LAB 4
ADVANCED xPC WITH THE PC/104 STACK

Objectives	To gain familiarity with using xPC Target and Matlab/Simulink for real-time control with your PC/104 stack.
Preparation	Read Lab 4 and the webpages on xPC and the inputs and outputs on the breakout board.
Tools	Prototyping breadboard, PC/104 stack as the target real-time computer, and host PC running Matlab with Simulink, Real-Time Workshop, and xPC Target.

1 Getting Started

Begin charging your battery for use later in the lab. Start up Matlab and get your PC/104 stack ready to use. Summarizing:

- Plug in power to your stack, connect it by ethernet to your router, and connect the VGA cable to a monitor. Turn on your stack.
- Run the xPC Target Explorer, `xpcexplr`, in Matlab and make sure your Target PC IP address matches that on your compact flash disk. Other settings should probably be correct; if you have problems later, go back to the webpage and follow the instructions completely.
- Run `simulink` and create a new model. Under Simulation/Configuration Parameters, set the Solver Type to "Fixed-step," the Solver to "discrete (no continuous states)," and the Fixed-step size to "dt". In your Matlab window, set "**dt=0.001**" for 1 ms sample times. (You are free to experiment with this later; what is the shortest time step for which your model will have enough time to execute within the sample period?) In Data Import/Export, you will probably want to limit the data sent back to the last 10000 time steps or so. Under Real-Time Workshop, set the System target file to be xpctarget.tlc.
- Within the xPC Target Explorer, when you have TargetPC1 selected, you can use Target/Ping Target to "ping" your target PC. It will report whether you have successfully established a connection.

In this lab you will explore some more features of Simulink in creating the control program for your PC/104 target. There is a **huge** variety of features we will not explore in lab, however; expect to do a lot of learning on your own, using the help features.

2 PI Velocity Control of a Motor

Important note: the instructions below give only the broad strokes. You must fill in the details. You won't be able to follow the instructions without thinking and succed!

We will do PI velocity control of a motor using encoder feedback. (Typically derivative feedback is not used in velocity control of a motor.) Create a linear current amplifier using an LM348, a TIP31, and a TIP32. Connect one of your analog outputs to the + input of the op amp in your amplifier, and connect the output (transistor emitters) of the amplifier to the - input of the op amp as well as one of the motor terminals. Connect the other motor terminal to ground. Connect the encoder of the motor to one of your encoder inputs.

Now create a Simulink model where one block of the model is a Subsystem called "Motor." (Note: all sample times in all of your blocks should be "dt" or "-1," meaning that the sample time is "inherited.") This block takes "u" as input and gives "angle" as output. Inside that Subsystem, drag in a Sensoray526 encoder input and a Sensoray526 analog output block. In this lab we will measure the encoder angle in degrees, so insert a block after the encoder that converts encoder counts to degrees. Make sure your encoder sample time is dt and the count speed is 4x, and that the sample time of your analog output is dt and the reset vector [1]. Remember that our motor has a 6:1 gearbox and the encoder gives 400 counts per revolution of the motor (100 lines per revolution counted in the 4x mode). How do you convert the encoder counts to the angle of the gearbox output shaft, in degrees? You should have nothing else in this Subsystem except (1) the input "u" going to the Sensoray Analog Output, and (2) the Sensoray Encoder Input going to the converter going to the "angle" output of the block.

Save your model in the folder MATLAB71/work/TeamXY/Lab4, where XY is your team number.

Now, back in your main block, use a Signal Generator to provide your reference velocity. Make it give a sine wave of frequency 0.5 Hz and amplitude 500 degrees/sec. But we will actually want to follow a velocity that goes from 200 deg/sec to 1200 deg/sec. We can do this by adding a Constant (also in the Source blocks) of 700 to the output of the Signal Generator.

Create your "PI Controller" Subsystem block, with input "velerror" and output "u." Inside the Subsystem make "Kp" and "Ki" gain blocks which you will use to tune your controller. You can initialize Kp to 0.1 and Ki to zero. Use the Discrete-Time Integrator for your Integral control. After adding the P and I control contributions, remember to run the signal through a Saturation block to limit the output "u" to between -10 and 10 (volts).

You will also need a block to take the output angle "angle" from the motor and turn it into a velocity. You do this by taking the time derivative of the angle signal. The object that performs this action is sometimes called a "velocity filter." Create a Subsystem called "Velocity Filter" that takes "angle" as input and provides "velocity" as output. Use a "Discrete Derivative" block to actually calculate the velocity.

Now use a Sum block to subtract the actual velocity from the reference velocity to create the "velerror" signal that goes into your PI controller. Connect the output "u" of your PI controller to the input "u" of your Motor block. Complete any other connections needed to make your feedback control system. Finally, complete your model by using an xPC scope to plot the reference signal and the actual velocity (Mux'ed into one signal line), and also send these signals to an Out block, so you can plot them later in Matlab. You might need to adjust the parameters of the scope so the plot on your target monitor is useful.

Now tune your Kp and Ki gains until you get "good" performance. **Plot** about 2 seconds of data using Matlab (remember to use the tg.OutputLog and tg.TimeLog), **print** it out for your group's report, and write on the plot the Kp and Ki gains you used. Make sure the max velocity on your plot is no more than 1500 deg/sec.

How does your actual motor velocity look in your plot? Perhaps a bit noisy? You probably noticed that the measured velocity only takes a few different values. From zooming in on your plot, **measure** the difference between the discrete velocity levels. **Justify** this difference mathematically using the number of degrees per encoder count and the sample time dt, and the fact that the Discrete Derivative block just subtacts the angle from the previous time step from the current angle and divides by the time step to calculate a time derivative. Write your measurement and your justification (including a plain English explanation) on the back side of your plot. Explain why a higher resolution encoder would alleviate the problem.

3 Better Velocity Filtering

Our velocity measurement is not that great! We can do better. Go into your Velocity Filter block and get rid of the Discrete Derivative block. Drag into the Velocity Filter block an Integer Delay (in the Discrete category). Set the Number of delays to be "numdelays" and in your Matlab workspace, set "**numdelays = 5**." The Integer Delay block sends out its input signal numdelays time steps later.

Now drag in an Embedded MATLAB Function from the User-Defined Functions category. When you open it up, you will see a Matlab function ready for you to edit.

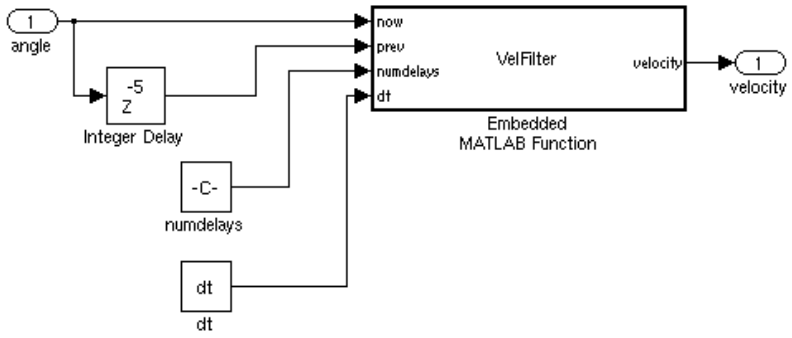
Embedded Matlab functions allow you to code directly in Matlab. This is a very powerful and flexible capability. Some of the capabilities of Matlab are not available in Embedded Matlab; see Matlab documentation for more details on some limitations of Matlab. Our example below will not use much of the flexibility Embedded Matlab offers; you could easily perform the same function in standard Simulink blocks. It's just an example to encourage you to explore further.

Make your Embedded Matlab function look something like this:

```
function velocity = VelFilter(now,prev,numdelays,dt)

velocity = (now-prev)/(numdelays*dt);
```

When you go back to your Velocity Filter block, you should see that it has four inputs and one output. The inputs are now, prev, numdelays, and dt. (Note: you can Mux these inputs together to make things neater. This is useful when you are passing around a lot of information. Don't worry about it for now.) The variable "now" is the current angle, the variable "prev" is the angle "numdelays" time steps ago, and "dt" is the timestep. You must provide "numdelays" and "dt" as inputs to this function; it can't find them from the Matlab workspace. So create two Constant blocks, name them "numdelays" and "dt," give them these values, and send them as inputs to your VelFilter embedded function. When you're done, you should have something that looks like this for your Velocity Filter subsystem:

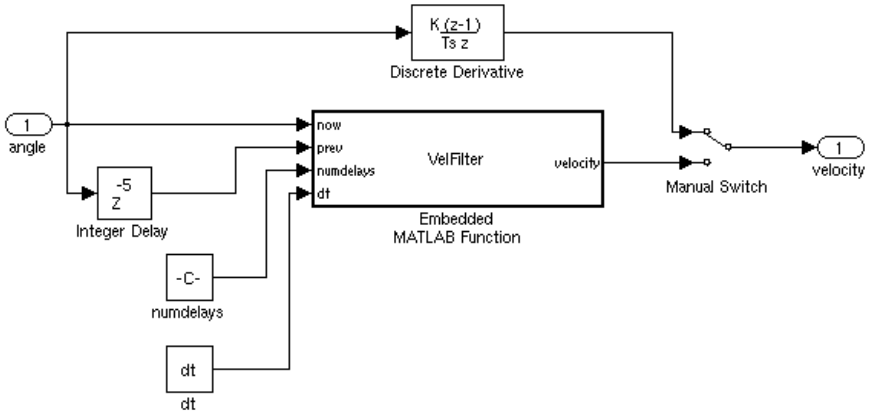


For numdelays = 5, experiment with gains (starting from Kp = 0.1, Ki = 0) until you get good response. The best way to do this is to set your program to run for a long time and to interactively change the gains as you watch the changing response. Once you are happy with the response, as before, print out 2 seconds of plotted data. **Explain** on this plot what your velocity filter does, and why it acts as a low-pass filter in addition to a velocity filter. **Explain** why values of numdelays that are very large are a problem.

This filter is the simplest of digital velocity filters. It would be better if we didn't simply ignore all the data between the current data and the data numdelays timesteps ago. In this course we won't get into digital filter design, but if you're curious, you can look up information on the Matlab commands `butter`, `cheby1`, and `cheby2`, which you can use to create better low-pass filters. These functions return the coefficients you would use in a Discrete Filter block. Take ECE 374 Introduction to Digital Control to learn more about this. A good low-pass filter, plus a Discrete Derivative block, would give you a "good" velocity estimate, given the limited resolution of the encoder.

For those of you who have programmed in C or C++ before, later you can use S-Functions to compile C and C++ files for inclusion in your target program. We will not get into that in this lab. (Remember, there's lots for you to learn on your own!) Another thing to try: if you want to compare the performance of your filter with the simple

Discrete Derivative filter, you can use a Manual Switch to make your Velocity Filter block look like below:



A nice thing about this is that you can double-click on the switch while the target is running and see the different behavior.

4 Understanding Your Motor

Here's something fun to try. Make sure everyone in your group does. Get another motor and spin the shaft by hand. It should spin relatively easily. Now get alligator clips and connect the two motor terminals to each other and try it again. It should be harder to turn now. Why? **Explain**. Remember the motor equations $V = I R + L \, dI/dt + k_e \omega$ and $T = k_t I$. (You can assume $L = 0$ in your explanation; its effect is not too important for what you are feeling.) This behavior explains why a running motor, when disconnected from power, is said to execute a "free running stop," whereas a running motor with its terminals shorted together is said to execute a "fast motor stop."

Now connect the motor's terminals to an oscilloscope and spin the motor's shaft by hand. Observe the back-emf on the oscilloscope.

Finally connect this motor to the motor you are controlling by a flexible coupler. As before, run your controlled motor with a velocity sine wave. Read the voltage at the other motor's terminals on the oscilloscope. **Calculate** the electrical constant k_e of your motor in volts/(rad/sec).

Although our electrical constant k_e and our torque constant k_t have different names, they really express the same underlying constant, in different units. Also, sometimes manufacturers use the term "speed constant," which is the reciprocal of our electrical constant. If we use SI units (amps, seconds, kg, m, volts, etc.), our electrical constant and torque constant have the same numerical value. See this by realizing that the electrical energy you put into your motor is equal to the mechanical energy out plus the resistive heating of the coil, $I V = T \omega + I^2 R$, then divide through by I to get $V = k_t \omega + I R$; here we have k_t where we had k_e before.

So, now that you have your electrical constant and torque constant, you can use it to **draw** the speed-torque curve for this motor. (To do this you also need the resistance across the terminals, so measure that too.) Assume the voltage across the motor is 24V, and indicate the max speed and max torque on your curve. **Fill out** your experimental motor data sheet, including resistance, stall (starting) current, stall torque, max speed (which corresponds to "no load speed"), speed constant (inverse of electrical constant), and torque constant. Verify that your experimentally derived data sheet matches the attached data sheet for your 24V motor.

5 Interaction Between Matlab on the Host and the Real-Time Target

You can send information back and forth between your host and the target while the target is executing. This allows you to, for example, make a Matlab GUI displaying information from the target. This is not real-time, of course; the data is being shipped back and forth by TCP/IP with no guaranteed cycle time. Still, in general it should be pretty fast.

In your Matlab window, type

```
tg.showsignals='on'; tg.showparameters='on'
```

You will see a long list of flags, parameters, and signals. Each signal has a Block Name, and each parameter has a Parameter Name and a Block Name. You can change the Block Names in your Simulink model. Signals travel on the lines interconnecting blocks, and parameters are things you can change without rebuilding your model.

Let's now create a Matlab script (an m-file, see Find/New in the Matlab window) to trigger execution of the target code and to display data in a Matlab window. We will plot the reference velocity and the actual velocity. Write an m-file like the following (the names of your blocks may be different if you gave them better names).

```
% get the signal and parameter ID numbers
ref_vel_ID = tg.getsignalid('Add');
act_vel_ID = tg.getsignalid('Velocity Filter/Manual Switch/SwitchControl');
Kp_ID = tg.getparamid('PI Controller/Kp','Gain');

% remember the original value of Kp
Kp_original = tg.getparam(Kp_ID);

% create the plotting window
ymax = 1500; ymin = 0;
runtime = 10;
figure(1);
clf;
axis([0 runtime ymin ymax]);
hold on;

+tg;                                        % start the model on the target
while(1)
    time = tg.get('ExecTime');              % get the current execution time
    ref_vel = tg.getsignal(ref_vel_ID);     % get reference velocity
```

```
    act_vel = tg.getsignal(act_vel_ID);         % get actual velocity
    plot(time,ref_vel,'o',time,act_vel,'x');    % plot them
    drawnow;                                    % let the figure update
    if (time > runtime/2)
        tg.setparam(Kp_ID,1.0);                 % halfway through, set Kp large
    end
    if (time > runtime)                         % quit after runtime seconds
        break;
    end
end
tg.setparam(Kp_ID,Kp_original);                 % reset the Kp value
-tg;                                            % stop the model
```

Run it by typing the name of the m-file in the Matlab window. Demonstrate to the TA.

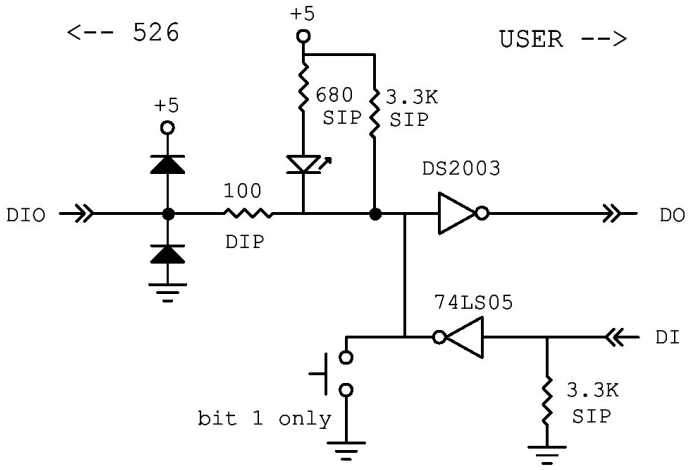
6 Using Digital Inputs and Outputs

In your main Simulink model window, make a block that compares the actual velocity to a constant reference of, say, 400 degrees/sec. If the actual velocity is greater than this constant, the function outputs a 1, otherwise a 0. You might decide to use Embedded Matlab to implement this function. Then send the output to Sensoray Digital Output channel 5. (Make sure its sample time is dt, and choose the reset vector to be [1].)

Now get an LED. The long lead of the LED should have positive voltage with respect to the other lead. Now, remember that your digital output is open-collector, so it can pull the output low, or it can let it float. If you send a 1 to your digital output, it will pull the output low. If you send a 0, it will let it float (i.e., disconnect it). Find a resistor to use with your LED so that no more than 30 mA flows through it, and use the digital output to turn the LED on if the actual velocity is greater than 400 degrees/sec, and turn it off if not. Demonstrate to the TA. Now change the digital output to channels 6, 7, and 8, and verify that all of your digital outputs work. (We won't bother to test channels 1-4.)

The BoB's protection circuit for the DIO's is shown below. The DS2003 and the 74LS05 are both open-collector inverters, capable only of pulling the output low (if the input is logic high) or disconnecting it (if the input is logic low). The figure below shows "pull-up" and "pull-down" resistors that result in the following behavior: digital inputs of +5V, GND, and floating sent to the BoB give you a reading in your Simulink model of 0, 1, 1, respectively, and digital outputs set in Simulink to 0 and 1 give you floating and GND at the BoB, respectively. **Explain** why for each of these five cases. Use this information to **explain** why a digital output pulse train of 0's and 1's commanded to your Sensoray 526 card is not read in as a pulse train by the card if you connect the outputs directly to the inputs.

Note the LED bar on your BoB, shown in the circuit below, can be used to determine the current state of your digital inputs and outputs.

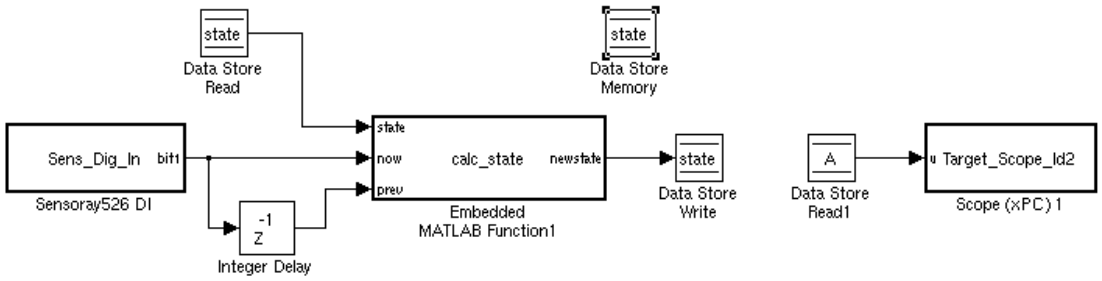


The "BIT 1" button will allow us to test digital input 1, and use it to implement a state machine, a very useful thing indeed. Let's work on the state machine part first. Into your main model window, drag a Data Store Memory block from the Signal Routing Category. Call the data stored there "state" and make the initial value 1. This memory location will store data that we can read and write to. In our case, this data stores the "state" of the system, which could represent the task the system is currently working on. Now drag in a Data Store Read block to read the variable "state" and send it to an xPC Scope in the Numerical mode. Now create an Embedded Matlab function "calc_state" which takes three inputs: state, now, and prev. It could look like this:

```
function newstate = calc_state(state,now,prev)

newstate = state;
if (now > prev)
    newstate = newstate+1;
end
if (newstate>3)
    newstate = 1;
end
```

This just takes the now and prev signal and increments the state when bit 1 goes high, wrapping the state variable around so it only takes the values 1, 2, and 3. The "state" input to this function should be from a Data Store Read block, and the now and prev signals come from bit 1 of the Sensoray Digital Input, where prev is just the value of bit 1 from the previous time step. You should have added something like this to your window:



(Note: the figure above was made with Simulink without xPC, so the Sensoray Digital In and xPC Scope look wrong.)

Now run your program and see that pressing the button changes the state of your state machine.

7 Conclusion

Turn off your PC/104 stack, unplug it, and plug in your battery instead. Turn your stack back on and verify you can run off battery power. One charge of the battery may last you 90 minutes or more.

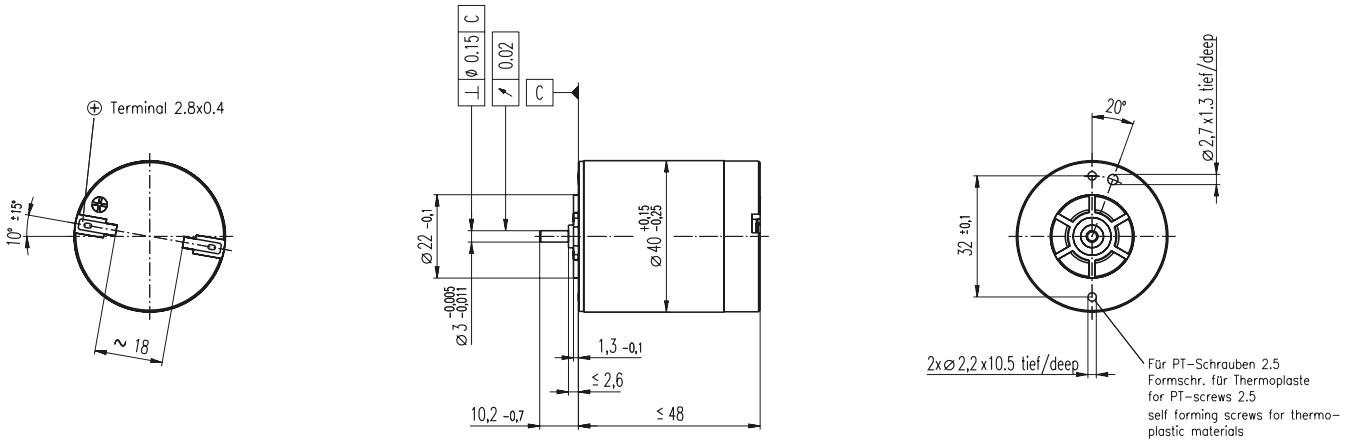
As you can see from all the Simulink blocks you have not yet explored, there is a lot you can do that we haven't covered. Begin exploring! You don't need to be running your PC/104 stack to do so; you can do all your development, and then at the end plug in your inputs and outputs. You can try using multiple time-scales (some operations running at a timestep dt, slower operations running at dt2, etc.) You can consider storing large amounts of data in m-files to be read from the Matlab workspace by Simulink, such as data storing motor or robot motion trajectories. Think about how you would implement logic or search-based programs. This can all be done, even if sometimes it is more awkward than traditional text-based programming.

As a reminder, you have 8 16-bit analog inputs which we have not yet used, but their use is straightforward. You have 4 16-bit analog outputs, 4 incremental encoder counters, and 8 digital inputs and outputs which you can use in blocks of 4: either all 8 inputs, all 8 outputs, or 4 inputs and 4 outputs (channels 1-4 and 5-8).

Summary

- PI velocity control of a motor and filtering
- Experimental characterization of a motor
- Advanced xPC and use of the PC/104 stack I/Os

F 2140 ∅40 mm, Graphite Brushes, 6 Watt, CE approved



M 1:2

Stock program
Standard program
Special program (on request!)

Order Number

2140. ... -22.116-050 (Insert winding number)

Winding number			931	932	933	934	935	936	937	939
Motor Data										
1	Assigned power rating	W	6.0	6.0	6.0	6.0	6.0	6.0	6.0	6.0
2	Nominal voltage	Volt	6.0	9.0	9.0	12.0	15.0	18.0	24.0	36.0
3	No load speed	rpm	3550	4320	3500	3880	3910	3710	3990	4030
4	Stall torque	mNm	26.3	34.4	27.9	31.2	31.6	29.5	31.9	31.1
5	Speed / torque gradient	rpm / mNm	152	136	136	132	130	132	130	134
6	No load current	mA	53	46	34	29	23	18	15	10
7	Starting current	mA	1830	1870	1230	1130	909	669	578	378
8	Terminal resistance	Ohm	3.28	4.81	7.35	10.7	16.5	26.9	41.5	95.2
9	Max. permissible speed	rpm	11000	11000	11000	11000	11000	11000	11000	11000
10	Max. continuous current	mA	839	692	572	476	384	303	244	162
11	Max. continuous torque	mNm	12.1	12.7	13.0	13.2	13.4	13.4	13.5	13.3
12	Max. power output at nominal voltage	mW	2250	3670	2410	3040	3120	2780	3250	3220
13	Max. efficiency	%	62	67	66	68	69	69	70	70
14	Torque constant	mNm / A	14.4	18.4	22.7	27.8	34.8	44.1	55.2	82.3
15	Speed constant	rpm / V	664	519	420	344	275	216	173	116
16	Mechanical time constant	ms	36	33	33	32	31	31	30	30
17	Rotor inertia	gcm²	22.9	23.5	23.2	23.0	22.7	22.1	22.1	21.1
18	Terminal inductance	mH	0.34	0.56	0.85	1.27	1.99	3.21	5.02	11.20
19	Thermal resistance housing-ambient	K / W	10	10	10	10	10	10	10	10
20	Thermal resistance rotor-housing	K / W	8.8	8.8	8.8	8.8	8.8	8.8	8.8	8.8
21	Thermal time constant winding	s	43	44	44	43	43	42	42	40

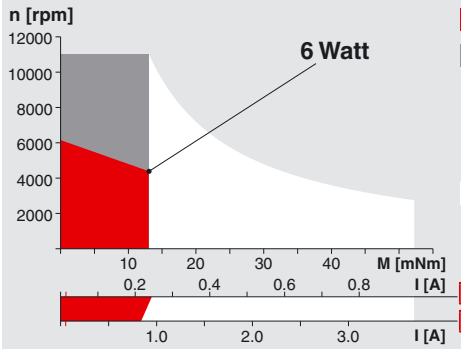
Specifications

- Axial play 0.2 - 0.3 mm
- Max. **sleeve bearing** loads
  axial (dynamic) 0.5 N
  radial (5 mm from flange) 2.5 N
  Force for press fits (static) 50 N
- Max. **ball bearing** loads
  axial (dynamic) 1.5 N
  radial (5 mm from flange) 7.5 N
  Force for press fits (static) 50 N
- Radial play **sleeve bearing** 0.014 mm
- Radial play **ball bearing** 0.025 mm
- Ambient temperature range -20 ... +65°C
- Max. rotor temperature +85°C
- Number of commutator segments 7
- Weight of motor 190 g
- 2 pole permanent magnet
- Values listed in the table are nominal.
  For applicable tolerances see page 43.
  For additional details please use the maxon selection program on the enclosed CD-ROM.

- Options: Our standard program includes ball bearings in place of sleeve bearings.

Operating Range

Comments Details on page 49



Recommended operating range

Continuous operation
In observation of above listed thermal resistances (lines 19 and 20) the maximum permissible rotor temperature will be reached during continuous operation at 25°C ambient.
= Thermal limit.

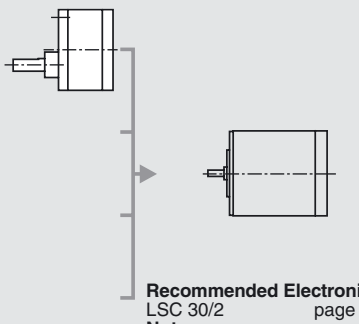
Short term operation
The motor may be briefly overloaded (recurring).

937 Motor with high resistance winding
931 Motor with low resistance winding

maxon Modular System

Overview on page 17 - 21

Spur Gearhead
∅38 mm
0.1 - 0.6 Nm
Details page 223



Recommended Electronics:
LSC 30/2 page 257
Notes **17**