#ifndef MYO\_CLASS\_HPP

#define MYO\_CLASS\_HPP

// comment the following line to remove debug output via DB\_MYO\_CLASS()

//#define DEBUG\_MYO\_CLASS

#ifdef DEBUG\_MYO\_CLASS

#define DB\_MYO\_CLASS(fmt, ...) printf(fmt, ##\_\_VA\_ARGS\_\_)

#else

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

#endif

#include "mex.h"

#include "myo/myo.hpp" // myo sdk cpp binding

#include <array> // myo sdk emg data

#include <vector>

#include <queue>

// --- Data Frames

// These structures are used as return types in MyoData and DataCollector

// methods to return a single sample of data from one time instance

// IMU data frame

struct FrameIMU

{

myo::Quaternion<float> quat;

myo::Vector3<float> gyro;

myo::Vector3<float> accel;

myo::Pose pose;

myo::Arm arm;

myo::XDirection xDir;

}; // FrameIMU

// EMG data frame

struct FrameEMG

{

std::array<int8\_t,8> emg;

}; // FrameEMG

// END Data Frames

// --- MyoData

// This class is used to keep track of the data for one physical Myo.

// Typical use may be to instantiate a MyoData for each unique myo received

// by a DataCollector. Then, subsequent calls into add<data> and get<data>

// can be used to receive contiguous collections of IMU, EMG, and Meta data

class MyoData

{

// Data frames for returning data samples

FrameIMU frameIMU;

FrameEMG frameEMG;

// Pointer to a Myo device instance provided by hub

myo::Myo\* pMyo;

bool addEmgEnabled;

// IMU data queues and state information

std::queue<myo::Quaternion<float>,std::deque<myo::Quaternion<float>>> quat;

std::queue<myo::Vector3<float>,std::deque<myo::Vector3<float>>> gyro;

std::queue<myo::Vector3<float>,std::deque<myo::Vector3<float>>> accel;

std::queue<myo::Pose,std::deque<myo::Pose>> pose;

std::queue<myo::Arm,std::deque<myo::Arm>> arm;

std::queue<myo::XDirection,std::deque<myo::XDirection>> xDir;

uint64\_t timestampIMU;

unsigned int countIMU;

// EMG data queues and state information

std::queue<std::array<int8\_t,8>,std::deque<std::array<int8\_t,8>>> emg;

unsigned int semEMG;

unsigned int countEMG;

uint64\_t timestampEMG;

// syncIMU

// Called before pushing quat, gyro, or accel

// This updates the timestampIMU member to keep track of the IMU datas.

// If it is detected that a sample of quat, gyro, or accel was skipped,

// the previous value for that data source is copied to fill the gap.

// This is zero-order-hold interpolation of missing timeseries data.

// Furthermore, since event-based meta data is sampled on the EMG vector

// as well, we fill their queues up to the future size of emg to maintain

// consistency there.

void syncIMU(uint64\_t ts)

{

if ( ts > timestampIMU ) {

// fill IMU data (only if we missed samples)

while ( quat.size() < countIMU ) {

myo::Quaternion<float> q = quat.back();

quat.push(q);

}

while ( gyro.size() < countIMU ) {

myo::Vector3<float> g = gyro.back();

gyro.push(g);

}

while ( accel.size() < countIMU ) {

myo::Vector3<float> a = accel.back();

accel.push(a);

}

countIMU++;

timestampIMU = ts;

// fill pose, arm, and xDir up to the new countEMG

myo::Pose p = pose.back();

while ( pose.size()<(countIMU-1) ) { pose.push(p); }

myo::Arm a = arm.back();

while ( arm.size()<(countIMU-1) ) { arm.push(a); }

myo::XDirection x = xDir.back();

while ( xDir.size()<(countIMU-1) ) { xDir.push(x); }

}

}

// sync<Pose/Arm/XDir>

// This event-based meta data is sampled on the EMG vector, so we fill

// their queues up to the future size of emg to maintain consistency.

// Things would theoretically break down if these events fired more

// frequently than the emg data, but I think that's impossible. It's

// highly unlikely for sure! But still, the boolean return values allow

// for guarding against this case.

bool syncPose(uint64\_t ts)

{

if (pose.size() == countIMU)

return false;

myo::Pose p = pose.back();

while ( pose.size()<(countIMU-1) ) { pose.push(p); }

return true;

}

bool syncArm(uint64\_t ts)

{

if (arm.size() == countIMU)

return false;

myo::Arm a = arm.back();

while ( arm.size()<(countIMU-1) ) { arm.push(a); }

return true;

}

bool syncXDir(uint64\_t ts)

{

if (xDir.size() == countIMU)

return false;

myo::XDirection x = xDir.back();

while ( xDir.size()<(countIMU-1) ) { xDir.push(x); }

return true;

}

// syncEMG

// Called before pushing emg data

// This updates the timestampEMG member to keep track of the EMG datas.

// If it is detected that a sample of emg was skipped, the previous value

// for that data source is copied to fill the gap. This operation sounds

// trivial, but it isn't quite as simple as you'd expect. Myo SDK

// provides emg data samples in pairs for each unique timestamp. That is,

// timeN emgN

// time0 emg1

// time0 emg2

// time1 emg3

// time1 emg4

// timeK emg(2\*K+1)

// timeK emg(2\*K+2)

// So then, we keep track of the number of new emg samples received

// without a new timestamp in semEMG. Then pad emg with the last value if

// it's detected that a sample was missed.

// This is zero-order-hold interpolation of missing timeseries data.

void syncEMG(uint64\_t ts)

{

if ( ts>timestampEMG ) { // new timestamp

if ( 0==(semEMG%2) ) {

std::array<int8\_t,8> e = emg.back();

emg.push(e);

}

semEMG = 0; // reset sem

} else {

semEMG++; // increment sem

}

countEMG++;

timestampEMG = ts;

}

public:

MyoData(myo::Myo\* myo, uint64\_t timestamp, bool \_addEmgEnabled)

: countIMU(1), countEMG(1), semEMG(0), timestampIMU(0), timestampEMG(0)

{

pMyo = myo; // pointer to myo::Myo

// perform some operations on myo to set it up before subsequent use

pMyo->unlock(myo::Myo::unlockHold);

if (\_addEmgEnabled) {

pMyo->setStreamEmg(myo::Myo::streamEmgEnabled);

countEMG = 1;

std::array<int8\_t,8> \_emg;

emg.push(\_emg);

timestampEMG = timestamp;

}

addEmgEnabled = \_addEmgEnabled;

// fill up the other private members

myo::Quaternion<float> \_quat; // dummy default objects

myo::Vector3<float> \_gyro;

myo::Vector3<float> \_accel;

quat.push(\_quat); // push them back onto queues

gyro.push(\_gyro);

accel.push(\_accel);

pose.push(myo::Pose::unknown);

arm.push(myo::armUnknown);

xDir.push(myo::xDirectionUnknown);

timestampIMU = timestamp;

}

// Myo is owned by hub... no cleanup necessary here

~MyoData() {}

// getFrameXXX

// Read a sample of data from the IMU or EMG queues

FrameIMU &getFrameIMU()

{

countIMU = countIMU - 1;

frameIMU.quat = quat.front();

frameIMU.gyro = gyro.front();

frameIMU.accel = accel.front();

frameIMU.pose = pose.front();

frameIMU.arm = arm.front();

frameIMU.xDir = xDir.front();

quat.pop();

gyro.pop();

accel.pop();

pose.pop();

arm.pop();

xDir.pop();

return frameIMU;

}

FrameEMG &getFrameEMG()

{

countEMG = countEMG - 1;

frameEMG.emg = emg.front();

emg.pop();

return frameEMG;

}

// getInstance

// Get the pointer to this myo::Myo\* object. Use this function to test

// equivalence of this MyoData's myo pointer to another.

myo::Myo\* getInstance() { return pMyo; }

// getCountXXX

// Get the number of valid samples in the IMU or EMG queues

unsigned int getCountIMU() { return countIMU; }

unsigned int getCountEMG() { return countEMG; }

// syncDataSources

// Pops data off of queues until there are at most two bad samples.

// Subsequently, a third bad sample may fill the read head of the queue.

// Use this functions to put the data vector into a known state. Throw

// away the first three samples of data read after this call. The rest

// should be contiguous on the maximum sample rate for the data source.

void syncDataSources()

{

FrameIMU frameIMU;

while ( getCountIMU() > 1 )

frameIMU = getFrameIMU();

FrameEMG frameEMG;

while ( getCountEMG() > 1 )

frameEMG = getFrameEMG();

}

// add<data> functions

// All of these perform two operations:

// \* sync<type>

// Syncs up the data queues that are being samples on the same time

// base.

// \* <data>.push(\_<data>) pushes new data onto its queue

void addQuat(const myo::Quaternion<float>& \_quat, uint64\_t timestamp)

{

syncIMU(timestamp);

quat.push(\_quat);

}

void addGyro(const myo::Vector3<float>& \_gyro, uint64\_t timestamp)

{

syncIMU(timestamp);

gyro.push(\_gyro);

}

void addAccel(const myo::Vector3<float>& \_accel, uint64\_t timestamp)

{

syncIMU(timestamp);

accel.push(\_accel);

}

void addEmg(const int8\_t \*\_emg, uint64\_t timestamp)

{

if (!addEmgEnabled ) { return; }

syncEMG(timestamp);

std::array<int8\_t,8> tmp;

int ii = 0;

for (ii;ii<8;ii++) {tmp[ii]=\_emg[ii];}

emg.push(tmp);

}

void addPose(myo::Pose \_pose, uint64\_t timestamp)

{

if ( syncPose(timestamp) )

pose.push(\_pose);

}

void addArm(myo::Arm \_arm, uint64\_t timestamp)

{

if ( syncArm(timestamp) )

arm.push(\_arm);

}

void addXDir(myo::XDirection \_xDir, uint64\_t timestamp)

{

if ( syncXDir(timestamp) )

xDir.push(\_xDir);

}

}; // MyoData

// END MyoData

// --- DataCollector

// This class provides the link to Myo SDK, encapsulation of the MyoData

// class that manages data queues for each Myo device, and provides access

// to that data.

// \* Register this class with a myo::Hub to trigger calls back into the

// on<event> functions below.

// \* Call myo::Hub::run to allow callbacks to write data into the

// encapsulated MyoData objects in knownMyos

// \* Call getFrameXXX(id) at most getCountXXX(id) times to read samples of

// FrameXXX data, where id is the 1-indexed id for a Myo device with

// maximum value getCountMyos()

class DataCollector : public myo::DeviceListener

{

std::vector<MyoData\*> knownMyos;

public:

bool addDataEnabled; // unset to disable callbacks (they'll fall-through)

bool addEmgEnabled;

DataCollector()

: addDataEnabled(false), addEmgEnabled(false)

{}

~DataCollector()

{

// destruct all MyoData\* in knownMyos

int ii=0;

for (ii;ii<knownMyos.size();ii++)

{

delete knownMyos[ii];

}

}

// --- Wrappers for MyoData members

// These functions basically vectorize similarly named members of MyoData

// on the elements of knownMyos

unsigned int getCountIMU(int id) { return knownMyos[id-1]->getCountIMU(); }

unsigned int getCountEMG(int id) { return knownMyos[id-1]->getCountEMG(); }

const FrameIMU &getFrameIMU( int id ) { return knownMyos[id-1]->getFrameIMU(); }

const FrameEMG &getFrameEMG( int id ) { return knownMyos[id-1]->getFrameEMG(); }

void syncDataSources()

{

int ii = 0;

for (ii;ii<knownMyos.size();ii++)

knownMyos[ii]->syncDataSources();

}

// getCountMyos

// Get current number of myos

const unsigned int getCountMyos() { return knownMyos.size(); }

// getMyoID

// Returns the (1-indexed) ID of input myo in knownMyos. If myo isn't in

// knownMyos yet, it's added. This function can be used to index into

// knownMyos with a myo pointer by: knownMyos[getMyoID(myo)-1].

const unsigned int getMyoID(myo::Myo\* myo,uint64\_t timestamp)

{

// search myos in knownMyos for myo

for (size\_t ii = 0; ii < knownMyos.size(); ii++)

if (knownMyos[ii]->getInstance() == myo) { return ii+1; }

// add myo to a new MyoData\* in knowmMyos if it doesn't exist yet

knownMyos.push\_back(new MyoData(myo,timestamp,addEmgEnabled));

return knownMyos.size();

}

// on<event> Callbacks

// \* Refer to the Myo SDK documentation for information on the mechanisms

// that trigger these callback functions in myo::Hub.

// \* All of these invoke getMyoID() so as to automatically add myo to

// knownMyos without explicitly expressing this logic.

// \* The on<data>Data functions fall-through when !addDataEnabled

// \* Some device state meta data is maintained in the state change events

void onPair(myo::Myo\* myo, uint64\_t timestamp, myo::FirmwareVersion firmwareVersion)

{

unsigned int tmp = getMyoID(myo,timestamp);

}

void onConnect(myo::Myo \*myo, uint64\_t timestamp, myo::FirmwareVersion firmwareVersion)

{

unsigned int tmp = getMyoID(myo,timestamp);

}

void onDisconnect(myo::Myo\* myo, uint64\_t timestamp)

{

knownMyos.erase(knownMyos.begin()+getMyoID(myo,timestamp)-1);

}

void onLock(myo::Myo\* myo, uint64\_t timestamp)

{

// shamelessly unlock the device

myo->unlock(myo::Myo::unlockHold);

}

void onOrientationData(myo::Myo\* myo, uint64\_t timestamp, const myo::Quaternion<float>& q)

{

if (!addDataEnabled) { return; }

knownMyos[getMyoID(myo,timestamp)-1]->addQuat(q,timestamp);

}

void onGyroscopeData (myo::Myo\* myo, uint64\_t timestamp, const myo::Vector3<float>& g)

{

if (!addDataEnabled) { return; }

knownMyos[getMyoID(myo,timestamp)-1]->addGyro(g,timestamp);

}

void onAccelerometerData (myo::Myo\* myo, uint64\_t timestamp, const myo::Vector3<float>& a)

{

if (!addDataEnabled) { return; }

knownMyos[getMyoID(myo,timestamp)-1]->addAccel(a,timestamp);

}

void onEmgData(myo::Myo\* myo, uint64\_t timestamp, const int8\_t \*e)

{

if (!addDataEnabled||!addEmgEnabled) { return; }

knownMyos[getMyoID(myo,timestamp)-1]->addEmg(e,timestamp);

}

void onPose(myo::Myo\* myo, uint64\_t timestamp, myo::Pose p)

{

if (!addDataEnabled) { return; }

knownMyos[getMyoID(myo,timestamp)-1]->addPose(p,timestamp);

}

//void onUnpair(myo::Myo\* myo, uint64\_t timestamp) {}

void onArmSync(myo::Myo\* myo, uint64\_t timestamp, myo::Arm arm, myo::XDirection xDirection) {

if (!addDataEnabled) { return; }

knownMyos[getMyoID(myo,timestamp)-1]->addArm(arm,timestamp);

knownMyos[getMyoID(myo,timestamp)-1]->addXDir(xDirection,timestamp);

}

void onArmUnsync(myo::Myo\* myo, uint64\_t timestamp) {

if (!addDataEnabled) { return; }

// infer state changes of arm and xdir

myo::Arm newArm = myo::Arm::armUnknown;

myo::XDirection newXDir = myo::XDirection::xDirectionUnknown;

knownMyos[getMyoID(myo,timestamp)-1]->addArm(newArm,timestamp);

knownMyos[getMyoID(myo,timestamp)-1]->addXDir(newXDir,timestamp);

}

//void onUnlock(myo::Myo\* myo, uint64\_t timestamp) {}

}; // DataCollector

// END DataCollector

#endif // ndef MYO\_CLASS\_HPP