HANDLE

Developmental pathway towards autonomy and dexterity in robot in-hand manipulation

Deliverable 3

Report and Physical Delivery

Augmented Sensing Object

Due date of deliverable: **Month 4**
Actual submission date: **Month 4**
Partner responsible: SHADOW
Used Person/months: [number]

Version [number and status]

Classification: [indicate]
Grant Agreement Number: 231640
Contract Start Date: February 2, 2009-03-18
Duration: 48 Months
Project Coordinator: UPMC
Partners: UPMC, SHADOW, UC3M, FCTUC, KCL, ORU, UHH, CEA, IST
Project website address: www.handle-project.eu


1 Version management

Version	Date	Status	Author	Modification
A	29th May 2009		Jake Goldsmith	None – New Document

2 Initial Proposed Design

As part of WP6 Shadow were tasked with the design and production of a number of Sensing objects which could be used in the initial stages of manipulation testing. This report details the specification discussions along with the design decisions taken during the process. Below is the initial list of requirements and specification points which were raised during the initial discussions.

1. Must be based on an object which is freely available to all partners of Handle Europe Wide.

2. Must be of a size to allow for in hand manipulation

3. Must have sufficient tactile sensor covering so as to be able to easily detect types of grasps

4. The tactile sensing Should be sensitive enough to detect light-touch grasping

5. Should have secondary sensing method ideally a multi-axis accelerometer or gyro.

6. Metal Elements should be kept to a minimum so as to avoid interference with the Polhemus Sensors to be used later.

3 Refined Specification

The considerations were taken by Shadow and refined into the following specifics.

1. Copied object must be able to be purchased Europe wide. Most Likely candidates include international companies, specifically toy companies as they produce objects which are likely to fit other factors.

2. Given the average finger spread of a human hand is around 140mm with a palm width of around 90mm the object dimensions should be less than this. In reality in order to easily manipulate an object within the palm it should be considerably less than this. The fingers should be able to close around the object. Given the distance between the first and little finger this means that the maximum dimension in at least one of the axes of the object should be no more than 65mm.

3. The Tactile sensor coverage should allow data to be collected with regards to the force applied by the hand along with the position of the grip. At the very least this should be able to tell which quarter of each side of the object is being interacted with. Provision for edge force detection and measuring it's own weight should be considered as realistic options.

4. Light-Touch grasping is to be defined in the order of 0.5N/Sqcm. This is less sensitive than a human which is in the order of 0.05N/Sqcm. But given the available sensing technology this would be a good compromise between cost and effectiveness. It will easily be enough to perform the vast majority of delicate grasps.

5. Secondary sensing should bring additional functionality in addition to tactile response. The most useful additional data to be collected would most likely be in the form of world position. This could be used in conjunction with the Polhemus sensors to give position within Palm as opposed to simply position at finger tips. For this data to be useful it should sense orientation in at least 3 axes.

6. All metal elements of the device should be kept to a minimum. At the least the object should contain no ferrous metals. As these would have a particularly detrimental effect on the Polhemus sensors. This should be considered closely when choosing the tactile sensing medium as they will be spread over the whole surface of the object.

4 Design Decisions

4.1 Specification Comparison

1. The object chosen to base the sensing object on is a Rubik's cube. This was deemed to be a readily assessable object across the whole EU. An added benefit is that it will look good in a demo due to the Rubik's cube being a cult object. In addition to this it's faces are already divided into suitable sensing areas (3x3 squares on each face). This will mean it could be used in conjunction with the vision system at a later date. Replacement stickers for Rubik's Cubes are readily available. These will be used on the final unit in order to maintain consistency.

2. The Rubik's cube measures 57mm x 57mm x 57mm. This fits within the specified envelope.

3. The obvious placement of sensing areas on the Rubik's cube would be one per coloured square. This means that each face contains 9 sensing regions. This exceeds the specified sensing per quarter face requirements. It was decided that the pressure sensitive pads would be extended as far as possible up to the edges in order to cover any edge grasping.

4. Various sensing methods have been investigated given the specification and layout of the sensors as specified above. For a more detailed beak down on which sensors were chosen please refer to the following section “Breakdown of Sensor Choice”

5. The decision was taken to incorporate accelerometers. Each face will have it's own 3-axis accelerometer. This obvious duplication of data will allow for easy error correction. If necessary later it may also be possible to incorporate a full 3-axis gyroscope into the centre.

6. Most of the design is constructed using Delrin, an acetal plastic. The exception is the support panel for each face had to be made from Aluminium. The thin size and the forces it would be under left no choice but to go for a metal. The PCBs behind each face are also 4-layers copper. This was unavoidable as a certain amount of the signal processing had to happen within the unit. Failure to do this would have lead to unacceptable amounts of electrical noise on the analogue signals.

4.2 Breakdown of Sensor Choice

The methods detailed below are a selection of sensing methods which could be relevant to the design of object. As most of the following sensing methods are industry standard, the description will be solely in reference to their suitability to the sensing object. Main reason for not choosing the type of sensor highlighted in red

4.2.1 QTC (Quantum Tunneling Composite)

Advantages: Very Slim Profile, small package size, wide force range.

Disadvantages: Poor at static force measurement, suffer from drift, requires regular calibration, Difficult to source commercially.

N.B. For further information regarding QTC refer to www.peratech.co.uk

4.2.2 Load Cells

Advantages: Generally digital output, Very high accuracy, wide availability, range of force sensitivity.

Disadvantages: Large package size.

4.2.3 Strain Gauges

Advantages: High accuracy, widely available, range of force sensitivity.

Disadvantages: Requires a metal support framework

4.2.4 Optical Sensing

Advantages: High accuracy, widely available, range of force sensitivity.

Disadvantages: Over complex support structure and electronics, Relatively large amount of space required for installation.

4.2.5 FSR (Force Sensing Resistors)

Advantages: Very Slim Profile, Low Cost, small package size, wide availability, range of force sensitivity.

Disadvantages: Poor at static force measurement, suffer from drift, requires regular calibration.

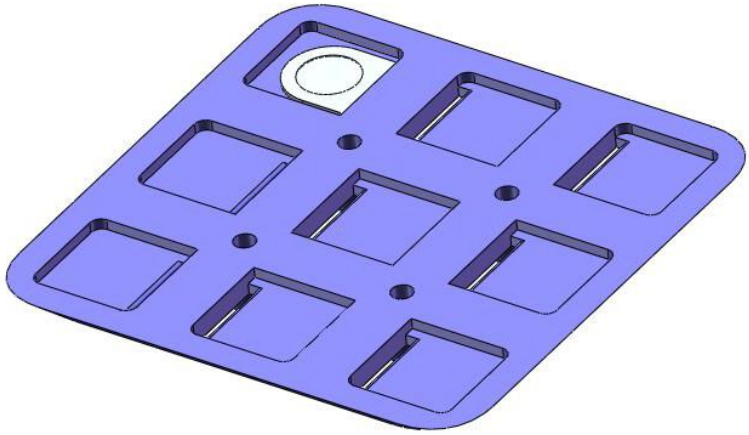
Given the reasons highlighted above the FSR is the best choice. The 2 main manufactures of FSRs are Tekscan with the Flexiforce range (by far the best selling) and Interlink. Initially the Flexiforce sensor (0-1lb range) was purchased and initial tests were carried out to access the suitability of the technology. It was found to be able to sense within the range as specified in Specification point 4. However as the design of the object progress it became clear that the Flexiforce sensors were going to be too large to easily fit into the 3x3 spacing available. At this stage the decision was made to change over to the Interlink sensors as they had a much smaller pad size (5mm diameter).

5 Final Product

The following is a step by step breakdown of the design with annotation.

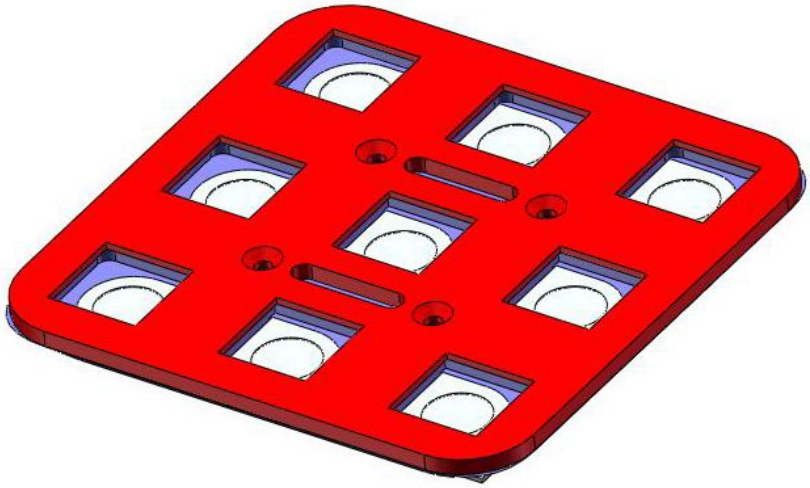
5.1 Lower Button Layer

This is the main support for each side of the cube. It holds the 9 FSR sensors, the connections of which pass through to connect to the PCB below.



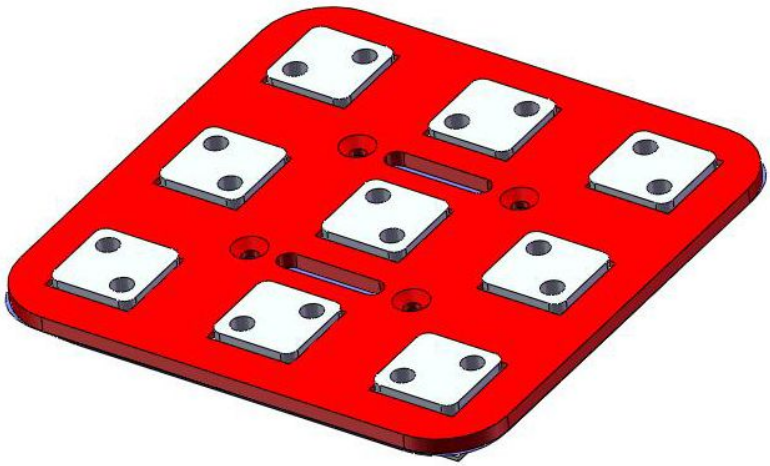
5.2 Lower Button Half

The Bottom half of the button is then added on top of the sensor. Beneath the button is a small raised circle. This is used to concentrate the force on to the sensor.



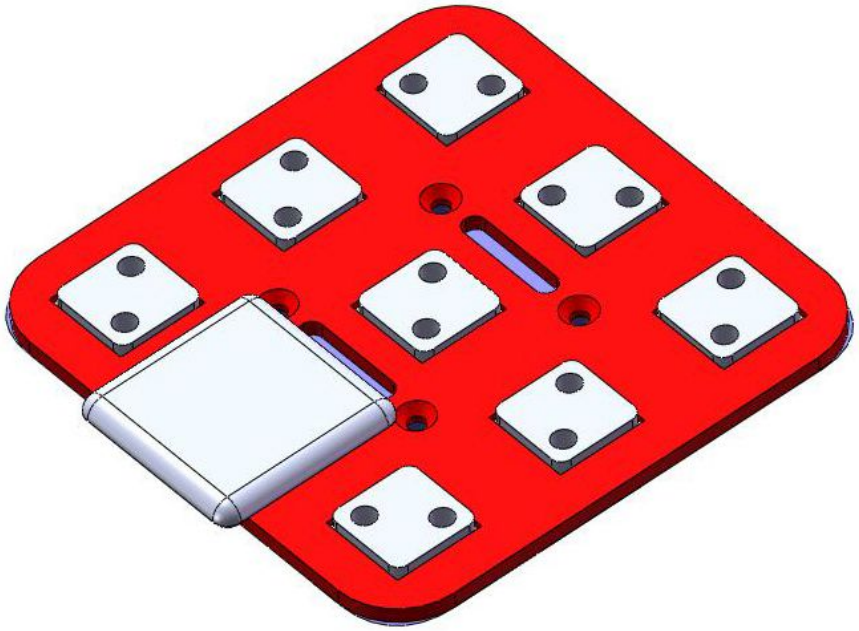
5.3 The Upper Button Layer

This layer is then added to hold the lower button in place.

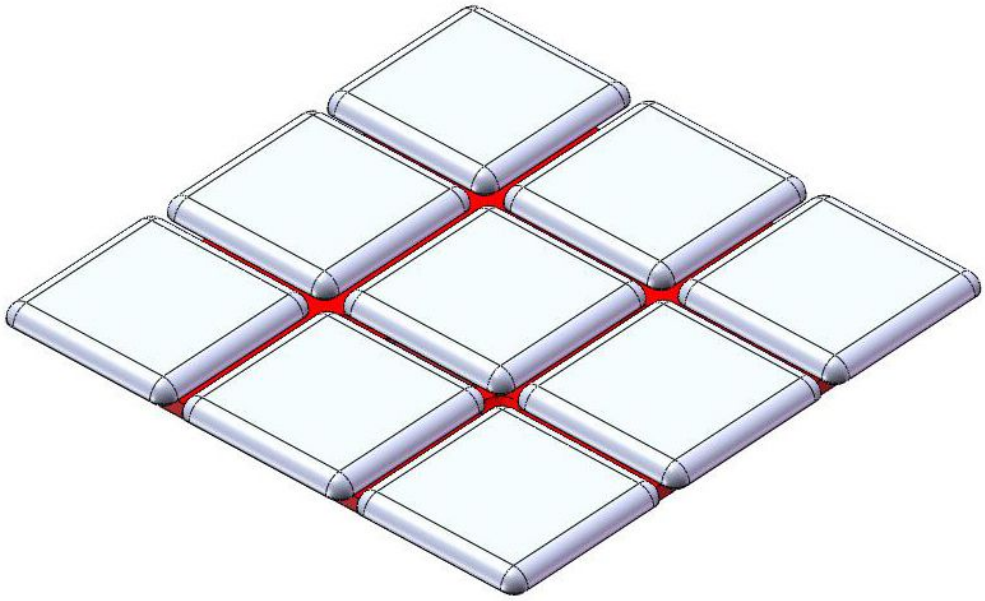


5.4 The Button Top Half

The top half of the button is added over the top of the upper button layer. The top of the button over hangs slightly towards the open edge. This is to allow the edge to trigger when grasped.

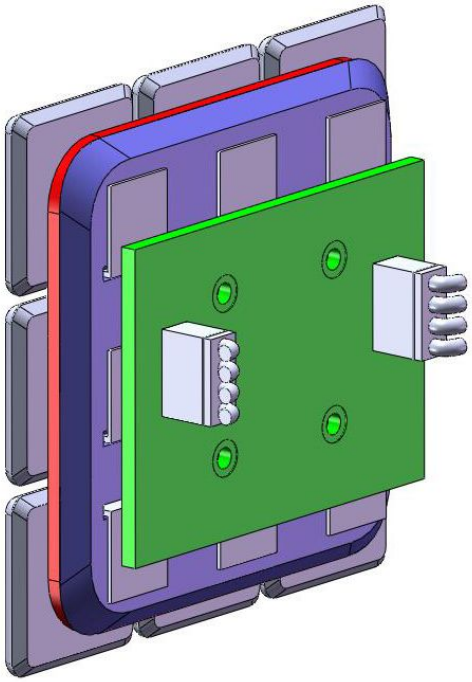


5.5 All the Buttons in Place



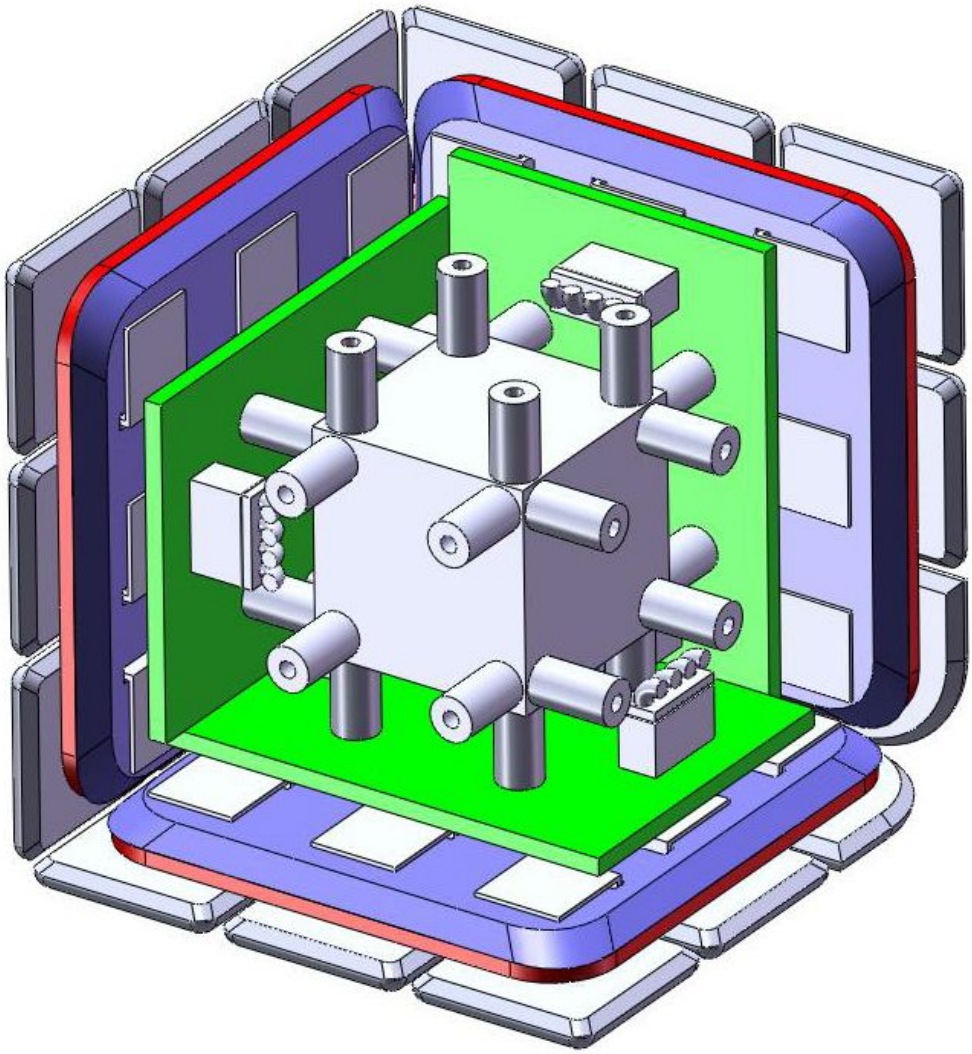
5.6 The PCB is Held Below Lower Button Plate

The tabs of all the FSRs are connected to the top side of the PCB (For more detailed information on the design and function of the PCB please refer to the section 'Electronics Overview')



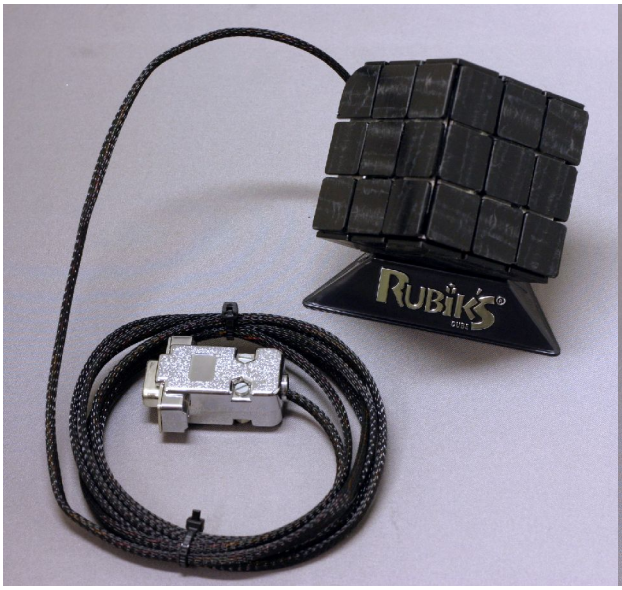
5.7 The Side units are attached to the central cube.

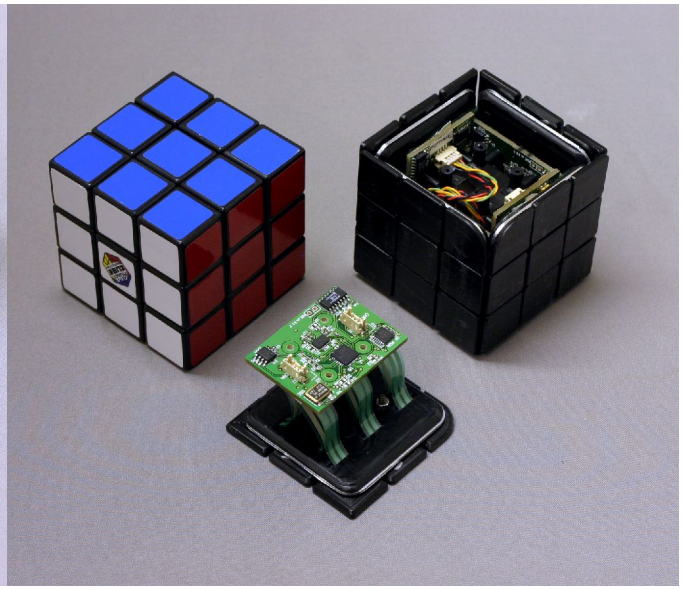
Each board has a CANbus output. All the boards are connected together and fed to a single output. For Further information on the firmware and electronic communication please refer to the below section 'Electronics Overview' or Appendix II.

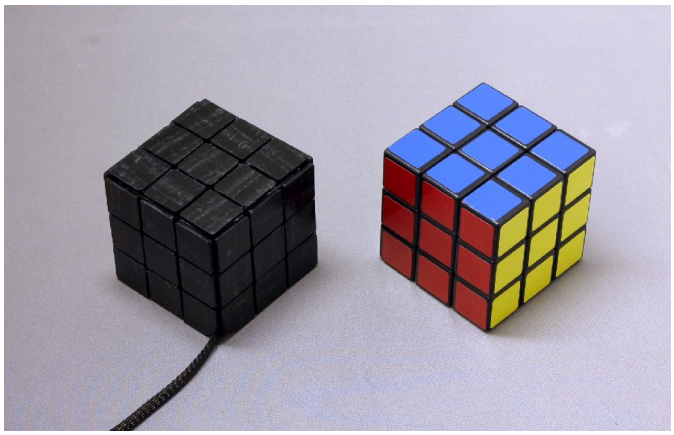


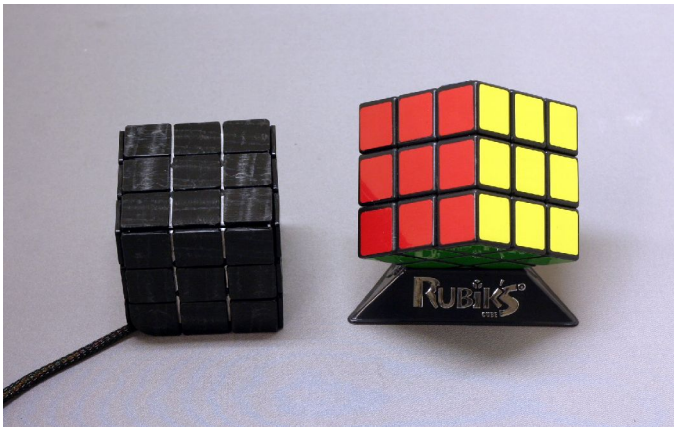
5.8 Final Assembly

With all the buttons and panels in place the main CAN wire exits the unit at the cut away corner.









6 Electronics Overview

6.1 Description

This board is the ADC and accelerometer board that makes up one face of the HANDLE sensing object. It has board type 0210 in the ShadowProtocol (Appendix II)

It implements the full Hand protocol.

Each board is programmed with a default transmit base address and default receive base address. These can be changed using the ShadowProtocol. The board is set up to automatically start transmitting on power-up.

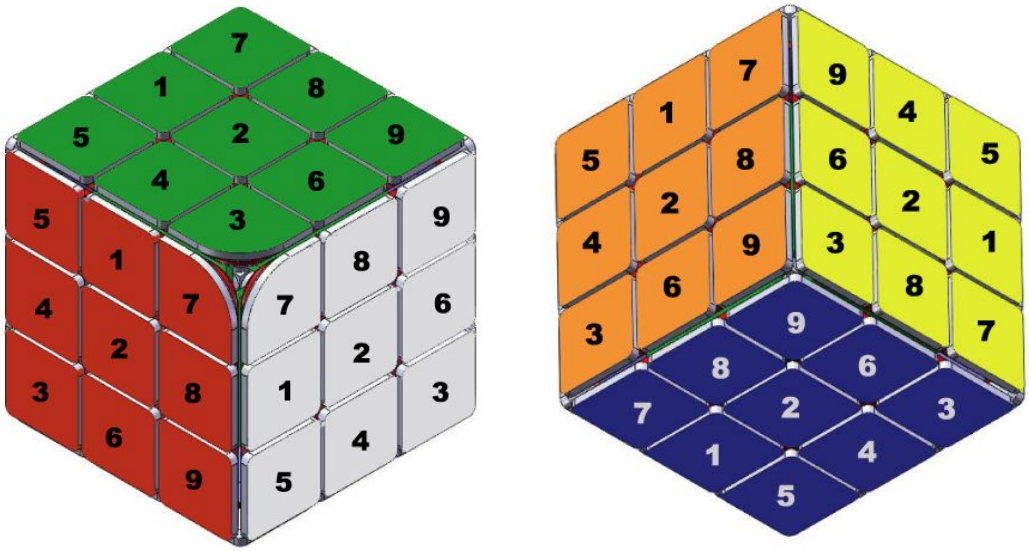
The board implements 12 sensing channels. These are transmitted in CAN messages as follows:

s(x)L is the low byte of sensor (x).
s(x)H is the high byte of sensor (x).

ID	D0	D1	D2	D3	D4	D5	D6	D7
txbase	s0L	s0H	s1L	s1H	s2L	s2H	s3L	s3H
txbase+4	s4L	s4H	s5L	s5H	s6L	s6H	s7L	s7H
txbase+8	s8L	s8H	s9L	s9H	s1Ll	s10H	s11L	s11H

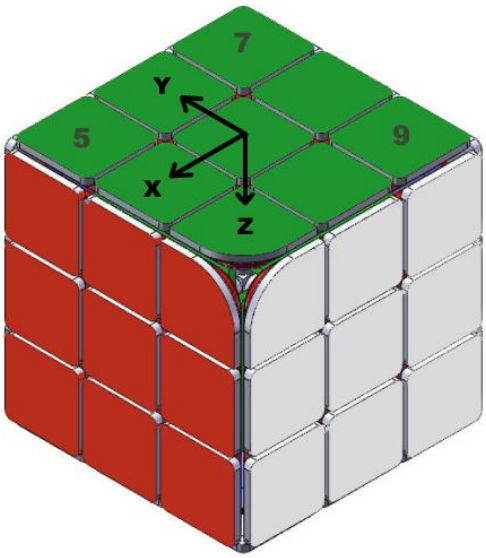
Touch sensor data is 12-bit resolution.

Sensors s0..s8 are the touch sensors on the face of the cube. Sensor s0 is connected to connector C1 on the face of the object, and so through to sensor s8 is connected to C9. The arrangement of the connectors on the face is:



Sen ints
from the face into the cube. X and Y are along the plane of the corresponding face. The accelerometer data is 8-bit resolution.

The accelerometer axis orientation is shown below: The Accelerometers are mounted so as to be on the surface of a theoretical cube



The Accelerometer is set by default on 0 – 2g range. This can be changed to be less sensitive if necessary. The variable gain discussed above is only relative to the tactile sensing not the accelerometers.

By default, the sensing board sends one CAN message every millisecond, and cycles through the three messages to be sent. The ShadowProtocol can be used to change this, so that it sends messages less frequently, using the "txrate" setting.

The touch sensors have gain settings also. These are set by default to 1, but can be modified by messages sent to the receive base address. A message like:

ID	D0
rxbase+3	d

will set the gain of sensor 3 to 'd'.

ID	D0	D1	D2
rxbase+2	a	b	c

will set the gain of sensor 2 to 'a', sensor 3 to 'b' and sensor 4 to 'c'.

Available gain values are

Value sent to board	Resulting gain
0	+1
1	+2

2	+4
3	+5
4	+8
5	+10
6	+16
7	+32

6.2 Hardware description

The board is a CANbus board implementing our standard protocol. You will need to reference the circuit diagram in Appendix I within this section.

It has 9 connectors CONN1..CONN9 designed for use with force-sensing resistors - each is also connected to a 4.7k pulldown providing a voltage divider.

CONN1..CONN5 feed a programmable gain amplifier MCP6S26 (U7) channels CH0..CH4. CONN6..CONN9 feed another MCP6S26 (U9) channels CH0..CH3.

The output of U7 goes to an ADC MCP3204 (U6) channel CH0,

The output of U9 goes to CH1. CH2, CH3 of U6 are connected to ADC inputs 2,1 respectively

U6 connects to the PIC 18F2480 (U5) SPI port and chip select is RA1.

The PGA's U7,U9 are also connected to the PIC SPI port. CS for U7 is RA2. CS for U9 is RA3.

An accelerometer MMA7455LR1 is fitted as U8. It has an SPI interface to the PIC. CS is RA0. In addition, INT1/DRDY is connected to RC6.

Power is supplied over the CANbus connector. A pair of MIC5205's, U1 and U2 generate 5Vd and 5Va respectively digital and analogue 5V. 5Va is used for the PGAs U7,U9 and to supply Vref for U6 the ADC.

LEDs are fitted as follows:

Red	D2	Chip select for ADC U6
Red	D1	Chip select for Accelerometer U8
Green	D3	RA5 on PIC
Red	D4	Chip select for PGA U9
Red	D5	Chip select for PGA U7

There are no Power or CAN TX/RX/ERR LEDs fitted.

6.3 Using the Hardware

The CANbus hardware is designed to transmit data continuously from all hardware nodes at all times.

Each face has a hard-coded transmit and receive base address. These are:

Face	Transmits	Receives
Green	0x500	0x50C
White	0x520	0x52C
Orange	0x540	0x54C
Blue	0x560	0x56C
Red	0x580	0x58C
Yellow	0x5a0	0x5aC

The CANbus is operated at maximum speed, 1MBit.

Shadow's internal use of CANbus is based around PIC18 microcontrollers, where the baud rate registers BRG1,BRG2,BRG3 are set to values 1,5,1 to achieve this.

Using a PEAK PCI card, which has a ST ?? CAN controller, the BTR0, BTR1 registers would be set to 00, 14 respectively.

Each face will transmit one of the three messages it can transmit every millisecond. The data in that message has just been read, so on receipt of a message the CPU millisecond timestamp could be applied.

The data returned by the sensing object is not calibrated. We have in the past used piecewise linear interpolation from a small number of measured points to calibrate the Dextrous Hand and other robot components; if you connect the Sensing Object to a computer running the full Hand software stack then calibration files can be set up to automatically perform this task.

Example Code

```
/*
 * Sensing object test code, using PEAK PCAN interface card
 *
 * (C) Shadow Robot Company, 2009.
 *
 * This code is example code and put into the public domain
 *
 * The Sensing Object streams data continuously from each of the six
 * faces. This programme receives the messages, identifies which face
 * they are from, and unpacks them. The results are stored in the
 * struct face corresponding to the face of the sensing object.
 *
```

```
 */
#include <errno.h>
#include <error.h>
#include <libpcan.h>
#include <stdio.h>
#include <fcntl.h>
#include <ncurses.h>

/* PEAK card has two devices: this is the port nearest the motherboard */
char * port = "/dev/pcan0";

HANDLE h;

/* This is the data returned by one of the faces.
 *
 * data[0]..data[8] are the force sensors, data[9]..data[11] are the
 * accelerometers.
 *
 * base is the base address of the face.
 */

struct face {
  int data[12];
  unsigned int base;
  char * name;
};

int messages_seen[0x800];

/* Setup the base addresses for the faces of the object */
struct face object[6] = {
  { .base = 0x500, .name="green" },
  { .base = 0x520, .name="white" },
  { .base = 0x540, .name="orange" },
  { .base = 0x560, .name="blue" },
  { .base = 0x580, .name="red" },
  { .base = 0x5a0, .name="yellow" }
};

#define FACE_DATA_SIZE 12 // There are 12 data items sent by the face.

/* This is just for diagnostic purposes. In the real world you would want to store this data somewhere
useful. */
void print_object(void) {
  int i,j;
  for (i=0; i<6; i++) {
    mvprintw(i+1,0,"%d: %s",i, object[i].name);
    /* The force sensors */
    mvprintw(i+1,12,"");
    for (j=0;j<9;j++)
      printw(" %03x", object[i].data[j]);
    printw(":");
    /* The acceleration sensors */
    for (j=0;j<3;j++)
      printw("%03x ", object[i].data[j+9]);
```

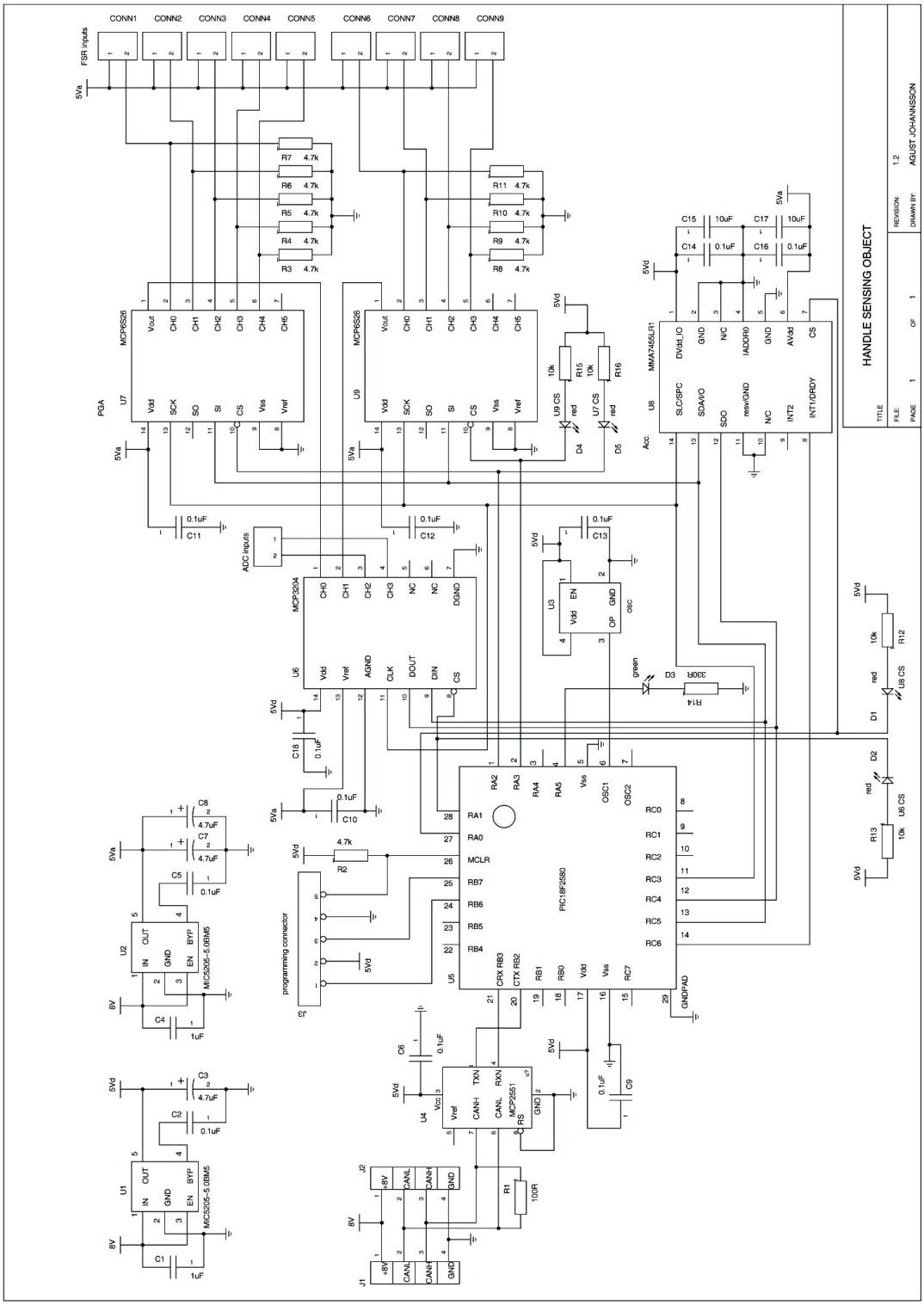
```
    clrtoeol();
  };
}

/* Call with a message that has just been received. */
void process_message(TPCANMsg *m) {
  int i,j;
  mvprintw(9,1,"ID=%x", m->ID); clrtoeol();
  messages_seen[m->ID]++;
  for (i=0; i<6; i++) {
    /* Search the faces until we find the one that matches this message */
    if ((object[i].base <= m->ID) && ((object[i].base+FACE_DATA_SIZE)>m->ID)) {
      /* The message contains LEN/2 16-bit data items. Unpack them and store them */
      for (j=0;j<m->LEN/2;j++) {
            unsigned int v;
            /* Convert the two bytes into an integer */
            v = m->DATA[j*2];
            v |= m->DATA[(j*2)+1]<<8;
            object[i].data[ m->ID - object[i].base + j] =v;
      };
    };
  };
}


int list_messages(void) {
  int i;
  int count=0;
  for (i=0;i<0x800;i++) {
    if (messages_seen[i]) {
      count++;
      printf("%04x: %12d  ", i, messages_seen[i]);
      if ((count%8)==0)
            printf("\n");
    };
  };
}

int main(int argc, char ** argv) {
  __u32 status;
  /* Open the CAN poer. */
  h = LINUX_CAN_Open(port, O_RDWR);
  if (!h) error(1,errno, "can't open %s", port);
  /* clear status */
  CAN_Status(h);
  /* Set it up for 1MBit */
  errno = CAN_Init(h, 0x14, 0);
  if (errno) error(1,errno, "CAN_Init");

  atexit(list_messages);

  initscr();
  cbreak();
  noecho();
  nodelay(stdscr, 1);
```

```
  clear();
   while (getch()!='q') {
     TPCANMsg m;
     /* Get a message */
     errno=CAN_Read(h, &m);
     if (errno) error(0,errno, "on CAN_read, continuing");
     else process_message(&m);
     /* Clear any status info */
     status = CAN_Status(h);
    if ((int)status < 0) {
      errno = nGetLastError();
      error(0,errno," CAN_Status()");
    } else
      mvprintw(10,0,"pending CAN status 0x%04x read.\n", (__u16)status); clrtoeol();
    print_object();
    refresh();
  };
   endwin();
   list_messages();
}
```

7 Further Sensing Objects to be Supplied

The second sensing object which will be supplied is a gyro/accelerometer standard house hold object. Shadow has designed a CANbus based gyro/accelerometer Unit. This can be fixed into any object I.e. A mug, bowl etc. This allows for a large range of objects to be used during testing.

8 Appendix I – PCB Schematic



9 Appendix II – Shadow Code Protocol

9.1 Chapter 1. Hand Protocol. Booting and Configuration of nodes.

9.1.1 The Protocol

We've tried to keep the protocol used over the CANbus simple. We were trying to make CAN behave usefully for our purposes. There are various specifications for providing a protocol layer on top of CAN: but we decided to implement another simple one.

There are two useful views of CAN: shared memory and message passing.

In the shared memory view,a node sends periodic updates of a variable (e.g. sensor data). The conversion from messages to shared memory is highly dependent on the nature of the message, and so our interface code provides for a separate decoding routine to be registered for each message ID used.

In the message-passing view, a CAN message provides data as an update on the state of some part of the system (e.g. a command: open valve).

In the same way that both point-to-point and multicast messages are useful in internet communications, we found uses for both shared memory and message-passing, and so we have produced a bus design supporting both. Data from sensors, valve settings, and other variable-type data can be regarded as shared memory. Point-to-point messages are used for configuration settings (see configuration detail) and other protocol data. The short size of the can message (8 bytes) requires that we do certain optimisations on these.

The CANbus as used in the Dexterous Hand runs at the maximum possible speed, 1Mbit. Data is transmitted from any node on the system to all others. This data includes

sensor data

> readings from sensors on the Hand, packed to have multiple sensor values in each message

valve commands

> control signals sent to valves on the Hand, typically durations for turning on

configuration data

> system data sent to the micro-controllers to adjust their protocol settings

protocol data

> information sent to all nodes to announce that a node has booted up or to tell a node to enter a new mode

Sensor data is continuously transmitted by nodes with sensors attached. Because all nodes see this data, it can be used internally within the Hand as well as by the host PC. An example of this is that

the PID controllers on valve drivers will take set point and target sensor values directly out of messages on the bus. Sensor data formats are documented below.

Valve commands are normally sent only by the host PC. Details of the commands are below. All nodes see valve command data, but only the appropriate valve driver will take any action on the data.

Configuration data is sent from the host PC to one node at a time. The node must first be set into configuration mode, using a protocol data message. Configuration data can cover the settings of controllers, the CAN message ID's to use, the rate to transmit at, whether to drive certain hardware, settings for PID controllers, and so forth. (You can see the configuration data for a node in the file `/proc/robot/nodes/`*`NODENAME`*`/config`)

Protocol data is sent by all nodes from time to time. When a node powers up, or recovers from reset, it sends out a BOOTED message. This is acknowledged by the configuring host, which then sends configuration messages to it. The node responds with acknowledgement messages, which might be CRCs of the data, or the node ID of the node.. Finally, the host sends protocol messages to set the node active. Other protocol messages can make a node idle, or reset the node.

9.2 NUUID - Node Universally Unique IDentifiers

In order to allow nodes to be identified when configuring a system, each node has a 64-bit identifier that should be unique to it, the NUUID. The NUUID is used throughout the RTIO system as an identifier for a node.

A typical NUUID carries three pieces of information.

1. The bottom 16 bits describe the class of the node (referred to as board_type in the code). For example, type 0x0001 is a 64-channel 8-bit-per-channel analogue input board, which will by default be named Palm_*X*, where X is the middle part of the NUUID.
2. The 44 bits from bits 16 to 59 enumerate the nodes of the given board_type, and so in a Hand you will see multiple nodes of type smart_motor or valve_pid_control. This is the part that is substituted for the X above. So, for example, a board number 65537 (0x10001) will be named Palm_1, and board number 131073 (0x20001) is the same type and has name Palm_2.
3. The top four bits are used internally to the RTIO software to make multiple nodes from a single node. This is useful because some boards are most simply described by producing several declared busnodes for each board - for example, the `finger_driver` board is of this type. For these boards, we use the top 4 bits of the node-id to declare "virtual" nodes. The out-of-band state of a virtual node (for example, configuration data) is sent to the underlying real node; the NUUID of the virtual node should never be seen on the bus.

A list of current assigned types can be seen in `include/robot/hand_protocol.h`, and descriptions of them can be found in `rtio/descriptions.c`.

Table 1-1. NUUID

44 bit NUUID		
4 bit	44 bit	16 bit
Virtual node number - should always be `0` on the bus	ID of real node	Class of real node

9.3 Sending and Receiving Messages

A command of the form

```
echo MSG_ID D0 D1 . . . > /proc/robot/bus/bus0/send
```

sends a message from the PC to the CANbus. The values are treated as decimal unless you precede them with 0x to indicate they are hex. The specified message is sent immediately. The file `/proc/robot/bus/bus0/status` shows what messages are being received. This file contains a list of the most recent message of each length and each message ID that has been received.

9.4 Format of Sensor Data on the Bus

The CAN Bus transmits data as a series of individual packets. Each packet contains an 11-bit Message ID, and up to 8 bytes of data (where 0x indicates hex value). e.g.:

```
MsgID - D0 D1 D2 D3 D4 D5 D6 D7
0x048 - 01 03 10 00 00 00 00 00
```

is a CAN message with a Message ID of 0x048, and 8 bytes of data. The least significant byte (D0=01) comes first, and the most significant (D7=00) comes last.

Messages do not have a source or destination. Any node may potentially receive any message, not caring where it came from, only what it contains.

Sensor data is typically packed into 16-bit chunks. So a message containing one such sensor value would look like:

```
0x118 - FB 03
```

This means that the value 0x03FB is sent to sensor location 0x118. Any node interested in that sensor location will take note of that message.

A message containing two sensor values would normally look like this:

```
0x118 - FB 03 29 5A
```

This means that the value 0x03FB is sent to sensor location 0x118, and the value 0x5A29 is sent to sensor location 0x119. It is equivalent to the following two messages:

```
0x118 - FB 03
0x119 - 29 5A
```

9.4.1 Packed sensor data encodings

The touch sensor values are sometimes packed more densely. Six 10-bit values are stuffed into an 8-byte CAN message: the top 8-bits of each value are placed into successive bytes of their own, and the lowest 2-bits of each message are all packed into bytes D6 and D7 of the CAN message, with bits from lower numbered sensors residing in less significant bits.

	D0	D1	D2	D3	D4	D5	D6	D7
Sensor0:	98765432	========	========	========	========	========	======10	========
Sensor1:	========	98765432	========	========	========	========	====10==	========
Sensor2:	========	========	98765432	========	========	========	==10====	========
Sensor3:	========	========	========	98765432	========	========	10======	========
Sensor4:	========	========	========	========	98765432	========	========	======10
Sensor5:	========	========	========	========	========	98765432	========	====10==

Note: D7 bits 7-4 will always be zero.

A CAN Message containing sensor data may contain one to four 16-bit sensor values if the sensor is a 16-bit sensor. For touch sensors, the message always contains 6 sensor values. In this implementation there are no sensor messages with odd numbers of bytes.

9.4.2 Sensors with 2 Hall Effect Devices

Two of the joint sensors have twin Hall Effect units to improve wide angle accuracy. They are located in the wrist and the thumb. Each pair of linked sensors is offset from the other by 90 degrees. They read from the same ring magnet in quadrature.

In order to handle these values, they are both put into their own category of CAN message. Both are packed in sequential messages (in one message), and the values are packed:

```
d0    d1     d2    d3     d4    d5     d6    d7
a0L   a0H    b0L   b0H    a1L   a1H    b1L   b1H
```

Where a0, b0 are the results of the first pair of Hall Effect sensors, and a1, b1 are the results of the second pair of Hall Effect sensors.

Specialist code within hand/rtio/hand_protocol.c parses these sensor values. It unpacks the a,b values for a sensor, and the compound calibration system is used to produce an interpolated position. This is then sent back out on the bus for use by controllers.

9.5 Message Types

The messages that have defined names and numbers in the protocol are at present:

BOOTED `0x48`

Payload: NUUID

When a node wakes up, it sends a BOOTED message containing its NUUID. See the Chapter about NUUID about how to determine which node booted. The node is now in an idle state; the SWITCH_ACTIVE message will start it transmitting, and SWITCH_CONFIGURE will allow it to be configured.

RESET `0x3`

Payload: None / NUUID

A 0-byte RESET message with no data resets all the bus.

An 8-byte RESET message with a 64-bit NUUID resets only the node with matching NUUID.

SWITCH_CONFIGURE `0x14`

Payload: NUUID

The node whose NUUID matched the data of the message is switched into configure mode. That node then acknowledges with a CONFIGURE_CRC message. The node then accepts all transmitted CONFIGURE_DATA messages until SWITCH_ACTIVE, SWITCH_IDLE or SWITCH_NONCONFIGURE is sent. No other node will accept CONFIGURE_DATA messages.

CONFIGURE_DATA `0x16`

Payload: Address, Data

Two bytes of address and up to six bytes of data are sent to the node and used to update the configuration. The Message is acknowledged by a CONFIGURE_CRC message. See the section below on configuration detail. If there are any errors in the data that has been received, then a CONFIGURE_ERROR message will be sent.

CONFIGURE_CRC `0x17`

Payload: CRC32

When SWITCH_CONFIGURE is sent to a node, it sends out this message with the data payload being the crc32 value calculated over the whole configuration of the board. (Note that old firmware might use a different crc algorithm.) Some boards send their node ID instead.

SWITCH_ACTIVE `0x13`

Payload: NUUID

The node specified by the NUUID in the rest of the message is switched into running mode, and so will accept normal command signals and send sensor data at the configured transmission rate. The Message is acknowledged by a CONFIGURE_CRC message.

SWITCH_NONCONFIGURE `0x15`

Payload: NUUID

This message tells the currently configuring node to exit that mode. It sends back a final CONFIGURE_CRC and switches out of configure mode. It does not become active if it was originally idle. The Message is acknowledged by a CONFIGURE_CRC message.

SWITCH_IDLE `0x12`

Payload: NUUID

This message tells the selected node to stop operating normally, and wait for further instructions. The Message is acknowledged by a CONFIGURE_CRC message.

CONIFIGURE_ERROR `0x19`

Payload: 16-bit error value, 0-3 16-bit error subcodes

This message tells the host that an error occurred in the configuration. For some errors, the host will delete that configuration value from the list so it is not retransmitted. The full list of errors can be seen in the firmware header `shadow_protocol.h` or in the RTIO header `hand/include/robot/hand_protocol.h`.

The error code is stored in the per-node protocol data, and can be seen in `/proc/robot/nodes/`*`NODENAME`*`/protocol`: `cat /proc/robot/nodes/smart_motor_3/protocol resets: 0 configs_started: 1 configs_sent: 13 configs_acked: 14 configs_timedout: 0 configs_finished: 2 acks_wrong: 0 any_overflows: 0 self_overflows: 0 overflow_during_other: 0 reboot_during_config: 0 booted: 1 configuration errors: 0 last configuration error: 0 Unknown fault or no error yet flags: CRC is hash`

9.6 Can Node States

Each node has a 64-bit ID number, assigned in the firmware of each board. When the board powers up or resets, it transmits a boot message with its node ID. For example:

```
0x048 - 01 03 10 00 00 00 00 00
```

The above is an example of a 64-bit node ID of 0x0000000000100301

Each board has two orthogonal types of state: Active state and Config state. Each of these states has an opposite, Idle (inactive) and Non Config, therefore there are a total of four possible states.

9.6.1 Active/Idle States

The board may be either Active, or Idle. If the board is in Idle state, it will not sample the ADCs, transmit sensor messages, or turn on any valves. If it is in Active state, it will sample ADCs, and transmit sensor messages if it has been told to. It will also accept valve pulse messages and run controllers.

Active State

To switch to Active State, send a message SWITCH_ACTIVE, with the node ID. e.g.:

```
0x013 - 01 03 10 00 00 00 00 00
```

This can also be achieved by

```
echo active > /proc/robot/nodes/0.100301/state
```

The board will respond by flashing all its lights twice.

Note: State Interaction

Switching to Active State, will also cause the board to switch to Non-Configure state.

Idle State

To switch to Idle State, send a message SWITCH_IDLE, with the node ID. e.g.:

```
0x012 - 01 03 10 00 00 00 00 00
```

This can also be achieved by

```
echo idle > /proc/robot/nodes/0.100301/state
```

The board will respond by flashing all its lights once.

9.6.2 Configuration (Config) States

The board may be in either Config or Non-Config state. In Config State, the board will accept all Configure messages. In Non-Config state it will ignore all Config messages. Only one board can be in Config State at any one time.

Config State

To switch to Config State, send a message SWITCH_CONFIGURE, with the node ID. e.g.:

```
0x014 - 01 03 10 00 00 00 00 00
```

The board will respond by twinkling all its lights twice.

Non-Config State

To switch to Non-Config State, send a message SWITCH_NONCONFIGURE, with the node ID. e.g.:

```
0x015 - 01 03 10 00 00 00 00 00
```

The board will respond by twinkling all its lights once and sending a CONFIGURE_CRC message.

Note: Flashing Lights

Flashing light patterns indicate that the board has changed state. This is useful for debugging. The light patterns happen concurrently with all other functions on the board. The board can continue to run controllers and transmit messages during this time. When the light pattern has finished, the lights will return to their normal functions.

9.6.3 State Transitions

Table 1-2. Transition Table

Message and Payload	State=Idle	State=Active	State=Idle+Config	State=Active+Config
RESET	Reboot	Reboot	Reboot	Reboot
SWITCH_ACTIVE (Our NUUID)	Active	Active	Active	Active
SWITCH_ACTIVE (Other NUUID)	Ignore	Ignore	Ignore	Ignore
SWITCH_CONFIGURE (Our NUUID)	Idle config	Active config	Signal error	Signal error
SWITCH_CONFIGURE (Other NUUID)	Ignore	Ignore	Signal error, reboot	Signal error, reboot
CONFIGURE_DATA	Ignore	Ignore	Config	Config
SWITCH_NONCONFIGURE (Our NUUID)	Ignore	Ignore	Idle	Active
SWITCH_NONCONFIGURE (Other NUUID)	Ignore	Ignore	Reboot	Reboot

9.7 Initialising CAN boards

9.7.1 Resetting

Either an individual node or all nodes on the bus can be reset with a single message. The message

```
0x003 - 01 03 10 00 00 00 00 00
```

will cause the board with NUUID 0000000000100301 to reset. Alternately,

```
echo reset > /proc/robot/nodes/0.100301/state initialises the board.
```

To reset the whole bus, issue command:

```
echo reset  >  /proc/robot/bus/bus0/control
```

or send the message

```
0x003 - with no data.
```

9.7.2 Booting

When a node boots, it sends out a BOOTED message containing the NUUID of the node. E.g:

```
0x48 - 01 03 10 00 00 00 00 00
```

means that node 0x0000000000100301 has just booted.

9.7.3 Configuring

The hand_protocol code receives the boot messages. When the hand_protocol code services the node, it sends out SWITCH_CONFIGURE message with the NUUID of the node, e.g.:

```
0x14 - 01 03 10 00 00 00 00 00
```

As soon as the board is switched into configuration mode, it returns its configuration checksum. e.g.:

```
0x17 - 12 4f 07 11 00 00 00 00
```

The hand_protocol code declares or finds the appropriate busnodes, and sends their configuration to the node. Finally, the message SWITCH_ACTIVE with NUUID is sent to put that node in active mode, and the node begins transmission.

Sending Configure Messages

Configuration messages have the following format:

```
0x16 - LL HH (+config data)
```

0xHHLL is the configuration code, telling the board what configuration is being set. There then follow up to 6 bytes of configuration data.

Config messages can also be set by issuing commands like

```
echo 1 0x020A > /proc/robot/nodes/pressure_manifold_40/config
echo txbase 0x020a > /proc/robot/nodes/pressure_manifold_40/config
```

9.8 Component Configuration

There is a fair amount of configuration data that can usefully be passed around. The current list of well-defined configuration values is pretty short. Because of the nature of the interface through `/proc/robot/nodes/x.y/config`, certain words have well-defined meanings for configuration, and corresponding values are used up. These are:

```
BUSNODE_SET_TX_FREQ=0x101,
BUSNODE_SET_TX_MASK=0x102,
BUSNODE_SET_TX_BASE=1,
BUSNODE_SET_RX_BASE=2,
BUSNODE_SET_LOG_BASE=3,
```

9.8.1 Set Transmit Mask

This configures which ADC channels to transmit.

```
0x0102 + 8-bit or 16-bit value
```

Each bit of the transmitted value, if set, enables a group of sensors to be transmitted. Bit n controls the transmission of sensors 4n..4n+3. e.g.:

```
0x16 - 01 01 0F
```

will enable transmission of sensors 0-15.

9.8.2 Set Transmit Message Base Address (txbase)

Sets the lowest bus address that the node transmits at.

```
0x0001   +16-bit value
```

e.g. for the Palm boards

```
0x16 - 01 00 D0 02 00 03
```

will set the sensor base address to 0x2D0 and the Fingertip touch sensor base address to 0x300 for the Pressure Module.

```
0x16 - 01 00 00 03
```

will set the sensor base address to 0x300. Sensor channels 0-3 will be transmitted on a Message ID of 0x300, Sensor channels 4-7 will be transmitted on a Message ID of 0x304, Sensor channels 8-12 will be transmitted on a Message ID of 0x308 etc.

The txbase value can also be set in a more readable form: `echo txbase 0x030002c0 > /proc/robot/nodes/palm_sensor/config` will set the Palm base to 0x2c0 and the Touch Sensor base to 0x300.

9.8.3 Set Receive Message Base Address (rxbase)

Sets the lowest bus address that the node receives data at.

```
0x0002    + 16-bit value
```

e.g.

```
0x16 - 02 00 18 01
```

Message ID of 0x118 will control valve 0, Message ID of 0x119 will control valve 1 etc.

```
echo rxbase 0x0240 > /proc/robot/nodes/valve_pid_control_3a/config
```

9.8.4 Set Transmit Delay Multiplier (rate)

Sets the transmission rate or rates for the node.

```
0x0101    + 8-bit value
```

e.g.

```
0x16 - 01 01 04
```

```
echo rate 0x04 > /proc/robot/nodes/pressure_manifold_40/config
```

If the multiplier is zero, no transmission occurs.

If multiple bytes are sent, each byte is a multiplier for a separate item. For example for the Palm boards, there are five values, one for each fingertip, and one for all the joint positions on the hand. The Palm board alternates between sending a joint data message and sending the next finger's data message. The multipliers adjusts the rates of each finger and the joint sensors separately. The default setting for the Hand is 08 08 08 08 08 02 - so every other time the transmit check is done, it sends joint sensor data. Every 8 times it checks a given finger, data from that finger is sent.

```
echo rate 0x080808080801 > /proc/robot/nodes/palm_sensor_4/config
```

tells the palm node to set the joint sensors to maximum transmit rate, and the fingertip sensors to transmit 8 times more slowly.

9.8.5 Add controller (control)

When a valve driver board is in Config State, it will also accept messages to add PID (Proportional Integral Derivative) Controllers. These take the following form:

```
      0x016 - 0x19 N P I 0 F S T
          0x19: Specifies that this configure message contains a
controller
                  N:    Controller Number.      [0..19]
                  P:    Proportional Gain Value [-128..127]
                  I:    Integral Gain Value     [-128..127]
                  D:    Derivative Gain Value [-128..127]
                  F:    Flags
                          Bit 7: reserved, write 0
                          Bit 6: reserved, write 0
                          Bit 5: Enable Bit.
                                  Write 1 to activate controller.
                                  Write 0 to disable
                          Bit 4-0: [0..19] Deadband is set to 2^this.
                  S:    Sensor channel number.
                  T:    Target channel number.
```

For sensor and target channel numbers: If the board has valve base 0x218, then the block is 0x200..0x2FF

S and T channel numbers are sensor values offset within the block.

When a message within a block is received, it is stored by the Valve Driver board. Then when the controller runs, it looks up the value.

Setting the Deadband value establishes a magnitude of error that will be ignored by the controllers. This helps reduce un-necessary valve operation and so reducce noise, improve power and air consumption, and increase valve life.

Controllers may be added or removed at any time while the device is in Config State. If the device is also in Active State, then the controllers will begin running immediately.

The contrlr command

A simpler method of modifying the controllers is to use the **contrlr** command. This takes the form

```
    contrlr -e FF3_Flex_Fill sensor FFJ3_Pos target FFJ3_Target p 64 i 12
d 4
```

to set up and enable a position controller running on FFJ3 taking target values from the sensor FFJ3_Target, with P value +64, I value +12, and D value +4.

The controller will be run every 4.5ms, updating the valve on/off ratio. The valve handling code is run every 0.25ms, giving very precise fast control of the valve opening times.

9.8.6 Adjust Valve Switching Speed (control)

```
    0x0103    + 8bit value:
```

The valve switching rate may be adjusted, to allow for different valves. This only applies to valves with controllers running on them. The default speed is set on Reset. This value cannot be set above 64 and must not be set below 8.

```
    0x16 - 03 01 06
```

This sets the valve speed to its default value. Higher numbers give slower switching speeds.

A command can also set a controller configuration word for a node, for example

```
    echo control 0x914 0x12356789abc >
/proc/robot/nodes/valve_pid_control_60/config
```

sets up controller 9 on node valve_pid_control_60.

9.8.7 Standardised "echo" commands

More recent versions of the RTIO code have implemented a pair of interface forms to the configuration values. These are all set with commands either of the form of:

```
echo motor_sensor 0x00112233 > /proc/robot/nodes/X/config
echo motor_sensor 0x2233 0x0011 > /proc/robot/nodes/X/config
```

Note that these two commands both send 4 bytes of data!

or of the form of

```
echo pid_sensor 0 0x00112233 > /proc/robot/nodes/X/config
```

```
echo pid_sensor 0 0x2233 0x0011 > /proc/robot/nodes/X/config
```

The first form sets a simple value on a single board. The second form sets one value of a set of values on a board - as when the board implements multiple controllers.

The current list of simple values can be found in `hand/rtio/proc.c`, but a recent list is below.

Configuration using simple names

motor_setp

One 16-bit value is required. This value selects the bus sensor address that is used to provide the position control set point for a smart_motor unit. Address 0 is treated as being hard-wired to 0.

```
echo motor_setup 0x123 > config
```

selects CAN message ID 0x123 as the value to use for the position set point.

motor_pid1

Three 16-bit values are required. They set the Proportional, Integral, and Derivative gain for a smart_motor unit's inner (force) PID control loop.

```
echo motor_pid1 0x1000 0x0800 0xf000 > config
```

sets the force P value to 4096, the I value to 512, and the D value to -4096.

motor_pid2

One 16-bit value, one 16-bit value bits, and one 8-bit value, set the maximum value for the Integral accumulator, the force safety cut out threshold, and the output shift for a smart_motor unit's inner (force) control loop.

```
echo motor_pid2 0x1000 0x0400 0x01 > config
```

Sets the maximum accumulator value to 4096*256, the maximum force to 1024, and the output scaling shift to 1.

motor_pid3

Two 16-bit values set the deadband and output offset value for a smart_motor unit's inner (force) control loop. If the input error is less than +/- deadband, then the input error will be treated as 0. The output offset is used to move the output away from zero once it is non-zero, so an output of +64 will become offset+64, and an output of -64 will become -64-offset.

```
echo motor_pid3 0x0040 0x00c0  > config
```

sets the deadband for the input sensor error to 64, and the offset for the output to 192.

motor_maxforce

One 16-bit value sets the maximum output force value for a smart_motor unit's inner (force) control loop.

```
echo motor_maxforce 0x1000   > config
```

sets the force limit to 4096.

sensor_pid1

Three 16-bit values are required. They set the Proportional, Integral, and Derivative gain for a smart_motor unit's outer (usually position) PID control loop.

```
echo sensor_pid1 0x1000 0x0800 0xf000 > config
```

sets the position P value to 4096, the I value to 512, and the D value to -4096.

sensor_pid2

One 16-bit value, one 8-bit value, 8 un-used bits, and one 8-bit value, set the maximum value for the Integral accumulator, the scaling for the thermal safety cycle, and the output shift for a smart_motor unit's outer (usually position) control loop.

```
echo sensor_pid2 0x1000 0x02 0x00 0x01 > config
```

Sets the maximum accumulator value to 4096*256, the thermal safety cycle to 1, and the output scaling shift to 1. The value 0x00 is unused.

sensor_pid3

Two 16-bit values set the deadband and output offset value for a smart_motor unit's outer (usually position) control loop. If the input error is less than +/- deadband, then the input error will be treated as 0. The output offset is used to move the output away from zero once it is non-zero, so an output of +64 will become offset+64, and an output of -64 will become -64-offset.

```
echo sensor_pid3 0x0040 0x00c0  > config
```

sets the deadband for the input sensor error to 64, and the offset for the output to 192.

motor_strain_amp0, motor_strain_amp1

Both of these configuration values affect the processing of the strain gauges by the amplifier. Three 8-bit values are provided, gain1, gain2, and ref. If the value of gain1 is not 255, then gain1 and gain2 are used to program the gain settings of the AD8556, and ref is used to set the output offset for the AD8556. If the value of gain1 is 255, then automatic tracking is enabled on both amplifiers.

txfreq, rate

One or more bytes is sent to set transmission frequency for groups of sensors on the board. How many bytes are needed will be dependent on the exact hardware of the board. The value sent is used as a frequency divisor for the transmit frequency, with zero disabling transmission entirely.

```
echo txrate 0x040100 > config
```

will set one sensor group to transmit at a quarter the rate of the second group, and disable the third group entirely.

txbase

One or two 16-bit values select the lowest addresses for each group of transmitted data from the board. For most boards only one value will be required.

```
echo txbase 0x3c0 > config
```

Sets the board to transmit sensor data starting at address 0x3c0.

The complex values are:

pid_pid

Three 16-bit values are required. They set the Proportional, Integral, and Derivative gain for the PID control loop.

```
echo pid_pid 1 0x1000 0x0800 0xf000 > config
```

sets the P value to 4096, the I value to 512, and the D value to -4096 for controller number 1 on the device.

pid_imax

Two 16-bit values and one 8-bit value are required. They set the maximum value for the Integral accumulator, the maximum value to be output to the hardware and the output shift for the PID controller specified.

```
echo pid_imax 3 0x1000 0x2000 0x01 > config
```

Sets, for controller 3, the maximum accumulator value to 4096*256, the maximum output value to +/- 8192, and the output scaling shift to 1.

pid_dead

Two 16-bit values set the input deadband and output offset for the specified controller. If the input error is less than +/- deadband, then the input error will be treated as 0. The output offset is used to move the output away from zero once it is non-zero, so an output of +64 will become offset+64, and an output of -64 will become -64-offset.

```
echo pid_dead 2 0x0040 0x00c0  > config
```

sets, for controller 2, the deadband for the input sensor error to 64, and the offset for the output to 192.

pid_sensor

Two 16-bit values set the sensor and setpoint channels to be used by the controller. Each is given in terms of a bus message ID. If the ID 0 is used, then the value is hard-wired to zero.

```
echo pid_sensor 0 0x03f0 0x0000  > config
```

sets, for controller 0, the sensor being controlled to message ID 0x3f0, and ensures that the setpoint is permanently set at zero.

9.8.8 Sending arbitrary configuration values

Sending out arbitrary configuration values can also be done. For example, the frontend program for the finger board does this. It opens the file `/proc/robot/nodes/x.y/config`, and then prints data into it. This data is provided as a 2-byte address and a 6-byte data value. The exact ascii representation of the 6-byte data is used to determine the number of bytes actually sent, so `0x119 0x001122` sends out a configure message with 3-bytes of data, and `0x119 0x11223344` sends out one with 4-bytes of data.