# iRobot Dexter Hand Owner's Manual

*8/7/2013*

# Contents


Overview ..... 3

Versions ..... 3

Quick Start ..... 4

Differences from the ARM-S Training Session ..... 5

Info on the Gumstix ..... 5

Conventions ..... 6

- Accelerometer Axes: ..... 6
- Finger Tactile Sensor Numbering: ..... 8
- Palm Tactile Sensor Numbering: ..... 9
- Firmware Numbering ..... 10

ROS Stack ..... 10

Sensing ..... 11

Control ..... 11

Handle Controller ..... 11

- Command line switches ..... 12
- Compiling on the Overo ..... 12

Firmware ..... 13

Parameters ..... 13

Utilities ..... 15

Calibration ..... 16

- Sensors ..... 16
- Utilities ..... 19

Using the GUI ..... 21

Using the “Annan Device” ..... 22

Testing ..... 22

Cautions ..... 23

Troubleshooting ..... 23

Support ..... 23

Appendix ..... 24

- Diagnosing and Fixing an Unresponsive Part of the Hand ..... 24
- Switching Between Smart and Dumb Fingers ..... 25

## Overview

The hand has a Gumstix Overo Earth inside that is the point of contact between the hand's microcontrollers and your controlling computer. Communication to and from the hand is over a TCP or UDP socket. A C library is provided for this. Additionally, we have provided a ROS wrapper for convenience.

Note, early iterations of the hand were named HANDLE. Therefore most of the code has this naming convention.

## Versions

There are 2 versions of the Dexter hand; an ARM-S version and a DRC version. They are easily identifiable based on the back housing. The ARM-S hand has a 3D printed plastic back housing with ventilation holes, while the DRC version has a machined aluminum back housing with etched logos. Additionally, the ARM-S hand has a Barrett WAM arm mount, and the DRC hand has a BDI Atlas hand mount. See images below:

ARM-S Hand


DRC Hand

Here are the notable differences between the hand versions:

The DRC hand is more environmentally sealed. It has sealed housings (expect for tendon entry points), and the cooling fan has been removed.

The DRC hand is more robust. It has an aluminum back housing. The smart fingers use rigid flex PCBs.

The DRC smart fingers have an added proximal accelerometer.

The finger spread control is stronger and more reliable on the DRC hand. It also has speed control as opposed to bang-bang control.

Some internal status LEDs and connectors have been moved between the two versions.

Note: Smart fingers between the two versions cannot be interchanged. The pinout on the ribbon cable is different.

## Quick Start

For ARM-S hands: power on the hand by providing it 48 Volts. Steady state current when not driving motors is around 250 mA. Peak draw when stalling motors and get up to a few Amps.

For DRC hands: power the hand with 24 volts.

Plug in an Ethernet cable from the hand to your computer.

For ARM-S hands: Note: you should use an Ethernet cable with no strain relief jacket around the plug.

Put your computer on the same subnet as the hand with a command like:

```
sudo ifconfig eth0:0 192.168.40.2
```

Or add to /etc/network/interfaces:

```
auto eth0:0
iface eth0:0 inet static
address 192.168.40.2
netmask 255.255.255.0
```

Download source code from: `svn://armtestsite.com/irobot_armh`

Add the handle directory to your `ROS_PACKAGE_PATH`.

Compile ROS stack with: `rosmake handle -s -k --profile --pre-clean`

Run ROS interface: `rosrun handle_launch control.sh <Hand#>`
Where "`<Hand#>`" is the last part of the hand's IP address. For example "63" if the hand's IP address is 192.168.40.63. Note that if you don't have the "Annan device" connected, you can kill the `/annan_control` node.

To view the raw sensor data: `rostopic echo /right_hand/sensors/raw`

To move a finger:

```
rostopic pub -1 /right_hand/control handle_msgs/HandleControl
    '{type: [2, 0, 0, 0, 0],
     value: [1500, 0, 0, 0, 0],
     valid: [True, False, False, False, False]}'
```

To visualize hand in rviz:

```
rosrun handle_launch visualize.launch
rosrun rviz rviz
```

Then in rviz:
Set the fixed frame to `/base_link`

Enable the “Robot Model” display with Robot Description: “robot_description”
Enable the “Marker Array” display on Topic: “visualization_marker_array”

To view plots of sensor data: `rosrun handle_ros graph_*`

Some of these scripts require you to also pass in the zero-based finger number. For example, to view the first 8 tactile sensors on finger 1’s proximal link, run:
`rosrun handle_ros graph_F_proximal_tactile.sh 0`

## Differences from the ARM-S Training Session

The hand’s IP addresses are now in the `192.168.40.*` subnet instead of `192.168.33.*`. This change is to make the hands networking easier with the rest of the ARM-S system.

Due to the possibility of the robot having two iRobot hands on the robot at the same time, the launch scripts now remap the ROS topics to `/right_hand/*` and `/left_hand/*` instead of `/handle/*`.

The SD cards are now read-only by default. This can be changed by running a command on the Overo.

There is a longer bootup time for the software on the Overo.

## Info on the Gumstix

The hands are configured to have a static IP address: `192.168.40.XX`. Where XX is the hand’s serial number clearly labeled on the side of the hand. See Quick Start section for info on configuring your networking.

Log into the hand as “root” with password: “awareroot”.

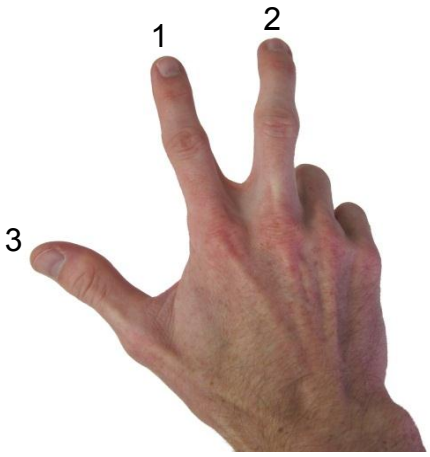
The software on the Overo is configured to run when the hand boots up. However, there is a delay of 16 seconds to avoid stepping on the bootloaders that run when the microcontrollers are powered up. The script that controls this can be found at: `/etc/init.d/handle_control.sh`. See the “Handle Controller” section for more info.

To avoid SD card corruption, the drive is mounted as read-only during the boot process. To make the drive read-write, execute “makerw” after logging into the hand. IMPORTANT: remember to execute “makero” to make the drive read-only before powering down the hand.

The hand is running a Linux variant called CommonOS. We are not providing the build tool-chain for this, however you should be able to scp code onto the Overo and compile natively. Also, you should be able to download additional packages with the “opkg” routine (similar to “apt-get”). The hand must be plugged into a network port that will give it a DHCP address. And you will have to reconfigure the networking on the Overo to get this DHCP address. IMPORTANT: you must keep an alias with a static IP address or you will not be able to log into the hand anymore. If you break the networking on the hand, the hand must be returned to iRobot to remove the SD card and reset the networking.

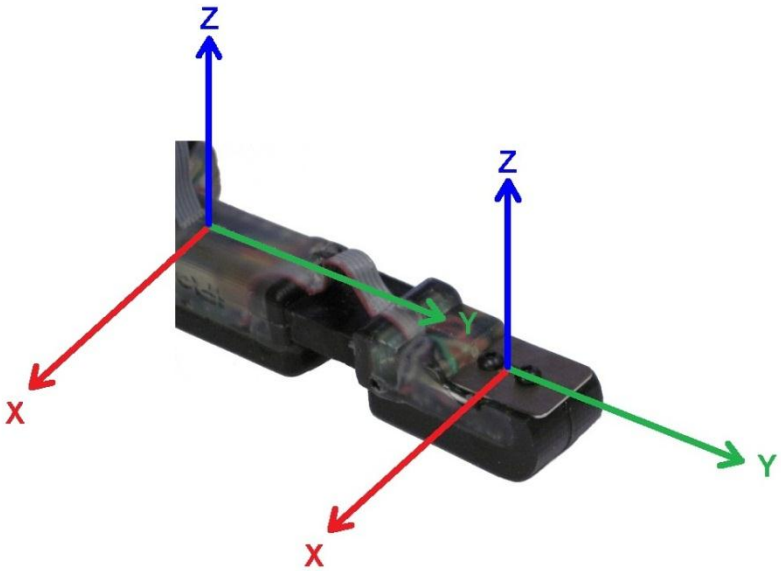
## Conventions

The finger numbering is numbered as if the hand is your right hand, with 1 being your index finger, 2 being your middle finger, and 3 being your thumb. Motor 4 is the thumb antagonistic motor, and motor 5 is finger spread motor. This numbering applies to both controlling the motors as well as interpreting sensor data. See the HandleSensors.msg definition in the `handle_msgs` package for more info on sensor units and ranges.



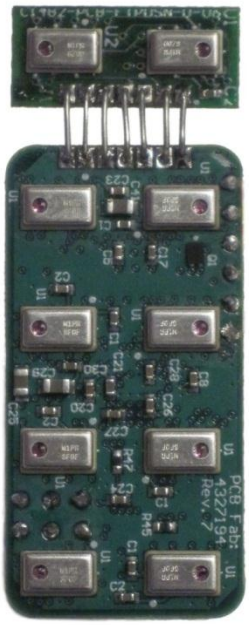


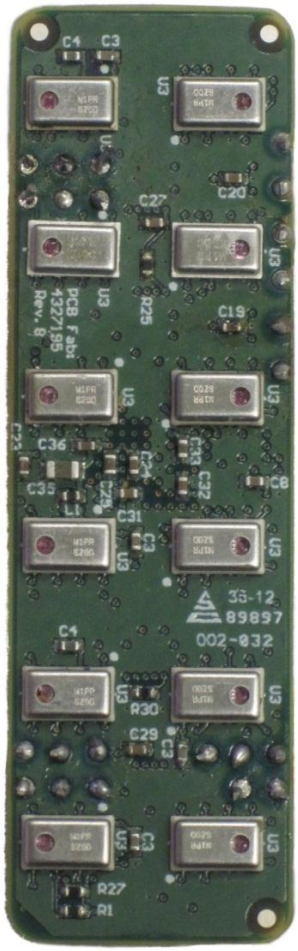
## Accelerometer Axes:



Note: ARM-S hands don’t have a proximal accelerometer.

## Finger Tactile Sensor Numbering:

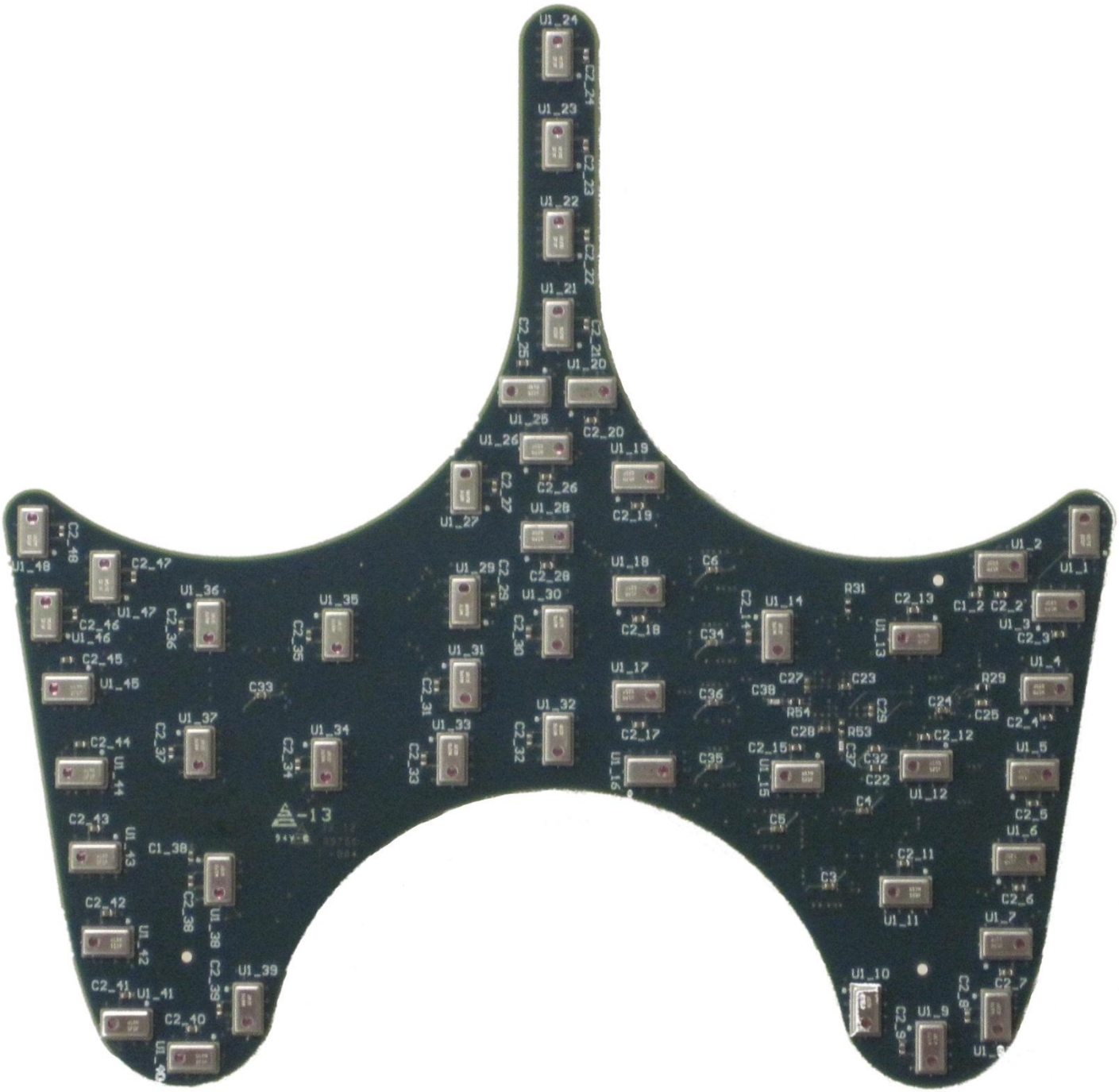


Distal Board Proximal Board

(Up is towards fingertip in both images.)

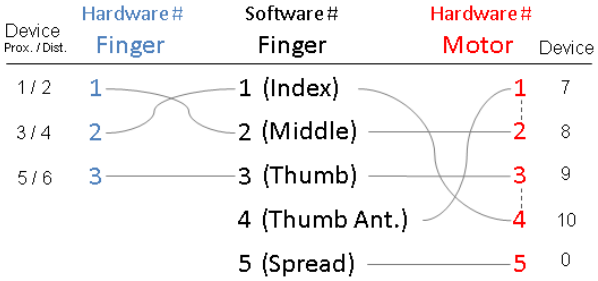
## Palm Tactile Sensor Numbering:

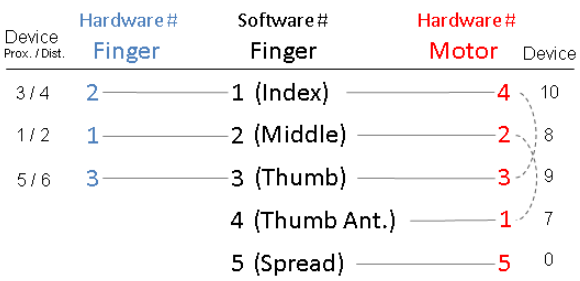
The sensors are numbered U1_X where X is the sensor number from 1 to 48. Sensor 1 is in the upper right of the image.

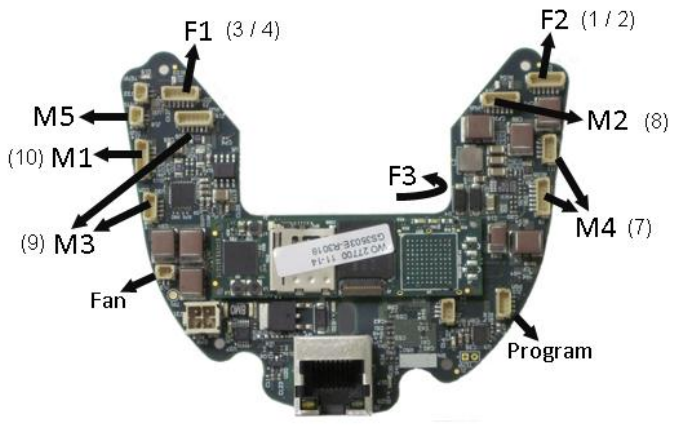


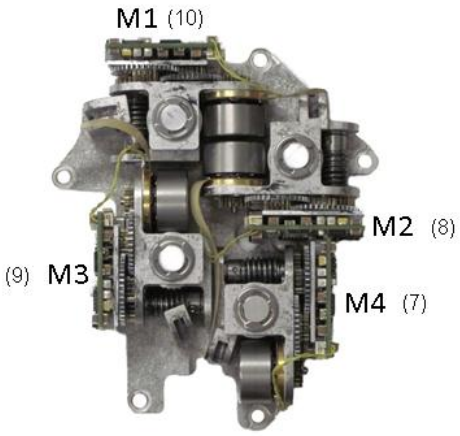
## Firmware Numbering

You will most likely not have to deal with this, but if the need arises to run some of the low-level utilities, the finger and motor numberings in the firmware differ from the above conventions. The table below summarizes the conversion between hardware (firmware), and software (post handle_lib) numbering.




Note: DRC hand connectors will be in slightly different locations.

## ROS Stack

- `handle_ros`: The HANDLE interface node. This is the node that interacts directly with the hardware.
- `handle_msgs`: Definitions for all HANDLE topics and services.
- `handle_lib`: Common headers for handle_controller, handle_ros, annan_control, and gui_control.
- `handle_controller`: The code that runs on the Overo on the hand. There are also some helpful utilities that run on the hand.
- `keyboard_control`: Simple control using the keyboard of the 5 motors.
- `annan_control`: Controller for the Annan 5 slider device.
- `gui_control`: A QtGUI that mimics the Annan controller. This used to work in ROS Electric, but I have been unable to compile it in Fuerte.
- `handle_collisions`: Sensor processing nodes for the HANDLE hand.

- `handle_description`: urdf description of the HANDLE hand.
- `handle_launch`: launch scripts.
- `handle_sensors`: Sensor processing functions for the HANDLE hand.
- `joint_publisher`: Convert sensor publications to urdf joints.

## Sensing

Please see `handle_msgs/msg/HandleSensors.msg` for info on the units and ranges of the sensors. Some notes:

- The `motorHallEncoder` range will vary depending on how much tendon is wound on the spindle.
- The temperature sensor for the spread motor is not populated so will not have a meaningful value.
- The current sensor for the spread motor is not populated so will not have a meaningful value.
- There are a few other sensors that are read by the low-level firmware, but are not passed all the way up to the ROS layer.

## Control

Please see `handle_msgs/msg/HandleControl.msg` for info on how to control the hand. Some notes:

- The spread motor only supports velocity and position mode.
- For ARM-S hands: The spread motor only has on/off control.
    - Position mode is implemented as bang-bang control with a large deadband.
    - In velocity mode, only the sign of the desired velocity is inspected.
- For DRC hands: the spread has speed control and a proper proportional control. However the gain is very high.
- Even slow velocities can command large forces in the four tendon motors. Because there is a PID loop trying to match the desired velocity, so if the motor stalls, the current will ramp up quickly.

## Handle Controller

As previously mentioned, the `/etc/init.d/handle_control.sh` script is run on boot to start the `handle_controller`. This is the piece of code that handles the low-level interface with the microcontrollers in the hand. While in theory this code is not complicated, simply forwarding messages between the microcontroller API and C structs, a number of factors make this code more complex. Some complicating factors:

- Sensor messages arrive asynchronously
- Different numbers of microcontrollers can be included in the sensor messages

- Motors may fail to acknowledge a command (regardless of the command's actual success)
- We can only send one low-level motor command at a time. So we iterate through a queue of motor commands
- Safety features such as high-level thermal protection of the motors
- Debugging and status information

`handle_controller` along with all the utilities are kept in the root user's home directory: `/home/root`. This is simply for convenience.

## Command line switches

Executing "`handle_controller --help`" will display the list of command line flags available. Also summarized here:

```
usage: handle_controller [options]
options:
  -r, --rate, --hz <hz> Set sensor rate in hz.  Default: 100
  -p, --port <port>     Set the TCP/UDP port number.  Default: 3490
  -c, --calibrate       Run a calibration routine on startup.  Default: false
  -d, --dumb            Hand has "dumb fingers", don't ask for finger data.
                        Default: false
  -u, --udp             Use UDP instead of TCP.  Default: false
  -o, --timeout <secs>  Set this control rate timeout.  Default: 5.000000
                        <secs> is a float.  Set <= 0 to disable.
                        Motors are stopped when timeout triggers.
  -t                    Debug: print measured rates.  Default: false
  -m                    Debug: print motor info.  Default: false
  -e                    Debug: print error codes.  Default: false
  -h, --help            Display this help message and exit.  Default: false
  -v, --version         Display version string and exit.  Default: false
```

## Compiling on the Overo

While we are not providing the CommonOS toolchain to cross-compile on your machine, you can simply push the code onto the Overo and compile natively. To do so:

1. Loginto the hand
2. Make the file system read/writeable by executing "`makerw`"
3. scp the code to the hand
    a. To recompile the controller and utilities, you will need both the handle_controller and handle_lib directories. It will be easiest to keep these directories intact and maintain their relative positions. (i.e. they should be next to each other).
4. To recompile handle_controller:
    a. cd into handle_controller/src
    b. run "make"
    c. FYI: If you look at the Makefile, you will notice that g++ is named "arm-commonos-linux-gnueabi-g++". This is simply an alias for g++ on the Overo.

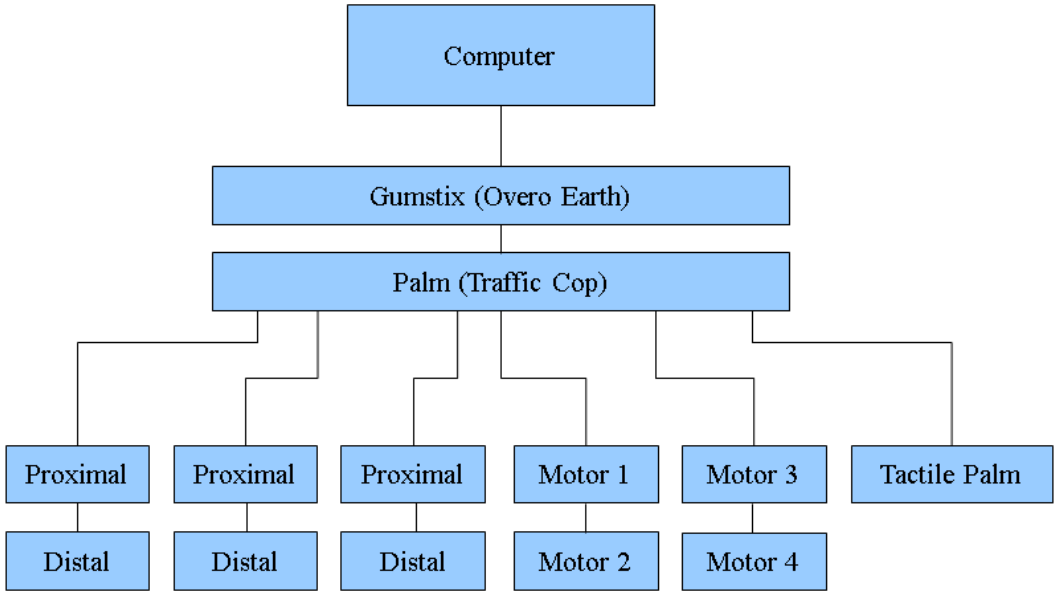
d. Then copy the executable into /root/home to use it.
e. You may want to back up the original version first.
f. If handle_controller is currently running, the copy will fail.

5. To recompile the utilities:
   a. cd into handle_controller/utils
   b. run “make”. This will compile all utilities.
   c. Then copy the executables into /root/home
6. Note that if you change some of the header files in handle_lib you will likely need to recompile both handle_controller and utilities.

## Firmware

The firmware source code for the microcontrollers has been provided to you as a reference. There are bootloaders on all the microcontrollers that allow the firmware to be updated. If a bug is found, or there is a significant demand for a new feature, it is possible that we will provide you with updated hex files and instructions on how to update the firmware.

An appendix documenting the low-level communications between the microcontrollers can be made available upon request.




## Parameters

There are number of EEPROM parameters that reside in the microcontrollers. See table below.

| Parameter | Valid for | Description | Default Value | Encoding |
|---|---|---|---|---|
| 0 | Motors | Proportional PID constant for Torque control | 0.03 | float |

| | | | | |
|---|---|---|---|---|
| 1 | Motors | Integral PID constant for Torque control | 0.01 | float |
| 2 | Motors | Derivative PID constant for Torque control | 0.0 | float |
| 3 | Motors | Proportional PID constant for Velocity control | 0.005 | float |
| 4 | Motors | Integral PID constant for Velocity control | 0.0025 | float |
| 5 | Motors | Derivative PID constant for Velocity control | 0.0 | float |
| 6 | Motors | Proportional PID constant for Power control | 1.0 | float |
| 7 | Motors | Integral PID constant for Power control | 0.3 | float |
| 8 | Motors | Derivative PID constant for Power control | 0.0 | float |
| 9 | Motors | Motor winding electrical resistance at 25 °C ($\Omega$) | 28.6 | float |
| 10 | Motors | Thermal resistance winding to case (°C) | 2.66 | float |
| 11 | Motors | High temperature trip point (°C) | 65.0 | float |
| 12 | Motors | Low temperature trip point (°C) | 65.0 | float |
| 13 | Motors | Thermal time constant of motor windings (ms) | 1777.0 | float |
| 14 | Motors | Absolute maximum motor temperature (°C) | 85.0 | float |
| 15 | Motors | Temperature coefficient of copper($°C^{-1}$) | 0.0039 | float |
| 16 | Motors | Off time required for thermal equilibrium($\tau_W$) | 5.0 | float |
| 17 | Motors | Target temperature for Power Control (°C) | 65.0 | float |
| 18 | Motors | Maximum Allowable RPM (RPM) | 12000.0 | float |
| 19 | Motors | Reserved | 402.0 | float |
| 20 | Motors | Maximum Allowed Voltage Command (255 max) | 150.0 | float |
| 21 | Motors | Proportional PID constant for Position control | 15.0 | float |
| 22 | Motors | Deadband for position control | 25.0 | float |
| … | … | … | … | |
| 26 | Palm | Proportional PID constant for spread motor Position control | 500 | int |
| 27 | Palm | Deadband for spread motor control | 5 | int |
| 28 | Proximals, Palm | Encoder Offset | [Finger dependent] | int |
| 29 | All | Firmware Version | [Device dependent] | int |

| | | | | |
|---|---|---|---|---|
| 30 | All | ID | [Device dependent] | int |
| 31 | All | LED state | 1 | bool |

Note the state of the finger LEDs are controlled via this EEPROM parameter. So if you don't want the LEDs on, you can set this parameter to 0. This commands will accomplish this:
`./parametertest2 s 31 0 a`
See the Utilities section for more info.

## Utilities

There are a number of low-level utilities on the Overo that can be used for debugging purposes. Before using any, you must first kill the handle_controller program. This can be done the traditional way by running "`top`" or "`ps`", getting the PID number, then calling kill. Or you can use the "`./killhandle.sh`" script. This script will return nothing if successful, but warn you if it didn't find anything to kill. Note that due to the long sleep before handle_controller starts, it is possible to log into the hand, and try to kill handle_controller before it actually starts. So you may want to wait about 20 seconds before logging into the hand.

After killing handle_controller, you should ensure that the serial port is cleaned up and ready to use. The easiest way to do this is to simply run the "`./stopall`" utility. Make sure that the commands it executes don't return 0xFFFFFFFF. Simply run again until all commands return a small integer.

Here is a list of the compiled utilities:

- `bootload`: put a microcontroller into bootloader mode for 15 seconds.
- `calibrate_opcode`: send the calibration opcode to a microcontroller for a specific sensor.
- `chainmask`: set the chainmask in the traffic cop. The chainmask tells the traffic cop to ignore some microcontrollers to achieve faster data rates.
- `debug_poll`: display the number of checksum errors in all the microcontrollers.
- `get_sensor`: poll once for a sensor. Suitable for use in scripts.
- `ledflash`: flash the led on the palm for a while.
- `motortest`: send a low-level motor command to a specific motor.
- `parametertest`: get/set 1 or all parameters from a microcontroller.
- `parametertest2`: get/set multiple parameters from multiple microcontrollers.
- `raw_spin`: like `sensor_spin`, but displays raw packets.
- `sensor_poll`: poll for a sensor at about 2 Hz.
- `sensor_spin`: put palm in sensor spin mode and display all sensors.
- `stopall`: stop sensor spinning and stop all motors.
- `test_motor`: low-level verification of motor board working.
- `test_palm_tactile`: low-level verification of tactile board working.

Here is a list of the helper scripts:

- `breakinspread.sh`: exercise the spread motor
- `calibrate_all.sh`: calls `calibrate_opcode` on every sensor on every microcontroller.
- `firmware-up-not-palm`: shorthand to call `avrdude` to flash firmware onto a avr32xmega microcontroller.
- `firmware-up-palm`: shorthand to call `avrdude` to flash firmware onto the palm avr128xmega microcontroller.
- `flash_all.sh`: updates firmware on every microcontroller.
- `flash_distals.sh`: updates firmware on the distal microcontrollers.
- `flash_fingers.sh`: updates firmware on the proximal and distal microcontrollers.
- `flash_motors.sh`: updates firmware on the motor microcontrollers.
- `flash_palm_firsttime.sh`: updates firmware on the palm microcontroller when it does not have any code on already.
- `flash_palm.sh`: updates firmware on the palm microcontroller when it currently has code.
- `flash_proximals.sh`: updates firmware on the proximal microcontrollers.
- `flash_tactile.sh`: updates firmware on the tactile microcontroller.
- `get_firmware_versions.sh`: print the firmware versions for all microcontrollers.
- `iflash.sh`: This is probably the only utility you will need to flash the microcontrollers.
- `killhandle.sh`: stop the `handle_controller` program.
- `testscript.sh`: move the hand through a few pre-programmed motions.

# Calibration

There are a number of different sensors on the hand that require different kinds of calibration. As a result, there are three different calibration utilities. Below is a description of the different sensors and their calibration requirements. The next section has descriptions of the various utilities that can be used for calibration.

## Sensors

***Proximal pin joint***: This is an absolute encoder that is factory calibrated to read zero when the finger is lying flat. A simple offset is stored in the proximal microcontroller's EEPROM. This value remains across power cycles and re-tendoning. You will probably not need to recalibrate this sensor. Note that due to the way the finger cable is routed, it is possible that the zero location is not exactly level with the palm. Use the "`calibrate_opcode`" utility to calibrate this sensor, see below.

***Finger spread***: This is an absolute encoder similar to the proximal pin joint. Its offset is similarly stored in the palm microcontroller's EEPROM. And again, the "`calibrate_opcode`" utility is used to set this. However, through a gear reduction, it travels more than half of the encoder's range. So when the hand is started up, it must be within 512 ticks (60 degrees) of the actual zero configuration. However, this requirement can be bypassed with a calibration routine in the `handle_controller`, see below.

The included calibration jig can be used to calibrate this joint. Stops on the rig stop the fingers at 0 and 90 degrees. However, note that the 0 degree stop may be a little too small, resulting in fingers that are not parallel.

***Motor tendon cable excursion***: These are incremental encoders that zero when the hand is powered on. We typically keep the zero point such that the finger can lay flat with no tension on the cable. And the zero point for motor 4 (thumb antagonistic) is such that the finger can move through its entire range without needing to move motor 4. This is typically set by closing finger 3 all the way with motor 3, then reeling in motor 4 until the finger just starts to move back. The `handle_controller` calibration routine will do this automatically, see below.

***Distal flexure joint***: This sensor requires the most amount of calibration. Currently, there are three different methods you can use (or combine) to estimate the position of the distal link.

Method 1: flexure joint sensors. This sensor has limited range but measures the most information about multi-DOF flexure deformation. It consists of 4 pairs of IR receivers. Due to manufacturing variability, the values and ranges of the sensors can vary greatly and are calibrated using linear regression. The idea behind this method is to measure flex and twist at each end of the flexure, and interpolate them into a series of 9 2DOF joints. Use the `calibrate_joint_sensors.py` node in the `handle_sensors` package. This code will walk you through the calibration for this sensor method. The resulting parameters are then saved in the finger .yaml files.

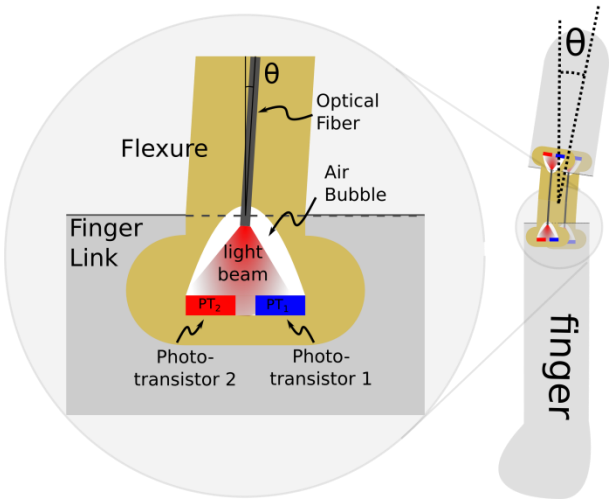



Figure 1: Flexion Sensor

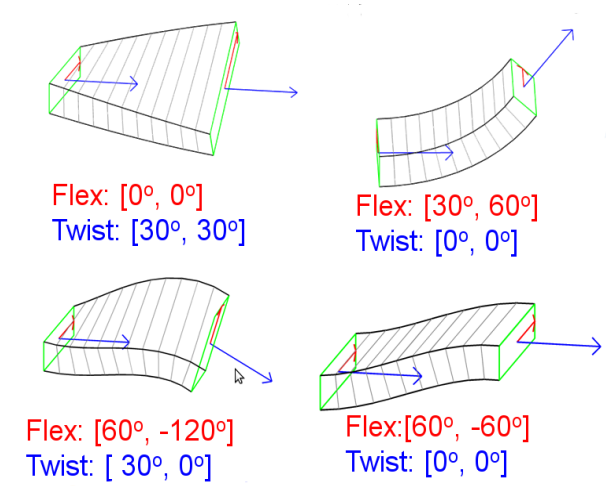



Figure 2: Deformations measured by Flexure Sensor

Method 2: motor tendon cable excursion. By using the proximal pin joint in combination with the motor tendon cable excursion it is possible to determine the angle of the distal link relative to the proximal link. This does not give you information on finger twist, or if the finger is bent under external load but gives the most accurate flexion measurement under internal loads.

Method 3: using the distal accelerometers directly. This is the most straightforward and direct method but does not measure angles around the gravity vector (i.e. if the hand is sideways). Also, it requires additional information, namely the orientation of the hand relative to gravity and the proximal pin joint encoders or accelerometers. As there is no accelerometer in the palm base, this information must come externally (e.g. from the forward kinematics of the arm).

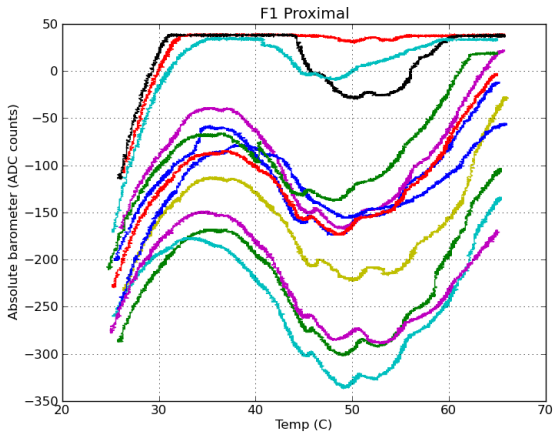
Currently, the firmware and lower levels of the C API simply pass on the 8 raw sensor values. Further processing is handled in the higher levels of the provided ROS interface. (In `JointFlexible.py` which is used by the `sensors_publisher.py` node to be exact). Currently, `JointFlexible.py` uses the flexure sensors to estimate finger twist, and tendon excursion to estimate finger flexion.

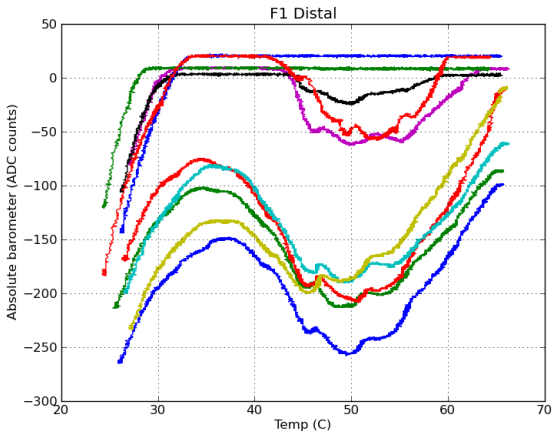
***Tactile sensors***: These sensors do not require any permanent calibration. However, some sensors will read higher or lower than the others. So a calibration service is provided:

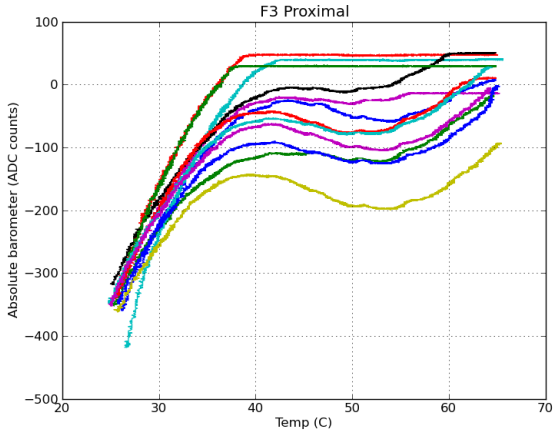
`/handle/events/sensors/calibrate`

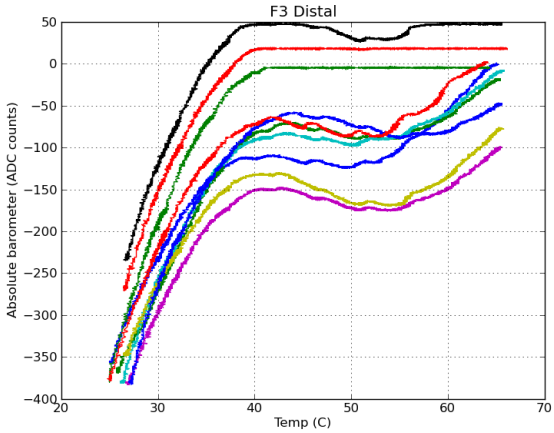
that sets a zeroing offset which is subtracted from the values in sensors/raw before they are published to /sensors/calibrated. This can be called after the hand is booted and during manipulation when the hand is not in contact.

The tactile sensors also drift significantly with temperature. See this paper for more information: *Y.Tenzer, L. P. Jentoft, R. D. Howe Inexpensive and Easily Customized Tactile Array Sensors using MEMS Barometers Chips, IEEE, 2012.* For lab temperatures close to 25° C this is a roughly linear relationship. However, at warmer temperatures it becomes non-linear, and can even saturate. See plots below from some selected fingers:






The data for these graphs were taken over a long timescale (about 2 hours), at constant atmospheric pressure, in a programmable oven. When some of the sensors flat-line around zero, the sensor has saturated and won't return any signal when pressed. Note that the sensor's linear region extends down into colder temperatures until around 10° or 15° C. Also note that the sensors' sensitivity decreases with increased temperature. We did not collect this sensitivity vs. temperate data, nor attempt to compensate for it. Simply shown here is the baseline reading when not pressed vs. temperature.

We fit a line to these data under 35.5° C and used this compensation in the Overo code. We don't do anything special to deal with saturation or the non-linear region. This fit was generated from 44 sensors. As you can see in the plots above, the thermal drift varies from sensor-to-sensor. So there is plenty of opportunity to do something more sophisticated, and more accurate on a sensor-by-sensor basis if you choose to do so.

***Accelerometer***: This sensor should not require calibration.

***Motor currents and temperatures***: these sensors should not require calibration.

## Utilities

***calibrate_opcode utility***: This is a low-level utility for sending the calibration opcode to any sensor on any microcontroller. Note that only some sensor and microcontroller pairs are valid. This is typically only used to calibrate the proximal pin joint and finger spread encoders. It is not intended that you will have to use this utility, as these sensors are factory calibrated.

Run this program with no command line options to view usage, and for interactive use. Typical usage is:

```
  calibrate_opcode <sensor_type>
```

or

```
  calibrate_opcode <sensor_type> <destination>
```

where "`sensor_type`" is an integer from 0-16:

```
 [0] All
```

```
 [2] Motor Hall Encoder
 [3] Motor Winding Temp
 [4] Air Temp
 [5] Supply Voltage
 [6] Motor Velocity
 [7] Motor Housing Temp
 [8] Motor Current
 [9] Tactile Array
[10] Finger Rotation
[11] Proximal Joint Angle
[12] Distal Joint Angle
[13] Cable Tension
[14] Dynamic Sensors
[15] Acceleration
```

and "`destination`" is either an integer 0-11 or "`all`", or an abbreviated name:

```
 [0] [p] Palm
 [1] [f1p] Finger 1 Proximal
 [2] [f1d] Finger 1 Distal
 [3] [f2p] Finger 2 Proximal
 [4] [f2d] Finger 2 Distal
 [5] [f3p] Finger 3 (Thumb) Proximal
 [6] [f3d] Finger 3 (Thumb) Distal
 [7] [m1] Motor 1
 [8] [m2] Motor 2
 [9] [m3] Motor 3
[10] [m4] Motor 4
[11] [t]  Tactile
```

Note that these finger and motor numbers are "hardware" numbers. See the Firmware Numbering section for more info.

***handle_controller calibrate routine***: There is a calibration routine in the `handle_controller` (the code that runs on the Overo), that you can trigger by sending an empty message on the `/right_hand/calibrate` topic if using the ROS interface. (Or by setting the "calibrate" Boolean in the `HandleCommand` class to true if using the straight C interface). There is also a command line switch to run this calibration routine when the `handle_controller` starts up. The first part of this calibration routine opens the spread for three seconds until the finger tips collide. This is close enough to zero that the value can be inspected and if it is less than -512, then you know that you need an offset of 1023. The `handle_controller` takes care of this for you; so all further commands and sensor readings are as if it was zeroed properly.

The second part of the calibration routine attempts to zero the motor tendon excursion encoders. It does this by closing the finger slightly, then opening the finger until the proximal pin joint is zero. Then it stops the motor and sets the motor excursion sensor to zero.

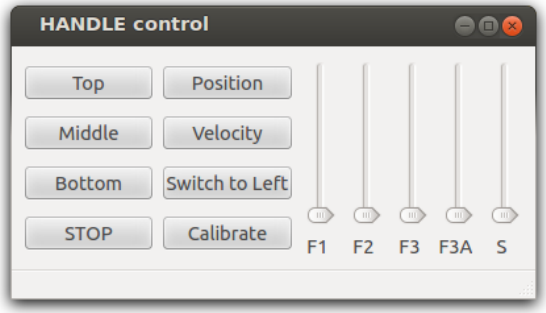
It should be noted that this calibration routine is not exactly bullet proof. In order for it to work the proximal joint and finger spread encoders must first be calibrated and the motor tendon excursions must be close to the proper zero points. It also has the possibility to get stuck in an infinite loop and potentially snap finger tendons. While this routine was intended to be run automatically every time the hand boots up, due to the reasons listed above it is only run upon request and should be used with care.

Also note that this routine is not suitable for getting the fingers unstuck from each other if the hand power cycles during a grasp.

***calibrate_joint_sensors.py***: This Python script walks you through the calibration routine for the distal flexure joints. It calibrates both the flexure sensor as well as the coefficients for using tendon excursion and writes them to configuration files in the `handle_sensors` package named `finger<i>.yaml`. Before running, the calibration jig should be on the palm, the hand should be level and pointing up, and all other sensors should be calibrated.

This node can be started with this command:
`roslaunch handle_launch calibrate_flexure.launch`

## Using the GUI




Use "`rosrun handle_launch gui_control.sh <hand#> [<hand#>]`" to control one or two hands with this Qt GUI.

Toggle the "Position" and "Velocity" buttons to put the hand the mode you want. Then use the sliders on the right to control each motor individually. The "Top", "Middle", and "Bottom" buttons put all the sliders at that respective position. Note that the zero position for position mode is with the slider all the way down, and zero for velocity mode is with the slider in the middle. Entering velocity mode puts the sliders in the middle at zero velocity. Entering position mode puts all the sliders to the bottom for zero position.

The "Calibrate" button sends the calibrate message to the `handle_controller`.

The "Switch to" button toggles the GUI between controlling the right and left hands.

The “STOP” button puts the hand in velocity mode and all sliders in the middle (zero velocity). Note that a few other operations in the GUI will fully stop the hand, such as switching between left/right hands, entering velocity mode, and exiting the GUI.

## Using the “Annan Device”


The “Annan device” is a handy controller box for actuating all the motors on the hand. Four buttons on the left control various control modes.

Position mode: Control the motors in position mode. The zero position is with each slider at the bottom. Typically, you should put the sliders all the way down before entering this mode.

Velocity mode: Control the motors in velocity mode. The zero position is wherever the sliders are when you enter this mode. Typically, you should put the sliders in the middle before entering this mode.

Hand button: when enabled, the device will send commands to `/left_hand/*` topics instead of `/right_hand/*`.

Calibrate button: press this button to start the calibration routine in the `handle_controller`. You should press this button again to un-light the button while the calibration routine is going.

## Testing

Besides the commands listed in the “quick start” section of the document, there are 2 scripts you can use to test functionality of the hand.

The first is a script which sends ROS commands to the handle_controller:
`rosrunhandle_ros testscript.sh`
Note that if the “`annan_control`” node is running, it may conflict with this script.

The second is a utility that can be run by logging into the hand:
`testscript.sh`

## Cautions

- **Reeling in tendons**. These motors are very powerful, and the tendons are very thin. When reeling in tendons use a screwdriver or pen to provide tension on the cable. If your finger gets caught in there it could do damage.
- **Closing fingers too far without object**. We found that closing the fingers too far when there is no object in the palm significantly increases tendon wear.
- **Sustained buckling of fingers**. You should take care not to buckle the distal flexures for long periods of time. They are probably susceptible to creep.
- **Sustained motor stall**. While there are a few thermal protections on the motors, you should probably still take care not to keep them stalled for too long. Sustained stall will increase the motor temperature and limit the amount of torque they can provide until they cool down. The motors should have full power when their winding temperatures are below 90° C.
- **Unwinding tendons too far**. If you unwind the motors too far, they can then reel around the motor spindle backwards. They can also catch on things inside the hand.

## Troubleshooting

**For ARM-S hands: The IP address sticker came off my hand.**

If the sticker comes off, you can determine the address of the hand by looking at the sticker on the Ethernet jack. (You must take off the hand's cover to access this). The sticker will have a number such as "`1263`". The last two digits of this number indicate the last 2 digits of the IP address. So in this case, the IP address will be: "`192.168.40.63`".

**My motors are backwards.**

It is possible the motor tendon became wound around the spindle backwards. Remove the back housing and inspect the tendon.

## Support

For support, please email: `ARMH_Support@irobot.com`

If we cannot walk you through the hand repair, you may be directed to return the hand to:

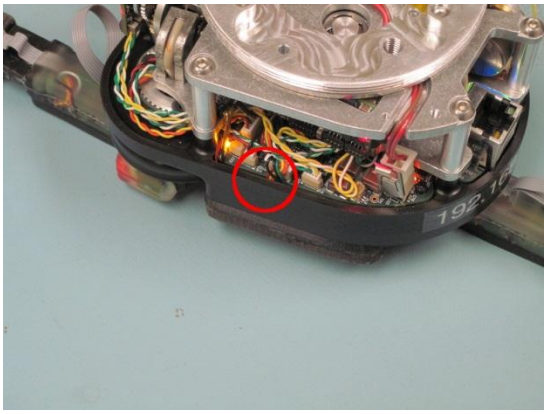
Mark Claffee
iRobot Corporation
8 Crosby Drive, M/S 8-1
Bedford, MA 01730

# Appendix

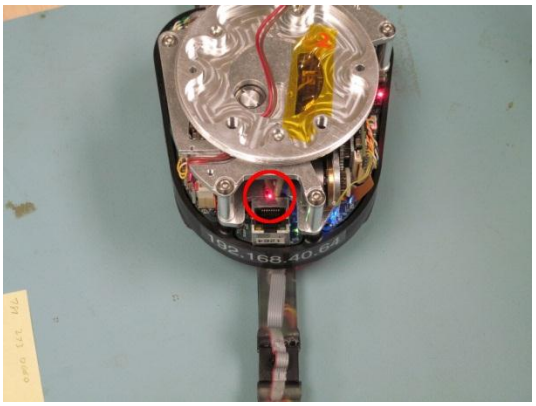
## Diagnosing and Fixing an Unresponsive Part of the Hand

If a part of the hand is not responsive it is possible that a microcontroller is stuck in its bootloader. To verify this, the red LED for that microcontroller will flash indefinitely. The LEDs for the fingers are trivial to observe, and if you remove the hand's back housing, the LEDs on the motor drivers are also easy to observe. The tactile palm, and traffic cop are not as easy.

To see the tactile palm's LED, you need to peek around the traffic cop circuit board. See image. You will see a reflection of a red LED.



For ARM-S hands: To see the LED on the traffic cop, look on the back side of the Ethernet jack between the tendon guides. See image:



On DRC hands, the traffic cop LED is next to the Ethernet jack, closer to the edge of the board.

Once you have identified that you have a microcontroller stuck in bootloader, loginto the hand and follow the standard sequence of commands to run a utility:
`./killhandle.sh` (must return with no warning once)
`./stopall`

Then, run "`iflash.sh`". Then pass in arguments depending on which microcontroller needs to be re-flashed.

It is safe to re-flash a working microcontroller. Depending on how many microcontrollers you are flashing, it can take between 30 seconds and several minutes. Do not power-off the hand during flashing. Sometimes flashing does fail. If so, simply try again. If repeated flashings don't work, contact me for assistance.

Some more info:

- If the traffic cop is stuck in bootloader, then it will appear as though the entire hand is unresponsive.
- The proximal / distal micros are daisy chained, so if the proximal is stuck in bootloader, the distal will not return sensor values either.
- The finger motors are daisy chained as well. So if the first in the chain is in bootloader, the second in the chain will seem unresponsive as well.
- If a finger is unresponsive, but does leave the bootloader after 20 seconds, it is possible that another portion of memory is corrupted. Try to flash it even if it is out of its bootloader. This may or may not work. If not, you can easily put the finger into bootloader by dislocating and reseating it. You will need to act quickly to flash it while it is in its bootloader. To do so, you may need to execute the command to flash that specific microcontroller instead of using the helper scripts described above. Note that if the proximal is fine, but the issue is with distal, you may need to wait until the proximal is out of the bootloader, but before the distal leaves the bootloader. There is a brief 5 second window for this. You should probably ask me for help before attempting this.
- Do not power-off the hand when the bootloaders are flashing. This opens up another possibility for the microcontrollers to be erased.
- It is possible for portions of the eeprom memory to be erased as well. This is can affect the motors by changing PID and thermal constants. To check this, run "`./parametertest2 g a a`" and compare the output with the proper values in the Parameters section.

## Switching Between Smart and Dumb Fingers

After swapping the finger, there are some minor software changes to make:

1. Loginto the hand
2. `makerw`
3. edit the `/etc/init.d/handle_control.sh` file:

To go from dumb to smart fingers:

4. Remove the `--dumb` or `-d` flag.
5. Save and exit the file
6. `makero`
7. power cycle hand
    a. if power cycling the hand is not possible:
    b. `./killhandle.sh`
    c. `./stopall`
    d. Run the last line in `/etc/init.d/handle_control.sh`. Be sure to run it in the background by using an "&".
    e. Logout

To go from smart to dumb fingers:

4. Add the `--dumb` or `-d` flag.
5. Save and exit the file
6. `makero`
7. power cycle hand
    a. if power cycling the hand is not possible:
    b. `./killhandle.sh`
    c. `./stopall`
    d. Run the last line in `/etc/init.d/handle_control.sh`. Be sure to run it in the background by using an "&".
    e. Logout