// comment the following line to remove debug output via mexPrintf()

//#define DEBUG\_MYO\_MEX

#ifdef DEBUG\_MYO\_MEX

#define DB\_MYO\_MEX(fmt, ...) mexPrintf(fmt, ##\_\_VA\_ARGS\_\_)

#else

#define DB\_MYO\_MEX(fmt, ...)

#endif

#include <mex.h> // mex api

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

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

#include <queue> // standard type for fifo queue

#include "myo/myo.hpp"

#include "myo\_class.hpp"

// macros

#define MAKE\_NEG\_VAL\_ZERO(val) (val<0)?(0):(val)

// indeces of output args (into plhs[\*])

#define DATA\_STRUCT\_OUT\_NUM 0

// indeces of data fields into data output struct

#define QUAT\_FIELD\_NUM 0

#define GYRO\_FIELD\_NUM 1

#define ACCEL\_FIELD\_NUM 2

#define POSE\_FIELD\_NUM 3

#define ARM\_FIELD\_NUM 4

#define XDIR\_FIELD\_NUM 5

#define EMG\_FIELD\_NUM 6

#define NUM\_FIELDS 7

const char\* output\_fields[] = {"quat","gyro","accel","pose","arm","xDir","emg"};

// program behavior parameters

#define STREAMING\_TIMEOUT 5

#define INIT\_DELAY 1000

#define RESTART\_DELAY 500

#define READ\_BUFFER 2

// program state

volatile bool runThreadFlag = false;

// global data

DataCollector collector;

myo::Hub\* pHub = NULL;

myo::Myo\* pMyo = NULL;

unsigned int countMyosRequired = 1;

// threading

unsigned int threadID;

HANDLE hThread;

HANDLE hMutex;

// thread routine

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

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

// acquire lock then write data into queue

DWORD dwWaitResult;

dwWaitResult = WaitForSingleObject(hMutex,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(hMutex)) { return FALSE; } // acquired bad mutex

break;

case WAIT\_ABANDONED:

return FALSE; // acquired bad mutex

}

} // end thread and return

\_endthreadex(0); //

return 0;

}

// These functions allocate and assign mxArray to return output to MATLAB

// Pseudo example usage:

// mxArray\* outData[...];

// makeOutputXXX(outData,...);

// fillOutputXXX(outData,...);

// // then assign matrices in outData to a MATLAB struct

// plhs[...] = mxCreateStructMatrix(...);

// assnOutputStruct(plhs[...],outData,...);

// Note: The size of outData must be consistent with hard code in the

// makeOutdataXXX and fillOutdataXXX functions.

void makeOutputIMU(mxArray \*outData[], unsigned int sz) {

outData[QUAT\_FIELD\_NUM] = mxCreateNumericMatrix(sz,4,mxDOUBLE\_CLASS,mxREAL);

outData[GYRO\_FIELD\_NUM] = mxCreateNumericMatrix(sz,3,mxDOUBLE\_CLASS,mxREAL);

outData[ACCEL\_FIELD\_NUM] = mxCreateNumericMatrix(sz,3,mxDOUBLE\_CLASS,mxREAL);

outData[POSE\_FIELD\_NUM] = mxCreateNumericMatrix(sz,1,mxDOUBLE\_CLASS,mxREAL);

outData[ARM\_FIELD\_NUM] = mxCreateNumericMatrix(sz,1,mxDOUBLE\_CLASS,mxREAL);

outData[XDIR\_FIELD\_NUM] = mxCreateNumericMatrix(sz,1,mxDOUBLE\_CLASS,mxREAL);

}

void makeOutputEMG(mxArray \*outData[], unsigned int sz) {

outData[EMG\_FIELD\_NUM] = mxCreateNumericMatrix(sz,8,mxDOUBLE\_CLASS,mxREAL);

}

void fillOutputIMU(mxArray \*outData[], FrameIMU f,

unsigned int row,unsigned int sz) {

\*( mxGetPr(outData[QUAT\_FIELD\_NUM]) + row+sz\*0 ) = f.quat.w();

\*( mxGetPr(outData[QUAT\_FIELD\_NUM]) + row+sz\*1 ) = f.quat.x();

\*( mxGetPr(outData[QUAT\_FIELD\_NUM]) + row+sz\*2 ) = f.quat.y();

\*( mxGetPr(outData[QUAT\_FIELD\_NUM]) + row+sz\*3 ) = f.quat.z();

\*( mxGetPr(outData[GYRO\_FIELD\_NUM]) + row+sz\*0 ) = f.gyro.x();

\*( mxGetPr(outData[GYRO\_FIELD\_NUM]) + row+sz\*1 ) = f.gyro.y();

\*( mxGetPr(outData[GYRO\_FIELD\_NUM]) + row+sz\*2 ) = f.gyro.z();

\*( mxGetPr(outData[ACCEL\_FIELD\_NUM]) + row+sz\*0 ) = f.accel.x();

\*( mxGetPr(outData[ACCEL\_FIELD\_NUM]) + row+sz\*1 ) = f.accel.y();

\*( mxGetPr(outData[ACCEL\_FIELD\_NUM]) + row+sz\*2 ) = f.accel.z();

\*( mxGetPr(outData[POSE\_FIELD\_NUM]) + row ) = f.pose.type();

\*( mxGetPr(outData[ARM\_FIELD\_NUM]) + row ) = f.arm;

\*( mxGetPr(outData[XDIR\_FIELD\_NUM]) + row ) = f.xDir;

}

void fillOutputEMG(mxArray \*outData[], FrameEMG f,

unsigned int row,unsigned int sz) {

int jj = 0;

for (jj;jj<8;jj++)

\*( mxGetPr(outData[EMG\_FIELD\_NUM]) + row+sz\*jj ) = f.emg[jj];

}

void assnOutputStruct(mxArray \*s, mxArray \*d[], int id) {

int ii = 0;

for (ii;ii<NUM\_FIELDS;ii++) {

DB\_MYO\_MEX("Setting field %d of struct element %d\n",ii+1,id);

mxSetFieldByNumber(s,id-1,ii,d[ii]);

}

}

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

{

// check for proper number of arguments

if( nrhs<1 )

mexErrMsgTxt("myo\_mex requires at least one input.");

if ( !mxIsChar(prhs[0]) )

mexErrMsgTxt("myo\_mex requires a char command as the first input.");

if(nlhs>1)

mexErrMsgTxt("myo\_mex cannot provide the specified number of outputs.");

char\* cmd = mxArrayToString(prhs[0]);

if ( !strcmp("init",cmd) ) {

// ----------------------------------------- myo\_mex init -------------

if ( mexIsLocked() )

mexErrMsgTxt("myo\_mex is already initialized.\n");

if ( nrhs<2 )

mexErrMsgTxt("myo\_mex init requires 2 inputs.\n");

if( !mxIsDouble(prhs[1]) || mxIsComplex(prhs[1]) ||

!(mxGetM(prhs[1])==1 && mxGetM(prhs[1])==1) )

mexErrMsgTxt("myo\_mex init requires a numeric scalar countMyos as the second input.");

// Get input counyMyos and set up collector accordingly

countMyosRequired = \*mxGetPr(prhs[1]);

if (countMyosRequired==1)

collector.addEmgEnabled = true;

// Instantiate a Hub and get a Myo

pHub = new myo::Hub("com.mark-toma.myo\_mex");

if ( !pHub )

mexErrMsgTxt("Hub failed to init!");

pMyo = pHub->waitForMyo(5);

if ( !pMyo )

mexErrMsgTxt("Myo failed to init!");

// configure myo and hub

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

pHub->addListener(&collector);

// instantiate mutex

hMutex = CreateMutex(NULL,FALSE,NULL);

if (hMutex == NULL)

mexErrMsgTxt("Failed to set up mutex.\n");

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

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

pHub->run(INIT\_DELAY);

if (countMyosRequired!=collector.getCountMyos())

mexErrMsgTxt("myo\_mex failed to initialize with countMyos.\n");

// Flush the data queues with syncDataSources

// Note: This pops the oldest samples of data off the front of all

// queues until only the most recent data remains

collector.syncDataSources();

// At this point we don't anticipate and errors, so we commit to

// locking this file's memory

// Note: The mexLock status is used to determine the initialization

// state in other calls

mexLock();

} else if ( !strcmp("start\_streaming",cmd) ) {

// ----------------------------------------- myo\_mex start\_streaming --

if ( !mexIsLocked() )

mexErrMsgTxt("myo\_mex is not initialized.\n");

if ( runThreadFlag )

mexErrMsgTxt("myo\_mex is already streaming.\n");

if ( nlhs>0 )

mexErrMsgTxt("myo\_mex too many outputs specified.\n");

collector.addDataEnabled = true; // lets collector handle data events

// dispatch concurrent task

runThreadFlag = true;

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

if ( !hThread )

mexErrMsgTxt("Failed to create streaming thread!\n");

DB\_MYO\_MEX("myo\_mex start\_streaming:\n\tSuccess\n");

} else if ( !strcmp("get\_streaming\_data",cmd) ) {

// ----------------------------------------- myo\_mex get\_streaming\_data

if ( !mexIsLocked() )

mexErrMsgTxt("myo\_mex is not initialized.\n");

if ( !runThreadFlag )

mexErrMsgTxt("myo\_mex is not streaming.\n");

if ( nlhs>1 )

mexErrMsgTxt("myo\_mex too many outputs specified.\n");

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

unsigned int countMyos = collector.getCountMyos();

if ( countMyos != countMyosRequired )

mexErrMsgTxt("myo\_mex countMyos is inconsistent with initialization... We lost a Myo!");

// Declarations and initializations and stuff

unsigned int iiIMU1=0; // Index into output matrices when reading queue

unsigned int iiEMG1=0;

unsigned int iiIMU2=0;

unsigned int iiEMG2=0;

unsigned int szIMU1 = 0; // Size of samples to read from queue

unsigned int szEMG1 = 0;

unsigned int szIMU2 = 0;

unsigned int szEMG2 = 0;

FrameIMU frameIMU1, frameIMU2; // Data structures returned from queue read

FrameEMG frameEMG1, frameEMG2;

// Output matrices hold numeric data

mxArray \*outData1[NUM\_FIELDS];

mxArray \*outData2[NUM\_FIELDS];

// Compute size of output matrices

szIMU1 = collector.getCountIMU(1)-READ\_BUFFER;

if (countMyos<2) {

szEMG1 = collector.getCountEMG(1)-READ\_BUFFER;

} else {

szIMU2 = collector.getCountIMU(2)-READ\_BUFFER;

}

szIMU1 = MAKE\_NEG\_VAL\_ZERO(szIMU1);

szEMG1 = MAKE\_NEG\_VAL\_ZERO(szEMG1);

szIMU2 = MAKE\_NEG\_VAL\_ZERO(szIMU2);

szEMG2 = MAKE\_NEG\_VAL\_ZERO(szEMG2);

// Initialize output matrices

makeOutputIMU(outData1,szIMU1);

makeOutputEMG(outData1,szEMG1);

makeOutputIMU(outData2,szIMU2);

makeOutputEMG(outData2,szEMG2);

// Now get ahold of the lock and iteratively drain the queue while

// filling outDataN matrices

DWORD dwWaitResult;

dwWaitResult = WaitForSingleObject(hMutex,INFINITE);

switch (dwWaitResult)

{

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

// --- CRITICAL SECTION - holding lock

while (iiIMU1<szIMU1) { // Read from Myo 1 IMU

frameIMU1 = collector.getFrameIMU(1);

fillOutputIMU(outData1,frameIMU1,iiIMU1,szIMU1);

iiIMU1++;

}

while (iiEMG1<szEMG1) { // Read from Myo 1 EMG

frameEMG1 = collector.getFrameEMG(1);

fillOutputEMG(outData1,frameEMG1,iiEMG1,szEMG1);

iiEMG1++;

}

while (iiIMU2<szIMU2) { // Read from Myo 2 IMU

frameIMU2 = collector.getFrameIMU(2);

fillOutputIMU(outData2,frameIMU2,iiIMU2,szIMU2);

iiIMU2++;

}

while (iiEMG2<szEMG2) { // Read from Myo 2 EMG

frameEMG2 = collector.getFrameEMG(2);

fillOutputEMG(outData2,frameEMG2,iiEMG2,szEMG2);

iiEMG2++;

}

// END CRITICAL SECTION - release lock

if ( !ReleaseMutex(hMutex))

mexErrMsgTxt("Failed to release lock\n");

break;

case WAIT\_ABANDONED:

mexErrMsgTxt("Acquired abandoned lock\n");

break;

}

// Assign outDataN matrices to MATLAB struct matrix

plhs[DATA\_STRUCT\_OUT\_NUM] = mxCreateStructMatrix(1,countMyos,NUM\_FIELDS,output\_fields);

assnOutputStruct(plhs[DATA\_STRUCT\_OUT\_NUM], outData1, 1);

if (countMyos>1) {

assnOutputStruct(plhs[DATA\_STRUCT\_OUT\_NUM], outData2, 2);

}

} else if ( !strcmp("stop\_streaming",cmd) ) {

// ----------------------------------------- myo\_mex stop\_streaming ---

if ( !mexIsLocked() )

mexErrMsgTxt("myo\_mex is not initialized.\n");

if ( !runThreadFlag )

mexErrMsgTxt("myo\_mex is not streaming.\n");

if ( nlhs>0 )

mexErrMsgTxt("myo\_mex too many outputs specified.\n");

// Terminate thread and reset state

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

WaitForSingleObject( hThread, INFINITE );

CloseHandle( hThread );

hThread = NULL;

// Terminate data logging and reset state

collector.addDataEnabled = false; // stop handling data events

collector.syncDataSources(); // sync data up again (flushes queue)

} else if ( !strcmp("delete",cmd) ) {

// ----------------------------------------- myo\_mex delete -----------

if ( !mexIsLocked() )

mexErrMsgTxt("myo\_mex is not initialized.\n");

if ( runThreadFlag )

mexErrMsgTxt("myo\_mex cannot be deleted while streaming. Call stop\_streaming first.\n");

if ( nlhs>0 )

mexErrMsgTxt("myo\_mex too many outputs specified.\n");

CloseHandle (hMutex);

hMutex = NULL;

mexUnlock();

if (pHub!=NULL)

delete pHub;

} else {

mexErrMsgTxt("unknown command!\n");

}

return;

}