

```

/**
 * @file sensors.c
 * @brief Implementation of hardware abstraction for sensors
 * Copyright (C) 2017 Ethan Wells
 *
 * This program is free software: you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the Free
 * Software Foundation, either version 3 of the License, or (at your option) any
 * later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 * FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program. If not, see <https://www.gnu.org/licenses/>
 */

#include "../include/sensors.h"
#include <math.h>

static inline float defaultRecalc(int n) {
    return n;
} /* defaultRecalc */

unsigned char imeNum = 0;
bool isImeInit = false;

int readSensorValue(Sensor *s) {
    switch (s->_type) {
        case Digital:
            return digitalRead(s->port);

        case Analog:
            return (s->calibrate) ?
                analogReadCalibrated(s->port) :
                analogRead(s->port);

        case AnalogHR:
            return analogReadCalibratedHR(s->port);

        case Sonic:
            return ultrasonicGet(s->_pros);

        case Quad:

```

```

        return encoderGet(s->_pros);

    case Gyroscope:
        return gyroGet(s->_pros);

    default:
        return 0;
} /* switch */
} /* readValue */

void sensorRefresh(Sensor *s) {
    if (!s) {
        return;
    }

    if (s->child) {
        sensorRefresh(s->child);
    }

    if (!mutexTake(s->_mutex, 5)) {
        #ifdef DEBUG_MODE
            print("= Cannot take mutex =");
        #endif
        return;
    }

    int val;
    if (s->_type == IME)
        imeGet(s->port, &val);
    else
        val = (s->_type == Digital) ? readSensorValue(s) :
                                         (readSensorValue(s) - s->zero);

    if (s->inverted) {
        if (s->_type == Digital) {
            val = !val;
        } else {
            val = -val;
        }
    }

    if (s->recalc) {
        val = round(s->recalc(val));
    }

    s->value = val;
}

```

```

s->averageVal = s->child ? ((s->value + s->child->averageVal) / 2) : s->value;

// int lastVel = s->velocity;

if (s->_type == IME && imeGetVelocity(s->port, &s->velocity))
    s->_lastUpdate = millis();
else if (s->_type != IME) {
    s->velocity = ((float)(val - s->_lastValue) /

    s->_lastValue = val;
    s->_lastUpdate = millis();
} else
    s->velocity = 0;

s->averageVel = s->child ? ((s->velocity + s->child->averageVel) / 2)
                        : s->velocity;

mutexGive(s->_mutex);
} /* sensorRefresh */

void sensorReset(Sensor *s) {
    if (s->child) {
        sensorReset(s->child);
    }

    switch (s->_type) {
        case Gyroscope:
            gyroReset(s->_pros);
            s->zero = 0;
            break;

        case Quad:
            encoderReset(s->_pros);
            s->zero = 0;
            break;

        case IME:
            imeReset(s->port);
            break;

        default:
            s->zero = readSensorValue(s);
            break;
    } /* switch */
} /* sensorReset */

```

```

Sensor newSensor(SensorType    type,
                 unsigned char  port,
                 bool           inverted,
                 unsigned short calibrate) {
    if (port < 1) {
        port = 1;
    }

    Sensor s = {
        ._type      = type,
        .recalc     = &defaultRecalc,
        .port       = port,
        .inverted   = inverted,
        .calibrate  = calibrate,
        ._mutex     = mutexCreate(),
    };

    switch (type) {
        case Digital:
            pinMode(port, INPUT);
            break;

        case Analog:
            if (calibrate) {
                analogCalibrate(port);
            }
            break;

        case AnalogHR:
            analogCalibrate(port);
            break;

        case Sonic:
            s._pros = ultrasonicInit(port, (char)calibrate);
            break;

        case Quad:
            s._pros = encoderInit(port, (char)calibrate, false);
            break;

        case Gyroscope:
            s._pros = gyroInit(port, calibrate);
            break;

        default:
            break;
    }
}

```

```

        } /* switch */

        sensorReset(&s);
        return s;
    } /* newsensor */

    Sensor newDigital(unsigned char port, bool inverted) {
        return newSensor(Digital, port, inverted, false);
    } /* newDigital */

    Sensor newSonic(unsigned char orange, unsigned char yellow) {
        return newSensor(Sonic, orange, false, yellow);
    } /* newSonic */

    Sensor newQuad(unsigned char top, unsigned char bottom, bool inverted) {
        return newSensor(Quad, top, inverted, bottom);
    } /* newQuad */

    Sensor newAnalog(unsigned char port, bool calibrate) {
        return newSensor(Analog, port, false, calibrate);
    } /* newAnalog */

    Sensor newAnalogHR(unsigned char port) {
        return newSensor(AnalogHR, port, false, true);
    } /* newAnalogHR */

    Sensor newGyro(unsigned char port, bool inverted, int calibration) {
        return newSensor(Gyroscope, port, inverted, calibration);
    } /* newGyro */

    Sensor newIME(unsigned char num, bool inverted) {
        return newSensor(IME, num, inverted, 0);
    } /* newIME */

```