` or `Real` in the function name. Aside from the difference in the numeric argument's type, the syntax is the same as that used for `MLPutInteger` or `MLGetInteger`.

`MLPutShortInteger`	`MLGetShortInteger`
`MLPutLongInteger`	`MLGetLongInteger`
`MLPutFloat`	`MLGetFloat`
`MLPutDouble`	`MLGetDouble`
`MLPutLongDouble`	`MLGetLongDouble`

Input/output functions for alternative C numeric types. The argument syntax for these functions is similar to the syntax shown for `MLPutInteger`, `MLPutReal`, etc., on page 22.

In current versions, `MLPutReal` and `MLGetReal` are equivalent to `MLPutDouble` and `MLGetDouble`, but the definitions of `MLGetReal` and `MLPutReal` might differ in some future versions of *MathLink*.

Data Type Checking

`MLGetNext(`*link*`)`	return the type of the next data element to be read from the link (always go to a new data element)
`MLGetType(`*link*`)`	return the type of the current data element being read from the link (go to a new data element only if the most recent element has been completely read)

Functions for checking incoming data types.

Before you read data from a link, you can use `MLGetNext` or `MLGetType` to find out what kind of data element is coming. These functions return an integer constant corresponding to the data type of an incoming element.

`MLGetNext` always looks at a new data element. It skips past any data element that has already been looked at by any `MLGet` call.

`MLGetType` is like `MLGetNext` except that it does not skip ahead to a new data element unless the previous element has been completely read. Therefore it can be called repeatedly for the same element.

The data type codes returned by `MLGetNext` and `MLGetType` are listed in the following table.

`MLTKSTR`	*Mathematica* string
`MLTKSYM`	*Mathematica* symbol
`MLTKINT`	integer
`MLTKREAL`	real number
`MLTKFUNC`	*Mathematica* composite expression
`MLTKERROR` (== 0)	error getting data type

Predefined constants corresponding to data types.

The code for reading a complicated expression may look like the following.

```
...
switch (MLGetType(link)) {
case MLTKSTR:
    MLGetString(link, &s);
    process_string(s);
    MLDisownString(link, s);
    break;
case MLTKINT:
    MLGetInteger(link, &n);
    process_integer(n);
```

```
    break;
...
case MLTKFUNC:
    MLGetArgCount(link, &count);
    ...
    break;
default:
  ...
}
```

C code segment for reading part of a complicated expression.

`MLCheckFunction(`*link*`, `*string*`, &`*count*`)`	check whether the element to be read next is a composite expression whose head is the symbol *string*; if so, return the argument count as *count*

Function for checking incoming function expression.

If you are expecting a composite expression with a certain function name at its head, you can check for this function name with `MLCheckFunction(`*link*`, `*string*`, &`*count*`)`. This function returns 0 if the data on the link do not match the function name you expected. If the data do match, `MLCheckFunction` returns nonzero and stores the argument count of the expression in the location specified by `&`*count*.

Moving from One Expression to the Next

`MLEndPacket(`*link*`)`	mark the end of an outgoing expression
`MLNewPacket(`*link*`)`	discard what is left of the current incoming expression
`MLNextPacket(`*link*`)`	identify the type of the next incoming packet

Functions involving the boundary between one complete expression and another.

When writing to a link, you should call the *MathLink* function `MLEndPacket` each time you finish putting a complete expression onto the link. `MLEndPacket` should be used for any complete top-level expression (but not for a subexpression of a larger expression), whether or not it has a head that is a *Mathematica* packet name.

When reading from a link, you can call `MLNewPacket` in the middle of reading an expression to discard the remainder of that expression and go to the beginning of the next top-level expression. `MLNewPacket` can be used whether or not incoming expressions are in packet format.

When reading expressions that are in packet format, you can use `MLNextPacket` at the beginning of each incoming expression to determine what kind of packet it is. `MLNextPacket` returns an integer constant corresponding to the packet type. It returns 0 and sets the *MathLink* error condition if the head of the received expression is not a legal packet name.

Packet format is normally used for all expressions sent by *Mathematica* when it runs as a subprogram in *MathLink* mode. *MathLink* mode and the packet types recognized by `MLNextPacket` are described in Chapter 5.

Using `MLNextPacket`, you can easily construct a C `switch` statement that dispatches to a different part of your program for each packet type.

```
...

switch (MLNextPacket(link)) {
case INPUTPKT:
    MLGetString(link, &s);
    take_input(s);
    MLDisownString(link, s);
    break;
...
case MESSAGEPKT:
    MLGetSymbol(link, &sym);
    MLGetString(link, &s);
    process_message(n);
    MLDisownSymbol(link, sym);
    MLDisownString(link, s);
    break;
...
case RETURNPKT:
    read_expression(link);
    ...
    break;
default:
  ...
}
```

Typical C code for processing a packet.

`MLNextPacket` checks that you are at the beginning of a packet and uses `MLGetFunction` to read the packet head. Subsequent `MLGet` calls read the arguments of the expression. It is an error to call `MLNextPacket` before all the subexpressions of the current packet have been read.

■ Closing a Link

`MLClose(`*link*`)`	disconnect the link

Terminating a link from a C program.

Use `MLClose(`*link*`)` to close a link. Your program must close all links it has opened before terminating.

5 Running *Mathematica* as a Subprogram

■ *MathLink* Mode

When *Mathematica* is started in *MathLink* mode by another process, all expressions sent via *MathLink* from *Mathematica* to the parent process have heads that specify a packet type. For example, `TextPacket[`*string*`]` represents text, as produced by the *Mathematica* function `Print`. `MessagePacket[`*symbol*`,`*name*`]` represents a *Mathematica* warning message. The result of a calculation is given in the form `ReturnPacket[`*expr*`]` or `ReturnTextPacket[`*expr*`]`.

`InputPacket[`*string*`]`	`INPUTPKT`	prompt for input, as generated by *Mathematica*'s `Input` function
`TextPacket[`*string*`]`	`TEXTPKT`	text output from *Mathematica*, as produced by `Print`
`ReturnPacket[`*expr*`]`	`RETURNPKT`	result of a calculation
`ReturnTextPacket[`*string*`]`	`RETURNTEXTPKT`	formatted text representation of a result
`MessagePacket[`*symbol*`, `*string*`]`	`MESSAGEPKT`	*Mathematica* message identifier (*symbol*`::`*string*)
`CallPacket[`*integer*`, `*list*`]`	`CALLPKT`	request to invoke the external function numbered *integer*, with arguments in *list*
`InputNamePacket[`*string*`]`	`INPUTNAMEPKT`	name to be assigned to the next input (usually `In[`*n*`]:=`)
`OutputNamePacket[`*string*`]`	`OUTPUTNAMEPKT`	name to be assigned to the next output (usually `Out[`*n*`]=`)
`DisplayPacket[`*string*`]`	`DISPLAYPKT`	part of a PostScript graphic description
`DisplayEndPacket[`*string*`]`	`DISPLAYENDPKT`	end of graphic description
`SyntaxPacket[`*integer*`]`	`SYNTAXPKT`	position at which a syntax error was detected in the input line
`MenuPacket[`*integer*`, `*string*`]`	`MENUPKT`	a number specifying a particular menu (*e.g.*, the interrupt menu) and a prompt string

Packet names, type codes returned by `MLNextPacket`, and description of enclosed expression(s).

Typically several types of packets are generated for a single evaluation.

When a message is generated, the message packet is normally followed by a text packet giving the full text of the error message.

Menu packets are generated when *Mathematica* wants input to tell it how to proceed in special circumstances when there are several options, such as when you interrupt a calculation. In a menu packet, a menu number of 0 indicates that the previous menu selection was invalid. If this is the case, the menu packet is followed by a text packet giving detailed instructions.

You will probably want to use *MathLink* mode if you have written a program that will call *Mathematica* as a subprogram, particularly if your program is designed to be a front end for *Mathematica*. To start a *Mathematica* kernel in *MathLink* mode on a Unix system, you should specify the option `-mathlink` in the `math` command line. If you have a Macintosh computer, refer to the README file in the *MathLink* Developer's Kit for information on starting a *Mathematica* kernel in *MathLink* mode.

■ An Example of an External Program That Runs *Mathematica* in *MathLink* Mode

Here is a simple example of a C program that is designed to run *Mathematica* in *MathLink* mode. When the program is run in a Unix environment with the command-line arguments `-linkname 'math -mathlink'`, it launches *Mathematica* and has it calculate the sum of two integers. (The way in which `argc` and `argv` are supplied is system dependent; for Macintosh systems you should follow the `addinteger.c` example or other examples provided in the *MathLink* Developer's Kit.)

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

int main(argc, argv)
    int argc;
    char *argv[ ];
{
    int     i, j, sum;
    MLINK   link;

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

    link = MLOpen(argc, argv);
    if (link == NULL) return 1;

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

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

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

    printf("sum = %d\n", sum);

    /* quit Mathematica then close the link */
    MLPutFunction(link, "Exit", 0);
    MLEndPacket(link);
    MLClose(link);
    return 0;
}
```

addinteger.c.

What *Mathematica* Does When in *MathLink* Mode

In the *MathLink* mode main loop, *Mathematica* reads an expression from *MathLink*, evaluates it, and writes results back to *MathLink*. The usual standard input and output channels are replaced by the link object `$ParentLink`. Therefore, a user cannot interact directly with *Mathematica* when it is in this mode.

In *MathLink* mode, all output data are wrapped with appropriate packet heads. The result of a calculation is wrapped in `ReturnPacket` (or `ReturnTextPacket`, if the input expression has the head `Enter`; see the following paragraph). Other kinds of output have different heads. For example, the *Mathematica* function `Print` writes its data to `$ParentLink` wrapped with `TextPacket`. Similarly, `Message` sends data inside `MessagePacket`, and `Display` sends data inside `DisplayPacket`. (Note: `Display` for a single picture can produce a sequence of `DisplayPackets`; to mark the end of the picture, the last data packet is wrapped with `DisplayEndPacket`.)

If your external program is to act as a complete front end to *Mathematica*, then you will probably want to define several kinds of requests that you can send to *Mathematica*. You can implement different requests by wrapping expressions in functions that specify what to do with the expressions inside. A special case is the `Enter` function; if *Mathematica* receives the expression `Enter["`*string*`"]`, it interprets *string* as an input expression and completely processes it. The processing includes incrementing the line number and sending input and output labels. The final result generated by `Enter` is converted into a string and sent as a `ReturnTextPacket`.

`Enter["`*string*`"]`	perform complete processing of the *string* as *Mathematica* input, sending output through *MathLink* to the parent process or front end

Special form of input to *Mathematica* in *MathLink* mode.

You can easily see the form of output produced by *Mathematica* in *MathLink* mode by starting a *Mathematica* subprocess from within a normal interactive *Mathematica* session.

This sort of experiment can help you get a better idea of what *Mathematica* does when it runs in *MathLink* mode, as well as letting you try out some *MathLink* operations manually from a *Mathematica* session. The built-in *MathLink* functions used in this example are described in Chapter 6.

Running a *Mathematica* subprocess from within *Mathematica*.

Start another copy of *Mathematica* in `mathlink` mode. Assign the link object to the symbol `link`.

```
In[1]:= link = LinkOpen["math -mathlink -noinit"]

Out[1]= LinkObject[math -mathlink -noinit, 1, 1]
```

First the child process sends an input prompt label.

```
In[2]:= LinkRead[link]

Out[2]= InputNamePacket[In[1]:= ]
```

Send 2+2, without first evaluating it locally.

```
In[3]:= LinkWriteHeld[link, Hold[2+2]]

In[4]:= LinkRead[link]
```

The result is wrapped in `ReturnPacket`.

```
Out[4]= ReturnPacket[4]
```

The expression 1/0 will generate a warning message.

```
In[5]:= LinkWriteHeld[link, Hold[1/0]]
```

A message packet comes first.

```
In[6]:= LinkRead[link]

Out[6]= MessagePacket[Power, infy]
```

The text of the message follows.

```
In[7]:= LinkRead[link]

Out[7]= TextPacket[

>                                                  1            ]
          Power::infy: Infinite expression - encountered.
                                                   0
```

The return packet comes last.

```
In[8]:= LinkRead[link]

Out[8]= ReturnPacket[ComplexInfinity]
```

Now send an input string wrapped in `Enter`.

```
In[9]:= LinkWrite[link,
        Enter["Factor[3x^2 - 7x + 2]"] ]
```

In response, *Mathematica* first sends an output label.

```
In[10]:= LinkRead[link]

Out[10]= OutputNamePacket[Out[1]= ]
```

The result computed from the `Enter` expression comes in the form of a string wrapped in `ReturnTextPacket`.

```
In[11]:= LinkRead[link]

Out[11]= ReturnTextPacket[(-2 + x) (-1 + 3 x)]
```

Here is the new input prompt.

```
In[12]:= LinkRead[link]

Out[12]= InputNamePacket[In[2]:= ]
```

In the example, the command-line switch `-noinit` is used to suppress startup messages. Try it without this switch to see what happens. You may find that the second *Mathematica* process sends you a few extra packets before it sends the first input name packet.

Notice that the line number in the child session is incremented and input/output labels generated only when an `Enter` expression is processed.

■ Other Ways to Call *Mathematica* from an External Program

You do not have to use *MathLink* mode to call *Mathematica* from an external program. If you start *Mathematica* and an external program independently and then create a peer-to-peer connection between them, you can handle reads and writes to the external program manually from within *Mathematica*. Another alternative is to launch *Mathematica* from within an external program, but with a special `init.m` file or a batch input file giving specific *Mathematica* commands for establishing and maintaining communication with the parent program.

These methods require careful application of *Mathematica*'s built-in *MathLink* functions and correct use of *MathLink* connection parameters when you open your links. *Mathematica*'s built-in *MathLink* functions are discussed in Chapter 6. To learn how to establish different kinds of connections, see the discussion of link parameters in Chapter 7.

You might also find it useful to put *Mathematica* into *MathLink* mode after establishing a peer-to-peer connection with an external program. There is an example on page 42 that does this.

6 Using a Link Manually from Inside *Mathematica*

The previous chapters in this guide describe two ways to have *Mathematica* handle *MathLink* communication automatically. When you install an external program by the method of Chapter 2, its functions are automatically invoked when you evaluate certain expressions within *Mathematica*; and when you call *Mathematica* in *MathLink* mode from an external program, *Mathematica* automatically uses *MathLink* to read input and write output to the external program.

For more general applications, you might want to handle a link manually within an interactive *Mathematica* session by using built-in functions for *MathLink* communication. These functions are discussed in this chapter.

Opening a Link

`LinkOpen["`*name*`", LinkMode->`*mode*`, LinkProtocol->`*protocol*`, LinkHost->`*host*`]`	
	open a link to the external program indicated by *name*, using the specified link parameters

Opening a link from *Mathematica*.

`LinkOpen` is *Mathematica*'s built-in analog to the *MathLink* library function `MLOpen` (see page 21). Like `MLOpen`, it opens a link according to link parameters that tell it how to establish a connection and what to connect to. The link name parameter is given as an argument to `LinkOpen`; this is the name of an external program to be launched or the name of a communication port to be used in making a connection. The other parameters are specified as options. `LinkOpen` assumes default values for parameters that are not supplied.

The link name argument can be omitted if you are opening a link in Listen mode and you want *MathLink* to pick a port name for you.

The `LinkMode` option can be set to `Launch`, `ParentConnect`, `Listen`, or `Connect`. `LinkProtocol` can be set to `"TCP"` or `"pipes"` in Unix versions, and to to `"TCP"` or `"PPC"` in Macintosh versions (PPC is the program-to-program communication protocol built into Macintosh System 7). `LinkHost` is used when the other partner in the communication is located on another machine, in which case `LinkHost` is set to a string giving the remote host's name. The various link parameters are discussed further in Chapter 7.

`LinkOpen` returns a *Mathematica* link object, which has the form `LinkObject["`*name*`",` *serialno*`,` *channo*`]`. The second argument, *serialno*, is a unique "serial number", which specifies which invocation of `LinkOpen` this particular link is associated with. By including a serial number in the link object, *Mathematica* allows you to run several copies of the same external process, and deal with each one separately. The third argument, *channo*, is a "channel number"; *MathLink* uses it internally.

`LinkOpen` is called internally when you use `Install` to launch an external program, as described in Chapter 2.

The link object returned by `LinkOpen` should be stored in a variable; you will have to supply it as the first argument in subsequent operations on the link.

Write and Read Commands

The *MathLink* input/output commands built into *Mathematica* are `LinkWrite`, `LinkWriteHeld`, `LinkRead`, and `LinkReadHeld`.

`LinkWrite[`*link*`, `*expr*`]`	write *expr* to the link
`LinkWriteHeld[`*link*`, Hold[`*expr*`]]`	write *expr* to the link without evaluating it
`LinkRead[`*link*`]`	read an expression from the link
`LinkReadHeld[`*link*`]`	read an expression from the link and wrap it in `Hold` to keep it unevaluated

Writing to or reading from a link in *Mathematica*.

Each of these commands writes or reads one *Mathematica* expression.

`LinkWriteHeld` or `LinkReadHeld` will send or receive an expression without evaluating it. `LinkWriteHeld` takes an expression wrapped in `Hold` and writes it to the link without the `Hold`. `LinkReadHeld` receives an expression and wraps it in `Hold`.

Closing a Link

`LinkClose[`*link*`]`	close the link

Closing a link from *Mathematica*.

`LinkClose[`*link*`]` is *Mathematica*'s built-in command for closing a link. For external programs that you have linked to *Mathematica* by using the `Install` function, you may close the link by entering `Uninstall[`*link*`]`.

Other *MathLink*-related Commands

These other *MathLink*-related commands are built into *Mathematica*.

`LinkReadyQ[`*link*`]`	returns `True` if data are immediately available to be read from the link
`LinkError[`*link*`]`	returns the *MathLink* error status of the link and the corresponding error message string
`LinkInterrupt[`*link*`]`	interrupts a calculation being performed by an installed external program or a second *Mathematica* process (see page 46)
`$LinkSupported`	is `True` for versions of *Mathematica* that support *MathLink*
`$ParentLink`	the link object being served by the *Mathematica* main loop, or `Null` if not in *MathLink* mode
`Links[ ]`	returns a list of the currently active links

Some other *MathLink* functions built into *Mathematica*.

`LinkReadyQ` allows you to test for incoming data on a link before attempting to read. This allows you to have *Mathematica* do other things instead of blocking and waiting for data to arrive.

`LinkError` gives you the error status of a link.

`LinkInterrupt` will interrupt a calculation in an external program if the program has certain special interrupt-handling functions built in. Installed external functions that were built with `mcc` and `mprep` are generally able to respond to `LinkInterrupt`; see page 46 for more on this topic.

`LinkReadyQ`, `LinkError`, and `LinkInterrupt` are related to the *MathLink* library functions `MLReady`, `MLError`, and `MLPutMessage`, which are described in Chapter 8.

`$ParentLink` is a global variable that determines whether *Mathematica* operates in *MathLink* mode and if so, what link it takes input from. Page 42 has an example showing the effect of setting `$ParentLink` manually.

7 More about Opening a *MathLink* Connection

Connection Parameters

In your C programs that use *MathLink*, the parameters in the argument list you pass to `MLOpen` or `MLMain` tell *MathLink* what kind of connection you want to create and where to find the other partner in the communication. `MLOpen` looks for the following sequences.

`-linkname` *name*	gives the name of the entity to connect to; this may be a port name or a program command line
`-linkmode` *mode*	gives the mode of opening the link; this must be Listen, Connect, Launch, or ParentConnect (these can be entered with capital or lower-case letters)
`-linkprotocol` *protocol*	specifies the data transfer protocol to be used; choices are TCP (Unix or Macintosh systems), PPC (Macintosh), and pipes (Unix)
`-linkhost` *hostname*	identifies the machine on which the other partner to the link is to be found

Connection parameters taken by `MLOpen`.

When you open a link in *Mathematica*, you give the link name as an argument to `LinkOpen` or `Install`, and you may give the mode, protocol, and host by setting the options `LinkMode`, `LinkProtocol`, and `LinkHost`.

```
LinkOpen["name", LinkMode->mode, LinkProtocol->protocol, LinkHost->host]
Install["name", LinkMode->mode, LinkProtocol->protocol, LinkHost->host]
```

Specifying connection parameters when opening a link in *Mathematica*. For most purposes, you do not need to specify all of the options.

In most cases, it is not necessary to specify all four of the link parameters; `MLOpen` or `LinkOpen` will try to infer the correct values for missing parameters. It may also check the environment and prompt the user for more information if necessary. Note that some combinations of parameters are not valid. For example, you cannot open a link in Connect or Listen mode using the "pipes" protocol.

Mode

MathLink provides two methods for establishing a connection between two processes. In one case a parent process creates and connects to a child process; in the other, the two processes are started independently and act as peers.

The relationship that each process has to its partner when the connection is being established is called its link mode. For parent-child connections, the parent uses Launch mode and the child uses ParentConnect mode; for peer-to-peer connections one process uses Listen mode and the other uses Connect mode. Once a connection has been established, no further reference needs to be made to the link mode.

Name

A parent process that wants to launch a child process must provide a filename to launch. For peer-to-peer connections, one side must create a named port and listen for connection requests, and the other side must name the port it wishes to connect to. So, when establishing a *MathLink* connection, a name must be given as well as a mode.

A port is an operating system resource used for communication. Ports are named so that one port can call and connect to another by name. The format of a port's name is determined by the underlying interprocess communication protocol. For example, a TCP port name is a positive integer (such as `2300`), whereas a PPC port name is an arbitrary word (such as `otherProgram`).

If the link mode is Listen and the link name is not specified, `MLOpen` or `LinkOpen` will select an arbitrary valid port name.

Protocol

On Unix systems, *MathLink* connections may operate through pipes or over TCP. On Macintosh systems, *MathLink* can run over PPC or TCP. PPC stands for program-to-program communication, a feature that is built into Macintosh System 7.

Not all of these protocols are supported for every link mode on a given system. Therefore, the choice of link mode alone may be enough to determine which protocol is used.

Host

If the link name given to `MLOpen` or `LinkOpen` refers to a communication port on another computer, the link host parameter must be specified to identify the other host machine.

Examples of Parameter Settings

A C program running on a machine called `spider` that executes this code:

```
MLINK link;
int argc = 6
char *argv[ ] = {
                "-linkmode",     "listen",
                "-linkname",     "3000",
                "-linkprotocol", "TCP",
                0 };
link = MLOpen(argc, argv);
```

Opening a link from C in Listen mode.

will accept and establish a connection with a *Mathematica* session that evaluates the following input.

```
In[1]:= LinkOpen["3000", LinkMode->Connect,
                LinkProtocol->"TCP", LinkHost->"spider"]
```

Opening a link from *Mathematica* in Connect mode.

A *Mathematica* session that evaluates this input:

```
In[1]:= link = LinkOpen["prog", LinkMode->Launch]
```

Opening a link from *Mathematica* in Launch mode.

will start and connect to a C program named `prog` if that program executes the following statements.

```
MLINK link;
int argc = 2
char *argv[ ] = {"-linkmode", "parentconnect", 0};
link = MLOpen(argc, argv);
```

Opening a link from C in ParentConnect mode.

Note that the function `MLOpen` is designed to take command-line arguments. So, rather than constructing *argc* and *argv* in the text of your program as in the foregoing examples, you would likely use the command-line arguments passed to your `main` function by the runtime environment. Also notice that the link name parameter is the only argument to the *Mathematica* function `LinkOpen`. The other parameters appear as options.

Default Behavior

Actually, `MLOpen` and `LinkOpen` provide a great deal of default behavior if some or all of the link parameters are not specified. In particular, if the link mode is not specified it will be set to Launch if a link name is given, or to ParentConnect if not. Then the link protocol, if not specified, is chosen to be some default based on the link mode. The link host, if not given, is chosen to be "this" computer. If no link name is specified, one will be chosen for you, or you may be asked to provide one.

If no information at all is provided via *argc* and *argv*, `MLOpen` will ask the environment in some system-dependent way if it can provide any information beyond what it passed to `main`. Finally, if the environment provides no information and `MLOpen` can determine that this process was launched by a user rather than a candidate parent process, it asks the user for help. Otherwise, `MLOpen` assumes the ParentConnect mode.

■ Using Install with Listen and Connect Modes

Here is an example in which the external program `addtwo` listens on port "5000" and waits for a connection request. *Mathematica* then connects to this port to install `addtwo`.

External environment.

Start the `addtwo` program from outside of *Mathematica*, and have it listen on port "5000".

```
addtwo -linkmode listen -linkname 5000
```

Mathematica session.

Supply the port name as the argument to `Install`.

```
In[1]:= addlink = Install["5000", LinkMode->Connect]

Out[1]= LinkObject[5000@spider, 1, 1]

In[2]:= AddTwo[3, 4]

Out[2]= 7

In[3]:= Uninstall[addlink]

Out[3]= 5000@spider
```

In the following example, the listening port is created by *Mathematica*. Once the connection is established, the two examples are indistinguishable. Use whichever you find more convenient.

Mathematica session.

Create a link in Listen mode on port "5000".

```
In[1]:= link = Install["5000", LinkMode->Listen]
```

External environment.

Start the external program and have it use Connect mode to connect to port "5000".

```
addtwo -linkmode connect -linkname 5000
```

Mathematica session.

```
Out[1]= LinkObject[5000@spider, 1, 1]

In[2]:= AddTwo[3, 4]

Out[2]= 7
```

```
In[3]:= Uninstall[link]

Out[3]= 5000@spider
```

Use of Standard Input and Output

In Unix environments, a program launched by *MathLink* cannot use standard terminal input and output because the standard input and output channels (`stdin` and `stdout`) are used by *MathLink* for communication with the parent program. However, the child program can use `stderr` for its output, or it can open a terminal file independently for input and output that avoids the *MathLink* channels.

If you use Listen and Connect modes to establish a peer-to-peer connection between *Mathematica* and another program, then you are free to use `stdin` and `stdout` as you wish in the external program.

Using *Mathematica* through a Peer-to-Peer Connection

In the following example, two *Mathematica* processes establish and communicate through a peer-to-peer connection. This allows both processes to run as interactive sessions.

Session "A".

Rename the *Mathematica* prompts for clarity.

```
In[1]:=  (Unprotect[In, Out];
Format[In] = ASideIn; Format[Out] = ASideOut;)

ASideIn[2]:= linkToB = LinkOpen["5500@moose",
LinkMode->Listen]

ASideOut[2]= LinkObject[5500@moose, 1, 1]
```

Session "B".

```
In[1]:= (Unprotect[In, Out];
Format[In] = BSideIn; Format[Out] = BSideOut;)

BSideIn[2]:= linkToA = LinkOpen["5500@moose",
LinkMode->Connect]

BSideOut[2]= LinkObject[5500@moose, 1, 1]
```

Session "A".

Send an expression from "A" to "B".

```
ASideIn[3]:= LinkWrite[linkToB, N[Pi]]
```

Session "B".

```
BSideIn[3]:= LinkRead[linkToA]

BSideOut[3]= 3.14159265358979
```

Send an expression from "B" to "A".

```
BSideIn[4] := LinkWrite[linkToA, Sqrt[%]]
```

Session "A".

```
ASideIn[4] := LinkRead[linkToB]

ASideOut[4]= 1.77245385090552
```

You should notice that the processes operate as peers and that expressions sent over the link are not wrapped in packet heads, because neither copy of *Mathematica* was started in *MathLink* mode.

You may want to switch *Mathematica* into *MathLink* mode after establishing a peer-to-peer connection with another process. To do so, simply set `$ParentLink` equal to the link object attached to the other process.

Watch what happens in this example when `$ParentLink` in session "A" is set to `linkToB`. From session "B", the newly subordinated "A" side now acts as if it were started in *MathLink* mode. From the "A" side, it appears as if the setting of `$ParentLink` does not return (it actually returns its output to session "B"). When `$ParentLink` is subsequently set to `Null`, the two sides again act as peers.

Session "A" continued.

Tell "A" to go into *MathLink* mode serving session "B".

```
ASideIn[5] := $ParentLink = linkToB
```

Session "B".

Session "B" receives the output from `ASideIn[5]`.

```
BSideIn[5] := LinkRead[linkToA]

BSideOut[5]= OutputNamePacket[ASideOut[5]=]

BSideIn[6] := LinkRead[linkToA]

BSideOut[6]= ReturnTextPacket[LinkObject[5500@moose, 1, 1]]

BSideIn[7] := LinkRead[linkToA]

BSideOut[7]= InputNamePacket[ASideIn[6]:=]
```

Send an expression for simple evaluation.

```
BSideIn[8] := LinkWriteHeld[linkToA, Hold[2+2]]

BSideIn[9] := LinkRead[linkToA]

BSideOut[9]= ReturnPacket[4]
```

Send an input string for full processing.

```
BSideIn[10] := LinkWrite[linkToA, Enter["2+2"]]

BSideIn[11] := LinkRead[linkToA]

BSideOut[11]= OutputNamePacket[ASideOut[6]=]
```

Note that the result is sent in a return text packet.

```
BSideIn[12]:= LinkRead[linkToA]

BSideOut[12]= ReturnTextPacket[4]

BSideIn[13]:= LinkRead[linkToA]

BSideOut[13]= InputNamePacket[ASideIn[7]:=]

BSideIn[14]:= LinkWriteHeld[linkToA,
                     Hold[$ParentLink = Null]]
```

Session "A".

```
ASideIn[8]:=
```

Notice that the first assignment to `$ParentLink` resulted in "full processing". That is, an output prompt was produced, the answer was written in a `ReturnTextPacket`, the line number was incremented, and an input prompt was produced. Subsequent expressions sent from session "B" are not given full processing unless they are sent as input strings wrapped in `Enter`. Compare the results of `BSideIn[8]` and `BSideIn[10]` to see the difference between full processing and simple evaluation.

In the following example, the `addinteger` program from page 30 is rerun using a peer-to-peer connection.

Mathematica session.

Open a listening link. For Listen mode, `LinkOpen` can choose a port name for you.

```
In[1]:= link = LinkOpen[LinkMode->Listen]

Out[1]= LinkObject[1430@spider, 1, 1]
```

External environment.

Start `addinteger` from outside of *Mathematica*.

```
addinteger -linkmode connect -linkname 1430
    -linkhost spider
```

Type the input.

```
==>3 4
```

Mathematica session.

From *Mathematica*, see what was sent.

```
In[2]:= LinkReadHeld[link]

Out[2]= Hold[3 + 4]
```

Cooperate by sending back the result. (In *MathLink* mode, of course, *Mathematica* would evaluate the expression and return the result without your help.)

```
In[3]:= LinkWrite[link, ReturnPacket[ReleaseHold[%]]]
```

External environment.

```
sum = 7
```

8 Link Status and Interrupt Functions

Error Functions and Conditions

Many kinds of errors can occur while you are putting or getting data via *MathLink*. Whenever an error occurs, the *MathLink* function you have called returns 0, and *MathLink* goes into an inactive state, in which *MathLink* functions have no effect and always return 0.

`MLError(`*link*`)`	indicate whether an error has occurred since `MLClearError` was last called, and if so, what kind
`MLClearError(`*link*`)`	clear a *MathLink* error
`MLErrorMessage(`*link*`)`	a text string describing the current error

Error functions.

To find out whether an error has occurred on a particular *MathLink* link, and what kind of error it was, you can call the function `MLError`. `MLError` will return the same value repeatedly until you call `MLClearError`. `MLError` returns `MLEOK` (== 0) if no error has occurred, and returns a nonzero error code otherwise.

When you are trying to read and store a complicated data structure with *MathLink*, it is sometimes convenient to avoid checking the return value from each *MathLink* function you call, and instead to call `MLError` when you are finished, to see if any errors in fact occurred. The fact that *MathLink* functions become inactive after any error occurs makes this a fairly safe procedure.

The following code segments illustrate two ways to read a list of two real numbers with error checking.

```
MLCheckFunction(alink, "List",  &len);
if (len != 2) ERROR;
MLGetReal(alink, &x);
MLGetReal(alink, &y);
if (MLError(alink)) ERROR;
```

Code for reading a list of two reals from *MathLink*.

```
MLGetRealList(alink, &rvec, &len);
if (MLError(alink) || len != 2) ERROR;
use_list(rvec, len);
MLDisownRealList(alink, rvec, len);
```

Another way to read a list of two reals.

`MLEOK` (== 0)	no error has occurred
`MLEDEAD`	an unrecoverable error has occurred; the other side may have exited
`MLEGBAD`	inconsistent data were encountered in the stream
`MLEGSEQ`	an `MLGet` function was called out of sequence
`MLEPBTK`	a bad data type was passed to `MLPutNext`
`MLEPSEQ`	an `MLPut` function was called out of sequence
`MLEPBIG`	more data were put to the stream using `MLPutData` than was indicated by `MLPutSize`
`MLEOVFL`	machine integer overflow in `MLGetInteger`
`MLEMEM`	not enough space to allocate memory for a string
`MLEACCEPT`	failure to accept a connection
`MLECONNECT`	connection has not yet been established with partner
`MLECLOSED`	link closed by other side; you may still get undelivered data
`MLEPUTENDPACKET`	unexpected or missing call of `MLEndPacket`
`MLENEXTPACKET`	`MLNextPacket` called while the current expression has unread data
`MLEUNKNOWNPACKET`	`MLNextPacket` read in an unknown packet head
`MLEGETENDPACKET`	unexpected end of expression
`MLEABORT`	a put or get was aborted before affecting the link

Error codes.

■ MLReady and MLFlush

If your program makes an `MLGet` call when there are no data waiting on the link, your program blocks until more data arrive. When new data arrive from *Mathematica*, the get operation proceeds, and your program continues. If, on the other hand, incoming data arrive before your program asks for them, *MathLink* buffers the data until the next `MLGet` call, which can immediately process the data in the buffer.

If you want to find out whether data are waiting before you call an `MLGet` function, you should use *MathLink*'s `MLReady` function. `MLReady` returns nonzero when the incoming buffer has data, and 0 when it has none. If no incoming data are present, you may have your program perform other operations for a while before testing again.

`MLReady[`*link*`]`	returns nonzero if data are ready to be read from *link* immediately
`MLFlush[`*link*`]`	transmit immediately any outgoing data that are currently buffered on *link*

Functions for checking incoming data and flushing outgoing data.

Also, data being sent from your program to *Mathematica* are buffered, so that they may be collected and transmitted in an efficient manner. Occasionally you might want to make sure that all buffered data are sent before your program proceeds further. This is called flushing the buffer. To flush the outgoing data buffer for a link, call the *MathLink* function `MLFlush(`*link*`)`.

■ Interrupting a Calculation over *MathLink*

Mathematica can send an interrupt to an installed external program to abort the current operation. This happens, for example, when the user pressesCONTROL-C or COMMAND-. and chooses `abort` from the interrupt menu. The result is that the global variable `MLAbort` is asynchronously set to `1` in the external program.

In your installable C programs, any function that takes more than a moment to execute should periodically check the value of `MLAbort`. If it is set, the function should clean up and return as quickly as possible. If the function has a manual return type, it should put the symbol `$Aborted` on the link to *Mathematica* (usually `stdlink`) before returning. Otherwise, it can return any value. `MLAbort` is reset to 0 before processing begins on the next call.

Interrupt key (CONTROL-C or COMMAND-.), then `abort`	abort a calculation being done by an installed external function from *Mathematica*
`LinkInterrupt[`*link*`]`	abort a calculation in another *Mathematica* process (if it is running in *MathLink* mode)

Interrupting a linked process from *Mathematica*.

`LinkInterrupt[`*link*`]` is a *Mathematica* command that is called internally when you abort an external calculation. You would not type it in yourself in such a case. (You could try to type it in while the external calculation was in progress, but it would not be evaluated until the calculation was complete.)

However, it can be useful to type in `LinkInterrupt[`*link*`]` if you are running a second *Mathematica* process as a subprogram from within a *Mathematica* session; in this case, `LinkInterrupt` can interrupt a calculation being performed by the second *Mathematica* process.

An external program that runs *Mathematica* in *MathLink* mode can interrupt a *Mathematica* calculation by making the call `MLPutMessage(`*link*`, MLInterruptMessage)`.

`MLPutMessage(`*link*`, MLInterruptMessage)`	interrupt *Mathematica* from an external program

Interrupting *Mathematica* from an external program.

9 Putting and Getting Data in Text Form

For most purposes, the *MathLink* functions described in Chapter 4 are sufficient for sending and receiving any expression you want to send or receive, as long as your data fit in C's native types. However, in some special cases, you may need a general way to transmit some data in the form of arbitrarily long text sequences. *MathLink* has *textual interface* functions that allow you to do this.

■ Textual-Interface Functions in the *MathLink* Library

MathLink's textual interface represents each atomic data object (integer, real number, symbol, or string) as a *data string* composed of ASCII characters, with an associated data type. To put an element in text form, you must first use `MLPutNext` to give the type, then `MLPutSize` to specify the size in bytes of the data string, then one or more `MLPutData` calls to put the data string itself. Between `MLPutData` calls, you can call `MLBytesToPut` to check how many more bytes are left to put.

Several `MLPutData` calls can be used to put data for a single atom, provided the total length is equal to that specified by `MLPutSize`.

`MLPutNext(`*link*`,` *type*`)`	specify the type of the data to follow; *type* is an integer equal to one of the data type codes listed on page 49
`MLPutSize(`*link*`,` *count*`)`	specify the length in bytes of the data string
`MLPutData(`*link*`,` *string*`,` *count*`)`	put *count* bytes from *string* onto the link
`MLBytesToPut(`*link*`, &`*count*`)`	find out the number of bytes that still need to be put and store this number in *count*

Putting data as text. In all cases, *link* is of type `MLINK` and *count* is of type `long`.

To receive an element in text form, you must first check its type by calling `MLGetNext` or `MLGetType`, then you can read the data by using one or more `MLGetData` calls. Before you call `MLGetData`, you may call `MLBytesToGet` to find out how many bytes of data remain to be read.

`MLGetNext(`*link*`)` or `MLGetType(`*link*`)`	check the type of the next data element from *link* (type `MLINK`); `MLGetNext` will always advance to a new element, but `MLGetType` only advances if the previous element has been completely read
`MLGetData(`*link*`,` *buff*`,` *max*`, &`*count*`)`	read at most (`long`) *max* bytes from link into (`char *`) *buff*; write the number of bytes actually read to *count*
`MLBytesToGet(`*link*`, &`*count*`)`	find out the number of bytes that remain to be gotten and store this number in *count*

Getting data as text. In all cases, *link* is of type `MLINK` and *count* is of type `long`.

Note the difference between `MLGetNext` and `MLGetType`. `MLGetNext` gets the type of the next expression in the *MathLink* data stream, discarding any data not yet read before it; it returns this type as an integer constant. `MLGetType` gets the type of the current elementary expression in the *MathLink* data stream; it does not discard data or move on to the next element, and therefore a program can call `MLGetType` several times for the same data element.

The data types used by `MLPutNext`, `MLGetNext`, and `MLGetType` are specified as integer constants. These values are defined in the *MathLink* header file `mathlink.h`.

`MLTKSTR`	*Mathematica* string
`MLTKSYM`	*Mathematica* symbol
`MLTKINT`	integer
`MLTKREAL`	real number
`MLTKFUNC`	*Mathematica* composite expression

Predefined constants corresponding to data types.

Note that text data are never immediately associated with the composite expression type `MLTKFUNC`. The next put call after `MLPutNext(`*link*`, MLTKFUNC)` would be an `MLPutArgCount` call; and when `MLGetNext` or `MLGetType` returns `MLTKFUNC`, the next get call would be `MLGetArgCount` or `MLGetFunction`. See "Putting and Getting Composite Expressions" on page 24.

When to Use the Text Format

The main reason to use the textual format is to get more flexibility in dealing with big numbers and long strings. For example, the integer

123456789123456789123456789123456789

is probably too big to fit into a machine word of your computer. Therefore it cannot be passed to `MLPutInteger`, but you can send it over *MathLink* by using the following sequence.

```
MLPutNext(alink, MLTKINT);
MLPutSize(alink, 36L);
MLPutData(alink, "123456789123456789123456789123456789", 36L)
```

Sending a very long integer through *MathLink*.

When you receive long strings over *MathLink*, you can process parts of them independently. In some cases you may need to process only the beginning of the string and ignore the rest. The textual-interface functions described in this chapter are more efficient for this than `MLGetString`, which may allocate a buffer for the whole string and read it there. The following example illustrates this technique.

```
if (MLGetType(link) == MLTKSTR) {
    long len, truelen;
    char buff[BUFSIZ];

    while(MLBytesToGet(link, &len), len > 0) {
        if (len > BUFSIZ) len = BUFSIZ;
        MLGetData(link, buff, len, &truelen);
        assert(len == truelen);
        if (ProcessData(buff) == ProcessingDone) {
            MLGetNext(link);
            break;
        }
    }
}
```

Processing a long string with a small buffer.

Here `ProcessData` is a function that processes the partial string and returns `ProcessingDone` when it decides that no more data in this string present any interest.

To send numbers using `MLPutData`, use text strings consisting of ASCII digits. Floating-point numbers are given in the traditional scientific notation accepted in programming languages like C and Fortran. For example, 6.626×10^{-34} is written as `6.626e-34`.

10 Listing of *MathLink* Library Functions

MLBytesToGet

`int MLBytesToGet(`*link*`, `*countptr*`)`
`MLINK `*link*`;`
`long *`*countptr*`;`

`MLBytesToGet(`*link*`, &`*count*`)` determines how many bytes remain to be read in the textual representation of the current element and writes this number to the `long` variable *count*.

See page 48. ■ See also: `MLGetNext`, `MLGetType`, `MLGetData`.

MLBytesToPut

`int MLBytesToPut(`*link*`, `*countptr*`)`
`MLINK `*link*`;`
`long *`*countptr*`;`

`MLBytesToPut(`*link*`, &`*count*`)` determines how many bytes remain to be written in the textual representation of the current element and writes this number to the `long` variable *count*.

See page 48. ■ See also: `MLPutNext`, `MLPutSize`, `MLPutData`.

MLCheckFunction

`int MLCheckFunction(`*link*`, `*string*`, `*countptr*`)`
`MLINK `*link*`;`
`char *`*string*`;`
`long *`*countptr*`;`

`MLCheckFunction(`*link*`, `*string*`, &`*count*`)` reads a function name and argument count from the specified link and compares the function name with *string*. If the name of the function matches *string*, `MLCheckFunction` returns nonzero and writes the argument count to the `long` variable *count*.

`MLCheckFunction` returns 0 if the function name currently waiting on the link does not match *string*, or if the current *MathLink* data element is not of the `MLTKFUNC` type. If an error has occurred, it can be identified by calling `MLError`. ■ See page 27. ■ See also: `MLError`.

MLClearError

`int MLClearError(`*link*`)`
`MLINK `*link*`;`

`MLClearError(`*link*`)` clears the *MathLink* error condition for *link*, if possible.

See page 44. ■ See also: `MLError`.

`MLClose`

`void MLClose(`*link*`)`
`MLINK` *link*`;`

`MLClose(`*link*`)` closes the *link*.

A program must close all links that it has opened before terminating. ■ When `MLClose` is called, any buffered outgoing data are flushed, that is, sent to the other partner (if its end of the link is still open). ■ See page 28. ■ See also: `MLOpen`.

`MLDisownIntegerList`

`void  MLDisownIntegerList(`*link*`,  `*ptr*`,  `*count*`)`
`MLINK` *link*`;`
`int *`*ptr*`;`
`long` *count*`;`

`MLDisownIntegerList(`*link*`,  `*ptr*`,  `*count*`)` allows *MathLink* to recycle memory it has previously allocated for temporary storage of a list of integers read by `MLGetIntegerList`. The values of *ptr* and *count* should correspond to values returned by `MLGetIntegerList`.

`MLDisownIntegerList` does not return any error codes. ■ Calling `MLDisownIntegerList` with a pointer that was not returned by `MLGetIntegerList` results in unpredictable behavior. ■ See page 23. ■ See also: `MLGetIntegerList`.

`MLDisownRealList`

`void MLDisownRealList(`*link*`,  `*ptr*`,  `*count*`)`
`MLINK` *link*`;`
`double *`*ptr*`;`
`long` *count*`;`

`MLDisownRealList(`*link*`,  `*ptr*`,  `*count*`)` allows *MathLink* to recycle memory it has previously allocated for temporary storage of a list of real numbers (of type `double`) read by `MLGetRealList`. The values of *ptr* and *count* should correspond to values returned by `MLGetRealList`.

`MLDisownRealList` does not return any error codes. ■ Calling `MLDisownRealList` with a pointer that was not returned by `MLGetRealList` results in unpredictable behavior. ■ See page 23. ■ See also: `MLGetRealList`.

`MLDisownString`

`void  MLDisownString(`*link*`,  `*ptr*`)`
`MLINK` *link*`;`
`char *`*ptr*`;`

`MLDisownString(`*link*`,  `*ptr*`)` allows *MathLink* to recycle memory it has previously allocated for temporary storage of a string read by `MLGetString`. The value of *ptr* should correspond to a value returned by `MLGetString`.

`MLDisownString` does not return any error codes. ■ Calling `MLDisownString` with a pointer that was not returned by *MathLink* from a previous `MLGetString` results in unpredictable behavior. ■ See page 22. ■ See also: `MLGetString`.

MLDisownSymbol

```
void  MLDisownSymbol(link, ptr)
MLINK link;
char *ptr;
```

`MLDisownSymbol(`*link*`, `*ptr*`)` allows *MathLink* to recycle memory it has previously allocated for temporary storage of a string read by `MLGetSymbol`. The value of *ptr* should correspond to a value returned by `MLGetSymbol`.

`MLDisownSymbol` does not return any error codes. ■ Calling `MLDisownSymbol` with a pointer that was not returned by *MathLink* from a previous `MLGetSymbol` has unpredictable results. ■ See pages 22 and 24. ■ See also: `MLGetSymbol`

MLEndPacket

```
int MLEndPacket(link)
MLINK link;
```

`MLEndPacket(`*link*`)` marks the end of an expression sent to *link*.

`MLEndPacket` should be called after each expression is put on the link; it is needed for *MathLink*'s internal mechanisms. ■ See page 27.

MLError

```
int MLError(link)
MLINK link;
```

`MLError(`*link*`)` returns an integer code identifying the most recent *MathLink* error that has been encountered on *link*.

If no error has occurred for this link, or if no error has occurred since `MLClearError` was called, `MLError` returns `MLEOK`, which is equal to 0. The *MathLink* error codes are shown in the following list.

`MLEOK`	no error has occurred (equal to 0)
`MLEDEAD`	an unrecoverable error has occurred; the other side may have exited
`MLEGBAD`	inconsistent data were encountered in the stream
`MLEGSEQ`	an `MLGet` function was called out of sequence
`MLEPBTK`	a bad data type was passed to `MLPutNext`
`MLEPSEQ`	an `MLPut` function was called out of sequence
`MLEPBIG`	bytes put by `MLPutData` exceeded number indicated by `MLPutSize`
`MLEOVFL`	machine integer overflow in `MLGetInteger`
`MLEMEM`	not enough space to allocate memory for a string
`MLEACCEPT`	failure to accept a connection
`MLECONNECT`	connection has not yet been established with partner
`MLECLOSED`	link closed by other side; you may still get undelivered data
`MLEPUTENDPACKET`	unexpected or missing call of `MLEndPacket`
`MLENEXTPACKET`	`MLNextPacket` called while the current expression has unread data
`MLEUNKNOWNPACKET`	`MLNextPacket` read in an unknown packet head
`MLEGETENDPACKET`	unexpected end of expression
`MLEABORT`	a put or get was aborted before affecting the link

■ See page 44. ■ See also: `MLClearError`, `MLErrorMessage`.

MLErrorMessage

`char *MLErrorMessage(`*link*`);`

`MLErrorMessage(`*link*`)` returns a text string describing the most recent *MathLink* error that has occurred on *link*.

See page 44. ■ See also: `MLError`.

MLFlush

`int MLFlush(`*link*`)`
`MLINK` *link*`;`

`MLFlush(`*link*`)` transmits immediately any data buffered for sending over the connection specified by *link*.

`MLFlush` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See page 46.

MLGetArgCount

`int MLPutArgCount(`*link*`, `*countptr*`)`
`MLINK` *link*`;`
`long *`*countptr*`;`

`MLGetArgCount(`*link*`, &`*count*`)` reads the argument count for an expression being read from *link* and writes this number to the `long` variable *count*.

This call can be used after an `MLGetNext` or `MLGetType` call returns the value `MLTKFUNC`. ■ The argument count should always be followed by the *MathLink* representation of the head of a *Mathematica* expression; the head is followed immediately by a sequence of parts (if there are any). The count value returned by `MLGetArgCount` tells a receiving program how many parts it should expect to find. ■ See pages 22 and 25.
■ See also: `MLGetNext`, `MLGetType`.

MLGetData

`int MLGetData(`*link*`, `*buff*`, `*max*`, `*countptr*`)`
`MLINK` *link*`;`
`char *`*buff*`;`
`long` *max*`;`
`long *`*countptr*`;`

`MLGetData(`*link*`, `*buff*`, `*max*`, &`*count*`)` gets a variable number of bytes from *link*, placing them in buffer *buff*. The argument *max* is the maximum number of characters that will be read. The number of bytes actually read is returned in the `long` variable *count*.

`MLGetData` returns nonzero if it is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See page 48. ■ See also: `MLError`.

MLGetDouble

```
int MLGetDouble(link, dptr)
MLINK link;
double *dptr;
```

`MLGetDouble(`*link*`, &`*dnum*`)` reads a real number from the *MathLink* connection specified by *link* and writes its C `double` representation to the variable *dnum*.

`MLGetDouble` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ▪ `MLGetDouble` performs the same function as `MLGetReal`. ▪ See page 25. ▪ See also: `MLGetFloat`, `MLGetLongDouble`, `MLGetReal`, `MLError`.

MLGetFloat

```
int MLGetFloat(link, fptr)
MLINK link;
float *fptr;
```

`MLGetFloat(`*link*`, &`*fnum*`)` reads a real number from the *MathLink* connection specified by *link* and writes its C `float` representation to the variable *fnum*.

`MLGetFloat` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ▪ See page 25. ▪ See also: `MLGetDouble`, `MLGetLongDouble`, `MLGetReal`, `MLError`.

MLGetFunction

```
int MLGetFunction(link, stringptr, countptr)
MLINK link;
char **stringptr;
long *countptr;
```

`MLGetFunction(`*link*`, &`*stringvar*`, &`*count*`)` reads a *Mathematica* function name and argument count from the *MathLink* connection specified by *link*. It stores the function name as a string and writes the string's address to *stringvar*. It writes the argument count to *count*.

When the calling program has finished copying or examining the function name, it should call `MLDisownSymbol` to allow the memory to be reused for other operations.

`MLGetFunction` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError` or a related function. ▪ The calling program may examine or copy the contents of the returned function name string, but it should not alter or free the string, because it resides in the link's internally managed memory pool. ▪ If no errors occur, `MLGetFunction(`*link*`, &`*string*`, &`*count*`);` has the same effect as `MLGetNext(`*link*`); MLGetArgCount(`*link*`, &`*count*`); MLGetSymbol(`*link*`, &`*string*`);` ▪ `MLGetFunction` does not advance the *MathLink* data stream if the current data element is not a composite expression element. ▪ See pages 22 and 24. ▪ See also: `MLDisownSymbol`, `MLError`.

`MLGetInteger`

`int MLGetInteger(`*link*`,` *iptr*`)`
`MLINK` *link*`;`
`int *`*iptr*`;`

`MLGetInteger(`*link*`, &`*inum*`)` reads an integer from the *MathLink* connection specified by *link* and writes its C `int` representation to the variable *inum*.

`MLGetInteger` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See page 22. ■ See also: `MLGetShortInteger`, `MLGetLongInteger`, `MLError`.

`MLGetIntegerList`

`int MLGetIntegerList(`*link*`,` *vectorptr*`,` *countptr*`)`
`MLINK` *link*`;`
`int **`*vectorptr*`;`
`long *`*countptr*`;`

`MLGetIntegerList(`*link*`, &`*vector*`, &`*count*`)` receives a list of integers from the *MathLink* connection specified by *link*, stores them as an array of `int`s, and writes a pointer to the received data to the `int *` variable *vector*. The length of the received list is written to the `long` variable *count*.

When the calling program has finished copying or examining the vector, it should call `MLDisownIntegerList` to make the memory available for reuse by the link. ■ The calling program should not alter the contents of the vector created by `MLGetIntegerList`, because the vector resides in the link's internally managed memory pool. ■ See page 23. ■ See also: `MLDisownIntegerList`.

`MLGetLongDouble`

`int MLGetLongDouble(`*link*`,` *ldptr*`)`
`MLINK` *link*`;`
`long double *`*ldptr*`;`

`MLGetLongDouble(`*link*`, &`*ldnum*`)` reads a real number from the *MathLink* connection specified by *link* and writes its C `long double` representation to the variable *ldnum*.

`MLGetLongDouble` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See page 25. ■ See also: `MLGetDouble`, `MLGetFloat`, `MLGetReal`, `MLError`.

`MLGetLongInteger`

`int MLGetLongInteger(`*link*`,` *lptr*`)`
`MLINK` *link*`;`
`long *`*lptr*`;`

`MLGetLongInteger(`*link*`, &`*lnum*`)` reads an integer from the *MathLink* connection specified by *link* and writes its C `long` representation to the variable *lnum*.

`MLGetLongInteger` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See page 25. ■ See also: `MLGetShortInteger`, `MLGetInteger`, `MLError`.

MLGetNext

`int MLGetNext(`*link*`)`
`MLINK` *link*`;`

`MLGetNext(`*link*`)` returns the type of the next element in the expression currently being read from *link*.

The "next" element always means an element that has not yet been examined by any *MathLink* get call. If a preceding element has been examined but not completely read, it will be discarded. To check the type of a partially read element without advancing to the following element, call `MLGetType`. ■ The possible data element types are

`MLTKSTR`	*Mathematica* string
`MLTKSYM`	*Mathematica* symbol
`MLTKINT`	integer
`MLTKREAL`	real number
`MLTKFUNC`	composite expression (having a head and zero or more arguments)
`MLTKERROR`	error getting type

■ `MLGetNext` returns `MLTKERROR` (== 0) if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See pages 22, 25, 26 and 48. ■ See also: `MLError`, `MLGetType`, `MLBytesToGet`, `MLGetData`.

MLGetReal

`int MLGetReal(`*link*`,` *rptr*`)`
`MLINK` *link*`;`
`double *`*rptr*`;`

`MLGetReal(`*link*`, &`*rnum*`)` reads a real number from the *MathLink* connection specified by *link* and writes its C `double` representation to the variable *rnum*.

`MLGetReal` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See pages 22 and 25. ■ See also: `MLGetDouble`, `MLGetFloat`, `MLGetLongDouble`, `MLError`.

MLGetRealList

`int MLGetRealList(`*link*`,` *vectorptr*`,` *countptr*`)`
`MLINK` *link*`;`
`double **`*vectorptr*`;`
`long *`*countptr*`;`

`MLGetRealList(`*link*`, &`*vector*`, &`*count*`)` receives a list of real numbers from *link*, stores them as an array of `doubles`, and writes a pointer to the received data to the `double *` variable *vector*. The length of the received list is written to the `long` variable *count*.

When the calling program has finished copying or examining the vector, it should call `MLDisownRealList` to make the memory available for reuse by the link. ■ The calling program should not alter the contents of the vector created by `MLGetRealList`, because the vector resides in the link's internally managed memory pool. ■ `MLGetRealList` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See page 23. ■ See also: `MLDisownRealList`, `MLGetIntegerList`.

MLGetShortInteger

`int MLGetShortInteger(`*link*`, `*sptr*`)`
`MLINK `*link*`;`
`short *`*sptr*`;`

`MLGetShortInteger(`*link*`, &`*snum*`)` reads an integer from the *MathLink* connection specified by *link* and writes its C `short` representation to the variable *snum*.

`MLGetShortInteger` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See page 25. ■ See also: `MLGetInteger`, `MLGetLongInteger`, `MLError`.

MLGetString

`int MLGetString(`*link*`, `*stringptr*`)`
`MLINK `*link*`;`
`char **`*stringptr*`;`

`MLGetString(`*link*`, &`*stringvar*`)` receives a *Mathematica* character string from *link*. `MLGetString` stores the string in the link's private memory area and writes a pointer to that string to the variable *stringvar*.

`MLGetString` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ When the calling program has finished copying or examining the string, it should call `MLDisownString` to allow the memory to be reused for other operations. ■ The calling program should not alter the contents of the received string, because it resides in the link's internally managed memory pool. ■ The following code fragment reads a string from the link, copies it to the program's data pool where it can be manipulated, and then informs *MathLink* that it is safe to free the string.

```
char *mathlink_string;
char my_string[50];
if (!MLGetString(thelink, &mathlink_string)) error_handler();
strncpy(my_string, mathlink_string, 50);
MLDisownString(thelink, mathlink_string);
```

■ See page 22. ■ See also: `MLDisownString`, `MLError`.

MLGetSymbol

`int MLGetSymbol(`*link*`, `*stringptr*`)`
`MLINK `*link*`;`
`char **`*stringptr*`;`

`MLGetSymbol(`*link*`, &`*stringvar*`)` receives a *Mathematica* symbol from *link*. `MLGetSymbol` stores the symbol as a string in the link's private memory area and writes a pointer to that string to the variable *stringvar*.

`MLGetSymbol` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ When the calling program has finished copying or examining the string, it should call `MLDisownString` to allow the memory to be reused for other operations. ■ The calling program should not alter the contents of the received string, because it resides in the link's internally managed memory pool. ■ A *Mathematica* symbol is a sequence of characters naming a certain object; it is distinct from a *Mathematica* string. Special symbols like + or * are not carried over *MathLink* as such, but are expressed as functions with alphabetical names like `Plus` and `Times`. ■ See page 22. ■ See also: `MLGetString`, `MLDisownString`, `MLError`.

MLGetType

```
int MLGetType(link)
MLINK link;
```

`MLGetType(`*link*`)` returns the current element type in the *MathLink* data stream.

The possible data element types are

`MLTKSTR`	*Mathematica* string
`MLTKSYM`	*Mathematica* symbol
`MLTKINT`	integer
`MLTKREAL`	real number
`MLTKFUNC`	composite expression (having a head and zero or more arguments)
`MLTKERROR`	error getting type

■ `MLGetType` returns `MLTKERROR` (== 0) if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ The difference between `MLGetType` and `MLGetNext` is that `MLGetNext` always looks ahead to a fresh data element, that is, one that has not been examined by any get call. `MLGetType` will stay at the data element that was last accessed if it was not read completely. Therefore `MLGetType` can be called more than once for the same data element. ■ See pages 22, 25, 26 and 48. ■ See also: `MLError`, `MLGetNext`.

MLMain

```
int MLMain(argc, argv)
int argc;
char *argv[ ];
```

`MLMain(`*argc*`,` *argv*`)` sets up communication between an installable program and *Mathematica*. It opens a *MathLink* connection using the link parameters specified in *argv*, if any, and goes into a loop to await the arrival of call packets from *Mathematica*.

`MLMain` is not a *MathLink* library function; it is generated by the utility program `mprep` to implement the external function calls specified in a *MathLink* template file. It is linked into an installable program when you run `mcc` or when you supply the source file generated by `mprep` to your C compiler along with your external function module. ■ `MLMain` calls `MLOpen` with arguments *argc* and *argv*. ■ See pages 3 and 37. ■ See also: `MLOpen`.

MLNewPacket

```
int MLNewPacket(link)
MLINK link;
```

`MLNewPacket(`*link*`)` discards the remaining data in the expression currently being read from *link*.

See page 27. ■ See also: `MLNextPacket`.

MLNextPacket

```
int MLNextPacket(link)
MLINK link;
```

`MLNextPacket(`*link*`)` reads the head of the packet expression presently waiting to be read from *link* and returns an integer code corresponding to the packet type.

The packet heads recognized by `MLNextPacket` and the corresponding packet type codes are as follows.

Packet head	*Packet type*	*Description*
`InputPacket`	`INPUTPKT`	Request for input, as generated by `Input` function
`TextPacket`	`TEXTPKT`	Text output from *Mathematica*, as produced by `Print`
`ReturnPacket`	`RETURNPKT`	Result of a calculation
`ReturnTextPacket`	`RETURNTEXTPKT`	Formatted text representation of a result
`MessagePacket`	`MESSAGEPKT`	*Mathematica* message identifier (*symbol*`::`*name*)
`CallPacket`	`CALLPKT`	Request to invoke an external function
`InputNamePacket`	`INPUTNAMEPKT`	Name of next input cell in a Notebook (*e.g.,* `In[3]:=`)
`OutputNamePacket`	`OUTPUTNAMEPKT`	Name of next output cell in a Notebook (*e.g.,* `Out[3]=`)
`SyntaxPacket`	`SYNTAXPKT`	Position at which a syntax error was detected
`MenuPacket`	`MENUPKT`	“Menu number” and prompt string
`DisplayPacket`	`DISPLAYPKT`	String of PostScript sent as part of a graphic
`DisplayEndPacket`	`DISPLAYENDPKT`	Last string of PostScript code in a graphic

■ If the head of the current expression does not match any of these packet types, `MLNextPacket` returns 0.
■ See page 27. ■ See also: `MLNewPacket`.

MLOpen

```
MLINK MLOpen(argc, argv)
int argc;
char *argv[ ];
```

`MLOpen(`*argc*`, `*argv*`)` opens a *MathLink* connection, returning a pointer to a link data structure. The arguments are an argument count *argc* and an array of character strings *argv*.

`MLOpen` scans the argument list for the following parameters.

`-linkmode` *mode*	Launch, ParentConnect, Listen, or Connect
`-linkprotocol` *protocol*	data transport mechanism (*e.g.,* “TCP”, “PPC ”, “pipes”)
`-linkname` *name*	name of communication partner (a command line or port name)
`-linkhost` *hostname*	machine where partner is to be found

■ `MLOpen` is not sensitive to upper and lower case in the parameter list. ■ `MLOpen` returns a value of type `MLINK`, which is a pointer to the *MathLink* structure for the connection, or `NULL` if there is an error. ■ `MLOpen` is called by `MLMain` in installable programs that are created by the utilities `mcc` and `mprep`. ■ Note that `MLOpen` does not modify the contents of *argc* or *argv*, so if the calling function checks *argc*/*argv* for other arguments, it should be able to deal with the presence of `MLOpen` parameters, usually by ignoring them. ■ See pages 21 and 37. ■ See also: `MLMain`.

MLPutArgCount

```
int MLPutArgCount(link, count)
MLINK link;
long count;
```

`MLPutArgCount(link, count)` gives the argument count for a composite expression being sent over *link*.

An `MLPutArgCount` call should follow the call `MLPutNext(link, MLTKFUNC)`. ■ See pages 22 and 24. ■ See also: `MLPutNext`.

MLPutData

```
int MLPutData(link, buff, count)
MLINK link;
char *buff;
int count;
```

`MLPutData(link, buff, count)` writes *count* bytes from buffer *buff* to *link*.

`MLPutData` returns nonzero if it is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ `MLPutData` is part of *MathLink*'s textual interface, by which an arbitrary data element can be represented as a text sequence. ■ One or more `MLPutData` calls may follow calls of `MLPutNext` and `MLPutSize`. ■ See page 48. ■ See also: `MLPutNext`, `MLPutSize`.

MLPutDouble

```
int MLPutDouble(link, dnum)
MLINK link;
double dnum;
```

`MLPutDouble(link, dnum)` writes the real number represented by *dnum* to the *MathLink* connection specified by *link*.

`MLPutDouble` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ `MLPutDouble` performs the same function as `MLPutReal`. ■ See page 25. ■ See also: `MLPutReal`, `MLPutFloat`, `MLPutLongDouble`, `MLError`.

MLPutFloat

```
int MLPutFloat(link, fnum)
MLINK link;
double fnum;
```

`MLPutFloat(link, fnum)` writes the real number represented by *fnum* to the *MathLink* connection specified by *link*.

`MLPutFloat` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See page 25. ■ See also: `MLPutReal`, `MLPutDouble`, `MLPutLongDouble`, `MLError`.

MLPutFunction

```
int MLPutFunction(link, string, count)
MLINK link;
char *string;
long count;
```

`MLPutFunction(`*link*`, `*string*`, `*count*`)` places the function name represented by *string* and the argument count *count* on the *MathLink* connection specified by *link*.

`MLPutFunction` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by by calling `MLError`. ■ See pages 22 and 24. ■ See also: `MLError`.

MLPutInteger

```
int MLPutInteger(link, inum)
MLINK link;
int inum;
```

`MLPutInteger(`*link*`, `*inum*`)` writes the integer represented by *inum* to the *MathLink* connection specified by *link*.

`MLPutInteger` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it may be identified by calling `MLError`. ■ See page 22. ■ See also: `MLPutShortInteger`, `MLPutLongInteger`, `MLError`.

MLPutIntegerList

```
int MLPutIntegerList(link, vector, count)
MLINK link;
int *vector;
long count;
```

`MLPutIntegerList(`*link*`, `*vector*`, `*count*`)` writes a list of *count* integers from *vector* to *link*.

`MLPutIntegerList` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See pages 8 and 23. ■ See also: `MLError`.

MLPutLongInteger

```
int MLPutLongInteger(link, lnum)
MLINK link;
long lnum;
```

`MLPutLongInteger(`*link*`, `*lnum*`)` writes the integer represented by *lnum* to the *MathLink* connection specified by *link*.

`MLPutLongInteger` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it may be identified by calling `MLError`. ■ See page 25. ■ See also: `MLPutInteger`, `MLPutShortInteger`, `MLError`.

MLPutLongDouble

`int MLPutLongDouble(`*link*`, `*ldnum*`)`
`MLINK` *link*`;`
`long double` *ldnum*`;`

`MLPutLongDouble(`*link*`, `*ldnum*`)` writes the real number represented by *ldnum* to the *MathLink* connection specified by *link*.

`MLPutLongDouble` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See page 25. ■ See also: `MLPutReal`, `MLPutFloat`, `MLPutDouble`, `MLError`.

MLPutMessage

`int MLPutMessage(`*link*`, `*messagecode*`)`
`MLINK` *link*`;`
`int` *messagecode*`;`

`MLPutMessage(`*link*`, MLInterruptMessage)` sends a request to *Mathematica* to interrupt the current calculation.

The nature of the "interrupt request" is a *MathLink* implementation detail which might vary from version to version. ■ `MLInterruptMessage` is an integer constant defined in the *MathLink* header file `mathlink.h`. Other message codes are reserved for *MathLink* internal use or future development. ■ See page 47. ■ See also: *Mathematica* built-in function `LinkInterrupt`.

MLPutNext

`int MLPutNext(`*link*`, `*dtype*`)`
`MLINK` *link*`;`
`int` *dtype*`;`

`MLPutNext(`*link*`, `*dtype*`)` identifies the type of data element that is to be sent over *link*.

The data types are identified by integer codes. The possible values for *dtype* are as follows.

`MLTKSTR`	*Mathematica* string
`MLTKSYM`	*Mathematica* symbol
`MLTKINT`	integer
`MLTKREAL`	real number
`MLTKFUNC`	composite expression (having a head and zero or more arguments)

■ A call of `MLPutNext` should be followed by calls of `MLPutSize` and `MLPutData` for the first four data types, or by a call of `MLPutArgCount` for the `MLTKFUNC` type. ■ Most data elements can be sent without calling `MLPutNext`, by using the specific put functions `MLPutInteger`, `MLPutReal`, etc. However, `MLPutNext` must be used to put expressions whose heads are not mere symbols (*e.g.*, `Derivative[1][f]`). ■ `MLPutNext` is used along with `MLPutSize` and `MLPutData` to put data in textual form. ■ `MLPutNext` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See pages 22, 24 and 48. ■ See also: `MLPutInteger`, `MLPutReal`, `MLPutSymbol`, `MLPutString`, `MLPutFunction`, `MLPutArgCount`, `MLPutSize`, `MLPutData`.

MLPutReal

```
int MLPutReal(link, rnum)
MLINK link;
double rnum
```

`MLPutReal(`*link*`, `*rnum*`)` writes the real number represented by *rnum* to the *MathLink* connection specified by *link*.

`MLPutReal` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See page 22. ■ See also: `MLPutDouble`, `MLPutFloat`, `MLPutLongDouble`, `MLError`.

MLPutRealList

```
int MLPutRealList(link, vector, count)
MLINK link;
double *vector;
long count;
```

`MLPutRealList(`*link*`, `*vector*`, `*count*`)` writes a list of *count* real numbers from *vector* to *link*.

`MLPutRealList` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ See page 23. ■ See also: `MLError`.

MLPutShortInteger

```
int MLPutShortInteger(link, snum)
MLINK link;
int snum;
```

`MLPutShortInteger(`*link*`, `*snum*`)` writes the integer represented by *snum* to the *MathLink* connection specified by *link*.

`MLPutShortInteger` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it may be identified by calling `MLError`. ■ See page 25. ■ See also: `MLPutInteger`, `MLPutLongInteger`, `MLError`.

MLPutSize

```
int MLPutSize(link, count)
MLINK link;
long count;
```

`MLPutSize(`*link*`, `*count*`)` defines the textual size in bytes of a data element being written to *link*.

To put a data element in textual form, your program should first call `MLPutNext` to give the type of the data element, then `MLPutSize` to give its size in bytes, then `MLPutData` to give the value of the data element in text form. ■ `MLPutSize` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it may be identified by calling `MLError`. ■ See page 48. ■ See also: `MLPutNext`, `MLPutData`, `MLBytesToPut`.

MLPutString

```
int MLPutString(link, string)
MLINK link;
char  *string;
```

`MLPutString(`*link*`, `*string*`)` writes the *Mathematica* character string represented by *string* to the *MathLink* connection specified by *link*.

`MLPutString` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ The string is assumed to reside in memory allocated by the calling program; it will not be altered or freed by `MLPutString`. ■ See page 22. ■ See also: `MLError`.

MLPutSymbol

```
int MLPutSymbol(link, string)
MLINK link;
char *string;
```

`MLPutSymbol(`*link*`, `*string*`)` writes the *Mathematica* symbol represented by *string* to the *MathLink* connection specified by *link*.

`MLPutSymbol` returns nonzero if the operation is successful, or 0 if an error has occurred. If an error has occurred, it can be identified by calling `MLError`. ■ The string is assumed to reside in memory allocated by the calling program; it will not be altered or freed by `MLPutSymbol`. ■ A *Mathematica* symbol is a sequence of characters naming a certain object; it is distinct from a *Mathematica* string. Special symbols like + or * are not carried over *MathLink* as such, but are expressed as functions with alphabetical names (like `Plus` and `Times`). ■ See page 22. ■ See also: `MLError`.

MLReady

```
int MLReady(link)
MLINK link;
```

`MLReady(`*link*`)` returns nonzero if there are data ready to be read from the connection specified by *link*, and returns 0 if not.

See page 46. ■ See also: *Mathematica* built-in function `LinkReadyQ`.

11 Listing of *Mathematica*'s Built-in *MathLink* Functions

`Enter`

`Enter["`*string*`"]`, in *MathLink* mode only, processes the input `"`*string*`"` and sends results back to the parent program.

Mathematica responds to `Enter["`*string*`"]` by sending a sequence of packets back to the calling program or front end. ■ The returned packets generally include an `OutputNamePacket`, followed by a `ReturnTextPacket` or `SyntaxPacket` expression, and finally an `InputNamePacket` to set up for the next input. ■ See page 31.

`Install`

`Install["`*name*`"]` launches or connects to the external program specified by `"`*name*`"` and installs *Mathematica* definitions to call functions in the external program.

The *Mathematica* definitions set up by `Install` are typically specified in the *MathLink* template file that is used to create the external program. ■ The `"`*name*`"` argument can be the name of a program file or the name of a port through which *MathLink* will communicate with an external program. ■ `Install` accepts the options `LinkMode`, `LinkProtocol`, and `LinkHost`, which have the meanings given under `LinkOpen`. ■ `Install` returns a `LinkObject` expression representing the new link. ■ `LinkPatterns[`*link*`]` gives a list of the patterns defined when the specified link was set up. ■ You can remove these definitions and close the link to the external program by calling `Uninstall[`*link*`]`. ■ If you call `Install["`*command*`"]` multiple times with the same *command*, the later calls will overwrite definitions set up by earlier ones, unless `ThisLink` is included in the template argument list, or the definitions depend on the values of global variables that have changed. ■ See pages 3, 7 and 40. ■ See also: `LinkOpen`, `Uninstall`, `ThisLink`.

`LinkClose`

`LinkClose[`*link*`]` closes a previously opened *MathLink* connection.

See page 35. ■ See also: `LinkOpen`.

`LinkError`

`LinkError[`*link*`]` returns error information for *link* in the form `{`*errorNumber*`, `*errorExplanation*`}`.

See page 36. ■ See also: *MathLink* library functions `MLError`, `MLErrorMessage`.

`LinkHost`

`LinkHost` is an option to `LinkOpen` or `Install`. It gives the name of the host computer on which the other partner to the link is to be found.

If `LinkHost` is not specified, `LinkOpen` assumes that the partner is located on the same computer as *Mathematica*. ■ See page 37. ■ See also: `LinkOpen`, `Install`, `LinkMode`, `LinkProtocol`.

`LinkInterrupt`

`LinkInterrupt[`*link*`]` interrupts a calculation in an installed external program connected over *link*.

`LinkInterrupt` can be executed while *Mathematica* is waiting for an external program to finish a calculation by pressing the interrupt key (usually Control-C or Command-.) and then choosing "abort" from the interrupt menu. ■ `LinkInterrupt[`*link*`]` sets `MLAbort` to 1 in an installed external program. ■ `LinkInterrupt` can also be used from one *Mathematica* process to interrupt a second *Mathematica* process that is running in *MathLink* mode with `$ParentLink` pointing to the first process. ■ See pages 36 and 46.

`LinkMode`

`LinkMode` is an option to `LinkOpen` or `Install` that sets the mode of the connection.

The possible modes are as follows.

`Launch`	start up a child process and link to it
`ParentConnect`	establish a link to the parent process
`Listen`	listen on a port and wait for other process to link up
`Connect`	connect to a listening port established by another process

■ Launch and ParentConnect modes are used for the two sides in a parent-child connection. Listen and Connect modes are used for the two sides in a peer-to-peer connection. ■ See page 37. ■ See also: `LinkOpen`, `Install`, `LinkProtocol`, `LinkHost`.

`LinkObject`

`LinkObject[`*name*`, `*serialno*`, `*channo*`]` represents a link to an external process started using `LinkOpen` or `Install`.

A `LinkObject` expression is generated by a successful `LinkOpen` or `Install` command. It must be supplied as the first argument to the *MathLink* functions that are used to communicate over the link. ■ For convenience, assign the link object returned by `LinkOpen` or `Install` to a variable to be used as the link argument. ■ The *serialno* argument specifies which invocation of `LinkOpen` or `Install` created this particular link. ■ *channo* is a "channel number"; *MathLink* uses it internally. ■ See pages 7 and 34. ■ See also: `LinkOpen`, `Install`.

LinkOpen

`LinkOpen["`*name*`"]` returns a *MathLink* connection to the external program specified by *name*.

The `"`*name*`"` argument can be the name of a program file or the name of a port through which *MathLink* will communicate with an external program. ■ `LinkOpen` accepts the following options.

Option	*Default*	*Description*
`LinkHost`	`""`	machine where partner is to be found
`LinkMode`	`Automatic`	Launch, ParentConnect, Listen, or Connect
`LinkProtocol`	`Automatic`	data transport mechanism (*e.g.*, "TCP", "PPC ", "pipes")

■ `LinkOpen` takes its argument and the given options and passes them as connection parameter arguments to the *MathLink* function `MLOpen`. ■ `LinkOpen` returns a *Mathematica* link object, which has the form `LinkObject["`*name*`",` *serialno*`,` *channo*`]`. The *serialno* argument specifies which invocation of `LinkOpen` the link is associated with. ■ The link object returned by `LinkOpen` should be stored in a variable; you will have to supply it as the first argument in subsequent operations on the link. ■ You can use `LinkOpen` repeatedly to run several copies of the same external process, which can be dealt with separately by reference to the distinct link objects associated with them. ■ *channo* is a "channel number"; *MathLink* uses it internally. ■ See pages 34 and 39. ■ See also: `LinkObject`, `LinkClose`, `Install`; *MathLink* library function `MLOpen`.

LinkPatterns

`LinkPatterns[`*link*`]` gives a list of the patterns defined in the installed program that is accessed via *link*.

See pages 10 and 15. ■ See also: `Install`.

LinkProtocol

`LinkProtocol` is an option to `LinkOpen` or `Install` that tells *MathLink* what data transport mechanism the connection should use.

The choices available for `LinkProtocol` depend on the type of system you are using and what connection mode you use. ■ See also: `LinkOpen`, `Install`, `LinkMode`, `LinkHost`.

LinkRead

`LinkRead[`*link*`]` reads an expression via *MathLink* from *link*.

See page 35. ■ See also: `LinkReadHeld`, `LinkReadyQ`.

LinkReadHeld

`LinkReadHeld[`*link*`]` reads an expression via *MathLink* from *link* and returns it wrapped in `Hold`.

See page 35. ■ See also: `LinkRead`, `LinkReadyQ`.

LinkReadyQ

`LinkReadyQ[`*link*`]` returns `True` if *link* has incoming data ready.

See page 36. ■ See also:`LinkRead`, `LinkReadHeld`.

`Links`

`Links[ ]` returns a list of all active *MathLink* links.

See page 36. ■ See also: `LinkOpen`, `LinkClose`.

`LinkWrite`

`LinkWrite[`*link*`, `*expr*`]` writes *expr* via *MathLink* to *link*.

See page 35. ■ See also: `LinkWriteHeld`.

`LinkWriteHeld`

`LinkWriteHeld[`*link*`, Hold[`*expr*`]]` writes *expr* (without the `Hold`) via *MathLink* to *link*.

If the second argument to `LinkWriteHeld` is not an expression wrapped in `Hold`, and does not evaluate to an expression wrapped in `Hold`, `LinkWriteHeld` gives an error message and no expression is sent. ■ See page 35. ■ See also: `LinkWrite`.

`ThisLink`

In a *MathLink* template entry's `:Pattern:` line, `ThisLink` stands for the link object that is created when the external program is installed. It allows you to install and use separate copies of a single external program.

If the pattern specified for an external function includes `ThisLink`, you must include the appropriate link object as an argument in place of `ThisLink` to access the function. ■ If you have installed two or more instances of an external program, the use of a link object argument allows you to tell *Mathematica* which copy of the function to use. ■ While `Install` is linking *Mathematica* to an external program, `ThisLink` is temporarily set to the `LinkObject` expression that has been assigned to that particular instance of the external program. ■ See page 13. ■ See also: `Install`, `LinkObject`.

`Uninstall`

`Uninstall[`*link*`]` closes the link to an installed external program, and removes *Mathematica* definitions set up for that link by `Install`.

The argument of `Uninstall` is the link object representing a *MathLink* connection to an external program, as returned by `Install`. ■ `Uninstall[`*link*`]` calls `Unset` to remove definitions for the patterns listed in `LinkPatterns[`*link*`]`. ■ See page 3. ■ See also: `Install`, `LinkObject`.

`$LinkSupported`

`$LinkSupported` is `True` if *MathLink* can be used in the version of *Mathematica* you are running, and is `False` otherwise.

`$LinkSupported` is `True` in Unix-based and Macintosh versions, and for most systems that support multitasking. ■ See page 36.

`$ParentLink`

`$ParentLink` is the link object through which *Mathematica* communicates with a front end when it is called in *MathLink* mode.

`$ParentLink` is assigned implicitly when *Mathematica* is started with the `-mathlink` command-line option. It can also be assigned explicitly. Assigning `$ParentLink` to `Null` resets to terminal mode. ■ See pages 31 and 42.

Appendix: Using *MathLink* with Other Programming Languages

In order to call *MathLink* library functions from a language other than C, you need to understand two concepts. The first concept, called *data representation*, is the way a type in a language is represented at the machine level. The second concept is the language-specific *calling conventions*, which dictate how a called function manipulates its arguments.

Both the data representations and the calling conventions of a language are discussed in the compiler's documentation. Sometimes this information is provided in an appendix to the reference manual. Other times, there is a separate user's guide that contains a section providing this information.

The supplied *MathLink* libraries are always compiled with the native C compiler on Unix workstations. For Macintosh systems, *MathLink* libraries have been compiled for the Think C and MPW environments. In order to allow *MathLink* libraries to be called from a language other than C, you can use one of two solutions, depending on the situation.

- If the target language contains data representations and calling conventions that closely match those of C, it might be possible to call the *MathLink* library directly from your program.
- If the target language is not able to use the C compiler's calling conventions, or if the data representation is very different from C, you will need to write a series of interlude functions in either C or assembly language that allow the target language to call the *MathLink* functions.

Data representations vary between languages; for example, a string in C is commonly a null-terminated array of characters. In other languages, a string might be a length and an array of characters, which need not be null-terminated.

Some computers have multiple real number formats. For example, a Macintosh supports both an 80-bit and a 64-bit real. The target language might use a different real number format than C. Virtually all systems provide conversion functions when this is the case. Additionally, the target language may provide a method to declare all machine level types of real numbers.

MathLink makes extensive use of pointers in its operation. Not all languages support pointers in a manner as flexible as does C, but in many languages where this is nonstandard, such as Fortran, vendors often provide a pointer syntax for compatibility with C.

Calling conventions differ widely between compiler vendors. Traditionally, C compilers use the stack for this purpose. Fortran compilers do not always use the stack for compatibility with the Fortran 66 standard, or they might use the stack to pass a pointer into static storage.

Whatever the situation, you will need to make extensive use of the target compiler's documentation, as well as the documentation of your native C compiler.

Table of *MathLink* Functions

	MathLink library	*Mathematica*
Opening a link	`MLOpen`	`LinkOpen`

Installed function interface		`Install`
		`Uninstall`
		`LinkPatterns`

	`MLMain` (created by `mprep`) `MLEvaluate` (created by `mprep`)	
Closing a link	`MLClose`	`LinkClose`
Writing data to link		`LinkWrite` `LinkWriteHeld`

	`MLPutInteger`
	`MLPutShortInteger`
	`MLPutLongInteger`
	`MLPutReal`
	`MLPutDouble`
	`MLPutLongDouble`
	`MLPutFloat`
	`MLPutString`
	`MLPutSymbol`
	`MLPutFunction`
	`MLPutIntegerList`
	`MLPutRealList`	

	`MLPutNext`
	`MLPutArgCount`
	`MLPutSize`
	`MLBytesToPut`
	`MLPutData`	

	`MLEndPacket`	
Reading data from link		`LinkRead` `LinkReadHeld`

Reading data from link, *continued*	`MLGetInteger`
	`MLGetShortInteger`
	`MLGetLongInteger`
	`MLGetReal`
	`MLGetDouble`
	`MLGetLongDouble`
	`MLGetFloat`
	`MLGetString`
	`MLGetSymbol`
	`MLGetFunction`
	`MLGetRealList`
	`MLGetIntegerList`
	`MLCheckFunction`
	`MLDisownIntegerList`
	`MLDisownRealList`
	`MLDisownString`
	`MLDisownSymbol`
	
	`MLGetType`
	`MLGetNext`
	`MLGetArgCount`
	`MLBytesToGet`
	`MLGetData`
	
	`MLNextPacket`
	
	`MLNewPacket`	

MathLink buffer status	`MLReady` `MLFlush`	`LinkReadyQ`

Error condition	`MLError`
	`MLErrorMessage`
	`MLClearError`
		`LinkError`

Urgent data	`MLPutMessage`	`LinkInterrupt`

Other functions and variables		`Links`
		`$ParentLink`
		`Enter`
		`LinkObject`
		`ThisLink`
		`$LinkSupported`

Index