SenseBoard Protocol

The Sense Board is a USB board developed by the OU which allows the user to interface with various sensors, lights and motors. The board at present (version 3) provides the following:

- 4 resistive inputs
- A microphone
- A linear slider
- A button
- An IR sensor
- 7 LEDs (no brightness control, either on or off, the 7th LED can also be used to drive an included infra-red LED)
- A stepper motor
- Two servos

The resistive inputs, microphone and linear slider all provide 10 bit values. The button and IR sensor provide boolean values.

The protocol used to drive the board also contains instructions for additional features not currently present on the board, presumably for future expansion. These include:

- An 8th LED
- Three additional stepper motors (4 in total)
- Two additional servos (4 in total)
- Eight continuous DC motor outputs

Where "Protocol only" is mentioned, this is documenting one of these features that isn't yet implemented in hardware at the time of writing (but could be for a future release.)

Connection

Connection to the board is USB over serial (classic FTDI FT232RL chip on the underside of the board handling serial to USB.) The settings should be 115200 baud, 8 data bits, 1 stop bit and no parity.

Sending commands

Commands are sent as either 3 or 4 byte groups. When sending commands to the board, the first two bytes are always the first command header (0x54) and the second command header (0xFE), the third byte is the command and the fourth byte is an optional argument byte.

For example, to send a reset command to the device the following array of bytes would be sent:

```
[0x54, 0xFE, 0x10]
```

Side node: Byte numbers in this document are given in either hexadecimal, decimal or binary depending on what seems easiest to understand given the context. Digits prefixed with "0x" are hexadecimal, "0b" are binary, and those with no prefix are decimal.

The available commands (used for the third byte) are as follows.

StepperCommand	*0xF0*	
SetPwmDuty	*0xD0*	
LedOn	*0xC1*	
LedOff	*0xC0*	
Ping	*0x00*	
Reset	*0x10*	
SetBurstMode	*0xA0*	
MotorCommand	*0x80/0x81*	**Not implemented in current device*

Stepper Command – 0xF0

This is used for turning the stepper motor a specified number of steps, if attached. Note that for this to work, the external power must be provided to the board. There appears to be no way within the protocol to determine if this is the case since the "ack" (covered later) will be sent back whether power is attached or not.

This command makes use of the fourth argument byte, which is a standard two's compliment signed byte representing the number of steps the motor should take, ranging from -128 to +127. Negative values will turn the motor anticlockwise, positive values clockwise.

The general pattern for stepper commands is therefore given as follows:

```
[0x54, 0xFE, 0xFi, 0bdssssssss]
```

…Where `i` is the stepper motor index, `d` is the direction of the stepper and `s` is the number of steps the motor should turn.

So to turn the motor clockwise 100 steps the following bytes would be sent:

```
[0x54, 0xFE, 0xF0, 100]
```

And to turn the motor anticlockwise 57 steps:

```
[0x54, 0xFE, 0xF0, -57]
```

Side note: Stepping more than 127 steps appears to be implemented in the Sense protocol by sending multiples of these commands – so to step 300 steps clockwise you would step 127, 127 then 46 steps. However, this doesn't seem to provide the desired behaviour in Sense, with the latter commands being overwritten or ignored – perhaps a bug in hardware? Since nothing is received from the board after the motor has stopped turning, it doesn't seem possible to solve this one at present.

Protocol only: The protocol also provides for 3 additional stepper motors, not present on the board (though the board does use the L293D motor driver which is capable of driving 4 discreet channels.) With the stepper motors numbered 0-3, the command byte becomes "0xF0 OR num". In practice, this means the first motor (the one currently implemented) is 0xF0, the next 0xF1, then 0xF2, and 0xF3. So to turn the 4th stepper motor clockwise 50 steps the following would be used:

```
[0x54, 0xFE, 0xF3, 50]
```

SetPWMDuty – 0xD0

This is used for the servo ports on the board, turning the servo (if attached) through 180 degrees. As for the stepper motor command above, the external power must be connected for this command to work.

This command makes use of the fourth argument byte, which is a standard two's compliment signed byte representing the servo position, ranging from -128 to +127. The servo is set to its central position when 0 is passed as an argument.

The protocol allows for 4 such outputs which are selected by 'OR'ing the motor number with the command byte (as with the stepper motor above.) So the command byte becomes 0xD0 or 0xD1 depending on which output should be switched. The two further servo outputs specified by the protocol but not currently present on the board would likewise be specified by 0xD2 and 0xD3.

The general pattern for PWM duty (servo) commands is therefore given as follows:

```
[0x54, 0xFE, 0xDi, 0bdaaaaaaa]
```

…Where `i` is the servo index, `d` is the sign bit representing direction, and `a` is the angle.

So to turn servo A as far as possible in one direction:

```
[0x54, 0xFE, 0xD0, 128]
```

And to turn servo B as far as possible in the other direction:

```
[0x54, 0xFE, 0xD1, -127]
```

MotorCommand – 0x80/0x81 (Protocol only)

This command is used to select the power level and turn on and off up to 8 continuous DC motors. No DC motor ports appear to be present on the current board, so this entire command only exists in the protocol at present.

The command byte is 0x80 if the motor should run backwards, 0x81 if it should run forwards. The argument byte specifies the motor number and speed - the motor number is specified in the bottom 3 bits, speed is in the top 3 bits. (The middle two bits are unused.) If the speed is set to 0, the motor is turned off.

The general pattern for motor commands is therefore given as follows:

```
[0x54, 0xFE, 0x8n, 0bppp00iii]
```

…Where `n` is either 0 or 1 depending on the direction, `p` is the desired speed and `i` is the desired motor index.

So to turn the first motor on full power forwards:

```
[0x54, 0xFE, 0x81, 0b11100000]
```

And to turn the fourth motor on roughly half power backwards:

```
[0x54, 0xFE, 0x80, 0b10000100]
```

LEDOn / LEDOff – 0xC1 / 0xC0

This command is used to turn the 7 (8 in the protocol) LEDs on or off. These outputs are purely digital, setting them to different brightness levels is not supported. 0xC1 is the command byte used to turn the LEDs off, 0xC0 is used to turn them on. An argument byte is used which is the mask specifying what LEDs should be turned on or off. The least significant bit in the argument byte represents LED 1, the MSB represents LED 8 (the "protocol only" LED.)

The pattern for LED commands is therefore given as follows:

```
[0x54, 0xFE, 0xCn, 0xmmmmmmmm]
```

…Where `n` is either 1 or 0 depending on whether the LEDs should be switched on or off, and `m` is the mask of LEDs that should be switched.

So to turn LEDs 1, 3 and 4 on the following bytes would be sent:

```
[0x54, 0xFE, 0xC1, 0b00001101]
```

Note that this will not affect the behaviour of the other LEDs, their state will remain the same (it will not turn them off.) Turning off the LEDs can only be achieved via the LEDOff command, 0xC0. So to turn off all the other LEDs we could send these bytes in addition to the ones above:

```
[0x54, 0xFE, 0xC0, 0b11110010]
```

Reset – 0x10

This command simply resets the device, no argument byte needed. So to use it simply send the following bytes:

```
[0x54, 0xFE, 0x10]
```

Ping – 0x00

This command will simply cause the device to ping back its version / firmware revision. As above, no argument byte needed, just the following:

```
[0x54, 0xFE, 0x00]
```

SetBurstMode – 0xA0

Setting the device in burst mode will cause it to start streaming back bytes that contain the sensor data (reading the sensor data is covered later.) Contrary to what might be expected it does take a argument byte – this byte is a mask of the 8 separate sensor values, which are labelled as follows:

1. Slider
2. Infrared
3. Sound
4. Button

5. Input A
6. Input B
7. Input C
8. Input D

Taking 1 as the LSB and 8 as the MSB, these values are then arranged into a mask – 1 at the relevant location means we want this sensor data, 0 means we don't.

The burst mode command is therefore sent using the following pattern:

```
[0x54, 0xFE, 0xA0, 0bmmmmmmmm]
```

…Where `m` represents the mask bits.

So to start streaming back all sensor data (the most common scenario), we simply send the following:

```
[0x54, 0xFE, 0xA0, 0xFF]
```

There are potential cases however when we may only want some data, and want this data to take up all the available bandwidth. The most likely case would perhaps be the infrared sensor, which if streamed on its own could feasibly be fast enough to detect separate pulses from remote controls.

So if we wanted to just start streaming data from the infrared sensor, we could use the following:

```
[0x54, 0xFE, 0xA0, 0b00000010]
```

There appears to be no way to stop burst mode other than to reset the device (0x10 covered above) though this shouldn't be an issue.

Receiving data

The Sense Board also sends data to the PC which falls into two main categories, sensor data (requested using the burst mode command covered above) and acknowledgements (sent back for each command that's sent to the device.) The data comes back in 3 byte packets, so it's possible in theory to set the receive threshold to 3 on the PC side, pick the packets up 3 at a time and then examine the bytes to determine the value. However, in practice it's best to pick them up individually and then process them using a finite state machine, since otherwise the sync often seems temperamental (presumably because the packets may be out of alignment.)

Sensor Data

The first byte of all 3 byte sensor data is the "burst header", which is **0x0C**. If this byte is detected, the next two bytes will contain the data from a certain sensor on the board. The sensor number is the top three bits of the second byte. The data value is the bottom two bits of byte two plus all eight bits of byte three. The sensor numbers are given in the documentation for SetBurstMode above. Note that sensor numbers run from 0-7, not 1-8, so subtract one from the numbers there to match the ones sent back in the protocol.

The sensor data that's returned should therefore match the following pattern:

```
[0x0c, 0bsss000ee, 0bddddddddd]
```

…Where `sss` is the sensor number and `eedddddddd` is the (10-bit) data value.

The data value is 10 bits for all the sensors, even the boolean ones such as IR and button. However, for such boolean sensors the value will always be either 0 or 1023 (all data bits 0 or all data bits 1.)

Given a 3 byte array containing sensor bits 0, 1 and 2 in order, the Java code to extract the data and sensor number would look like this:

```
int data = ((arr[1] << 8) & 1023) | (arr[2] & 0xFF);

int sensorNum = arr[1] >> 4
```

Note that we don't need to touch arr[0] since it's the known header byte (0x0C).

Acknowledgements

Whenever a command is sent to the board, an acknowledgement is sent back. This is useful since it allows a basic degree of error correction by waiting for acknowledgements when a command is sent, then potentially re-trying if a timeout occurs. No strict recommendation is given for a timeout value, but on my machine they are generally returned in under 20ms; a timeout value of 500ms+ would therefore likely be plenty.

Acknowledgements are always of exactly the same 3 byte format, which is the FirstAckHeader (0x55), SecondAckHeader (0xFF), then the Ack (0xAA).