The High-Energy Physicist's Guide to MathLink

T. Hahn[a]

[a]Max-Planck-Institut für Physik
Föhringer Ring 6, D–80805 Munich, Germany

MPP-2011-88
PACS: 02.70.Wz, 07.05.Bx, 07.05.Wr

MathLink is Wolfram Research's protocol for communicating with the Mathematica Kernel and is used extensively in their own Notebook Frontends. The Mathematica Book insinuates that linking C programs with MathLink is straightforward but in practice there are quite a number of stumbling blocks, in particular in cross-language and cross-platform usage. This write-up tries to clarify the main issues and hopefully makes it easier for software authors to set up Mathematica interfacing in a portable way.

1. Introduction

Wolfram Research's Mathematica is conceptually set up in two pieces, the Kernel (the computational engine) and the Notebook Frontend (the GUI, also used for formatting, rendering, etc.). The MathLink protocol is the means of communicating with a running Mathematica Kernel, both from the Kernel invoking an external program and an external program invoking the Kernel, where most commonly the 'external program' is the Frontend. If necessary, the communication can take place across the internet through a TCP connection.

The MathLink SDK is installed together with Mathematica and its API is documented in Mathematica's Documentation Center ('Help' menu), which describes the operations for linking a C program with MathLink as rather trivial. In everyday work this is true only for simple C programs, however. Cross-language or cross-platform use in particular is fairly non-straightforward.

This article does not cover interpreted languages such as Java or Python for some of which own interfaces exist (e.g. J/Link).

MathLink has been a moving target over the Mathematica versions and platforms. The scripting code given herein represents the state of affairs up to Mathematica 8, MacOS 10.6, Windows 7/Cygwin 1.7, and, of course, any flavour of Linux. The scripts are described in the following for better understanding but can largely be used as black boxes. All code is available for download at `http://feynarts.de/mathlink/`. The line numbers of the code excerpts in this text do not necessarily correspond to the actual scripts.

2. General setup

There are various ways of setting up a MathLink program, some of which depend on the programming environment (e.g. XCode). We concentrate here on the 'standard' MathLink template file (`.tm`) which is portable across all platforms.

The following is only a brief introduction to the template file format. For a more thorough treatment and reference please see the Mathematica Help Browser under '.tm file'.

A template file consists of three parts:
1. A header identifying the functions visible from Mathematica, for example

```
:Begin:
:Function: a0
:Pattern: A0[m_, opt___Rule]
:Arguments: {N[m],
   N[Delta /. {opt} /. Options[A0]],
   N[Mudim /. {opt} /. Options[A0]]}
:ArgumentTypes: {Real, Real, Real}
:ReturnType: Real
:End:

:Evaluate: Options[A0] =
   {Delta -> 0, Mudim -> 1}
```

Ostensibly, argument and option processing happens at this point, and the C function receives exactly the quantities given under `:Arguments:`.

2. C code implementing those functions,

```
#include "mathlink.h"

static double a0(const double m,
const double delta, const double mudim) {
  return (m == 0) ? 0 :
    m*(1 - log(m/mudim) + delta);
}
```

3. A main function which might set up global variables, invoke initialization routines, etc., but eventually hands control over to `MLMain`:

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

`MLMain` returns if the MathLink program is uninstalled either by an explicit `Uninstall` or by quitting the corresponding Mathematica Kernel. After that `main` may want go through some finalization procedure (e.g. closing files) before finally terminating.

The MathLink API functions are documented in Mathematica's Help Browser, including a tutorial under 'tutorial/MathLinkAndExternalProgramCommunicationOverview'.

Compiling such a program should in principle be as easy as substituting `mcc` for `cc` on the command line, as in:

```
mcc -o mlprog mlprog.tm
```

More on compilation in Sect. 4 below, however.

Once the MathLink program has been built successfully, it can be installed in Mathematica using

```
Install["mlprog"]
```

Unless invoked with an explicit path, the program has to be on either the system `PATH` or on Mathematica's `$Path`.

Alternately, start the MathLink program on the command line:

```
> ./mlprog
Create link:
```

Choose an arbitrary string as a name for the link and enter it here. In Mathematica, type

```
Install[LinkConnect["linkname"]]
```

where `linkname` is the chosen name, to establish the connection. In this way, `mlprog` can also be started in the debugger, which is the routine way of debugging MathLink programs.

If the link target is on another machine, start the MathLink program with

```
./mlprog -linkname port \
         -linkprotocol TCPIP
```

where `port` is an integer larger than 1024 (a port number not in the reserved range) and connect from Mathematica via

```
Install[LinkConnect["port@host",
  LinkProtocol -> "TCPIP"]]
```

3. Linking

Unix (Linux, MacOS, ...) linkers are one-pass linkers (except when linking shared libraries). What this means is that if library `libneed.a` references symbol X and `libprovide.a` provides X, the following command nevertheless fails to resolve X:

```
cc ... -lprovide -lneed
```

because the linker does not 'go back' to earlier libraries. If necessary, `-lprovide` has to appear several times on the command line.

When providing ready-made executables to the general public, I strongly advertise statically linking the executable as far as possible (`-static` in gcc, `-st` in mcc). A statically linked 32-bit x86 Linux executable, for example, should run on essentially all Linux flavours currently available and will likely do so 10 years from now.

Besides, executables dynamically linked with the MathLink libraries require the MathLink library directory to be included on the library path (`LD_LIBRARY_PATH` or `/etc/ld.so.conf`) when invoked outside of Mathematica or in Mathematica versions before 6.

On MacOS and Windows it is not possible to statically link the system libraries, as e.g. Apple reserves the right to change their system library in an upward-compatible way. One should try to statically link at least the external libraries of the compiler(s) used, such as Fortran run-time libraries, so as to make the MathLink executable independent of those compilers. In the case of gcc, replace `-static` by `-static-libgcc`.

4. The `mcc` Compiler

The MathLink C compiler, `mcc`, is a shell script which supposedly replaces `cc`, the C compiler, in MathLink applications using approximately the same command line. Even after many e-mails to Wolfram Support, `mcc` still fails to take care of library ordering, however.

Luckily, `mcc` observes the `CC` and `CXX` environment variables, making it possible to substitute the actual C/C++ compilers with a shell script that corrects the library ordering. Such a script would look like

```
 1 #! /bin/sh
 2 # script to compile C programs that are
 3 # linked against Fortran libraries
 4
 5 args=
 6 objs=
 7 ldflags=
 8 fldflags=
 9 compileonly=
10
11 cc="${REALCC:-cc}"
12 cxx="${REALCXX:-c++}"
13 test `basename $0` = f++ && cc="$cxx"
14
15 while test $# -gt 0 ; do
16   case "$1" in
17   -st | -b32 | -b64)
18         ;; # ignore mcc-specific flags
19   -arch)
20         shift ;;
21   -lstdc++)
22         cc="$cxx" ;;
23   -[Ll]* | -Wl*)
24         ldflags="$ldflags '$1'" ;;
25   *.tm.o)
26         objs="'$1' $objs" ;;
27   *.a | *.o | *.so)
28         objs="$objs '$1'" ;;
29   *.cc)
30         args="$args '$1'"
31         cc="$cxx" ;;
32   -c)
33         compileonly="-c" ;;
34   -o)
35         args="$args -o '$2'"
36         shift ;;
37   *)
38         args="$args '$1'" ;;
39   esac
40   shift
41 done
42
43 eval "set -x ; exec $cc $args \
44 ${compileonly:-$objs $ldflags $fldflags}"
```

Lines 5–12 initialize a few variables. Since the environment variable `CC` is obviously taken, we need `REALCC` and `REALCXX` as replacements for choosing the actual C and C++ compilers, respectively.

This script is named `fcc` ('C compiler for linking with Fortran') and is symlinked to `f++`. When invoked as `f++`, the C++ compiler is taken as default (line 13).

The `while` loop starting in line 15 runs over all arguments and categorizes them: object files into `objs`, libraries and linker flags into `ldflags`, a `-c` into `compileonly`, all other arguments (e.g. source files, compiler flags) into `args`.

An `-arch x` argument is removed (line 17–18) because `mcc` tries to build the executable for all admissible platforms, e.g. on Mac for both Intel and PPC, and this conflicts during linking if an external library does not contain object code for all the given architectures. An explicit `-lstdc++` has to be removed likewise (line 21–22) as it would render a possible `-static-libstdc++` flag useless. The explicit (re-)quoting and the final `eval` (line 43) are necessary to treat arguments with spaces properly.

Finally, the actual compiler is invoked with the arguments in the proper order (lines 43–44). If `compileonly` is set, the object files and linker flags are omitted. The `set -x` makes the shell echo the actual compiler command line, which is a very good diagnostic if anything goes wrong.

The seemingly unused variable `fldflags` can be modified to contain external libraries needed for linking with the object files of a particular compiler, e.g. `-lgfortran` in the case of gfortran (see App. B). In this way, `fcc` serves also as a compile script for non-MathLink programs linked with code from another compiler.

The invocation of `mcc` must now include the definition of `CC` and `CXX`. Though `fcc` acts largely transparently, it is probably not a good idea to set `CC` and `CXX` permanently as they are understood by several tools. In a makefile, the following syntax might be used:

```
FCC = path/to/fcc
mlprog: mlprog.tm
   REALCC="$(CC)" REALCXX="$(CXX)" \
   CC="$(FCC)" CXX="$(FCC)" \
   mcc -o mlprog $(CFLAGS) mlprog.tm \
     $(LDFLAGS) $(LIBS)
```

5. Finding `mcc`

Except on Linux, `mcc` is not installed in a location which is on the `PATH`, so we have to find it first. To this end we make a script of the same name, `mcc`, and place that in a directory which we append to the `PATH`. If a true `mcc` is on the `PATH`, as in Linux, this gets called. If not, the substitute script gets called.

The substitute script first dispatches to the appropriate OS-specific routine:

```
case `uname -s` in
Darwin) macmcc "$@" ;;
CYG*) cygmcc "$@" ;;
*) defaultmcc "$@" ;;
esac
```

Since a naive `find` over the entire hard disk is not feasible, we resort to heuristics: we search for the Mathematica Kernel in a list of typical locations and run that to determine the `$TopDirectory`, which is where the copy of Mathematica to which that Kernel belongs is installed. This is done by the `sdkpath` function (described below) which receives the name of the Kernel (argument 1) and the list of possible locations (arguments 2–end).

On MacOS that is all, i.e. once the Mathematica directory is known, we start `mcc` from that location and exit:

```
macmcc() {
  sdkpath MathKernel \
    {/Applications,$HOME/Desktop}/\
Mathematica*/Contents/MacOS
  exec "$sdk/mcc" "$@"
}
```

The `cygmcc` function for Windows has to do quite a bit more and will be discussed in Section 6 on Cygwin below. The `defaultmcc` function acts as a catch-all. It tries a few standard places in case e.g. some ignorant system administrator installed Mathematica in a location not on the path:

```
defaultmcc() {
  sdkpath math \
    /usr/local/bin \
    /usr/local/Wolfram/bin \
    /usr/local/Wolfram/\
Mathematica/*/Executables \
    /opt/Wolfram/bin \
    /opt/Wolfram/\
Mathematica/*/Executables
  exec "$sdk/mcc" "$@"
}
```

But now the `sdkpath` function:

```
sdkpath() {
  mathcmd="$1"
  shift
  mathcmd=`IFS=:
    PATH="$PATH:$*" which $mathcmd`

  eval `"$mathcmd" -run '
Print["sysid=\"", $SystemID, "\""];
Print["topdir=\"", $TopDirectory, "\""];
Exit[]
  ' < /dev/null | tr '\r' ' ' | tail -2`

     # check whether Cygwin's dlltool
     # can handle 64-bit DLLs
  test "$sysid" = Windows-x86-64 && {
    ${DLLTOOL:-dlltool} --help | \
    grep x86-64 > /dev/null || \
      sysid=Windows
  }

  topdir=`cd "$topdir" ; echo $PWD`

45   for sdk in \
46     "$topdir/SystemFiles/Links/MathLink/\
47 DeveloperKit/$sysid/CompilerAdditions" \
48     "$topdir/SystemFiles/Links/MathLink/\
49 DeveloperKit/CompilerAdditions" \
50     "$topdir/AddOns/MathLink/\
51 DeveloperKit/$sysid/CompilerAdditions"
52   do
53     test -d "$sdk" && return
54   done
55
56   echo "MathLink SDK not found" 1>&2
57   exit 1
58 }
```

Lines 24–27: Search for the specified Mathematica Kernel in the given list of locations by temporarily adding them to the `PATH`. Then run the resulting Kernel (line 29) and have it print out (lines 30–31) the `$SystemID`, a string identifying the platform, and `$TopDirectory`, the installation directory of that Mathematica copy.

On Windows-64, downgrade the `$SystemID` to 32-bit (line 37–41) if Cygwin's `dlltool` professes not to handle 64-bit libraries (see Sect. 6 below). Canonicalize `$TopDirectory` (line 43), mainly to get rid of Windows-style path names (`C:\x\y`).

Go through the list of typical places underneath `$TopDirectory` (lines 46–51) and return the first match (line 53). If there is no match, exit with an error (lines 56–57).

6. Cygwin

Cygwin is a Unix-like environment running natively (i.e. not virtualized) on Windows. In the sense in which the colloquial 'Linux' is more correctly GNU/Linux (GNU utilities, Linux kernel), Cygwin might be termed GNU/Windows. It is likely the least painful way of porting programs from Unix to Windows.

The installation is straightforward even for non-expert users (`http://cygwin.com`), though one should review the package selection as many tools obviously necessary to build MathLink programs, such as `gcc`, `g++`, `make`, are not included in the default setup.

The `cygmcc` function starts in much the same way as the `macmcc` function, by finding the path to Mathematica:

```
59 cygmcc() {
60   sdkpath math \
61     "‘cygpath ’$ProgramW6432’‘/\
62 Wolfram Research/Mathematica"/* \
63     "‘cygpath ’$PROGRAMFILES’‘/\
64 Wolfram Research/Mathematica"/*
```

`$ProgramW6432` and `$PROGRAMFILES` point to the 64- and 32-bit Applications directory on Windows ("`C:\Program Files`"), respectively. `cygpath` turns Windows-style into Unix-style path names (`/cygdrive/c/Program Files`).

From Mathematica 7 on, the MathLink SDK does include Cygwin libraries and tools, but these are broken so badly (e.g. filename quoting) that not even simple programs can be built successfully. For this reason we skip the `cygwin` directory underneath `$sdk` and move to one of the `mldev` directories (e.g. `mldev32`):

```
65 for sdk in "$sdk"/m* ; do
66   break
67 done
```

The chosen `mldev` directory contains native Windows DLLs. To make Cygwin's `ld` accept these, one needs to create a so-called 'library stub' which contains the library's location and symbol table. Cygwin's `dlltool` provides this information and can be used e.g. as follows:

```
68 cache=MLcyg-cache
69 test -d $cache || mkdir $cache
70
71 MLversion=3
72 for OSbits in 32 64 ; do
73   dllname=ml${OSbits}i$MLversion
74   libname="$sdk/lib/${dllname}m.lib"
75   test -f "$libname" && break
76 done
77
78 lib="$cache/${dllname}m"
79 test -f "$lib.a" || {
80   ( echo "EXPORTS"
81     ${NM:-nm} -C --defined-only \
82       "$libname" | \
83       awk ’/ T [^.]/ { print $3 }’
84   ) > "$lib.def"
85   ${DLLTOOL:-dlltool} -k \
86     --dllname "$dllname.dll" \
87     --def "$lib.def" \
88     --output-lib "$lib.a"
89 }
```

Creating a library stub is a one-time process, so we can store it in a cache directory, here named `MLcyg-cache` (lines 68–69). Lines 72–76 check which MathLink library (32- or 64-bit) exists in the given path. If the corresponding library stub is not yet in the cache (line 79) the relevant information is extracted using `nm` and `dlltool` and the library stub is created (lines 80–88).

Finally, we have to emulate `mcc`:

```
90  tmp=
91  args="-DWIN$OSbits -I'$sdk/include'"
92  for arg in "$@" ; do
93    case "$arg" in
94    *.tm)
95        cp "$arg" "$arg.tm"
96        "$sdk"/bin/mprep -lines \
97          -o "$arg.c" "$arg.tm"
98        tmp="$tmp '$arg.c' '$arg.tm'"
99        args="$args '$arg.c'" ;;
100   *)
101       args="$args '$arg'" ;;
102   esac
103 done
104
105 trap "rm -f $tmp" 0 1 2 3 15
106 eval "set -x ; \
107   ${CC:-gcc} $args $lib.a -mwindows"
```

All `.tm` files are converted to C code using the `mprep` utility (lines 96–97). Note the explicit `cp` in line 95 which is necessary in case the original `.tm` file is a symlink (symlinks are understood by Cygwin only, not by native Windows programs such as `mprep`). The temporary files are added to `tmp` (line 98) and scheduled for deletion at exit with the `trap` statement in line 105. The `-mwindows` flag (line 107) adds the Windows system libraries.

A somewhat regrettable feature is that executables produced using Cygwin compilers and libraries require Cygwin (or at least `cygwin1.dll`) to be installed also on the system the executable is run on. If one does not need `fork`, `wait`, or the `pthread_*` functions (and some few more), it is possible to build executables on Cygwin that do not depend on any Cygwin runtime libraries. There are two ways:

- Either install the `gcc-3` packages (including `g77-3` if necessary) and run with

  ```
  CC=gcc-3 CFLAGS=-mno-cygwin
  FC=g77-3 FFLAGS=-mno-cygwin
  ```

  This is restricted to 32-bit, however.

- Or, install the `mingw` packages for the desired target (`i686` or `x86_64`) and work with the targeted versions of compilers and binutils, i.e. use *h*-`gcc`, *h*-`nm`, *h*-`dlltool` instead of `gcc`, `nm`, `dlltool`, etc., where *h* is the 'host triplet', e.g. `i686-pc-mingw32` or `x86_64-w64-mingw32`.

7. Strings in MathLink

MathLink has an impressive number of string-related functions which differ mainly in how non-ASCII characters are treated. We will concentrate here on two methods only, character strings and byte strings.

If the string exchange is known to be in pure ASCII, both methods are pretty much equivalent in functionality and one can select the more convenient one.

Strings from Mathematica should always be considered immutable (`const`). If it becomes necessary to modify such a string, make a copy before. A string read with one of the `MLGet*String` functions needs to be de-allocated after use with `MLRelease*String`.

7.1. Character strings

Character strings are ordinary null-terminated 7-bit C strings. Non-ASCII characters (not just accented characters but greek letters, mathematical symbols, special punctuation marks, etc.) are encoded by Mathematica as escape sequences, such as `\[Alpha]` ('α', 8 characters).

Character strings are the method of choice if one receives a string from Mathematica, a function or symbol name, say, and uses that string only in the communication with Mathematica in a transparent way, i.e. without 'looking into' it.

For example, the MathLink function template might include a user-defined inspector function for debugging (`Identity` if none) to be wrapped around (a part of) the result. The name of that function would have no meaning to the MathLink program other than that it gets sent back around the right expression. In this case the MathLink programmer would be happy to let Mathematica encode the string in whatever way it needs to recognize it as the same later.

The MathLink API functions for character strings are `MLGetString` and `MLPutString`. In the template definition use `String` or `Symbol` (depending on the desired pattern matching) and in the function declaration `const char *`.

7.2. Byte strings

Byte strings are 8-bit character arrays plus a length, i.e. are not terminated with a special character. This is conceptually very similar to a Fortran string (see App. A.6). In C one would operate on them with the `mem*` family of functions (`memcpy`, `memchr`, etc.).

The advantage of byte strings is that the characters map 1:1 onto C or Fortran strings (no escape sequences, no variable-length characters as in UTF-8). On the other hand, a single byte is not wide enough to hold an arbitrary Mathematica character and thus the `MLGetByteString` function has one argument specifying an 8-bit substitute for characters wider than 8 bits.

The MathLink API functions for byte strings are `MLGetByteString` and `MLPutByteString`. In the template definition use `ByteString`, in the function declaration `const unsigned char *, const int`.

8. `stdout` and `stderr`

The typical physicist's practice of writing error, warning, progress messages etc. on `stdout` (file descriptor 1, Fortran unit 6) or `stderr` (file descriptor 2, Fortran unit 0) is not very effective in a MathLink program, for `stdout` is suppressed entirely and `stderr` appears on the terminal only if running the Mathematica Kernel directly (no Frontend). Silently dropping messages can be anything from not helpful to outright dangerous.

An easy though somewhat clumsy solution is to redirect `stdout` and/or `stderr` to a file instead. More elegant is to capture the output and send it to Mathematica for display. Even more flexible is to let the user specify a file into which to write the output and recognize e.g. `"stdout"` as a special name in the case of which the output is sent to Mathematica.

Capturing the output requires a second thread which reads the `stdout` output generated by the main thread through a pipe and sends it to the Kernel. First we need a few global variables and a copy of the original `stdout` file descriptor 1:

```
 1 static int stdoutorig;
 2 static int stdoutpipe[2];
 3 static pthread_t stdouttid;
 4 static int stdoutthr;
 5
 6 static inline void IniRedirect() {
 7   int fd;
 8   do fd = open("/dev/null", O_WRONLY);
 9   while( fd <= 2 );
10   close(fd);
11   stdoutorig = dup(1);
12 }
13
14 int main(int argc, char **argv) {
15   IniRedirect();
16   return MLMain(argc, argv);
17 }
```

So as not to overlap with standard file descriptors in the I/O redirection later, we open `/dev/null` as many times as it takes to obtain a file descriptor not in 0, 1, 2 (lines 8–10), in case Mathematica closed any of them.

Capturing is set off by invoking `BeginRedirect` at the beginning of the MathLink function in which the messages are generated:

```
18 static inline void BeginRedirect() {
19   stdoutthr = pipe(stdoutpipe) != -1 &&
20     pthread_create(&stdouttid, NULL,
21       MLstdout, NULL) == 0;
22   if( !stdoutthr ) stdoutpipe[1] = 2;
23   dup2(stdoutpipe[1], 1);
24   close(stdoutpipe[1]);
25 }
```

In the (somewhat hypothetical) case that `pipe` or `pthread_create` fail, we fall back on the original `stderr` (2) so that at least on a terminal there is a chance of seeing the output (line 22). Alternately one could exit with an error code here. Then we connect `stdout` (1) to the write end of the new pipe (lines 23–24).

The thread function `MLstdout` collects all output in a buffer and sends it to Mathematica for display only at the end. Depending on the typical running time and message volume of the underlying function one could also send output back line by line, when a certain buffer volume is reached, or similar.

```
26 static void *MLstdout(void *dummy) {
27   static unsigned char *buf = NULL;
28   static long size = 0;
29   enum { unit = 10240 };
30   long len = 0, n = 0;
31 
32   do {
33     len += n;
34     if( size - len < 128 )
35       buf = realloc(buf, size += unit);
36     n = read(stdoutpipe[0],
37       buf + len, size - len);
38   } while( n > 0 );
39 
40   if( len ) {
41     MLPutFunction(stdlink,
42       "EvaluatePacket", 1);
43     MLPutFunction(stdlink,
44       "WriteString", 2);
45     MLPutString(stdlink, "stdout");
46     MLPutByteString(stdlink, buf, len);
47     MLEndPacket(stdlink);
48     MLNextPacket(stdlink);
49     MLNewPacket(stdlink);
50   }
51   return NULL;
52 }
```

The output buffer `buf` stays allocated, it usually needs just a few kilobytes. If space runs low, `buf` grows in units of 10 kbytes (lines 34–35).

Due to the static variables, `MLstdout` is not reentrant but this is not really necessary, either: to redirect e.g. `stderr`, too, just connect descriptor 2 to `stdoutpipe[1]` as well, similar to lines 23–24, rather than create another thread.

The `EndRedirect` function finally cancels the redirection (line 55) which automatically closes the pipe and causes the reader thread to wrap up (lines 40–51) such that it can be joined (line 57).

```
53 static inline void EndRedirect() {
54   void *ret;
55   dup2(stdoutorig, 1);
56   if( stdoutthr )
57     pthread_join(stdouttid, &ret);
58 }
```

Mathematica is still waiting for the function results by the time `EndRedirect` is called, which is why the `WriteString` (line 43) has to be issued out-of-band in an `EvaluatePacket` (line 41). The final `MLNextPacket` and `MLNewPacket` (lines 48–49) discard the `WriteString` return value (`Null`).

Note that the transfer of redirected output to Mathematica happens in a well-defined sequence, i.e. there is no race condition here: it is triggered by the `dup2` in line 55 and guaranteed to be finished by the time `pthread_join` returns. The function's return value is transmitted strictly after the `EndRedirect`, so there is no way the communication with Mathematica could be upset.

If such a sequence cannot be taken for granted, e.g. if `MLstdout` is rearranged to send output after each `read` while the main thread engages in more communication with the Mathematica Kernel in the meantime, it must be enforced using a mutex around the transmission of the EvaluatePackets (e.g. around lines 41–49 and corresponding ones in the main thread).

A final word on buffering: Fortran maintains its own I/O buffers over which the C program has no control, thus the message output may not be completely transferred by the time the Fortran routine returns. The simplest and most portable solution is to explicitly flush unit 6 before cancelling the redirection. Fortran's `flush` subroutine is considered an intrinsic by some compilers and therefore cannot portably be called from C directly, so a trivial wrapper is necessary:

```
subroutine fortranflush()
call flush(6)
end
```

Add `fortranflush_();` to `EndRedirect` before the `dup2` statement and `fortranflush.o` to the `mcc` command line. Alternately, the call to `flush` may be added to each Fortran routine directly.

For pure C functions a simple `fflush(stdout)` suffices, of course.

9. Summary

The preceding sections have collected the information needed to get MathLink to work portably across at least the more popular platforms currently available, Linux, Mac OS, and Windows.

It is not easy to avoid the impression that this requires more workarounds than actual code. The quality of the MathLink SDK has improved somewhat over the versions but is still far from perfect. On Mac OS and Windows it would in fact seem that MathLink virtually eschews any cooperation with the user, which may be connected to the fact that these are commercial platforms where users typically do not ('are not meant to') build their own executables.

On the positive side it has to be pointed out that providing Mathematica connectivity to a piece of C or Fortran code opens up fantastic new possibilities for interactive use (think of functions like `Manipulate`) and combination with Mathematica's sophisticated functions (if unconvinced, try doing a `ContourPlot` in Fortran, for example). For users not sufficiently familiar with C or Fortran, it makes these functions available at all.

The only significant work is writing interfacing code in a `.tm` program. Apart from that, a package author only needs to add the `fcc` and `mcc` substitute scripts and tweak the makefile as described in Sect. 4. All things considered, this is a fairly moderate effort.

The shell scripts together with demo code are available from `http://feynarts.de/mathlink/`. They can be witnessed 'in action' in the packages LoopTools [1], Cuba [2], FeynHiggs [3], Diag [4], and in FormCalc's Mathematica interface [5].

Comments, improvements, and bug-fixes are welcome at `hahn@feynarts.de`.

A. C–Fortran interfacing

MathLink's native tongue is C so in order to link Mathematica with Fortran code one needs to know at least the basics of C–Fortran interfacing. Fortran is less flexible in its calling conventions, thus in general the C program has to adapt, not the Fortran program.

A.1. Function names

Fortran names (subroutines, functions, common blocks, block data) are lowercased and an underscore is appended by the time they end up in the object file. The very few compilers which do not add an underscore (HP-UX's fort77, for example) should largely be extinct by now. Steer clear of underscores in Fortran names, as compilers have different ways of mangling those, e.g. some compilers add *two* underscores if the Fortran name already contains one (cf. gfortran's `-fsecond-underscore` option).

Modern C compilers require prototypes for all external routines. In C++, the prototype must be wrapped in `extern "C" {...}` to suppress C++ name mangling.

Example: `subroutine FOO` in Fortran becomes `void foo_()` in C.

In case of problems, check the spelling of the symbols as visible to the linker by using `nm` on the Fortran object file (and, for comparison, possibly the C object file as well).

A.2. Function arguments

Fortran always calls by reference, i.e. passes a pointer, not the variable itself (exceptions only when using the much-deprecated `%VAL`).

Example:

```
subroutine foo(i)
integer i
```

becomes `void foo_(int *i)` in C – mind the `*`.

Hint 1: Since Fortran passes by reference, and rather indiscriminately so (for example, many compilers silently ignore mismatches between formal and actual arguments even within the same source file), it is an extremely good idea to set up prototypes as strictly as possible, including `const` in all places where the argument should not be modified (even though Fortran has no way of controlling this).

Hint 2: For portability between C and C++, use the `__cplusplus` preprocessor variable around the `extern "C"` bits:

```
#ifdef __cplusplus
extern "C" {
#endif

void a0_(double *res, const double *m);
(... possibly more prototypes ...)

#ifdef __cplusplus
}
#endif
```

Hint 3: It makes life a lot easier to wrap Fortran functions in C inline functions, with proper C calling conventions. The wrapper function can be used just as a regular C function and 'inline' means there is no extra calling overhead, e.g.

```
static inline double A0(const double m) {
  double res;
  a0_(&res, &m);
  return res;
}
```

Take care that the `static` attribute pertains to the wrapper function only, as the Fortran function most certainly does not have file scope but comes from an external library.

A.3. Return values

Prefer subroutines over functions in Fortran, especially if the return value is not an `integer` or `double precision` (`real*8`). Conventions vary most notably for `double complex` functions. If a function is required on the Fortran side e.g. because of an API, add a subroutine wrapper:

```
      double complex foo(args...)
      ...
      end

      subroutine foowrapper(res, args...)
      double complex res, foo
      external foo
      res = foo(args...)
      end
```

A.4. Data types

Most scalar types have an obvious counterpart in C, e.g.

- `integer` (`integer*4`) — `int`,
  `integer*2` — `short`,
  `integer*8` — `long long int`,
- `double precision` (`real*8`) — `double`,
  `real` (`real*4`) — `float`,
- `double complex` (`complex*16`)
  — `double complex` (C99),
  — `struct { double re, im; }` (C89),
  — `std::complex<double>` (C++),
- `character` — `char`
  (but for strings see below).

There is no portable equivalent of `logical` in C, however; it is better to use an integer in Fortran instead. If you must use `logical`, interface with an `int` and test for the lowest bit only.

These correspondences can also be coded with `typedef` statements. Not only does this make the Fortran types stand out visually, but the compiler will automatically add casts or warn about incompatibilities when mixing with other C types. Note the `const` versions for strict prototyping.

```
typedef int INTEGER;
typedef const INTEGER CINTEGER;
typedef double REAL;
typedef const REAL CREAL;
typedef struct { REAL re, im; } COMPLEX;
typedef const COMPLEX CCOMPLEX;
typedef char CHARACTER;
typedef const CHARACTER CCHARACTER;
```

For portability between C and C++, for example in header files, one may want to provide wrapper code for the homogeneous treatment of complex numbers:

```
#define Real double
#define ToReal(r) (r)

#ifdef __cplusplus

#include <complex>
typedef std::complex<Real> Complex;
```

```
#define ToComplex(c) \
  Complex(ToReal((c).re), ToReal((c).im))
#define ToComplex2(r,i) \
  Complex(r, i)
#define Re(x) std::real(x)
#define Im(x) std::imag(x)

#else

#include <complex.h>
typedef Real complex Complex;
#define ToComplex(c) \
  (ToReal((c).re) + I*ToReal((c).im))
#define ToComplex2(r,i) \
  (r + I*(i))
#define Re(x) creal(x)
#define Im(x) cimag(x)

#endif
```

Referring to reals indirectly through the `Real` and `ToReal` macros is useful for switching to a different precision (`float`, `long double`; see App. C).

A.5. Common blocks

Common blocks map onto C structs, with the members in the same order. The `struct` should be declared `extern` to prevent instantiation in C. In C++, `extern "C"` must be used in addition to the `extern` (storage class modifier) because of name mangling, as with the function prototypes.

Padding might be an issue if common members are ordered unsuitably for alignment, e.g.

```
   integer i
   double precision r
   common /c/ i, r
```

is not correctly aligned for a 64-bit architecture because the integer is a 32-bit quantity. Ideally, the common block should be reordered (widest types first, e.g. `double complex` before `double precision` before `integer` before `character`). If this is not possible due to API requirements or similar, wrap the common definition in

```
#pragma pack(push, 1)
extern struct {
  int i;
  double r;
} c_;
#pragma pack(pop)
```

This switches off padding (in gcc at least) but carries a performance penalty, and on RISC platforms such as the Alpha likely triggers unaligned-access exceptions (in both Fortran and C).

A.6. Strings

There are no strings in Fortran, only character arrays (padded with spaces as necessary), and these are handled specially by the compiler, i.e. differently from other arrays.

The C function gets called with two arguments for every Fortran string: a `char *` pointer to the characters, in the same place as the string argument in Fortran, and a `const int` following the Fortran argument list. Example:

```
   subroutine strfoo(s1, i1, s2, i2)
   character*(*) s1, s2
   integer i1, i2
```

in C becomes

```
void strfoo_(char *s1, int *i1,
  char *s2, int *i2,
  const int s1_len, const int s2_len)
```

Fortran strings are *not* null-terminated and it is in general, but in particular for functions invoked with string literals as arguments, not advisable to write a zero-byte into the Fortran string *in situ* and hope it won't disturb Fortran later.

The two 'clean' options are: use the `mem*` family of functions (`memcpy`, `memchr`, etc.) which take a length argument and do not require a terminating zero-byte, or null-extend the string in allocated memory. The latter is particularly simple in C99/C++ where space can easily be allocated on the stack (and is automatically de-allocated when the object goes out of scope), e.g.

```
void sf_(char *s, const int s_len) {
  char sn[s_len + 1];
  memcpy(sn, s, s_len);
  sn[s_len] = 0;
  ...
}
```

Note that, while the string `sn` is correctly null-terminated now and may be worked on with the `str*` functions, it may well include trailing spaces as Fortran indicates only the allocated size in `s_len`, not the actual length (sans trailing spaces) of the character array.

In the opposite direction, invoking a Fortran function with a string argument is straightforward, e.g.

```
fortfoo_(s, strlen(s));
```

When returning a string of a given length, i.e. (over)writing one of the subroutine's string arguments, the unused characters should be filled with spaces, as in:

```
void cfoo_(char *s, const int s_len) {
  ...
  n = strlen(result);
  if( n >= s_len )
    memcpy(s, result, s_len);
  else {
    memcpy(s, result, n);
    memset(s + n, ’ ’, s_len - n);
  }
}
```

B. Required flags and libraries

The rules above are sufficient to obtain object-level compatibility, i.e. making C and Fortran routines talk to each other. For successful linking one has to add the Fortran compiler's run-time libraries (e.g. I/O, extended math) to the C command line as well.

To determine the necessary flags, the Fortran compiler is run in verbose mode on a test program and all flags relevant to linking are collected from the output. Here is how:

```
 1 getldflags() {
 2   ldflags="$LDFLAGS"
 3   while read line ; do
 4     set -- ‘echo $line | tr ’:,()’ ’    ’‘
 5     case $1 in
 6     */collect2* | */ld* | ld*) ;;
 7     *) continue ;;
 8     esac
 9     while test $# -gt 1 ; do
10       shift
11       case $1 in
12       *.o | -lc | -lgcc*)
13         ;;
14       -l* | -L* | *.a)
15         ldflags="$ldflags $1" ;;
16       -Bstatic | -Bdynamic | *.ld)
17         ldflags="$ldflags -Wl,$1" ;;
18       /*)
19         ldflags="$ldflags -L$1" ;;
20       -rpath*)
21         ldflags="$ldflags -Wl,$1,$2"
22         shift ;;
23       -dynamic-linker)
24         shift ;;
25       esac
26     done
27   done
28   echo $ldflags
29 }
```

Lines 5–8 select only the linker command lines (note that gcc uses `collect2`, not `ld`). The `while` loop starting in line 3 picks out the relevant items from the ld command line.

The above shell function is used as in

```
LDFLAGS=‘f77 -v -o test test.f 2>&1 | \
  getldflags‘
```

where `test.f` is a simple test program, preferably one producing output so that the I/O libraries are referenced, e.g.

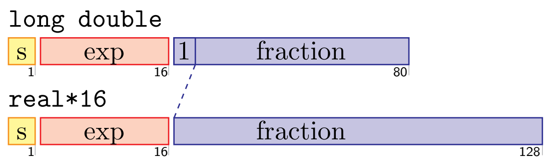
```
  program hw
  print *, "Hello World"
  end
```

C. Extended precision

MathLink provides a data type by the name of `Real128`. Contrary to expectations, this does not represent a genuine quadruple-precision 128-bit floating-point number but ‘only’ C's `long double` data type which on Intel x86 hardware is usually the 80-bit ‘extended-precision’ float, even though C stores this in 12 (in 32-bit) or 16 bytes (in 64-bit mode) for alignment.

Those few Fortran compilers that offer higher than double precision typically have the `real*16` data type which implements true 128-bit precision, albeit in software emulation (i.e. slow).

The two formats are actually quite similar, the main difference being that the most-significant bit of the (normalized) fraction is implicit in `real*16` and explicit in `long double` (which can thus also represent denormalized fractions):



Conversion can be achieved e.g. as follows (note that this code is specific to Intel x86 hardware):

```
 1 #pragma pack(push, 1)
 2 typedef union {
 3   long double r10;
 4   struct {
 5     unsigned long long frac;
 6     unsigned short exp;
 7   } i10;
 8   struct {
 9     char zero[6];
10     unsigned long long frac;
11     unsigned short exp;
12   } i16;
13   unsigned long long i8[2];
14 } real16;
15 #pragma pack(pop)
16
17 static inline real16 ToREAL(long double r) {
18   real16 new;
19   new.i8[0] = 0;
20   new.i16.frac =
21     ((real16 *)&r)->i10.frac << 1;
22   new.i16.exp = ((real16 *)&r)->i10.exp;
23   return new;
24 }
25
26 static inline long double ToReal(real16 r) {
27   real16 new;
28   const long long z = r.i16.frac |
29     (r.i16.exp & 0x7fff);
30   new.i10.frac = (r.i16.frac >> 1) |
31     ((z | -z) & 0x8000000000000000);
32   new.i10.exp = r.i16.exp;
33   return new.r10;
34 }
35
36 #define Real long double
37 typedef real16 REAL;
```

One needs to apply `ToREAL` when transferring arguments from C to Fortran, and `ToReal` when transferring from Fortran to C. Defining `Real` as in line 36 automatically upgrades the `Complex` data type declared in App. A.4, too.

Double-precision Fortran code can be upgraded to quadruple precision almost automatically if a few rules are observed, through compiler flags like `-r16` (ifort). This can be very helpful to find out whether a wrong result comes from a programming mistake (bad algebra) or from e.g. accumulation of round-off errors (bad numerics). The rules:

- Use `double precision`/`double complex`, not `real*8`/`complex*16`, the latter are not affected by the automatic `-r16` promotion.
- Use generic functions only, e.g. use `abs`, not `dabs`. To force a type conversion, make an explicit cast, e.g. use `sqrt(DCMPLX(x))`, not `cdsqrt(x)` (on `DCMPLX` see next item).
- Despite their name, `real`, `imag`, `conjg`, and `cmplx` are not generic but single-precision real (`real*4`) functions and thus not promoted automatically, either.

  Solution: use the explicitly double-precision functions `DBLE`, `DIMAG`, `DCONJG`, and `DCMPLX` in the original code. Choose identical capitalization (e.g. all-caps as given here so as to stand out) and have the preprocessor substitute the double- by quadruple-precision variants by adding the following flags to the command line:

  ```
  -DDBLE=QEXT -DDCONJG=QCONJG \
  -DDCONJG=QCONJG -DDCMPLX=QCMPLX
  ```

  Note: The file extension should be `.F` in this case, not `.f`, otherwise the preprocessor will not be invoked automatically.

REFERENCES

1. T. Hahn, M. Perez-Victoria, Comput. Phys. Commun. **118** (1999) 153–165 [hep-ph/9807565].
2. T. Hahn, Comput. Phys. Commun. **168** (2005) 78–95 [hep-ph/0404043].
3. M. Frank, T. Hahn, S. Heinemeyer, W. Hollik, H. Rzehak, G. Weiglein, JHEP **0702** (2007) 047 [hep-ph/0611326].
4. T. Hahn, physics/0607103.
5. T. Hahn, Comput. Phys. Commun. **178** (2008) 217 [hep-ph/0611273].