|  |
| --- |
| WESTA |
| ChapR Development Code |
| v.02 |

|  |
| --- |
| Rachel  8/22/2013 |

Contents

[ChapR.ino 3](#_Toc364939840)

[BT.cpp 7](#_Toc364939841)

[ChapRName.cpp 13](#_Toc364939842)

[VDIP.cpp 15](#_Toc364939843)

[VDIPSPI.cpp 22](#_Toc364939844)

[VDIPlow.cpp 25](#_Toc364939845)

[blinky.cpp 30](#_Toc364939846)

[button.cpp 31](#_Toc364939847)

[nxt.cpp 32](#_Toc364939848)

[sound.cpp 35](#_Toc364939849)

[NameReset.ino 36](#_Toc364939850)

[ChapRName.cpp 37](#_Toc364939851)

# ChapR.ino

#include "config.h"

#include "VDIPSPI.h"

#include "VDIP.h"

#include "BT.h"

#include "blinky.h"

#include "sound.h"

#include "nxt.h"

#include "ChapRName.h"

#include <EEPROM.h>

#include "button.h"

#include <avr/sleep.h>

// it is really strange, but BT.h won't compile correctly unless this .ino

// file also includes SoftwareSerial.h...really weird...

#include <SoftwareSerial.h>

// LIGHTS

// - Power Light - slow flash during boot

// - solid when operational

// - fast flash means battery is low

// - off means the ChapR is off or battery dead

//

// - BT Light - off during boot

// - solid when BT connected

// - fast flash when discoverable (NEW BRICK MODE)

// - slow flash when trying to connect to a brick (AUTO CONNECTION MODE)

//

#ifdef V01

#define BT\_9600BAUD A5

#define BT\_CONNECTED A4

#define BT\_MODE A3

#define BT\_RX 12

#define BT\_TX 13

#define BT\_RESET 11

#define VDIP\_CLOCK 6

#define VDIP\_MOSI 7

#define VDIP\_MISO 8

#define VDIP\_CS 9

#define VDIP\_RESET 10

#define LED\_POWER A2

#define LED\_INDICATE A1

#define BUTTON A0

#define TONEPIN 5

#endif

#ifdef V02

#define BT\_9600BAUD 5

#define BT\_CONNECTED 3

#define BT\_MODE 4

#define BT\_RX 12

#define BT\_TX 13

#define BT\_RESET 11

#define VDIP\_CLOCK 6

#define VDIP\_MOSI 7

#define VDIP\_MISO 8

#define VDIP\_CS 9

#define VDIP\_RESET 10

#define LED\_POWER A2

#define LED\_INDICATE A1

#define BUTTON A0

#define TONEPIN A3

#endif

VDIP vdip(VDIP\_CLOCK, VDIP\_MOSI, VDIP\_MISO, VDIP\_CS, VDIP\_RESET);

BT bt(BT\_RX, BT\_TX, BT\_RESET, BT\_MODE, BT\_9600BAUD, BT\_CONNECTED);

blinky powerLED(LED\_POWER);

blinky indicateLED(LED\_INDICATE);

button theButton(BUTTON);

ChapRName myName; //myName() doesn't work because it thinks it's declaring a function with return type ChapRName

sound beeper(TONEPIN);

#define LOCAL\_SERIAL\_BAUD 38400

void setup()

{

powerLED.fast(); // flash the power LED during boot

Serial.begin(LOCAL\_SERIAL\_BAUD); // the serial monitor operates at this BAUD

Serial.write("ChapR v0.2 up!\n");

Serial.println(myName.get());

// standard init stuff happens here

// NOTE, though, that the object init from the VDIP and BT objects has already occured

// which is good, because the VDIP takes some time to get up and running

// check the button to see if it was pressed upon boot, if so, enter config mode

if (digitalRead(BUTTON) == HIGH) { // the button has a pull-down, so normally LOW

bt.configMode(myName.get());

powerLED.slow();

} else {

bt.opMode();

powerLED.on();

}

Serial.write("Waiting on VDIP sync\n");

while(!vdip.sync()) { // while waiting, update the LED status

powerLED.update();

}

Serial.write("Out of VDIP sync\n");

vdip.deviceUpdate(); // get initial device setup

beeper.confirm();

}

byte emptyJSData[] = { 0x80, 0x80, 0x80, 0x80, 0x08, 0x00, 0x04, 0x00 };

byte joy1data[] = { 0x80, 0x80, 0x80, 0x80, 0x08, 0x00, 0x04,0x00 };

byte joy2data[] = { 0x80, 0x80, 0x80, 0x80, 0x08, 0x00, 0x04,0x00 };

void jsprint(byte \*js,char \*label)

{

Serial.print(label);

for (int i=0; i < 8; i++) {

Serial.print(js[i],HEX);

Serial.print(" ");

}

Serial.println("");

}

void enterZombieMode()

{

powerLED.off();

indicateLED.off();

bt.zombieMode();

vdip.zombieMode();

// turn off the Pro Mini green LED - it is directly connected to pin 13

// so turn it off at the very last minute so we don't interfere with whatever

// else happens to be connected to pin 13

digitalWrite(13,LOW); // turns off the green LED

set\_sleep\_mode(SLEEP\_MODE\_PWR\_DOWN);

sleep\_enable();

sleep\_cpu();

}

long lastAnyAction = 0;

long lastJSAction = 0;

bool isLowPower = false;

//these two constants define the time before entering power saving mode and are in milliseconds

// (the second commented out versions are for debugging)

#define LOWPOWERTIMEOUT 300000

//#define LOWPOWERTIMEOUT 10000

#define ZMODETIMEOUT 600000

//#define ZMODETIMEOUT 20000

#define DEVICE\_UPDATE\_LOOP\_COUNT 50

void loop()

{

static int loopCount = 0;

static bool wasConnected = false;

bool js1 = false;

bool js2 = false;

bool wfs = false;

// check each joystick that is connected, and grab a packet of information from it

// the joysticks return 8 bytes of info

if (vdip.getJoystick(1,(char \*)joy2data) == 8) {

js2 = true;

// jsprint(joy2data,"(2):");

}

if (vdip.getJoystick(0,(char \*)joy1data) == 8) {

js1 = true;

// jsprint(joy1data,"(1):");

}

if (theButton.hasChanged()){

wfs = true;

}

if((loopCount % DEVICE\_UPDATE\_LOOP\_COUNT) == 0) {

vdip.deviceUpdate();

}

// check to see if we're connected to the brick - turn on the light if so

// if not connected, blink the thing

if(bt.connected()) {

indicateLED.on();

powerLED.on();

if (!wasConnected) {

wasConnected = true;

beeper.squeep();

}

} else {

if (wasConnected) {

wasConnected = false;

}

indicateLED.slow();

}

// if we're connected, send out a joystick update

if(bt.connected() && (!isLowPower || wfs)) {

byte outbuff[25];

int size;

int UserMode = 0;

int StopPgm = (theButton.isPressed())?0:1;

size = nxtMsgCompose(outbuff,

UserMode,

StopPgm,

// (js1)?joy1data:emptyJSData,

// (js2)?joy2data:emptyJSData);

joy1data,

joy2data);

(void)bt.btWrite(outbuff,size);

}

//checks to see if we should enter a power saving mode (if 5 min has passed)

if (js1 || js2 || wfs){ //if something has happened, make note of the time since boot

lastAnyAction = millis();

}

if (js1 || js2) { //if a joystick changed

lastJSAction = millis();

isLowPower = false;

}

if (millis() - lastJSAction >= LOWPOWERTIMEOUT){

isLowPower = true;

powerLED.slow();

}

if (millis() - lastAnyAction >= ZMODETIMEOUT){

enterZombieMode();

}

// update the state of the LEDs - this should always be done at the end of the loop

powerLED.update();

indicateLED.update();

// allow only a certain number of updates - saves battery

delay(5);

loopCount++;

}

## BT.cpp

#include <Arduino.h>

#include "config.h"

#include "BT.h"

#include "SoftwareSerial.h"

//

// constructor

//

BT::BT(uint8\_t receive,

uint8\_t transmit,

uint8\_t reset, // TO BT> used to reset the BT module (necessary?)

uint8\_t automode, // TO BT> sets auto connect mode - off when pairing

uint8\_t baud, // TO BT> sets 9600 baud mode

uint8\_t connect) : // FROM BT< high when connected

SoftwareSerial(receive,transmit),

\_rstPin(reset),

\_autoModePin(automode),

\_9600BaudPin(baud),

\_connectPin(connect)

{

pinMode(\_connectPin,INPUT);

// the following pins are wrong on the board - they use a 1 resistor

// mechanism that needs to be replaced by a 2 resistor voltage divider

// like that used on the TX pin. when that is fixed, the following

// will be used:

//

// pinMode(\_rstPin,OUTPUT);

// pinMode(\_autoModePin,OUTPUT);

// pinMode(\_9600BaudPin,OUTPUT);

pinMode(\_rstPin,INPUT);

#ifdef V01

pinMode(\_autoModePin,INPUT);

pinMode(\_9600BaudPin,INPUT);

#endif

#ifdef V02

pinMode(\_autoModePin,OUTPUT);

pinMode(\_9600BaudPin,OUTPUT);

#endif

}

//

// btSpecialPin() - the three pins above need to be fixed on the board

// to a voltage divider. Until then, this call

// needs to be used for these pins. when that is

// fixed, blow away this routine.

//

void BT::specialPin(int pin, int value)

{

if (value == HIGH) {

pinMode(pin,INPUT); // lets the pull-up resistor take over

// high-impedance mode

} else {

digitalWrite(pin,LOW); // drive the low first

pinMode(pin,OUTPUT); // then turn it into an output

}

}

//

// reset() - reset the bluetooth device - note that it is separate

// from the sync/baud rate setup. Use sync() for that.

// Note that at least 500 ms needs to lapse after this

// call before sync() is called.

//

void BT::reset()

{

specialPin(\_rstPin,LOW); // to be changed to digitalWrite(\_rstPin,LOW)

// needs to be low for min 160us, but 10 ms works great

delay(10);

specialPin(\_rstPin,HIGH); // to be changed to digitalWrite(\_rstPin,HIGH)

// reset seems to need a nice long wait before the RN-42 will respond well

// to serial input - I'm seeing a weird immediate echo on the TX line if I

// don't wait for this long

delay(500);

}

//

// BT MODES -------------------------------------------------------

//

// The BT RN-42 module only has a UART connection, so getting the

// thing up and running requires that you connect with the RN-42 at

// the same speed. The devices always comes from the factory set at

// 115200 baud. This would be a fine speed to communicate at, EXCEPT

// that we need to use the SoftwareSerial Arduino library to talk to

// it, and that thing can't reliably do 115200 on a 16MHz Uno R3.

// From all reports, it can do 38400 reliably (send and recv).

//

// So, we, instead, operate the RN-42 in two modes "configation"

// mode and "operational" mode. The configuration mode is only

// entered when needed, and the "operational" mode is the normal mode

// of operation. Here how it works:

//

// CONFIG mode is entered whenever the ChapR needs to pair with a

// brick. CONFIG mode is entered by pressing and holding the button

// upon boot.

//

// NOTE that the following pins are defined:

//

// GPIO6 - the \_autoModePin

// GPIO7 - the \_9600BaudPin

// GPIO2 - the \_connetPin

//

// When CONFIG mode is requested, the following things occur:

//

// pull LOW GPIO6 // disconnects and terminates auto connect mode (has pull-up)

// allow HIGH GPIO7 // (normally held low) sets 9600 baud

// send "$$$" // enters command mode (after each command, wait)

// SU,38 // sets baud to 38400 into firmware for next time

// SN,<name> // sets the name (use the EEPROM stored value)

// SM,4 // sets the auto connect mode for next time

// U,38.4,N // switches to 38400 and exits command mode

// pull LOW GPIO7 // removes 9600 baud mode

//

//

// At this point, wait for a connect just like normal. HOWEVER,

// we are discoverable because we've left GPIO6 LOW. This means

// that it may not try to reconnect after the first time with a

// connection. We MAY want to change this to notice the connection

// the first time, and re-enter auto mode.

//

// If no CONFIG mode requested, enter OPERATIONAL MODE

//

// pull LOW GPIO7 // this should be the boot-up default

// allow HIGH GPIO6 // turns on auto-connect mode

//

// then wait for a connect just like normal

void BT::configMode(char \*name)

{

// get out of auto connect mode and set 9600 baud

autoConnectMode(false);

baud9600mode(true);

begin(9600); // set the SoftwareSerial baud rate appropriately

// each mode configuration starts with a reset() to ensure that

// the latest modes are enabled and baud rates set

reset();

delay(100); //was at 10, which worked fine, but when it mysteriously stopped replying, 100 worked better

btSend("$$$"); // get into command mode

delay(30);

btSend("SU,"); // set appropriate baud

btSend(BT\_SU\_BAUD);

btSend("\r");

delay(30);

btSend("SN,"); // set the appropriate name

btSend(name);

btSend("\r");

delay(30);

btSend("SM,4"); // auto connect mode

btSend("\r");

delay(30);

btSend("SR,Z"); // erased previously stored connection

btSend("\r");

delay(30);

btSend("SX,1"); // set bonding mode (only stored device can attach)

btSend("\r");

delay(30);

btSend("S?,1"); // attempt master/slave flip

btSend("\r");

delay(30);

btSend("U,"); // do an immediate baud rate setting

btSend(BT\_U\_BAUD); // to eliminate the need for a reboot

btSend(",N\r"); // (exits command mode too)

baud9600mode(false); // get out of 9600 mode, but leave auto connect off

}

void BT::setRemoteAddress(char \*address)

{

autoConnectMode(false);

baud9600mode(true);

begin(9600); // set the SoftwareSerial baud rate appropriately

// each mode configuration starts with a reset() to ensure that

// the latest modes are enabled and baud rates set

reset();

delay(100); //was at 10, which worked fine, but when it mysteriously stopped replying, 100 worked better

btSend("$$$"); // get into command mode

delay(30);

btSend("SR,");

btSend(address);

btSend("\r");

baud9600mode(false); // get out of 9600 mode, but leave auto connect off

autoConnectMode(true);

reset();

delay(100);

}

void BT::opMode()

{

// fire-up auto connect mode and kill 9600 baud

autoConnectMode(true);

baud9600mode(false);

begin(BT\_OP\_BAUD); // set the SoftwareSerial baud rate appropriately

// each mode configuration starts with a reset() to ensure that

// the latest modes are enabled and baud rates set

reset();

delay(100); //gives time for the reset to occur (things didn't work without it)

// we're in 38400 baud in this case, or should be

btSend("$$$"); // get into command mode

delay(30);

btSend("Q,1"); // make the chapr undiscoverable (don't know if this works)

btSend("\r");

delay(30);

btSend("---"); // and out of command mode

btSend("\r");

}

void BT::zombieMode()

{

// fire-up auto connect mode and kill 9600 baud

autoConnectMode(false);

baud9600mode(false);

begin(BT\_OP\_BAUD); // set the SoftwareSerial baud rate appropriately

// each mode configuration starts with a reset() to ensure that

// the latest modes are enabled and baud rates set

reset();

delay(100); //gives time for the reset to occur (things didn't work without it)

// we're in 38400 baud in this case, or should be

btSend("$$$"); // get into command mode

delay(30);

btSend("K,"); // disconnects any connection

btSend("\r");

delay(30);

btSend("Z"); //enters deep sleep mode

btSend("\r");

}

void BT::autoConnectMode(bool turnOn)

{

#ifdef V01

if (turnOn) {

specialPin(\_autoModePin,HIGH);

} else {

specialPin(\_autoModePin,LOW);

}

#endif

#ifdef V02

if (turnOn) {

digitalWrite(\_autoModePin,HIGH);

} else {

digitalWrite(\_autoModePin,LOW);

}

#endif

}

void BT::baud9600mode(bool turnOn)

{

#ifdef V01

if (turnOn) {

specialPin(\_9600BaudPin,HIGH);

} else {

specialPin(\_9600BaudPin,LOW);

}

#endif

#ifdef V02

if (turnOn) {

digitalWrite(\_9600BaudPin,HIGH);

} else {

digitalWrite(\_9600BaudPin,LOW);

}

#endif

}

//

// connected() - returns true if the BT is connected, false otherwise

//

bool BT::connected()

{

return(digitalRead(\_connectPin) == HIGH);

}

void BT::btSend(char \*string)

{

write(string); // may end-up with a delay in here

}

void BT::btWrite(byte \*buffer, int size)

{

write(buffer,size);

}

//

// getResponse() - this routine knows about all of the potential responses from the RN-42

// so it listens for one of them. It will continue to listen until it

// gets SOME response (even a blank line) or times-out. So, this routine

// will BLOCK (stop the rest of the program) until the response is received

// or the timeout occurs.

//

//btResponse BT::getResponse(long timeout)

//{

//

//}

//

// recv() - receive data back from the BT module. This code assumes that a line is

// always return - that is, something ending in a '\r'. The given timeout

// (in ms) will cause this routine to return if a whole line doesn't

// come back (either nothing or a partial line).

//

//void BT::recv(char \*buffer, long timeout)

//{

// long \*target = millis() + timeout;

//

// do {

// if( read(

## ChapRName.cpp

#include <Arduino.h>

#include <EEPROM.h>

#include "ChapRName.h"

ChapRName::ChapRName()

{

}

char \*ChapRName::read()

{

static char buffer[15];

for (int i = 0; i < 15; i++){

buffer[i] = EEPROM.read(i);

}

return (buffer);

}

void ChapRName::write(char \*name)

{

int i = 0;

do {

EEPROM.write(i, name[i]);

i++;

} while (name[i] != '\0');

while (i < 15){

EEPROM.write(i, '\0');

i++;

}

}

char \*ChapRName::get()

{

return (read());

}

void ChapRName::setFromString(char \*name)

{

write(name);

}

bool ChapRName::setFromConsole()

{

Serial.println("Hit space then enter if you want to rename your ChapR.");

char buffer[100];

String name = "";

static int index = 0;

bool enteringName = false;

if (Serial.available() > 0){

int input = Serial.read();

buffer[index] = input;

if (buffer[index - 1] == ' '){

Serial.write("Enter your new name (must be 1 - 20 characters without spaces): ");

index = 0;

enteringName = true;

}

else if (buffer[index] == '\r'){

for (int i = 0; i <= index; i++){

name = name + buffer[i];

}

Serial.println(name);

Serial.println("Congrats! Your ChapR has been named " + name);

}

if (!enteringName){

index++;

}

}

}

bool ChapRName::setFromFlashDrive()

{

}

## VDIP.cpp

#include <Arduino.h>

#include "VDIPSPI.h"

#include "VDIP.h"

#include "nxt.h"

#include "BT.h"

//#define HAVE\_JOY1 (\_p1 && \_p1\_dev != -1 && !(\_p1\_devtype & CLASS\_BOMS) )

//#define HAVE\_JOY2 (\_p2 && \_p2\_dev != -1 && !(\_p2\_devtype & CLASS\_BOMS) )

//

// deviceUpdate() - update the USB devices if necessary.

//

void VDIP::deviceUpdate()

{

char incoming[32];

portConfig portConfigBuffer;

int i;

// Serial.println("attempting device update");

for(i=2; i--; ) {

ports[i].flag = false;

}

// first, get the data for the connected devices - note that only the first

// two logical USB ports are checked. If this code supported HUBs, it would

// be different, but it doesn't...

// Note, too, that each port must be accounted for, or its configuration gets

// zero'd out.

for (int i = 0; i < 2; i++) {

cmd(VDIP\_QD,incoming,100,i);

// given a configuration buffer, get this port's data

mapDevice(i,incoming,&portConfigBuffer);

if(portConfigBuffer.port >= 0 && portConfigBuffer.port < 2) { // we have a good assignment

ports[portConfigBuffer.port].flag = true; // mark this port as assigned

if(ports[portConfigBuffer.port].port != portConfigBuffer.port ||

ports[portConfigBuffer.port].type != portConfigBuffer.type ||

ports[portConfigBuffer.port].usbDev != portConfigBuffer.usbDev) {

// this port HAS CHANGED

ports[portConfigBuffer.port].port = portConfigBuffer.port;

ports[portConfigBuffer.port].usbDev = portConfigBuffer.usbDev;

ports[portConfigBuffer.port].type = portConfigBuffer.type;

// check incoming device and call process routines if needed

if(portConfigBuffer.type == DEVICE\_DISK) {

processDisk();

}

if(portConfigBuffer.type == DEVICE\_NXT) {

processNXT(&portConfigBuffer);

}

}

}

}

for(i=2; i--; ) {

if(!ports[i].flag) {

if (ports[i].usbDev >= 0) {

// this port must now be empty

ports[i].usbDev = -1;

if(ports[i].type == DEVICE\_DISK) {

ejectDisk();

}

if(ports[i].type == DEVICE\_NXT) {

ejectNXT();

}

}

}

}

// Serial.println("done with device update");

}

// #define DEBUG

#ifdef DEBUG

void DEBUG\_HEX\_BYTE(unsigned char c)

{

if(c < 16) {

Serial.print("0");

}

Serial.print(c,HEX);

}

//

// DEBUG\_USB\_QD() - given a buffer, print out the QD data

//

void DEBUG\_USB\_QD(int dev, unsigned char \*buffer)

{

Serial.print("UA (");

Serial.print(dev);

Serial.print("): ");

DEBUG\_HEX\_BYTE(buffer[0]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[1]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[2]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[3]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[4]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[5]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[6]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[7]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[8]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[9]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[10]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[11]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[12]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[13]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[15]);

DEBUG\_HEX\_BYTE(buffer[14]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[17]);

DEBUG\_HEX\_BYTE(buffer[16]); Serial.print(" ");

DEBUG\_HEX\_BYTE(buffer[19]);

DEBUG\_HEX\_BYTE(buffer[18]); Serial.print(" ");

Serial.println();

}

void DEBUG\_PORT\_CONFIG(portConfig \*config)

{

Serial.print("port: ");

Serial.print(config->port);

Serial.print(" - logical dev: ");

Serial.print(config->usbDev);

Serial.print(" - type: ");

Serial.print(config->type);

Serial.println("");

}

void VDIP::debug\_port\_config()

{

DEBUG\_PORT\_CONFIG(&ports[0]);

DEBUG\_PORT\_CONFIG(&ports[1]);

}

#endif

//

// mapDevice() - given an incoming array from a QD command, map the device to the

// internal variables used for interacting with the device.

// The given "dev" is the logical USB port that is used when talking

// to the VDIP. Each of two USB ports can have a logical dev, but

// only if it is connected.

//

void VDIP::mapDevice(int dev, char \*deviceReport, portConfig \*returnPortConfig)

{

// DEBUG\_USB\_QD(dev, deviceReport);

// given the report, map the port number = -1 means that it not a useful report

returnPortConfig->port = deviceReport[DEV\_LOCATION] - 1; // maps 1 => 0, and 2=> 1, and 0 => -1

returnPortConfig->usbDev = dev;

if((deviceReport[DEV\_VID] == '\x94') && (deviceReport[DEV\_VID+1] == '\x06')) {

returnPortConfig->type = DEVICE\_NXT;

} else if(deviceReport[DEV\_TYPE] == '\x08') {

returnPortConfig->type = DEVICE\_CONTROLLER;

} else if(deviceReport[DEV\_TYPE] == '\x20') {

returnPortConfig->type = DEVICE\_DISK;

} else {

returnPortConfig->type = DEVICE\_UNKNOWN;

}

}

//

// cmd() - submit a VDIP command.

//

// This routine is somewhat "low-level" in that it communicates with the VDIP

// about sending/receiving data - but much of that data is to/from the devices

// that are CONNECTED to the VDIP. So other routines often use this lower-level

// routine to implement sending/recieving to the specific device.

//

// A single VDIP command can send or receive a variable amount of data - but not

// both send AND receive a variable amount of data in a single command. So, some

// commands need a buffer of data to send out while some need a buffer for return

// data. This command uses the same buffer for both input and output.

//

// The given argument is used for the command or to specify the number of bytes

// that should be SENT with the buffer. For routines that return data, this

// routine will fill in the buffer with the return data. The caller needs to

// know what to look for from the command, however.

//

// Each command waits for one line of return feedback (after maybe waiting for data).

//

// Does a sync() before executing the command.

//

// The number of bytes put into buf is returned. This hides a lot of errors...

//

// NOTE - the timeout isn't fully implemented here - could be a problem

int VDIP::cmd(vdipcmd cmd, char \*buf, int timeout, int arg /\* = 0 \*/)

{

int i = 0;

char cbuf[32]; // command buffer - must be big enough for biggest "rbytes" below

int rbytes = 0; // how many bytes to expect

bool twoStage = false; // true if return has a number of bytes as the first stage

int sendingCmd = false; // true if the command sends out data

sync();

switch(cmd) {

case VDIP\_SUM: // suspend monitor

{

cbuf[i++] = '\x17';

}

break;

case VDIP\_DSD: // send data out command - the arg is the message size

rbytes = 0; // don't expect any return bytes for DSD

sendingCmd = true;

{

cbuf[i++] = '\x83';

cbuf[i++] = '\x20';

cbuf[i++] = (char) arg;

}

break;

case VDIP\_SC: // "set current" command - the arg is the device

rbytes = 0; // don't expect any return bytes for SC

{

cbuf[i++] = '\x86';

cbuf[i++] = ' ';

cbuf[i++] = (char)arg;

}

break;

case VDIP\_QD: // "query device" command - the arg is the device

rbytes = 32; // whole bunch back for a QD

{

cbuf[i++] = '\x85';

cbuf[i++] = ' ';

cbuf[i++] = (char)arg;

}

break;

case VDIP\_DRD: // "data read" - reads data from current device

rbytes = 1; // will return a certain number of bytes

twoStage = true;

{

cbuf[i++] = '\x84';

}

break;

case VDIP\_QP: // "query port" command

rbytes = 2; // two bytes back for the QP commands

{

cbuf[i++] = ((arg == 1)?'\x2b':'\x2c');

}

break;

}

cbuf[i++] = '\r';

(void)sendBytes(i,cbuf,0); // send the command - blocking-style

if(sendingCmd) {

(void)sendBytes(arg,buf,timeout);

}

// at this point, bytes come back for good command - this code assumes

// that the commands are all "good" - ie - no mispelled commands

if (rbytes) { // we are looking for return bytes

if(!twoStage) {

// not a two stager, so just read directly into the return buffer

readBytes(rbytes,buf,timeout);

} else {

// we have a two-stager at this point, where the first byte of the

// rbytes above specifies how many bytes to receive. If it is 0, then

// only a return line is expected.

readBytes(rbytes,cbuf,timeout); // don't overwrite buffer yet

rbytes = cbuf[0];

// there seems to be the possibility that either "Bad Command" (BC) or "Command Failed" (CF)

// comes back. In this case, ignore it and return zero.

switch(rbytes) {

case '\r':

case 'B':

case 'C': // any of these indicates zero

flush(); // so flush the rest and return

return(0);

default:

readBytes(1,cbuf,timeout); // consume the '\r'

readBytes(rbytes,buf,timeout); // then return the real bytes

break;

}

}

}

return(rbytes);

}

//

// processDisk() - called when a disk is discovered and properly located

// on P2. This routine does everything that the Chapr can

// do with a flash drive, like update the name of the Chapr.

//

void VDIP::processDisk()

{

Serial.println("disk inserted apparently");

}

void VDIP::ejectDisk()

{

Serial.println("ejected disk");

}

void VDIP::processNXT(portConfig \*portConfigBuffer)

{

Serial.println("NXT connected apparently");

byte output[50];

int size;

int i = 0;

switch(1) {

case 1:

output[i++] = 0x80; // direct command with no response

output[i++] = 0x02; // the direct command for play sound file

output[i++] = 0x00; // play sound only once

output[i++] = 'w'; // name of the sound file - '\0' terminated

output[i++] = 'o'; // 2

output[i++] = 'o'; // 3

output[i++] = 'p'; // 4

output[i++] = 's'; // 5

output[i++] = '.'; // 6

output[i++] = 'r'; // 7

output[i++] = 's'; // 8

output[i++] = 'o'; // 9

output[i++] = '\0'; // 10

output[i++] = '\0'; // 11

output[i++] = '\0'; // 12

output[i++] = '\0'; // 13

output[i++] = '\0'; // 14

output[i++] = '\0'; // 15

output[i++] = '\0'; // 16

output[i++] = '\0'; // 17

output[i++] = '\0'; // 18

output[i++] = '\0'; // 19

output[i++] = '\0'; // 20

break;

case 2:

output[i++] = 0x80; // direct command with no response

output[i++] = 0x03; // the direct command for play tone

output[i++] = 0xb8; // 440 Hz (LSB)

output[i++] = 0x02; // (MSB)

output[i++] = 0x88; // 5000 msecs (0x1388) (LSB)

output[i++] = 0x13; // (MSB)

break;

case 3:

output[i++] = 0x80; // direct command with no response

output[i++] = 0x01; // STOP!

break;

}

//cmd(VDIP\_SC,NULL,100,portConfigBuffer->usbDev);

//cmd(VDIP\_DSD,(char \*)output,100,i);

{

char \*name;

char \*btAddress;

long freeMemory;

extern BT bt;

if(nxtQueryDevice(this,portConfigBuffer->usbDev,&name,&btAddress,&freeMemory)){

Serial.println(name);

Serial.println(freeMemory);

Serial.println(btAddress);

bt.setRemoteAddress(btAddress);

delay(100);

bt.reset();

}

}

}

void VDIP::ejectNXT()

{

Serial.println("eject NXT");

}

//

// getJoyStick() - try to read data from the given joystick. Assumes

// that it is a Joystick! (for now)

//

int VDIP::getJoystick(int num, char \*databuf)

{

if (ports[num].usbDev >= 0 && ports[num].type == DEVICE\_CONTROLLER) {

cmd(VDIP\_SC,NULL,100,ports[num].usbDev);

return(cmd(VDIP\_DRD,databuf,100));

} else {

return(0); // if not a controller, return zero bytes

}

}

//

// zombieMode() - enters low-power zombie mode on the VDIP. There is no

// coming back from this - you have to reset the device.

//

void VDIP::zombieMode()

{

cmd(VDIP\_SUM,NULL,100);

}

## VDIPSPI.cpp

//#include <avr/interrupt.h>

//#include <avr/pgmspace.h>

#include <Arduino.h>

//#include <inttypes.h>

//#include <Stream.h>

#include "VDIPSPI.h"

//

// standard definitions of the clock pin

//

#define SPI\_CLK\_UP\_1 digitalWrite(\_clockPin, HIGH)

#define SPI\_CLK\_DOWN\_1 digitalWrite(\_clockPin, LOW)

//

// these definitions slow things down a bit

//

#define SPI\_CLK\_UP SPI\_CLK\_UP\_1

#define SPI\_CLK\_DOWN SPI\_CLK\_DOWN\_1

//

// constructor

//

VDIPSPI::VDIPSPI(uint8\_t clockPin, uint8\_t mosiPin, uint8\_t misoPin, uint8\_t csPin)

{

pinMode(clockPin,OUTPUT); this->\_clockPin = clockPin;

pinMode(mosiPin,OUTPUT); this->\_mosiPin = mosiPin;

pinMode(misoPin,INPUT); this->\_misoPin = misoPin;

pinMode(csPin,OUTPUT); this->\_csPin = csPin;

// start the pins off in known states

SPI\_CLK\_DOWN;

digitalWrite(\_mosiPin,LOW);

digitalWrite(\_csPin,LOW);

}

//

// private methods

//

//

// \_header() - sends out the header for a read/write for the status

// or data bytes. clock is low upon exit. cs is high

// upon exit.

//

void VDIPSPI::\_header(bool do\_read, bool statusByte)

{

SPI\_CLK\_DOWN;

digitalWrite(\_csPin, HIGH);

digitalWrite(\_mosiPin,HIGH); // prepare for interaction

SPI\_CLK\_UP;

SPI\_CLK\_DOWN;

digitalWrite(\_mosiPin,(do\_read?HIGH:LOW)); // read or write

SPI\_CLK\_UP;

SPI\_CLK\_DOWN;

digitalWrite(\_mosiPin,(statusByte?HIGH:LOW)); // targeting status or data

SPI\_CLK\_UP;

SPI\_CLK\_DOWN;

}

//void VDIPSPI::\_oldheader(bool do\_read, bool statusByte)

//{

// SPI\_CLK\_UP;

// digitalWrite(\_csPin, HIGH);

// digitalWrite(\_mosiPin,HIGH); // prepare for interaction

// SPI\_CLK\_DOWN;

// SPI\_CLK\_UP;

// digitalWrite(\_mosiPin,(do\_read?HIGH:LOW)); // read or write

// SPI\_CLK\_DOWN;

// SPI\_CLK\_UP;

// digitalWrite(\_mosiPin,(statusByte?HIGH:LOW)); // targeting status or data

// SPI\_CLK\_DOWN;

//}

//

// \_statusbit() - read the status bit for either a read or a

// write, and return true if the read/write

// worked, or false otherwise. When reading a

// it will fail if the data being read has been

// read before. When writing it will fail if

// the buffer space overflowed. In either case

// the read/write needs to be done again.

//

// Upon exit the clock and cs are low.

//

bool VDIPSPI::\_statusbit()

{

unsigned char statusbit;

//

// the next two lines were previously reversed - it fixed the problem

// putting them this way! Which makes sense if you study "recv()".

//

statusbit = digitalRead(\_misoPin);

SPI\_CLK\_UP;

SPI\_CLK\_DOWN;

digitalWrite(\_csPin,LOW);

SPI\_CLK\_UP; // give the VDIP an extra clock pulse to consume CS

SPI\_CLK\_DOWN;

return(statusbit != HIGH); // HIGH on the last bit indicates old data or bad write

}

//

// public methods

//

//

// send() - sends data on the SPI buss to the VDIP. Returns

// true if the write was accepted, or false otherwise.

// If false, that means that the VDIP buffer was full

// and the write needs to be attempted again.

//

bool VDIPSPI::send(unsigned char data, bool statusByte /\* = false \*/)

{

\_header(false,statusByte); // set up for a write

for(int i=8; i--; ) {

digitalWrite(\_mosiPin,(data&0x80)?HIGH:LOW);

SPI\_CLK\_UP;

SPI\_CLK\_DOWN;

data <<=1;

}

return(\_statusbit());

}

//

// recv() - the receive routine that grabs a byte from

// either status or data. The routine returns true if

// the read was of new data, false otherwise. The given

// data pointer is filled-in in either case.

//

bool VDIPSPI::recv(unsigned char \*data, bool statusByte /\* = false \*/)

{

\*data = 0;

\_header(true,statusByte); // set up for a read

for( int i = 8; i--; ) {

\*data <<= 1;

\*data |= (digitalRead(\_misoPin) == HIGH)?0x01:0x00;

SPI\_CLK\_UP;

SPI\_CLK\_DOWN;

}

return(\_statusbit());

}

//bool VDIPSPI::oldrecv(unsigned char \*data, bool statusByte /\* = false \*/)

//{

// \*data = 0;

// \_header(true,statusByte); // set up for a read

//

// for( int i = 8; i--; ) {

// \*data <<= 1;

// SPI\_CLK\_UP;

// \*data |= (digitalRead(\_misoPin) == HIGH)?0x01:0x00;

// SPI\_CLK\_DOWN;

// }

//

// return(\_statusbit());

//}

## VDIPlow.cpp

//

// VDIPlow.cpp

//

// This is the low-level VDIP code. It was moved from here to keep make the replacement

// of the VDIP processing code easier. This includes the class initialization and such.

//

#include <Arduino.h>

#include "VDIPSPI.h"

#include "VDIP.h"

#define RESET\_DELAY\_SECS 2

//

// constructor

//

VDIP::VDIP(uint8\_t clockPin, uint8\_t mosiPin, uint8\_t misoPin, uint8\_t csPin, uint8\_t resetPin) :

VDIPSPI(clockPin,mosiPin,misoPin,csPin),

\_resetPin(resetPin),

\_resetDelay(false)

{

digitalWrite(\_resetPin,HIGH); // low is reset, done before shifting to output mode

pinMode(\_resetPin,OUTPUT);

ports[0].port = -1; // both ports start out as not initialized

ports[1].port = -1;

// we do a reset whenever the VDIP object is initialized - since there is only one

// and it is only initialized upon power-up, then we're ok with this

reset();

}

//

// readBytes() - given a byte count, do a read of the number of bytes

// requested from the monitor. The array better be big enough...

// this has a timeout that will cause a return if hit. If timeout

// is given as zero, THIS IS A BLOCKING CALL - which should never

// be done. The timeout is given in milliseconds.

//

// RETURNS true if data was received, false if the timeout was hit.

//

bool VDIP::readBytes(int count, char \*retarray, int timeout)

{

long target = millis() + timeout;

for (int i=0; i < count; i++) {

while(!recv((unsigned char \*)retarray+i)) {

if((timeout != 0) && (millis() > target)) {

return(false);

}

}

}

return(true);

}

//

// sendBytes() - given a byte count - send those bytes to the monitor. It is

// assumed that the monitor is in a good place to receive the

// bytes, or this routine may timeout. If timeout is given as

// zero, this routine WILL BLOCK - which is bad. The timeout

// is given in milliseconds.

//

// RETURNS true if data was sent, false if the timeout was hit.

//

bool VDIP::sendBytes(int count, const char \*buffer, int timeout)

{

long target = millis() + timeout;

for (int i=0; i < count; i++) {

while(!send(buffer[i],false)) {

if((timeout != 0) && (millis() > target)) {

return(false);

}

}

}

}

//

// flush() - Since it is possible that async data can come from the

// monitor before or after executing a command, this routine

// checks for and consumes all data sitting in the monitor buffer.

//

// This routine assumes that if there is nothing in the buffer

// when it is called, then nothing will be in the buffer by

// the time this routine finishes. sure, that means that there

// is a race condition.

//

// If there IS something in the buffer, then the routine will wait

// for the configured number of milliseconds for the buffer to clear,

// while consuming the data in the meantime. Then it will try again

// for a clear buffer, until the flush is completely "flushed."

//

// Soooo, this routine may actually take more time than the "timeout".

//

void VDIP::flush(int timeout)

{

long target = millis() + timeout;

unsigned char buffer[1];

while(recv(buffer)) {

// if was something in the buffer, assume there is more and consume

// it until the timeout clears

while(millis() < target) {

(void)recv(buffer);

}

// loop around to check again

}

}

//

// sync() - "sync-up" with the VDIP board prior to issuing a command. This makes

// sure that the board is ready for a command before interacting with it.

// Granted there is still the possibility that some darned async data

// could fill the VDIP buffer before the command after this sync. But

// there isn't much we can do about that...alas.

//

// BIG NOTE - this routine doesn't have a timeout. The point here is

// that if this routine can't get in sync with the board, then we have

// insurmountable problems! In other words, major bummer.

//

// RETURNS:

// false - not yet in sync, need to call it again to get in sync (and

// go ahead and do other stuff in the mean-time, like reset

// something else or blink an LED)

// true - we're in sync with the VDIP - go ahead and issue a command

//

// ASSUMPTIONS:

// - reset() will set \_resetDelay and \_resetTarget appropriately

//

bool VDIP::sync()

{

// note that we're using the "E" (echo) to sync - this is the same in

// both short and extended command set - so this can be used during reset

const char sBuffer[2] = { 'E', '\r' };

// check to see if we are still in reset delay

if(\_resetDelay) {

if(millis() < \_resetTarget) {

flush(); // just keep flushing the VDIP output buffer

return(false); // still in reset delay

}

// we just got out of reset delay, time to initialize the VDIP.

// To do this, get in sync, then execute the init() command(s).

\_resetDelay = false;

sync(); // RECURSIVE!

init();

// we assume (rightly/wrongly) that we are in sync after this

// so just return - no need to do it again below

return(true);

}

// all reset-style processing is done above - at this point we're just in the

// middle of operations looking to ensure that we're sync'd with the VDIP

// the "new" sync is blocking style, that is, if the VDIP misbehavesw, then

// it will never return...but if that is the case, we have bigger problems anyway.

// Below, we send out the sync/echo command and simply wait for its return.

// This has the same effect as a flush, by-the-way.

sendBytes(2,sBuffer,0); // a blocking send of the sync/echo string

while(true) {

unsigned char RChar, prevRChar;

if(recv(&RChar)) {

if(RChar == '\r' && prevRChar == 'E') {

// we're sync'd up here!

return(true);

}

prevRChar = RChar;

}

}

// we never get here

}

//

// reset() - reset the VDIP. After calling this routine, nothing can really be done

// with the VDIP until it is out of reset delay. The other low-level commands,

// though, will take care of this for us.

//

// In essence, there is a four-part reset strategy. This call is the first part:

// resetting the device. The second part is to wait for an appropriate amount

// of time for the device to get itself reset. Part 3 - get in sync with the

// the device, and Part 4 - set the appropriate operational parameters.

//

// The sync() command processes steps 2,3, and 4. This routine just kicks it

// off.

//

void VDIP::reset()

{

\_resetDelay = true; // we are beginning the reset delay

digitalWrite(\_resetPin,LOW); // low is reset

// note that there used to be a delay in here to hold the reset low for awhile

// because the VDIP doc didn't say anything about how the reset pin worked (falling edge

// for example). However, the delay doesn't work when the reset is being called as

// part of the object initialization because the delay() timer isn't yet up and running.

// Soooo, we just count on the VDIP watching for the falling edge.

digitalWrite(\_resetPin,HIGH);

\_resetTarget = millis() + (RESET\_DELAY\_SECS\*1000);

}

//

// init() - initializes the VDIP. This is only done after a reset/sync. Anything that

// needs to be done to get the VDIP ready for operations is done here.

// Given the importance of init() it blocks. That is, if it doesn't return

// we're in big trouble anyway.

//

// This routine assumes that we're in sync.

//

void VDIP::init()

{

// the only thing we need to do here is go into shortened command mode

// we are, by default, in binary value mode already upon reset so we don't

// need to deal with that. Note, by-the-way, that this command works in

// either extended or shortened command mode.

const char sBuffer[4] = { 'S', 'C', 'S', '\r' };

unsigned char rBuffer[1];

(void)sendBytes(4,sBuffer,0); // just blocks waiting to send

while(!recv(rBuffer)) { // spin around waiting for first return byte

}

while(rBuffer[0] != '\r') { // will potentially block waiting for '\r'

recv(rBuffer);

}

}

## blinky.cpp

//

// blinky.cpp

//

// this file implements blinky lights! To use it:

// 0 - include blinky.h

// 1 - define a blinky object by giving it a digital pin where there is an LED

// 2 - call the appropriate funtion to set the state of the light: on, off, fast, slow

// 3 - somewhere in your "loop()" call the blinky update function to update the

// status of the blinky lights. If you use multiple blinky's, you need to call

// the update for each one.

// NOTES:

// - blinky keeps the lights in sync with the system clock, so they will blink in

// unison normally. If you don't like this, call "phaseShift" and it will make

// the blinky "half-off" of the normal blinky pattern.

// So if you have two blinky's and one is shifted, then they

// will blink exactly opposite of each other at the same speed.

// - the phaseShift is only useful for blinking blinky's - "on" and "off" are not

// affected.

#include <Arduino.h>

#include "blinky.h"

//

// NOTE that the \_shift has been turned on during init - this makes it so that the booting

// power LED starts off ON.

//

blinky::blinky(int pin) : \_pin(pin), \_state(BLINKY\_OFF), \_shift(true)

{

pinMode(\_pin,OUTPUT);

}

void blinky::on()

{

\_state = BLINKY\_ON;

update();

}

void blinky::off()

{

\_state = BLINKY\_OFF;

update();

}

void blinky::fast()

{

\_state = BLINKY\_FAST;

update();

}

void blinky::slow()

{

\_state = BLINKY\_SLOW;

update();

}

//

// update() - this is the workhorse routine of blinky. It checks to see if

// this LED should be on or off based upon the system clock

// and current settings of the blinky.

//

void blinky::update()

{

// this routine uses the millis() routine - which returns the number of milliseconds

// that the Arduino has been running since it was booted. This number won't roll over

// unless you leave it running for 50 days :-) but that really doesn't matter for us.

// the way blinky update works is that is attaches some meaning to the different

// bits in the millis() reading. Since it is counting milliseconds, we want our

// blinking to be:

// BLINKY\_SLOW - 512 millis "bit"

// BLINKY\_FAST - 128 millis "bit"

// By checking to see if the given bit is on/off, the LEDs can be set to blink

int output;

unsigned long now = millis();

switch(\_state) {

case BLINKY\_ON: output = HIGH; break;

case BLINKY\_OFF: output = LOW; break;

case BLINKY\_SLOW: output = (now & 512)?(\_shift)?LOW:HIGH:(\_shift)?HIGH:LOW; break;

case BLINKY\_FAST: output = (now & 128)?(\_shift)?LOW:HIGH:(\_shift)?HIGH:LOW; break;

}

digitalWrite(\_pin,output);

}

void blinky::phaseShift()

{

\_shift = ! \_shift;

// as with slow/fast the LED is only affect when update is run

}

## button.cpp

//this is the button class

#include <Arduino.h>

#include "button.h"

button::button (int pin) : \_pin(pin), wasPressed(false)

{

pinMode(\_pin, INPUT);

}

bool button::isPressed()

{

wasPressed = digitalRead(\_pin);

return wasPressed;

}

bool button::hasChanged()

{

int current = digitalRead(\_pin);

if (current != wasPressed){

wasPressed = current;

return true;

}

wasPressed = current;

return false;

}

## nxt.cpp

#include <arduino.h>

//

// nxt.cpp

//

// NXT communication routines. This file implements all NXT communication routines.

// It supports both USB and BT communication.

//

// Note there are "normal" commands as well as a set of extended "direct commands."

// These routines don't worry the user with the difference between them.

//

#include "vdip.h"

#include "nxt.h"

typedef unsigned char byte;

void dumpDataHex(char \*buffer, int count)

{

int i = 0, value;

for(;count > 0; count--, i++, buffer++) {

value = (unsigned char)\*buffer;

Serial.print("0x");

if( value< 16) {

Serial.print("0");

}

Serial.print(value,HEX);

Serial.print(" ");

if( i%8 == 7) {

Serial.println("");

}

}

}

char hexConverter[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};

//

// nxtQueryDevice() - USB - queries the NXT device for it's settings by issuing a GET DEVICE INFO

// command. This should only be done via USB (for now). This routine returns

// TRUE (non-zero) if the command worked or FALSE (zero) if it didn't. If it

// did work, the name buffer is filled in with the name of the NXT, the btaddress

// buffer is filled in, and the freeMemory long is filled in with the amount

// of free memory on the NXT.

//

// Since only one NXT is supposed to be connected at any one time, this routine

// uses simple static buffers to return the name and btAddress. The next call

// will wipe them out.

//

bool nxtQueryDevice(VDIP \*vdip, int usbDev, char \*\*name, char \*\*btAddress, long \*freeMemory)

{

// assumes we are connected, otherwise this routine shouldn't be called

char cbuf[50]; // enough for return data from NXT (plus slop)

static char namebuf[15]; // 14 chars plus \0 termination

static char btAddressbuf[13]; // 6 bytes of BT address (ignores the last one)

\*name = namebuf;

\*btAddress = btAddressbuf;

\*freeMemory = 0;

vdip->cmd(VDIP\_SC,NULL,100,usbDev); // set the current VDIP port to the NXT

cbuf[0] = NXT\_SYS\_CMD;

cbuf[1] = NXT\_GET\_DEV\_INFO;

vdip->cmd(VDIP\_DSD,cbuf,100,2); // send the command

delay(1000); // half-second delay for return of command

int r = vdip->cmd(VDIP\_DRD,cbuf,100);

if(r != 33) {

Serial.println("yikes, didn't get values back");

return(0);

} else {

Serial.println("yeah! got 33 back");

dumpDataHex(cbuf,33);

for (int i = 0; i < 15; i++){

namebuf[i] = cbuf[3 + i];

}

int j = 0;

for (int i = 18; i < 24; i++){

//the last byte is always zero (sorta like a null terminator)

btAddressbuf[j++] = hexConverter[(cbuf[i]>>4)&0x0F];

btAddressbuf[j++] = hexConverter[cbuf[i]&0x0F];

}

btAddressbuf[j] = '\0';

long m = 1;

for (int i = 29; i < 33; i++, m \*= 256){

\*freeMemory += cbuf[i]\*m;

}

return(1);

}

}

//

// nxtGamepadUSBTranslate() - translate a single USB gamepad reading into the fields that would

// part of a RobotC joystick structure as serialized to be transmitted

// by BT. This routine is COMPLETELY dependent upon the way in which

// the RobotC "Joystick.h" is coded, and will need to change if it does.

// - the output array must be 7 bytes

// - the input array must be 6 bytes

// - NOTE that this routine doesn't take into account the MODE SWITCH

// which is byte #7 in the USB data.

//

void nxtGamepadUSBTranslate(byte \*usbdata, byte \*output)

{

// output[0] = usbdata[0]; // joystick 1 (left) X axis

// output[1] = usbdata[1]; // joystick 1 (left) Y axis

// output[2] = usbdata[2]; // joystick 2 (right) X axis

// output[3] = usbdata[3]; // joystick 2 (right) Y axis

output[0] = usbdata[0]-128; // joystick 1 (left) X axis

output[1] = usbdata[1]-128; // joystick 1 (left) Y axis

output[2] = usbdata[2]-128; // joystick 2 (right) X axis

output[3] = usbdata[3]-128; // joystick 2 (right) Y axis

output[4] = (usbdata[4]&0xf0)>>4 | (usbdata[5]&0x0f)<<4; // buttons 1-8

output[5] = (usbdata[5]&0xf0)>>4; // buttons 9-12

output[6] = usbdata[4] & 0x0f; // tophat

if (output[6] & 0x08) { // NXT expects -1 if no tophat pressed

output[6] = 0xff;

}

// we don't send the last byte that is mode and controller data

}

//

// nxtJoystickTranslate() - given the joystick output from a USB joystick

// report, convert it the joystick structure as

// defined in RobotC - which, therefore, is used

// by all of the RobotC-style programs.

// Note that the USB joystick report is 8 bytes,

// and the RobotC joystick structure is 18 bytes

// with the last being a NULL.

// Note, too, that you need to specify the setting

// of "wait for start" and UserMode as arguments

// to this routine.

// If one/both of the joysticks aren't connected, this

// routine should be sent "blank" readings for that

// joystick.

//

int nxtJoystickTranslate(byte \*output, byte UserMode, byte StopPgm, byte \*USBJoystick1, byte \*USBJoystick2)

{

output[0] = 0; // this isn't actually used in the Joystick.c code

output[1] = UserMode; // normally set to zero

output[2] = StopPgm; // false to turn OFF wait for start

nxtGamepadUSBTranslate(USBJoystick1,&output[3]); // used to have +1 when the \r wasn't being read

nxtGamepadUSBTranslate(USBJoystick2,&output[10]);

output[17] = 0;

return(18);

}

// nxtDCMessageWrite() - do the Direct Command for MessageWrite

// as well as wrap the message for transport.

// Fills in the output buffer and returns the

// size of the resultant message.

// NOTE that the message MUST have a '\0' at

// the end to be legal, so this routine

// enforces that.

//

int nxtDCMessagePackage(byte \*output, byte \*input, int size, int mbox)

{

// note that in the code below only the LSB of size is sent - that's

// because we will never send a message that needs the MSB of size

input[size-1] = '\0'; // enforce the NULL termination

output[0] = size + 4; // size does NOT include this byte itself

output[1] = 0x00; // this is the MSB of size - always zero

output[2] = 0x80; // direct command with no response

output[3] = 0x09; // the direct command for MessageWrite

output[4] = mbox; // should normally use zero here

output[5] = size;

memcpy(output+6,input,size);

return(size+6); // includes the size byte

}

//

// nxtMsgCompose() - given two joystick structures and other data, compose a message to

// be sent to the NXT.

//

int nxtMsgCompose(byte \*output, // the output buffer to scribble things to - min 24 bytes

byte UserMode, // the usermode value

byte StopPgm, // the wait-for-start value

byte \*USBJoystick1, // buffers for the two joysticks

byte \*USBJoystick2)

{

byte buffer[20];

int size;

int mbox = 0;

size = nxtJoystickTranslate(buffer, UserMode, StopPgm, USBJoystick1, USBJoystick2);

return(nxtDCMessagePackage(output,buffer,size,mbox));

}

## sound.cpp

#include <Arduino.h>

#include "sound.h"

sound::sound(int pin) :

\_pin(pin)

{

pinMode(\_pin,OUTPUT);

}

void sound::confirm()

{

tone(\_pin,440,50);

delay(50);

tone(\_pin,880,50);

}

void sound::squeep()

{

noTone(\_pin);

for (int i = 880; i < 3520; i += 20) {

tone(\_pin,i);

delay(1);

}

for (int i = 3520; i > 880; i -= 20) {

tone(\_pin,i);

delay(1);

}

noTone(\_pin);

}

# NameReset.ino

#include <Arduino.h>

#include <EEPROM.h>

#include "ChapRName.h"

ChapRName myName;

void setup()

{

Serial.begin(38400);

delay(100);

Serial.println("");

Serial.println("The current name of your ChapR is: ");

Serial.println(myName.get());

Serial.println("Enter the name for your ChapR: ");

int index = 0;

char buffer[15];

while (true){

if (Serial.available() > 0){

byte val = Serial.read();

if (val != '\r' && index != 14){

buffer[index] = val;

} else {

buffer[index] = '\0';

break;

}

index++;

}

}

Serial.print("Your ChapR is now named: ");

Serial.println(buffer);

myName.setFromString(buffer);

}

void loop()

{

}

## ChapRName.cpp

#include <Arduino.h>

#include <EEPROM.h>

#include "ChapRName.h"

ChapRName::ChapRName()

{

}

char \*ChapRName::read()

{

static char buffer[15];

for (int i = 0; i < 15; i++){

buffer[i] = EEPROM.read(i);

}

return (buffer);

}

void ChapRName::write(char \*name)

{

int i = 0;

do {

EEPROM.write(i, name[i]);

i++;

} while (name[i] != '\0');

while (i < 15){

EEPROM.write(i, '\0');

i++;

}

}

char \*ChapRName::get()

{

return (read());

}

void ChapRName::setFromString(char \*name)

{

write(name);

}

void ChapRName::setFromConsole()

{

Serial.println("Hit space then enter if you want to rename your ChapR.");

char buffer[100];

String name = "";

static int index = 0;

bool enteringName = false;

if (Serial.available() > 0){

int input = Serial.read();

buffer[index] = input;

if (buffer[index - 1] == ' '){

Serial.write("Enter your new name (must be 1 - 20 characters without spaces): ");

index = 0;

enteringName = true;

}

else if (buffer[index] == '\r'){

for (int i = 0; i <= index; i++){

name = name + buffer[i];

}

Serial.println(name);

Serial.println("Congrats! Your ChapR has been named " + name);

}

if (!enteringName){

index++;

}

}

}

bool ChapRName::setFromFlashDrive()

{

}