# Use case: motor control node

This is an example use case for how our comms library might eventually be used on nodes. This example is for the purposes of identifying important functionality and aiding with the underlying structural design (i.e. building a function call diagram) – this means that the functionality is provisional, and the library implementation may still change significantly. Also, the example is chosen to demonstrate a range of functionality, and may not be anything like the actual application built together with the Electronics team later.

The use case is a simplified example of a motor control node for a single DC motor with a relative encoder on an Arduino Mega. The node needs to offer functionality to control motor speed and direction, and read the current motor speed and direction. In addition, there needs to be functionality for monitoring the motor’s power consumption. As other nodes are unlikely to need access to both functionalities, these will be contained on two different services: motor1motion and motor1power.

motor1power will publish a double representing the power consumption. motor1motion will publish a variable motorMotion, which is a struct containing a double for the speed and a boolean for the direction. If a value for the motion is received, it is passed to the setMotorMotion function. In the main loop, the power and speed are read, any messages received are processed (such as requests for data or instructions to set the motor speed and direction), and finally the updated values are published.

Some pseudo-code for the motor control node (keywords in our libraries highlighted in orange):

include Comms, Service

HardwareSerial[] ports = [Serial, Serial1, Serial2]

double motorPower

typedef motion [double speed, boolean direction]

motion motorMotion

Comms nodeComms

Service motionService

Service powerService

void setup() {

PubService<int> s = Anansi.createPubService(37, “bla”);

motionService = new Service(nodeComms, 37, “motor1motion”,

&motorMotion, “setMotorMotion”)

powerService = new Service(nodeComms, 38, “motor1power”, &motorPower)

nodeComms = new Comms()

nodeComms.addService(motionService)

nodeComms.addService(powerService)

nodeComms.initialiseNode(ports) // creates the look-up table

setMotorDirection(true) // e.g. true for forward

setMotorSpeed(0)

}

void loop() {

motorMotion.speed = readMotorSpeed()

motorPower = readMotorPower()

Comms.processMessages() /\* process any messages in the queue

for services on this node \*/

motionService.publish()

powerService.publish()

}

void setMotorMotion(motion m) {

setMotorDirection(m.direction)

setMotorSpeed(m.speed)

}

void setMotorDirection(boolean dir) {

motorMotion.direction = dir

/\* Set direction on motor controller \*/

}

void setMotorSpeed(double speed) {

/\* Set speed on motor controller \*/

}

double readMotorSpeed() {

/\* Return motor speed value from encoder \*/

}

double readMotorPower() {

/\* Return power consumption value from motor controller \*/

}

Other nodes can then call any of the following functions to interact with the motor node:

// global variables:

motion myMotionVar

double myPowerVar

// service function calls:

localService.subscribe(“motor1motion”, &myMotionVar)

localService.request(“motor1power”, &myPowerVar)

LocalService.unsubscribe(“motor1motion”)

localService.setValue(“motor1motion”, myMotionVar)

Explanation of each of the library keywords used above:

* Comms – library for handling resource discovery and communication between services
  + Comms.initialiseNode(ports)

Runs resource discovery on the supplied array of ports, and starts MPQ. After this is called the Comms library has access to a service lookup table, and any messages are automatically passed on or queued in the background using interrupts.

* + Comms.processMessages()

Processes any messages on the queue (i.e. messages for this node). Executes any instructions contained within them, including sending return messages if necessary.

* Service – library class representing a service
  + Service(comms, id, name[, var[, setter]])

Service constructor: creates a new service and adds it to the lookup table. Takes parameters for the comms object that will be used to send messages, id number and name of the service, as well as optional parameters for a pointer to an associated global data variable and the name of the setter function for that variable. (Variable can’t be set remotely if the latter is not supplied.) Services cannot be constructed before node has been initialised.

* + Service.advertise()

Advertise service to neighbouring nodes. This propagates through the network so that all nodes will then know about this service.

* + Service.publish()

Sends a message to each subscriber of this service containing the current variable value.

* + Service.subscribe(name, var)

Subscribe to a service. The first parameter is the name of the service being subscribed to, the second is a pointer to a global variable which will be updated with the subscribed data. The calling service will be added to the subscribers list of the named service.

* + Service.request(name, var)

Request data from a service. Similar to subscribe, but results in the data being sent back exactly once.

* + Service.unsubscribe(name)

Removes this service from the subscribers list of the named service.

* + Service.setValue(name, data)

Set the value associated with the named service to the supplied data. This calls the setter function of the named service with the supplied data as the parameter.