**Proposal for Sensact Code Changes**

I am reluctant to propose changes to Sensact code. It has already been through one re-write and there is an awful tendency for new software developers to come into a project and change everything to suit themselves – or just for the fun of re-thinking everything.

However, I see problems with the code as it is. The code ought to be the basis for where we would like to be the future. At the moment, however, both versions of the code are not as well organized as they should be. It is not easy to see how many of the future changes being discussed can be incorporated into the current structure.

I see ways the code could be made more portable, by encapsulating and isolating parts of the coding which are specific to the Arduino platform family. This could make a future move to a Pi or Omega 2 easier.

I see a need to insure that there is a clear, logical and simple way to add new sensors and new output devices in the future. Ideally it should be possible to add a new device by adding a module (one .c and one .h) that defines the new device and its function – and add a bit of linkage code in a one well-defined area - without having to pollute the rest of the code with fragments of complex logic.

## Sensact Functionality - in general terms

What is Senact – in very simple terms? Sensact receives input signals, evaluates these signals against a set of triggers and generates outputs. Quite simple really.

The elements of the system are:

Sensor – An input signal.

Sensor Data – The data required to describe the state of the input signal at any one time.

Trigger – Logic for evaluating a sequence of sensor data and triggering actions

Action – An output signal

Action Data – The data required to describe an action and all its parameters.

Controller – Code to control the interactions of the other elements.

Triggers

Sensors

Controller

Actions

Each of these elements could be a separate class, with the ability to extend its capabilities through sub-classing if necessary.

Pseudo-code for the controller can be quite simple. The following code expresses the concept. The actual code will have to deal with a few complications – such as how the lists are traversed.

setup() {

// Do any necessary initializations

Sensors.init();

Triggers.init();

Actions.init();

}

loop() {

SensoreData \*sdata;

ActionData \*adata;

// Get a pointer to a list of data

sdata = Sensors.getData();

// Get a pointer to a list of actions

adata = Triggers.evaluateData(sdata);

// If actions are required – do them

if (adata != null) {

Actions. doActions(adata);

}

delay (awhile)

}

Note that the controller does not need to know any details of how the sensors or outputs are initialized, nor does it need to understand or examine the content of Sensor or Action data objects. New sub-types of Sensor, Trigger, Action, Sensor data and Action data could be added without requiring any changes to the controller code.

## Other Functionality

The controller also needs to be able to pass a stream of data from a configuration utility into the triggers and pass a data stream from the triggers back to the config utility. Note that the controller does not need to know anything about the format or meaning of these data streams. The triggers, on the other hand, should be able to operate on a data stream without knowing anything about where the data came from – via Bluetooth or a serial port or WiFi.

## Data Storage

Trigger information needs to be saved somewhere. Knowledge of how this is done and where the data is stored should be hidden in the Trigger classes and should not be known outside of them.

# The Challenge

There are two different philosophies for building computer related things – chips, operating systems, and other stuff:

1. Create lots of powerful complex commands for doing complex powerful stuff.
2. Create a bunch of simple but elegantly designed functions which can be dynamically assembled to do complex and powerful stuff.

Personally, I prefer the second approach. It is more flexible and results in smaller and simpler code for the sensact. The challenge is to come up with a representation for sensors, triggers and actions which:

* can support all the types of sensors, triggers and actions we can imagine.
* is as simple and flexible as possible.
* allows the parts to be assembled easily

What follows is an attempt to do this. **This needs to be reviewed by the team and modified to fit everyone’s vision of the product.**

## Sensors

Sensors are of two basic types – I’ll call them “analog” and “digital”.

Analog sensors return a continuous range of related values. For example a pressure sensor returns a value from 0 to 1023 depending on how hard it is pressed. The data for these sensors may come from an analog input pin or through an I2C device.

Digital sensors return discrete values which are not necessarily related. An example would be a small keypad attached via an I2C interface which returns a specific value for each key pressed.

There are also sensors which return multiple values – such as a joystick or a gyro. We could treat these as multiple sensors. While the sensor internals would know of a single joystick device, everything external to the sensor code would see two independent sensors: joystick\_x and joystick\_y. This will work as long as the values returned by these sensors never have to be treated as related.

With these assumptions the sensor data (return from Sensors) becomes quite simple.

struct sensor\_datum {

int sensorID;

int value;

}

## Actions

Actions can be one-shot commands – like “send ‘A’ via HID” or “Mouse-up”.

They can also be potentially continuous things like – turn on a relay. We need to define when these turn off – and whether control of this should be in the trigger code (which would send an ON action and an OFF action) or in the Action code (which would use a timer of some sort).

An action may take a number of parameters. The current system allows only one parameter, but there may be cases where more than that is needed: e.g. “sound the buzzer at 400 hz for 500 ms.” – where the numbers 400 and 500 are parameters to a “buzzer” action.

## Triggers

### Trigger organization

Version one of Sensact organizes trigger data as follows: For each sensor there is a list of outputs and a set of trigger conditions.

Sensor1: List of outputs Trigger conditions

Sensor2: List of outputs Trigger conditions

Sensor3: List of outputs Trigger conditions

etc.

In version two for each sensor there two triggers and each trigger defines trigger conditions and actions.

Sensor1: Trigger 1 conditions Action 1

Trigger 2 conditions Action 2

Sensor2: Trigger 1 conditions Action 1

Trigger 2 conditions Action 2

Sensor3: Trigger 1 conditions Action 1

Trigger 2 conditions Action 2

In both cases you start with a list of sensors and attach triggers to them. I think this can be improved.

Sensact is a device which contains a bunch of triggers, so why not define it that way? Have a variable-length list of triggers. Each trigger is associated with a sensor, trigger conditions, some sensor state history and an action.

Trigger1: SensorID, Conditions, Action

Trigger2: SensorID, Conditions, Action

Trigger3: SensorID, Conditions, Action

etc.

With this structure there are no limits to the number of triggers associated with a particular sensor or action.

### Trigger Complexity

What if we wanted to have a trigger generate two actions – say ring a bell and turn on a light. We could create a new trigger class capable of holding two actions or we could create two simple triggers – both referring to the same sensor and the same conditions but to different actions.

Keeping with the idea of using simple elements to build complex processes, I would go with the second solution.

Using the second solution has many benefits. Keeping triggers simple will keep the code small and simple. Building solutions via a combination of triggers will provide much more flexibility. Imagine, for example, if we wanted a trigger to generate three actions. If we went with the first solution we would need to write more code, whereas the second solution works for any number of actions. The only downside to the second solution is that it may make the configuration a bit more complex, but it may be possible to solve this within the design of the configuration utility.

### Trigger Data

Trigger data has four possible forms:

A human friendly form

A machine-readable form for data transfer between the sensact and config

An in-memory set of data structures inside the Triggers part of the code.

A storage form on the sensact

I am confused by the current data transfer form – a stream of comma separated ASCII values - which appears to be more of a human-friendly form than a machine-readable form. The Sensact code has to parse out the commas and convert the ASCII data to the corresponding numbers.

I would recommend changing the data stream to bytes which can be parsed easily – ideally in a state-machine loop which examines each byte once and does not require storage of the entire string prior to processing. Also the data should contain some check information which will be able to detect data corruption.

Details need to be worked out …

### Trigger control of actions

We may want actions which happen only once for a short time.

We may want actions which happen repeatedly as long as a sensor is in a particular state.

We may want actions which only happen after a sensor has been in a particular state for a certain period of time.

Triggers need to be able to control all of these.

## Trigger Code and Structure Proposal

Here is my attempt to define triggers and trigger processing. Again, the pseudo-code is designed to illustrate the concepts. Most types have been omitted. Actual code will be somewhat different.

struct trigger {

// Static elements

sensorID; // The sensor associated with this trigger

requiredState; // See state-machine below

condition; // greater-than, less-than or equal-to

value; // value to be compared with the sensor data

delay; // time to delay before doing the action

repeat; // True if action should be repeated

actionID; // Action to be taken

actionParam; // Action parameters

actionState; // See state-machine below

// Dynamic elements

bool actionTaken; // True when action has been triggered

time\_t onTime; // The time when the condition was met

The trigger code might look something like this:

for each data from the sensors

for each trigger

if (data.sensorID == trigger.sensorID)

if (data.value meets the condition AND

trigger.requiredState == sensor.state) {

// Conditions for triggering are met

if (onTime == 0) { // Not triggered previously

onTime = currentTime;

if (delay == 0) {

sensor.state = trigger.actionState;

Queue the Action

actionTaken = true;

}

} else if (actionTaken == false) { // Waiting for delay

if ((currentTime – onTime) > delay) {

sensor.state = trigger.actionState;

Queue the Action

actionTaken = true;

}

} else if (repeat) { // triggered at least once

Queue the Action // … again

}

} else {

// Conditions for triggering are no longer met

actionTaken = false;

onTime = 0;

}

}

}

}

### State Machine

Attaching a requiredState and an actionState allows for the creation of state-machine solutions. One state value would be stored per sensor (not per trigger) and could be used to control the action of triggers. See the use cases below for examples of this.

The advantage of state machines is that they make it possible to build very complex responses out of very simple elements. The code above is all that would be required to create – for example – a menu system.

The significant disadvantage is that unless you have a comp-sci background defining a state machine may be beyond you. I would like to explore the possibility of creating a configuration utility which presents pre-built state-machines in terms the expected user can understand and manipulate, but use the state machine concept in the background to control the device. This would mean that many functionality extensions could be done by changing only the configuration utility.

# Use Cases

Here are some examples of what can be done with the trigger, sensor and action definitions given above.

**A joystick which controls a mouse**

There are two sensors (as far as the triggers are concerned): joy\_x and joy\_y. Both return values between -100 and +100. States are not used and are omitted in the example.

Trigger 1

SensorID: joy\_y

Condition: > 50

Repeat: true

Action: mouse up

Trigger 2

SensorID: joy\_y

Condition: < -50

Repeat: true

Action: mouse down

Trigger 3

SensorID: joy\_x

Condition: > 50

Repeat: true

Action: mouse right

Trigger 4

SensorID: joy\_x

Condition: < -50

Repeat: true

Action: mouse left

**A two-action button**

In version 2 of the code there is the concept of a button which would do one thing if pressed and released quickly, and something different if pressed and held. This can be accomplished using the code above and states.

We will call the sensor ‘button’. The initial state is 1.

Trigger 1

SensorID: button

RequiredState: **1**

Condition: > 70

Repeat: false

Delay: 0

Action: None

ActionState: **2**

When the button is pressed we go immediately to state 2. There are two triggers defined for state two. The first one does Action A if the button is released quickly. The second triggers a different action if the button is held. Note that either trigger 2 or trigger 3 will fire, but both cannot.

Trigger 2

SensorID: button

RequiredState: **2**

Condition: < 70

Repeat: false

Delay: 0

Action: Action A

ActionState: **1** - return to initial state

Trigger 3

SensorID: button

RequiredState: **2**

Condition: > 70

Repeat: false

Delay: 500ms

Action: Action B

ActionState: **3**

The 4th trigger is used to wait for the held button to be released. No action is taken in this case – we just return to state 1.

Trigger 4

SensorID: button

RequiredState: **3**

Condition: < 70

Repeat: false

Delay: 0

Action: None

ActionState: **1**

A State Diagram for this:

Action A

Action B

value < 70

value < 70

value held

value > 70

**A small menu system**

This example illustrates how a menu system could be created. This system is quite simple. If the button is held a buzzer sounds every second – at a different pitch. Each buzzer sound indicates a transition to a different state. When the button is released subsequent button presses do the action appropriate to the selected state. If the button is left unpressed for 2 seconds the system reverts to state 1. This set of triggers controls volume-up, volume-down and power on/off with a single button.

Trigger 1

SensorID: button

RequiredState: **1**

Condition: > 70

Repeat: false

Delay: 1000 ms

Action: Buzzer – 100hz

ActionState: **2**

Trigger 2

// This trigger takes you to the next menu item at Trigger 5

SensorID: button

RequiredState: **2**

Condition: > 70

Repeat: false

Delay: 1000 ms

Action: Buzzer – 200hz

ActionState: **3**

Trigger 3

// This trigger takes you to the volume-up option

SensorID: button

RequiredState: **2**

Condition: < 70

Repeat: false

Delay: 0

Action: None

ActionState: **21**

Trigger 4

// This trigger stays in the volume-up action

SensorID: button

RequiredState: 2**1**

Condition: > 70

Repeat: true

Delay: 0

Action: Volume-up

ActionState: **21**

Trigger 5

// This trigger takes you to the next menu item at Trigger 8

SensorID: button

RequiredState: **3**

Condition: > 70

Repeat: false

Delay: 1000 ms

Action: Buzzer – 300hz

ActionState: **4**

Trigger 6

// This trigger takes you to the volume-down option

SensorID: button

RequiredState: **3**

Condition: < 70

Repeat: false

Delay: 0

Action: None

ActionState: **31**

Trigger 7

// This trigger does the volume-down action

SensorID: button

RequiredState: 3**1**

Condition: > 70

Repeat: true

Delay: 0

Action: Volume-down

ActionState: **31**

Trigger 8

// This trigger takes you to the power on/off option

SensorID: button

RequiredState: **4**

Condition: < 70

Repeat: false

Delay: 0

Action: None

ActionState: **41**

Trigger 9

// This trigger does the volume-down action

SensorID: button

RequiredState: 4**1**

Condition: > 70

Repeat: true

Delay: 0

Action: Power On/Off

ActionState: **41**

Trigger 10

// This trigger resets the system when the button is not pressed

SensorID: button

RequiredState: **Any**

Condition: < 70

Repeat: false

Delay: 2000 ms

Action: none

ActionState: **1**

Yes, I know. It’s a lot of triggers. Three per menu state plus one. But it could all be driven by the fairly simple bit of code shown on page 6. And we could play with a lot of triggering mechanisms – try them on real people – and then package them under a more comprehensible façade in the config utility.