MATLAB for Lab Automation:

A Guide to Interfacing with

MATLAB

Brian Rasnow*

Received …

*Brian Rasnow, Ph.D., Amgen Inc., Research Automation Technologies, One Amgen Center Dr. 29-1-A, Thousand Oaks, CA 91320 (brasnow@amgen.com)

Table of Contents

1. Abstract.

Integrating an automated research platform involves building interfaces between numerous disparate devices (e.g., motion controllers, liquid handlers, detectors, etc.), and writing programs that couple the operation and states of these devices to perform unified high-level tasks such as drug screening experiments. MATLAB can accelerate the design and construction of complex apparatus by providing an integrated environment for communication, control, and data acquisition, analysis, and visualization. Features useful for research automation and automated platform control include:

- Rapid application development tools
- ASCII script interpreter engine
- Powerful methods for data processing, visualization and exploratory analysis
- Full featured programming language with procedural structure with simple syntax
- Extensible grammar using scripts, functions, and C- routines (called MEX functions)

This tutorial focuses on how to write and debug C-language MEX extensions to MATLAB 5.2 using Windows 95 and Microsoft Visual C++5 (MSVC) for interfacing with hardware (e.g., serial ports, CCD cameras, and i/o boards). It assumes a working knowledge of C (including structures and pointers), MATLAB, and Windows95. The following two sections describe how to set up MATLAB, MSVC, and Lemmy (a vi-like text editor) into a unified development environment to facilitate building MEX files; and how to write, compile, and debug MEX files. The final section presents several MEX file examples.

2. Installation, Configuration, and Documentation Sources

2.1. MATLAB

2.1.1. Installation and Documentation

Install all the on-line help files. Also required for MEX development is the API module. Note that MATLAB and MSVC interact to make MEX files so they need to know each other's paths. If you develop code for multiple machines, it is much easier if all the machines have MATLAB and MSVC in consistent paths, otherwise you will frequently be changing #include's in the MEX C files.
MEX support files and examples live in matlab\extern\. I suggest developing MEX files in a subdirectory of this path to facilitate opening header and example files. In fact, the header files are the most accurate documentation, as they contain prototypes of all interface functions, structure declarations, etc. The printed documentation for MEX extensions is the Application Program Interface Guide, but it is incomplete. Use the helpdesk command and click on the hypertext link for Application Program Interface for function listings and descriptions.

2.1.2. Search Path

When MATLAB launches, it reads the file: matlab\toolbox\local\matlab.rc, loads the search path from matlab\toolbox\local\pathdef.m (if it exists), and executes startup.m (if it exists). Typing foo on the command line causes the interpreter to look in the following order: for a workspace variable called foo, a built-in function foo, file foo.dll in the current directory, file foo.m in the current directory, and then traverse the explicit search path for foo.dll and foo.m. To access MEX (or .m) files outside the current directory, those directories should be added to the explicit search path. This can be done in a variety of ways (e.g., using the addpath function, editing pathdef.m, or using the GUI from the command window menu: File: Set Path …). Note that MATLAB functions and scripts are identified by their file names (so for example, file foo.m that contains as its first line: function a = bar(b) will be executed when foo is typed on the command line, and not when bar is typed). Also, if modifications to a file are not working, type which *filename*to make sure your file is the one being executed and not another one encountered earlier in the path.

2.1.3. MEX configuration

MEX files are compiled from the MATLAB command line using mex.m, which launches a Perl script (mex.bat) that in turn launches and instructs the MSVC compiler and linker to create the .dll MEX file. The current Perl script configuration file for the MSVC5 is available from the Mathworks website (www.mathworks.com) as MSVC50OPTS.BAT. Download this file, and save it as matlab\bin\mexopts.bat. Edit the first two set commands to define MATLAB and MSVC paths, e.g.,

```
set MATLAB=%MATLAB%
set MSVC_ROOT=c:\program files\devstudio
```

File mex.bat required some tinkering in addition to the top line:

```
set MATLAB=C:\MATLAB
```

The second line below has some changes:

```
if ($RC_LINKER =~ /\S/) {
    $rc_line = "$ENV{'MATLAB'}\\extern\\include\\mexversion.rc
$ENV{'MEX_NAME'}.$extension";
```

2.2. Microsoft Visual C++

MSVC is a feature-glutted, confusing environment with a seriously underpowered text editor. Metroworks Codewarrior maybe a better alternative, although MSVC was on hand and is therefore described. A minimal install is probably adequate for MEX file development in C. Documentation for writing simple C programs in MSVC is sparse, although there are volumes devoted to Microsoft Class Library and API exotica. Simple C programs like hello.c and shells to test MEX functions can be built using the Win32 Console Application template.

A couple of useful features and customization are the following. Right-clicking on a line of source lets you set or clear a breakpoint. An icon can be added to the debugging toolbar to skip to the source line with the cursor on it (enabling you for example to repeatedly execute a line without restarting the run). From the Tools:Customize… menu, select the Toolbar tab, activate the Debug toolbar, and drag the bold yellow arrow with blue curved arrow over it to the main toolbar.

2.3. Lemmy (vi) Text Editor

2.3.1. What difference does an editor make?

Programming (in any language) is facilitated by a capable text processor. Looping through the edit-compile-run-debug cycle of C and edit-run-debug cycle of MATLAB can be speeded up and tunnel carpal minimized with editor features such as rapid navigation and cursor control, powerful search and replace, indentation control, color-coding of comments, strings, and keywords, ctags, auto-parentheses balancing, etc. MATLAB and MSVC each come with integrated text editors that sport some of these features. The two editors that dominate the unix world, vi and emacs, are vastly more powerful (they compare to the integrated editors like Word compares to Notepad). Further discussion and executables can be found at http://www.cs.vu.nl/~tmgil/vi.html, where I downloaded a vi clone for Windows called Lemmy. Lemmy implements the cryptic but powerful vi command set, in addition to supporting multiple windows, mouse navigation, cut-and-paste (control-c, control-v), and color-coded text.

Neither vi nor emacs are easy to learn, and I doubt they can be learned by exploration alone. A good vi book is Lamb, “Learning the vi Editor”, O’Reily & Assoc. (1994). Many Unix and Linux books also have a chapter on vi.

2.3.2. Configuring Lemmy

After running the Lemmy installation program, select View:Properties… On the GUI Settings tab set the startup window size to something comfortable on a large monitor – I use 40 rows by 120 columns, and click the Save button. The Fonts and Colors tab is the quirkiest part of Lemmy, and took me lots of pounding and hacking to get a visible cursor on a white background. I chose Custom… color scheme and

set the background of everything except the cursor and highlight text to white. There is no MATLAB Parse Engine so I used Global Types and picked different foreground colors preprocessor, comment, keyword, and string. I gave cursor a red foreground and light blue background, but it took a few saves and tries to get that to stick (even though it says FG only for cursor, the BG color selection has an effect). Hit the Save button frequently. Next, on the vi tab, I set autoindent, magic, mesg, remap, showmatch, timeout, warn, wrapscan, report=5, shiftwidth=4, tabstop=4.

2.3.3. Using Lemmy from Windows, MATLAB, and MSVC

Windows can be configured to open *.m (and *.c and *.h files) with Lemmy as follows. From an Exploring window select View:Options…, select the File Types tab. Select from the list of file types M file, click on the Edit… button, and in the new dialog box click on its Edit… button, then enter or Browse… the path for Lemmy. Repeat for C Source files and C Header files if desired. Placing a Shortcut to Lemmy in both the C:\Windows\Start Menu and C:\Windows\SendTo directories is also helpful – the later allows you to right-click on any text file and open it with (SendTo) Lemmy.

MATLAB can be configured to launch Lemmy by default: from File:Preferences, under Editor Preferences click on the radio button next to the text field and enter or Browse the Lemmy path. Typing from the MATLAB command line:

```
» edit          % opens a new lemmy window
» edit startup  % opens startup.m (wherever it resides in the MATLAB search path)
» edit foo.c    % opens foo.c in the current directory
```

MSVC does not appear to be configurable to use external editors. However, when you edit source files in Lemmy and have the same file open in MSVC, you are notified when switching between programs that the file has been modified and prompted to reload the file. I generally use Lemmy for initial C source composition and major edits.

2.4. Windows 95 and NT

Windows 95 and NT differ fundamentally on how they permit user processes (e.g., programs such as MEX files) to communicate with hardware. Under NT (and many other Oses like Unix and Linux), user processes are forbidden from directly accessing hardware, and must instead communicate indirectly through a kernel-level process (e.g., device driver). While this can protect the computer in some instances, it adds substantial complexity (especially during the hardware/software construction phase). Furthermore, the protection is only as good as the device driver code, which (at least in Linux, and I presume also in NT) runs without any of protections afforded to user processes. Therefore bugs in the device driver code will crash the machine just as they would in a user process under Win 95 – except that recovery from crashes in NT (and Linux) tends to be more problematic.

The source code presented here is written for Windows 95. Hardware addresses are #defined, and accessed directly using inp() and outp() C functions, which are not supported under NT.

3. Writing, porting, and debugging MEX files

3.1. MEX structure

Generally, it is more time consuming (and dangerous) to program in C than in a higher level language such as MATLAB. Furthermore, MEX functions run in a somewhat unsupervised environment within the MATLAB process space, so errors can have catastrophic consequences. Therefore MEX functions should be as simple and short as possible to get their task done, and return control (and perhaps data) to MATLAB, where additional steps can be performed in .m files. I particularly recommend this minimalist philosophy for hardware interface functions, where bus errors and other communication failures can hang

the system with few clues for debugging. The structure of my MEX functions is therefore always very simple:

1. Extract (and perhaps check) arguments from the MATLAB command line
2. Do the necessary task(s)
3. Package return values for MATLAB
4. Return

List 1. General MEX file structure.

The prototype of all MEX functions is the same:

```
#include "c:\matlab\extern\include\mex.h"
void mexFunction(int nlhs,  mxArray *plhs[], int nrhs, const mxArray *prhs[])
```

The first and third arguments are the number of right and left hand arguments on the command line, and the second and forth arguments are pointers to arrays of pointers to each of the left and right side arguments. This is very analogous to the prototype for a classical C program entry point, `void main(int argc, char *argv[])`. Nrhs and nlhs can be zero. Numerous functions are provided to create and manipulate the mxArray structures and their data (see examples and the helpdesk\Application Program Interface .html pages). The mxArray structure has frequently changed with upgrades so using the provided interface functions is preferable to directly accessing data elements. MATLAB generally works with only two internal data types: double and char, so type conversions will probably be necessary in steps 1 and 3.

```
short foo(double bar)
{      // do something useful
….
}

#ifdef MATLAB_MEX_FILE
#include
"c:\matlab\extern\include\mex.h"
void mexFunction(
       int nlhs,
       mxArray *plhs[],
       int nrhs,
       const mxArray *prhs[] )
{
// unpack arguments
…
value = foo(b); // do something useful
// clean-up and return value to MATLAB
…
return;
}      //mexFunction

#else

#include <stdio.h>
void main(void)
{
…
a = foo(b);
}      // main
#endif
```

Listing 2. Typical MEX source file structure.

Porting stand-alone C functions is the most straight-forward, as the code for step 2 in the above list is already written and code for the other steps can generally be copied from some other MEX file. On the other hand, if the exact programming steps are not initially known and must be determined with anticipated trial-and-error, it may be easier to first make a stand-alone C program in MSVC. Start with the Win32 Console Application template, and structure main() so it calls another function that performs the core task. The MEX Perl script defines the preprocessor symbol MATLAB_MEX_FILE, so the same file can contain source for both the MSVC and/or MEX environment. For example, the source file in List 2 will enter at main() when compiled within MSVC (and mexFunction() will be ignored). Function foo can be studied by setting breakpoints (right-clicking on a line, within the MSVC editor). Once running satisfactorily, the MEX file can be built by switching to MATLAB and typing the command: mex *filename.c* followed by any additional source or library files. Perl will launch another MSVC process to compile (this time, main() will be ignored since MATLAB_MEX_FUNCTION is defined) and link to *filename.dll*. The MEX file can now be invoked on the command line *[outargs] = filename(inargs)*;. As a last step, use the editor to create *filename.m* (in the same directory as *filename.dll*), containing commented lines that will be echoed whenever *help filename*is entered.

3.2. Debugging

Debugging as just described (using the MATLAB_MEX_FILE preprocessor flag) is a good way to get the core routine(s) debugged, however it has serious limitations. The program may behave differently running the outside the MATLAB environment, and code within mexFunction() is not even compiled. Perhaps the simplest mechanism for debugging MEX files from within MATLAB is to sprinkle the code with printf() statements, and this method should not be discounted. Source level debugging is possible, but has not worked until recently, and my only (successful) experience with it is to debug and verify the following procedure (Listing 3), modified from the Mathworks web site. The procedure documented in the API Guide does not work.

[*from the Mathworks website*]
The instructions for how to debug MEX-files in Microsoft are not correct. This has been brought to the attention of our development staff so that it can be addressed in a future release of the documentation.

The correct steps are as follows:

1.) Compile the MEX-file with the -g option. For example

mex -g filename.c

2.) Start the Microsoft Development Studio. Select File:Open Workspace…, enable all file types, and select filename.dll.

3.) In the Microsoft environment, go under Project, and select the Settings option. In the window that opens, select the Debug Tab. In this options window, there will be edit boxes. In the edit box labeled Executable For Debug Session enter the full path to where MATLAB 5 resides. All other edit boxes should be empty.

4.) Open the source files and set a break point on the desired line of code by right-clicking your mouse on the line of code.

5.) From the Build menu, select Debug, and click Go.

You will now be able to run the MEX-file in MATLAB and use the debugging environment of Microsoft.

List 3. Source-level debugging of MEX files running within MATLAB.

4. Sample MEX files

4.1. Programming Conventions

Programming is a highly stylistic but constrained form of expression, and it involves many compromises between algorithmic and execution efficiency, readability, and other factors. The inherently small size of my MEX files affects some of these compromises – for example, Hungarian naming conventions or other mechanisms that are essential to navigating large projects I believe can obfuscate and add clutter to very short programs.

4.2. Sleep

Blocking pauses are frequently needed in automation protocols. The built-in MATLAB pause() function is limited to integral seconds (or indefinitely until a key is hit). (Note, to block until receiving user input, also use input() or ginput().) Pausing for fractions of a second should be doable by this simple .m function:

```
function sleep(howLong)
% sleep(howLong) blocks for howLong seconds
t0 = clock;
while etime(clock, t0) < howLong;
end;
```

However, I have associated Windows crashes with calling a similar routine, and therefore wrote the following MEX function, using Windows timers instead. Note, no effort has been made to measure or achieve high accuracy in the delay of this function. It will always take longer than the specified time due to additional latencies in the interpreter and execution engine.

```
/*
  sleep.c  -- like pause, but can sleep for fraction seconds.
  30jan98, BKR, Amgen Inc.
  Note, this is not tested to be accurate to a millisecond -- interrupts
  and other latencies may affect it.
  The calling syntax is:

          sleep(seconds);
*/

#include <time.h>
#include "c:\matlab\extern\include\mex.h"

void sleep(clock_t wait)
{       // Pauses for a specified number of milliseconds
   clock_t goal;
   goal = wait + clock();
   while(goal > clock()) ;
}

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
   double       secs;
   clock_t      msecs;    // milliseconds to sleep

// check inputs
  if (nrhs != 1)
   mexErrMsgTxt("sleep: needs an argument (seconds)");

  msecs = (clock_t)(1000.0 * mxGetScalar(prhs[0])); // extract the argument
  sleep(msecs);
  return;
}
```

Listing 4. Sleep.c.

The source contains two function declarations. The first, sleep() uses API code (in <time.h>) to do what we want. MexFunction first verifies there is one input argument (and if not, exits after printing a message). Next, the value of the input is extracted using mxGetScaler, converted from (double) seconds to (clock_t) milliseconds, and passed to sleep(). Nothing is returned. The MEX function, sleep.dll is created by typing in MATLAB mex sleep.c.

4.3. Advantech digital output card

The Advantech PCL-720 is an ISA board supporting digital i/o and counting. The MEX function dout() simply sets the digital outputs to the states specified by the bits in its argument. The board is assigned an i/o address when installed in the computer. Writing to this address with outp() is all that is needed to communicate with the board. If the board is moved or removed, then outp() may cause a bus error – and for this reason hard-coding addresses is frowned upon in commercial programming. However, it is a very simple mechanism (avoiding a plethora of function calls querying the OS for the presence and location of this board). The address is #defined both for readability and easy recompile should the address change.

```
/*
  dout.c  -- mex file for Advantech PCL-720 digital
output board
  13dec97, BR, Amgen Inc.
  compile from matlab >> mex dout.c
The calling syntax is:
      dout(value);
value must be a decimal number.
  dout(hex2dec('hex#') works for hex args.
*/

#include <conio.h>  // defines low level i/o
routines (in libc.lib)
#include "c:\matlab\extern\include\mex.h"

#define DOUT_ADDR 0x2a1   // baseAddress + 1

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
  unsigned short    value;

// check inputs
  if (nrhs != 1)
   mexErrMsgTxt("dout: needs an argument");
  if (!mxIsNumeric(prhs[0]))
   mexErrMsgTxt(
    "dout: arg must be integer (or use
hex2dec())");

  value = (unsigned short)mxGetScalar(prhs[0]);
  // extract the arg
  outp(DOUT_ADDR, value); // write to hardware
return;
}
```

Listing 5. Dout.c

MexFunction first checks that it has one numeric input argument (using nrhs and mxIsNumeric) , then extracts it (with mxGetScalar), converts the resulting double to a short, and outp's it to the hardware. Nothing is returned.

4.4. Serial Communication routines

Serial communication is a common interface mechanism between microcontrollers and other semi-intelligent devices. Windows implements serial communication using the file model (copying Unix device drivers). Adhering to this same style, I wrote the following four MEX functions and associated .m files:

File openserial.m

```
% hPort = openSerial('COM2', 'rw', baudRate, parity, stopBits, byteSize);
% returns a handle to a serial port structure. This handle is used for subsequent
% calls to readSerial, writeSerial, closeSerial (don't do math with it).
% permission can be 'r', 'w', 'rw'(default)
% baudRate is the usual 9600, 19200, etc.
% parity = 0-4 for {no(default),odd,even,mark,space}
% stopBits = 0,1,2 for {1(default), 1.5, 2}
% byteSize = 4-8 (default=8)
```

File: closeserial.m

```
% closeSerial(hSerPort);
% closes a serial port opened with openSerial
```

File: readserial.m

```
% str = readSerial(hSerPort, timeout);
% reads ascii from serial port specified by hSerPort returned from previous call
% to openSerial. timeout (in seconds) is how long it will wait for characters
% (but it returns immediately after receiving a string).
```

File: writeserial.m

```
% status = writeSerial(hSerPort, string);
% writes string to serial port specified by hSerPort returned from previous call
% to openSerial. The number of bytes written is returned. There is no handshaking
% or verifying that the data is read at the other end.
```

Some highlights of these functions are:

- I developed the code initially from a MSVC project that called the core functions from main(). I first tested with a modem (seeing if writing "ATH" took the phone off-hook), and proceeded debugging with a serial snooper box.
- I found documentation for the API calls in the MSVC on-line help and by doing multifile searches through header files. The library containing the API calls is by default linked into the MEX file.
- There is a bug (or to quote Microsoft customer support, "its not a bug, but a documentation error, a pitfall") in the Windows 95 serial drivers, and they don't quite work as they should. I am not sure my work-around in writeserial is 100% successful.
- Openserial returns a type Handle, which I cast as a double to return to MATLAB, and later recast back to a Handle within the other three routines that take it as an input. In general, be careful not to get burned with improper typecasting.
- Writeserial does not need an argument giving the length of the string since the string is passed within a mxArray structure which contains a size field (in contrast, the C version of write needs the size in addition to the char *string). This applies to any array (string or numeric) passed to a MEX file.
- I often use static variables to hold arguments, initialize them to reasonable default values, and use a switch statement without breaks to read the right-hand arguments. This results in the convenient behavior that the later arguments of a function, if omitted, will retain their previous values.
- writeserial appends CR and LF to the string passed from MATLAB, since this was an easy fix to what appears to be a universal senario. In hindsight, it would have been better to do this by renaming the MEX file anything else, e.g., writeserialmex, and have writeserial.m append the string as follows:

  ```
  function status = writeserial(hSerPort, string)
  % … (help documentation) …
  status = writeserialmex(hSerPort, [string char(13) char(10)]);
  ```

  This would have saved development time (trial-and-error to generate the above line is much quicker than the few edit-compile-run cycles required to get the C code working), shortened and simplified the source code, and provided the flexibility to call writeserialmex directly should some device not want CRLF appended.

File: openserial.c

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#include <winbase.h>
#include "c:\programming\matlab\extern\include\mex.h"

/*
  hPort = openSerial('COM2', 'rw', baudRate, parity, stopBits, byteSize);
  returns a handle to a serial port structure. This handle is used for
  subsequen calls to readSerial, writeSerial, closeSerial (don't do math
  with it).
  permission can be 'r', 'w', 'rw'(default)
  baudRate is the usual 9600, 19200, etc.
```

```
  parity = 0-4 for {no(default),odd,even,mark,space}
  stopBits = 0,1,2 for {1(default), 1.5, 2}
  byteSize = 4-8 (default=8)

  Written for microsoft visual C++5
  Compile from matlab with: mex openserial.c

  11 feb 98, BKR

*/

HANDLE openSerial(char *sPort, char *sPermissions, int iBaud, int iParity,
                  int iStopBits, int iByteSize);
die(char  *str, long   val)
{
   printf("%s  %ld\n", str, val);
   return(1);
}

#define SIZ 5  /* char buf static alloc */

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    HANDLE          hSerPort;
    static int      byteSize=8, stopBits=0, parity=0, baudrate=9600;
    static char     sPortName[SIZ], permissions[SIZ];
    long            buflen, status;
    double          *dpReturn;

   if(nlhs != 1)
      mexErrMsgTxt("openSerial needs an output arg");
// check inputs
   switch(nrhs)
   {
      case 6:
         byteSize = (int)mxGetScalar(prhs[5]);
      case 5:
         stopBits = (int)mxGetScalar(prhs[4]);
      case 4:
         parity   = (int)mxGetScalar(prhs[3]);
      case 3:
         baudrate = (int)mxGetScalar(prhs[2]);
      case 2:
         buflen   = (mxGetM(prhs[1]) * mxGetN(prhs[1]))+1;
         if(buflen > SIZ) mexErrMsgTxt("openSerial: problem with arg2");
         status   = mxGetString(prhs[1], permissions, buflen);
         if(status)
            mexErrMsgTxt("openSerial: couldnt parse permissions");
      case 1:
         buflen   = (mxGetM(prhs[0]) * mxGetN(prhs[0]))+1;
         if(buflen > SIZ) mexErrMsgTxt("openSerial: problem with arg1");
         status   = mxGetString(prhs[0], sPortName, buflen);
         if(status)
            mexErrMsgTxt("openSerial: couldnt parse sPortName");
         break;
      default:
         mexErrMsgTxt("Bad arguments. Type help openserial");
         break;
   }

   if((hSerPort = openSerial(sPortName, permissions, baudrate, parity,
```

```
                        stopBits, byteSize)) == NULL)
       mexErrMsgTxt("failed to openSerial port");

   plhs[0]   = mxCreateDoubleMatrix(1, 1, mxREAL);
   dpReturn  = mxGetPr(plhs[0]);
   buflen    = (long)hSerPort;  // can't convert void * to double directly
   *dpReturn = (double)buflen;
   return;
}      // mexFunction

static HANDLE   openSerial(char *sPort, char *sPermissions, int iBaud,
                int iParity, int iStopBits, int iByteSize)
{
   static HANDLE       fid;
   DWORD        permission = 0;
   DCB          dcb;

   if(strpbrk(sPermissions, "r") != NULL)
      permission |= GENERIC_READ;
   if(strpbrk(sPermissions, "w") != NULL)
      permission |= GENERIC_WRITE;

   fid = CreateFile(sPort, permission, 0, NULL,
                    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
   if(fid == INVALID_HANDLE_VALUE) die("CreateFile failed", 0);

   if(!GetCommState(fid, &dcb))
      die("GetCommState failed", 0L);
   dcb.BaudRate = iBaud;
   dcb.Parity = iParity;         // 0-4=no,odd,even,mark,space
   dcb.StopBits = iStopBits;   // 0,1,2 = 1, 1.5, 2
   dcb.ByteSize = iByteSize;     //4-8
   if(!SetCommState(fid, &dcb))
      die("SetCommState failed", 0L);
   return(fid);
}      // openSerial
```

file: closeserial.c

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#include <winbase.h>
#include "c:\matlab\extern\include\mex.h"
/*
  closeSerial(hSerPort);
  closes serial port opened with openSerial

  Compile with: mex closeSerial.c

  11 feb 98, BKR
*/
long closeSerial(HANDLE hPort);

die(char  *str, long   val)
{
   printf("%s  %ld\n", str, val);
   return(1);
}

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
```

```
{
    HANDLE          hSerPort;
    char        *str;
    long        buflen, tmp, status;
    double          *dpReturn;

// check inputs
   if(nrhs != 1) mexErrMsgTxt("closeSerial needs 1 arg");

   tmp = (long)mxGetScalar(prhs[0]);
   hSerPort = (HANDLE)tmp; // don't know how to check if valid
   status = closeSerial(hSerPort);
   return;
}      // mexFunction

int closeSerial(HANDLE hPort)
{
   if(!CloseHandle(hPort)) die("CloseHande failed", 0L);
   return(true);
}
```

File: readserial.c

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#include <winbase.h>
#include <time.h>
#include "c:\matlab\extern\include\mex.h"

/*
  str = readSerial(hSerPort, timeout);
  reads ascii from serial port specified by hSerPort returned from previous
call
  to openSerial. timeout (in seconds) is how long it will wait for characters
  (it returns immediately after receiving a string).

  Compile with: mex readSerial.c

  11 feb 98, BKR

*/

char * readSerial(HANDLE hPort, int timeOut);

die(char  *str, long   val)
{
   printf("%s  %ld\n", str, val);
   return(1);
}

void sleep(clock_t wait)
{   // Pauses for a specified number of milliseconds
   clock_t goal;
   goal = wait + clock();
   while(goal > clock()) ;
}

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    HANDLE          hSerPort;
```

```
    char            *str;
    static long     timeout = 0, tmp;

// check inputs
   if(nrhs < 1) mexErrMsgTxt("readSerial needs at least 1 args");
   if(nrhs == 2) timeout = (long)mxGetScalar(prhs[1]);
   tmp = (long)mxGetScalar(prhs[0]);
   hSerPort = (HANDLE)tmp; // don't know how to check if valid

   str = readSerial(hSerPort, timeout);

// check outputs
   if(nlhs == 1)
      plhs[0] = mxCreateString(str);
   else
      printf("%s\n", str);
   return;
}     // mexFunction

/*  ReadFile has a bug in that it doesn't timeout properly
    so we can check timeout manually.
*/
char *readSerial(HANDLE hPort, int timeOut)
{
   long         numCharsRead;
   static char      str[500];
   BOOL         status;
   COMMTIMEOUTS timeouts;
   clock_t          pitfall;

// TotalTimeout (in msec) = TimeoutMultiplier * sizeof(str)
// so multiplying timeOut * 2 ==> seconds
   GetCommTimeouts(hPort, &timeouts);
   timeouts.ReadIntervalTimeout = 50;  // msec, a guess
   timeouts.ReadTotalTimeoutMultiplier = timeOut*2;
   timeouts.ReadTotalTimeoutConstant = timeOut*2;
   SetCommTimeouts(hPort, &timeouts);

   pitfall = clock();
   status = ReadFile(hPort, str, sizeof(str), &numCharsRead, NULL);
   while(numCharsRead == 0 && clock() < pitfall + timeOut * 1000)
      status = ReadFile(hPort, str, sizeof(str), &numCharsRead, NULL);
printf("numcharsread=%ld\n", numCharsRead);
   str[numCharsRead] = 0; // terminate str
   return(str);
}     // readSerial
```

File: writedserial.c

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#include <winbase.h>
#include "c:\programming\matlab\extern\include\mex.h"

#undef DBUG

/*
  status = writeSerial(hSerPort, string);
  writes string to serial port specified by hSerPort returned from previous
call
  to openSerial. The number of bytes written is returned. There is no
```

```
  handshaking or verifying that the data is read at the other end.

  Compile with: mex writeSerial.c

  11 feb 98, BKR

*/

long writeSerial(HANDLE hPort, char *string);

die(char  *str, long   val)
{
   printf("%s  %ld\n", str, val);
   return(1);
}

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    HANDLE          hSerPort;
    char            *str;
    long            buflen, tmp, status;
    double          *dpReturn;

// check inputs
   if(nrhs != 2) mexErrMsgTxt("writeSerial needs 2 args");

   buflen = (mxGetM(prhs[1]) * mxGetN(prhs[1]))+1;
   buflen += 2; // add room for crlf
   str = mxCalloc(buflen, sizeof(char));
   status = mxGetString(prhs[1], str, buflen);
   if(status)
   {
      free(str);
      mexErrMsgTxt("writeSerial couldnt parse string");
   }
   str[buflen-3] = 13;    // cr & lf
   str[buflen-2] = 10;
   str[buflen-1] = 0;
   tmp = (long)mxGetScalar(prhs[0]);
   hSerPort = (HANDLE)tmp; // don't know how to check if valid

   status = writeSerial(hSerPort, str);
   free(str);
   if(status != buflen-1) die("writeSerial wrote chars:", status);
   if(nlhs == 1)
   {
      plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
      dpReturn = mxGetPr(plhs[0]);
      *dpReturn = (double)status;
   }
   return;
}     // mexFunction

long writeSerial(HANDLE hPort, char *str)
{
   long         numCharsWritten = 0;

#ifdef DBUG
printf("writeSerial: hPort = %ld\n", (long)hPort);
printf("writeSerial: str = %s\n", str);
printf("strlen=%ld\n", strlen(str));
```

```
#endif

   if(!WriteFile(hPort, str, (DWORD)strlen(str), &numCharsWritten, NULL))
      die("WriteFile failed", 0L);
   return(numCharsWritten);
}      //writeSerial
```

4.5. National Instruments Multifunction I/O cards

4.5.1. Digital output

This function configures the NI digital i/o port for output and sets the 8 bits to the value of its argument. Although the task is identical to the Advantech dout function, the code is orders of magnitude more complicated and execution again wanders through multiple DLLs before a writing a single byte to the card. Documentation for the NI interface functions are in the NI-DAQ User Manual for PC Compatibles and NI-DAQ Function Reference Manual for PC Compatibles. A register-level programming manual is not available yet for this card, although the draft version I squeezed out of NI is very dense. Nidout.c consists of two functions: main() and mexFunction(), which are conditionally compiled with the MATLAB_MEX_FILE flag.

File: nidout.c

```
/****
   NIdout.c, 10 jan 98, BKR

   write to the digital port on National Instruments PCI-MIO16-E4
   board.
   This file can compile as a matlab .mex file or a standalone
   msvc++5 console application. Note, I never tried the standalone.

    Compile from matlab> mex NIdout.c nidac32.lib
   Call from matlab> NIdout(pattern)
   e.g., nidout(1) sets DIO0 high.

   Note: the NIDAQ driver probably must be kept open for the board to
   function.  This can be done by keeping the NI-DAQ Configuration
   Utility application open in the background.
****/

#include <stdio.h>
#include "nidaq.h"
#include "nidaqcns.h"  /* constants */
#define HWDEVICE 1   /* hardware device # assigned to board */

#ifdef MATLAB_MEX_FILE

#include "c:\programming\matlab\extern\include\matrix.h"
#include "c:\programming\matlab\extern\include\mex.h"

void mexFunction(int nlhs,       mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{      // NIdout(pattern)
   static double          dPattern;
   static short           status;

// check and unpack arguments
   if(nrhs != 1)
      mexErrMsgTxt("NIdout(pattern) needs one argument\n");
   dPattern = *mxGetPr(prhs[0]);
   if((status = DIG_Prt_Config(HWDEVICE, 0, 0, 1)) != 0)
      mexErrMsgTxt("NIdout: failed to config digital port\n");
```

```
   if((status = DIG_Out_Port(HWDEVICE, 0, (short)dPattern)) != 0)
      mexErrMsgTxt("NIdout: failed to output digital port\n");
   return;
}     // mexFunction()

#else  /* make a standalone application for debugging */

void main(void)
{
   short              status;

   status = DIG_Prt_Config(HWDEVICE, 0, 0, 1); // port is output
   printf("config status = %hg]n", status);
   status = DIG_Out_Port(HWDEVICE, 0, 0x55AA);
   printf("output status = %hg]n", status);
   getchar();   // wait
}     // main

#endif    /* MATLAB_MEX_FILE */
```

4.5.2. Analog input

This function synchronously records analog waveforms from an arbitrary set of channels with arbitrarily specified gains on each channel. Static arrays store the channels and gains so once specified, they need not be passed in successive function calls. Copying the data directly into a MATLAB matrix results each channel being a different row. Since making each channel a separate column is preferable (so MATLAB functions like mean() return means of each channel), there is a bit of index gymnastics near the end to flip the array around. Perhaps a better way to do the flipping could be to use a MATLAB callback to do a transpose, but I have never done this.

File: niain.c

```
/****
   niain.c, 20mar98, BKR

   synchronously record analog waveforms from National Instruments
   PCI-MIO16-E4 board, board is configured differential

    Compile from matlab> mex niain.c nidac32.lib

   Call from matlab> waveform = niain(numsamples, rate, channels, gains);
   rate (samples/sec) ranges from ~.00153 to 250,000
   channels = 0 to 7, any combination of channels
   gains = 1,2,10,100, same size array as channels
****/

#include <stdio.h>
#include "nidaq.h"
#include "nidaqcns.h"  /* constants */

#define HWDEVICE 1  /* hardware device # assigned to board */
#define DIFFERENTIAL 0 /* configuration of analog input */
#define REFSINGLEENDED 1
#define BIPOLAR 0     /* analog input */
#define UNIPOLAR 1    /* analog input */
#define MAX(a, b)  (((a) > (b)) ? (a) : (b))

#include "c:\matlab\extern\include\mex.h"

void mexFunction(int nlhs,       mxArray *plhs[],
```

```
                int nrhs, const mxArray *prhs[])
{     // niain
   static double          *dWaveform, dRate = 1000., *dPtr, adc2v;
   static unsigned long   ulNumSamples = 100, m, n;
   static short        i, numChans = 1, chans[8] = {0,0,0,0,0,0,0,0}, iStatus;
   static short        gains[8]={1,1,1,1,1,1,1,1}, trigger = 0, *iWf = NULL,
*iPtr;

// unpack arguments.
   switch(nrhs)
   {
      case 5:
         trigger = (short)mxGetScalar(prhs[4]);
      case 4:
         dPtr = mxGetPr(prhs[3]);
         m = mxGetM(prhs[3]);          n = mxGetN(prhs[3]);
         for (i=0; i<(short)MAX(m, n); i++)
            gains[i] = (short)dPtr[i];
         for ( ; i<8; i++) // in case numgains < numchans
            gains[i] = gains[0];
      case 3:
         dPtr = mxGetPr(prhs[2]);
         m = mxGetM(prhs[2]);          n = mxGetN(prhs[2]);
         numChans = (short)MAX(m, n);
         for (i=0; i<numChans; i++)
            chans[i] = (short)dPtr[i];
      case 2:
         dRate = mxGetScalar(prhs[1]);
      case 1:
         ulNumSamples = (unsigned long)mxGetScalar(prhs[0]);
         break;
      default:
         mexErrMsgTxt("niain: bad arguments");
   }
// do it
   if((iWf = (i16 *) calloc(ulNumSamples * numChans, sizeof(i16))) == NULL)
      mexErrMsgTxt("niain: malloc failed\n");
   iStatus = AI_Clear(HWDEVICE);       // empty FIFO
   iStatus = AI_Configure (HWDEVICE, chans[0], DIFFERENTIAL, BIPOLAR,
                           BIPOLAR, 0);
   iStatus = SCAN_Op(HWDEVICE, numChans, chans, gains, iWf,
                ulNumSamples * numChans, dRate * numChans, 0);
   if(iStatus != NO_ERROR)
      mexErrMsgTxt("niain: internal failure");
// return data to matlab. Need to flip the matrix to make channels columns.
   plhs[0] = mxCreateDoubleMatrix(ulNumSamples, numChans, mxREAL);
   dPtr = mxGetPr(plhs[0]);      // get the pointer
   for(i = 0; i< numChans; i++)
   {
      adc2v = 5. / 2048. / gains[i];
      iPtr = iWf + i;
       // pointer arithmetic, increments by i * sizeof(short)
      for(m = 0; m < ulNumSamples; m++)
      {
         *dPtr++ = (double)*iPtr * adc2v;
         iPtr += numChans;
      }
   }
   return;
}     // mexFunction()
```

4.5.3. Asynchronous analog i/o

The next example is fairly simple to describe (unfortunately that's about the only simple aspect of it). The goal is to generate an analog voltage waveform, supplied from MATLAB as a vector and scalar duration, and simultaneously record the associated current. The MEX function must return immediately back to MATLAB, before the waveform has completed (or even before it has begun, if it is configured to start with a hardware trigger). I choose the National Instruments (NI) PCI-MIO-16E-4 board for this task because it contains 512 word FIFO buffers on both the ADC and DACs, along with timing circuitry to clock the buffers and converters without CPU intervention.

The board is programmed using NI-supplied MSVC libraries and associated header files, and other DLL files -- so the actual executable code is strewn all over the disk and has all sorts of hidden dependencies. Making this situation worse, the code is strewn with undocumented bugs, that require undesirable tangles of source code or other workarounds to circumvent. For example, after the waveform buffer is downloaded to the DAC FIFO, the RAM waveform buffer should not be needed. However, as soon as free() is called, waveform generation aborts. The only workaround is to not call free() and manually manage the static buffer, which leaves a potential for a memory leak. After diagnosing this bug, I found the waveform again aborts as soon as the MEX function exits. According to NI, the driver is trying to be intelligent and when it sees no processes are using it, it closes itself. A workaround is to launch the NI Configuration Utility application and keep it open as a background process, which in turn keeps the driver from resetting the board. The result of all this is a software package that is at best fragile, and crashes are fairly common, but fortunately occur mostly during startup. NI-DAQ 6.1 (ftp://ftp.natinst.com/support/daq/pc/ni-daq/6.1/) appears much more stable than the v5 shipping with the cards.

The first call to niaout specifies the waveform and sample rate and immediately returns to MATLAB. Static variables keep the current waveform buffer (which hasn't been written yet) valid between function calls. The simplest way to later return the current buffer to MATLAB is to call the same function requesting it. I used the duration parameter as a flag: if duration < 0, niaout returns the current buffer, and otherwise it outputs a waveform and reads the current buffer from the ADC. The details of this implementation are completely encapsulated and hidden by the file getcurrent.m:

File: getcurrent.m

```
function current = getCurrent
% function current = getCurrent;
% called after niaout(waveform, duration, trigger) has finished its waveform
% it returns analog input collected during the waveform generation. Analog
input
% is measured differentially between #defined channels, and has units of mV
% if the sense resistor, etc. match the #defines.

% 14feb98, BKR

current = niaout([], -1);
% negative duration tells niaout to just return the current buffer
```

The file niaout.c contains both main() and mexFunction(), so it can be compiled in both MSVC and MATLAB environments without any modifications. The communication and configuration of the hardware is done in the function asyncWaveform, which returns a pointer to the current buffer. Note that type conversions again had to be done carefully. Variable declarations in asyncWaveform are of typedefs in the NI header files, and type conversions are done just before and after this call. The NI file nidaq32.lib must be included in either project, and a couple of NI header files must be in the same directory. Several parameters such as the hardware base address are #defined. And I repeat for emphasis, the NI Configuration Utility must be open for niaout to execute properly in real-time.

File: niaout.c

```
/****
   niaout.c, 14feb98, BKR

   asynchronously output an analog waveform from National Instruments
   PCI-MIO16-E4 board (DAC0) and simultaneously acquire differential
   analog input (ACH7-ACH15) at the same rate and for the same number
   of samples.
   The output waveform is loaded to the board's FIFO buffer (only
   single buffering is implemented, max 512 samples) and
   the iTrigger is armed. This function should return before the
   waveform is finished (or before it has even started if the external
   trigger is enabled and hasn't hit). Since this function returns
   immediately, it must be called again (i.e., after takepictures returns)
   to return the current waveform.
   The input waveform is read into a buffer calloced and held valid by
   just a static pointer. This is rather dangerous and has nasty
   initialization and closure problems, but life is short. Furthermore,
   freeing iWaveform after loading it to the dac destroys the waveform
   (a reported bug in NIDAQ), so a memory leak is inevitable.

    Compile from matlab> mex ainout.c nidac32.lib

   Call from matlab> status = ainout(waveform, duration, iTrigger);
             or        aout = ainout([], -1); % aout in mA

   if iTrigger != 0 then waveform begins on hi2low on pin PFI5.
   if duration < 0  then the analog input buffer is returned and
   no analog output is generated. channel is 0:7 (0 is AIN0-AIN8)


****/

#include <stdio.h>
#include "nidaq.h"
#include "nidaqcns.h"  /* constants */

#define HWDEVICE 1   /* hardware device # assigned to board */
#define AOCHAN   0      /* analog output channel */
#define ADC_FIFO_SIZE 512 // number of bytes in the fifo
#define AINCHAN  7      /* analog input channel */
#define AINGAIN 100     /* NOTE: ain range is 5V */
#define DIFFERENTIAL 0 /* configuration of analog input */
#define BIPOLAR  0      /* analog input */
#define UNIPOLAR 1      /* analog input */
#define CURRENT_SENSE_RESISTOR 10 /* ohms, current = AIN/R */

short *asyncWaveform(double *dWaveform, unsigned short uCount,
                     double duration, short trigger);

#ifdef MATLAB_MEX_FILE

#include "c:\matlab\extern\include\mex.h"

void die(char   *str, long      errCode)
{
   mexPrintf("asyncWaveform error %ld\n", errCode);
   mexErrMsgTxt(str);
}

void mexFunction(int nlhs,       mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{      // niaout(waveform, duration, trigger);
```

```
   static double          duration = 1, *dWaveform;
   static unsigned short  uCount;
   static unsigned int    m, n;
   static short           trigger = 0, *iWf = NULL;

// check and unpack arguments. Trigger first
   if(nrhs == 3)
      trigger = (short)mxGetScalar(prhs[2]);
// duration next. if < 0 then just readback the adc buffer and return
   duration = *(mxGetPr(prhs[1]));
   if(duration < 0)
   {
      register double     *dptr, adc2mA;

      if(iWf == NULL || uCount < 1)
         mexErrMsgTxt("ainout: the waveform buffer is empty");
      if(nlhs == 0)
         mexErrMsgTxt("ainout needs an output arg to return waveform");
      plhs[0] = mxCreateDoubleMatrix(uCount, 1, mxREAL);
      dptr = mxGetPr(plhs[0]);  // get the pointer
      adc2mA = 1000.0 / AINGAIN / CURRENT_SENSE_RESISTOR / 2048. * 5.;
/*    iStatus = DAQ_Check(1, &iStopped, &c);  // check if wf is done
       while ((iStopped != 1) && (iStatus == 0))
         iStatus = DAQ_Check(1, &iStopped, &c); // wait till wf is done
*/    for(m = 0; m < uCount; m++)
         dptr[m] = (double)(iWf[m]) * adc2mA;
      return;
   }
// get & check the waveform
   dWaveform = mxGetPr(prhs[0]);
   m = mxGetM(prhs[0]);                // size the vector
   n = mxGetN(prhs[0]);
   if(m == 1 && n == 1)
   {     // handle special case, set the dac to scalar value
      AO_VWrite(1, 0, *dWaveform);
      return;
   }
   else if(m == 1 && n > 1) uCount = n;       // row vector
   else if(n == 1 && m > 1) uCount = m; // col vector
   else mexErrMsgTxt("waveform must be a vector");
   if(uCount > ADC_FIFO_SIZE)
      mexErrMsgTxt("waveform length must be <= ADC_FIFO_SIZE samples");
// start waveform
   iWf = asyncWaveform(dWaveform, uCount, duration, trigger);
   if(iWf == NULL)
      mexErrMsgTxt("asyncWaveform returned error");
   return;
}     // mexFunction()

#else  /* MEX: make a standalone application for debugging */

#include <time.h>
#include <stdlib.h>    /* for calloc */

void sleep(clock_t wait)
{     // pauses wait msec
   clock_t goal;
   goal = wait + clock();
   while(goal > clock()) ;
}

#define die(str, errCode)                                              \
```

```
   { printf("asyncWaveform error  %ld\n\t%s\n", errCode, str); \
   getchar(); exit(1); }

void main(void)
{
   const unsigned short uCount = 100;  // length of waveform buffer
   const double        dur = 10.0;            // duration of waveform
(seconds)
   unsigned short      index;
   short               a,b;
   long                c;
   short               status, *wfm;
   double              maxV = 4.0, minV = 2.0, dV, V;
   double              *dWaveform;

   status = AO_VWrite(1, 0, minV+1.0);
   printf("status = %hd, voltage = %f\n", status, minV+1);
   getchar();       // pause
// alloc storage for waveform
   if((dWaveform = (double *) calloc(uCount, sizeof(double))) == NULL)
   {
      printf("calloc dWaveform failed");
      getchar();
   }
// generate a ramp waveform from minV to maxV
   dV = (maxV-minV)/uCount*2;    // voltage increment
   V = minV;
   for(index = 0; index < uCount/2; index++)  // rising ramp
      dWaveform[index] = V+= dV;
   for( ; index < uCount; index++)     // falling ramp
      dWaveform[index] = V-= dV;
   dWaveform[index] = minV;      // in case uCount is odd
// doit!!
   wfm = asyncWaveform(dWaveform, uCount, dur, 0);
   for(status = 1; status < 10; status ++)
   {
      a = DAQ_Check(1, &b, &c);
      printf("%hd\t%hd\t%ld\t%hd\t%hd\n", a, b, c, wfm[2], wfm[20]);
      sleep(800);
   }
   getchar();

   //Sleep(5000);  // ==> this gives 5 seconds of waveform!!
}      // main

#endif    /* !_MEX */

short *asyncWaveform(f64 *dWaveform, u16 uCount, f64 duration, i16 iTrigger)
{
    i16         iStatus = 0;
    static i16  *iWaveform;                   // storage for the dac values
    u32         updateInterval;    // # of 50 usec ticks between DAC updates
    i16         chanArray[1] = {AOCHAN};      // array of channel(s) to output
   const double volt2dac = 204.8;      // 10 V = 2048 dac in bipolar mode (1
dac = 4.88 mV)
   i16       i;

// check that parameters are ok.
   if(uCount > ADC_FIFO_SIZE)
      die("waveform vectors must be <= ADC_FIFO_SIZE points", uCount);
   updateInterval = (u32) (duration / uCount / 50e-9); // 50 nsec clock
```

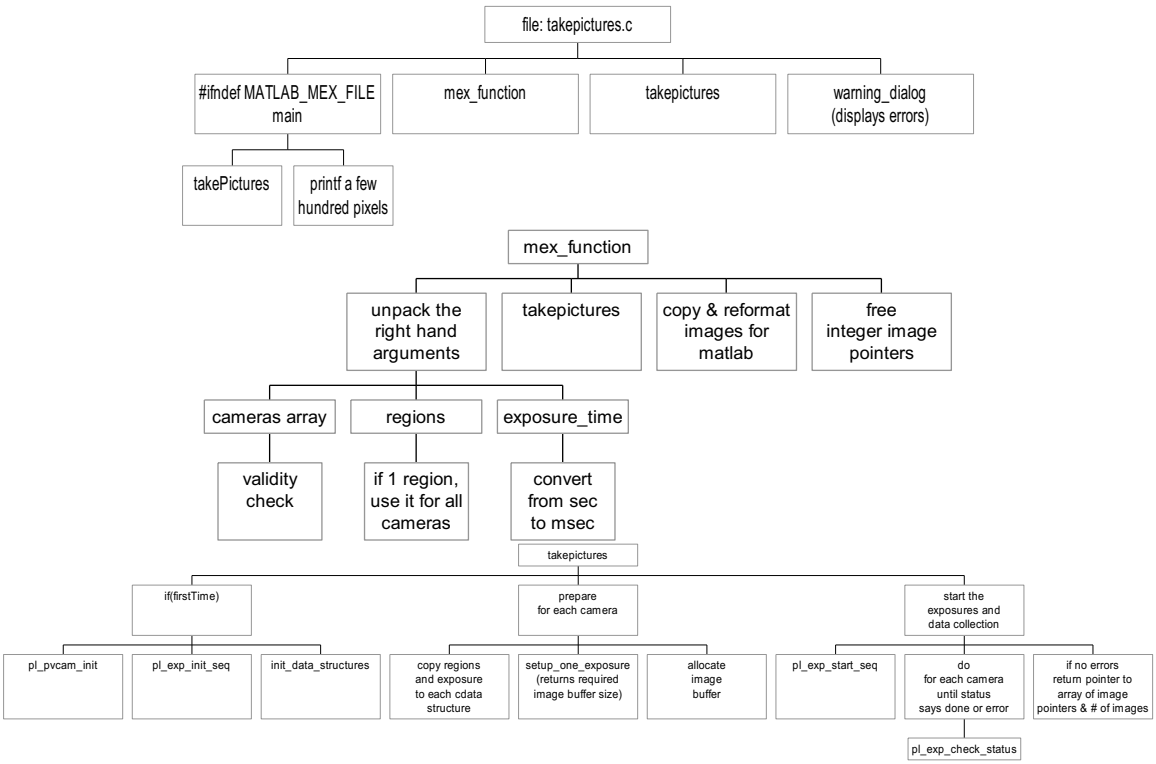
```
   if(updateInterval < 2 || updateInterval > 16777216)     // see NIDAQ Ref
Man 2-454
      die("updateInterval out of bounds", updateInterval);

// initialize (NIDAQ Ref Man 2-435 suggests I had better do this)
   iStatus = WFM_Group_Control(HWDEVICE, 1, 0); // clear
   iStatus = DAQ_Clear(HWDEVICE);
   iStatus = AO_Configure(HWDEVICE, AOCHAN, BIPOLAR, 0, 10.0, 0);
   iStatus = AI_Clear(HWDEVICE);       // empty FIFO
   iStatus = AI_Configure (HWDEVICE, AINCHAN, DIFFERENTIAL, BIPOLAR, BIPOLAR,
0);
// unit conversion: volts --> dacs
   if(iWaveform != NULL) free(iWaveform);     // problems first and last
time!!
/* Ideally iWaveform would not be declared as static, and I would not
   need to do the above step, rather I would free it just after WFM_Load() */
   if((iWaveform = (i16 *) calloc(uCount, sizeof(i16))) == NULL)
      die("failed to calloc iWaveform", uCount);
   for(i = 0; i<uCount; i++)
      iWaveform[i] = (i16)(dWaveform[i] * volt2dac);

// load the waveform buffer, see NIDAQ Ref Man 2-439
   if((iStatus = WFM_Load(HWDEVICE, 1, chanArray, iWaveform, uCount, (u32)1,
1)) != 0)
   {
      free(iWaveform);
      die("WFM_Load failed", (long)iStatus);
   }
// free(iWaveform);// the waveform SHOULD BE in the FIFO, but this kills the
wf!!

// set the clock update rate: see NIDAQ Ref Man 2-418 (very confusing)
   if((iStatus = WFM_ClockRate(HWDEVICE, 1, 0, -3, updateInterval, 0)) != 0)
      die("WFM_ClockRate failed", (long)iStatus);

// set the trigger to free run or external
   if(iTrigger)
      if((iStatus = Select_Signal(HWDEVICE, ND_OUT_START_TRIGGER, ND_PFI_5,
         ND_HIGH_TO_LOW)) != 0)
          die("Select_Signal failed", (long)iStatus);

// ready to go!!!
    if((iStatus = WFM_Group_Control(HWDEVICE, 1, 1))!= 0)  // start
      die("WFM_Group_Control failed", (long)iStatus);
// start data acquistion into the waveform buffer
   iStatus = DAQ_Start(HWDEVICE, AINCHAN, AINGAIN, iWaveform, uCount, 2,
               (short)(updateInterval/200)); // using 100 kHz clock
   return(iWaveform);  // scary, returning a pointer to an array that is
   // still being written by hardware.
}     // asyncWaveform()
```

4.6. Photometrics CCD camera

This MEX file began as a stand-alone C program to instruct a Photometrics CH-350 Camera to take an exposure and return an image. To debug the main() program, I printed out several lines of pixel values, and made sure they changed appropriately with exposure and aperture setting. To view the image, I could either write a million pixels to a file or make a MEX file – I decided to do the later, and very soon had an image in MATLAB. The takepicture.c function was next modified to control multiple cameras and return multiple images, and renamed takepictures.c.

Takepictures.c contains 3 major functions: takepictures(), main(), and mexFunction(). MexFunction keeps a static array of 4 regions (called r -- one per camera, these regions define image offsets, size, and on-camera binning) and the exposure static so these parameters will retain their values between calls. It loops over the number of region arguments to unpack each region vector from the argument list into the region structure. It then calls takepictures(), who's first argument is a pointer to the results -- an array of image pointers (a short ***), and who's return value is the number of images successfully taken. (In hindsight, it might be more elegant to have the function return a short ** and pass a short * numImages as an argument.) The images are simply copied to MATLAB matrix structures and the image buffers are freed.

The code for takepictures() is a little cryptic even by my standards! This code calls many functions documented in the Photometrics PPK Professional Programmers Kit for the PVCAM Library. Several initialization functions must be called just once, and others once per camera. Opening an open camera fails, and the ways I tried closing cameras didn't work properly. I therefore used a static flag variable (firstTime) to conditionally run some initialization code. I am unusually thorough (for me) about checking



Software interconnect diagram for takepictures.c and its functions.

for errors, in part because this was essential for debugging, and there is no watchdog timer to abort an exposure if the camera should fails to respond (so I am also careful not to start an exposure if the camera is not behaving properly). Although I tried to not to change Photometrics code, some small changes were necessary to cam_func.c and cam_func.h, and are commented.

File: takepictures.c

```
/****
   takePictures.c, 30dec97, BKR

   take pictures with photometrics CH350 cameras and pvcam software.
   This file can compiles with the msvc++5 compiler either as a standalone
```

```
   app or a matlab mex file.
   compile from matlab> mex takepictures.c cam_func.c pvcam32.lib
   Can take images from up to 4 cameras, call from matlab:
   > [im1 im2 im3 im4] = takepictures(exposure_time, region1, region2, region3, region4);
    % the 1 is ignored now
   > imagesc(im1); colormap('gray');
***/

/* mod history:
   6 jan 98 BKR: shortened the matlab arg unpacking; started making camera readout
   test all cameras to make sure they were done, but didnt finish.
*/

#include <stdio.h>
#include <string.h>
                          /// PVCAM INCLUDES ////////////////////////////////
#include "master.h"        // PVCAM types, system-dependant constants
#include "pvcam.h"         // function prototypes for all PVCAM functions
#include "cam_func.h"      // camera-control structures and func prototypes

short takePictures(uns16 ***uspImg, uns32 exposure_time, region *r);
// returns a pointer to the image array
static short firstTime = TRUE;        //flag to initialize camera

#ifdef MATLAB_MEX_FILE

#include "c:\matlab\extern\include\matrix.h"
#include "c:\matlab\extern\include\mex.h"

void mexFunction(
                 int nlhs,       mxArray *plhs[],
                 int nrhs, const mxArray *prhs[] )
{
    unsigned int   m, n;
    long           i, j;
    uns16          **ppImage, *pImage, numCameras;
    double         *dptr;     // pointer to double array
    static uns32   exposure_time = 1;         // seconds
    static region  r[4] = {{0, 1024, 1, 0, 1024, 1}, {0, 1024, 1, 0, 1024, 1},
                           {0, 1024, 1, 0, 1024, 1}, {0, 1024, 1, 0, 1024, 1}};
    char           errMsg[50];

// unpack the rhs arguments
    if(nrhs > 1) // copy the regions to r; if only 1 region then make all r[i] the same
        for(i=3, j=nrhs-1; i>=0; i--, j--)
        {
            if(j == 0) j = 1;
            dptr = mxGetPr(prhs[j]);  m = mxGetM(prhs[j]);   n = mxGetN(prhs[j]);
            if(m != 6 && n != 6)
                mexErrMsgTxt("bad region def:{s_offset, s_size, s_bin, p....}");
            r[i].s_offset = (uns16)dptr[0];          r[i].s_size   = (uns16)dptr[1];
            r[i].s_bin    = (uns16)dptr[2];          r[i].p_offset = (uns16)dptr[3];
            r[i].p_size   = (uns16)dptr[4];          r[i].p_bin    = (uns16)dptr[5];
        }
    if(nrhs > 0)
        exposure_time = (uns32)(*mxGetPr(prhs[0])) * 1000;   // convert to msec
        // check range or live dangerous??
// takePictures
    numCameras = takePictures(&ppImage, exposure_time, r);
    if(numCameras < nlhs)
    {
        sprintf(errMsg,"only %hd cameras", numCameras);
        mexErrMsgTxt(errMsg);
    }
// reformat the images for return to matlab
    for(i=0; i<nlhs; i++)
    {
        m = r[i].s_size / r[i].s_bin; n = r[i].p_size / r[i].p_bin;
        plhs[i] = mxCreateDoubleMatrix(m, n, mxREAL);
        dptr = mxGetPr(plhs[i]);      // get the pointers
        pImage = ppImage[i];
        j = m * n;
```

```
        // for(j=0; j<m*n; j++)       // the while loop is generally faster
        while(--j)
            dptr[j] = (double)pImage[j];
        dptr[0] = (double)pImage[0];
        free(pImage);
    }
    for(; i<numCameras; i++)  // in case numCameras > nlhs
        free(ppImage[i]);
}

#else  /* make a standalone application for debugging */

#include <stdlib.h>       /* for calloc */
#define mexErrMsgTxt printf

void  main(void)
{
    uns16  **hImage, *pImage;
    int    i, j;
    short  numCameras;
    region  r[2] = {{0, 1024, 2, 0, 512, 1}, {0, 1024, 2, 0, 512, 1}};
    uns32  exposure_time = 100;      /* msec */

    numCameras = takePictures(&hImage, exposure_time, r);
    for(j = 0; j < numCameras; j++)
    {
        pImage = hImage[j];
        printf("\ncamera %d\n",j);
        for(i=0; i<100; i++)
            printf("%hd\t", pImage[i]);
        printf("\n");
    }
    getchar(); // pause
}       // main

#endif  /* define _MEX */

short takePictures(uns16 ***uspImg, uns32 exposure_time, region *r)
{
    static uns32 m_streamSize;     // size of image to collect, in bytes
    static camera_data_type cdata[MAX_CAMERAS];
    static int16 status;           // status of data collection
    static uns32 bytesCollected;   // number of bytes collected so far
    static uns16 *m_pnImage[MAX_CAMERAS], i;
    extern short firstTime;

    if(firstTime)
    {
        if(!pl_pvcam_init())
        {
            warning_dialog(pl_error_code(),"pvcam_init failed");
            mexErrMsgTxt("pvcam_init failed");       // exit
        }
        if(!pl_exp_init_seq())
        {
            warning_dialog(pl_error_code(),"Cannot init exp seq");
            mexErrMsgTxt("exp_init failed");  // exit
        }
        if( !init_data_structures(&(cdata[0])))      // call once for all cameras
        {
            warning_dialog(pl_error_code(),"Cannot open camera(s)");
            mexErrMsgTxt("init_data_structures failed");     // exit
        }
        firstTime = FALSE;
    }   // firstTime   Note: if region doesn't change, can wrap setup_one_exposure
        // in firstTime. p 66 of pvcam manual

    for(i=0; i<cdata[0].total_cameras; i++)
    {
        cdata[i].exposure_time = exposure_time;
        cdata[i].region = r[i];       // copy the region
        if( !setup_one_exposure(&cdata[i], &m_streamSize) ) // get m_streamSize
```

```
        {
            warning_dialog(pl_error_code(),"cannot setup exposure");
            mexErrMsgTxt("Cannot setup exposure");
        }
    // allocate memory
        if(m_pnImage[i] != NULL) free(m_pnImage[i]);
        if((m_pnImage[i] = (uns16 *) calloc((long)m_streamSize, 1L)) == NULL)
            mexErrMsgTxt("Couldn't calloc image memory");
    }
    // start the exposures
    for(i=0; i<cdata[0].total_cameras; i++)
        if( !pl_exp_start_seq(cdata[i].hcam, m_pnImage[i]) )
            warning_dialog(i, "Cannot start exposure");
    do // wait for exposure to finish
    {                                   // enter a tight polling loop
        status = EXPOSURE_IN_PROGRESS;     // default value
        for(i=0; i<cdata[0].total_cameras; i++)
            if(!pl_exp_check_status(cdata[i].hcam, &status, &bytesCollected))
            {
              warning_dialog(pl_error_code(),"Error during data collection");
              status = READOUT_FAILED;         // quit readout
            }
        // living dangerous, deleted watchdog timer code
    } while(status==EXPOSURE_IN_PROGRESS || status== READOUT_IN_PROGRESS);
    if(status == READOUT_FAILED)
        mexErrMsgTxt("READOUT_FAILED");
    *uspImg = m_pnImage;
    return(cdata[0].total_cameras);
}

void warning_dialog(int16 error_code,char_const_ptr message)
{
  char errStr[512];
  sprintf(errStr,"%s\n\nPVCAM error code = %d",message,error_code);
  printf(errStr);
}
```

5. Controlling commercial lab automation instruments from MATLAB

In this section I describe how MATLAB controlled a commercial robot and liquid handler. This permitted these devices to be tightly integrated with other hardware and software being developed with MATLAB. In addition, MATLAB provided an efficient human interface for editing and debugging the software for these devices.

5.1. Zymark Rapidplate96 pipettor

The Rapidplate has a pair of serial interfaces to its System 7 controller. One serial port is used to write and debug scripts in System 7 (a Forth-like language), and the other communicates with MATLAB through the serial port functions described above. The following MATLAB script fragment initially establishes communication with the Rapidplate and executes System 7 commands to load tips. It leaves a handle to the serial port in the workspace variable “rp”. This script then passes variables defining transfer volume and aspiration height from MATLAB workspace variables to the Rapidplate, and finally tells the Rapidplate to rotate. Writing and debugging is done by entering readserial and writeserial commands on the command line, and later pasting them into script or function (.m) files.

```
if exist('rp') == 0,
   rp = openserial('COM1','rw',600);  % handle to the serial port
   writeserial(rp,'attach.tips');     % execute attach.tips
   s = readserial(rp, 10);     % block until done or 10 sec timeout
   if isempty(findstr(s, 'READY')), error(s); end;
end;
s = readserial(rp,.1); % purge the input buffer
writeserial(rp, ['transfer.volume = ' int2str(txvol)]);
```

```
s = readserial(rp, 10);
writeserial(rp, ['aspirate.height = ' num2str(height)]);
s = readserial(rp, 10);
writeserial(rp, 'rp.index=2'); % rotate
writeserial(rp,'select.index'); % to position 2
if isempty(findstr(s, 'READY')), error(s); end;
```

5.2. Sagian ORCA Robot

The ORCA robot can be controlled through a DDE connection to Sagian's MDS (Methods Development Software, version 2.07) as follows:

```
orca = ddeinit('MDS', 'CPNoWait');  % returns a "channel"
rc = ddeexec(orca, 'grip 0');       % returns 0, 1 for failure, success
…
rc = ddeterm(orca);                 % all done
```

Other than establishing that the above works, I have not had the opportunity to explore this direction further.

6. Conclusions

MATLAB is a powerful software environment for rapid development of sophisticated laboratory automation systems. It offers superb tools for numerical, graphical, and exploratory data analysis, and it can be integrated easily and directly with hardware devices for process control and data acquisition.