From Simulink model to DLL
A tutorial

By

Roland Pfeiffer

Abstract

This document attempts to give a hands-on description on how to make a DLL from a controller built in Simulink. This makes it possible to develop advanced controllers in Simulink, and then use them from an arbitrary application.

Persons with knowledge in the C programming language should have no problems converting the steps in this tutorial to build Linux/Unix libraries, thus expanding the scope of this document to be applicable not only on Windows.

1 Purpose

The purpose of this document is to describe how to use a controller built in Matlab/Simulink from other software. The target platform for this tutorial is Windows. The steps described here should be easily modified to use it on Linux/Unix-platforms. A description of how the controller built in Simulink can be called from Delphi is also included.

2 Validity of this document

The method described in this tutorial has not been extensively tested. This means that this tutorial can contain ambiguities or outright errors. Reports of errors to the author are much appreciated.

The steps in this tutorial have been performed using Matlab v7 and Simulink v6. It is quite possible, but not guaranteed, that the tutorial is valid for other versions as well.

3 Prerequisites

It is assumed that the reader is already familiar with Matlab/Simulink. Thus it will not be described in detail how to build the Simulink model mentioned in this tutorial. It is also desirable to have some familiarity with the C programming language. However, it should be possible for anyone with some kind of programming experience to perform the changes necessary.

It is of course also required to have Matlab/Simulink installed.

4 Tutorial

The following is an attempt to lead the reader from scratch to a dll that can be used from the software of choice. In several places in this tutorial it is possible to find statements looking like this *($STATEMENT)*. These statements (including the braces) are depending on either the name of the model being used or the platform used. How to replace these statements should be obvious.

4.1 Preparations

To be able to perform the operations described in later sections it is necessary to add a few directories into the search path of the computer. Add the following to the *path* environment variable:

($MATLABHOME)\rtw\bin\win32

($MATLABHOME)\sys\lcc\bin

Note that ($MATLABHOME) should be replaced with the directory in which Matlab is installed.

For information on how to change the path variable see the Windows help function.

4.2 Building a controller

The first step is to build a controller in Simulink. The appearance of the controller should be similar to the one illustrated in Fig. 1.

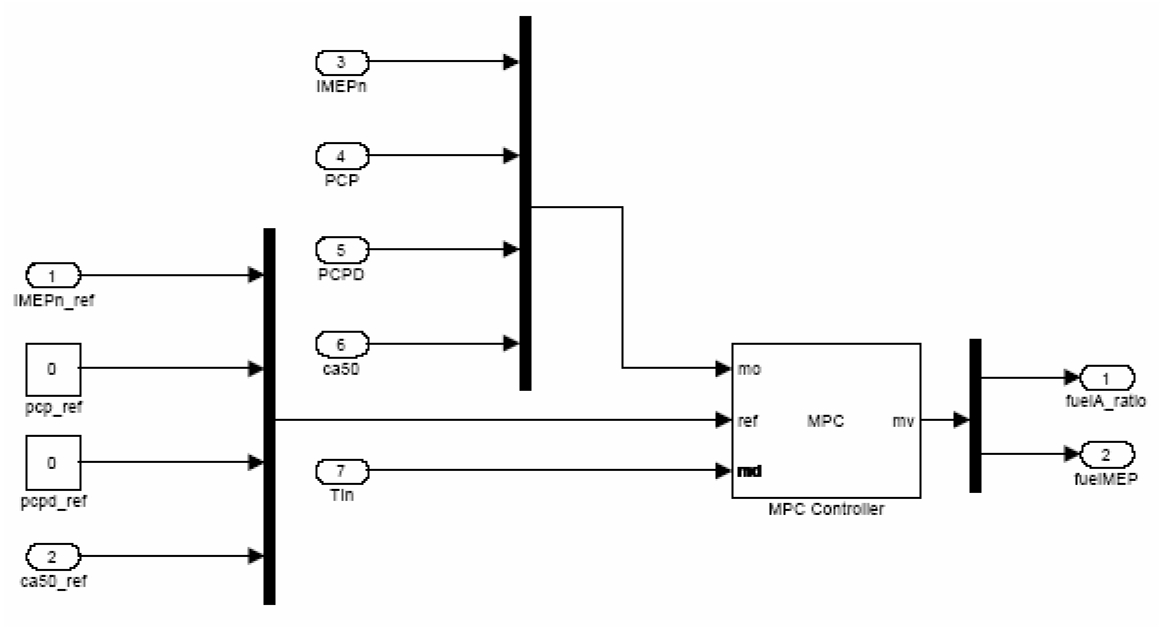

Fig. 2 The controller used in this tutorial.

As can be seen in the figure above the model should make use of external in and outputs. This means that all signal routing should be handled by the calling software. The number of inputs and outputs to/from the controller is arbitrary. The illustrated controller is of MPC-type. However this is of course not a necessity, it could be of any desired type. In fact it does not have to be a controller at all.

4.3 Generating C code

When the controller has been given the desired form (and preferably tuned and tested in another Simulink model) it should be compiled into C-code. For this purpose Real Time Workshop is used. The following are the steps to take to obtain the desired results. All instructions are intended to be carried out from the Simulink model window.

1. Expand the *Tools* menu.
2. Expand the *Real Time Workshop* menu item.
3. Select the *Options* menu item. This will open the *Configuration Parameters* window.
4. In the leftmost window; select *Solver*.
5. Check that *Simulation time: Stop time* is set to Inf.
6. Check that *Solver options: Type* is set to Fixed-step.
7. In the leftmost window; select *Real-Time Workshop: Interface*.
8. Check that Software environment: Target floating point math environment is set to ANSI-C.
9. Select the OK button. This will close the *Configuration Parameters* window.
10. Expand the *Tools* menu.
11. Expand the *Real Time Workshop* menu item.
12. Select the *Build model* menu item.

Assuming that no problems occurred two new directories have been created in the directory where the model is situated. These two directories are named *slrpj* and *($NAMEOFSIMULINKMODEL)_grt_rtw*.

4.4 Writing a wrapper

The file *grt_shell.c* (the source code can be viewed in section 6.1) wraps around the controller developed in Simulink. The original source code generated by Real Time Workshop contains no functionality that allows it to be called from external functions; this functionality is provided by this file. It should be noted that the version in section 6.1 is applicable to the example implementation only. This is unless the controller has the same number of; and name for the inputs and outputs. To work on another controller (with other inputs and outputs) it has to be modified slightly.

It is not a requirement for the shell functions to look the way they do in this tutorial. However, it is the author's opinion that the design chosen here is a good one.

The file grt_shell.c is actually a modified version of the file grt_main.c which can be found in ($MATLABHOME)\rtw\c\grt. In most cases it should be sufficient to modify the existing version of grt_shell.c. If mere modifications do not suffice; a new version of the file will have to be written. Such a procedure is not described in this tutorial but should be possible to achieve by studying example file supplied here.

The following list contains the steps to take to modify grt_main.c to use x inputs named in_1, in_2 … in_x and y outputs named out_1, out_2 … out_y.

1. Open the file *grt_shell.c* in your favorite text editor.
2. locate the function *getControllerOutput*.
3. Alter the lines checking the number of input and output arguments. Instead of checking that the number of inputs are 7 and the number of outputs are 2, they should instead test that the number of inputs are x and the number of outputs are y.
4. This function makes use of a struct[1] called *($NAMEOFMODEL)_U*. This struct handles the inputs to the model. For the example implementation the inputs are called IMEPn_ref, ca50_ref and so on. These lines should be altered to look something like this:
   ```
   ($NAMEOFMODEL)_U.in_1 = inputs[0];
   ($NAMEOFMODEL)_U.in_2 = inputs[1];
   …
   ($NAMEOFMODEL)_U.in_x = inputs[x-1];
   ```
   It is important to keep track of what order the input arguments are given since they only have numbers and no names.
5. Close to the end of the same function the outputs from the calculations are entered into the output array. As with the inputs; the outputs are stored in a struct, for the outputs the name of the struct is *($NAMEOFMODEL)_Y*. The elements of this struct have the names of the outputs in the Simulink model. This means that the lines assigning values to the output array should be changed to look like as follows:
   ```
   outputs[0] = ($NAMEOFMODEL)_Y.out_1;
   ```

[1] A structure for handling data in the C programming language

```
outputs[1] = ($NAMEOFMODEL)_Y.out_2;
…
outputs[y-1] = ($NAMEOFMODEL)_Y.out_y;
```

As with the inputs it is important to keep track of the order in which the outputs are entered into the output array.

6. At this point the file should be modified to suit the new configuration. Save it and go on to the next section.

4.5 Creating a definition file

When creating a DLL it is necessary to make use of a definition file where the functions to be exported are listed. In this case only three functions are exported and unless the function declarations in *grt_shell.c* are altered the content of *controller.def* will not have to be changed. The definition file for the example implementation is located in section 6.2.

4.6 Creating the DLL

At this point the controller built in Simulink has been compiled into C-code. The next step is to add some functions making the resulting DLL usable. Following is a list of steps to take to build the DLL

1. Copy the file *grt_shell.c* into the *($NAMEOFMODEL)_grt_rtw* directory.
2. Copy the file *controller.def* into the *($NAMEOFMODEL)_grt_rtw* directory.
3. Open a command prompt.
4. In the command prompt go to the *controller_grt_rtw* directory.
5. Enter the command *gmake –f ($NAMEOFMODEL).mk*. This will compile the generated C code into an executable with the same name as the Simulink model. It will also generate object files which will be used later steps.
6. Look at the text written to the command window by the command given in the previous step. Written to the command prompt are the commands that *gmake* used to compile and link the executable mentioned above. Locate the command starting with *($SOMEPATH)\lcc –c -Fogrt_main.obj* and ending with *($ANOTHERPATH)\grt_main.c*. Copy this command and alter it to start with *lcc –c -Fogrt_main.obj* and end with *grt_shell.c*. Enter this command in the command prompt. This will create a new version of the objective file grt_main.obj.
7. Now it is time to perform the linking creating the actual DLL. To do this, locate the last command given by *gmake*. This command should start with *($SOMEPATH)\lcclnk –s* and end with *($ANOTHERPATH)\rtwlib_lcc.lib*. Begin by altering the start from *($SOMEPATH)\lcclnk* to *lcclnk*.
8. Some distance into this command it is possible to find the following *–o ..\($NAMEOFMODEL).exe ($NAMEOFMODEL).obj*. Replace this with the following instead *–dll ($NAMEOFMODEL).obj*.
9. At the end of the command; add *controller.def*.
10. Run the command. This command will link the object files created in previous steps to a DLL called *controller.dll*.

5 Using the DLL

The DLL created in previous sections can be used from the programming language of the user's choice. Even though the DLL can be used in code from several programming languages it has been tested only from Delphi.

An example implementation of how to use the DLL from Delphi can be viewed in section 6.3.

It should be noted that the C compiler will rename the functions in *grt_shell.c*. This means that the function name *func()* will be changed to *_func()* in the DLL.

6 Source code

This section contains the source for the files mentioned previously.

6.1 grt_shell.c

This is the wrapper file creating a user friendly user interface.

6.1.1 Comments on the interface

The interface of *grt_shell.c* contains three functions:

```
     char* initiateController();
     char* getControllerOutput(int nbrInputArgs, double*
                inputArgs, int nbrOutputArgs,
                double* outputArgs);
     char* performCleanup();
```

Each of the functions returns pointers to a character array. They all return an empty pointer if no error occurred during the function call. If an error has occurred, the pointer will point to a character array containing a description of the error.

Function 1 performs initiation of the controller. This function must be called before any other function is called.

Function 2 performs the controller calculations. This is the function to call once every sample. Both inputs and outputs are supplied by the caller in the form of pointers to arrays.

Function 3 performs cleanup. This function should be called before closing the application.

6.1.2 The code

```
/*
 * Modified version of grt_main.c
 * Modifications made by Roland Pfeiffer
 */

/* $Revision: 1.68.4.6 $
 * Copyright 1994-2004 The MathWorks, Inc.
 *
 * File    : grt_main.c
 *
 * Abstract:
 *      A Generic "Real-Time (single tasking or pseudo-multitasking,
 *      statically allocated data)" main that runs under most
 *      operating systems.
 *
 *      This file may be a useful starting point when targeting a new
 *      processor or microcontroller.
 *
 *
 * Compiler specified defines:
 *    RT              - Required.
 *      MODEL=modelname - Required.
 *    NUMST=#         - Required. Number of sample times.
 *    NCSTATES=#      - Required. Number of continuous states.
 *      TID01EQ=1 or 0  - Optional. Only define to 1 if sample time
task
 *                        id's 0 and 1 have equal rates.
 *      MULTITASKING    - Optional. (use MT for a synonym).
 *    SAVEFILE        - Optional (non-quoted) name of .mat file to
create.
 *                  Default is <MODEL>.mat
 *      BORLAND         - Required if using Borland C/C++
 */

#include <float.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

#include "tmwtypes.h"
# include "rtmodel.h"
#include "rt_sim.h"
#include "rt_logging.h"
#ifdef UseMMIDataLogging
#include "rt_logging_mmi.h"
#endif
#include "rt_nonfinite.h"

/* Signal Handler header */
#ifdef BORLAND
#include <signal.h>
#include <float.h>
#endif

#include "ext_work.h"

/*=========*
 * Defines *
 *=========*/
#ifndef TRUE

#define FALSE (0)
#define TRUE  (1)
#endif

#ifndef EXIT_FAILURE
#define EXIT_FAILURE  1
#endif
#ifndef EXIT_SUCCESS
#define EXIT_SUCCESS  0
```

```
#endif

#define QUOTE1(name) #name
#define QUOTE(name) QUOTE1(name)    /* need to expand name    */


#ifndef RT
# error "must define RT"
#endif

#ifndef MODEL
# error "must define MODEL"
#endif

#ifndef NUMST
# error "must define number of sample times, NUMST"
#endif

#ifndef NCSTATES
# error "must define NCSTATES"
#endif

#ifndef SAVEFILE
# define MATFILE2(file) #file ".mat"
# define MATFILE1(file) MATFILE2(file)
# define MATFILE MATFILE1(MODEL)
#else
# define MATFILE QUOTE(SAVEFILE)
#endif

#define RUN_FOREVER -1.0

#define EXPAND_CONCAT(name1,name2) name1 ## name2
#define CONCAT(name1,name2) EXPAND_CONCAT(name1,name2)
#define RT_MODEL            CONCAT(MODEL,_rtModel)

/*====================*
 * External functions *
 *====================*/
extern RT_MODEL *MODEL(void);

extern void MdlInitializeSizes(void);
extern void MdlInitializeSampleTimes(void);
extern void MdlStart(void);
extern void MdlOutputs(int_T tid);
extern void MdlUpdate(int_T tid);
extern void MdlTerminate(void);

#if NCSTATES > 0
  extern void rt_ODECreateIntegrationData(RTWSolverInfo *si);
  extern void rt_ODEUpdateContinuousStates(RTWSolverInfo *si);

# define rt_CreateIntegrationData(S) \
    rt_ODECreateIntegrationData(rtmGetRTWSolverInfo(S));
# define rt_UpdateContinuousStates(S) \
    rt_ODEUpdateContinuousStates(rtmGetRTWSolverInfo(S));
# else
# define rt_CreateIntegrationData(S)  \
      rtsiSetSolverName(rtmGetRTWSolverInfo(S),"FixedStepDiscrete");
# define rt_UpdateContinuousStates(S) \
      rtmSetT(S, rtsiGetSolverStopTime(rtmGetRTWSolverInfo(S)));
#endif

/*==================================*
 * Global data local to this module *
 *==================================*/
static struct {
  int_T    stopExecutionFlag;
  int_T    isrOverrun;
  int_T    overrunFlags[NUMST];
  const char_T *errmsg;
} GBLbuf;

static char errorMsg[200];

#ifdef EXT_MODE
#  define rtExtModeSingleTaskUpload(S)                              \
   {                                                                \
        int stIdx;                                                  \
        rtExtModeUploadCheckTrigger(rtmGetNumSampleTimes(S));       \
        for (stIdx=0; stIdx<NUMST; stIdx++) {                       \
            if (rtmIsSampleHit(S, stIdx, 0 /*unused*/)) {           \
                rtExtModeUpload(stIdx,rtmGetTaskTime(S,stIdx));     \
            }                                                       \
        }                                                           \
   }
#else
#  define rtExtModeSingleTaskUpload(S) /* Do nothing */
#endif

/*=================*
 * Local functions *
 *=================*/
#ifdef BORLAND
/* Implemented for BC++ only*/

typedef void (*fptr)(int, int);

/* Function: divideByZero
=====================================================
 *
 * Abstract: Traps the error Division by zero and prints a warning
 *           Also catches other FP errors, but does not identify them
 *           specifically.
 */
void divideByZero(int sigName, int sigType)
{
  signal(SIGFPE, (fptr)divideByZero);
  if ((sigType == FPE_ZERODIVIDE)||(sigType == FPE_INTDIV0)){
      sprintf(errorMsg, "Warning: Division by zero");
    return;
  }
  else{
      sprintf(errorMsg, "Warning: Floating Point error");
    return;
  }
} /* end divideByZero */

#endif /* BORLAND */

/*
 * Calculate the controller output. Supply input arguments:
 * inputArgs[0] - IMEPn_ref
 * inputArgs[1] - ca50_ref
 * inputArgs[2] - IMEPn
 * inputArgs[3] - PCP
 * inputArgs[4] - PCPD
 * inputArgs[5] - ca50
 * inputArgs[6] - TIn
 *
 * Also supply a pointer to an array capable of containing the
following values:
 * outputArgs[0] - fuelA_ratio
 * outputArgs[1] - fuelMEP
 */
static RT_MODEL *S;
char* getControllerOutput(int nbrInputArgs, double* inputArgs,
                           int nbrOutputArgs, double* outputArgs) {
  real_T tnext;
  if (nbrInputArgs != 7) {
    sprintf(errorMsg,"nbrInputArgs should be 7 but was %d",
nbrInputArgs);
    return errorMsg;
  }
  if (nbrOutputArgs != 2) {
    sprintf(errorMsg, "nbrOutputArgs should be 2 but was %d",
nbrOutputArgs);
    return errorMsg;
  }
  controller_U.IMEPn_ref = inputArgs[0];
  controller_U.ca50_ref = inputArgs[1];
  controller_U.IMEPn = inputArgs[2];
  controller_U.PCP = inputArgs[3];
  controller_U.PCPD = inputArgs[4];
  controller_U.ca50 = inputArgs[5];
  controller_U.TIn = inputArgs[6];

  /***********************************************
   * Check and see if base step time is too fast *
   ***********************************************/

  if (GBLbuf.isrOverrun++) {
    GBLbuf.stopExecutionFlag = 1;
    sprintf(errorMsg, "GBLbuf.isrOverrun");
    return errorMsg;
  }

  /***********************************************
   * Check and see if error status has been set  *
   ***********************************************/
  if (rtmGetErrorStatus(S) != NULL) {
    GBLbuf.stopExecutionFlag = 1;
    sprintf(errorMsg, "rtmGetErrorStatus(S) != null");
    return errorMsg;
  }

  /* enable interrupts here */

  /*
   * In a multi-tasking environment, this would be removed from the
base rate
   * and called as a "background" task.
   */
  rtExtModeOneStep(rtmGetRTWExtModeInfo(S),
               rtmGetNumSampleTimes(S),
               (boolean_T *)&rtmGetStopRequested(S));

  tnext = rt_SimGetNextSampleHit();
  rtsiSetSolverStopTime(rtmGetRTWSolverInfo(S),tnext);

  MdlOutputs(0);

  rtExtModeSingleTaskUpload(S);

  GBLbuf.errmsg = rt_UpdateTXYLogVars(rtmGetRTWLogInfo(S),
                              rtmGetTPtr(S));
  if (GBLbuf.errmsg != NULL) {
    GBLbuf.stopExecutionFlag = 1;
    sprintf(errorMsg, "GBLbuf.errmsg != null");
    return errorMsg;
  }
  MdlUpdate(0);
  rt_SimUpdateDiscreteTaskSampleHits(rtmGetNumSampleTimes(S),
                             rtmGetTimingData(S),
                             rtmGetSampleHitPtr(S),
                             rtmGetTPtr(S));

  if (rtmGetSampleTime(S,0) == CONTINUOUS_SAMPLE_TIME) {
    rt_UpdateContinuousStates(S);
  }

  GBLbuf.isrOverrun--;

  rtExtModeCheckEndTrigger();
  outputArgs[0] = controller_Y.fuelA_ratio;
  outputArgs[1] = controller_Y.fuelMEP;
  return errorMsg;
}

/* Function: initiateController
=============================================================
 *
 * Abstract:
 *      Execute model on a generic target such as a workstation.
 */
char* initiateController() {
  const char *status;
  /*****************************
   * MathError Handling for BC++ *
   *****************************/
#ifdef BORLAND
  signal(SIGFPE, (fptr)divideByZero);
#endif
  {
    /****************************
     * Initialize global memory *
     ****************************/
    rtExtModeParseArgs(argc, argv, NULL);
    (void)memset(&GBLbuf, 0, sizeof(GBLbuf));

    /************************
     * Initialize the model *
     ************************/
    rt_InitInfAndNaN(sizeof(real_T));

    S = MODEL();
    if (rtmGetErrorStatus(S) != NULL) {
        (void)fprintf(stderr,"Error during model registration: %s\n",
                      rtmGetErrorStatus(S));
        exit(EXIT_FAILURE);
        sprintf(errorMsg, "Error during model registration: %s",
rtmGetErrorStatus(S));
        return(errorMsg);
    }
    rtmSetTFinal(S,RUN_FOREVER);

    MdlInitializeSizes();
    MdlInitializeSampleTimes();

    status = rt_SimInitTimingEngine(rtmGetNumSampleTimes(S),
                                    rtmGetStepSize(S),
                                    rtmGetSampleTimePtr(S),
                                    rtmGetOffsetTimePtr(S),
                                    rtmGetSampleHitPtr(S),
                                    rtmGetSampleTimeTaskIDPtr(S),
                                    rtmGetTStart(S),
                                    &rtmGetSimTimeStep(S),
                                    &rtmGetTimingData(S));

    if (status != NULL) {
      (void)fprintf(stderr,
                "Failed to initialize sample time engine: %s\n",
status);
      exit(EXIT_FAILURE);
      sprintf(errorMsg, "Failed to initialize sample time engine: %s",
status);
      return errorMsg;
    }
    rt_CreateIntegrationData(S);
    GBLbuf.errmsg = rt_StartDataLogging(rtmGetRTWLogInfo(S),
                                        rtmGetTFinal(S),
                                        rtmGetStepSize(S),
                                        &rtmGetErrorStatus(S));
    rtExtModeCheckInit(rtmGetNumSampleTimes(S));
    rtExtModeWaitForStartPkt(rtmGetRTWExtModeInfo(S),
                             rtmGetNumSampleTimes(S),
                             (boolean_T *)&rtmGetStopRequested(S));

    MdlStart();
    if (rtmGetErrorStatus(S) != NULL) {
      GBLbuf.stopExecutionFlag = 1;
    }
    errorMsg[0] = 0;
    return errorMsg;
    /*
      Initialization should be finished at this point
    */
  }
}

char* performCleanup() {
#ifdef UseMMIDataLogging
  rt_CleanUpForStateLogWithMMI(rtmGetRTWLogInfo(S));
#endif
  rt_StopDataLogging(MATFILE,rtmGetRTWLogInfo(S));

  rtExtModeShutdown(rtmGetNumSampleTimes(S));

  if (GBLbuf.errmsg) {
    (void)fprintf(stderr,"%s\n",GBLbuf.errmsg);
    exit(EXIT_FAILURE);
    sprintf(errorMsg, "%s", GBLbuf.errmsg);
    return errorMsg;
  }

  if (GBLbuf.isrOverrun) {
    (void)fprintf(stderr,
              "%s: ISR overrun - base sampling rate is too fast\n",
              QUOTE(MODEL));
    exit(EXIT_FAILURE);
    sprintf(errorMsg, "%s: ISR overrun - base sampling rate is too
fast",QUOTE(MODEL));
    return errorMsg;
  }

  if (rtmGetErrorStatus(S) != NULL) {
    (void)fprintf(stderr,"%s\n", rtmGetErrorStatus(S));
    exit(EXIT_FAILURE);
    sprintf(errorMsg,"%s", rtmGetErrorStatus(S));
    return errorMsg;
  }
  return errorMsg;
}

BOOL WINAPI __declspec(dllexport) LibMain(HINSTANCE hDLLInst,DWORD
fdwReason,LPVOID lpvReserved)
{
      return 1;
}
```

6.2 controller.def

This file contains the names of the functions to be exported from the DLL.

```
EXPORTS
      initiateController
      getControllerOutput
      performCleanup
```

6.3 Controller.pas

This is the Delphi wrapper file for the DLL created in this tutorial.

6.3.1 Comments on the implementation

In the declaration of the external functions there are two details to pay attention to. The first thing is the location of the DLL file. In the example implementation it is assumed that the DLL is available in the search path of the Delphi project. If this is not the case, the name of the DLL must contain the complete path.

Another important thing is that the external functions have to be declared as `cdecl`. This declaration causes the order of the parameters to be rearranged in such a way that the application will not crash.

Displayed in this implementation is an example of how to make use of the returned error status from the DLL.

6.3.2 The code

```
unit Controller;

interface
   type
      TController = class
      private
         nbrInputParams: Integer;
         nbrOutputParams: Integer;
         inputs: PDouble;
         outputs: PDouble;
         errorMsg: PChar;
      public
         constructor Create(nbrInputs: Integer; inputPointer: PDouble;
                      nbrOutputs: Integer; outputPointer: PDouble);
         procedure initiateController();
         procedure getControllerOutput();
         procedure performCleanup();
   end;
   function _initiateController(): PChar; cdecl external
      'controller.dll'
   function _getControllerOutput(nbrInputArgs: Integer;
      inputArgs: PDouble; nbrOutputArgs: Integer; outputArgs: PDouble):
      PChar; cdecl external 'controller.dll'
   function _performCleanup(): PChar; cdecl external
      'controller.dll'
implementation
   uses
       SysUtils;
   constructor TController.Create(nbrInputs: Integer;
      inputPointer: PDouble; nbrOutputs: Integer;
      outputPointer: PDouble);
   begin
      nbrInputParams := nbrInputs;
      nbrOutputParams := nbrOutputs;
      inputs := inputPointer;
      outputs := outputPointer;
   end;

   procedure TController.initiateController();
   begin
      errorMsg := _initiateController();
      if Length(errorMsg) <> 0 then
         raise Exception.Create(errorMsg);
   end;

   procedure TController.getControllerOutput();
   begin
      errorMsg := _getControllerOutput(nbrInputParams, inputs,
                                       nbrOutputParams, outputs);
      if Length(errorMsg) <> 0 then
         raise Exception.Create(errorMsg);
   end;

   procedure TController.performCleanup();
   begin
      errorMsg := _performCleanup();
      if Length(errorMsg) <> 0 then
         raise Exception.Create(errorMsg);
   end;
end.
```

7 Contact information

Comments or bug reports are much appreciated. Please send them to Roland.Pfeiffer@spray.se.