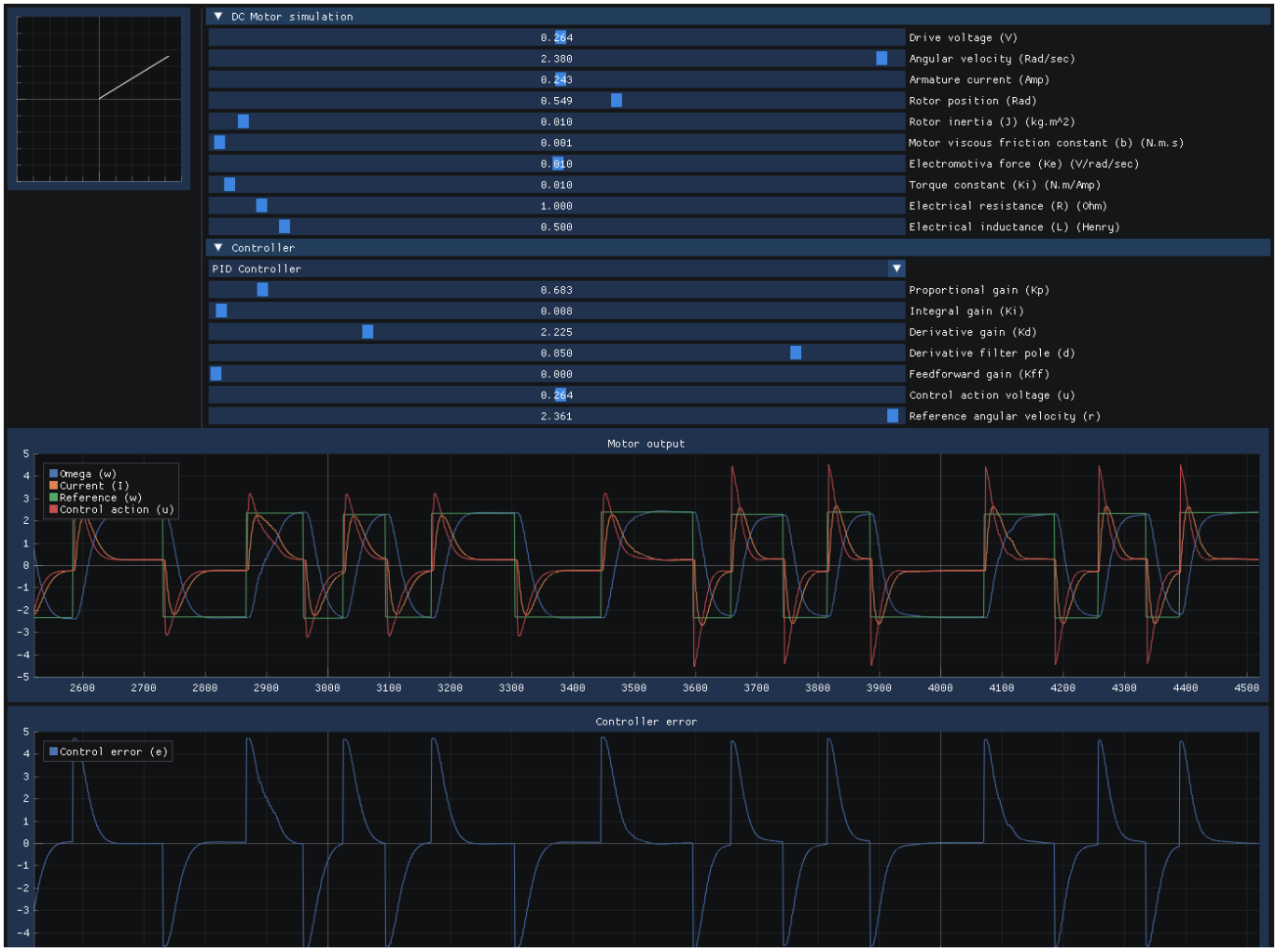
Instrumentation For Embedded Firmware Development

swedishembedded.com



- Embedded Firmware Consulting: get help developing embedded firmware application.
- Firmware Training: get training.
- Swedish Embedded Platform SDK: get the platform SDK.
- Instrumentation For Embedded Firmware Development: repository this documentation describes.

Embedded firmware development differs from traditional software development in one very important way: embedded systems have specialized hardware so majority of the software development for an embedded system has real world hardware effects.

On a general purpose computer, most of these effects are entirely limited to pixels on a screen and so making instrumentation for conventional software development is fairly easy.

For embedded systems, what is needed is a way to model the physical and electrical effects in software as well, so that we can take the physical problem entirely into the software domain where we can easily instrument every aspect of our system.

This is precisely what Swedish Embedded Instruments are for.

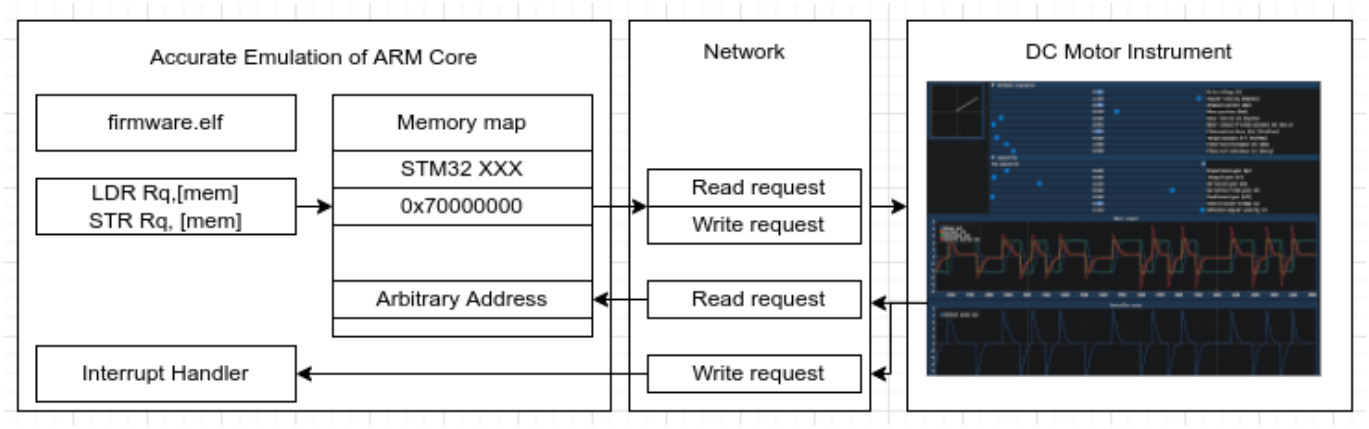
Instrumentation is essential for embedded firmware development. Instrumentation allows you to inspect the internal workings of your firmware in ways that normally would not be possible.

Of course you can not completely replace a physical system and you should never only test in software - but many software bugs can easily be identified using software models which will

save you a lot of time by reducing the number of tests you need to do on a real physical system.

How this instrumentation kit works

Instruments in this kit are all standalone applications which expose a standard socket interface to the emulation software. They are designed to be used with emulation "adapter" such as the one found in Swedish Embedded Platform SDK for renode which simply spawns an instrument as a separate process, instructing it to connect to two network sockets through which all subsequent communication is done.



The emulator adapter taps into the emulation core and catches when firmware wants to read or write memory - which are requests that can be then converted to the instrument link protocol (instrulink).

This is why the instrument protocol closely follows the format of simple load/store word instructions, having fixed packet structure and packet length. Essentially the procol is an implementation of "load/store" assembly instructions designed to operate on bus width of up to 64 bits.

Two sockets

Each instrument accepts three command line arguments:

- **Main socket**: this is a socket number (automatically generated by OS) to which the instrument must connect to receive requests from the emulation master. This socket is used for marster to slave request/response communication.
- **IRQ socket**: this is a separate socket which works exactly the same way as the main socket but instead has requests originated by instrument where emulation instead responds with replies. This socket is used for sending asynchronous messages from the instrument to the emulation (such as interrupt requests).
- **Address**: this is the IP address to which the instrument should connect (where sockets are listening). It is the same for both sockets.

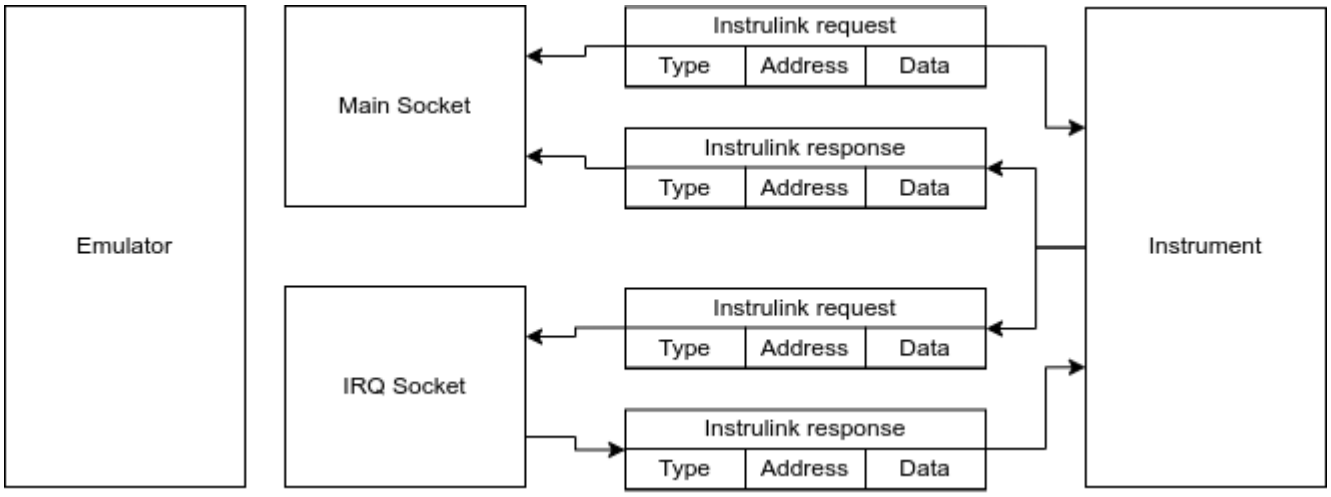
Communication

The communication protocol structure is defined in instrulink_packet.

Each message consists of three fields:

- **Type (uint32_t)**: this is a packet type.
- **Address (uint64_t)**: this is an address that the operation concerns.
- **Value (uint64_t)**: this is the data passed with the operation.

Each packet is exactly the same size - making it easy to implement the protocol without the need for framing since we always assume that packets are simply the same size. Usually the communication is done on the local machine so packets always arrive very reliably (we can make simplifications to the protocol in this way taking into account the assumption that instruments typically are started and stopped many times and while an instrument is connected, there is a high level of reliability in the localhost communication).



Majority of communication usually consists of reads and writes from the firmware side over the main socket. Occasionally the emulator side needs to either read a value or notify the firmware of an interrupt. This is done over the IRQ socket.

How fast is localhost?

Current protocol has been chosen to accomodate for flexibility and a little bit of margin. For example, we could have used 8 bit id. But would it matter?

Let's investigate.

If we want to transfer 10 values 1000 times each second (typical for a control system instrument) this amounts to 20 bytes / per value * 1000 * 10 * 8 = 1.6Mbit/s.

Localhost connections are much faster than this:

```
$ iperf3 -s -p 3000
-----------------------------------------------------------
Server listening on 3000
-----------------------------------------------------------
Accepted connection from 127.0.0.1, port 60954
[  5] local 127.0.0.1 port 3000 connected to 127.0.0.1 port 60956
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  3.29 GBytes  28.2 Gbits/sec
[  5]   1.00-2.00   sec  2.54 GBytes  21.8 Gbits/sec
[  5]   2.00-3.00   sec  3.34 GBytes  28.7 Gbits/sec
[  5]   3.00-4.00   sec  3.27 GBytes  28.1 Gbits/sec
...
$ iperf3 -c localhost -p 3000 -f M
Connecting to host localhost, port 3000
[  5] local 127.0.0.1 port 60956 connected to 127.0.0.1 port 3000
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  3.29 GBytes  3365 MBytes/sec    0   3.18 MBytes
[  5]   1.00-2.00   sec  2.54 GBytes  2597 MBytes/sec    0   3.18 MBytes
[  5]   2.00-3.00   sec  3.34 GBytes  3423 MBytes/sec    0   3.18 MBytes
[  5]   3.00-4.00   sec  3.27 GBytes  3351 MBytes/sec    0   3.18 MBytes
```

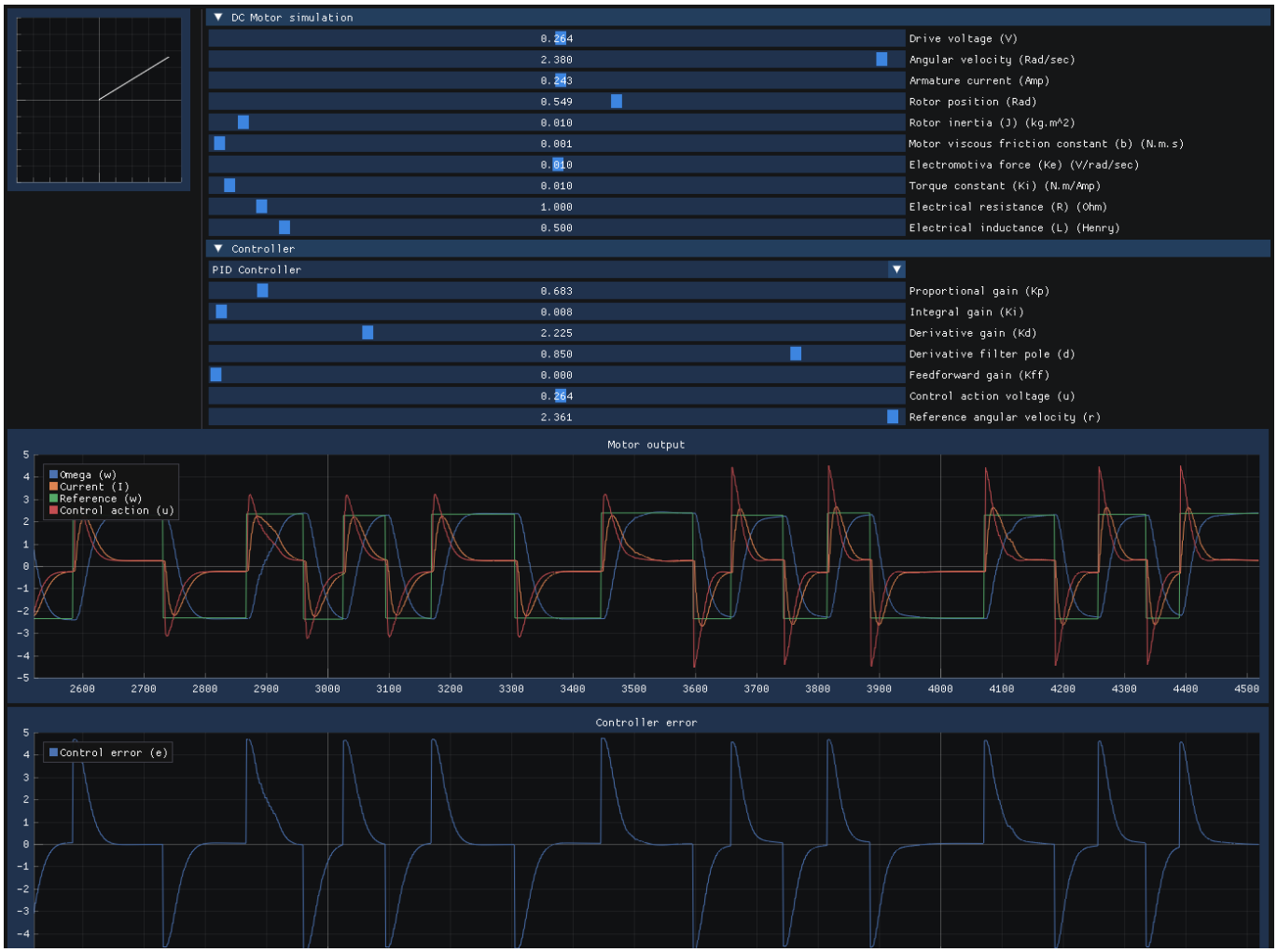
Quite a lot faster. So given that a typical localhost connection is capable of doing 28Gbit/sec on a laptop, it's absolutely fine to have 32 bit header for now. If we want to pass this protocol over UART, we would have an adapter anyway that does uart framing and probably reduces the number of bits transmitted back and forth - but that would be a special use case and it would still fit nicely into the 64 bit bus width to which instruments are connected.

Instruments summary

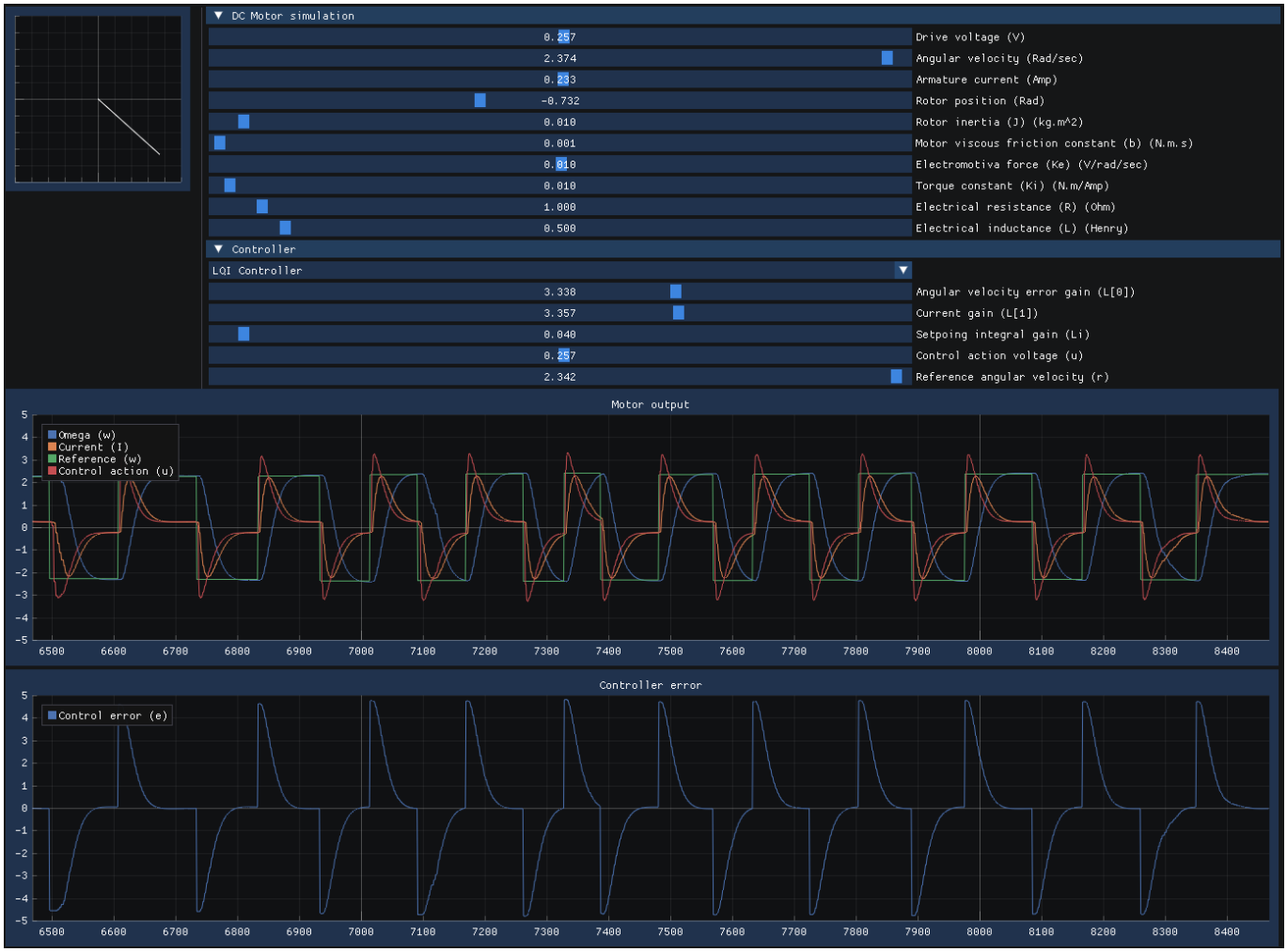
- **DC Motor Instrument**: a mathematically correct DC motor simulation. Useful for creating embedded DC motor controllers and observing realtime motor data (you will need to create an adapter for your UART connection).

DC Motor Instrument

This instrument implements a DC motor simulation and also provides means to input values into the controller (controller typically runs on an embedded system).



Here is the instrument operating in LQI mode (controller runs in firmware with instrument only simulating the motor and providing settings for the controller):



Peripheral Data Structure

Offset	Type	Name	Description
0x00	uint32_t	controller	Instrument controller selection
0x04	float	L[0]	LQI L matrix element
0x08	float	L[0]	LQI L matrix element
0x0c	float	Li	LQI Li integrator gain
0x10	float	Kp	PID controller proportional gain
0x14	float	Ki	PID controller integral gain
0x18	float	Kd	PID controller derivative gain
0x1c	float	d	PID controller derivateive filter discrete pole (0..1)
0x20	float	Kff	Feedforward gain
0x24	float	reference	Controller reference
0x28	float	omega	Measured angular velocity
0x2c	float	control	Controller control action (V)

Offset	Type	Name	Description
0x30	uint32_t	tick	Writing to this register runs the simulation one iteration
0x38	uint32_t	INTF	Interrupt register (currently not used)

These are parameters shared with the controller running on the embedded firmware. If you are using Swedish Embedded Platform SDK then you can use the Renode plugin to map the instrument directly into your microcontroller address space:

.repl file:

```
device: SocketPeripheral @ sysbus <0x70000000, +0x800>
    IRQ -> gpioPortB@8
```

.resc file:

```
device PeripheralPath @/usr/bin/instrument-dcmotor
```

The registers are then available at address 0x70000000 on your microcontroller. You can include the **`instruments/dcmotor.h`** file for easy access to these values from within your simulated firmware.

In such a case, accessing the instrument registers is as simple as defining a pointer and then using it directly:

```
#include <instruments/dcmotor.h>

static volatile struct dcmotor_instrument *vdev =   ((volatile struct
dcmotor_instrument *)0x70000000);
float y[YDIM] = { -(vdev->reference - vdev->omega) };
```

You must use volatile with all instrument data structures because they are modified outside of the knowledge of your application (volatile ensures that memory location is never cached!)

Parameters explanation

Table 1. Motor simulation parameters

Parameter	Description
Drive voltage (V)	This is the drive voltage applied to the motor
Angular velocity (Rad/sec)	Measured angular velocity of the rotor
Armature current (Amp)	Current through the motor coils
Rotor position (Rad)	Position of the rotor
Rotor inertia (J) (kg.m^2)	Inertia of the rotor (heavier rotor is slower to respond)
Motor viscous friction constant (b) (N.m.s)	Friction that depends on angular velocity (high friction results in high cruising current)

Parameter	Description
Electromotiva force (Ke) (V/rad/sec)	Also sometimes expressed as RPM/V
Torque constant (Ki) (N.m/Amp)	This is same value as Ke
Electrical resistance ® (Ohm)	Resistance across motor terminals
Electrical inductance (L) (Henry)	Inductance of the motor coils

Table 2. PID controller parameters:

Parameter	Description
Proportional gain (Kp)	Gain applied proportionally to the error
Integral gain (Ki)	Gain applied to integrated error
Derivative gain (Kd)	Gain applied to filtered derivative of the error
Derivative filter pole (d)	Derivative filter discrete pole

Table 3. LQR controller parameters

Parameter	Description
Angular velocity error gain (L[0])	Control law with respect to angular velocity
Current gain (L[1])	Control law with respect to winding current
Setpoing integral gain (Li)	Integrator gain for input error signal

Table 4. Common variables

Parameter	Description
Feedforward gain (Kff)	Feedforward gain for user input
Control action voltage (u)	Output of the controller
Reference angular velocity (r)	Angular velocity reference set by user

As you modify these values and have a controller running in your simulated firmware, you can experiment with different settings to achieve best possible control action.

Mathematical Modelling

The motor simulation was designed as follows.

You will need to use symbolic modelling tools available in Swedish Embedded Control Systems

Toolbox for this (default octave tools don't support symbolic expressions that we need here).

Start with a continuous time state-space model:

```
syms s J b K R L z Ts
A = [-b/J   K/J;
    -K/L   -R/L];
B = [0;
    1/L];
C = [1   0];
D = 0;
sys = ss(A, B, C, D);
```

Discretize the model using sample time Ts

```
sys = c2d(A, B, C, D, Ts);
```

Generate C code

```
display(sys)
ccode(sys)
```

The resulting model is a parametrized model of the DC motor:

```
self->A[0 * 2 + 0] =
    J * (L + R * Ts) / (powf(K, 2) * powf(Ts, 2) + (J + Ts * b) * (L + R * Ts));
self->A[0 * 2 + 1] = K * L * Ts / (powf(K, 2) * powf(Ts, 2) + (J + Ts * b) * (L + R *
Ts));
self->A[1 * 2 + 0] = -J * K * Ts / (powf(K, 2) * powf(Ts, 2) + (J + Ts * b) * (L + R
* Ts));
self->A[1 * 2 + 1] =
    L * (J + Ts * b) / (powf(K, 2) * powf(Ts, 2) + (J + Ts * b) * (L + R * Ts));
self->B[0 * 1 + 0] =
    K * powf(Ts, 2) /
    (J * L + J * R * Ts + powf(K, 2) * powf(Ts, 2) + L * Ts * b + R * powf(Ts, 2) *
b);
self->B[1 * 1 + 0] =
    Ts * (J + Ts * b) /
    (J * L + J * R * Ts + powf(K, 2) * powf(Ts, 2) + L * Ts * b + R * powf(Ts, 2) *
b);
self->C[0 * 2 + 0] = 1;
self->C[0 * 2 + 1] = 0;
self->D[0 * 1 + 0] = 0;
```

This model is recalculated at every tick since user is able to change these values.

Controller Design

Controller can be designed using the model above.

This uses octave control systems toolbox (since we are calculating with doubles instead of symbolics so we can use it).

Start with a symbolic model

```
syms s J b K R L z Ts
A = [-b/J   K/J;
    -K/L   -R/L];
B = [0;
    1/L];
C = [1   0];
D = 0;
```

Substitute symbols with doubles

```
A = double(subs(A, [J b K R L], [0.01, 0.001, 0.01, 1, 0.5]))
B = double(subs(B, [J b K R L], [0.01, 0.001, 0.01, 1, 0.5]))
```

Convert into continuous time state space object

```
sys = ss(A, B, C, D);
```

Determine observability and controllability

```
e = eig(A) % all must be negative!
assert(e(1) < 0 && e(2) < 0)
M_o = obsv(sys)
M_c = ctrb(sys)
assert(length(A) - rank(M_o) == 0)
assert(length(A) - rank(M_c) == 0)
```

Convert into a discrete time model

```
sys = c2d(sys, 0.1)
```

Design controller

```
Q = [9 0; 0 1]
R = [1]
sys
[sys_k, K] = kalman(sys, 2.3, 1)
[L, S, P] = lqr(sys, Q, R)
dc_g = dcgain(sys)
```

This gives us the matrices that can be used for running the controller on our microcontroller. The resulting controller can be implemented like this:

```
float u[RDIM] = { 0 };
float r[RDIM] = { 0 };
float y[YDIM] = { -(vdev->reference - vdev->omega) };

if (vdev->controller == CONTROLLER_PID) {
    pid_set_gains(&pid, vdev->pid.Kp, vdev->pid.Ki, vdev->pid.Kd, vdev->pid.d);
    u[0] = -(vdev->Kff * y[0] + pid_step(&pid, y[0]));
} else if(vdev->controller == CONTROLLER_LQI) {
    float L[RDIM * ADIM] = { vdev->lqi.L[0], vdev->lqi.L[1] };
    float Li[RDIM] = { vdev->lqi.Li };

    float qi = 0.1;
```

```
    //Control LQI
    lqi(y, u, qi, r, L, Li, x, xi, ADIM, YDIM, RDIM, 0);
    u[0] -= (vdev->Kff * y[0]);
}

// send control input to the plant
vdev->control = u[0];

// Execute state estimator regardless of controller
kalman(A, B, C, K, u, x, y, ADIM, YDIM, RDIM);

vdev->tick = 1;
```

This example is from Swedish Embedded Platform SDK sample (samples/lib/control/dcmotor).

Data Structure Reference

dcmotor_instrument

```
#include <instruments/dcmotor.h>

struct dcmotor_instrument
```

This defines our virtual device

Public Variables	**controller** Instrument controller selection **L** LQI controller L gain **Li** LQI controller integrator gain **lqi** LQI regulator parameters **Kp** Proportional gain **Ki** Integral gain **Kd** Derivative gain **d** Derivative filter pole (0..1) **pid** PID controller parameters **Kff** Feedforward gain **reference** Reference angular velocity **omega** Measured angular velocity **control** Control signal (voltage) **tick** Execute a tick **INTF** Interrupt flag register

Members

```
uint32_t controller
```

Instrument controller selection

`float L`

LQI controller L gain

`float Li`

LQI controller integrator gain

`struct dcmotor_instrument::@0 lqi`

LQI regulator parameters

`float Kp`

Proportional gain

`float Ki`

Integral gain

`float Kd`

Derivative gain

`float d`

Derivative filter pole (0..1)

`struct dcmotor_instrument::@1 pid`

PID controller parameters

`float Kff`

Feedforward gain

`float reference`

Reference angular velocity

`float omega`

Measured angular velocity

`float control`

Control signal (voltage)

`uint32_t tick`

Execute a tick

```
uint32_t INTF
```

Interrupt flag register

$$\hat{x} = Ax + Bu$$

Data Structures

instrulink_packet

```
#include <instruments/protocol/protocol.h>

struct instrulink_packet
```

Instrulink packet

Public Variables	**type** Packet type (instrulink_message_type) **addr** Address for current operation (if applicable) **value** Value for current operation (if applicable)

Members

```
uint32_t type
```

Packet type (instrulink_message_type)

```
uint64_t addr
```

Address for current operation (if applicable)

```
uint64_t value
```

Value for current operation (if applicable)

---

instrulink_message_type

```
#include <instruments/protocol/protocol.h>

enum instrulink_message_type
```

Instrulink packet type codes

MSG_TYPE_INVALID = 0	Invalid message
MSG_TYPE_TICK_CLOCK = 1	Standard "tick" message
MSG_TYPE_WRITE = 2	Write value to address
MSG_TYPE_READ = 3	Read value from address
MSG_TYPE_RESET = 4	Reset the device

MSG_TYPE_IRQ = 5	Interrupt request
MSG_TYPE_ERROR = 6	Error response
MSG_TYPE_OK = 7	Success response
MSG_TYPE_DISCONNECT = 8	Disconnect from socket
MSG_TYPE_HANDSHAKE = 9	Handshake (first message sent by master)