Compiling MATLAB M-Files for Usage Within C-Applications

Peter M. Roth, Martin Winter, Bernhard Jung
{pmroth, winter}@icg.tu-graz.ac.at, bernhard.jung@oefai.at

February 16, 2005

1 Introduction

1.1 General Notes

The following guideline describes the steps to do to a proper compilation of MATLAB source code (M-files) for integration into a C/C++ framework, i.e the FSP-Framework *zwork* for Computer Vision applications. All examples presented in this work can be downloaded at [19].

Due to some bugs and unresolved known problems in the MATLAB Compiler (Version 3.0) of MATLAB 6.5 Rel.13, it is essential to follow some restrictions and a handful of workarounds. The presented solutions are partially officially solutions published by *MathWorks Inc.* but most of them result from our observations during the attempts to compile and integrate our MATLAB code into stable applications under Linux.

1.2 Environment for that this guide is valid:

- MATLAB 6.5 Rel.13 (Linux, Windows)
- Windows XP : *Visual Studio 6.0*, *gcc 2.3.2*
- Linux: *gcc 2.95*, *gcc 3.2.2*, *gcc 3.3.1*

1.3 Structure of the Document

Section 2 gives a general overview of the usage of the MATLAB Compiler *mcc* that is valid for Windows as well as for Linux operating system. **Section 3** gives a detailed guideline for compiling M-files under Windows XP using *Visual Studio 6.0,* as **Section 4** gives a detailed guideline for compiling M-files under a Linux environment. In the end **Section 5** gives a short introduction how to include existing compiled M-files into a C framework.

2 MATLAB Compiler *mcc*

2.1 Preliminary Notes

The MATLAB Compiler *mcc* can translate M-files into C or C++ source code. The resulting files can be used in any of the supported executable types (MEX, stand-alone executable, library) by generating an appropriate wrapper file. A wrapper file contains the required interface between the *mcc*-generated code and a supported executable type. For compilation and linking any ANSI C or C++ compiler may be used. Therefore the *mcc* itself is not a ANSI C or C++ compiler! At the linking stage the resulting object files are linked against the MATLAB C/C++ Math and Graphics Libraries (see **Figure 1**).

To create a stand-alone C or C++ application, the the following steps are processed by the MATLAB Compiler *mcc*:

1. Translation of the given M-files into C or C++ source code.
2. Generation of additional C or C++ source code modules (wrapper files).
3. Invocation of a C or C++ compiler and linker.

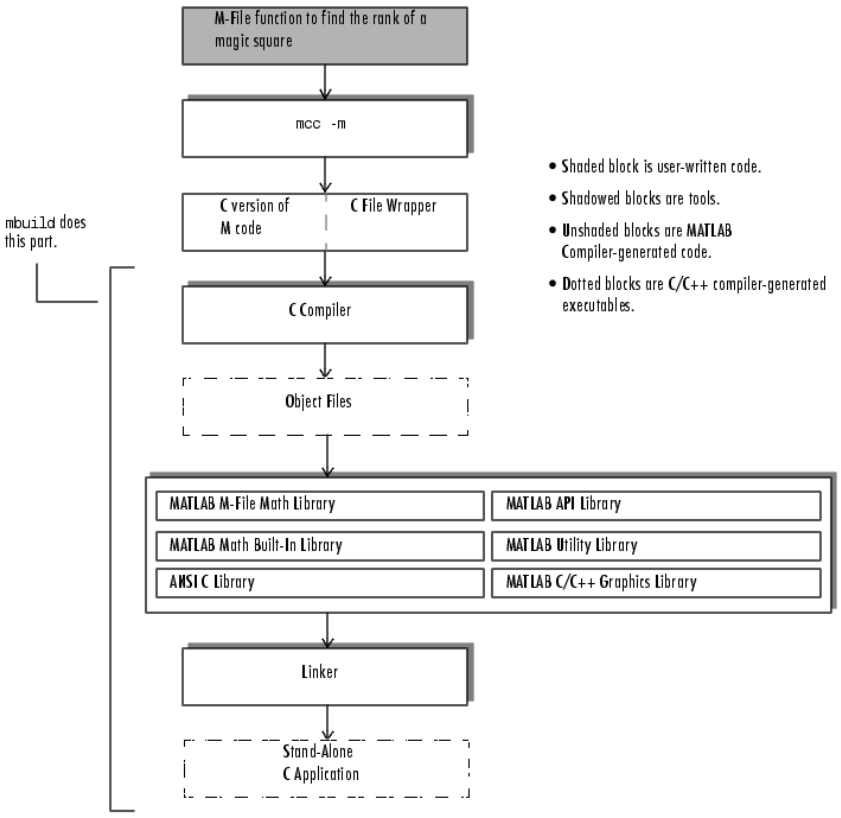

Figure 1: Creating a C stand-alone application [4]

2.2 *mcc* Compiler Option Flags

The following table gives a short overview of the most important compiler option flags needed for generation of C/C++ code. For a complete list of all option flags refer to [4].

Option	**Description**
-B *filename*	Passes a bundle of compiler settings stored in *filename* to the mcc. A set of predefined bundles are located in ***matlab_dir****/toolbox/compiler/bundles*.
-B sgl	Builds an C stand-alone application linked to the C/C++ Graphics Library. Equivalent to: *-m -W mainhg libmwsglm.mlib*
-c	When used with a macro option, generates C code only.
-g	Generates debugging information.
-G	Debug only, no execution.
-h	Compiles helper functions too.
-I *directory*	Adds *directory* to path.
-L *language*	Specifies output target language. *language* = {C, Cpp}
-m	Macro to generate a C stand-alone application. Equivalent to: *-W main -L C -t -T link:exe -h libmmfile.mlib*
-o *outputfile*	Specifies name/location of final executable.
-p	Macro to generate a C++ stand-alone application. Equivalent to: *-W main -L Cpp -t -T link:exe -h libmmfile.mlib*
-t	Translates M code to C/C++ code
-T *target*	Specify the desired output stage, available targets given in the list below: codegen Translates M-files to C/C++ files and generates a wrapper file (default). compile:exe Same as codegen plus compiles C/C++ files to object form suitable for linking into a standalone executable. compile:lib Same as codegen plus compiles C/C++ files to object form suitable for linking into a shared library/DLL. link:exe Same as compile:exe plus links object files into a standalone executable. link:lib Same as compile:lib plus links object files into a shared library/DLL.
-v	Verbose mode: displays compilation steps.
-W *type*	Controls the generation of function wrappers. *type* = {main, lib:*libname*}
libmwsglm.mlib	Link to this library whenever needed.

Table 1: *mcc* compiler option flags for generating C/C++ code

2.3 Limitations and Restrictions

There are some limitations and restrictions using the MATLAB Compiler. Therefore *mcc* (Version 3.0) cannot compile:

- M-files containing scripts (e.g. functions that call scripts can not be compiled).
- M-files that use objects.
- Calls to the MATLAB Java interface.
- M-files that use the MATLAB commands *input* or *eval* to manipulate workspace variables. *Input* and *eval* calls that do not use workspace variables will compile and execute properly.
- M-files that use the MATLAB command *exist* with 2(!) input arguments.
- M-files that load text files. Data exchange via files is best done using MAT-files. This ensures for example that files created under Linux would be readable under Windows.

As a consequence of some bugs and known unresolved problems there exist some workarounds to cope with this problems:

- Sometimes the *mcc* doesn't find, for some reasons, M-files of external toolboxes, even though the correct path was set within a *-I* statement. Therefore these M-files must be copied to the same directory as the main M-files that need this helper functions.
- Many functions of the Image Processing Toolbox, i.e. those functions for morphological operations, are not found by the compiler. To enable compiling this functions, these M-files must be copied into the working directory.
- Functions that use the MATLAB command *imlincomb*, i.e. many functions of the Image Processing Toolbox, would return a compiler error when generating C++ stand-alone applications. Therefore functions that use functionality of the Image Processing Toolbox can not be translated to C++ code in general. As a workaround, generate C stand-alone applications instead of C++ stand-alone applications.

2.4 Stand-Alone MATLAB Compiler

There are two versions of the MATLAB Compiler, one that is included in the MATLAB environment and a stand-alone version. The latter one may be used in a *Makefile* to automatize the build process (for some workarounds).

Unlike the MATLAB version, which inherits the paths from MATLAB, the stand-alone version has no initial path. So you will have to set a *-I* option (e.g. in a *Makefile*) for all directories you want to include into your search path. Another method is to set up a default path by making an *mccpath* file:

1. Create a text file containing the text *-I **dir_name*** for each directory you want on the default path, and name this file *mccpath*.

2. Place this file in your preferences directory. To do so, run the following commands at the MATLAB prompt:

   cd(prefdir)
   mccsavepath

The stand-alone version of the MATLAB Compiler searches for the *mccpath* file in your current directory and then your preferences directory. Note that you may still use the *-I* option on the command line to add other directories to the search path. Directories specified this way are searched after those directories specified in the *mccpath* file.

2.5 MBUILD

MBUILD compiles and links source code files that call functions in the MATLAB C/C++ Math Library and/or Graphics Library into a stand-alone executable or a shared library. Because MBUILD does nothing else than invoking the ANSI C or C++ compiler, we used MBUILD only for debugging reasons to determine which libraries we needed to link with:

*mbuild -n -I**usr_incl_dir**/ -L**usr_lib_dir** main.c -l**usr_lib_1** -l**usr_lib_2** ...*

When specifying the *-n* option, MBUILD sets up the compile and link command lines necessary to build a stand-alone application but does not execute the commands. View the output of *mbuild -n* to determine the list of libraries you must link your application with and the order in which you must specify them. Therefore MBUILD may be a good tool for porting C/C++ source code from Windows to Linux and vice versa.

For more detailed information about MBUILD refer to [4] or call *mbuild -help* from the Linux command line.

2.6 Tips for writing M-code that should be compiled

The following section summarizes some tips that facilitate writing MATLAB code that may be compiled later on:

- Any code that may be compiled should be encapsulated to functions.
- Since the mcc compiler prefers mex files over .m files. Be careful if you have both of them in the same directories, check ordering of directories specified with the -I parameter (e.g. toolbox/images/images/bwpack.m is an empty placeholder and preferred over mexglx function in the same directory).
- Persistent variables should be preferred over global variables, because name conflicts can thus be prevented. Do not forget to initialize persistent variables, e.g.:

```
if(isempty(X))
  X = 1;
end
```

3 Windows XP

3.1 Compiling M-files to Stand-Alone Applications

- Extract *MatLab_Compiling_Files_Win32.zip* to the local compiling directory. This ZIP-file contains some MATLAB M-files not found by the compiler although they are within the MATLAB's default path (see **Section A.1** for details). Additionally some dynamic linked libraries which are necessary for compiling due to an unknown reason (it seems, that the MATLAB Compiler uses these binaries as substitute for the missing corresponding source-code files). Note, that there is a *remove.bat* batch-file which removes the extracted files from the harddisk in an easy way.
- Check, that the folder with the M-files to compile is the current folder within MATLAB.
- Start compiling with the following flags:

  mcc -m -B sgl ***name_of_function****.m*

Result: An executable file, c-files, h-files and some other files are written to the actual compiling directory.

Note: For executing the *.exe it is necessary to extract *MatLab_Runtime_DLLs_Win32.zip* to the current directory or have them within in your local path. This ZIP-file contains all the necessary libraries needed during execution of compiled code (see **Section A.2** for details). This is due to the fact, that some MATLAB-functions are only available as binaries (no sourcecode available). Note, that there is a *remove.bat* batch-file which removes the extracted files from the harddisk in an easy way.

3.2 Compiling M-Files to a DLL:

- Extract *MatLab_Compiling_Files_Win32.zip* to the local compiling directory
- Check, that the folder with the m-file to compile is current folder within MATLAB
- Start compiling with the following flags:

  *mcc -t -W lib:****name_of_function****_lib -L C -T link:lib -h libmwsglm.mlib libmmfile.mlib* ***name_of_function****.m*

Result:

- the shared library **name_of_function**_lib.dll
- library info file **name_of_function**_lib.lib
- library header file **name_of_function**_lib.h
- some other c-files, h-files a.s.o

Notes:

- You can use the .c and .h files for compiling a library with another compiler.
- For executing an application it is necessary to have all *.dll files in the actual folder or within your local path-variable. For executing the *.exe it is necessary to extract *MatLab_Runtime_DLLs_Win32.zip* to the current directory or have them within in your local path.

3.3 Compiling an Stand-Alone Application using the Shared Library in Visual Studio

3.3.1 Necessary Includes for C/C++ Code Generation

Name: "matrix.h"

Path: C:\Programme\MATLAB\extern\lib\win32\microsoft\msvc60

3.3.2 Necessary Libraries for Linking

Name: libeng.lib libfixedpoint.lib libmat.lib libmatlb.lib libmatlbmx.lib libmex.lib libmmfile.lib libmwarpack.lib libmwlapack.lib libmwmcl.lib libmwrefblas.lib libmwservices.lib libmwsglm.lib libmx.lib libut.lib sgl.lib b

Path: C:\Programme\MatLab\extern\lib\win32\microsoft\msvc60

3.4 Compiling a MATLAB-Project with Visual Studio

Use the Project-Wizard integrated to Visual Studio during MATLAB installation.

3.5 Compiling using *gcc* for Windows

Since *gcc* is the standard compiler for Linux it would be desirable, e.g. for reasons of portability, to use it for compiling under Windows as well. For any reason, *gcc* is not officially supported my Mathworks Inc., so no suitable libraries[1] are distributed with MATLAB. Thus, you need to create import libraries for the dependent DLLs. Therefore go to the directory ***matlab_dir****extern**include*, where you will find the definition files *.def, and run the following command:

dlltool –def libname.def –dllname libname.dll –output-lib libname.a

This will create an import library for an existing library libname.dll that now can be linked against an application using *gcc*. An example bat-file creating these libraries is given in **Section A.3**.

Thus, a Makefile written for Linux (see **Listing 1**) can be used under Windows as well, if the following restrictions are conidered:

- Special UX compiler and linker flags must be removed:
  CFLAGS = -DUNIX -ansi -D_GNU_SOURCE -pthread
  LDFLAGS = -pthread
- When creating a library the output *libname.so* must be renamed to *libname.dll*.
- When creating an executable the target *exename* may be renamed to *exename.exe*. Some versions of the make command will add the exe-extension automatically if missing.

Note that building MATLAB API applications or standalone MATLAB applications with GCC on Windows are not supported; you will have to address any incompatibility issues that may arise.

3.6 Compiling using *lcc*

lcc is a free C-compiler that is shipped with the MATLAB package. For compiling M-files within the MATLAB environment you can choose either to use *lcc* or, if present, Visual C++. Of course you may use *lcc* for compiling in command line as well. Therefore you only need to link against the correct versions of the MATLAB libraries that are located in ***matlab_dir****extern**lib**win32**lcc*. If *lcc* was selected as standard compiler for MATLAB, use *mbuilt* to determine the correct parameters.

Be careful, *lcc* is a plain ANSI C compiler. Thus, you necessarily have to set the target language to C when converting the M-files! For more detailed information refer to [18] or run the *lcc*-help that is included in the MATLAB package: ***matlab_dir****sys**lcc*.

[1] The original Libraries (*.dll Files) are located in ***matlab_dir****bin**win32*.

4 Linux

4.1 Preliminary Notes

All steps described in this section may be done inside the MATLAB environment much easier. On the one hand side all the paths are set correctly and therefore the system would find all include files and all necessary libraries and on the other hand side the MATLAB Compiler would automatically link to needed libraries. Therefore it is more reasonable to do the generation of C/C++ source code, compiling and linking within the MATLAB environment if you only need an executable stand-alone application. But for our intended purpose, distributing C/C++ source files that should be compiled on a different machine, combined with unresolved problems concerning functions of the Image Processing Toolbox, there is no way around the following steps.

4.2 Libraries and Include Paths

For some reasons, i.e. for porting the MATLAB code to some other machines where no MATLAB environment is available, it is necessary to compile and link the M-files outside the MATLAB environment. Therefore the (relative) paths of libraries and include files must be passed to the compiler manually at compile stage. The include files needed for compilation are located in

matlab_dir*/extern/inlcude*

and the MATLAB Libraries needed for linking are located in

matlab_dir*/extern/lib/glnx86*
matlab_dir*/bin/glnx86*
matlab_dir*/sys/os/glnx86.*

The libraries located in the *sys/os* branch are are only needed if the *Motif*[2]-libraries on the local machine are missing or if the *libstdc++-libc* library on the local machine is different from the version included in the installed MATLAB system. For reasons of portability this libraries should be included in the linker path anyway. If the correct libraries are installed on the local machine, the locally installed libraries would be linked.

If the C source code generated by the *mcc* compiler should be compiled and linked on different machines the include files and the 3 library directories must be included in the distribution package.

[2]Motif is the standard library for graphical user interface for the MATLAB system under Linux.

4.3 Compiling M-Files to Stand-Alone Applications

It is assumed that there is an M-file that serves as main file, that maybe calls some helper functions. Moreover the main files may use functions of the MATLAB C/C++ Graphics Library. To generate an executable stand-alone application from these files 3 steps are necessary. Therefore it is convenient to use a *Makefile* rather than calling the compiler from the command line. For reasons of readability the compiling and linking step is given as a scratch of a *Makefile* [1], respectively (see **Listing 1**):

(i) Convert M-Files to C source code using the *mcc* (no compiling):

mcc -t -W main -h -c libmmfile.mlib ***main****.m*
mcc -t -B sgl -h -c libmmfile.mlib ***main****.m*

(ii) Compile C source files:

gcc -c -I$(INCL) $(CFLAGS) $(SRCS)

(iii) Link compiled object files against MATLAB libraries to an executable:

gcc $(LDFLAGS) -o test $(OBJS) $(LDLIBS)

```
...

SRCS = $(wildcard *.c)
OBJS = $(wildcard *.o)

INCL        =  MATLAB_DIR/extern/inlcude
LIBBIN      =  MATLAB_DIR/bin/glnx86
LIBEXTERN   =  MATLAB_DIR/extern/lib/glnx86
LIBSYS      =  MATLAB_LIB_DIR/sys/os/glnx86

LIBS    = -lmwsgl -lmwhg -lmwsglm -lmmfile -lmatlb -lmx -lmwservices\
          -lmex -lut -lm -lm

LDLIBS = -Wl,--rpath-link,$(LIBBIN),--rpath-link,$(LIBEXTERN),...
          ...--rpath-link,$(LIBSYS) -L$(LIBBIN) -L$(LIBEXTERN) -L$(LIBSYS) $(LIBS)

LDFLAGS       = -pthread -g -O
CFLAGS        = -DUNIX -ansi -D_GNU_SOURCE -pthread -g -O -DNDEBUG

...
```

Listing 1: Makefile

Annotation

(i) Using the second line the wrapper main-file would enable to link to the MATLAB C/C++ Graphics Library at linking stage.

(ii) The CLFAGS used for compiling can be obtained by calling the *mmc* in the verbose and debug mode. View the output of

mcc -t -W main -h -c -v -G libmmfile.mlib ***main****.m*

to determine the CLFAGS for compiling the code correctly. They should be the same for all M-files.

(iii) To determine the LDFLAGS and the necessary libraries for linking the code correctly, look again at the output of the *mmc* in the verbose and debug mode.

The linker option (initialized by *-Wl,*) *−−rpath-link,****lib_dir_name***[3] must be set for all directories that include libraries that the application is linked with. Otherwise some linkers would not find these shared libraries [2].

The usage of wildcard expressions is not a very pretty solution, but at compile time you would not know which C-files the MATLAB Compiler would generate. Another bothering consequence of using these wildcard expression in *Makefiles* is the fact that you cannot run the convert, compile and link target within the same make command. Since the wildcard expressions are evaluated when the *Makefile* is called, there would not exist any *.c or *.o files that can be compiled or linked. As a simple workaround to fix this, you can write a *Makefile* containing the targets convert, compile and link, that call a Sub-*Makefile* with the same target, respectively.

4.4 Compiling M-Files to Shared Libraries

For creating a shared library the same 3 steps as in **Section 4.3** are necessary. Only some compiler option flags are different. The *Makefile*-notation refers again to **Listing 1**:

(i) Convert M-Files to C source code using the *mcc* (no compiling):

*mcc -v -t -W lib:lib****name_of_lib*** *-L C -T link:lib -h -c libmmfile.mlib* ***fun_2****.m* ***fun_1****.m ... .*
*mcc -v -t -W lib:lib****name_of_lib*** *-L C -T link:lib -h -c libmmfile.mlib libmwsglm.mlib* ***fun_2****.m* ***fun_1****.m ...*

(ii) Compile C source files:

gcc -c -fPIC -I$(INCL) $(CFLAGS) $(SRCS)

(iii) Link compiled object files against MATLAB libraries to a library:

gcc $(LDFLAGS) -shared -o libtest.so $(OBJS) $(LDLIBS)

[3]The ... are inserted for readability reasons only. Using the *-Wl* option you must not use blanks, otherwise the option would not be recognized by the linker. Therefore you cannot insert a manual line break using a slash, since this will unfortunately insert a blank at the position of the line break.

Annotation

(i) Libraries under Linux are restricted to the prefix *lib* in the library name. Using the second line the library will be linked to the MATLAB C/C++ Graphics Library at linking stage.

(ii) The *-fPIC* option is to tell the compiler to create Position Independent Code (create libraries that use relative addresses), so a compiled library can be loaded multiple times at run-time [2].

(iii) Using compiler the flag *-shared* a shared library will be linked.

4.5 Distribution of MATLAB Executables

To distribute a stand-alone application for Linux, you must create a package containing these components:

(i) The stand-alone executable

(ii) The contents, if any, of a directory named *bin*

(iii) Any MEX-files used by the application

(iv) The MATLAB Run-Time Libraries

(v) Any other library that was linked with the application

Annotation

(ii) The directory *bin* may contain the *.fig files that are required for correct displaying the toolbar of a graphical application. This directory must be placed in the same directory as the executable.

(iii) To determine which MEX-files would be needed, search for C-files called **_mex_interface_*.c* generated by the MATLAB Compiler. The MEX-files appertaining to these C-files must be placed in the subdirectory called *bin*. MEX-files installed in a private directory in MATLAB must also be installed in a private directory within the directory *bin*. For an example see [11]. If this still does not work, all those the MEX-files will have to be copied into the same directory as the executable. Unlike described in [11] this even works for functions of the Image Processing Toolbox.

(iv) The MATLAB Run-Time libraries can be distributed using the executable *mglinstaller*[4], which can be found under ***matlab_dir****/extern/lib/glnx86*. This application installs the MATLAB Math and Graphics Run-Time Libraries.

Once the installer has finished, the directory ***matlab_rt_dir****/bin/glnx86* must be added to the *LD_LIBRARY_PATH* environment variable as well as the parent path of your own MATLAB libraries.

[4]For privacy policy and license agreement for this installer refer to [8].

5 Include compiled MATLAB functions to C-code

5.1 Preliminary Notes

C Code only

Generally it is possible to generate C code as well as C++ code from existing MATLAB files. But there are still some problems in compiling functions that need the C/C++ Graphics Library (e.g. [10, 12, 13, 14]). Although there exist some workarounds, there are still some unresolved problems remaining. So we can only generate C code from our MATLAB functions.

Naming Convention

All MATLAB functions translated to C code follow the same naming conventions. The name of the C functions start with the prefix *mlf* followed by original function name, whereby the first letter is capitalized. For example the translated C function of the well known function *svd* (singular value decomposition) is called *mlfSvd*.

Include Libraries

The simplest way to include MATLAB functions into a C environment is to link them together into a shared library. The result of the linking process are a shared library (*libname.dll*/Windows, *libname.so*/Linux) and a header file *libname.h*. To enable the execution of the library's source code the library must be initialized. Therefore the the MATLAB Compiler generated the functions ***libname****Initialize()* and ***libname****Terminate()*, which can be compared to constructor and destructor speaking in terms of C++ programming. For an example see **Listing 2**.

If no libraries were generated you can link the compiled object files to your executable as well. If so you have to add a ***name_of_function****Initialize()* and ***name_of_function****Terminate()* command for every compiled function you call to your C main file.

Include MEX-Files

When MEX files are reported to be missing, they are not in the path (i.e., for many toolboxes the distribution of MEX-files as described in **Section 4.5** doestn't work). To add them to the path, use the following code fragment must be included into the (converted) C-code:

```
const char *paths[]  = {"../MATLAB/MEX", "../MATLAB/MEX/iofun"};\
mclAddPaths(2, paths);\
```

If *filename.mexglx* missing is reported, just add the path where the mexglx file resides. If ***foo****/private/filename.mexglx* is reported to be missing, add foo to the path (and put the MEX-file into *foo/private*).

Linking Library Bug

Due a bug in the *mcc* it is not possible to link more than one *mcc* generated library to a C program directly on a Linux system. The variables *_lib_info* and *_reference_count* are defined *extern* for each library. Therefore initializing a second library overrides

the settings of the first one. This causes a segmentation fault at run-time. As a workaround for this problem, these extern defined variables must be renamed to a unique name, e.g., by running PERL (see **Listing 9**) script within a Makefile.

Generating Unsupported Libraries
Nevertheless linking of libraries against the C/C++ Graphics Library is not officially supported by Mathworks [10, 13], we were able to create such libraries and link them to pure C functions and got stable executables.

Non Critical Compiler Warnings
Resolving the compile errors e.g. by creating getline.m dummy files does not work. An attempt to just use the required functionality (computeArea) was also not successful, because computeArea is an internal function.

However, ignoring the compilation errors leads to a working solution, because e.g. getline is referenced, but never used in the executed code

Functions of Image Processing Toolbox
Many functions of the Image Processing Toolbox (i.e., morphological operations) use MATLAB objects that can not be compiled using *3.0*. A workaround for this problem is to copy all the needed files (including the private directory) into the compile path of MATLAB (or any other directory and using -I option). For example, for compilation of function using the strel function y copying all files from @strel . For detailed information see [9, 17]

unresolved symbol glXGetProcAddressARB
incompatible GLU/GLUT version! The glu/glut libraries in the MATLAB distribution are different from system-wide installed ones. Try to remove the libraries from the MATLAB runtime directory.

"Strange errors"
At runtime there can be some strange errors (supposed to be side-effects of *mxGetPr* (on global variables?) when reading results from a structure returned by a compiled MATLAB function:

- mwmem.c:1218: Assert : hdr->in_use != 0, "Attempt to free previously freed memory"
- ERROR: Matrix dimensions must agree.

They occur on calling a compiled MATLAB function and seem to depend on operations performed on results of the function from a previous call.

A work-around for this problem is to use *mxDuplicateArray*. See the following example:

```
// may have side-effects
// temparray=mxGetPr(mxGetField(res, i, "Centroid"))
// b.Centroid.X = (int)*mxGetPr(tempArray);
// b.Centroid.Y = (int)*(mxGetPr(tempArray)+1);
```

```
// known to work
tempArray = mxDuplicateArray(mxGetField(res, i, "Centroid"));
b.Centroid.X = (int)*mxGetPr(tempArray);
b.Centroid.Y = (int)*(mxGetPr(tempArray)+1);
mxDestroyArray(tempArray);
```

checknargin

ERROR: Reference to unknown function 'checknargin' from FEVAL in stand-alone mode. This error happens because the FEVAL performs a dynamic function invocation and the MATLAB compiler cannot resolve this dependency. To workaround, either include the called function explicitly in the mcc compiler statement or add the following comment to the source (see Technical Solutions, Solution Number: 1-1AIEJ):

5.2 Example

The following simple example shows, how to integrate a library generated from MATLAB functions into a C main-file.

```
1  #include "matlab.h"
2  #include "libsimple.h"
3
4  int main(int argc, char* argv[])
5  {
6      /* define variables */
7      double A = 2; double B = 3;
8      double C = 0; double D = 0;
9      char Head[] = "This is a Test String.";
10
11     mxArray *pmxA, *pmxB, *pmxC, *pmxD, *pmxHead;
12
13     /* enable automated memory management */
14     mlfEnterNewContext(0, 0);
15
16     /* initialize MatLab environment */
17     libsimpleInitialize();
18
19    /* initialize mxArrays */
20     pmxC  = mxCreateDoubleMatrix(1,1,mxREAL);
21     pmxD  = mxCreateDoubleMatrix(1,1,mxREAL);
22
23     /* copy C double value to MatLab interface array */
24     pmxA = mlfScalar(A);
25     pmxB = mlfScalar(B);
26
27     /* assign values to mxArrays */
28     mlfAssign(&pmxHead, mxCreateString(Head));
29     mlfAssign(&pmxC, mlfSimple(&pmxD, pmxA, pmxB, pmxHead));
30
31     /* read data form MatLab interface arrays back to C variables */
32     C = mxGetScalar(pmxC);
```

```
33      D = mxGetScalar(pmxD);
34
35      /* print values */
36      printf("Output by C API Interface Function\n");
37      mlfPrintMatrix(pmxC);
38      mlfPrintMatrix(pmxD);
39
40      printf("Output by C Native Function\n");
41      printf("%f   %f\n%f   %f\n", A, B, C, D);
42
43      /* free memory of MatLab interface arrays */
44      mxDestroyArray(pmxA); mxDestroyArray(pmxB);
45      mxDestroyArray(pmxC); mxDestroyArray(pmxD);
46
47      /* terminate the MatLab environment */
48      libsimpleTerminate();
49
50      /* disable automated memory management */
51      mlfRestorePreviousContext(0, 0);
52
53      return 0;
54  }
```

Listing 2: main.c

```
function [C, D] = simple(A, B, TestStr)

fprintf([TestStr,'\n\n']);

C = A + B;
D = A * B;

return;
```

Listing 3: simple.m

```
This is a Test String.

Output by C API Interface Function
     5
     6

Output by C Native Function
2.000000   3.000000
5.000000   6.000000
```

Listing 4: Output of the Test Program

5.2.1 Additional Explanation for the Example Source Code

line 1: The include file *matlab.h* provides the interfaces for the MATLAB C Interface API.

line 2: The include file *libsimple.h* is created by the *mcc* compiler and must be included in the beginning of the C-source code.

line 11: The data interface between any MATLAB functions and the C-source Code is defined by a pointer to the data type *mxArray* (for more details see: [6]).

line 24: A more general way to convert double values to mxArrays, is to use a *mxCreateDoubleMatrix* (see: line 20) and then make a *memcpy* operation. This would be valid for all arrays:

$$memcpy(mxGetPr(pmx\mathbf{DoubleVar}), \&\mathbf{DoubleVar}, \mathbf{n}*sizeof(double));$$

line 28: When working with MATLAB-functions the equal operator should be replaced by the function *mlfAssign*. This ensures a robust memory management. *mlfAssign* returns a pointer to the target array.

line 29: If a function has more than one output parameter, these parameters are passed to the function as the first input parameter of the type ***mxArray*. All other input parameters are of the type **mxArray*.

Some other examples for mixing M-files and C-code can be find in [4]. For a complete list of MATALB C/C++ API Functions with detailed short examples refer to [5, 7].

5.2.2 Compile the Example Source Code

Referring again to **Listing 1** and following the steps in **Section 4.4** an instruction for compiling and linking the example in **Section 5.2** is given below:

(i) Convert M-Files to C and create a library:

```
mcc -v -t -W lib:libsimple -L C -T link:lib -h -c
    libmmfile.mlib simple.m
gcc -c -fPIC -I$(INCL) $(CFLAGS) $(SRCS)
gcc $(LDFLAGS) -shared -o libsimple.so $(OBJS) $(LDLIBS)
```

(ii) Compile C main source file and link it to library:

```
gcc -c -I$(INCL) $(CFLAGS) main.c
gcc $(LDFLAGS) -o example main.o -lsimple $(LDLIBS)
```

Annotation

(ii) After the main file was compiled, it is linked to the created library *libsimple* as well to the origin MATLAB Libraries. If you didn't copy your library into a directory the linker will find it, you have to add a *–rpath-link* as well as a *-L* option for the parent directory of your library.

5.3 Compile a Converted C Program on an Other Machine

(i) Make a package containing the MATLAB include files and the MATLAB libraries described in **Section 4.2**, the C source code generated by *mcc*, any MEX-files that are needed and the *mglinstaller*.

(ii) Write a Makefile referencing to the MATLAB include files and libraries. To determine the libraries you have to link with your library call *mmc* in the verbose and debug mode(see **Section 4.3**). To determine the libraries you have your C-program to link with call mbuild (see **Section 2.5**).

(iii) Compile and link your library using the compiler and linker settings of **Section 4.3**.

(iv) Run the *mglinstaller*.

(v) Add the directory ***matlab_rt_dir****/bin/glnx86* created by the *mglinstaller*to the *LD_LIBRARY_PATH* environment variable as well as the parent path of your own MATLAB libraries.

5.4 Simple C++ Template for a MATLAB Library

In the following we show a template for a wrapper-class for simply including compiled MATLAB source code into an existing system. **Listing 5** is a scratch of a simple main function that calls a constructor, runs a function defined within the wrapper class and finally calls a destructor. The class according to the created instance is given by **Listing 6**.

The main advantage of using such structure is that the programmer at C++ level must not have any knowledge about the MATLAB-world. Initializing and destructing the MATLAB-environment, converting the data to the correct format and calling functions of the MATLAB C/C++ API is done within the wrapper-class. The same thing can be done within only one C-function as well, but therefore the MATLAB environment has to be initialized every time the function is called. For return values *call by reference* is used, since the MATLAB functions often return more than one parameter. This is not a must, but would be easier.

```
#include "wrapper.h"

int main(int argc, char* argv[])
{
    // create the component
    Template_Component* pComp = new Template_Component();

    // create the matrices in C-style
    int Dim1 = 2; int Dim2 = 3;
    double Matrix[2][3] = {{1,2,3},{4,5,6}};
    double Result[2][3] = {0,0,0,0,0,0};
```

```
    // call the service
    pComp->subscribe_demo((double*)Matrix, Dim1, Dim2, (double*)Result1);

        ...

    // display the result in C-style
    printf("Sourcematrix:\n");
    printf("%f %f %f\n",Matrix[0][0], Matrix[0][1], Matrix[0][2]);
    printf("%f %f %f\n\n",Matrix[1][0], Matrix[1][1], Matrix[1][2]);

    printf("Result Matrix:\n");
    printf("%f %f %f\n",Result1[0][0], Result1[0][1], Result1[0][2]);
    printf("%f %f %f\n\n",Result1[1][0], Result1[1][1], Result1[1][2]);

        ...

    // delete the component
    delete pComp;

    return 0;
}
```

Listing 5: main.cpp

```
#include "wrapper.h"
#include "libcppdemo.h"

// constructor
Template_Component::Template_Component()
{
    //  initialize MATLAB environment
    libcppdemoInitialize();
}

//destructor
Template_Component::~Template_Component()
{
    // terminate the MatLab environment
    libcppdemoTerminate();
}

// example for simple member function
void Template_Component::subscribe_demo(double* Matrix, ... , double* Result)
{
    // define variables
    int i,j;
    double* Matrix_M = new double[Dim1*Dim2];
    double* Result_M = new double[Dim1*Dim2];
    mxArray *pmxInput, *pmxNumber, *pmxResult, *pmxResult2;

    // convert the input-matrix from C-style to MATLAB-style
    for (i=0; < Dim1; i++) {
        for(j=0; j<Dim2; j++) {
            Matrix_M[j*Dim1+i] = Matrix[i*Dim2+j];
        }
```

```
    }

    // call compiled MATALB function
    mlfAssign(&pmxInput,  mlfDoubleMatrix(Dim1, Dim2, Matrix_M, NULL));
    mlfAssign(&pmxResult, mlfDoubleMatrix(Dim1, Dim2, Result, NULL));
    mlfAssign(&pmxResult, mlfDemo(pmxInput, pmxNumber));

    // read data from MATLAB interface arrays back to C variables
    double* pResult = mxGetPr(pmxResult);
    double* pError = mxGetPr(pmxErrorcode);

    for (i=0; < Dim1; i++) {
        for(j=0; j<Dim2; j++) {
            Result_M[i*Dim2+j] = pResult[j*Dim1+i];
        }
    }
    memcpy(Result, Result_M, Dim1*Dim2*sizeof(double));

    // free memory: MATLAB-arrays, temporary MATLAB-style matrices
    delete[] Matrix_M;
    mxDestroyArray(pmxInput);
       ...
}
```

Listing 6: wrapper.cpp

5.5 Creating image matrices from C code

The following code fragment creates an image array. Watch out for differences between uint8 and double matrices! DOUBLE requires values to be in the range of 0 to 1, UINT8 in the range of 0 to 255. (rgb2yuv takes DOUBLE matrices with values greater 1.0 without complaining.)

```
mxArray *arr;

int dims[3] = {height, width, 3};
arr = mxCreateNumericArray(3,dims,mxDOUBLE_CLASS,mxREAL);

double *ptr = mxGetPr(arr);
int x,y,pos;

for (y=0;y<height;y++)
{
    for (x=0;x<width;x++)
    {
        ... // get r,g,b from somewhere

        dims[0] = y;
        dims[1] = x;
        dims[2] = 0;
        pos = mxCalcSingleSubscript(arr,3,dims);
        *(ptr+pos) = r;
```

```
        dims[2] = 1;
        pos = mxCalcSingleSubscript(arr,3,dims);
        *(ptr+pos) = g;
        dims[2] = 2;
        pos = mxCalcSingleSubscript(arr,3,dims);
        *(ptr+pos) = b;
    }
}
```

Listing 7: Code fragment: image matrix

Appendix A

A.1 Needed M-files for Compiling

We needed to copy all files given in the list below to the local working directory when generating C-source code:

applylut.m	axes2pix.m	bweuler.m
bwlabeln.m	bwlabelnmex.dll	bwmorph.m
bwpack.dll	bwpack.m_	bwunpack.dll
bwunpack.m_	checkconn.m	checkinput.m
checknargin.m	checkstrs.m	conndef.m
dataread.dll	disp.m	display.m
fspecial.m	getcurpt.m	getheight.m
getimage.m	getline.m	getneighbors.m
getnhood.m	getpts.m	getsequence.m
im2double.m	imadd.m	imcomplement.m
imdilate.m	imerode.m	imfill.m
imfilter.m	imfilter_mex.dll	imformats.m
imftype.m	imhist.m	imhistc.dll
imlincomb.m	imlincombc.dll	imopen.m
imread.m	imreconstruct.m	imshow.m
msubtract.m	imwrite.m	ntline.m
ptgetpref.m	ptprefs.m	ptregistry.m
sflat.m	utbridge.m	utclean.m
utdiag.m	utdilate.m	uterode.m
utfatten.m	utfill.m	uthbreak.m
utiso.m	utmajority.m	utper4.m
utper8.m	utremove.m	utshrink.m
utsingle.m	utskel1.m	utskel2.m
utskel3.m	utskel4.m	utskel5.m
utskel6.m	utskel7.m	utskel8.m
utspur.m	utthin1.m	utthin2.m
utthin3.m	utthin4.m	kconstarray.m
morphmex.dll	morphop.m	num2ordinal.m
padarray.m	reflect.m	regionprops.m
rgb2gray.m	roipoly.m	stem.m
strel.m	strelcheck.m	strread.m
empname.m	translate.m	truesize.m
urlwrite.m	usejava.m	xlim.m
ylim.m	zlim.m	

These files are packed within *MatLab_Compiling_Files_Win32.zip* mentioned in **Section 3** and can be found within your local MATLAB installation path (standard MATLAB toolbox, Image Processing Toolbox):

A.2 Needed MEX-files for compiling

Additionally to the functions that exist as M-file you would need to include those functions that are compiled to a MEX-file. The list below gives all MEX-files (if using Linux replace *dll* by *mexglx*) we needed in order to compile our applications:

bwlabelnmex.dll
bwpack.dll
bwunpack.dll
dataread.dll
imfilter_mex.dll
imhistc.dll
imlincombc.dll
morphmex.dll

These files are packed within *MatLab_Runtime_DLLs_Win32.zip* mentioned in **Section 3** and can be found within your local MATLAB installation path (standard MATLAB toolbox, Image Processing Toolbox).

A.3 Create *gcc* Libraries (Windows)

Listing 8 shows an example batch-file for generating import libraries that can be used with *gcc* for Windows. The created libraries are moved into a separate folder, that may be included into the system variable *PATH* to ensure that the libraries are found at runtime.

```
dlltool --def libeng.def --dllname libeng.dll --output-lib libeng.a
dlltool --def libfixedpoint.def --dllname libfixedpoint.dll --output-lib libfixedpoint.a
dlltool --def libmat.def --dllname libmat.dll --output-lib libmat.a
dlltool --def libmatlb.def --dllname libmatlb.dll --output-lib libmatlb.a
dlltool --def libmex.def --dllname libmex.dll --output-lib libmex.a
dlltool --def libmmfile.def --dllname libmmfile.dll --output-lib libmmfile.a
dlltool --def libmwmcl.def --dllname libmwmcl.dll --output-lib libmwmcl.a
dlltool --def libmwservices.def --dllname libmwservices.dll --output-lib libmwservices.a
dlltool --def libmwsglm.def --dllname libmwsglm.dll --output-lib libmwsglm.a
dlltool --def libmx.def --dllname libmx.dll --output-lib libmx.a
dlltool --def libut.def --dllname libut.dll --output-lib libut.a
dlltool --def mclcom.def --dllname mclcom.dll --output-lib mclcom.a
dlltool --def mclcommain.def --dllname mclcommain.dll --output-lib mclcommain.a
dlltool --def mclxlmain.def --dllname mclxlmain.dll --output-lib mclxlmain.a
dlltool --def sgl.def --dllname sgl.dll --output-lib sgl.a

md gcclibs
move *.a .\gcclibs
```

Listing 8: gengcclibs.bat

A.4 Helper Scripts for Compiling

Listing 8 shows an example batch-file for generating import libraries that can be used with *gcc* for Windows. The created libraries are moved into a separate folder, that may be included into the system variable *PATH* to ensure that the libraries are found at runtime.

```
#!/usr/bin/perl

$nbr = $ARGV[0];
open IN, $ARGV[1];

while (<IN>) {

  s/_lib_info([;) ,\n])/_lib_info$nbr$1/g;
  s/_reference_count([;) ,\n])/_reference_count$nbr$1/g;
  print;
}

close IN;
```

Listing 9: changestatic.pl

```
/     mxArray \* pref = NULL;/a\
\
\
    /* add mex-path manually */\
    const char *paths[]  = {"../MATLAB/MEX", "../MATLAB/MEX/iofun"};\
    mclAddPaths(2, paths);\
\
```

Listing 10: insmex.sed

References

[1] Stallman R.M., McGrath R., Smith P., *GNU Make*, Free Software Foundation, 2002

[2] Stallman R.M., *Using and Porting the GNU Compiler Collection (v2.95)*, Free Software Foundation, 1999

[3] The MathWorks, Inc., *MATLAB C/C++ Graphics Library User's Guide (Version 2)*, 2001

[4] The MathWorks, Inc., *MATLAB Compiler User's Guide (Version 2)*, 2002

[5] The MathWorks, Inc., *MATLAB C Math Library Reference (Version 2)*, 1990

[6] The MathWorks, Inc., *MATLAB C Math Library User's Guide (Version 2)*, 1999

[7] The MathWorks, Inc., *MATLAB External Interfaces Reference (Version 6)*, 2002

[8] About The MathWorks - Policies & Statements, *http://www.mathworks.com/company/aboutus/policies_statements*

[9] MATLAB Product Support 1607 - The Compiling Functionality of the Image Processing Toolbox, *http://www.mathworks.com/support/tech-notes/1600/1607.html*

[10] MATLAB Technical Solutions Number: 1-1BE7I, *http://www.mathworks.com/support/solutions/data/1-1BE7I.html*

[11] MATLAB Technical Solutions Number: 27365, *http://www.mathworks.com/support/solutions/data/27365.html*

[12] MATLAB Technical Solutions Number: 29113, *http://www.mathworks.com/support/solutions/data/29113.html*

[13] MATLAB Technical Solutions Number: 30672, *http://www.mathworks.com/support/solutions/data/30672.shtml*

[14] MATLAB Technical Solutions Number: 32633, *http://www.mathworks.com/support/solutions/data/32633.shtml*

[15] MATLAB Technical Solutions Number: 34757, *http://www.mathworks.com/support/solutions/data/34757.html*

[16] MATLAB Solution Number: 1-19YRP, *http://www.mathworks.com/support/solutions/data/1-19YRP.html*

[17] MATLAB Solution Number: 1-18LLM, *http://www.mathworks.com/support/solutions/data/1-18LLM.html*

[18] *http://www.cs.virginia.edu/~lcc-win32*

[19] *http://www.icg.tu-graz.ac.at/~pmroth/M2C/examples*