// myo\_sfun.cpp

//

#define S\_FUNCTION\_NAME myo\_sfun

#define S\_FUNCTION\_LEVEL 2

#include "simstruc.h"

#include "myo\_sfun.h" // macros and defines for general behavior

#include "myo\_sfun\_wiring.h" // macros and defines for sfcn block and data

#include "myo/myo.hpp" // Myo SDK bindings

#include "myo\_class.hpp" // application class for Myo implementation

#include <windows.h> // win api for threading support

#include <process.h> // process/thread support

// ==================================================

// Global variables

unsigned int threadID;

HANDLE ghThread;

HANDLE ghMutex;

volatile bool gRunThreadFlag = false;

real\_T gCountMyosRequired = 1;

real\_T gEmgEnabledRequired = 1;

// ==================================================

// Thread function

// This thread calls myo::Hub::runOnce() for the

// lifetime of the application

unsigned \_\_stdcall runThreadFunc( void\* S\_ ) {

SimStruct \*S = (SimStruct \*)S\_;

myo::Hub\* pHub = (myo::Hub \*) ssGetPWork(S)[IDX\_HUB];

while ( gRunThreadFlag ) { // unset isStreaming to terminate thread

// acquire lock then write data into queue

DWORD dwWaitResult;

dwWaitResult = WaitForSingleObject(ghMutex,INFINITE);

switch (dwWaitResult)

{

case WAIT\_OBJECT\_0: // The thread got ownership of the mutex

// --- CRITICAL SECTION - holding lock

pHub->runOnce(STREAMING\_TIMEOUT); // run callbacks to collector

// END CRITICAL SECTION - release lock

if (! ReleaseMutex(ghMutex)) { return FALSE; } // acquired bad mutex

break;

case WAIT\_ABANDONED:

return FALSE; // acquired bad mutex

}

} // end thread and return

\_endthreadex(0); //

return 0;

}

// ==================================================

// Utility functions

// User defined functions for convenience

static void setOutputDimensionInfo(SimStruct \*S, int\_T port, int\_T len, int\_T sz)

{

DECL\_AND\_INIT\_DIMSINFO(di);

int\_T dims[2];

di.numDims = 2;

dims[0] = sz;

dims[1] = len;

di.dims = dims;

di.width = sz\*len;

ssSetOutputPortDimensionInfo(S, port, &di);

}

// ==================================================

// Model callback functions

#define MDL\_CHECK\_PARAMETERS /\* Change to #undef to remove function \*/

#if defined(MDL\_CHECK\_PARAMETERS) && defined(MATLAB\_MEX\_FILE)

static void mdlCheckParameters(SimStruct \*S)

{

DB\_MYO\_SFUN("ENTER mdlCheckParameters()\n",);

// check data types

const mxArray \*pEmgEnabled = ssGetSFcnParam(S,IDX\_EMG\_ENABLED\_REQUIRED);

const mxArray \*pCountMyos = ssGetSFcnParam(S,IDX\_COUNT\_MYOS\_REQUIRED);

if ( !IS\_PARAM\_SCALAR\_DOUBLE(pEmgEnabled)) {

ssSetErrorStatus(S,"EMG Enabled parameter must be a scalar int8");

return;

}

if ( !IS\_PARAM\_SCALAR\_DOUBLE(pCountMyos)) {

ssSetErrorStatus(S,"Count Myos parameter must be a scalar int8");

return;

}

DB\_MYO\_SFUN("Parameter datatypes OK\n");

// check values

real\_T emgEnabled = \*mxGetPr(pEmgEnabled);

real\_T countMyos = \*mxGetPr(pCountMyos);

if ( !((emgEnabled==0.0)||(emgEnabled==1.0)) ) {

ssSetErrorStatus(S,"EMG Enabled parameter must be 0 or 1");

return;

}

if ( !((countMyos==1.0)||(countMyos==2.0)) ) {

ssSetErrorStatus(S,"Count Myos parameter must be 1 or 2");

return;

}

DB\_MYO\_SFUN("Parameter values OK\n");

DB\_MYO\_SFUN("EXIT mdlCheckParameters()\n",);

}

#endif /\* MDL\_CHECK\_PARAMETERS \*/

static void mdlInitializeSizes(SimStruct \*S)

{

DB\_MYO\_SFUN("ENTER mdlInitializeSizes()\n",);

int\_T numOutputPorts;

ssSetNumContStates( S, 0);

ssSetNumDiscStates( S, 0);

ssSetNumSFcnParams(S,NUM\_SFCN\_PARAMS);

ssSetSFcnParamTunable(S,IDX\_EMG\_ENABLED\_REQUIRED,SS\_PRM\_NOT\_TUNABLE);

ssSetSFcnParamTunable(S,IDX\_COUNT\_MYOS\_REQUIRED,SS\_PRM\_NOT\_TUNABLE);

DB\_MYO\_SFUN("Checking parameters ...\n");

#if defined(MATLAB\_MEX\_FILE)

if(ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S) ) {

mdlCheckParameters(S);

if(ssGetErrorStatus(S) != NULL) return;

} else {

return; /\* The Simulink engine reports a mismatch error. \*/

}

#endif

// get parameters from block

gEmgEnabledRequired = \*mxGetPr(ssGetSFcnParam(S,IDX\_EMG\_ENABLED\_REQUIRED));

gCountMyosRequired = \*mxGetPr(ssGetSFcnParam(S,IDX\_COUNT\_MYOS\_REQUIRED));

DB\_MYO\_SFUN("Parameter values:\n");

DB\_MYO\_SFUN("\tgEmgEnabledRequired = %f\n",gEmgEnabledRequired);

DB\_MYO\_SFUN("\tgCountMyosRequired = %f\n",gCountMyosRequired);

// Determine number of output ports based on parameters

// This is the process parameters routine to calculate numOutputPorts

// gCountMyosRequired

// | gEmgEnabledRequired

// | | numOutputPorts

// 1 0 6 Default - IMU for Myo 1

// 1 1 7 Adds EMG for Myo 1

// 2 0 12 Adds IMU for Myo 2

// 2 1 ERROR

if ((gEmgEnabledRequired==0.0)&&(gCountMyosRequired==1.0)) {

numOutputPorts = NUM\_OUTPUT\_PORTS\_IMU;

} else if ((gEmgEnabledRequired==1.0)&&(gCountMyosRequired==1.0)) {

numOutputPorts = NUM\_OUTPUT\_PORTS\_IMU+NUM\_OUTPUT\_PORTS\_EMG;

} else if ((gEmgEnabledRequired==0.0)&&(gCountMyosRequired==2.0)) {

numOutputPorts = 2\*NUM\_OUTPUT\_PORTS\_IMU;

} else if ((gEmgEnabledRequired==1.0)&&(gCountMyosRequired==2.0)) {

ssSetErrorStatus(S,"EMG Cannot be enabled with more than one Myo.");

return;

}

DB\_MYO\_SFUN("Computed number of output ports:\n");

DB\_MYO\_SFUN("\tnumOutputPorts = %d\n",numOutputPorts);

if (!ssSetNumOutputPorts(S, numOutputPorts)) return;

// MYO 1 IMU - These ports are always hooked up

DB\_MYO\_SFUN("Configuring ports for Myo 1 IMU ...\n");

setOutputDimensionInfo(S,OUTPUT\_PORT\_IDX\_QUAT, LEN\_QUAT ,SZ\_IMU);

setOutputDimensionInfo(S,OUTPUT\_PORT\_IDX\_GYRO,LEN\_GYRO ,SZ\_IMU);

setOutputDimensionInfo(S,OUTPUT\_PORT\_IDX\_ACCEL,LEN\_ACCEL,SZ\_IMU);

setOutputDimensionInfo(S,OUTPUT\_PORT\_IDX\_POSE,LEN\_POSE ,SZ\_IMU);

setOutputDimensionInfo(S,OUTPUT\_PORT\_IDX\_ARM,LEN\_ARM ,SZ\_IMU);

setOutputDimensionInfo(S,OUTPUT\_PORT\_IDX\_XDIR,LEN\_XDIR ,SZ\_IMU);

if (gEmgEnabledRequired==1.0) {

DB\_MYO\_SFUN("Configuring ports for Myo 1 EMG ...\n");

// Add EMG port for Myo 1

setOutputDimensionInfo(S,OUTPUT\_PORT\_IDX\_EMG,LEN\_EMG,SZ\_EMG);

} else if (gCountMyosRequired==2.0) {

DB\_MYO\_SFUN("Configuring ports for Myo 2 IMU ...\n");

// Add IMU ports for Myo 2

setOutputDimensionInfo(S,NUM\_OUTPUT\_PORTS\_IMU+OUTPUT\_PORT\_IDX\_QUAT,LEN\_QUAT ,SZ\_IMU);

setOutputDimensionInfo(S,NUM\_OUTPUT\_PORTS\_IMU+OUTPUT\_PORT\_IDX\_GYRO,LEN\_GYRO ,SZ\_IMU);

setOutputDimensionInfo(S,NUM\_OUTPUT\_PORTS\_IMU+OUTPUT\_PORT\_IDX\_ACCEL,LEN\_ACCEL,SZ\_IMU);

setOutputDimensionInfo(S,NUM\_OUTPUT\_PORTS\_IMU+OUTPUT\_PORT\_IDX\_POSE,LEN\_POSE ,SZ\_IMU);

setOutputDimensionInfo(S,NUM\_OUTPUT\_PORTS\_IMU+OUTPUT\_PORT\_IDX\_ARM,LEN\_ARM ,SZ\_IMU);

setOutputDimensionInfo(S,NUM\_OUTPUT\_PORTS\_IMU+OUTPUT\_PORT\_IDX\_XDIR,LEN\_XDIR ,SZ\_IMU);

}

ssSetNumSampleTimes( S, 1);

ssSetNumRWork( S, 0); /\* number of real work vector elements \*/

ssSetNumIWork( S, NUM\_IWORK); /\* number of integer work vector elements\*/

ssSetNumPWork( S, NUM\_PWORK); /\* number of pointer work vector elements\*/

ssSetNumModes( S, 0); /\* number of mode work vector elements \*/

ssSetNumNonsampledZCs( S, 0); /\* number of nonsampled zero crossings \*/

ssSetSimStateCompliance(S, USE\_CUSTOM\_SIM\_STATE);

ssSetOptions( S, 0); /\* general options (SS\_OPTION\_xx) \*/

DB\_MYO\_SFUN("EXIT mdlInitializeSizes\n");

} /\* end mdlInitializeSizes \*/

static void mdlInitializeSampleTimes(SimStruct \*S)

{

DB\_MYO\_SFUN("ENTER mdlInitializeSampleTimes\n");

ssSetSampleTime(S, 0, 0.001\*SAMPLE\_TIME\_BLK);

ssSetOffsetTime(S, 0, 0.0);

DB\_MYO\_SFUN("EXIT mdlInitializeSampleTimes()\n");

} /\* end mdlInitializeSampleTimes \*/

#define MDL\_START /\* Change to #undef to remove function \*/

#if defined(MDL\_START)

static void mdlStart(SimStruct \*S)

{

DB\_MYO\_SFUN("ENTER mdlStart()\n");

// declare pointers

DataCollector\* pCollector = NULL;

myo::Hub\* pHub = NULL;

myo::Myo\* pMyo = NULL;

DB\_MYO\_SFUN("Setting up DataCollector ...\n");

ssGetPWork(S)[IDX\_COLLECTOR] = (void \*) new DataCollector;

pCollector = (DataCollector \*)ssGetPWork(S)[IDX\_COLLECTOR];

if (gEmgEnabledRequired==1.0)

pCollector->addEmgEnabled = true; // lets collector handle data events

DB\_MYO\_SFUN("Setting up Hub ...\n");

// Instantiate a Hub and get a Myo

ssGetPWork(S)[IDX\_HUB] = (void \*) new myo::Hub("com.mark-toma.myo\_sfun");

pHub = (myo::Hub \*)ssGetPWork(S)[IDX\_HUB];

if ( !pHub ) {

ssSetErrorStatus(S,"Hub failed to init!");

return;

}

pHub->setLockingPolicy(myo::Hub::lockingPolicyNone); // TODO: What does this do?

pHub->addListener(pCollector);

DB\_MYO\_SFUN("Setting up Myo ...\n");

pMyo = pHub->waitForMyo(5);

if ( !pMyo ) {

ssSetErrorStatus(S,"Myo failed to init!");

return;

}

DB\_MYO\_SFUN("Setting up mutex lock ...\n");

// instantiate mutex

ghMutex = CreateMutex(NULL,FALSE,NULL);

if (ghMutex == NULL) {

ssSetErrorStatus(S,"Failed to set up mutex.\n");

return;

}

DB\_MYO\_SFUN("Running Hub for INIT\_DELAY to validate countMyos ...\n");

// Let Hub run callbacks on collector so we can figure out how many

// Myos are connected to Myo Connect so we can assert gCountMyosRequired

pHub->run(INIT\_DELAY);

if (gCountMyosRequired != pCollector->getCountMyos()) {

ssSetErrorStatus(S,"myo\_sfun failed to initialize with countMyos.\n");

return;

}

DB\_MYO\_SFUN("Running Hub for BUFFER\_DELAY to preload buffers ...\n");

// Flush the data queues with syncDataSources

pCollector->syncDataSources();

// Enabled data and initialize buffer

pCollector->addDataEnabled = true;

pHub->run(BUFFER\_DELAY);

DB\_MYO\_SFUN("Dispatching thread to run Hub ...\n");

// dispatch concurrent task

gRunThreadFlag = true;

ghThread = (HANDLE)\_beginthreadex( NULL, 0, &runThreadFunc, S, 0, &threadID );

if ( !ghThread ) {

ssSetErrorStatus(S,"Failed to create streaming thread!\n");

return;

}

DB\_MYO\_SFUN("EXIT mdlStart()\n");

}

#endif /\* MDL\_START \*/

static void mdlOutputs(SimStruct \*S, int\_T tid)

{

DB\_MYO\_SFUN("ENTER mdlOutputs()\n");

// get pHub and pCollector

myo::Hub\* pHub = (myo::Hub \*) ssGetPWork(S)[IDX\_HUB];

DataCollector\* pCollector = (DataCollector \*) ssGetPWork(S)[IDX\_COLLECTOR];

int\_T iter = ssGetIWork(S)[IDX\_ITER]; // get iteration counter

int\_T ii, jj;

real\_T \*pQuat1, \*pGyro1, \*pAccel1, \*pPose1, \*pArm1, \*pXDir1;

real\_T \*pQuat2, \*pGyro2, \*pAccel2, \*pPose2, \*pArm2, \*pXDir2;

real\_T \*pEMG1;

FrameIMU frameIMU1[SAMPLES\_PER\_FRAME\_IMU];

FrameIMU frameIMU2[SAMPLES\_PER\_FRAME\_IMU];

FrameEMG frameEMG1[SAMPLES\_PER\_FRAME\_EMG];

int\_T countIMU1 = 0;

int\_T countIMU2 = 0;

int\_T countEMG1 = 0;

countIMU1 = pCollector->getCountIMU(1);

if (gCountMyosRequired==2)

countIMU2 = pCollector->getCountIMU(1);

else if ((gCountMyosRequired==1)&&(gEmgEnabledRequired==1))

countEMG1 = pCollector->getCountEMG(1);

// fail if the queue is falling behind

if (countIMU1 < 1+SAMPLES\_PER\_FRAME\_IMU\*BUFFER\_FRAMES\_MIN) {

ssSetErrorStatus(S,"IMU1 buffer is less than minimum size.");

return;

}

if ( (gEmgEnabledRequired==1) &&

(countEMG1 < 1+SAMPLES\_PER\_FRAME\_EMG\*BUFFER\_FRAMES\_MIN) ) {

ssPrintf("EMG Buffer is %d samples.\n",countEMG1);

ssSetErrorStatus(S,"EMG buffer is less than minimum size");

return;

}

if ( (gCountMyosRequired==2) &&

(countIMU2 < 1+SAMPLES\_PER\_FRAME\_IMU\*BUFFER\_FRAMES\_MIN) ) {

ssSetErrorStatus(S,"IMU2 buffer is less than minimum size.");

return;

}

// Verify that collector still has all of its Myos, otherwise error out

if ( pCollector->getCountMyos() != gCountMyosRequired ) {

ssSetErrorStatus(S,"myo\_sfun countMyos is inconsistent with initialization... We lost a Myo!");

return;

}

// move above into another method

DB\_MYO\_SFUN\_ITER("IMU1=%d\tEMG1=%d\n",countIMU1,countEMG1);

DWORD dwWaitResult = WaitForSingleObject(ghMutex,INFINITE);

switch (dwWaitResult)

{

case WAIT\_OBJECT\_0: // The thread got ownership of the mutex

// --- CRITICAL SECTION - holding lock

// in initial run set the buffer lengths

if (iter==0) {

DB\_MYO\_SFUN("Initializing data buffers on iteration zero ...\n");

while(pCollector->getCountIMU(1)>1+SAMPLES\_PER\_FRAME\_IMU\*BUFFER\_FRAMES\_DES )

\*frameIMU1 = pCollector->getFrameIMU(1);

if(gEmgEnabledRequired) {

while(pCollector->getCountEMG(1)>1+SAMPLES\_PER\_FRAME\_EMG\*BUFFER\_FRAMES\_DES )

\*frameEMG1 = pCollector->getFrameEMG(1);

}

if (gCountMyosRequired==2) {

while(pCollector->getCountIMU(2)>1+SAMPLES\_PER\_FRAME\_IMU\*BUFFER\_FRAMES\_DES)

\*frameIMU2 = pCollector->getFrameIMU(2);

}

}

for (ii=0;ii<SAMPLES\_PER\_FRAME\_IMU;ii++) {

frameIMU1[ii] = pCollector->getFrameIMU(1);

}

if ( gEmgEnabledRequired ) {

for (ii=0;ii<SAMPLES\_PER\_FRAME\_EMG;ii++) {

frameEMG1[ii] = pCollector->getFrameEMG(1);

}

}

if (gCountMyosRequired==2) {

for (ii=0;ii<SAMPLES\_PER\_FRAME\_IMU;ii++) {

frameIMU2[ii] = pCollector->getFrameIMU(2);

}

}

// END CRITICAL SECTION - release lock

if ( !ReleaseMutex(ghMutex)) {

ssSetErrorStatus(S,"Failed to release lock\n");

return;

}

break;

case WAIT\_ABANDONED:

ssSetErrorStatus(S,"Acquired abandoned lock\n");

return;

break;

}

pQuat1 = ssGetOutputPortRealSignal(S,OUTPUT\_PORT\_IDX\_QUAT);

pQuat1[0] = frameIMU1[0].quat.w();

pQuat1[1] = frameIMU1[1].quat.w();

pQuat1[2] = frameIMU1[0].quat.x();

pQuat1[3] = frameIMU1[1].quat.x();

pQuat1[4] = frameIMU1[0].quat.y();

pQuat1[5] = frameIMU1[1].quat.y();

pQuat1[6] = frameIMU1[0].quat.z();

pQuat1[7] = frameIMU1[1].quat.z();

pGyro1 = ssGetOutputPortRealSignal(S,OUTPUT\_PORT\_IDX\_GYRO);

pGyro1[0] = frameIMU1[0].gyro.x();

pGyro1[1] = frameIMU1[1].gyro.x();

pGyro1[2] = frameIMU1[0].gyro.y();

pGyro1[3] = frameIMU1[1].gyro.y();

pGyro1[4] = frameIMU1[0].gyro.z();

pGyro1[5] = frameIMU1[1].gyro.z();

pAccel1 = ssGetOutputPortRealSignal(S,OUTPUT\_PORT\_IDX\_ACCEL);

pAccel1[0] = frameIMU1[0].accel.x();

pAccel1[1] = frameIMU1[1].accel.x();

pAccel1[2] = frameIMU1[0].accel.y();

pAccel1[3] = frameIMU1[1].accel.y();

pAccel1[4] = frameIMU1[0].accel.z();

pAccel1[5] = frameIMU1[1].accel.z();

pPose1 = ssGetOutputPortRealSignal(S,OUTPUT\_PORT\_IDX\_POSE);

pPose1[0] = frameIMU1[0].pose.type();

pPose1[1] = frameIMU1[1].pose.type();

pArm1 = ssGetOutputPortRealSignal(S,OUTPUT\_PORT\_IDX\_ARM);

pArm1[0] = frameIMU1[0].arm;

pArm1[1] = frameIMU1[1].arm;

pXDir1 = ssGetOutputPortRealSignal(S,OUTPUT\_PORT\_IDX\_XDIR);

pXDir1[0] = frameIMU1[0].xDir;

pXDir1[1] = frameIMU1[1].xDir;

if ( gEmgEnabledRequired ) {

pEMG1 = ssGetOutputPortRealSignal(S,OUTPUT\_PORT\_IDX\_EMG);

for (ii=0;ii<8;ii++) {

for (jj=0;jj<8;jj++) {

//DB\_MYO\_SFUN\_ITER("ii=%d\tjj=%d\tkk=%d\n",ii,jj,ii+8\*jj);

pEMG1[ii+8\*jj] = frameEMG1[ii].emg[jj];

}

}

}

if (gCountMyosRequired==2) {

pQuat2 = ssGetOutputPortRealSignal(S,NUM\_OUTPUT\_PORTS\_IMU+OUTPUT\_PORT\_IDX\_QUAT);

pQuat2[0] = frameIMU2[0].quat.w();

pQuat2[1] = frameIMU2[1].quat.w();

pQuat2[2] = frameIMU2[0].quat.x();

pQuat2[3] = frameIMU2[1].quat.x();

pQuat2[4] = frameIMU2[0].quat.y();

pQuat2[5] = frameIMU2[1].quat.y();

pQuat2[6] = frameIMU2[0].quat.z();

pQuat2[7] = frameIMU2[1].quat.z();

pGyro2 = ssGetOutputPortRealSignal(S,NUM\_OUTPUT\_PORTS\_IMU+OUTPUT\_PORT\_IDX\_GYRO);

pGyro2[0] = frameIMU2[0].gyro.x();

pGyro2[1] = frameIMU2[1].gyro.x();

pGyro2[2] = frameIMU2[0].gyro.y();

pGyro2[3] = frameIMU2[1].gyro.y();

pGyro2[4] = frameIMU2[0].gyro.z();

pGyro2[5] = frameIMU2[1].gyro.z();

pAccel2 = ssGetOutputPortRealSignal(S,NUM\_OUTPUT\_PORTS\_IMU+OUTPUT\_PORT\_IDX\_ACCEL);

pAccel2[0] = frameIMU2[0].accel.x();

pAccel2[1] = frameIMU2[1].accel.x();

pAccel2[2] = frameIMU2[0].accel.y();

pAccel2[3] = frameIMU2[1].accel.y();

pAccel2[4] = frameIMU2[0].accel.z();

pAccel2[5] = frameIMU2[1].accel.z();

pPose2 = ssGetOutputPortRealSignal(S,NUM\_OUTPUT\_PORTS\_IMU+OUTPUT\_PORT\_IDX\_POSE);

pPose2[0] = frameIMU2[0].pose.type();

pPose2[1] = frameIMU2[1].pose.type();

pArm2 = ssGetOutputPortRealSignal(S,NUM\_OUTPUT\_PORTS\_IMU+OUTPUT\_PORT\_IDX\_ARM);

pArm2[0] = frameIMU2[0].arm;

pArm2[1] = frameIMU2[1].arm;

pXDir2 = ssGetOutputPortRealSignal(S,NUM\_OUTPUT\_PORTS\_IMU+OUTPUT\_PORT\_IDX\_XDIR);

pXDir2[0] = frameIMU2[0].xDir;

pXDir2[1] = frameIMU2[1].xDir;

}

// increment and store new iteration value

ssGetIWork(S)[IDX\_ITER] = ++iter;

DB\_MYO\_SFUN("EXIT mdlOutputs()\n");

} /\* end mdlOutputs \*/

static void mdlTerminate(SimStruct \*S)

{

DB\_MYO\_SFUN("EXIT mlTerminate()\n");

DB\_MYO\_SFUN("Fetch Hub and DataCollector pointers ...\n");

myo::Hub\* pHub = (myo::Hub \*) ssGetPWork(S)[IDX\_HUB];

DataCollector\* pCollector = (DataCollector \*) ssGetPWork(S)[IDX\_COLLECTOR];

DB\_MYO\_SFUN("Unsetting runThreadFlag, waiting for thread, deleting thread ...\n");

gRunThreadFlag = false; // thread sees this flag and exits

WaitForSingleObject( ghThread, INFINITE );

CloseHandle( ghThread );

ghThread = NULL;

CloseHandle (ghMutex);

ghMutex = NULL;

DB\_MYO\_SFUN("Deleting Hub and DataCollector ...\n");

if (pHub!=NULL)

delete pHub;

if (pCollector!=NULL)

delete pCollector;

DB\_MYO\_SFUN("EXIT mlTerminate()\n");

}

#ifdef MATLAB\_MEX\_FILE /\* Is this file being compiled as a MEX-file? \*/

#include "simulink.c" /\* MEX-file interface mechanism \*/

#else

#include "cg\_sfun.h" /\* Code generation registration function \*/

#endif