Wolfram *Mathematica*® Tutorial Collection

MATHLINK® DEVELOPMENT IN C

For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Contents

MathLink Development in C (Windows)

This document describes how to compile and run *MathLink* programs written in the C language on computers running a Microsoft Windows operating system. ("*MathLink* and External Program Communication" describes how to write *MathLink* programs in both the *Mathematica* language and the C language.) This document also describes how *MathLink* is implemented for Windows.

This document does not teach you, in general, how to use your compiler and other development tools, nor does it teach you how to program in C or generally how to use a Windows-based computer. It is assumed that you have at least worked through the tutorial introduction that came with your set of tools. If you have any trouble building or running your *MathLink* programs, see the "Troubleshooting" section at the end of this document.

Most of what is described in this document is Windows specific. To learn how to compile and run *MathLink* programs for another platform, see the Developer Guide for that platform.

Overview

MathLink is implemented as a collection of dynamically linked, shared libraries. When a *MathLink* program is launched, the main shared library is found in a standard place on disk, loaded by the operating system, and bound to the client *MathLink* program so that calls to *MathLink* functions are directed to code within the shared library. On a computer running a 32-bit Windows operating system (2000, XP, Vista), the main shared library is called ml32i3.dll and is typically placed in the Windows system directory. Any additional shared libraries (such as mishm32.mlp, mltcpip32.mlp, mltcp32.mlp and mlmap32.mlp) are found by ml32i3.dll by reading their locations from the configuration file mathlink.ini. If mathlink.ini is not present, the additional shared libraries are found by an algorithm internal to ml32i3.dll. This configuration file is typically located in the Windows directory.

On Windows 64 platforms (XP 64, Server 2003, Vista 64), the main shared library for 64-bit *MathLink* programs is ml64i3.dll, and the additional shared libraries are mlshm64.mlp, mltcpip64.mlp, and mltcp64.mlp.

An import library (a .lib file) is used when a *MathLink* program is built. This import library is added to a project file or included in the link command in a makefile. (This import library contains no code; it simply exports the same function names as the shared library in order to satisfy the linker.)

The C interface to *MathLink* is specified in the header file mathlink.h. This file is included in any C or C++ source files that call functions in the *MathLink* shared library.

The *MathLink* shared library ml32i3.dll or ml64i3.dll is needed at the time a *MathLink* program is run and needs to be placed where the Windows operating system can find it. The import library and header file are needed at the time a *MathLink* program is built and need to be placed where your compiler and linker can find them. The installation and use of these components and others are described in more detail below.

Supported Development Platforms

As a shared library, *MathLink* can be used with any development environment that adheres to the standard calling conventions and binary interface for DLLs specified by Microsoft. However, this document only describes how to build C programs that use *MathLink* using Microsoft C++ development environment.

Each development environment supports the notion of a "project document" whereby source files and options for the compiler, linker, and debugger are managed using a project window, a collection of dialog boxes, and other interactive interface elements. In addition to this integrated development environment (IDE), all the vendors supply tools that can be run from the command prompt or invoked by a make utility.

Installing the *MathLink* Components

This section describes how to install the components from the *MathLink* Developer Kit so that you can build and run *MathLink* programs. The *MathLink* Developer Kit is copied to your hard disk when you install *Mathematica.*

The following instructions assume that the *MathLink* Developer Kit for Windows have been installed in the location C:\Program Files\Wolfram Research\Mathematica\6.0\SystemFiles\

Links\MathLink\DeveloperKit\ by the *Mathematica* installer. For Windows there are normally two Developer Kits installed, one for 32-bit Windows platforms, in a folder named Windows, and one for Windows 64 platforms, in a folder named Windows-x86-64. If you have received a Developer Kit as a component separate from *Mathematica*, copy the folder Windows or Windows-x86-64 to your hard drive, for example to C:\MathLink. In this case, you will need to modify the following installation instructions to reflect the location of your Developer Kit.

Recommended Installation

SystemAdditions Installation for All Compilers

If your Developer Kit was installed as part of *Mathematica*, then the runtime *MathLink* components have already been installed into your Windows system directory by the *Mathematica* installer. However, as a *MathLink* developer, you should be aware of how to properly install these components.

CompilerAdditions Installation for Microsoft Compilers

Note that the following instructions assume that you have installed Microsoft's Visual Studio 2005 in the directory C:\Program Files\Microsoft Visual Studio 8\VC. This is the default directory for V8.0 of the Developer Studio. For V7.0, the default directory is C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7. For Windows 64 systems you will find the default Visual Studio installation in the C:\Program Files (x86)\Microsoft Visual Studio 8\VC.

1. Select the "Lib", "Include", and "Bin" directories within the "MLDev32" (on Windows 64 systems you will use the "Windows-x86-64\CompilerAdditions\MLDev64" folder) folder within the CompilerAdditions folder. You can open a window containing these folders by running the following command:

   ```
   explorer "C:\Program Files\Wolfram
   Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows\Co
   mpilerAdditions\MLDev32"
   ```

   The word `explorer` is optional but the quotation marks are not.

2. While holding down the Ctrl key, drag and drop these three folders into your C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK folder (C:\Program Files (x86)\Microsoft Visual Studio 8\VC\PlatformSDK on Windows 64 systems).

3. Click **Yes** or **Yes to All** in any confirmation boxes that appear.

   This will copy the files from these subdirectories of the "MLDev32" directory into the appropriate subdirectories of your "PlatformSDK" directory.

On Windows 64 systems the contents of the "C:\Program Files (x86)\Wolfram Research\ Mathematica \ 6.0 \ SystemFiles \ Links \ MathLink \ DeveloperKit \ Windows-x86-64 \ CompilerAdditions\MLDev64\lib" directory will need to go into the C:\Program Files (x86)\ Microsoft Visual Studio 8\VC\PlatformSDK\Lib\AMD64 folder.

Elements of the MathLink Developer Kit

MathLink is a layered software system consisting of components that are independent of the operating system and components that make direct use of the communication mechanisms provided by the operating system.

The primary role of the system-independent component is to encode expressions as a string of bytes. This software is implemented in the shared library called ml32i3.dll (for 32-bit programs) and in ml64i3.dll (for 64-bit programs). The role of the OS-specific component is to transfer the bytes from one program to another. There may be several implementations of this service. Each implementation is called a *MathLink* device and is implemented in a shared library with the .mlp extension.

MathLink Shared Libraries and Header Files

The following is a description of each file or directory in the *MathLink* Developer Kit (MLDK) for Windows. See the section "Recommended Installation" to install these components.

The *MathLink* Developer Kit for Windows 64 platform has a similar set of files, with the main difference appearing in the file names of the components. Where 32-bit Windows components contain a 32 in the component file name, 64 bit versions contain a 64 in the component file name.

SystemAdditions Directory

The files in the SystemAdditions directory are shared libraries which implement *MathLink*, and are usually installed into a place where the system can find them whenever *MathLink* programs are executed. (See the section "Recommended Installation" to install these components.) The six shared libraries needed by most *MathLink* programs are at the top level of this directory. These seven files for 32-bit Windows are ml32i3.dll, ml32i2.dll, ml32i1.dll, mlshm32.mlp, mltcpip32.mlp, mlmap32.mlp, and mltcp32.mlp. The five files for Windows 64 are ml64i3.dll, ml64i2.dll, mlshm64.mlp, mltcpip64, and mltcp64.mlp.

ml32i3.dll/ml64i3.dll

This is the shared library used by 32-bit *MathLink* Windows (64-bit *MathLink* Windows programs use ml64i3.dll) programs that implements Interface 3 of the *MathLink* application programming interface (API). It should be placed where the operating system's loader will find it when your 32-bit *MathLink* program is run. (It is not needed to build 32-bit *MathLink* programs—only to run them.) You could place this file next to your built *MathLink* program, or in the Windows system directory, or someplace on your PATH. See the following discussion for more information and alternative installation options. The *Mathematica* installer places this file in your Windows system directory and (for the convenience of installers and uninstallers) adds a reference to it in the system registry under `[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs]`.

ml32i2.dll/ml64i2.dll
ml32i1.dll

These shared libraries are similar to "ML32I3.DLL" but implements interface 1 and interface 2 of the *MathLink* API instead of interface 3. Like "ML32I3.DLL", it is used by 32-bit V3 *MathLink* programs. There is no shared library for the interface 1 implementation for Windows 64.

The *MathLink* Devices

The following shared libraries appropriate for your operating system should be placed where the operating system's loader will find them when your *MathLink* program is run. (They are not needed to build *MathLink* programs—only to run them.) Typically they are placed next to the *MathLink* libraries (any of the above .DLL files) in the Windows system directory.

All the *MathLink* devices follow a naming scheme that describes what protocol they implement and what types of programs may use them.

All devices and libraries in the SystemAdditions folder should be copied to the Windows system directory. On some 32-bit versions of Microsoft Windows, the Windows system directory is C:\WINDOWS\SYSTEM, whereas on others versions it is C:\WINNT\SYSTEM32 or C:\WINDOWS\SYSTEM32.

On 64-bit systems the device files will need to be copied to the correct Windows system folder depending on whether the device libraries are 32-bit or 64-bit libraries. The 32-bit libraries are located in the *Mathematica* layout at C:\Program Files (x86)\Wolfram Research\ Mathematica\ 6.0 \ SystemFiles\ Links\ MathLink \ DeveloperKit \ Windows \ SystemAdditions. The 64-bit device libraries are located in C:\Program Files (x86)\Wolfram Research\ Mathematica \ 6.0 \SystemFiles\Links\MathLink\DeveloperKit\Windows-x86-64\SystemAdditions. The 32-bit device libraries must go in C:\Windows\SysWOW64 and the 64-bit libraries must go C:\Windows\system32.

⚠ Some *MathLink* devices, when loaded, in turn load other *MathLink* devices—they will not work on their own. Therefore it is not recommended to install only some of the appropriate components—you should make all the appropriate devices available to your operating system.

SharedMemory Device

"MLSHM32.MLP"/"MLSHM64.MLP"

This is a *MathLink* device that uses the Win32 memory-mapped file mechanism to transfer data between processes. These shared libraries are used when the `"SharedMemory"` protocol is specified when opening a link. In *Mathematica* 6.0, it is the default device for local connections.

For example, a C program that executes:

```
char* argv[] = {"-linkname", "foo", "-linkprotocol", "SharedMemory", "-
linkmode", "connect"};
link = MLOpenArgv( stdenv, argv, argv + 6, 0);
```

will connect to a *Mathematica* program that evaluates:

```
LinkOpen["foo", LinkProtocol -> "SharedMemory", LinkMode -> Listen]
```

TCPIP Device

"MLTCPIP32.MLP"/"MLTCPIP64.MLP"

This is a *MathLink* device that uses the services of the TCP internet protocol to transfer data between processes. These shared libraries are used when the `"TCPIP"` protocol is specified when opening a link. In *Mathematica* 6.0, it is the preferred device for remote connections.

For example, a C program that executes:

```
char* argv[] = {"-linkname", "6000", "-linkprotocol", "TCPIP", "-
linkmode", "connect"};
link = MLOpenArgv( stdenv, argv, argv + 6, 0);
```

will connect to a *Mathematica* program that evaluates:

```
LinkOpen["6000", LinkProtocol -> "TCPIP", LinkMode -> Listen]
```

`LinkProtocol -> "TCPIP"` can be used to connect *MathLink* programs running on different computers on a network.

FileMap Device

"MLMAP32.MLP"

This is a *MathLink* device that uses the Win32 memory-mapped file mechanism to transfer data between processes. These shared libraries are used when the `"FileMap"` protocol is specified when opening a link.

For example, a C program that executes:

```
char* argv[] = {"-linkname", "foo", "-linkprotocol", "FileMap", "-
linkmode", "connect"};
link = MLOpenArgv( stdenv, argv, argv + 6, 0);
```

will connect to a *Mathematica* program that evaluates:

```
LinkOpen["foo", LinkProtocol -> "FileMap", LinkMode -> Listen]
```

These devices are used by default by *MathLink* programs when no `LinkProtocol` is specified.

Windows 64 does not support the `"FileMap"` protocol.

TCP Device

"MLTCP32.MLP"/"MLTCP64.MLP"

This is a *MathLink* device that uses the services of the TCP internet protocol to transfer data between processes. These shared libraries are used when the `"TCP"` protocol is specified when opening a link.

For example, a C program that executes:

```
char* argv[] = {"-linkname", "6000", "-linkprotocol", "TCP", "-linkmode",
"connect"};
link = MLOpenArgv( stdenv, argv, argv + 6, 0);
```

will connect to a *Mathematica* program that evaluates:

```
LinkOpen["6000", LinkProtocol -> "TCP", LinkMode -> Listen]
```

The `"TCP"` `LinkProtocol` can be used to connect *MathLink* programs running on different computers on a network.

The `"TCP"` `LinkProtocol` was deprecated in *Mathematica* 5.1 and only continues to exist for backwards compatibility. Users of `"TCP"` should convert their code to use `"TCPIP"`.

CompilerAdditions Directory

"MLDev32\INCLUDE\", "MLDev64\INCLUDE"

mathlink.h

mathlink.h is the header file that must be included in your C and C++ source files. It should be placed where your compiler can find it. You could place this header file in the same directory as your source files that include it, or in the same location as the standard header files provided with your development tools. Alternatively, you could add the location of mathlink.h to the search path for header files. (This is typically done using command line switches, an environment variable, or a setting in a dialog box.)

"MLDev32\LIB\", "MLDev64\LIB"

This folder contains import libraries that your linker would use to resolve references in your code to functions in the *MathLink* shared library. One of these .LIB files would be referenced in your project file, makefile, or link command line. Because the format of an import library is not standardized, one import library is supplied for each of the most popular development environments.

"ML32I3M.LIB"/"ML64I3M.LIB"

This is the import library for use with the Microsoft C/C++ development tools including Microsoft Visual Studio. (It is a COFF import library created by the Microsoft linker.) This file should be placed where it will be found by the Microsoft linker. To avoid having to specify a path, place this file in your C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\LIB directory. Place the 64-bit .LIB file in C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\Lib\AMD64 directory.

"MLDev32\BIN\", "MLDev64\BIN"

"MPREP.EXE"

"MPREP" is a 32-bit console program that writes *MathLink* programs automatically by processing "template" files. It may be convenient to place this file in the "Bin" directory of your development tools. On Windows 64, "MPREP" is a 64-bit console program.

PrebuiltExamples Directory

This folder contains prebuilt versions of the example programs. "Building *MathLink* Programs" describes how to build them yourself using the source code in the "MathLinkExamples" folder. "Running *MathLink* Programs" describes how to run these programs.

MathLinkExamples Directory

This folder contains the source code for some very simple *MathLink* programs. By using this source code, you can learn how to build and run *MathLink* programs without having to write any code yourself.

Extras Directory

This folder contains sample programs and utilities that you may find useful.

Alternative Components Directory

"DebugLibraries"

This is a copy of "MathLinkLibraries" that does extensive error checking and logs information that may be useful. See the "Troubleshooting" section for a description of how to use this library.

Building *MathLink* Programs

The general procedure for building *MathLink* programs is to include mathlink.h in any C or C++ source files that make *MathLink* function calls, to compile your source files, and then to link the resulting object code with the *MathLink* import library and any other standard libraries required by your application. If your application uses the *MathLink* template mechanism, then your template files must first be processed into a C source file using mprep. The details for several popular development environments are provided here.

The build instructions in "Building *MathLink* Programs with Microsoft Visual Studio" assume you have followed the recommended installation instructions for the *MathLink* compiler additions. If you have placed the compiler additions elsewhere on your hard disk, you may need to modify the instructions by setting environment variables, specifying additional command line arguments, or specifying full pathnames when referring to the *MathLink* compiler additions.

MathLink Versioning

As a shared library, each revision of *MathLink* must maintain compatibility with previous revisions. Yet, at times, new functionality needs to be added. *MathLink* adopts a simple versioning strategy that can be adapted to many compile-time and run-time environments.

Strategy

MathLink evolves by improving its implementation and by improving its interface. The values of `MLREVISION` or `MLINTERFACE` defined in mathlink.h are incremented whenever an improvement is made and released.

`MLREVISION` is the current revision number. It is incremented every time a change is made to the source and *MathLink* is rebuilt and distributed on any platform. (Bug fixes, optimizations, or other improvements transparent to the interface increment only this number.)

`MLINTERFACE` is a name for a documented interface to *MathLink*. This number is incremented whenever a named constant or function is added, removed, or its behavior is changed in a way that could break existing correct client programs. It is expected that the interface to *MathLink* will be improved over time so that implementations of higher numbered interfaces are more complete or more convenient to use for writing effective client programs. In particular, a specific interface provides all the useful functionality of an earlier interface.

For Windows, the different interfaces of the *MathLink* libraries are implemented in different shared libraries. The file "ml32i1.dll" contains the implementation of interface 1 of the latest *MathLink* revision, the file "ml32i2.dll" contains the implementation of interface 2, and the file "ml32i3.dll" contains the implementation of interface 3. You need only keep the latest revision (as shown in the **Properties** box for the shared library) of these files in your Windows system folder to run any *MathLink* program.

In addition, each *MathLink* interface has a separate import library to link against so that developers make a conscious decision as to which interface they require at compile time. So, for example, a developer using Microsoft's Visual Studio who needs functionality that was added for interface 3 must link with "ml32i3m.lib" rather than "ml32i2m.lib" or "ml32i1m.lib".

Using MathLink Template Files

If your program uses the *MathLink* template mechanism as described in "*MathLink* and External Program Communication", you must preprocess your source files that contain template entries using the mprep console program. A template entry is a sequence of lines that contain template keywords. (Each entry defines a *Mathematica* function that when evaluated calls an associated C function.) When mprep processes such source files, it converts template entries into C functions, passes other text through unmodified, and writes out additional C functions that implement a remote procedure call mechanism using *MathLink*. The result is a C source file that is ready for compilation.

For example, the command:

```
mprep addtwo.tm -o addtwotm.c
```

will produce a C source file "addtwotm.c" from the template entries and other text in "addtwo.tm". You would then compile the output file using the C compiler. If you use the "Make" utility to build your program, you could add a rule similar to the following one to your makefile.

```
addtwotm.c : addtwo.tm
  mprep addtwo.tm -o addtwotm.c
```

Building MathLink Programs with Microsoft Visual Studio

Using the Command Line Tools

In order to use the command line tools from a command window, you must run the batch file VCVARSALL.BAT to configure the environment. You can find the file in C:\Program Files\ Mirosoft Visual Studio 8\VC\VCVARSALL.BAT. Alternatively, you can use the shortcut provided in **Start ▸ All Programs ▸ Microsoft Visual Studio 2005 ▸ Visual Studio Tools ▸ Visual Studio 2005 Command Prompt**. VCVARSALL.BAT takes an argument to indicate which set of compiler tools the environment should use. For 32-bit Windows development use: VCVARSALL.BAT x86. For Windows 64 development use: VCVARSALL.BAT amd64. Alternatively on Windows 64 machines you can use the shortcut provided in **Start ▸ All Programs ▸ Microsoft Visual Studio 2005 ▸ Visual Studio Tools ▸ Visual Studio 2005 x64 Win64 Command Prompt**. VCVARSALL.BAT will correctly configure the `PATH`, `INCLUDE`, and `LIB` environment variables so that you can use the Microsoft Compiler tools from your command line environment.

Building a *MathLink* Program to Be Called by the *Mathematica* Kernel

To build the ADDTWO.EXE example program:

1. Start a command prompt window by running **Start ▸ All Programs ▸ Microsoft Visual Studio 2005 ▸ Visual Studio Tools ▸ Visual Studio 2005 Command Prompt** (On Windows 64 use **Start ▸ All Programs ▸ Microsoft Visual Studio 2005 ▸ Visual Studio Tools ▸ Visual Studio 2005 x64 Win64 Command Prompt**).

2. Change to the "addtwo" directory within the "MathLinkExamples" directory.

```
C:
cd "C:\Program Files\Wolfram Research\Mathematica\6.0\"
cd SystemFiles\Links\MathLink\DeveloperKit\Windows\MathLinkExamples\addtwo\
```

   On Windows 64 use the following commands:

```
C:
cd "C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\"
cd SystemFiles\Links\MathLink\DeveloperKit\Windows-
x86-64\MathLinkExamples\addtwo\
```

3. Type the following five commands.

```
SET CL=/nologo /c /DWIN32 /D_WINDOWS /W3 /O2 /DNDEBUG
SET LINK=/NOLOGO /SUBSYSTEM:windows /INCREMENTAL:no /PDB:NONE kernel32.lib
user32.lib gdi32.lib
MPREP addtwo.tm -o addtwotm.c
CL addtwo.c addtwotm.c
LINK addtwo.obj addtwotm.obj ml32i3m.lib /OUT:addtwo.exe
```

   On Windows 64 use the following command in place of the last command:

```
LINK addtwo.obj addtwotm.obj ml64i3m.lib /OUT:addtwo.exe
```

Building a *MathLink* Program That Calls the *Mathematica* Kernel

To build the FACTOR.EXE example program:

1. Start a command prompt window by running **Start ▸ All Programs ▸ Microsoft Visual Studio 2005 ▸ Visual Studio Tools ▸ Visual Studio 2005 Command Prompt** (On Windows 64 use **Start ▸ All Programs ▸ Microsoft Visual Studio 2005 ▸ Visual Studio Tools ▸ Visual Studio 2005 x64 Win64 Command Prompt**).

2. Change to the "factor" directory within the "MathLinkExamples" directory.

   ```
   C:
   cd "C:\Program Files\Wolfram Research\Mathematica\5.1\"
   cd AddOns\MathLink\DeveloperKit\Windows\MathLinkExamples\factor\
   ```

   On Windows 64 use the following commands:

   ```
   C:
   cd "C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\"
   cd SystemFiles\Links\MathLink\DeveloperKit\Windows-
   x86-64\MathLinkExamples\factor\
   ```

3. Type the following four commands.

   ```
   SET CL=/nologo /c /DWIN32 /D_CONSOLE /W3 /O2 /DNDEBUG
   SET LINK=/NOLOGO /SUBSYSTEM:console /INCREMENTAL:no /PDB:NONE kernel32.lib
   user32.lib
   CL factor.c
   LINK factor.obj ml32i2m.lib /OUT:factor.exe
   ```

   On Windows 64 use the following command in place of the last command:

   ```
   LINK factor.obj ml64i3m.lib /OUT:factor.exe
   ```

Debugging a *MathLink* Program Called from the *Mathematica* Kernel

To build a debug version of ADDTWO.EXE

1. Follow steps 1-3 in "Building a *MathLink* Program to Be Called by the *Mathematica* Kernel", replacing the commands in step 3 with the following.

   ```
   SET CL=/nologo /c /DWIN32 /D_WINDOWS /W3 /Z7 /Od /D_DEBUG
   SET LINK=/NOLOGO /SUBSYSTEM:windows /DEBUG /PDB:NONE /INCREMENTAL:no
   kernel32.lib user32.lib gdi32.lib
   MPREP addtwo.tm -o addtwotm.c
   CL addtwo.c addtwotm.c
   LINK addtwo.obj addtwotm.obj ml32i3m.lib /OUT:addtwo.exe
   ```

On Windows 64 use the following command in place of the last command:

```
LINK addtwo.obj addtwotm.obj ml64i3m.lib /OUT:addtwo.exe
```

To debug ADDTWO.EXE:

2. Start Microsoft Visual Studio.
3. From the **File** menu, choose **Open ▸ Project/Solution**.

   The **Open Solution** dialog box appears.
4. Select **Executable Files** from the **Files of type** drop-down list to display .EXE files in the **File Name** list.
5. Select the drive and directory containing ADDTWO.EXE.
6. Select ADDTWO.EXE and click the **Open** button.
7. To start debugging, press F5 or choose the **Start Debugging** command under the **Debug** menu.

   When you are done debugging and close the project solution, you will be asked if you want to save the new solution associated with ADDTWO.EXE. Choose **OK** if you want to retain your breakpoints and other debugger settings.

Debugging a *MathLink* Program That Calls the *Mathematica* Kernel

To build a debug version of FACTOR.EXE

1. Follow steps 1-3 in "Building a *MathLink* Program That Calls the *Mathematica* Kernel", replacing the commands in step 3 with the following.

```
SET CL=/nologo /c /DWIN32 /D_CONSOLE /W3 /Z7 /Od /D_DEBUG
SET LINK=/NOLOGO /SUBSYSTEM:console /INCREMENTAL:no /PDB:NONE /DEBUG
kernel32.lib user32.lib
CL factor.c
LINK factor.obj ml32i3m.lib /OUT:factor.exe
```

   On Windows 64 use the following command in place of the last command:

```
LINK factor.obj ml64i3m.lib /OUT:factor.exe
```

To debug FACTOR.EXE:

2. Start Microsoft Visual Studio.
3. From the **File** menu, choose **Open ▸ Project/Solution**.

   The **Open Solution** dialog box appears.

4. Select **Executable Files** from the **Files of type** drop-down list to display .EXE files in the **File Name** list.
5. Select the drive and directory containing FACTOR.EXE.
6. Select FACTOR.EXE and click the **Open** button.
7. From the **Project** menu, choose **Properties**.

   The **Project Settings** dialog box appears.
8. Click the **Debugging** item under **Configuration Properties**.

   The **Debug Settings** page appears.
9. In the **Command Arguments** textbox, type: `-linklaunch`.
10. Click the **OK** button.
11. To start debugging, press F5 or choose the **Start Debugging** command under the **Debug** menu.
12. When `MLOpenArgcArgv()` is executed, the **Choose a MathLink Program to Launch** dialog box appears. Open "`MathKernel.exe`".

    When you are done debugging and close the project solution, you will be asked if you want to save the new solution associated with FACTOR.EXE. Choose **OK** if you want to retain your breakpoints and other debugger settings.

Short Summary of Compiler Switches

Switch	*Action*
`/nologo`	do not display the copyright notice
`/W3`	display extended warnings
`/Z7`	store debugging information in the object files
`/Zi`	store debugging information in a separate project database file
`/Fdaddtwo.pdb`	specify name of the project database file—used with `/Zi`
`/Od`	turn off optimization (default)
`/O2`	optimizer prefers faster code over smaller code
`/D`	defines used by some standard header files and `mathlink.h`

`/c`	compile only without linking
`@filename`	read more command line arguments from the file
`CFLAGS`	environment variable containing more command line arguments

Short Summary of Linker Switches

Switch	*Action*
`/NOLOGO`	do not display the copyright notice
`/DEBUG`	store debugging information in the executable or project database
`/PDB:NONE`	store debugging information in the executable—used with `/DEBUG`
`/PDB:addtwo.pdb`	override the default name for the project database
`/OUT:addtwo.exe`	name the output file
`/INCREMENTAL:no`	links more slowly but keeps things smaller and neater
`/SUBSYSTEM:windows`	the application does not need a console because it creates its own windows (default when `WinMain()` is defined)
`/SUBSYSTEM:console`	a console is provided (default when `main()` is defined)

Standard System Libraries

Import library	*Base system services*
`kernel32.lib`	base OS support such as the file system, interprocess communication, process control, memory, and the console
`advapi32.lib`	support for security and Registry calls

Import library	*GUI system services*
`user32.lib`	support for user interface elements such as windows, messages, menus, controls, and dialog boxes
`gdi32.lib`	support for drawing text and graphics
`winspool.lib`	support for printing and print jobs
`comdlg32.lib`	support for the common dialogs such as those for opening and saving files and printing

Import library	*Shell system services*
`shell32.lib`	support for drag and drop, associations between executables and filename extensions, and icon extraction from executables

Import library	*OLE system services*
`ole32.lib`	support OLE v2 .1
`oleaut32.lib`	support for OLE automation
`uuid.lib`	support for universally unique identifiers used in OLE and RPC (static library)

Import library	*Database system services*
`odbc32.lib`	access to database management systems through ODBC
`odbccp32.lib`	ODBC setup and administration

Using the Program Build Utility NMAKE

`NMAKE` is a utility provided with Microsoft's development tools that manages the process of building programs. `NMAKE` reads a makefile which describes the dependencies and commands required to build and rebuild one or more programs. `NMAKE` rebuilds any components that have become out of date when one or more prerequisite files have been updated. This document does not describe `NMAKE` or makefiles in detail. A simple makefile is provided here that illustrates how the build commands listed above can be automatically executed by simply typing `NMAKE` at a command prompt. To learn more about `NMAKE`, its general and powerful mechanisms and how to use macros or special forms, see the `NMAKE` Reference in your "Microsoft Visual Studio Guide".

Using a Makefile to Build a Template Program That Uses the WIN32 API

To build ADDTWO.EXE using the `NMAKE` utility:

1. Using a text editor, create a file containing the following text.

```
# addtwo.mak a makefile for building the addtwo.exe example program

CFLAGS = /nologo /c /W3 /Z7 /Od /DWIN32 /D_DEBUG /D_WINDOWS

# Linking against gdi32.lib for access to windowing mechanisms
LFLAGS = /DEBUG /PDB:NONE /NOLOGO /SUBSYSTEM:windows /INCREMENTAL:no
kernel32.lib user32.lib gdi32.lib

# Uncomment the value below for working on a Windows 64 system
# PLATFORM = WIN64
PLATFORM = WIN32

!if "$(PLATFORM)" == "WIN32"
LIBFILE = ml32i3m.lib
!else
LIBFILE = ml64i3m.lib
!endif

addtwo.exe : addtwo.obj addtwotm.obj
  LINK addtwo.obj addtwotm.obj $(LIBFILE) /OUT:addtwo.exe @<<
$(LFLAGS)
<<

addtwo.obj : addtwo.c
  CL @<< addtwo.c
$(CFLAGS)
<<

addtwotm.obj : addtwotm.c
  CL @<< addtwotm.c
$(CFLAGS)
<<

# Need to call mprep to preprocess MathLink template
addtwotm.c : addtwo.tm
  mprep addtwo.tm -o addtwotm.c
```

2. Save the file as "addtwo.mak" in the "addtwo" directory within the "MathLinkExamples" directory.

3. Start a command prompt window by running **Start ▸ All Programs ▸ Microsoft Visual Studio 2005 ▸ Visual Studio Tools ▸ Visual Studio 2005 Command Prompt** (On Windows 64 use **Start ▸ All Programs ▸ Microsoft Visual Studio 2005 ▸ Visual Studio Tools ▸ Visual Studio 2005 x64 Win64 Command Prompt**).

4. Change to the "addtwo" directory.

5. Type the following command.

```
NMAKE /f addtwo.mak
```

Makefiles consist of a collection of build rules, macros and other special forms.

A build rule consists of a target file, followed by a colon, followed by a list of the target's prerequisite files (which must either exist or can be built by other build rules in the makefile), followed by one or more indented lines containing the commands required to build the target from its prerequisites. For example, the makefile above states that the file "addtwotm.c" depends on "addtwo.tm" and should be rebuilt any time "addtwo.tm" is modified. The build command `mprep addtwo.tm -o addtwotm.c` is used to rebuild the target "addtwotm.c".

Macros are named strings of text that can be inserted into the makefile using the notation `$` (*name*). For example, in this makefile, `$(CFLAGS)` is expanded by `NMAKE` wherever it appears into the string `/nologo /c /W3 /Z7 /Od /DWIN32 /D_DEBUG /D_WINDOWS`.

You might expect that the command to compile "addtwo.c" would appear in the makefile simply as `CL $(CFLAGS) addtwo.c`. However it would then be possible for you to edit the definition of `CFLAGS` so that the resulting compiler command exceeds the maximum allowable length of a command line. Because command lines are restricted in length, command line tools often provide a mechanism to read command line arguments from so-called response files. The syntax is generally *@responsefile.* This mechanism is used above along with `NMAKE`'s ability to produce temporary files using the following special form.

<<

text to put in temporary file

<<

Using a Makefile to Build a Console Program

To build FACTOR.EXE using the `NMAKE` utility:

1. Using a text editor, create a file containing the following text.

```
# factor.mak a makefile for building the factor.exe example program
# This makefile builds a console program

CFLAGS = /nologo /c /MLd /W3 /Z7 /Od /DWIN32 /D_DEBUG /D_CONSOLE
LFLAGS = /DEBUG /PDB:NONE /NOLOGO /SUBSYSTEM:console /INCREMENTAL:no
kernel32.lib user32.lib

# Uncomment the value below for working on a Windows 64 system
# PLATFORM = WIN64
PLATFORM = WIN32

!if "$(PLATFORM)" == "WIN32"
LIBFILE = ml32i3m.lib
!else
LIBFILE = ml64i3m.lib
!endif

factor.exe : factor.obj
  LINK factor.obj $(LIBFILE) /OUT:factor.exe @<<
$(LFLAGS)
<<

factor.obj : factor.c
  CL @<< factor.c
$(CFLAGS)
<<
```

2. Save the file as "factor.mak" in the "factor" directory within the "MathLinkExamples" directory.

3. Start a command prompt window by running **Start ▸ All Programs ▸ Microsoft Visual Studio 2005 ▸ Visual Studio Tools ▸ Visual Studio 2005 Command Prompt** (On Windows 64 use **Start ▸ All Programs ▸ Microsoft Visual Studio 2005 ▸ Visual Studio Tools ▸ Visual Studio 2005 x64 Win64 Command Prompt**).

4. Change to the "factor" directory.

5. Type the following two commands.

```
NMAKE /f factor.mak
```

Using the Integrated Development Environment Visual Studio 2005

Steps Common to All Projects

Steps required to use *MathLink* with Microsoft Visual Studio 2005:

1. Copy mathlink.h from the *MathLink* Developer Kit to the Microsoft Visual Studio 2005 Include directory.

   32-bit Windows

   Developer Kit path 32-bit Windows: C:\Program Files\Wolfram Research*Mathematica*\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows\CompilerAdditions\mldev32\include

   Visual Studio 2005 Include directory 32-bit Windows: C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\Include

   Windows 64

   Developer Kit path Windows 64: C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows-x86-64\CompilerAdditions\mldev64\include

   Visual Studio 2005 Include directory Windows 64: C:\Program Files (x86)\Microsoft Visual Studio 8\VC\PlatformSDK\Include

2. Copy the .lib files to the Microsoft Visual Studio Lib directory

   32-bit Windows

   Copy ml32i1m.lib, ml32i2m.lib, and ml32i3m.lib from:
   C:\Program Files\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows\CompilerAdditions\mldev32\lib
   to:
   C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\Lib

   Windows 64

   Copy ml64i2m.lib, and ml64i3m.lib from:
   C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\AddOns\MathLink\DeveloperKit\Windows-x86-64\CompilerAdditions\mldev64\lib
   to:
   C:\Program Files (x86)\Microsoft Visual Studio 8\VC\PlatformSDK\Lib\AMD64

3. Copy mprep.exe:

   32-bit Windows

   from:
   C:\Program Files\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows\CompilerAdditions\mldev32\bin
   to:
   C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\bin

   Windows 64

   from:
   C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows-x86-64\CompilerAdditions\mldev64\bin
   to:
   C:\Program Files (x86)\Microsoft Visual Studio 8\VC\PlatformSDK\bin

Creating a Project for "addtwo.exe"

To create a project solution that can be used to edit, build, and debug, "addtwo.exe":

1. Start Microsoft Visual Studio 2005.

2. Click **File ▸ New ▸ Project**.

   The **New Project** dialog box appears.

3. In the **Project Types** pane click the tree expand icon next to the **Visual C++ Projects**. Select **Win32**. In the **Templates** pane click the **Win32 Project** icon.

4. In the **Location** text field type:

   32-bit Windows

   C:\Program Files\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows\MathLinkExamples

   Windows 64

   C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows-x86-64\MathLinkExamples

   In the **Name** text field type `addtwo`.

   Click **OK**.

   The **Win32 Application Wizard** dialog box appears.

5. Click **Application Settings**. Under the **Additional options** set, click the **Empty Project** text box. Click **Finish**.

6. Select the addtwo project in the **Solution Explorer** by clicking once. From the **Project** menu select **Project ▸ Add Existing Item**.

   The **Add Existing Item** dialog box appears.

7. From the **Look in** pull-down menu select the following directory:

   32-bit Windows

   C:\Program Files\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows\MathLinkExamples\addtwo

   Windows 64

   C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows-x86-64\MathLinkExamples\addtwo

8. In the **File name:** text box enter: `"addtwo.c" "addtwo.tm"` separated by spaces. Click **Add**.

9. A prompt box might appear asking if you want to create a 'New Rule' for building .tm files. Click **No**.

10. In the **Solution Explorer** drag the addtwo.tm file into the **Source Files** folder.

11. Select the addtwo project in the **Solution Explorer** by clicking once. From the **Project** menu select **Project ▸ Add New Item**.

    The **Add New Item** dialog box appears.

    In the **Location** text field add:

    32-bit Windows

    C:\Program Files\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows\MathLinkExamples\addtwo

    Windows 64

    C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows-x86-64\MathLinkExamples\addtwo

    In the **Name:** text field type `addtwotm.c`.

    Click **Add**.

12. Right-click the addtwo project in the **Solution Explorer** and select **Properties**.

13. Click the expand tree button next to **Configuration Properties**.

14. Click the expand tree button next to **Linker**.

15. Click the **Input** entry.

16. In the **Additional Dependencies** text field enter `ml32i3m.lib`.

    Click **OK**.

17. Click addtwo.tm in the **Solution Explorer**. From the **Project** menu select **Project ▸ Properties**.

    The **addtwo.tm Property Pages** dialog box appears.

18. Click the expand tree button next to **Configuration Properties**.

19. Click the expand tree button next to **Custom Build Step**.

20. Click the **General** entry.

21. In the rightmost pane click the empty box across from the **Command Line**. In this text box type (include the quotes): `"$(VCInstallDir)PlatformSDK\bin\mprep.exe" "$(InputPath)" -o "$(ProjectDir)..\addtwotm.c"`

22. In the text field across from the **Outputs** text type: `..\addtwotm.c`.

    Click **OK**. After the build the file `addtwotm.c` will be created in the following location:

    32-bit Windows

    C:\Program Files\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows\MathLinkExamples\addtwo

    Windows 64

    C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows-x86-64\MathLinkExamples\addtwo

23. Right-click the addtwo project in the **SolutionExplorer** and select **Properties**.

24. Click the expand tree button next to **Configuration Properties**.

25. Click the **General** entry.

26. Click the **Project Defaults** expand tree button.

27. Set the pull-down menu opposite **Character Set** to **Not Set**.

    Click **OK**.

28. From the **Build** menu, select **Build ▸ Build Solution**.

29. After the project builds, Microsoft Visual Studio 2005 will display a dialog box informing you that the file addtwotm.c has changed and asking you if you would like to reload the file. Click **Yes**.

30. The addtwo.exe binary is now in:

    32-bit Windows

    C:\Program Files\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows\MathLinkExamples\addtwo\Debug

    Windows 64

    C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows-x86-64\MathLinkExamples\addtwo\Debug

Creating a Project for "factor.exe"

To create a project solution that can be used to edit, build, and debug "factor.exe":

1. Start Microsoft Visual Studio .NET.

2. Click **File ▸ New ▸ Project**.

   The **New Project** dialog box appears.

3. In the **Project Types** pane click the tree expand icon next to the **Visual C++ Projects**. Select **Win32**. In the **Templates** pane click the **Win32 Console Application** icon.

4. In the **Location** text field type:

   32-bit Windows

   C:\Program Files\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows\MathLinkExamples

   Windows 64

   C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows-x86-64\MathLinkExamples

   In the **Name** text field type `factor`.

   Click **OK**.

   The **Win32 Application Wizard** dialog box appears.

5. Click **Application Settings**. Under the **Additional options** set, click the **Empty Project** text box. Click **Finish**.

6. Select the factor project in the **Solution Explorer** by clicking once. From the **Project** menu select **Project ▸ Add Existing Item**.

   The **Add Existing Item** dialog box appears.

7. From the **Look in** pull-down menu select the following directory:

   32-bit Windows

   C:\Program Files\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows\MathLinkExamples\factor

   Windows 64

   C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows-x86-64\MathLinkExamples\factor

8. In the **File name:** text box enter: `factor.c`.

   Click **Add**.

9. Right-click the factor project in the **Solution Explorer** and select **Properties**.

10. Click the expand tree button next to **Configuration Properties**.

11. Click the expand tree button next to **Linker**.

12. Click the **Input** entry.

13. In the **Additional Dependencies** text field enter `ml32i3m.lib`.

    Click **OK**.

14. From the **Build** menu, select **Build ▸ Build Solution**.

15. The factor.exe binary is now in:

    32-bit Windows

    C:\Program Files\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows\MathLinkExamples\factor\Debug

    Windows 64

    C:\Program Files (x86)\Wolfram Research\Mathematica\6.0\SystemFiles\Links\MathLink\DeveloperKit\Windows-x86-64\MathLinkExamples\factor\Debug

Running *MathLink* Programs

The instructions in "Building *MathLink* Programs" describe how to build two *MathLink* programs using source code in the "MathLinkExamples" directory. These two programs, "addtwo.exe" and "factor.exe", are already built for you in the "PrebuiltExamples" folder. Before building them on your own, you should try to run the prebuilt examples to verify that the *MathLink* system additions are installed and working and to learn what to expect from these examples when they are properly built.

After trying out the two examples, you should read the following subsections to learn about other issues you will confront running *MathLink* programs on a Windows computer.

There are two basic types of *MathLink* program, epitomized by the "`addtwo`" and "`factor`" programs. The first is an installable program. An installable program provides new functionality to the kernel by linking a C program to the kernel through a calling mechanism. In order to get this new functionality, the user of *Mathematica* must run the `Install[]` function. With the addtwo example you will be adding a new function called `AddTwo[]` that adds two numbers (provided as arguments) together. The kernel and installable programs have a special relationship that allows them to communicate only with one another. When an installable program is run it requires that you provide some information in order for it to connect. The other type of program is a front end. Front ends do all the work of creating and managing their own links. In addition to the "`factor`" example, the *Mathematica* front end and the *Mathematica* kernel are also examples of the front end type. A front end does not require any extra information in order to run, but it will usually make a connection at some point during its execution.

Running a Prebuilt Example from the Mathematica Kernel

The first example program, “addtwo”, is a *MathLink* template program that is installed into *Mathematica*. That is, this program runs in the background, providing, as a service to *Mathematica,* one or more externally compiled functions. To the *Mathematica* user, these functions appear to be built-in. The “addtwo” program uses a template file that defines the *Mathematica* function `AddTwo[]` as a call to the C function `addtwo()`. (The template mechanism is described in "Setting Up External Functions to Be Called from *Mathematica*".) The source code for this program looks like this:

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

:Evaluate: AddTwo::usage = "AddTwo[x, y] gives the sum of two machine
integers x and y."

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

int __stdcall WinMain( HINSTANCE hinstCurrent, HINSTANCE hinstPrevious,
LPSTR lpszCmdLine, int nCmdShow)
{
    char  buff[512];
    char FAR * buff_start = buff;
    char FAR * argv[32];
    char FAR * FAR * argv_end = argv + 32;

    if( !MLInitializeIcon( hinstCurrent, nCmdShow)) return 1;
    MLScanString( argv, &argv_end, &lpszCmdLine, &buff_start);
    return MLMain( argv_end - argv, argv);
}
```

Evaluate the following two cells.

```
SetDirectory[$InstallationDirectory <>
  "/SystemFiles/Links/MathLink/DeveloperKit/" <> $SystemID <> "/PrebuiltExamples"]
```

```
link = Install["./addtwo"]
```

To see a list of the newly available functions, evaluate the following cell.

```
LinkPatterns[link]
```

This displays the usage message for the `AddTwo[]` function as defined in the file "addtwo.tm".

```
? AddTwo
```

Now try it.

```
AddTwo[2, 3]
```

See what happens if the sum of the two machine integers will not fit in a machine integer or if either argument is not a machine integer. (`2^31 - 1` is the largest machine integer. If your compiler uses 2-byte integers, then `2^15 - 1` is the largest C `int`.)

```
AddTwo[2^31 - 1, 1]
```

The AddTwo program is not prepared for big integers.

```
AddTwo[2^31, 1]
```

This does not match `AddTwo[_Integer, _Integer]`.

```
AddTwo[x, 1]
```

`Install[]` called `LinkOpen[]` and then exchanged information with the external program to set up the definition for `AddTwo[]`. You really do not have to worry about these details, but if you are curious, evaluate the following.

```
?? AddTwo
```

When you are finished using the external program, evaluate the following.

```
Uninstall[link]
```

Invoking the Mathematica Kernel from Within a Prebuilt Example

The second example program, "`factor`", is a *Mathematica* front end in the sense that the *Mathematica* kernel runs in the background providing, as a service to "`factor`", the computational services of the kernel—in this case the ability to factor an integer typed by the user.

This example program, like several of the others, is a console program. That is, rather than presenting a graphical user interface, it interacts with the user in a console window using the standard error, input and output streams defined by ANSI C. (As you experiment with *MathLink*, you may find it convenient, initially, to use a console window. This is discussed in detail in the section "Building *MathLink* Programs".)

To run the example program FACTOR.EXE

1. Start a command prompt.
2. Change to the "PrebuiltExamples" directory.
3. Type the following commands.

   ```
   factor -linklaunch
   ```

   The **Choose a MathLink Program To Launch** dialog box appears.

4. Open MathKernel.exe.

   After a moment, a prompt will appear requesting that you type an integer.

5. Type an integer with fewer than 10 digits and press Enter. (The other factor examples relax the restriction on the size of integer you may type in.)

   ```
   Integer to factor: 123456789
   ```

   The prime factors returned by *Mathematica* are printed and "factor" closes its link with *Mathematica*.

   ```
   Integer to factor: 123456789
   3 ^ 2
   3607 ^ 1
   3803 ^ 1
   ```

Supported Link Protocols

MathLink links are opened by the C function `MLOpenArgcArgv()` and the *Mathematica* functions `LinkCreate`, `LinkLaunch`, and `LinkConnect`, each of which accepts an option for specifying a link protocol. On 32-bit Windows platforms, the legal values for the `LinkProtocol` option are `"SharedMemory"`, `"TCPIP"`, `"FileMap"`, and `"TCP"`. On Windows 64 platforms the legal values for the `LinkProtocol` option are `"SharedMemory"`, `"TCPIP"`, and `"TCP"`. A link protocol is the mechanism used to transport data from one end of a connection to the other. The default is `"SharedMemory"` for all link modes except `LoopBack`.

Note that the `"FileMap"` and `"SharedMemory"` protocols allow link names to be arbitrary words with 31 or fewer characters, while link names are often unsigned 16-bit integers for the `"TCPIP"` and `"TCP"` protocols. Even though `"TCPIP"` and `"TCP"` link names are integers, they are still given as strings (of digits) to `MLOpenArgcArgv()` and `LinkOpen[]`.

Note that for `LinkMode -> Launch`, the link name is not just a pathname to a file to launch, but a command line with space-separated arguments. Hence, spaces in filenames need to be properly quoted. For example

```
LinkOpen["My Prog.exe"] (*wrong!*)
```

would try to launch the program "My" with command line argument of "Prog.exe". Whereas,

```
LinkOpen["\"My Prog.exe\""]
```

would launch "My Prog.exe".

Troubleshooting

- Check that the *MathLink* system additions are placed in the Windows system directory or are placed next to your executable. For Windows 9x and ME use C:\Windows\system. For Windows 2000, 32-bit Windows XP, and 32-bit Windows Vista use C:\Windows\system32. For Windows 64 XP, Windows Server 2003, and Windows 64 Vista, use C:\Windows\system32 for the 64-bit system additions and C:\Windows\SysWOW64 for the 32-bit system additions.
- Make sure that the *MathLink* system additions you are using are from the latest Developer Kit. Two programs that are using runtime components from different revisions of *MathLink* may not work, or if they do work may be slower than using the latest components.
- Turn off compiler optimization until your program is working. This makes compiling faster, debugging easier, and, besides, the optimizer may be broken and the cause of some problems. (Optimized code uses the stack and registers differently than unoptimized code in such a way that may expose or mask a bug in your code. For example, the common mistake of returning a pointer to a local variable may or may not cause a problem depending on stack and register use.)
- Check the return values from the *MathLink* library functions or call `MLError ()` at key points in your program. *MathLink* will often be able to tell you what has gone wrong. (If you do not assign the return value to a variable, you can still check the return value of *MathLink* functions using your debugger's register window. The 32-bit library returns its results in register EAX.)
- While developing your program, place the debug version "ML32I3.DLL" in the same folder. This library will do more extensive error checking and log information that may be useful.
- The files mathlink.h, mprep, "ML32I3.DLL", and the .LIB import libraries are a matched set. If you have used an earlier release of *MathLink*, or a different interface of *MathLink*, you should take care that you do not mix components when building your application.
- The network control panel must show that TCP/IP is installed before you can use `LinkProtocol -> “TCPIP“` or `LinkProtocol -> “TCP“`. Try typing "telnet" at a command prompt. Telnet will not function without TCP/IP installed.

MathLink Development in C (Mac OS X)

This document describes how to compile and run *MathLink* programs written in the C language on Mac OS X systems. ("*MathLink* and External Program Communication" describes how to write *MathLink* programs in both the *Mathematica* language and the C language.)

This document does not teach you, in general, how to use your compiler and other development tools, nor does it teach you how to program in C. If you have any trouble building or running your *MathLink* programs, see the troubleshooting section at the end of this document.

Most of what is described in this document is Unix specific and is applicable to all supported Unix platforms. However, there may be some information which is specific to Mac OS X machines. To learn how to compile and run *MathLink* programs for another platform, see the Developer Guide for that platform.

Supported Development Platforms

The mathlink.framework shared library in the *MathLink* Developer Kit (MLDK) for Mac OS X can be used for Mac OS X 10.3.9 and newer and Xcode 1.1 and newer. The framework includes universal binary support for the 32-bit PPC, 32-bit x86, and 64-bit x86 architectures.

Installing the *MathLink* Components

The *MathLink* Developer Kit is located in the `$InstallationDirectory`/SystemFiles/Links/MathLink/DeveloperKit/MacOSX directory within your *Mathematica* directory.

Recommended Installation

CompilerAdditions Installation

The *MathLink* components that you will need to build *MathLink* programs have already been installed in the *Mathematica* bundle. One way to use these components is to leave them in the Mathematica.app directory and specify their full pathname when you call your compiler. This approach is taken in the example “makefiles” in the section "Building *MathLink* Programs".

An alternative is to copy these components (“mathlink.h”, “libMLi3.a”, and “mathlink.framework”) into directories in which your compiler will automatically search for such files. These directories are commonly “/usr/include” or “/usr/lib” for the “libMLi3.a” and mathlink.h and /Library/Frameworks or ~/Library/Frameworks for “mathlink.framework”. On many systems not all users have write access to these directories.

MathLinkExamples Installation

Copy the “MathLinkExamples” directory to your home directory.

MathLink Framework Components

The following is a description of each file or directory in the MLDK.

CompilerAdditions Directory

mathlink.h

mathlink.h is the header file that must be included in your C and C++ source files. It should be placed where your compiler can find it. You could copy this header file into the same directory as your source files, copy it into the same location as the standard header files, or leave it where it is if you added the *MathLink* directory to the search path for header files.

libMLi3.a

libMLi3.a is the static library that contains all of the *MathLink* functions. It should be included in your project. You could copy this library into the same directory as your source files or leave it where it is if you added the *MathLink* directory to the search path for libraries.

mathlink.framework

mathlink.framework is the dynamic library framework that contains all the *MathLink* functions. Use the framework when you want to build a version of your program that links dynamically with the *MathLink* library. You could copy this library in the same directory as your source files or leave it where it is if you added the *MathLink* directory to the framework search paths.

mprep

mprep is an application that writes *MathLink* programs automatically by processing “template” files. It may be convenient to copy this application into the same directory as your project or to create an alias to it.

mcc

mcc is a script that preprocesses and compiles your *MathLink* source files.

AlternativeLibraries/libMLi3.a

In `$InstallationDirectory` / SystemFiles / Links / MathLink / DeveloperKit / MacOSX/CompilerAdditions/AlternativeLibraries, the *MathLink* Developer Kit contains a version of "libMLi3.a" that was compiled on a Mac OS X 10.4.x system. The byte size of the C `long double` type changed between Mac OS X 10.3.9 and 10.4.0. If you are building on a 10.4.x system, use the "libMLi3.a" in the AlternativeLibraries directory. If you need a "mathlink.framework" that has been updated for 10.4.x, use the "mathlink.framework" found in `$InstallationDirectory`/ SystemFiles / Links / MathLink / DeveloperKit / MacOSX-x86-64 / CompilerAdditions. This "mathlink.framework" contains a Tiger-built Universal version of the *MathLink* framework suitable for running on Mac OS X 10.4.x for PPC, PPC64, x86-64, and I386 architectures.

MathLinkExamples Directory

This directory contains the source code for some very simple *MathLink* programs. By using this source code, you can learn how to build and run *MathLink* programs without having to write any code yourself.

PrebuiltExamples Directory

This directory contains prebuilt versions of the example programs. "Running *MathLink* Programs" describes how to run two of these programs. "Building *MathLink* Programs" describes how to build them yourself using the source code in the "MathLinkExamples" directory.

Building *MathLink* Programs

The general procedure for building *MathLink* programs is to include mathlink.h in any C or C++ source files that make *MathLink* function calls, to compile your source files, and then to link the resulting object code with the "libMLi3.a" library or "mathlink.framework" and any other standard libraries required by your application. If your application uses the *MathLink* template mechanism, then your template files must first be processed into a C source file using mprep.

Using MathLink Template Files

If your program uses the *MathLink* template mechanism as described in "*MathLink* and External Program Communication", you must simultaneously preprocess source files containing template entries using the mprep application. (A template entry is a sequence of lines that contain template keywords. Each entry defines a *Mathematica* function that when evaluated calls an associated C function.) When mprep processes such source files, it converts template entries into C functions, without changing other text, and writes out additional C functions that implement a remote procedure call mechanism using *MathLink*. The result is a C source file that is ready for compilation.

For example, the command

```
mprep addtwo.tm -o addtwotm.c
```

will produce a C source file "addtwotm.c" from the template entries and the other text remaining in "addtwo.tm". You would then compile the output file using the C compiler. If you use the "`make`" utility to build your program, you could add a rule similar to the following to your makefile.

```
addtwotm.c  :  addtwo.tm
    mprep  addtwo.tm  -o addtwotm.c
```

Building MathLink Programs from the Command Line

What follows is a sample makefile needed to build the sample programs in the MLDK, including "`addtwo`" and "`factor`". To build a sample program, in this case "`addtwo`", evaluate the following command in the MathLinkExamples directory.

```
make addtwo
```

Using a Makefile

```
# This makefile can be used to build all or some of the sample
# programs. To build all of them, use the command
# 'make all'. To build one, say addtwo, use the command
# 'make addtwo'.

MLINKDIR =
/Applications/Mathematica.app/SystemFiles/Links/MathLink/DeveloperKit
SYS = MacOSX
CADDSDIR = ${MLINKDIR}/${SYS}/CompilerAdditions

INCDIR = ${CADDSDIR}
```

```
LIBDIR = ${CADDSDIR}

MPREP = ${CADDSDIR}/mprep

all : addtwo bitops counter factor factor2 factor3 quotient reverse
sumalist

addtwo : addtwotm.o addtwo.o
    ${CC} -I${INCDIR} addtwotm.o addtwo.o -L${LIBDIR} -lMLi3 -o $@

bitops : bitopstm.o bitops.o
    ${CC} -I${INCDIR} bitopstm.o bitops.o -L${LIBDIR} -lMLi3 -o $@

counter : countertm.o
    ${CC} -I${INCDIR} countertm.o -L${LIBDIR} -lMLi3 -o $@

factor : factor.o
    ${CC} -I${INCDIR} factor.o -L${LIBDIR} -lMLi3 -o $@

factor2 : factor2.o
    ${CC} -I${INCDIR} factor2.o -L${LIBDIR} -lMLi3 -o $@

factor3 : factor3.o
    ${CC} -I${INCDIR} factor3.o -L${LIBDIR} -lMLi3 -o $@

quotient : quotient.o
    ${CC} -I${INCDIR} quotient.o -L${LIBDIR} -lMLi3 -o $@

reverse : reversetm.o
    ${CC} -I${INCDIR} reversetm.o -L${LIBDIR} -lMLi3 -o $@

sumalist : sumalisttm.o sumalist.o
    ${CC} -I${INCDIR} sumalisttm.o sumalist.o -L${LIBDIR} -lMLi3 -o $@

.c.o :
    ${CC} -c -I${INCDIR} $<

addtwotm.c : addtwo.tm
    ${MPREP} $? -o $@

bitopstm.c : bitops.tm
    ${MPREP} $? -o $@
```

```
countertm.c : counter.tm
    ${MPREP} $? -o $@

reversetm.c : reverse.tm
    ${MPREP} $? -o $@

sumalisttm.c : sumalist.tm
    ${MPREP} $? -o $@
```

Building Mac OS X MathLink Programs with Xcode

Creating a Project

To create a project that can be used to edit, build, and debug "`addtwo`":

1. Start Xcode.
2. From the **File** menu, choose **New Project**.

   The **New Project** dialog box appears.
3. In the **New Project** dialog, select **Standard Tool** and then click **Next**.

   The **New Standard Tool** dialog box appears.
4. Create the Project.

   In the **New Standard Tool** dialog box, enter addtwo as the project name, and press the Tab key. The default location for your project will be ~/addtwo/. This is a directory in your home directory called addtwo (e.g. /Users/ < *login name* >/addtwo). Later steps assume that you use this default directory. Click **Finish**.
5. Copy the source files to the project directory.

   Start the Terminal application and change the directory (`cd`) to where *Mathematica* was installed. When you reach that directory, `cd` to `$InstallationDirectory` / Mathematica.app/SystemFiles/Links/MathLink/DeveloperKit/MacOSX/MathLinkExamples.

   ```
   cd
   "/Applications/Mathematica.app/SystemFiles/Links/MathLink/DeveloperKit/MacO
   SX/MathLinkExamples"
   ```

   Copy the addtwo source files to your project directory that you chose in step 4 using the copy (`cp`) command.

   ```
   cp addtwo.tm addtwo.c ~/addtwo/
   ```

6. Run mprep on the template file.

Change directory to the project directory.

```
cd ~/addtwo/
```

Use mprep to generate a source file.

```
/Applications/Mathematica.app/SystemFiles/Links/MathLink/DeveloperKit/
CompilerAdditions/mprep addtwo.tm -o addtwo.tm.c
```

7. Add files to the project.

- In the **addtwo** project window under the **Groups & Files** pane click the triangle next to the **addtwo** entry.
- Right-click on the **Source** entry.
- Select **Add ▸ Existing Files**.
- From the drop-down menu click *< user name >* directory and then click the addtwo directory, select **addtwo.c**, and click **Add**.
- In the next drop-down menu be sure that the program **addtwo** box is checked in the **Add To Targets** section and click **Add**.
- Right-click on the **Source** entry.
- Select **Add ▸ Existing Files**.
- From the drop-down menu click *< user name >* directory and then click the **addtwo** directory, select **addtwo.tm.c**, and click **Add**.
- In the next drop-down menu be sure that the program **addtwo** box is checked in the **Add To Targets** section and click **Add**.
- Click the Finder icon in the Dock to make the Finder active. Go to the folder where *Mathematica* is installed.
- Click on the *Mathematica* icon while pressing the Ctrl key. A popup menu will appear. Select **Show Package Contents** to view the contents of *Mathematica*. Open the folders SystemFiles/Links/MathLink/DeveloperKit/MacOSX/CompilerAdditions.
- Drag the file mathlink.h from the **Finder** window to the **Source** group to the **Groups & Files** list in Xcode. Make sure that **Copy items into destination folder** is not checked and click the **Add** button.
- Drag the file libMLi3.a from the **Finder** window to the **Source** group to the **Groups & Files** list in Xcode. Make sure that **Copy items into destination folder** is not checked and click the **Add** button.

8. Remove main.c from the project.

   Right-click on the file main.c and select **Delete**. In the drop-down menu click the **Delete References & Files** button.

9. Build the project.

   From the **Build** menu, select **Build**.

Creating a Project for "factor"

To create a project that can be used to edit, build, and debug "factor":

1. Start Xcode.

2. From the **File** menu, choose **New Project**.

   The **New Project** dialog box appears.

3. In the **New Project** dialog box, select **Standard Tool** and then click **Next**.

   The **New Standard Tool** dialog box appears.

4. Create the Project.

   In the **New Standard Tool** dialog, enter factor as the project name and press the Tab key. The default location for your project will be ~/factor/. This is a directory in your home directory called factor (e.g. /Users/ *< login name >*/factor). Later steps assume that you use this default directory.

   Click **Finish**.

5. Copy the source files to the project directory.

   Start the Terminal application and change the directory (`cd`) to where *Mathematica* was installed. When you reach that directory, `cd` to Mathematica.app / SystemFiles / Links / MathLink/DeveloperKit/MacOSX/MathLinkExamples.

   ```
   cd /Applications/Mathematica.app/SystemFiles/Links/MathLink/DeveloperKit/
   MacOSX/MathLinkExamples
   ```

   Copy the factor source files to your project directory that you chose in step 4 using the copy (`cp`) command.

   ```
   cp factor.c ~/factor/
   ```

6. Change directory to the project directory.

7. Add files to the project.

- In the **factor** project window under the **Groups & Files** pane click the triangle next to the **factor** entry.
- Right-click on the **Source** entry.
- Select **Add ▸ Existing Files**.
- From the drop-down menu click *< user name >* directory and then click the **addtwo** directory, select **factor.c**, and click **Add**.
- Click the Finder icon in the Dock to make the Finder active. Go to the folder where *Mathematica* is installed.
- Click the *Mathematica* icon while pressing the Ctrl key. A pop-up menu will appear. Select **Show Package Contents** to view the contents of *Mathematica*. Open the folders SystemFiles/Links/MathLink/DeveloperKit/MacOSX/CompilerAdditions.
- Drag the file mathlink.h from the **Finder** window to the **Source** group to the **Groups & Files** list in Xcode. Make sure that **Copy items into destination folder** is not checked and click the **Add** button.
- Drag the file libMLi3.a from the **Finder** window to the **Source** group to the **Groups & Files** list in Xcode. Make sure that **Copy items into destination folder** is not checked and click the **Add** button.

8. Remove main.c from the project.

Right-click on the file main.c and select **Delete**. In the drop-down menu click the **Delete References & Files** button.

9. Build the project.

From the **Build** menu, select **Build**.

Using mcc

mcc is a script that preprocesses and compiles your *MathLink* source files. It will preprocess *MathLink* templates in any file whose name ends with .tm, and then call "`cc`" on the resulting C source code. mcc will pass command-line options and other files directly to "`cc`". Following is a command that would build the "`addtwo`" application using mcc.

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

Running *MathLink* Programs

The instructions in "Building *MathLink* Programs" describe how to build two *MathLink* programs using source code in the “MathLinkExamples” directory. These two programs, “`addtwo`” and “`factor`”, are already built for you in the “PrebuiltExamples” folder. Before building them on your own, you should try to run the prebuilt examples to verify that the *MathLink* system additions are installed and working and to learn what to expect from these examples when they are properly built. The rest of the comments assume that you are using the programs found in the Developer Kit.

Running a Prebuilt Example from the Mathematica Kernel

The first example program, “`addtwo`”, is a *MathLink* template program that is installed into *Mathematica*. That is, this program runs in the background, providing, as a service to *Mathematica,* one or more externally compiled functions. To the *Mathematica* user, these functions appear to be built-in. In order to get this new functionality, the user of *Mathematica* must run the `Install[]` function. The “`addtwo`” program uses a template file that defines the *Mathematica* function `AddTwo[]` as a call to the C function `addtwo()`. (The template mechanism is described in "*MathLink* and External Program Communication".) The source code for this program looks likes this.

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

:Evaluate: AddTwo::usage = "AddTwo[x, y] gives the sum of two machine
integers x and y."

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

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

Evaluate the following two cells.

```
SetDirectory[$InstallationDirectory <>
  "/SystemFiles/Links/MathLink/DeveloperKit/PrebuiltExamples"]
```

```
link = Install["./addtwo"]
```

To see a list of the newly available functions, evaluate the following cell.

```
LinkPatterns[link]
```

This displays the usage message for the `AddTwo[ ]` function as defined in the file "addtwo.tm".

```
? AddTwo
```

Now try it:

```
AddTwo[2, 3]
```

See what happens if the sum of the two machine integers will not fit in a machine integer or if either argument is not a machine integer. (`2^31-1` is the largest machine integer. If your compiler uses 2-byte integers, then `2^15-1` is the largest C `int`.)

```
AddTwo[2 ^ 31 - 1, 1]
```

The "`addtwo`" program is not prepared for big integers.

```
AddTwo[2 ^ 31, 1]
```

This does not match `AddTwo[_Integer, _Integer]`.

```
AddTwo[x, 1]
```

`Install[]` called `LinkOpen[]` and then exchanged information with the external program to set up the definition for `AddTwo[ ]`. You really do not have to worry about these details, but if you are curious, evaluate the following.

```
?? AddTwo
```

When you are finished using the external program, evaluate the following.

```
Uninstall[link]
```

Invoking the Mathematica Kernel from Within a Prebuilt Example

The second example program, "`factor`", is a *Mathematica* front end in the sense that the *Mathematica* kernel runs in the background providing, as a service to "`factor`", the computational services of the kernel—in this case, the ability to factor an integer typed by the user. These examples assume that *Mathematica* is installed in the Applications directory.

Launch the "`factor`" application by executing the following command.

```
factor -linkmode launch -linkname
'"/Applications/Mathematica.app/Contents/MacOS/MathKernel" -mathlink'
```

After a moment, a prompt will appear requesting that you type an integer. Type an integer with fewer than 10 digits and press the Return key. (The other factor examples relax the restriction on the size of integer you may type in.)

```
Integer to factor:
```

The prime factors returned by *Mathematica* are printed, and "`factor`" closes its link with *Mathematica*.

```
Integer to factor: 123456789
3 ^ 2
3607 ^ 1
3803 ^ 1
```

Supported Link Protocols

The C function `MLOpenArgcArgv()` and the *Mathematica* function `LinkOpen[]` are documented in "*MathLink* and External Program Communication". On Macintosh OS X machines, the legal values for the `LinkProtocol` option are "`TCPIP`", "`TCP`", "`SharedMemory`", and "`Pipes`". A `LinkProtocol` is the mechanism used to transport data from one end of a connection to the other. "`Pipes`" is the default protocol for `LinkMode -> Launch` links. "`SharedMemory`" is the default for `LinkMode -> Listen` and `LinkMode -> Connect` links.

Note that link names are unsigned 16-bit integers for the "`TCPIP`" and "`TCP`" protocols. Even though "`TCPIP`" link names are integers, they are still given as strings (of digits) to `MLOpenArgcArgv()` and `LinkOpen[]`.

Troubleshooting

- Turn off compiler optimization until your program is working. This makes compiling faster, debugging easier, and, besides, the optimizer may be broken and the cause of some problems. (Optimized code uses the stack and registers differently than unoptimized code in such a way that may expose or mask a bug in your code. For example, the common mistake of returning a pointer to a local variable may or may not cause a problem depending on stack and register use.)
- Check the return values from the *MathLink* library functions or call `MLError()` at key points in your program. *MathLink* will often be able to tell you what has gone wrong.
- The files "mathlink.h" and "libMLi3.a" are a matched set. If you have used an earlier release of *MathLink*, you should take care that you do not mix components when building your application.
- Check whether the C compiler you are using supports prototypes. If it does not, you will need to change your code and the way you build your project. This is explained in the section "Building *MathLink* Programs".

MathLink Development in C (Unix and Linux)

This document describes how to compile and run *MathLink* programs written in the C language on Linux/Unix systems. ("*MathLink* and External Program Communication" describes how to write *MathLink* programs in both the *Mathematica* language and the C language.)

This document does not teach you, in general, how to use your compiler and other development tools, nor does it teach you how to program in C. If you have any trouble building or running your *MathLink* programs, see the "Troubleshooting" section at the end of this document.

Most of what is described in this document is Linux/Unix specific, and is applicable to all supported Linux/Unix platforms. To learn how to compile and run *MathLink* programs for another platform, see the Developer Guide for that platform.

Supported Development Platforms

As a shared library, *MathLink* can be used with any development environment that adheres to the standard calling conventions and binary interfaces as specified by the following compilers listed.

While some of the following compilers listed integrate with integrated development environments produced by the compiler creators, they also function equally well with a `make` utility.

$SystemID	C compiler	C++ compiler
AIX-Power64	IBM XL C Enterprise Edition V8 .0 for AIX	IBM XL C++ Enterprise Edition V8.0 for AIX
HPUX-PA64	HP92453-01 B.11.11.14 HP C Compiler	HP ANSI C++ B3910B A.03.67
Linux	gcc - 3.2.3 20030502 (Red Hat Linux 3.2.3-52)	g++ - 3.2.3 20030502 (Red Hat Linux 3.2.3-52)
Linux-IA64	Intel C 9.0	Intel C++ 9.0
Linux-x86-64	gcc - 3.2.3 20030502 (Red Hat Linux 3.2.3-34)	g++ - 3.2.3 20030502 (Red Hat Linux 3.2.3-34)
Solaris-SPARC	Sun C 5.8 2005/10/13	Sun C++ 5.8 2005/10/13
Solaris-x86-64	Sun C 5.8 Patch 121016-03 2006/06/07	Sun C++ 5.8 Patch 121018-04 2006/08/02

Installing the *MathLink* Components

The *MathLink* Developer Kit (MLDK) is located in the directory `$InstallationDirectory/` SystemFiles/Links/MathLink/DeveloperKit/`$SystemID`within your *Mathematica* directory.

Recommended Installation

CompilerAdditions Installation

The *MathLink* components that you will need to build *MathLink* programs have already been installed by the *Mathematica* installer. One way to use these components is to leave them in the *Mathematica* directory and specify their full path name when you call your compiler. This approach is taken in the example "makefiles" in the section “Building *MathLink* Programs”.

An alternative is to copy these components ("mathlink.h," "libML32i3.a," and "libML32i3.so") into directories in which your compiler will automatically search for such files. These directories are commonly /usr/include and /usr/lib, but may be different on your system. On many systems not all users have write access to these directories.

MathLinkExamples Installation

Copy the MathLinkExamples directory to your home directory.

MathLink Framework Components

The following is a description of each file or directory in the MLDK.

CompilerAdditions Directory

mathlink.h

mathlink.h is the header file that must be included in your C and C++ source files. It should be placed where your compiler can find it. You could copy this header file into the same directory as your source files that include it, or in the same location as the standard header files, or leave it where it is if you added the *MathLink* directory to the search path for header files.

libML32i3.a/libML64i4.a

This is the static library that contains all the *MathLink* functions. It should be included in your project. You could copy this library into the same directory as your source files, or leave it where it is if you added the *MathLink* directory to the search path for libraries. The 32/64 indicates whether the library is a 32-bit or a 64-bit version of the *MathLink* library.

libML32i3.so/libML64i3.a/(.sl on HPUX)

This is the dynamic shared library that contains all of the *MathLink* functions. It should be included in your project. You could copy this library into the same directory as your source files, into a systemwide location such as /lib or /usr/lib, or leave it where it is if you added the *MathLink* directory to the search path for libraries. The 32/64 indicates whether the library is a 32-bit or a 64-bit version of the *MathLink* library.

mprep

mprep is an application that writes *MathLink* programs automatically by processing “template” files. It may be convenient to copy this application into the same directory as your project or to create an alias to it.

mcc

mcc is a script that preprocesses and compiles your *MathLink* source files.

MathLinkExamples Directory

This directory contains the source code for some very simple *MathLink* programs. By using this source code, you can learn how to build and run *MathLink* programs without having to write any code yourself.

PrebuiltExamples Folder

This folder contains prebuilt versions of the example programs. "Running *MathLink* Programs" describes how to run two of these programs. "Building *MathLink* Programs" describes how to build them yourself using the source code in the MathLinkExamples folder.

Building *MathLink* Programs

The general procedure for building *MathLink* programs is to include mathlink.h in any C or C++ source files that make *MathLink* function calls, to compile your source files, and then to link the resulting object code with the “libML32i3.a”, “libML64i3”, “libML32i3.so”, or “libML64i3.so” library and any other standard libraries required by your application. If your application uses the *MathLink* template mechanism, then your template files must first be processed into a C source file using mprep.

Using MathLink Template Files

If your program uses the *MathLink* template mechanism as described in "*MathLink* and External Program Communication", you must simultaneously preprocess your source files that contain template entries using the mprep application. (A template entry is a sequence of lines that contain template keywords. Each entry defines a *Mathematica* function that when evaluated calls an associated C function.) When mprep processes such source files, it converts template entries into C functions, passes other text through unmodified, and writes out additional C functions that implement a remote procedure call mechanism using *MathLink*. The result is a C source file that is ready for compilation.

For example, the command

```
mprep  addtwo.tm  -o addtwotm.c
```

will produce a C source file "`addtwotm.c`" from the template entries and other text in "`addtwo.tm`". You would then compile the output file using the C compiler. If you use the "`make`" utility to build your program, you could add a rule similar to the following one to your makefile.

```
addtwotm.c  :  addtwo.tm
    mprep  addtwo.tm  -o addtwotm.c
```

Building MathLink programs

What follows is a sample makefile needed to build the sample programs in the MLDK, including "`addtwo`" and "`factor`". To build a sample program, in this case "`addtwo`", evaluate the following command in the MathLinkExamples directory.

```
make addtwo
```

Using a Makefile

```
# This makefile can be used to build all or some of the sample
# programs. To build all of them, use the command
# 'make all'. To build one, say addtwo, use the command
# 'make addtwo'.

MLINKDIR =
/usr/local/Wolfram/Mathematica/6.0/SystemFiles/Links/MathLink/DeveloperKit
SYS = Linux  # Set this value with the result of evaluating $SystemID
CADDSDIR = ${MLINKDIR}/${SYS}/CompilerAdditions

INCDIR = ${CADDSDIR}
LIBDIR = ${CADDSDIR}
```

```
EXTRALIBS = -lm -lpthread -lrt # Set these with appropriate libs for your
system.
MLLIB = ML32i3 # Set this to ML64i3 if using a 64-bit system

MPREP = ${CADDSDIR}/mprep

all : addtwo bitops counter factor factor2 factor3 quotient reverse
sumalist

addtwo : addtwotm.o addtwo.o
    ${CC} -I${INCDIR} addtwotm.o addtwo.o -L${LIBDIR} -l${MLLIB}
${EXTRALIBS} -o $@

bitops : bitopstm.o bitops.o
    ${CC} -I${INCDIR} bitopstm.o bitops.o -L${LIBDIR} -l${MLLIB}
${EXTRALIBS} -o $@

counter : countertm.o
    ${CC} -I${INCDIR} countertm.o -L${LIBDIR} -l${MLLIB} ${EXTRALIBS} -o $@

factor : factor.o
    ${CC} -I${INCDIR} factor.o -L${LIBDIR} -l${MLLIB} ${EXTRALIBS} -o $@

factor2 : factor2.o
    ${CC} -I${INCDIR} factor2.o -L${LIBDIR} -l${MLLIB} ${EXTRALIBS} -o $@

factor3 : factor3.o
    ${CC} -I${INCDIR} factor3.o -L${LIBDIR} -l${MLLIB} ${EXTRALIBS} -o $@

quotient : quotient.o
    ${CC} -I${INCDIR} quotient.o -L${LIBDIR} -l${MLLIB} ${EXTRALIBS} -o $@

reverse : reversetm.o
    ${CC} -I${INCDIR} reversetm.o -L${LIBDIR} -l${MLLIB} ${EXTRALIBS} -o $@

sumalist : sumalisttm.o sumalist.o
    ${CC} -I${INCDIR} sumalisttm.o sumalist.o -L${LIBDIR} -l${MLLIB}
${EXTRALIBS} -o $@

.c.o :
    ${CC} -c -I${INCDIR} $<
```

```
addtwotm.c : addtwo.tm
    ${MPREP} $? -o $@

bitopstm.c : bitops.tm
    ${MPREP} $? -o $@

countertm.c : counter.tm
    ${MPREP} $? -o $@

reversetm.c : reverse.tm
    ${MPREP} $? -o $@

sumalisttm.c : sumalist.tm
    ${MPREP} $? -o $@
```

Use the following table to determine the extra libraries needed for linking *MathLink* programs on your system.

$SystemID	EXTRALIBS
AIX – Power64	-lm -lpthread -lc128
HPUX – PA64	-lm /usr/lib/libdld.sl /usr/lib/libm.0 -lpthread -lrt
Linux	-lm -lpthread -lrt
Linux – IA64	-lm -lpthread -lrt
Linux – x86 – 64	-lm -lpthread -lrt
Solaris – SPARC	-lm -lsocket -lnsl -lrt
Solaris – x86 – 64	-lm -lsocket -lnsl -lrt

Using mcc

mcc is a script that preprocesses and compiles your *MathLink* source files. It will preprocess *MathLink* templates in any file whose name ends with `.tm`, and then call "`cc`" on the resulting C source code. mcc will pass command-line options and other files directly to `cc`. Following is a command that would build the "`addtwo`" application using mcc.

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

Running *MathLink* Programs

The instructions in "Building *MathLink* Programs" describe how to build two *MathLink* programs using source code in the “MathLinkExamples” directory. These two programs, “`addtwo`” and “`factor`”, are already built for you in the “PrebuiltExamples” folder. Before building them on your own, you should try to run the prebuilt examples to verify that the *MathLink* system additions are installed and working and to learn what to expect from these examples when they are properly built.

There are two basic types of *MathLink* program, epitomized by the "`addtwo`" and "`factor`" programs. The first is an installable program. An Installable program provides new functionality to the kernel by linking a C program to the kernel through a calling mechanism. In order to get this new functionality, the user of *Mathematica* must run the `Install[]` function. With the “`addtwo`” example you will be adding a new function called `AddTwo[]` that adds two numbers (provided as arguments) together. The kernel and installable programs have a special relationship that allows them only to communicate with one another. When an installable program is run it requires that you provide some information in order for it to connect. The other type of program is a front end. Front ends do all of the work of creating and managing their own links. In addition to the "`factor`" example, the *Mathematica* front end and the *Mathematica* kernel are also example of the front end type. A front end does not require any extra information in order to run, but it will usually make a connection at some point during its execution.

Running a Prebuilt Example from the Mathematica Kernel

The first example program, “`addtwo`”, is a *MathLink* template program that is installed into *Mathematica*. That is, this program runs in the background, providing, as a service to *Mathematica,* one or more externally compiled functions. To the *Mathematica* user, these functions appear to be built-in. The “`addtwo`” program uses a template file that defines the *Mathematica* function `AddTwo[]` as a call to the C function `addtwo()`. (The template mechanism is described in "*MathLink* and External Program Communication".) The source code for this program looks likes this.

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

:Evaluate: AddTwo::usage = "AddTwo[x, y] gives the sum of two machine
integers x and y."

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

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

Edit the path string and evaluate the following two cells.

```
SetDirectory[$InstallationDirectory <>
  "/SystemFiles/Links/MathLink/DeveloperKit/" <> $SystemID <> "/PrebuiltExamples"]
```

```
link = Install["./addtwo"]
```

To see a list of the newly available functions, evaluate this cell.

```
LinkPatterns[link]
```

This displays the usage message for the `AddTwo[ ]` function as defined in the file “addtwo.tm”.

```
? AddTwo
```

Now try it:

```
AddTwo[2, 3]
```

See what happens if the sum of the two machine integers will not fit in a machine integer or if either argument is not a machine integer. (`2^31-1` is the largest machine integer. If your compiler uses 2-byte integers, the `2^15-1` is the largest C `int`.)

```
AddTwo[2^31 - 1, 1]
```

The “`addtwo`” program is not prepared for big integers:

```
AddTwo[2^31, 1]
```

This does not match `AddTwo[_Integer, _Integer]`:

```
AddTwo[x, 1]
```

`Install[]` called `LinkOpen[]` and then exchanged information with the external program to set up the definition for `AddTwo[]`. You really do not have to worry about these details, but if you are curious, evaluate the following.

```
?? AddTwo
```

When you are finished using the external program, evaluate the following.

```
Uninstall[link]
```

Invoking the Mathematica Kernel from Within a Prebuilt Example

The second example program, "`factor`", is a *Mathematica* front end in the sense that the *Mathematica* kernel runs in the background providing, as a service to "`factor`", the computational services of the kernel—in this case the ability to factor an integer typed by the user.

Launch the "`factor`" application by executing the following command:

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

After a moment, a prompt will appear requesting that you type an integer. Type an integer with fewer than 10 digits and press the Enter key. (The other factor examples relax the restriction on the size of integer you may type in.)

```
Integer to factor:
```

The prime factors returned by *Mathematica* are printed and "`factor`" closes its link with *Mathematica*.

```
Integer to factor: 123456789
3 ^ 2
3607 ^ 1
3803 ^ 1
```

Supported Link Protocols

The C function `MLOpenArgcArgv()` and the *Mathematica* function `LinkOpen[]` are documented in "*MathLink* and External Program Communication". On Linux/Unix machines, the legal values for the `LinkProtocol` option are `"TCPIP"`, `"TCP"`, `"SharedMemory"`, and `"Pipes"`. A `LinkProtocol` is the mechanism used to transport data from one end of a connection to the other. `"Pipes"` is the default protocol for all `LinkMode -> Launch` links. `"SharedMemory"` is the default protocol for all `LinkMode -> Listen` and `LinkMode -> Connect` links.

Note that link names are unsigned 16-bit integers for the `"TCPIP"` and `"TCP"` protocols. Even though `"TCPIP"` link names are integers, they are still given as strings (of digits) to `MLOpenArgcArgv()` and `LinkOpen[]`.

Troubleshooting

- Turn off compiler optimization until your program is working. This makes compiling faster, debugging easier, and, besides, the optimizer may be broken and the cause of some problems. (Optimized code uses the stack and registers differently than unoptimized code in such a way that may expose or mask a bug in your code. For example, the common mistake of returning a pointer to a local variable may or may not cause a problem depending on stack and register use.)
- Check the return values from the *MathLink* library functions or call `MLError()` at key points in your program. *MathLink* will often be able to tell you what has gone wrong.
- The files "mathlink.h", "libML32i3.a", and "libML32i3.so" (libML64i3.* on 64-bit platforms) are a matched set. If you have used an earlier release of *MathLink*, you should take care that you do not mix components when building your application.
- Check whether the C compiler you are using supports prototypes. If it does not, you will need to change your code and the way you build your project. This is explained in the section "Building *MathLink* Programs".