This program is to be uploaded onto the Arduino system for the use of relaying sensor information to the Isadora computer program. It works by first reading the sensor, determining if a change in the sensor state exists; then, if a change in state is detected, it reports this to Isadora for further processing.

USER GUIDE

SETUP

User must wire the signal wire of each sensor to the analogue or the digital inputs on the Arduino board. Binary sensors must be wired to Digital In; analogue sensors, Analogue In.

Once wired, you must alter the settings in the program in accordance with your wiring, the sensor type, and any special instructions you wish to give the program. Open config/config.ino and scroll to the bottom. The program takes the sensor format in the following format:

```
const int pin_in[]{
    {[pin 1], [sensor type], [special instructions]},
    {[pin 2], [sensor type], [special instructions]}
};
```

Note how all the pin settings are enclosed with curly brackets, and the settings for each individual pin are also enclosed with curly brackets. Additionally, note how the individual pin settings, and the groups of pin settings are separated by a comma. Finally, note that the last setting and pin are not followed by a comma.

In the event that you have no special instruction to pass to the program, you should inform the program that the special instruction is NORMAL as shown in the example below:

```
const int pin_in[]{
    {[pin 1], [sensor type], NORMAL},
    {[pin 2], [sensor type], NORMAL}
};
```

Thus, a sample configuration for 4 normal sensors, and an ultrasonic ranger might appear as follows:

```
const int pin_in[]{
    {1, NORMAL, NORMAL},
    {2, NORMAL, NORMAL},
    {3, NORMAL, NORMAL},
    {4, NORMAL, NORMAL},
    {A0, ULTRASONIC_RANGER_2_0, 1}
};
```

Notice the special instruction for pin A0. In this case, it tells the program that it wants the distance from the Ultrasonic Ranger in cm. To discover which sensors require an explicit sensor type, and what the special instruction codes are, read the sensor type defines which are usually immediately preceding the USER SETTINGS — the comments will inform you of any special considerations and instructions.

Number of input pins limited by hardware.

Once written, the program should be compiled and programmed into the Arduino. A USB cable should be used to permit serial communication between the Arduino and a receiving computer.

USAGE

When the controller detects a change in sensor state, it transmits a serial message in the following format:

```
p[pin]d[1 = digital, 0 = analogue]r[reading]
```

`\n` signifies the end of a message. Analogue sensors tend to have high output rates as analogue values change regularly.

OPERATIONS

Modular Programs

Large programs often have many functions and the code for them can get quite long. To get around this issue, a single file containing code can be separated into different *modules* — essentially files — to make organization and referencing easier.

For the Arduino IDE, the module that the IDE will read first is the module which has the same name as the directory which contains it; thus, in a directory named `example/`, the Arduino IDE will look for, and run `example.ino` before other files.

As the sensor control module is quite large, it has been decided into modules, the large and/or important functions have their own modules whilst smaller, less significant functions have been grouped into config/general-system-control.ino and for simplicity, all special instructions for special sensors have been grouped into config/special-instructions.ino.

Modules

This section describes the modules of this program, list the functions within, and describe their general purpose. Owing to the constraints of the Arduino IDE, they are located in `config/`

config.ino

Library Includes

This program requires the `math.h` and `Ultrasonic.h` libraries. `math.h` allows for the calculations required by `Ultrasonic.h`.

Sensor Type Defines

As some sensors require special instructions, libraries, calculations, etc. special sensors may be declared here. The defines essentially tell the preprocessor to replace a string with something else. Defines are usually made in all caps, and for this program, they will consist of the sensor's full name with version. All spaces, dashes, or reserved characters will be replaced with underscores `_`.

An example of this may be found here:

```
#define ULTRASONIC_RANGER_2_0 1
```

Here, we declare that the string `ULTRASONIC_RANGER_2_0` will be replaced with `1`. Thus, before the compiler compiles the program, all instances of `ULTRASONIC_RANGER_2_0` will be replaced with `1`.

Special instructions for special sensor types should be defined underneath the declaration as a comment. Special instructions must be encoded as an integer. For more information, consult the tutorial on how to make a special instruction.

For information on how define works, consult cprogramming.com

Configuration Array

As mentioned earlier, this array stores the settings for each individual pin. You may have noticed that it is declared as a `const int`. You may have noticed that the configurations are however, entered as strings; however, as mentioned above, the C preprocessor replaces the string with a number as the program is being compiled; thus, all settings become numbers. For information on how to set-up the configuration, see setup

loop.ino

`loop.ino` contains `void loop()` and functions as the main working loop for the program. It's operations are described in OPERATIONS.

setup.ino

This module contains those global variables which were too inconvenient to declare in a different function, as well as setup-code for the program.

1. `const int number_of_pins` contains the number of pins
2. `old_sensor_value[]` See comment in module

3. `int i` The master clock as it where. It keeps track of iterations to ensure that the program properly cycles through each configured pin as declared in `config.ino`.

setup()

The Arduino standard setup function. It initializes the serial output and sets all configured pins as an input.

set_as_input()

Sets the pins declared in `config.ino` as inputs.

read-sensor.ino

`read-sensor.ino` contains only `int read_sensor()`. Read sensor takes the following arguments:

1. `pin` — The Pin number
2. `sensor_type` — the integer representing the sensor's type as defined in the sensor type defines.
3. `special_inst` — the corresponding special instruction as defined in the comments of the sensor type defines.
4. `digital` — Whether the pin received is digital or analogue. 1 for digital 0 for analogue.

It reads the sensors defined in the `switch` and having read the appropriate sensor, returns the sensor value to the calling function.

transmission.ino

Contains function `send_sensor`. The function takes the pin number, the sensor's value, and whether the sensor is digital or analogue and uses the serial connection to inform a computer as to the sensor's value. The format is defined in usage

special-instructions.ino

Contains the functions required to operate special sensors. All new special sensors must have a new function written for them, and should simply be added to the module in alphabetical order, by the name of the function. All function names ought to be made in lowercase, and must be of the `int` data type. The return value of the function should be the sensor's reading, whatever that may be.

Refer to creating a new special sensor for a basic tutorial as to the requirements of the addition of a new, special sensor.

general-system-control.ino

This module contains little functions added just to make the code more readable, or to achieve minor, but useful tasks. It contains:

1. `int change_exists()`
2. `int at_array_end()`
3. `int is_digital()`

Refer to the file for more information, descriptions are in the comments.

OPERATIONS

To understand the program, first read `loop()` located at config/loop.ino.
Loop reads pins defined in `pin_in[]` one at a time. It does this using `i` to select the array element through `pin_in[i]`. At the end of the loop, i is incremented though `i++`. When all elements in the array have been read, `i` is reset.

Each iteration — i.e. cycle — of loop works as follows.

1. the current pin is tested by `is_digital()` to see if it is a digital or analogue pin.
2. The sensor is then read by `read_sensor()` which reads the sensor fed to it, and stores it in the variable `sensor_value`.
3. Then, the program evaluates if there has been a change between the old sensor reading and the new sensor reading using `change_exists()`.
    1. If there is change, a message is sent though serial out using `send_sensor()`.
4. After, `i` is incremented. If `i` is greater than the total number of elements in the array (if it would cause one to over-read the array), `i` is reset to 0 and the program tests each sensor again.

Programmer's Guide

This section will guide you though the creation of new parts of this program.

New Special Sensor

Prerequisites

Before you add a new special sensor type, it is recommended that you obtain a reasonable understanding of Arduino's custom language, C, C++, or some other C-like language. I also recommend you look into, and understand the execution code for your special sensor, have an understanding of the various settings the sensor may come with, and how it may programmatically be set. Finally, it is strongly recommended that you try writing code for the sensor in a simple, stand-alone program to experiment with it's configurations and to develop an understanding with the sensor before you add it to this, much more complicated program.

Library Includes

If your sensor requires any sensor specific preprocessor includes, declare them with all the other `#includes` at the top of config.ino. Do-not run other initialisation codes here if it is possible. For example:

```
...

#include <math.h>
#include "Ultrasonic.h"

...
```

Assign a Keyword Define

Open config.ino Scroll down to the sensor type defines and comment out room for your sensor to the bottom of the list in a format consistent with the present sensor. Then, add the following line within the area that you have commented:

```
#define [SENSOR_NAME] [INTEGER]
```

Where `[SENSOR_NAME]` is the model of your sensor as written by the manufacturer on the sensor. Take care that all spaces and special characters are replaced with underscores `_`, and that it is written in all capital letters.

Where `[INTEGER]` is a number which may be used for the the identification of the sensor (done so that all systems using it can be an easy to handle int; aren't data-types fun?). Simply take the highest identification number in the sensor-type defines `++`.

Add any special instructions (an integer that can alter the behaviour of the function that will be written below) below the defines as comments. Example:

```
/*********************** ULTRASONIC_RANGER_2_0 ******************************/

#define ULTRASONIC_RANGER_2_0 1
/*
Ranger uses the special instruction for determining if the function should
return the distance to target in cm or inches.

0 == cm
1 == inches

*/
```

```
/********************** End of ULTRASONIC_RANGER_2_0 **********************/
```

Write an Execution Function

Open config/special-instructions.ino and begin insert your function into the module. Keep the functions in alphabetical order. You may include more than one function if you believe that it is necessary. Example:

```
...
int read_ultrasonic_ranger_2_0(int pin, int special_inst)
{
    Ultrasonic ultrasonic(pin); // call OEM library function

    //delay(250); // no idea why, I'm adapting it from the OEM example
    if(special_inst) // == 1, inches
    {
        return ultrasonic.MeasureInInches();
    }

    else // cm
    {
        return ultrasonic.MeasureInCentimeters();
    }
}
...
```

Design Requirements

Preconditions

Your program may receive:

1. `int pin` — The pin number of the sensor in question,
2. `int sensor_type` — This is the integer that represents the various sensor types.
3. `int special_instruction` — This is any special instruction you wish to encode as an integer. It is preprogrammed in `in_pins[]` and may be used to control your program however you wish.
4. `int digital` — If the pin is digital, 1, else 0.

Post-Condition

Your program will return an integer which represents the sensor value to be transmitted to a show-control device.

Add Function Call to Main Switch

Add your function to the main switch, preferably in alphabetical order. Be sure to keep the `default` at the bottom. Use the sensor-type define as the `case` test, and return the function call to the sensor read function. Don't forget to tell your function what pin to read and to terminate the case with a break. Example:

```
...
case ULTRASONIC_RANGER_2_0:
    return read_ultrasonic_ranger_2_0(pin, special_inst);
    break;
...
```

```
/******************************************************************************
************************ LIBRARY INCLUDES *************************************
******************************************************************************/

#include <math.h>
#include "Ultrasonic.h"

/******************************************************************************
************************ End of LIBRARY INCLUDES ******************************
******************************************************************************/

/******************************************************************************
************************ sensor type defines **********************************
******************************************************************************/

/*********************** NORMAL **********************************************/
// Sensor that doesn't require a library, special instruction

#define NORMAL 0

/*********************** END OF NORMAL ***************************************/

/*********************** ULTRASONIC_RANGER_2_0 *******************************/

#define ULTRASONIC_RANGER_2_0 1
/*
Ranger uses the special instruction for determining if the function should
return the distance to target in cm or inches.

0 == cm
1 == inches

*/

/*********************** End of ULTRASONIC_RANGER_2_0 ************************/

/******************************************************************************
************************ End of sensor type defines ***************************
******************************************************************************/

/*VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV USER SETTINGS HERE VVVVVVVVVVVVVVVVVVVVVVVVVVV
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV*/

const int in_pin[][3] = {
 // {pin, sensor type, special instructions}
  {A1, NORMAL, NORMAL},
  {A0, NORMAL, NORMAL},
  {4, NORMAL, NORMAL},
  {5, NORMAL, NORMAL},
  {6, NORMAL, NORMAL}
}; // sets the input pins

/*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ USER SETTINGS ABOVE ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^*/
```

```
/*      This module contains various small functions which have been grouped
together as they don't warrant their own module.  Functions contained here
are:

change_exists()
at_array_end()
is_digital()
*/

int change_exists(int *old_value, int new_value)
/* Function returns 1 if there has been change, 0 for no change it's basically
a bool function */
{
        if(*old_value != new_value)
        {
                *old_value = new_value;
                return 1;
        }

        else
                return 0;
}

/* checks if the array element counter (`i`) has exceeded the number of
elements in the in_pin array, if it has, the function returns 1, else
0.  This is done as `if' statements proceed when the calculations in the
brackets returns a non-zero value; thus, when the function returns 1, it is
true and the code in the `if' statement is executed.  This is also basically a
bool function */
int at_array_end()
{
        // (counts calculates the number of elements
        // in the in_pin array:
        // size of the array / size of one array element
        if(i >= (sizeof(in_pin) / sizeof(in_pin[0])))
                // if i should be reset to 0:
                return 1;
        else
                return 0;
}

/* checks if a pin's digital.  Takes a pin number.  If a pin is digital, the
 * function returns 1; else, 0.
 */
int is_digital(int pin)
{
        if(pin < 13)
                return 1; // pin is digital

        else
                return 0; // pin is analogue
}
```

```
/* This module contains function loop().  It serves as the primary execution
loop for the program */
void loop()
{

        // check if a sensor is digital or analogue.  Pass is_digital the
        // pertainant pin
        // if sensor is digital, 1; else, 0.
        int digital = is_digital(in_pin[i][0]);
// this is reset each time the loop is executed, old
// results are stored in old_sensor_value
        int sensor_value =      read_sensor(
                                /* should these become define macros? */
                                        in_pin[i][0], // pin number
                                        in_pin[i][1], // sensor type
                                        in_pin[i][2], // special instruction
                                        digital // if it's digital
                                );
        // change_exists returns 1 if change detected
        if(change_exists(&old_sensor_value[i], sensor_value)) // == 1
        {/* Address of --^^~~~~~~~~~~~~~~~~~^-- old value
                see note on pointers */
                // if the sensor state has changed, send the sensor value
                send_sensor(in_pin[i][0], sensor_value, digital);
        }

        i++; // move to the next array index

        // Check if we're at the end of the array, if so, reset i
        if(at_array_end()) // == 1
                i = 0;
}
```

```
/* NOTE ON FUNCTIONS

All a function is a block of code which is used to achieve a single tasks.  In
line with the UNIX philosophy, fuctions should only do one thing, and do that
one thing well.  This makes them more portable as they can be used in more
places, and it makes them quicker and easier to debug.

Almost all functions return a value.  What this means may be shown in the
example below:

...
int number = 5;
int number2 = 0;

number2 = add_1(number);

...

In this case, what is implied is that function add_1() will add one to the
variable, and if add_1 is set to return input + 1, the entire string

        add_1(number)

will become 6, which is then assigned to number2.  As such, to go step by
step this is what happens:

        INIT

                number2 = add_1(number);

        STEP 1 -- substitute the variable for the value

                number2 = add_1(5);

        STEP 2 -- add_1 adds 1 to 5 and returns the result

                number2 = 6;

        STEP 3 -- number2 is assigned the value 6

                number2

As you can see, when a function returns, what essentially happens is that the
function call is replaced by the return value before further calculations are
completed.  This can be circumvented somewhat with pointers (see note on
pointers).

As an exception, the only function that doesn't return a value (become the
result of whatever it processed) is a void function.
*/

/* NOTE ON POINTERS ( *, AND & ) Put simply, a pointer is a varable that stores
 * the address of another variable.  All variables are stored in memory, and,
 * like a house, each `register' in memory has an `address'; thus, variable
 * `a' might have the address of `1'.  For Arduino, we can access memory using
 * `*' and `&'.  `&' means address of --- in the example earlier, `&a' means
 * `the address of a'.  `*' means the contents of the address; thus, to
 * continue on the earlier example, `*a' means `the value stored in a'.  In
 * this case, the advantage of using a pointer is that it allows a function to
 * alter the value of a variable in a different function without the cost of
* a global variable --- since global variables can be accessed by the entire
* program, it can make it annoying to find names for variables that are
* only needed within one function.
*/
```

```
/********************* Pin Reading Functions *******************************/

/* Function reads takes the pin number, sensor type, special instructions, and
if the pin is digital and passes that information to a function that will read
the sensors.  The function then returns the value of the sensor. */

int read_sensor(int pin, int sensor_type, int special_inst, int digital)
{
        // check the sensor type
        switch(sensor_type)/* Determine the sensor type before processing */
        {
                case ULTRASONIC_RANGER_2_0:
                        return read_ultrasonic_ranger_2_0(pin, special_inst);
                        break;

                /* sensor type NORMAL is handled by default for simplicity
                        as well as the added benifit that a sensor which
                        theoretically requires a library might still have a
                        chance of working --- albiet badly */
                default:
                        // if it is digital, digital == 1, else == 0.
                        if(digital) // == 1
                        { // digital
                                return digitalRead(pin);
                        }

                        else // analogue
                        {
                                return analogRead(pin);
                        }
                        break;
        }
}
```

```
/* This module contains functions pertaining to setting up the arduino.  The
code is run to initialize the arduino on power-on.  It also contains all
global variables */
/* general constants */

const int number_of_pins = 19;

/* variables */
/* stores the sensor value for the compairison, index value corrisponds with
* the in_pin value */
int old_sensor_value[number_of_pins] = {0};

int i = 0; // used to set the index of in_pin[] in loop

void setup()
{
  Serial.begin(9600);
  set_as_input(); // sets the pins defined in in_pin as inputs
}

/* used to set the array elements in in_pin[] as input pins */
void set_as_input()
{
  // j used as i was taken.  j is the array index
        for(int j = 0; j < (sizeof(in_pin) / sizeof(in_pin[0])) + 1; j++)
        /*                 ^~~~~~~~~~~~~^ | ^~~~~~~~~~~~~~~~^
        The size of the entire array      | The size of a single array element
        Number of array elements == size of array / size of a single element */
        {
                pinMode(in_pin[j][0], INPUT);
        }
}
```

```
/********************** Special Sensor Instructions ************************/

/* This moduel contains all code nessessary to exicute the special instructions
for special sensors.  Add the logic for the instruction into a new function
below.  DOn't forget to include the update in the switch in read_sensor().
Add a case above default.  Default must be at the bottom.
*/

int read_ultrasonic_ranger_2_0(int pin, int special_inst)
{
        Ultrasonic ultrasonic(pin); // call OEM library function

        //delay(250); // no idea why, I'm adapting it from the OEM example
        if(special_inst) // == 1, inches
        {
                return ultrasonic.MeasureInInches();
        }

        else // cm
        {
                return ultrasonic.MeasureInCentimeters();
        }
}

/********************** END of Special Sensor Instructions *****************/
```

```
/* This function talks to a computer via the serial connection*/

/* sends the sensor value to isadora */
void send_sensor(int pin, int sensor_value, int digital)
{
        Serial.print("p");
        Serial.print(pin);
        Serial.print("d");
        Serial.print(digital);  // sends 1 for digital, 0 for analogue
        Serial.print("r");
        Serial.println(sensor_value);
}
```