Extracting a more accurate position from a quadrature signal using the SRI

Citation for published version (APA):

Document status and date:
Published: 01/01/2002

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Extracting a more accurate position from a quadrature signal using the SRI

DCT-nr: 2002.40

Internal traineeship
Author: Dennis Bruijnen
Supervisor: dr. ir. M.J.G. v.d. Molengraft

TU Eindhoven, Mechanical Engineering, October 15, 2002
Master Track: Dynamical Systems Design
Section: Control Systems Technology
Abstract

The SRI has been made to get more information out of a quadrature signal produced by an encoder. If you only use the quadrature signal for controlling an actuator the accuracy is limited. It will contain the so-called quantization noise. (It looks a little bit like noise with an maximum amplitude of half an encoder step.) Most controllers require knowledge of the position and the velocity to obtain a good tracking. When using an encoder the problem is to get an accurate velocity signal not to mention an acceleration signal. When differentiating the noisy quadrature signal the noise will be emphasized. When enlarging the gain, the closed-loop system will become faster unstable. Using a low-pass filter to get a better position and speed signal is not always a good solution because of the produced time delay.

Of course you could change the setup like buying another more accurate encoder but if you could use the same setup and upgrade the encoder with some cheap add-on it would be better. The SRI is a prototype of such add-on. It collects the times when a quadrature event takes place. At such event the position is very accurately known namely halfway the encoder step. Every sampletime the SRI will fit a line through the last couple of collected time stamps and extrapolate it over the calculation time of the fit. With this you have an accurate position determination, much better than when using the quadrature signal itself. You can also derive a better velocity signal and even an acceleration signal.

A sample frequency of 100 Hz has already been achieved with a first order polynomial fit through four points with a permitted calculation time of 0.001s. When putting a small fluctuation on a constant input for the motor the position and velocity go nicely through the quadrature signal. A sample frequency of 1000 Hz is easily reachable but that hasn’t been done yet. The only thing to do is calculating faster than one sampletime and that is possible because the algorithm takes about $2 \cdot 10^{-5}s$ of time. And another fact is: the faster you calculate the fit the more accurate will be the prediction of the position because less extrapolation is needed.
Contents

Introduction 3

1 An overview of the experimental setup 4
  1.1 Components of the experimental setup 4
  1.2 Global process 6

2 Reconstructing the real position 7
  2.1 Analysis of the fit-procedure 7
    2.1.1 Quantization 7
    2.1.2 Quadrature event 7
    2.1.3 Parameters of the polynomial fit 9
    2.1.4 Uncertainties 10
  2.2 A straight line 10
  2.3 A sine 14

3 Implementation of the fit algorithm on the DSP 17
  3.1 Global procedure 17
  3.2 Computational errors 18
  3.3 Fit-algorithm 19

4 Experiment 21
  4.1 Determining the encoder-resolution of both encoders 21
  4.2 Simulation mode of the SRI 22
  4.3 SRI experiment using the low-res encoder 24
  4.4 secSine 24

5 Conclusion & Recommendations 26

Bibliography 28

List of symbols 29

List of figures 30

A Testing and debugging the DSP 31

B Instructions for a realtime experiment 32
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C Data sheet of low-res encoder</td>
<td>33</td>
</tr>
<tr>
<td>D m-files used for simulations</td>
<td>34</td>
</tr>
<tr>
<td>E C-code to fit a straight line: main.c</td>
<td>46</td>
</tr>
<tr>
<td>F Application for host: srimain01.c</td>
<td>51</td>
</tr>
</tbody>
</table>
Introduction

Common encoders used for measuring an angular position have a finite resolution so the measured signal is quantized. This leads to a maximum deviation of half an encoder step. The velocity can be obtained by differentiating this position signal. Because of this quantization noise will arise. Often when controlling a system the position and velocity have to be known to obtain a good tracking. So improving the accuracy of these signals can lead to a better tracking.

There are several possibilities to improve this. An encoder with a higher resolution can be used, but this is more expensive or maybe sometimes impossible. Another option is using a low-pass-filter for the position. The position and velocity signals will be much smoother, but there will be phase loss what also can be a problem. The option which will be examined here is the event detection method. When a quadrature event takes place the position is almost exactly known, namely halfway two increments. When collecting the time stamps of such events, a polynomial can be fitted through these points. This fit can be extrapolated to have a very exact prediction of the position after the calculation time for the fit. The calculation will be done by a DSP (Digital Signal Processor).

A start with this topic has been made by H.M.A. van den Akker [1]. This report will continue and elaborate on his work. Another interesting article is [2] from Pierre R. Bélanger. It was published in 1992. The subject has many similarities with this research. He also detects quadrature events and tries to determine the position, velocity and acceleration as accurate as possible using a Kalman filter. The difference with this research is that with the DSP a much higher sample frequency will be possible.

An experimental setup is made to validate the working of the event detection method. An overview of this is presented in Chapter 1. After that the choice what kind of fit to use will be examined in Chapter 2. How this will be implemented at the DSP can be found in Chapter 3. And finally, an experiment to test some parts of the experimental setup will be presented in Chapter 4. At the end there will be made some conclusions. Because the project is much too large for one internal traineeship there will also be made some recommendations which are of use for further research.
Chapter 1

An overview of the experimental setup

1.1 Components of the experimental setup

The experimental setup is a chain of instruments which can be seen in figure 1.1. In figure 1.2 there can be seen some photos of the experimental setup. The mechanical system contains an electrical motor with two encoders attached to it. One with a low and one with a high resolution. The low-res encoder is a HEDS 5500 (see appendix C) and contains hundred increments per revolution. This results in a resolution of 400 quadrature events per revolution. The high-res encoder is 25,909 times more accurate according to the experiments in Chapter 4. The reason of this strange ratio is probably because of a gearbox. This was unknown during the experiments so it isn’t used. The intention was to compare the calculated position with this encoder signal. The electrical motor will be controlled with the TUeDACS/Qad using an amplifier. The crate contains the SRI-piggyback (SRI=Special R Interface), the MPM (=Multi Purpose Memory) and the power supply for these electrical devices. The DSP (=Digital Signal Processor), the A/B-memory and the C/D-memory are mounted on the MPM. The output of the crate, the low-res encoder and the high-res encoder will be connected to a TUeDACS/Qad. And finally this will be connected to a PC.

![Diagram of the experimental setup](image_url)

Figure 1.1: Schematic view of the experimental setup.
CHAPTER 1. AN OVERVIEW OF THE EXPERIMENTAL SETUP

Components of the Experimental Setup

1. electrical motor
2. low-res encoder
3. high-res encoder

4. crate
5. amplifier
6. electrical motor
7. TUeDACs/Quad
8. notebook (host)
9. PCMCIA-adapters for connection to TUeDACs/Quad and the crate
10. PC
11. JTAG-connector

12. MPM/DSP/SRI-combination
13. quadrature signal input
14. JTAG-connection
15. connection to notebook (host)
16. power switch

17. SRI-piggyback
18. MPM with DSP

Figure 1.2: Photos of the experimental setup.
1.2 Global process

The electrical motor drives the encoders. The low resolution encoder gives a quadrature signal to the SRI-piggyback. When the quadrature signal changes the SRI-piggyback generates a 64bit time stamp with a resolution of 100ns. (50ns was planned according to [3], but in the actual experimental setup a 10MHz clock was mounted on the SRI instead of a 20MHz clock. On the other hand in all simulations a time stamp resolution of 50ns has been used. From section 2.2 it is shown that the relative accuracy is reverse proportional to the clock frequency, so if you want to know what the accuracy is of the experimental setup, you have to multiply the relative accuracy in the figures by two.) The 32bit quadrature counter will give the current encoder position. These three 32bit words are called a record. The generated records by the SRI-piggyback will be saved in the A/B-memory of the MPM. Always one memory is collecting records and the other can be read out by the DSP. When a memory switch occurs the collected records during the last sampletime can be used for calculating the polynomial fit. The mechanism to predict the angular position every sampletime is visualized in figure 1.3. Keep in mind that also the angular velocity and, for higher order polynomial fits than one, the angular acceleration can be determined.

Figure 1.3: Overview of the mechanism to calculate the angular position every sampletime

The moment that the C/D-memory is available to be read by the PC is a fixed time after the memory switch of the A/B-memory. At that moment the calculation always has to be ready else there will occur an bus-error. Also the C/D-memory has a switch mechanism just like the A/B-memory. So that memory is available for the DSP to write to and after the memory switch of the C/D-memory that memory is available to be read by the host. This switching of the memory is very useful. There will always be one memory available for the SRI to write to. So no records get lost and the DSP is able to read the latest records.

Finally the high-res encoder is directly coupled with the TUeDACS/Qad. The signal can then be compared with the calculated position by the DSP. This would be a good way to validate the working of the fit algorithm; determining the angular position (and velocity etc...) better using an encoder with a lower resolution.

For a more detailed description about the SRI-piggyback see [3]. For more information about the MPM see [4].
Chapter 2

Reconstructing the real position

2.1 Analysis of the fit-procedure

2.1.1 Quantization

The position will be determined by fitting a line through the last increment changes before a memory switch. There are a lot of ways to do this. The effect of quantization will be investigated next to decide which way is preferred. The quantized discrete signal \( x_q(t) \) can be written as:

\[
x_q(t) = \Delta x \cdot \text{round}\left(\frac{x(t)}{\Delta x}\right), \quad t = k \cdot t_s, \quad k \in \{0, 1, 2, \ldots\}
\]

(2.1)

where the "round" operator rounds to the nearest integer value.

In figure 2.1 an example can be seen of a signal \( x(t) \) (blue line) with the quantized discrete signal \( x_q(t) \) (red dots) and the fitted polynomial \( x_f(t) \) (green line).

The goal is to determine the position at time \( T \) as accurate as possible using data of \( x_q(t) \) before time \( T_m \). \( T_m \) is the time when the memory switch of the A/B-memory occurs. In case of determining the position at time \( T \) using \( x_q(t) \) the maximum error will be \( \Delta x/2 \).

2.1.2 Quadrature event

Another possibility to determine the position is to fit a polynomial through several points before time \( T_m \) which are much more accurate determined. At these points the quantized discrete signal \( x_q(t) \) makes a step. At such step it can with certainty be said that between \( T_i \) and \( T_i - \Delta t \) the real position \( x(t) \) has passed the position \( X_{fi} \). \( X_{fi} \) is the position halfway the increments \( X_i \) and \( X_{i+1} \):

\[
X_{fi} = (X_i - X_{i+1})/2
\]

(2.2)

\( X_{fi} \) and \( T_{fi} \) are the actual points to fit a line through. A visualization of a quadrature event can be seen in figure 2.2. The real position \( x(t) \) can be anywhere at the blue surface. The reason of this uncertainty is a finite resolution of the time stamps. So what time between \( T_i \) and \( T_i - \Delta t \) the real position \( x(t) \) intersects with \( X_{fi} \) is unknown. This will be looked at
as a stochast with an uniform distribution of probability. If one should just take $T_i$ for $T_{fi}$, then the mean value of the deviation for a constant choice of $X_{fi}$ like formula 2.2 won’t be zero because the velocity can vary. This effect will increase for a higher velocity. This can easily be seen in figure 2.2. The slope of the line $x(t)$ will increase with a higher velocity. This results in a broader band (blue surface) so there is more uncertainty about the position. The vertical distance of the blue surface will increase. The middle of it will shift. Only halfway the samples $T_i$ and $T_{i-1}$ the middle of the blue surface doesn’t shift. So this means that the mean value of the stochast is velocity-independent for this time. So this is a good choice for $T_{fi}$:

$$T_{fi} = T_i - \Delta t/2$$  \hspace{1cm} (2.3)

Averagely the fit point $(T_{fi}, X_{fi})$ will be correct. Assuming that the real position $x(t)$ approximates a straight line on a timescale of $\Delta t$ the maximum deviation can be determined with formula 2.4.

$$\max |X_{fi} - x(T_{fi})| = \left| \frac{d}{dt}[x(T_{fi})] \right| \cdot \Delta t/2$$  \hspace{1cm} (2.4)
2.1. ANALYSIS OF THE FIT-PROCEDURE

- \( x(t) \) - surface where \( x(t) \) will be somewhere

- \( t \) - moment of intersection: \( x(t) = X_p \)

- \( \Delta x \), \( V \), \( Y_i \) - fit point (\( T_{fi}, X_{fi} \))

- \( \Delta t \) - actual position \( x(T_{fi}) \) at time \( T_{fi} \)

- \( \Delta t \) - fitpoint (\( T_{fi}, X_{fi} \))

- \( \Delta t \) - encoder signal \( x_q(t) \)

Figure 2.2: The accuracy of the points used for the fit.

The maximum in formula 2.4 will be substituted by \( \frac{1}{r} \Delta x \). \( r \) is the relative accuracy between the fit points and the quantized signal. This results in formula 2.5 which determines the maximum velocity where the relative accuracy \( r \) is guaranteed.

\[
v_{\text{max}} = \frac{1}{r} \frac{\Delta x}{\Delta t}
\]  

For example one wants an accuracy 10 times better than the quantized signal \( x_q(t) \) with a time stamp resolution \( \Delta t \) of 50ns and an encoder resolution \( \Delta x \) of 400 steps/revolution. The maximum angular velocity where the desired accuracy is guaranteed will be: 

\[
v_{\text{max}} = \frac{1}{10^5} \frac{50 \times 10^{-9}}{400} = 3.14 \times 10^4 \text{rad/s}.
\]

The fit points (\( T_{fi}, X_{fi} \)) are now well determined and can be used for the polynomial fit. Equation 2.4 tells exactly how the deviation of one fit point depends on the time stamp resolution and the angular velocity. Keep in mind that this is the deviation only for one fit point and not for the fitted position at time \( T \). The deviation for that position depends on the polynomial order, number of fit points and the extrapolation interval. These parts will be examined next.

2.1.3 Parameters of the polynomial fit

The choice has to be made how many fit points to use and which order polynomial gives the best result. This is not as easy as it seems. More points and a higher order polynomial is more computationally demanding which can cause the prediction of the position at time \( T \) to be worse after extrapolating. Suppose you have \( n \) points to fit a polynomial through. You could use an order of \( n - 1 \) which goes through every fit point. But those points are not completely exact as shown before, so why does the polynomial have to go through these points at all costs? An alternative is to fit a lower order polynomial using the least-square-method. The error of the fit points have an average equal to zero. A lower order polynomial will middle out some of the deviation which results in less deviation at time \( T \) caused by the extrapolation.
To simulate this the models in figures 2.3 and 2.8 are used for respectively a straight line and a sine. \( t_w \) is the time between the last fit point and the memory switch. Because these events are independent of each other you don’t know what time is in between. A duration will be taken that is uniformly distributed from 0 to \( \Delta x/v \) for a straight line and 0 to \( \sqrt{2n\Delta x/a} \) for a sine. The maximum is the time till the next fit point, but it will not be included because a memory switch has occurred just before that fit point. \( t_c \) is the calculation time. During the simulations a value of \( 2 \cdot 10^{-5}s \) has been used. This upper bound estimation is made during debugging the C-code of the fit-algorithm.

Also the kind of input signal \( x(t) \) is of importance. A straight line can be described by a first order polynomial, but a sine or some other curved line not. In sections 2.2 and 2.3 there will be determined what the best choice is in case of the input signal \( x(t) \) being a straight line and a sine respectively.

### 2.1.4 Uncertainties

The polynomial-fit can’t exactly reconstruct the real position at time \( T \). This is the consequence of several uncertainties:

- The accuracy of a time stamp \( T_f \) depends on the time stamp resolution \( \Delta t \). A restriction is that there can be just one step within a time \( \Delta t \). Together with the encoder-resolution \( \Delta x \) results in a maximum velocity \( v_{\text{max}} = \Delta x/\Delta t \). Near this velocity the accuracy of a fit point is about equal to the quantized signal \( x_q(t) \) so the fit will be bad. This becomes visible in the next two sections.

- The calculation of the polynomial-fit takes time and there is a wait time \( t_w \) between the memory switch and the last fit point. The polynomial has to be extrapolated to predict the position at the desired time \( T \). The accuracy of the calculated position at time \( T \) will decrease with increasing prediction horizon.

- The accuracy depends on the choice of the fit strategy. Using a high order fit, which goes through each point, at high speeds is unwise. Figure 2.2 has shown that the accuracy of the fit points decreases at higher speeds. The consequence is that the extrapolation of the fitted line will have a large deviation. after the most recent time stamp \( T_{f1} \) with regard to the real position.

- The steps can be close together but also very far apart what can result in numeric problems during the calculation. The cause of this can be found in Section 3.2. So there has to be looked carefully that the algorithm is accurate enough for every possible signal \( x(t) \) in the required class of signals and for an arbitrary time \( t \). In Chapter 3 this problem will be handled.

### 2.2 A straight line

For all simulations an encoder resolution of 400 steps/revolution, a time stamp resolution of 50ns and a calculation time of \( 2 \cdot 10^{-5}s \) is used. The function for a straight line is:

\[
x(t) = v \cdot t
\]  
\[(2.6)\]
CHAPTER 2. RECONSTRUCTING THE REAL POSITION

2.2. A STRAIGHT LINE

The simulation-model can be seen in figure 2.3. The maximum deviation of \( \frac{1}{2}v\Delta t \) has been arrived from equation 2.4.

![Figure 2.3: Used fit model when \( x(t) \) is a straight line.](image)

The maximum allowed angular velocity \( \Delta x/\Delta t \) in this case is \( 3.14 \cdot 10^5 \text{rad/s} \) which is extremely high. In practice the maximum angular velocity will be much lower so there is no real limitation in this case. In case \( \Delta x \) will be very small (in other experimental setups) this limit could become a serious upper limit. The behavior of this can also be seen in figures 2.4 and 2.12. The largest value on the horizontal axis corresponds to this upper limit.

In figure 2.4 (created with line4.m in appendix D) can be seen the standard deviation of the fit at time \( T \) relative to the standard deviation of the quantization error as a function of the angular velocity and the order of the polynomial. Four points have been used to fit a line. For 50 values of the angular velocity the accuracy has been determined using the average of 10000 drawings. It’s easy to derive the standard deviation of the quantization error when assuming this has a uniform distribution of probability:

\[
SD(x_q) = \sqrt{\frac{1}{\Delta x} \int_{-\frac{1}{2}\Delta x}^{\frac{1}{2}\Delta x} x^2 dx} = \frac{1}{\sqrt{12} \cdot \Delta x}
\]  

(2.7)

In figure 2.5 (created with line1.m in appendix D) the deviation of the angular position \( x_f(T) \) can be seen as a result of the deviation of the time stamps by an angular velocity of 100 rad/s. A first order polynomial is fitted through three points including noise with a uniform probability distribution. The maximum of this disturbance is determined with formula 2.4. The experiment has been carried out 1000000 times so the shape of the distribution of probability becomes clear. It looks like a normal distribution of probability with a standard deviation of about \( 2 \cdot 10^{-6} \text{rad} \). The mean value is equal to zero.

From figure 2.6 (created with line2.m in appendix D) can be made some conclusion about the amount of points and the order of the polynomial (10000 drawings each). It can be concluded that if the real signal \( x(t) \) is a straight line, the best choice of the fit-strategy is a first order polynomial fit. This is actually logic. It doesn’t make sense to fit a straight line with a higher order function. A higher order fit follows the points better, but those points...
probably have deviations. These deviations will be enlarged by the extrapolation. A first order fit will take a sort of average of the deviations of the four points, what reduces the effect of the deviation at the extrapolation. According to figure 2.6 the fit will always get better when using more points, but look out, this is more computational demanding. Using more than four points will not give much of improvement, only more calculation time. So if
$x(t)$ is a straight line the best choice is a first order fit through four points.

![Comparison of possible fit strategies.](image)

Figure 2.6: Comparison of possible fit strategies.

In figure 2.4 can be seen that the lines with different polynomial order lie totally above each other without crossing. So the first order polynomial is the best choice for the whole range of angular velocities.

The accuracy of the fit can also be improved by using a more accurate time stamp resolution. In figure 2.7 (created with line5.m in appendix D) you see that the relative accuracy is reverse proportional to the time stamp resolution, so using a 20MHz time stamp generator instead of 10MHz the fit will be twice as accurate.

![Influence of the time stamp resolution on the relative accuracy.](image)

Figure 2.7: Influence of the time stamp resolution on the relative accuracy.
2.3 A sine

Now it will be examined what the best fit-strategy is for curved lines. The used function is:

\[ x(t) = \sin(2\pi ft) \] (2.8)

The error of the time stamps is calculated using the maximum velocity of the sine. Further the sine is approximated by a polynomial with a constant acceleration. (2nd order polynomial, see figure 2.8) For this acceleration the maximum acceleration of formula 2.8 is taken. This is an extreme case so the following results are only an upper bound. In reality these maxima alternate so the real error will always be smaller. The velocity is equal to \( 2\pi f \) and the acceleration is equal to \( 4\pi^2 f^2 \).

In figure 2.9 (created with sinc1a.m in appendix D) a first order polynomial fitted through three points can be seen. 1000000 drawings have been taken. It is clearly to see that the mean value of the deviation is nonzero. The cause of this is that a straight line just can’t follow the curve when the line is extrapolated towards time \( T \).

Next a second order polynomial is fitted through four points. (See figure 2.10, created with sinc1b.m in appendix D). Also here 1000000 drawings have been taken. This polynomial can follow the curve when it’s extrapolated towards time \( T \) so the mean value is zero.

The above experiments are done for several amounts of points and also for 2nd and 3rd order polynomials. The result can be seen in figure 2.11 (created with sinc2.m in appendix D). For every combination 10000 drawings were taken. The simulation with a 1st order polynomial fit was not carried out, because the fit is very bad. (everywhere the relative accuracy was higher than one, but keep in mind this is only an upper bound. Very low frequencies look locally like a straight line so the fit will not be so bad after all) The 2nd order fit performs better than the 3rd order fit. This is because of the same reason as in the previous section; if you know that the curve looks locally like a 2nd order polynomial why use...
CHAPTER 2. RECONSTRUCTING THE REAL POSITION

2.3. A SINE

Figure 2.9: A 1st order polynomial fit through three points with \( f = 100\,Hz \)

Figure 2.10: A 2nd order polynomial fit through four points with \( f = 100\,Hz \)

Figure 2.11: Comparison of some possible fit strategies.
higher order polynomials? So if the highest frequency is low enough the fit will give a good result.

Finally this experiment can be done for the whole range of frequencies possible (with the angular velocity \( v \) as boundary). The result of this can be seen in figure 2.12. (created with sine4.m in appendix D)

The best choice is a 2nd order polynomial fit. Again using four points is sufficient. More points will not improve the accuracy much. If you want a 10 times better accuracy as the signal \( x_q(t) \) a maximum frequency of 550Hz is allowed in the real position \( x(t) \).

\[
\begin{align*}
1st \text{ order polynomial fit} & \quad \text{2nd order polynomial fit} & \quad \text{3th order polynomial fit}
\end{align*}
\]

\[
\begin{align*}
\text{SD}(\Delta x)(T)/\text{SD}(x(t)) & \quad \text{better} & \quad \text{worse}
\end{align*}
\]

Figure 2.12: Accuracy for the whole range of possible frequencies.

Just like in the previous section the accuracy is reverse proportional to the time stamp resolution. (see figure 2.13, sine5.m in appendix D)

\[
\begin{align*}
\text{SD}(\Delta x)(T)/\text{SD}(x(t)) & \quad \text{Time stamp resolution } \Delta t \quad \text{in s}
\end{align*}
\]

Figure 2.13: Influence of the time stamp resolution on the relative accuracy.
Chapter 3

Implementation of the fit algorithm on the DSP

3.1 Global procedure

In appendix A you see a short briefing to get started with the application to program the DSP. For information about programming in C you can look in [5] or other similar books. In appendix E you can find the C-code of a developed program which fits a 1st order polynomial through four points. The several parts are explained in the enumeration below.

The C-code consist of an initialization part and a running part. Everything what's inside the while-loop belongs to the running part. Everything above belongs to the initialization part. Because the fit-accuracy gets worse caused by the extrapolation the code has to be as fast as possible. So an efficient code is desirable.

- Waiting for the memory switch of the A/B-memory
  When the whole mechanism is activated the status of the DSP will be set to Idle after the initialization. It is then waiting for the memory switch of the A/B-memory. The A/B-memory is collecting time stamps during a time period of one sampletime. The variable InputReady will be set nonzero at the moment of the memory switch.

- Collecting time stamps
  When InputReady is set nonzero there will be evaluated some debug-code. After that the code starts with collecting the time stamps. For 4 points to fit you need 5 time stamps (see formula 2.2). You can’t just subtract $\Delta x/2$ because if the angular velocity is negative an error of $\Delta x$ occurs. $\text{nwords}$ gives the number of words collected by the A/B-memory. It always must be a triple number. If there are 15 words or more (=5 records or more), just 15 words are copied to the R/B-memory (=Record Buffer-memory). This R/B-memory is always available to the DSP and is introduced because at low enough angular velocity there will be collected less than 5 records by the A/B-memory. In such case the records in the R/B-memory will be moved up and new ones will be stored. Each possible case of $\text{nwords}$ is written out in the if-statement. This can also be written much more compact. But the reason of this manner of writing is that a sufficient small for-loop (one operation per loop) can be put in a register of the DSP. This is a built-in for-loop of the DSP-architecture which needs less clock ticks per loop so it speeds up the calculation. Since every gain in computational effort is desirable you can optimize code with smart programming like this. Also pointers are used which are
variables that point to a memory location. This also speeds up the calculation. There is also some code when there happened no quadrature events during the last sampletime. After maxnowords sampletimes the latest record will be copied so if the motor is standing still the fitted position will also become constant.

- **Conditioning time stamps for fitting**
Everything will be calculated with integers. The reason for this will be explained in the next section: Computational errors. The time stamps first have to be translated to fit points. The times will be determined relatively to the 4th record. The positions will be determined relatively to the position of the fifth record and multiplied by two, so calculation with integers can be continued. The first if-statement in the for-loop is an XOR-function. This is needed when the position changes from 0xffffffff to 0x00000000 or backwards. For the time stamps also a construction has been made to overcome this overflow problem. A 32bit integer can only represent a finite number.

- **Calculating the polynomial fit**
Now the actual fit can be calculated. This can also be done with integers only so there are no computational errors at this point. Because all values are conditioned the numbers will not overflow as a result of multiplying and summing up. This part of the code can easily be substituted by a 2nd order polynomial fit or others.

- **Data copy to the C/D-memory**
Now all variables are available to calculate the position at time T. The actual calculation will be done by the host. The variables will only be stored in de C/D-memory. The reason of this is that the host can calculate in 64bit float types and the DSP only with 32bit. Watch out when dividing by an integer. The result of such calculation has to be a double else the number will be rounded to an integer. See [5] for information about implicit and automatic type-conversions.

- **Set the status of the calculation to ready**
Finally two signs are given. One that the A/B-memory has authorization to switch and the other that the output of the C/D-memory is ready.

### 3.2 Computational errors
At first everything was calculated with float-types. But there showed up calculation errors which Matlab didn't produce during the same calculation. Then it came clear that the DSP uses a 32bit float-type which has an accuracy of about 6 digits. (Internally it is represented as 40bits, but it will be saved as 32bits.) That's way too little for the calculation. For example a calculation like $(double) \times \cdot 0xffffffff + y$ was implemented. This gives a totally wrong result because 4294967295 (\(= 0xffffffff\)) just can't be represented accurately enough by a 32bit floating-point type. The solution to this is using either a 64bit or higher floating-point type, and calculating as much as possible in integers and using smart scaling etc. This is realizable by using an available library of an emulation of a 96bit floating-point type. Emulation because the DSP hardware will keep calculating with 32bit words. Unfortunately this will increase the calculation time quite a lot. So what has been done is that the conversion to float (which is unavoidable) has been replaced to the host computer which is accurate enough. This is only
a temporary solution. It’s better to calculate everything at the DSP so the host computer only has to read the fitted position. Additionally it will have more time to do other things like plotting graphs realtime.

The disadvantage of calculating in integers is that there will be a lower bound on the velocity. If the velocity is too low, some used variables in the fit algorithm will overflow. The main cause is squaring of the time (specified with integer-values, resolution 100ns) between the first and the last time stamp. The lower bound for a 1st order polynomial fit algorithm is about 1000 inc/s. (using the low-res encoder this is about 16 rad/s)

When implementing a 2nd order polynomial fit algorithm, there will be variables raised to the power of three, so the lower bound will increase. Therefore, if you want to use this for a system with a non constant velocity and moving both ways you’ll need the above described emulator of a 64bit float-type or higher.

### 3.3 Fit-algorithm

There are several methods to fit a polynomial through a couple of points. We will only look at the case of 4 points because this has shown to be the best choice. A 3rd order fit goes exactly through the four points. In fact it is a set of linear equations that has to be solved. (see formula 3.1)

\[
A = \begin{bmatrix}
T_{f1}^3 & T_{f1}^2 & T_{f1} & 1 \\
T_{f2}^3 & T_{f2}^2 & T_{f2} & 1 \\
T_{f3}^3 & T_{f3}^2 & T_{f3} & 1 \\
T_{f4}^3 & T_{f4}^2 & T_{f4} & 1
\end{bmatrix}
A_p = X_f
\] (3.1)

A way to calculate the coefficients \( p \) is to explicitly convert the matrix \( A \) and multiply it with \( X_f \). A more efficient way to do this is LU-factorization. The algorithm will always converge because \( T_{f1} > T_{f2} > T_{f3} > T_{f4} \geq 0 \). For a second order fit the least-squares-method can be used followed by LU-factorization. (see formula 3.2)

\[
A = \begin{bmatrix}
T_{f1}^2 & T_{f1} & 1 \\
T_{f2}^2 & T_{f2} & 1 \\
T_{f3}^2 & T_{f3} & 1 \\
T_{f4}^2 & T_{f4} & 1
\end{bmatrix}
A^T A_p = A^T X_f
\] (3.2)

The last method can also be used for the first order fit. (see formula 3.3) A more efficient way to calculate this is to analytical write out the calculation. (see formula 3.4)

\[
A = \begin{bmatrix}
T_{f1} & 1 \\
T_{f2} & 1 \\
T_{f3} & 1 \\
T_{f4} & 1
\end{bmatrix}
A^T A_p = A^T X_f
\] (3.3)

\[
\begin{bmatrix}
T_{f1}^2 + T_{f2}^2 + T_{f3}^2 + T_{f4}^2 \\
T_{f1}^3 + T_{f2}^3 + T_{f3}^3 + T_{f4}^3 \\
T_{f1}^4 + T_{f2}^4 + T_{f3}^4 + T_{f4}^4
\end{bmatrix}
\begin{bmatrix}
p_0 \\
p_1 \\
p_2
\end{bmatrix}
= \begin{bmatrix}
T_{f1} X_{f1} + T_{f2} X_{f2} + T_{f3} X_{f3} + T_{f4} X_{f4} \\
X_{f1} + X_{f2} + X_{f3} + X_{f4}
\end{bmatrix}
\]
The first, second and third order polynomial-fit algorithm are tested with Matlab. For the second and third order fit LU-factorization is used and for the first order fit the last method is used. These are chosen because of efficiency. The needed flops are respectively 87, 94 and 36. The amount of flops is a standard for the calculation-time or efficiency. The 1st order fit is about twice as fast as the other two. This is only a quantitative indication because the speed depends on how the compiler translates the C-code to machine-code and what the DSP’s clock-frequency is.

\[
\begin{bmatrix}
    c_1 & c_2 \\
    c_2 & 4
\end{bmatrix}
\begin{bmatrix} p_1 \\ p_2 \end{bmatrix}
= \begin{bmatrix} c_3 \\ c_4 \end{bmatrix}
\]

\[p = \begin{bmatrix}
4c_3 - c_2c_4 \\
4c_1 - c_2^2 \\
c_4 - c_2c_3 \\
4c_1 - c_2^2
\end{bmatrix}\] (3.4)
Chapter 4

Experiment

4.1 Determining the encoder-resolution of both encoders

The experimental setup contains two encoders, one with a low resolution and one with a high resolution. The low-res encoder can be dismantled to count the lines on it. It contains one hundred lines. Because the encoder is a two channels encoder, there will be four quadrature events per line, so this results in a resolution of 400 steps/revolution. (see appendix C for the data sheet of this encoder (HEDS 5500 C)) Also measuring the amount of steps when rotating it 50 revolutions and next dividing by 50 gives an approximation of the resolution. This last experiment has been done with a TUeDACS/Qad connected to a notebook. The result of this is also 400 steps/revolution.

Next the determination of the resolution of the high-res encoder. According to the data sheet this encoder would have 500 lines, but that appeared not to be true. Again with a TUeDACS/Qad the position of both encoders were measured during a constant angular velocity experiment. If the two position are divided the relative accuracy can be obtained. This can be seen in figure 4.1. Strangely the ratio converges to 25,909.

![Figure 4.1: Determining the resolution ratio of the two encoders](image)

Multiplying this number by the number of lines of the low-res encoder gives 2590.9 which is
not an integer. That just can’t be right. A possible cause is that at the start of the experiment the phase difference between the two encoders is unknown. This would result in a maximum deviation of the sum of both encoder resolutions. But if we rotate the motor for a while the ratio-error caused by this will fade away. Also an experiment has been done with dividing the velocities of both encoder signals. The position was first filtered before differentiating it. Then you don’t have any trouble with unknown phase difference. But again the ratio of 25,909 emerges. Rounding this off to 25,91 will become visible during rotating the motor. So what resolution the high-res encoder has is still not exactly known. During the experiments the conclusion was that this encoder is not working properly so it hasn’t been used by further experiments. In a later stage it came clear that the probable cause is a gearbox.

4.2 Simulation mode of the SRI

Now it’s time trying to run the fit-algorithm on the DSP. The SRI is equipped with a simulation-mode. The QSE bit in the SRI Status Word Register (subaddress 0x20) has to be set to enable this mode. (see section 3.1 of [3]) *swmem* sets the time period between two memory switches in units of 100ns instead of 50ns as mentioned before in Chapter 1. (see section 3.3 of [3]) This change of clock frequency will also lead to some other small changes like the following. *scsr* sets the time period at which quadrature events are simulated. The formula is a little bit changed as written in section 3.2 of [3]. The correct formula is:

\[ T_{\text{sim}} = 2^{\text{scsr} - 2}[\mu s] \]  

This can be set with 5 bits in the Simulation Clock Select Register of the SRI (subaddress 0x21), so a minimum of 0.5\( \mu s \) and a maximum of 1073.74\( s \) is possible.

These settings must be set at the host. The main application is *srmain01.c*. (see appendix F)

The results showed in figure 4.2 and figure 4.3 (created with result01.m in appendix D) come from an experiment with the DSP using a first order polynomial fit (see appendix E). The variables of the simulation-mode of the SRI were set as follows: *swmem* = 100000, *scsr* = 12. This corresponds to a sampletime of 0.01s and a constant speed of about 977 inc/s. The timedelay was set to *swmem/10 = 0.001s*. Instructions to carry out this experiment can be found in appendix B. For both figures the blue line is the real (simulated) position \( x(t) \). The red line is the quantized signal \( x_q(t) \) and the red circles are the time stamps.

As can be seen in figure 4.2 the result is good. The black circle is the fitted position. The deviation of the fit is 4.8906 \( \cdot 10^{-5} \) increments. This is about 10000 more accurate as using the encoder signal itself. The deviation during a single experiment is not nicely distributed with an average of zero because of the simulation-mode the time stamps and the memory switches are correlated. This is not the case during an experiment with a real encoder signal. Because the DSP calculates with integers and the time stamps are successive with an exactly equal time period, the only possibility the deviation can come from is the rounding off as a result of the calculation with float-types at the end which is necessary. So using 64bit float-types can maximally give a 10000 times better accuracy.
CHAPTER 4. EXPERIMENT 4.2. SIMULATION MODE OF THE SRI

Figure 4.2: Calculated fit with the DSP using the simulation-mode of the SRI

Figure 4.3: Comparing the angular position of the simulated quadrature events and the fitted angular position.

In figure 4.3 can be seen the difference between the simulated encoder signal and the fitted angular positions. The difference is as expected quantization noise. However there is an offset of 0.1 increment. The cause of this will probably be a time difference between the fitted time \( T \) and the time that the quadrature count register is read out. This results in a constant offset as seen in figure 4.3. Also there are a few peaks which are not predicted. This could indicate a little firmware bug in the SRI.
4.3 SRI experiment using the low-res encoder

As seen in the previous section the algorithm works well at a sample frequency of 100Hz and a time delay of 0.001s (maximum calculation time). The next step is to read out real encoder data. The only thing what has been changed in srmain01.c (appendix F) is not setting the QSE-bit. Moreover, there has to be determined an input for the electrical motor. The choice of input for the motor is: \( u = 0.07 + 0.01\sin(2\pi t) \). Also feedback control would be a possibility, but for this experiment it is not necessary so it hasn’t been used. The response to the input can be seen in figure 4.4. The first two seconds are not included because the electrical motor has to startup. As mentioned earlier the currently implemented algorithm only works at sufficiently high velocities.

![Figure 4.4: Calculate the fit with the DSP using measurements of the low-res encoder](image)

The difference of the fitted angular position and the encoder signal looks partially like quantization noise with an offset. But there is now also deviation as a result of the extrapolation. The input to the motor is a sine which results in a little bit waving angular position. In a result was that a first order polynomial fit isn’t very suitable for non constant velocities.

4.4 secSine

a result was that a first order polynomial fit isn’t very suitable for non constant velocities.

When this wave disturbance is small/slow enough a better position and velocity can be predicted. If you look in figure 4.5 you see that the calculated position signal is much better than the encoder signal itself. Both signals are differentiated with the command `diff` in Matlab. The encoder signal has a limited velocity resolution of 100 rad/s and that’s quite
a lot. Using such signal for feedback control will be no good for the performance. The calculated position signal goes nicely through this quantized angular velocity. Using such signal for feedback control would improve performance. There is still some noise in the signal. This because of the large extrapolation time of 0.001s. According to earlier estimations this could be brought back to $2 \cdot 10^{-5}$s. Also it’s just a 1st order polynomial fit used every sample time. This was only created to fit straight line signals, so when implementing a 2nd order fit algorithm in the DSP it will also improve the quality of the signal.

Figure 4.5: Comparing the two angular position signals after differentiating.
Chapter 5

Conclusion & Recommendations

Conclusion

Fitting a polynomial through some points is not as easy as it looks like. First a fit strategy has to be chosen. From Chapter 2 it can be concluded that when you know that the real angular velocity is constant the best choice is a 1st order polynomial fit through 4 points. If the angular velocity isn’t constant a 2nd order polynomial fit through 4 points is the best choice.

Secondly it has to be calculated at high speed. If you want to use a sample frequency of 1000Hz, you have to calculate 1000 times per second a polynomial fit. This is possible because it has been shown in Chapter 3 that the calculation time of the fit will be in the order of $2 \cdot 10^{-5}$s. As long as this number is sufficiently smaller than the sample time it is possible to use this method. As seen in the last experiment a sample frequency of 100Hz has been reached using a timelag for the calculation time of $1 \cdot 10^{-3}$s. Also the other meaning of the word timelag (caused for example by A/D,D/A-converters) can be eliminated with this method. Just extrapolate a little more.

Finally calculation errors occur as a result of finite accuracy of float-types and finite sizes of integers. The used DSP only calculates with 32bit types, so the maximum accuracy of a float-type is about 6 digits and the maximum size of an integer-type is $2^{32} - 1 = 4,3 \cdot 10^{9}$.

The experiments showed that the algorithm works correctly during a realtime experiment at 100Hz and the calculated angular position is more accurate than the encoder signal itself (see Chapter 4). The calculated angular position couldn’t be compared with the signal of the high-res encoder because this encoder is not working properly. The sample frequency of 100Hz can easily be increased. Theoretically a sample frequency of 1000Hz or even more would be feasible and realistic. To accomplish this some recommendations for further research are done in the next section.

The developed method can also easily be used at many other setups, because it only uses a quadrature signal. Simple, inexpensive encoders can be used to obtain accurate position information. For high-accuracy-demanding setups it can dramatically bring down the costs.
Recommendations for further research

Firstly a 2nd order fit algorithm through 4 points has to be written which can follow curved trajectories. Still a big problem is the restrictions of calculating with 32bit-types. It is strongly recommended to use minimally a 64bit float-type emulator for calculating the fit in the DSP. Else you will not be able to follow low-velocity trajectories. With a non constant velocity such low-velocity events will definitely occur regularly. This algorithm also has to be optimized so the time delay can be decreased.

Secondly another setup would be preferable. The resolution of the high-res encoder couldn't be exactly determined with experiments. The reason is probably a gearbox. Further the low-res encoder is mounted with strong tape on the other end of the electrical motor so you can't see the shaft of the motor rotating. Also the low-res encoder-disc of the low-res encoder is a little bit bend.

Finally all code has to be optimized and the debug-code has to be removed so there will be less time needed for the calculation what means that the time delay can be decreased with a better fit as result.

If the items above are fulfilled the ultimate experiment can be done to validate the important objective; getting a better position determination using the low-res encoder with the SRI than the encoder signal of the high-res encoder. You will have an accurate position determination at a high sample rate with an encoder of low resolution without additional time delay.

Also research about the anti-aliasing behavior of a line fit through several points can be done. The time interval of the fit points determines the frequency content.

Finally research can be done about robustness against encoder failures, like blocked encoder-lines.
Bibliography


List of symbols

\[ a \quad \text{[rad/s}^2\text{]} \quad \text{Angular acceleration} \]
\[ A \quad \text{Matrix used for the fit algorithm} \]
\[ c_i \quad \text{Coefficients used for the 1st order polynomial fit algorithm} \]
\[ \Delta t \quad \text{[s]} \quad \text{Time resolution of the time stamps} \]
\[ \Delta x \quad \text{[rad]} \quad \text{Encoder resolution} \]
\[ f \quad \text{[Hz]} \quad \text{Oscillation frequency of the real position } x(t) \]
\[ k \quad [-] \quad \text{Natural number} \]
\[ n \quad [-] \quad \text{Amount of fit points} \]
\[ p \quad \text{calculated parameters of the polynomial} \]
\[ r \quad [-] \quad \text{The relative accuracy between the time stamps and the quantized signal} \]
\[ t \quad \text{[s]} \quad \text{Time} \]
\[ t_c \quad \text{[s]} \quad \text{Calculation time of the DSP for the fit algorithm} \]
\[ t_w \quad \text{[s]} \quad \text{Wait time between the A/B-memory switch of the and the most recent fit point} \]
\[ T \quad \text{[s]} \quad \text{Time when the calculated angular position is available for the Host PC} \]
\[ T_{fi} \quad \text{[s]} \quad \text{Time of fit point} \]
\[ T_i \quad \text{[s]} \quad \text{Time stamp} \]
\[ T_m \quad \text{[s]} \quad \text{Time when the memory switch of the A/B-memory occurs} \]
\[ t_s \quad \text{[s]} \quad \text{Sampletime of host} \]
\[ T_{sim} \quad \text{[s]} \quad \text{Time period between quadrature events in simulation-mode of the SRI} \]
\[ v \quad \text{[rad/s]} \quad \text{Angular velocity} \]
\[ v_{\text{max}} \quad \text{[rad/s]} \quad \text{Maximum speed where a relative accuracy } r \text{ is guaranteed} \]
\[ x(t) \quad \text{[rad]} \quad \text{Real angular position of the electrical motor} \]
\[ x_f(t) \quad \text{[rad]} \quad \text{Fitted line through fit points} \]
\[ x_q(t) \quad \text{[rad]} \quad \text{Quantized discrete encoder signal} \]
\[ X_f \quad \text{Column with all fit points } X_{fi} \]
\[ X_{fi} \quad \text{[rad]} \quad \text{Position of fit point} \]
\[ X_i \quad \text{[rad]} \quad \text{Quadrature count} \]
List of Figures

1.1 Schematic view of the experimental setup .................................................. 4
1.2 Photos of the experimental setup ................................................................. 5
1.3 Overview of the mechanism to calculate the angular position every sampletime 6

2.1 An example of a fit procedure ................................................................. 8
2.2 The accuracy of the points used for the fit .................................................. 9
2.3 Used fit model when \( x(t) \) is a straight line ........................................... 11
2.4 Accuracy for the whole range of possible angular velocity ....................... 12
2.5 A first order polynomial fit through three points with \( v = 100 \text{ rad/s} \) .... 12
2.6 Comparison of possible fit strategies ....................................................... 13
2.7 Influence of the time stamp resolution on the relative accuracy ............... 13
2.8 Used fit model when \( x(t) \) is a sine ......................................................... 14
2.9 A 1st order polynomial fit through three points with \( f = 100 \text{ Hz} \) .......... 15
2.10 A 2nd order polynomial fit through four points with \( f = 100 \text{ Hz} \) ......... 15
2.11 Comparison of some possible fit strategies .............................................. 15
2.12 Accuracy for the whole range of possible frequencies ............................ 16
2.13 Influence of the time stamp resolution on the relative accuracy ............. 16

4.1 Determining the resolution ratio of the two encoders .............................. 21
4.2 Calculated fit with the DSP using the simulation-mode of the SRI ............. 23
4.3 Comparing the angular position of the simulated quadrature events and the fitted angular position .............................................................. 23
4.4 Calculate the fit with the DSP using measurements of the low-res encoder ... 24
4.5 Comparing the two angular position signals after differentiating ................ 25
Appendix A

Testing and debugging the DSP

If you want to test and debug a made algorithm you have to run through the next steps:

- Make sure the PC is connected to the DSP at the JTAG-connection and the DSP is activated.
- Open de application $CC'\text{C3xC4x}$ used for editing C-code, compiling it and download the assembly-code in the DSP.
- Click on Project/Open and select the desired project-file (projectname.mak).
- Remove init.ge1 in Files/GEL Files (File-tree on the left).
- Use File/Load GEL to load mpm.ge1 and startup.ge1. These files are necessary to allocate the memory at the DSP so it can be used. To activate the memory click on GEL/MPM Target/Initialize.
- The main-code can be found in Files/Project/projectname.mak/source/*.c. Here the main can be found which can be edited.
- To compile the code click on Project/Rebuild all. A .out-file will be created.
- This file can be loaded into the DSP with File/Load program. In Project/Options can be set that after compiling this will be done automatically. So keep that in mind.
- Now you can add break points and profile points with the buttons on the left. For using profile points you first have to click at Profiler/Enable clock and Profiler/View statistics. With profile points you can evaluate calculation time.
- Finally you can press Debug/Run (or F5) to start the DSP and Debug/Halt (or shift-F5) to stop the DSP.
Appendix B

Instructions for a realtime experiment

When debugging of the C-code is finished (described in appendix A) you can take the .out-file which is created after building. The file has to be converted to a .c40-file which can be loaded into the DSP. This can be done by typing the following lines at the command line in a DOS-box:

```
hex30 comload.cmd sri.out -o sri.s
s2c40 sri.s c40
```

These two lines have been put into the file outtoc40.bat for convenience’s sake. Then the file srimain0x.c has to be compiled which includes sri.c40. With make0x.bat this will be automatically done. The file srimain0x.exe will be created if there are now syntax-errors in the code. By running this application the DSP and SRI will be initialized. If the following output is shown the initialization was successful:

```
SRI - starting MPM / SRI demo application
SRI - expecting MPM at TU800AC5 base address 0x800
Entering OpenPb...
pbInitPhyUSBDLLEntries ok.
pbLoadVXD ok.
pbOpen ok.
pbInit ok.
SRI - Interface Identification Register = 0x01f1
  interface identification code : 0x01
  MPM piggyback code : 0xf
  MPM revision code : 0x1
SRI - Control and Status Register = 0x00884600
  memory A present: size = 128 32-bit XWords (1 block(s))
  memory B present: size = 128 32-bit XWords (1 block(s))
  memory C present: size = 128 32-bit XWords (1 block(s))
  memory D present: size = 128 32-bit XWords (1 block(s))
loading sri.c40 ... done (2724 bytes)
bytes=2724
SRI - waiting for DSP heartbeat (HBT)...
SRI - Yo! DSP alive and kicking!
press enter to continue ...
```

srimain01.exe and srimain02.exe both write to a .log-file but it is also possible to write the text displayed on the screen to a file by typing: "srimain0x.exe >zzz.txt". Make sure you have the necessary applications installed such as the compiler salford which is used for compiling srimain0x.c and the conversion application to convert a .out-file to a .c40-file.
Appendix C

Data sheet of low-res encoder

---

### Encoders

#### Optical Encoders

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lin. per revolution</td>
<td>96 to 1024</td>
</tr>
<tr>
<td>Digital output</td>
<td>2 or 3 Channel</td>
</tr>
</tbody>
</table>

See beginning of the Encoder Section for Ordering Information.

**Specifications**

<table>
<thead>
<tr>
<th>Encoder Type</th>
<th>No. of Channels</th>
<th>Linear Per Revolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEDS 5500 K</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>HEDS 5500 C</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>HEDS 5500 D</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>HEDS 5500 E</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>HEDS 5500 F</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>HEDS 5500 G</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>HEDS 5500 H</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>HEDS 5500 A</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>HEDS 5500 B</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>HEDS 5500 J</td>
<td>2</td>
<td>96</td>
</tr>
</tbody>
</table>

- **Encoder**
  - The encoder provides a simple 5 V supply and the two or three channel digital signals are transmitted with a 5 V interface.
  - Optionally, the encoder can be connected to a 5 V interface.
  - The encoder is recommended for long distance operation at low shaft speeds and for elevated radial shaft loads.

### Diode Signals / Hall Diodes - 3 Phase Operation

- **Hall Function**
  - **Connection diagram**
    - Emitter: GND
    - Collector: +5V
    - Base: 1-3

---

**Note:**

- The encoder is designed for industrial and control of both shaft rotation and linear motion as well as for positioning.
- A 5 V power supply is recommended for long distance operation at low shaft speeds and for elevated radial shaft loads.
- The encoder is recommended for long distance operation at low shaft speeds and for elevated radial shaft loads.
Appendix D

m-files used for simulations

1. 1st order polynomial fit through 3 points (line1.m)

```plaintext
dt=0.009;  % [s]         time stamp resolution
inc=400;   % [inc/cycle] encoder resolution
dx=2*pi/inc;  % [rad/ino] encoder resolution
to=-5;     % [s]         calculation time
p=3;       % [-]         number of points
n=1000000; % [-]         order of polynomial
v=100;     % [rad/s]     number of drawings
Xf=zeros(p,1);  % [rad] deviation of the p points to fit
Tf=(0:p-1)*dx/v;  % [s] time vector of points to fit
A=ones(p,p+1);
for k=1:n
    A(:,k)=Tf.^(p+1-k);
end
AT=ones(1,n+1);
X=zeros(1,p);
F=zeros(p,n);
TT=zeros(1,n);

for i=1:n
    Xf=(rand(p,1)-.5)*v*dt;
    T=Tf(p)+rand*dx/v:tc;
    for k=1:o
        AT(k)=T.^(p+1-k);
    end
    X(i)=AT*A\Xf;
    F(:,i)=Xf;
    TT(i)=T;
end

figure(1)
cif
gr=max(max(abs([F; X])));
gr=ceil(gr/dt*v/20).*dt*v/20;
GR=-gr:dt:v/20:gr;
NF=histc(F(3,:),GR);
bar(GR,NF,'histc','r');
hold on
NX=histc(X,GR);
bar(GR,NX,'histc');
set(gca,'fontsize',16)
legend('X-f-.i','x-f
xlabel('deviation [rad]')
ylabel('distribution')
axis([-gr, gr, 0, max(NF)*1.1])
camwa(11)
camdolly(1.6,0,0)
```

34
2. 1-4 order polynomial fit through 2-10 points (line2.m)

dt=50e-9; %[s]  time stamp resolution
inc=400; %[inc/cycle] encoder resolution
dx=2*pi/inc; %[rad/inc] encoder resolution
tc=2e-5; %[s] calculation time
n=10000; %[-] number of drawings
v=100; %[rad/s] velocity
a=10; % maximum amount of points
b=4; % maximum order
XX=zeros(a,b);
X=zeros(1,n);

tf=[0:p-1]'*dx/v;

8=[.]

for k=1:n
    A(:,k)=Tf'*(p+1-k);
end

for i=1:n
    Xf=(rand(p,1)-.5)*v*dt;
    T=Tf(p)+rand*dx/v+tc;
    for k=1:n
        AT(k)=TA(:,k);
    end
    X(i)=AT*(A\Xf);
end
XX(p,:)=std(X);
end

dt=50e-9; %[s]
dt=50e-9; %[s]
dt=50e-9; %[s]
dt=50e-9; %[s]
dt=50e-9; %[s]

figure(1)
cif

XXX=zeros(a,b);
XXX(:,1)=XX(:,1);
XXX(:,2)=[0;0;XX(:,2)-XX(:,1)];
XXX(:,3)=[0;0;0;XX(:,3)-XX(:,2)];
XXX(:,4)=[0;0;0;0;XX(:,4)-XX(:,3)];
bar([1:p]',XXX*sqrt(12)/dx,'stacked')
set(gca,'fontsize',16)
legend('1st order','2nd order','3rd order','4th order')
xlabel('amount of points')
ylabel('SD(x-f(x))/SD(x-f(x))')
axis([1.5 10.5 0 1.1*max(max(XX*sqrt(12)/dx))])
camva(9)
camdolly(.6,0,0)
APPENDIX D. M-FILES USED FOR SIMULATIONS

3. 1st-3rd order polynomial fit through 4 points (line4.m)

dt=50e-9;  %[s]  time stamp resolution
inc=400;  %[inc/cycle]  encoder resolution
dx=2*pi/inc;  %[rad/inc]  encoder resolution
to=2e-3;  %[s]  calculation time
n=10000;  %[-]  number of drawings
X=zeros(1,n);  %[-]  maximum order
b=3;  %[-]  number of speeds
a=50;
x=zeros(a,b);
p=4;
Xf=zeros(p,1);  %[rad]  deviation of the p points to fit
vvv=logspace(1,log10(dx/dt),a);
sc=100000000;

for o=1:b
  for vvv=1:a
    AT=ones(o+1);
    A=ones(p,o+1);
    Tf=sc*(o:p-1)'*dx/v;  %[s]  timevector of points to fit
    for k=1:o
      A(:,k)=Tf.(o+1-k);
    end
    for l=1:n
      Xf=(rand(p,1)-.5)*v*dt*sc;
      T=Tf+p+sc*rand*dx/v+sc*tc;
      for k=1:o
        AT(k)=T.(o+1-k);
      end
      X(l)=AT*(A*Xf)/sc;
    end
    XX(vvv,o)=std(X);
  end
end

figure(1)
clf
loglog(vvv,XX*sqrt([12])/dx)
set(gca,'fontsize',16)
legend('1st order','2nd order','3th order')
xlabel('velocity v [rad/s]')
ylabel('SD(x(t))/SD(x(t))')
grid
axis tight
APPENDIX D. M-FILES USED FOR SIMULATIONS

4. 1st order polynomial fit through 4 points (line5.m)

dt=50e-9;  \quad \% [s] \quad \text{time stamp resolution}
inc=400;  \quad \% [inc/cycle] \quad \text{encoder resolution}
dw=2*pi/inc;  \quad \% [rad/inc] \quad \text{encoder resolution}
tc=2e-5;  \quad \% [s] \quad \text{calculation time}
n=100000;  \quad \% [-] \quad \text{number of drawings}
X=zeros(1,n);  
\% number of speeds
a=50;
b=3;
XX=zeros(a,b);
p=4;
Xf=zeros(p,1);  \quad \% [rad] \quad \text{deviation of the p points to fit}
sc=100000;
o=1;
dtt=linspace(l/1e7,1/5e8,a);
for vv=l:b
  \% \text{v=vv*50;}
  \% \text{for ndt=1:a}
    dt=dtt(ndt);
    AT=ones(1,otl);
    A=ones(p,otl);
    Tf=sc*(0:p-1)'*dx/v;
    for k=1:o
      T(:,:,k)=Tf.(o1-k);
    end
    for i=1:n
      Xf=(rand(p,1)-.5)*v*dt/sc;
      T=Tf(p)+sc*rand*dx/v*sc*tc;
      for k=1:o
        AT(k)=T.(o1-k);
      end
      X(i)=AT*(A\Xf)/sc;
    end
    XX(ndt,vv)=std(X);
  end
end
figure(1)
cif
plot(dtt,XX*sqrt(12)/dx)
set(gca,'fontsize',16)
legend('l/tv/rm = 50 rad/s','l/tv/rm = 100 rad/s','l/tv/rm = 150 rad/s')
xlabel('Time stamp resolution \it{\Delta t}at \it{rm} [s]')
ylabel('SD(l/tv_f(T)\it{rm})/SD(l/tv_q(T)\it{rm})')
grid
axis tight
5. 1st order polynomial fit through 4 points (sine1a.m)

dt=5e-9;  %[s] time stamp resolution
inc=400;  %[inc/cycle] encoder resolution
dx=2*pi/inc;  %[rad/inc] encoder resolution
tc=1e-5;  %[s] calculation time
p=3;  %[-] number of points
n=1000000;  %[-] number of drawings
f=100;
A=1;
v=A*2*pi*f;
Tf=sqrt((0:p-1)'*2*dx/a);  %[s] deviation of the p points to fit
timevector of points to fit
a=A*4*pi^2*f/a^2;
Xf=zeros(p,1);
for k=1:p
    A(:,k)=-Tf*(o+1-k);
end
AT=ones(1,o+1);
x=zeros(1,n);
F=zeros(p,n);
for i=1:n
    Xf=(rand(p,1)-.5)*v*dt+.5*a*Tf^2;
    T=Tf(p)+rand*sqrt(p^2*dx/a)+tc;
    for k=1:o
        AT(k)=T*(o+1-k);
    end
    X(i)=AT*A*Xf-.5*a*T^2;
    F(:,i)=Xf;
end
figure(1)
clf
gr=max(abs(F(1,:)));
GR=gr*gr/40:gr;
NF=histc(F(1,:),GR);
subplot(1,2,1),bar(GR,NF,'histc','y');
set(gca,'fontsize',16)
ylabel('distribution')
xlabel('deviation of points [rad]')
camva(6)

gr=max(max(abs(X)));
k1=min(min(abs(X)));
GR=gr-(gr-k1)/40:gr;
NX=histc(X,GR);
subplot(1,2,2),bar(GR,NX,'histc','y');
set(gca,'fontsize',16)
xlabel('deviation of x_f(T) [rad]')
axis([-1.1*gr,1.1*gr,0,1.1*max(NX)])
camva(6)
6. 2nd order polynomial fit through 4 points (sine1b.m)

dt=50e-9; \quad \% \text{[s]} \quad \text{time stamp resolution}
inc=400; \quad \% \text{[inc/cycle]} \quad \text{encoder resolution}
dx=2*pi/inc; \quad \% \text{[rad/inc]} \quad \text{encoder resolution}
te=2e-5; \quad \% \text{[s]} \quad \text{calculation time}
p=4; \quad \% \{-\} \quad \text{number of points}
o=2; \quad \% \{-\} \quad \text{order of polynomial}
n=1000000; \quad \% \{-\} \quad \text{number of drawings}

f=100;
A=1;
v=A*2*pi*{t}; \quad \% \text{[rad]} \quad \text{deviation of the p points to fit}
Xf=zeros(p,1); \quad \% \text{[rad]} \quad \text{deviation of the p points to fit}
Tf=sqrt([O:p-1]*2*dx/a); \quad \% \text{[s]} \quad \text{timevector of points to fit}
A=ones(p,o+1);
for k=1:o
    A(:,k)=Tf.^(o+1-k);
end
AT=ones(1,o+1);
X=zeros(1,n);
F=zeros(p,n);
for i=l:n
    Xf=(rand(p,1)-.5)*v*dt+.5*a*Tf.*2; %maximum deviation of v*dt/2
    T=Tf(p)+rand*sqrt(p^2*dx/a)+tc;
    for k=1:o
        AT{k}=T.^(o+1-k);
    end

    X{i}=AT*A\Xf-.5*a*T.*2;
    F(:,i)=Xf;
end
figure(1)
cIfl
gr=max(abs(F{1,:}));
GR=gr/40;gr;
NF=histc(F{1,:},GR);
subplot(1,2,1),bar(GR,NF,'histc','y');
set(gca,'fontsize',16)
ylabel('distribution')
xlabel('deviation of points [rad]')
cama(6)

gr=max(max(abs(X{1})));
kl=min(min(abs(X{1}));
GR=gr/(kr-xl)/40;gr;
NX=histc(X,GR);
subplot(1,2,2),bar(GR,NX,'histc','y');
set(gca,'fontsize',16)
xlabel('deviation of X_{i}(T) [rad]')
axis([-1.1*gr,1.1*gr,0,1.1*max(NX)])
cama(6)

39
7. 2nd-3rd order polynomial fit through 4 points (sine2.m)

dt=50e-9;  %[s] time stamp resolution
inc=400;  %[inc/cycle] encoder resolution
dx=2*pi/inc;  %[rad/inc] encoder resolution
to=2e-5;  %[s] calculation time
n=10000;  %[ ] number of drawings
a=10;  % maximum amount of points
b=5;  % maximum order
XX=zeros(a,b);  X=zeros(1,n);  f=100;  A=1;
v=4*pi^2*f;  aa=A^4*pi^2*f^2;
for o=1:b
    for p=2:a
        if p>o
            AT=ones(1,o+1);
            A=ones(p,o+1);
            Tf=sqrt((0:p-1+t)^2*dx/aa);
            Xf=zeros(p,1);
            for k=1:o
                A(:,k)=Tf.^(o+1-k);
            end
            for i=1:n
                Xf=(rand(p,1)-.5)*v*dt+.5*aa*Tf.^2;  %maximum deviation of v*dt/2
                T=Tf(p)+rand*sqrt((p^2*dx/aa)+tc);  % time T
                for k=1:o
                    AT (k)=T.^(o+1-k);
                end
                X(i)=AT(A)*Xf-.5*aa*T^2;
            end
            XX(p,o)=std(X);
        end
    end
end
figure(1)
cif
XXX=zeros(a,b);
XXX(:,3)=XX(3:p,2);  %XXX(:,3)=XX(:,3)-XX(:,3)
%XXX(:,4)=XX(:,4)-XX(:,4)
bar([1:p],XXX[:,2,:]*sqrt(12)/dx,'stacked')
set(gca,'fontsize',16)
legend('2nd order','3rd order')
xlabel('amount of points')
ylabel('SD(\frac{X}{T})/SD_{X_{avg}(T)}')
axis([0.5 10.5 0 1.1*max(max(XX[,2,:]*sqrt(12)/dx))])
camva(9)
camdolly(.6,0,0)
APPENDIX D. M-FILES USED FOR SIMULATIONS

8. 1st-3rd order polynomial fit through 4 points (sine4.m)

dt=50e-9;  %[s] time stamp resolution
inc=400;  %[inc/cycle] encoder resolution
dx=2*pi/inc;  %[rad/inc] encoder resolution
to=0e-5;  %[s] calculation time
n=10000;  %[-] number of drawings
b=3;  %[-] maximum order
a=50;  %[-] number of speeds
AA=1;
fx=dx/dt/AA/2/pi;
X=zeros(1,n);
XX=zeros(a,b);
p=4;
Xf=zeros(p,1);  %[rad] deviation of the p points to fit
vvv=logspace(-1,log10(fmax),a);
sc=100000;
for o=1:b
    for vv=1:a
        f=vvv(vv)/sc;
        v=AA*2*pi*f;
        aa=AA*4*pi*A2*fA2;
        AT=ones(1,o+1);
        A=ones(p,o+1);
        T=[sqrt([0:p-1]'*2*dx/aa)];  %[s] time vector of points to fit
        for k=1:o
            %(...)
        end
        for i=1:n
            Xf=(rand(p,1)-.5)*v*dt*sc+.5*aa*Tf.*2;  %maximum deviation of v*dt/C
            T=T+p*rand*sqrt(p*2*dx/aa)+sc*tc;
            for K=1:o
                %(...)
            end
            X(i)=AT*(A\Xf)-.5*aa*T^2;
        end
        XX(vv,o)=std(X);
    end
end

figure(1)
cf
loglog(vvv,XX(:,1:o)*sqrt(12)/dx)
set(gca, 'fontsize',20)
legend('1st order', '2nd order', '3th order')
xlabel('Frequentie f [Hz]')
ylabel('SD(x_f(T))/SD(x_q(T))')
grid
axis tight
APPENDIX D. M-FILES USED FOR SIMULATIONS

9. 2nd order polynomial fit through 4 points (sine5.m)

dt=50e-9; %[s] time stamp resolution ino=400; %[inc/cycle] encoder resolution dx=2*pi/inc; %[rad/inc] encoder resolution tc=2e-5; %[s] calculation time \n=100000; %[-] number of drawings X=zeros(1,n); b=3; % maximum order a=50; % number of speeds AA=1;

fmax=dx/dt/AA/2/pi;
XX=zeros(a,b);

Xf=zeros(p,1); %[rad] deviation of the p points to fit
sc=100000;
dtt=linspace(1/le7,1/5e8,a);
vvv=(1:3)*50;

for vv=1:b
    for ndt=1:a
        f=vvv(vv)/sc;
        dt=dtt(ndt);
        v=AA*2*pi*f;
        aa=AA*4*pi^2*f^2;
        AT=ones(1,o+1); A=ones(p,o+1);
        Tf=sqrt((0:p-1)^2*dx/aa); %time vector of points to fit
        for k=1:o
            A(:,k)=Tf.^2(0:1-k);
        end
        for i=1:n
            Xf=(rand(p,1)-.5)*v*dt*sc+.5*aa*Tf.^2; %maximum deviation of v*dt/2
            T=Tf(p)+rand*sqrt(p2*dx/aa)+sc*tc; %time T
            for k=1:o
                A(k,:)=-T^2(0:1-k);
            end
            X(i,:)=AT*(A*Xf)-.5*aa*T^2;
        end
        XX(ndt,vv)=std(X);
    end
end

figure(1)
cif
plot(dtt',XX(:,1:b)*sqrt(12)/dx)
set(gca,'fontsize',20);
legend('\"itf\"rm = 50 Hz','\"itf\"rm = 200 Hz','\"itf\"rm = 350 Hz')
xlabel('Time stamp resolution \lt\Deltat\rm [s]')
ylabel('SD(x_f(T))/SD(x_q(T))')
grid
axis tight
10. Fit with DSP using SRI simulation-mode (result01.m)

```matlab
load srimain01.log
T5=srimain01(:,1);  % 5th time stamp low [100us]
T6=srimain01(:,2);  % 6th time stamp low [100us]
T7=srimain01(:,3);  % 7th time stamp low [100us]
T2=srimain01(:,4);  % 2nd time stamp low [100us]
T3=srimain01(:,5);  % 3rd time stamp low [100us]
X1=srimain01(:,6);  % 6th quadrature value [1us]
TT=srimain01(:,7);  % time of ready-sign of DSP [4]
X6=srimain01(:,8);  % 8th position [1ns]
V5=srimain01(:,9);  % edited velocity [m/s]
EH=srimain01(:,10); % high-res encoder value at ready-sign of DSP [inc]
EL=srimain01(:,11); % low-res encoder value at ready-sign of DSP [inc]

FACT = 25.90914775834578; % factor between low-en high-res encoder
N = 16;  % translation of time steps
unit = 0; % 1 = rev/s  0 = inc/s with 100 inc/revolution
if unit==1
    facL=720/400;  % resolution of low-res encoder
    facH=720/400/FACT;
else
    facL=1;
    facH=1/FACT;
end

T1=T1/8; T2=T2/9; T3=T3/8; T4=T4/9; T5=T5/8;
X5=X5+L*facL; X5=X5+L*facL; EH=EH*facH;
ax=15/8; % ramp line [s]
td=st/10; % trim delay [s]
sp=inc/(2*(32)*1e-6); % Velocity
kk=33; % number of round to examine

figure(1)
cf
for k=kk:kk+1
    ns=ceil(T(k)/st); % memory switch line [5]
    subplot(1,2,k+1)
    plot(TS(k) T5(k) T3(k) T2(k) T1(k),X5(k)+facL,'or')
hold on
    plot(Td=ns, EL(k), 'xk')
    plot(Td=ns, X(k), '+k')
    plot(TS(k), X5(k)+L*facL, '+k')
    x=[ ];
    t=[ ];
    br=L; %
    for i=1:1:1
        plot(TS(k)-facL/sp;br X5(k)-facL/sp;
        x=[x t]
        t=[t i]
    end
    t=t+1;
    plot(TS(k-T1/3))
    xlabel('iti.k.

end

figure(2)
cf
for k=kk:kk+1
    subplot(1,1,1)
    plot(TS(k) EL(k), '+k')
    if unit==1
        ylabel('time [ms]')
    else
        ylabel('time [ns]')
    end
    axis tight
    xlabel('time [us]')
    plot(TS(k)(1-3)*facL T5(k)-3*facL, 'r')
    plot(TS(k)(1-3)*facL T5(k)-3*facL, 'r')
end

```
APPENDIX D. M-FILES USED FOR SIMULATIONS

legend('low-res encoder', 'fitted position', 'high-res encoder')

subplot(1,2,1)
plot(tt,XX-EL, 'r')
hold on
plot([tt(1), tt(1+length(tt))], mean(XX-EL)'[1 1], 'b')
xlabel('time [s]')
if unit=='
ylabel('error with low-res encoder [rad]')
else
ylabel('error with low-res encoder [inc]')
end

legend('fitted position minus low-res encoder', 'mean-value of difference')

subplot(1,2,2), plot(tt,XX-ER)
xlabel('time [s]')
if unit=='
ylabel('error with high-res encoder [rad]')
else
ylabel('error with high-res encoder [inc]')
text
11. Fit with DSP using measurements of the low-res encoder (result02.m)

```matlab
load srimain02.log
cc=sine(srimain02,1);
cp=round(cc/5)+1;
T1=srimain02(chrome,1); % time stamp low [100ns]
T2=srimain02(chrome,2); % time stamp low [100ns]
T3=srimain02(chrome,3); % time stamp low [100ns]
T4=srimain02(chrome,4); % time stamp low [100ns]
X1=srimain02(chrome,5); % fitted position [inc]
X2=srimain02(chrome,7); % fitted position [inc]
V=srimain02(chrome,9); % fitted velocity [Inc/s]
El=srimain02(chrome,11); % blue-res encoder value at read-sign of DSP [inc]
FAC = 25.9091477538345780; % factor between low-on high-res encoder
N = 16; % resolution of time stamps
dt0 = 0; % 1 = cm/s
unit = 1; % 1 = inc's with 400 inc/revolution
if unit=1
    facn=4*pi/(400); % resolution of low-res encoder
    fac2=2*pi/(400/FAC); % resolution of high-res encoder
else
    facn=1;
    fac2=(FAC)/FAC;
end
T1=T1/N;T2=T2/N;E3=T3/N;E4=E4/N;E5=T5/N;
X=X*X*fac2;XX=X*X*facc;V=V*V*facc;N=E=N*facn;EH=E*EH;EE=EH;facc=;
xt=165/8; % xplate time [s]
tdem=15; % time delay [s]
up-facn(2)*(12-1)*1e-6; % Velocity
kv=facn; % Number of round to examine
so-cell(tt(k)/st)*st; % memory switch time [s]
figure(1)
clf
subplot(1,2,1)
plot(tt,xx,'r',tt,xx,'b') % tt,xx
xlabel('tt [ms]')
if unit=1
    ylabel('it [inc]')
else
    ylabel('it [Inc]')
end
legend('low-res encoder', 'fitted position')
ax1=high-res encoder';
subplot(1,2,2)
plot(tt,xx-xx,'r')
hold on
plot(tt(tt1(tt)),mean(xx-xli)*[1 1], 'b')
xlabel('tt [ms]')
if unit=1
    ylabel('relative angular position [rad]')
else
    ylabel('relative angular position [inc]')
end
legend('fitted position minus low-res encoder', 'mean-value of difference')
end
figure(2)
cif
clf
plot(tt(tt1(tt)),diff(xx)/st, 'r', tt(tt1(tt))-1, diff(xx)/st, 'b')
xlabel('tt [ms]')
if unit=1
    ylabel('angular velocity [rad/s]')
else
    ylabel('angular velocity [Inc/s]')
end
legend('d/dt of encoder position', 'd/dt of fitted position')
```
Appendix E

C-code to fit a straight line: main.c
APPENDIX E. C-CODE TO FIT A STRAIGHT LINE: MAIN.C

FILE MAIN.C
AUTHOR DJH Brujin, 02-May-2002
Calculating a 1st order polynomial fit through 4 points.

***************************************************************************/
#include "unix49.h"
#include "e49.h"
#include "put40.h"
#include "rpm40.h"
#include "ds949.h"
#include <file.h>
#include <assert.h>
#include <stdio.h>
#include <intrc40.h>

/** globals */
volatile LONG InputReady = 0; /* set by interrupt handler */
volatile LONG OutputReady = 0; /* set by interrupt handler */

/** externals */
extern LONG InterruptVectorTable[IVT_SIZE];
extern void SelectExternalInterrupt(LONG ExternalIntPin);
extern void EnableExternalInterrupt(LONG ExternalIntPin);
extern void DisableExternalInterrupt(LONG ExternalIntPin);

#define RECORDSIZE 3 /* 3 longwords from SRI */

typedef struct {
    LONG qcount;
    LONG timehl;
    LONG tlmele;
} SRI_RECORD;

/*******************************************************************************/
FUNCTION void c-into3(void)
interrupt handler for IIOFI* pin if memory A/B signals data available to the DSP.
Sets global variable InputReady.
*******************************************************************************/
void c_int3(void)
{
    INT_DISABLE(); /* reset GIE bit in status register */
    InputReady = 1;
    INT_ENABLE(); /* set GIE bit in status register */
}

/*******************************************************************************/
FUNCTION void c_int4(void)
interrupt handler for IIOFI* pin if memory C/D signals data available to the DSP.
Sets global variable OutputReady.
*******************************************************************************/
void c_int4(void)
{
    INT_DISABLE(); /* reset GIE bit in status register */
    OutputReady = 1;
    INT_ENABLE(); /* set GIE bit in status register */
}

#define DEBUG_SRI 0
APPENDIX E. C-CODE TO FIT A STRAIGHT LINE: MAIN.C

```c
int main() {
    static LONG nwords; /* number of words to copy */
    static LONG ix; /* general purpose */
    static LONG data; /* general purpose */
    static LONG *src; /* pointer to source buffer, memory A/B */
    static LONG *dst, *dstl; /* pointer to destination buffer, memory C/D */
    static LONG inmemsize = 0;
    static LONG outmemsize = 0;
    static LONG offset = 0;
    volatile LONG *blkoutreg = (LONG *)DSP-BLKOUT; /* pointer to blk out register */
    volatile LONG *blkinreg = (LONG *)DSPSLKIN; /* pointer to blk in register */
    volatile LONG *dspcsr = (LONG *)DSP-CSR; /* pointer to dsp csr */
    static SRI-RECORD *recptr;
    static LONG nrecords;
    static LONG switchcount = 0;

    /* -----------------variables used for fit-algorithm-----------------*/
    static const LONG ndt = le5; /* Time stamp 
[tls] per sample time of host */
    static LONG TUX = 0; /* time stamp high of first time T of fit */
    static LONG TLX = le5/10 -1e5; /* Specified time delay in register 0x26 of SRI, 
resolution 50ns*/
    static LONG RB[15] = 10,0,0,0,0,0,0,0,0,0,0,0,0,0,0); /* buffer-memory useful when 
nwords<l5*/
    static LONG *RBend = &RB[14]; /* pointer to last word of RB-memory */
    static SLONG X5; /* first position in RB-memory */
    static LONG c1,c2; /* temporary variables used for XOR-function */
    static LONG TH4, TL4; /* 4th time stamp */
    static LONG n[4]; /* time after 4th time stamp */
    static LONG *ptl, *pt2, *RB3i; /* temporary pointers */
    static SLONG x[5] ={0,0,0,0,0),xx[4]; /* conditioned time stamp variables */
    static SLONG pp,p0,p1; /* polynomial parameters */
    static SLONG cl; /* used for "for"-statements */
    static LONG nwords=0, maxnwords=10; /* variables used when nwords=0 */
    int nowraped=17; /* amount of words to copy to the C/D-memory */

    MAKE-IACK();
    //printf("SRI/DSP test program\n");
    //uart_Init();
    inmemsize = nwp_InMemSize();
    outmemsize = nwp_OutMemSize();
    //printf("inmemsize = %d, outmemsize = %d\n", inmemsize, outmemsize);
    INT_DISABLE(); /* disable all interrupts */

    net_intpt ((void *) &InterruptVectorTable); /* install IVT */
    install_int_vector((ISR-FnPtr) c_int03, IIOFL_INT_NUM); /* set ISR address for IIOF0* pin in IVT */
    install_int_vector((ISR-FnPtr) c_int04, IIOFL_INT_NUM); /* set ISR address for IIOF1* pin in IVT */
    ix = lif_Value(); /* read IIF register to see its state */
    /* set up Interrupt Flag Register (IF) to enable IIOF0* and IIOF1* pins 
to function as interrupt pins to the DSP. */
    SelectExternalInterrupt(0);
    SelectExternalInterrupt(1);
    EnableExternalInterrupt(0);
    EnableExternalInterrupt(1);
    /* end of setup */
    ix = lif_Value(); /* read IIF register to see its state */
    INT_ENABLE(); /* enable all interrupts */
    InputReady = 0;
    OutputReady = 0;
    while (1) { /* endless program loop */
```

48
APPENDIX E. C-CODE TO FIT A STRAIGHT LINE: MAIN.C

```c
/* dspcsr */

void APPENDLX
E. C-CODE TO FIT A STRAIGHT LINE: MAIN.C

#include <stdio.h>
#include <stdlib.h>

#define MAXWORDS 100

int main()

int inputReady = 1;
int outputReady = 0;
int nwords = 0;
int switchcount = 0;
int src = (LONG *) IN_MEM_BASE;
int recptr = (SRI_RECORD *) src;
int dst = (LONG *) OUT_MEM_BASE;
int dstl = dst;
int blkoutreg = (LONG) dst;

while (inputReady == 1) /* wait for input data */

inputReady = 0; /* clear for next round */

switchcount++;

nwords = blkoutreg; /* memory switching used, see section 7.3 of MPM manual */

if (GENRL_SRI == 1)

printf("pass #4! %d longwords from SRI\n", switchcount, nwords);

/* source memory A/B starts at IN_MEM_BASE if forced memory switching is used: */

src = (LONG *) IN_MEM_BASE; recptr = (SRI_RECORD *) src;

for (ix = 0; ix < nrecords; ix++, recptr++)

printf("pass #4d - record #2d: count = %8d, timehi = %8d, timelo = %8d\n", switchcount, ix, recptr->qcount, recptr->timehi, recptr->timelo);

/* get address of destination memory C/D */

dst = (LONG *) OUT_MEM_BASE + outmemsize - nwords*2CD;

dstl = dst;

if (GENRL_SRI == 1)

printf("src: 0x%08x, nwords: %d\n", src, nwords);

printf("dst: 0x%08x\n", dst);

/* for test: overwrite C/D output memory with non-SRI data: */

for (ix = 0; ix < nwords*2CD; ix++, dstl++)

*dstl = 0xffffffff;

/* update time to calculate X */

TLX = TLX + ndt;

if (TLX >= 0xffffffff - nrX)

for (i = 0; i < 12; i++)

*(RB+i) = *(RB+3+i);

else if (nwords == 3) {

nowords = 0;

for (i = 0; i < 12; i++)

*(RB+i) = *(RB+3+i);

/* update record-buffer */

if (nwords == 0) {

nowords++;

if (nwords > maxnowords) {

for (i = 0; i < 12; i++)

*(RB+i) = *(RB+3+i);

}

else if (nwords == 1) {

nowords = 0;

for (i = 0; i < 12; i++)

*(RB+i) = *(RB+3+i);

}

else if (nwords == 2) {

nowords = 0;

for (i = 0; i < 12; i++)

*(RB+i) = *(RB+3+i);

}

}

}
```
APPENDIX E. C-CODE TO FIT A STRAIGHT LINE: MAIN.C

for(i=0; i<3; i++) *(RBend+i) = *(src+2-i); 

else if (nwords == 6) 
    nowords=0; 
    for(i=0; i<9; i++) [RBti] = *(RB+G+i); 
    for(i=0; i<6; i++) *(mend-i) = *(src+5-i); 

else if (nwords == 9) 
    nowords=0; 
    for(i=0; i<3; i++) *(RBend+i) = *(RB+12+i); 
    for(i=0; i<12; i++) *(mend-i) = *(src+11-i); 

else if (nwords == 12) 
    nowords=0; 
    for(i=0; i<3; i++) *(RBend+i) = *(RB+12+i); 
    for(i=0; i<12; i++) *(mend-i) = *(src+11-i); 

else 
    nowords=0; 
    for(i=0; i<15; i++) *(RBend+i) = *(src+words-1-i); 

/* conditioning data from RB-memory */
TH4 = *(RB+4); 
TL4 = *(RB+5); 
S5 = *RB; 
for(i=3; i>1; i--) 
    RB3i = *(RB3i-2); 
    c1 = *(RB3i-2); 
    c2 = *(RB3i-5); 
  if ((c1 > c2) ^ (-c1 == c2)) [x[i] = x[i+1] + 1; 
  else 
    [x[i] = x[i+1] - 1; 

xx[i] = x[i] = x[i+1]; 
if (*(RB3i-1) == TH4) [n[1] = *RB31 - TL4; 
else 
    [n[1] = *RB31 + TL4 + 1; 

/* calculation of 1st order polynomial */
p0 = 4*a - b*b; 
p1 = 4*c - b*d; 
p2 = d*a - b*c; 

/* datacopy to CD-memory */
*dst++=n[0]; 
*dst++=n[1]; 
*dst++=n[2]; 
*dst++=-b; 
*dst++=-c; 
*dst++=-d; 
*dst++=(RB+2); 
*dst++=(RB+5); 
*dst++=(RB+6); 
*dst++=(RB+11); 
*dst++=(RB+14); 
*dst++=mp; 
*dst++=p0; 
*dst++=p1; 
if (THX==TH4) 
    [*dst++ = TLX-TL4; 
else 
    [*dst++ = TLX-TL4+1; 

/*dspcsr = IPR; /* enable switching of input memory A/B */
/*dspcsr = OPR; /* signal output memory C/D ready, sets DONE bit in PSW */
} /* end while 1 */
Appendix F

Application for host: srimain01.c
# APPENDIX F. APPLICATION FOR HOST: SRIMAIN01.C

```c
/**
   * T1000CS/3 Parallel Sampling ADC (PARSAM) BLN 2000-14
   * FILE   T1000CS/3.TST2014.C
   * AUTHOR R. Smets, 11-apr-2001
   * 
   * #include <windows.h>
   * #include <assert.h>
   * #include "mp3642.h"
   * #include "Nchlib.c"
   * #include "proto.h"
   * #include "Tdlib Inc.c"
   *
   * // gettime prototypes
   * int GetFreq(void);
   * int ResetTime(void);
   * double GetTime(void);
   *
   * #define QUADRATURE_STATUS_WORD_REGISTER 0x030
   * #define QUADRATURE_MAINTENANCE_REGISTER 0x031
   * #define QUADRATURE_COUNTER_CHANNEL_0 0x032
   * #define QUADRATURE_COUNTER_CHANNEL_1 0x034
   * #define PI 3.14159265358979
   * #define FACT 25.90914775834578
   * #define MAX_LOOP 10
   *
   * WORD Base = MPM_DEFAULT_BASE+0x1000;
   * FILE *fp;
   *
   * ROUTINE void mpm_Reset(void)
   *
   * ROUTINE ULONG mpm_CheckMemoriesPresent(WORD MemNum);
   * memory bank A..D == 0..3.
   * Memory A and B are coupled (same for C and D)
   *
   * ULONG mpm_CheckMemoriesPresent(WORD MemNum)
   * {
   *   WORD bitnr;
   *   ULONG r;
   *   ULONG size = 0;
   *   ULONG sizecode;
   *   ULONG present = 0;
   *   assert(MemNum <= 3);
   *   MemNum /= 2; /* (memory A==B) = 0, (memory C==D) = 1 */
   *   r = GetLongPB(Base + MPM_CSR);
   *   bitnr = 19 + 4*MemNum; /* checks bits 19, 23 */
   *   present = (r & (0x1 << bitnr));
   *   if (present) {
   *     sizecode = ((r >> (bitnr - 3)) & 0x7); /* get size bits */
   *     size = 0x1 << (sizecode + 7); /* get size in kwords */
   *   }
   *   return size;
   * }
   *
   * ROUTINE void mpm_ResetDSP(void)
   *
   * ROUTINE void mpm_ResetDSP(void) {
   *   PutWordPB(Base + MPM_PSW, RST);
   * }
   */
```
**APPENDIX F. APPLICATION FOR HOST: SRMAIN01.C**

```c
ULONG csrval;
if ((GetLongPB(Base + MPM_CSR) & DSPP) == 0) {
    printf("SRI error - DSP not present, cannot reset that thing!\n");
    PressEnter();
    return;
}
csrval = GetLongPB(Base + MPM_CSR);
csrval = RDSP;
PutLongPB(Base + MPM_CSR, csrval); /* reset DSP */

/*ROUTINE void mpm_ResetDSP(void)
 ***********************************************/
WORD DSPLoadFile(const char *FileName, WORD Verbose, ULONG *NBytes) {
    FILE *dspfile;
    BYTE b;
    ULONG n;
    ULONG ix = 0;
    ULONG present = GetLongPB(Base + MPM_CSR) & DSPP;
    assert(FileName != NULL);
    assert(NBytes != NULL);
    *NBytes = 0; /* returns number of loaded bytes */
    if (present == 0) {
        return 1;
    }
    mpm_ResetDSP(); /* reset DSP, DSP waits for program download now */
    dspfile = fopen(FileName, "rb");
    if (dspfile == NULL) {
        return 2;
    }
    assert(dspfile != NULL);
    if (Verbose) {
        printf("loading %s.", FileName); /* load the DSP */
        FlushStdOut();
    }
    n = fread(&b, 1, 1, dspfile);
    while (n != 0) {
        PutWordPB(Base + MPM_DSP_LOAD, b);
        ix++;
        if (Verbose) {
            if ((ix % 1000) == 0) {
                printf(".");
                fflush(stdout);
            }
        }
        n = fread(&b, 1, 1, dspfile);
    }
    *NBytes = ix;
    if (Verbose) {
        printf(" done (%d bytes)\n", ix);
        n = fclose(dspfile);
    }
    if (n != 0) {
        return 3;
    }
    assert(n == 0);
    return 0;
}
```
APPENDIX F. APPLICATION FOR HOST: SRMAIN01.C

/** 
 * APPLICATION FOR HOST: SRMAIN01.C 
 */

FUNCTION void mpm_WaitForEmulator(void)
*******************************************************************/

void mpm_EmulatorMessage(void)
{
  printf("SR1 - load DSP program with the XDS510 emulator now!\n\n");
}

ROUTINE ULONG mpm_ReadiiemoryData(double *, double *, ULONG NLongs)
*******************************************************************/

ULONG mpm_ReadMemoryData(double *psrlpos, double *psrivel, ULONG NLongs)
{
  ULONG err = 0;
  ULONG ix;
  ULONG data32;
  ULONG pp, p0, pl, X5;
  ULONG tt, Z0, n1, n2, as, db, cc, d6;
  ULONG d1, d2, d3, d4, d5;
  double dp0, dtt, dpi, dX5, dpp;
  n0 = GetLongPB(Base + MPM-PBDR);
  n1 = GetLongPB(Base + MPM-PBDR);
  n2 = GetLongPB(Base + MPM-PBDR);
  as = GetLongPB(Base + MPM-PBDR);
  bb = GetLongPB(Base + MPM-PBDR);
  cc = GetLongPB(Base + MPM-PBDR);
  dd = GetLongPB(Base + MPM-PBDR);
  d1 = GetLongPB(Base + MPM-PBDR);
  d2 = GetLongPB(Base + MPM-PBDR);
  d3 = GetLongPB(Base + MPM-PBDR);
  d4 = GetLongPB(Base + MPM-PBDR);
  d5 = GetLongPB(Base + MPM-PBDR);
  p0 = GetLongPB(Base + MPM-PBDR);
  p1 = GetLongPB(Base + MPM-PBDR);
  tt = GetLongPB(Base + MPM-PBDR);
  X5 = GetLongPB(Base + MPM-PBDR);
  fprintf(fp,"%d %d %d %d %d %d \n");
  dp0=(double) p0;
  dtt=(double) tt;
  dpi=(double) pl;
  dX5=(double) X5;
  dpp=(double) pp;
  if (dpp==0.0) {
    printf("SR1: reconstruction not available.\n\n");
    *psrlpos=-t;
    *psrivel=-t;
  } else {
    *psrlpos=0.5*(dp0*(dtt-0.5)+dpi)/dpp+dX5;
    *psrivel=0.5*(dp0/dpp);
  }
  return(err);
}

ROUTINE void mpm_ShowConfig(void)
*******************************************************************/

void mpm_ShowConfig(void) {
  ULONG r;
  ULONG memnum;
  ULONG memsize;
  char memnames[8];

54
strcpy(memnames, "ABCD");

r = GetWordPB(Base + MPM_IID);
printf("\n\nMPM Interface Identification Register = 0x%04x\n", r);
printf("MPPM piggyback code : 0x%lx\n", (r >> 4) & 0x00ff);
printf("MPPM revision code : 0x%lx\n", r & 0x000f);

r = GetLongPB(Base + MPM_CSR);
printf("\n\nSR1 - Control and Status Register = 0x%08x\n", r);
for (memnum = 0; memnum < 4; memnum++) {
memsize = mpm_CheckMemoriesPresent(memnum);
memsize /= 128; /* convert to blocks, 1 block = 128 KWords */
printf("\n\nMemory %c %s present\n", memnames[memnum], (memsize == 0) ? "not" : "");
if (memsize > 0) {
printf("size = %d 32-bit KWords (%d block(s))\n", memsize * 128, memsize);
printf("\n\n\n");
}
}

FUNCTION ULONG sri_Setup(ULONG SimClock, ULONG SwMem, ULONG TimeDelay)
FUNCTION ULONG mpm_WaitForHartBeat(void)
APPENDIX F. APPLICATION FOR HOST: SRMAIN01.C

WORD pswval;

printf("SRI - waiting for DSP heartbeat (HBT)... ");
flushStdOut();
/* wait until DSP set HBT bit */
do {
    pswval = GetWordPB(Base + MPM_PSW);
    // if (StopTest()) {  
    // printf("SRI - waiting for HBT aborted by user break\n");  
    // return 1;  
    // }
    while ((pswval & HBT) == 0);
    return 0;
}  

FUNCTION main()
******************************************************************************************

int main(void) {
    int ok;
    double enc0, enc1, t, dt;
    double qref, vref, kp, kd, penc0, vel0;
    HANDLE hprocess, hthread;
    double srivpos, srivel;
    ULONG enc_data0, enc_data1;
    int i;

    ULONG result;
    ULONG scsr;
    ULONG swmem;
    ULONG timedelay;
    ULONG ongcount;
    ULONG err;
    ULONG looking;
    ULONG ongval;
    ULONG nbytes;
    WORD verbose = 0;
    LONG dummy;
    WORE errval;

    Base = 0x800; /* Rene's crate, leftmost slot */
    printf("SRI - starting MPM / SRI demo application\n");
    printf("SRI - expecting MPM at TUeDACS base address 0x%03x\n", Base);
    // open file for logging data
    fp=fopen("srirain01.log", "w");

    lib_InitStdioWindow();

    // result = OpenPo();
    // if (result != PB_OK) {
    //    printf("SRI error - cannot open TUeDACS bus, application stopped\n");
    //    PressEnter();
    //    exit(1);
    // }
    // pbGetSocketMap(&socketmap);
    // for (i=0;i<4;i++) {
    //    printf("Socketmap.open[%d] = %d\n", i, socketmap.open[i]);
    // }
    // initialize qad
    if (qad_Init(i)<10) {
        printf("TUeDACS error: QADs could not be initialized.\n");
        return 1;
    } else {
        printf("TUeDACS/1 QADs ready.\n");
    
    56
APPENDIX F. APPLICATION FOR HOST: SRIMAIN01.C

```c
// enable dac channels on qad link
dac_active();

// set process and thread priority
hprocess = GetCurrentProcess();
hthread = GetCurrentThread();
SetPriorityClass(hprocess, REALTIME_PRIORITY_CLASS);
SetThreadPriority(hthread, THREAD_PRIORITY_TIME_CRITICAL);

// setpoint
gref = 0;

// setpoint velocity in counts/s
vref = 10;

// control
w = 0;
kp = 1;
kd = 0.01;
kd = 0.0001;
penc0 = 0;

GetFreq();
PutWordPE(Base + SRI_SW, Ox0000);

mpm_Reset();
mpm_ResetDSP();
mpm_ShowConfig();

//==============================================================================
sscr = 12; /* T_sim = 2**(sscr-2) us (simulation mode) */
dt = 0.01; /* sampletime [s] */
sf = (ULONG) 1e7*dt; /* sampletime [100ns - le-7s] */
timedelay = swmem/10; /* time delay [100ns - le-7s] */

//==============================================================================
result = sri_Setup(sscr, swmem, timedelay);
if (result != 0)
{
    printf("SRI error - sri_Setup failed!
");
    Sleep(2000);
}

// clear SRI counter
PutWordPB(Base+0x30, OxEO);

// clear QAD counters
PutWordPB(QUADRAT_PHYBUS_STATUS_WORD_REGISTER+0x96, OxEO);

printf("Counters have been cleared!!!
");

#define COMLOAD 1

if (COMLOAD == 0)
/
// load the DSP for debugging via JTAG now. Make sure the DSP sets
// the HBT bit.
+
/*
mpm_EmuatorMessage();
stopflag = mpm_WaitForHartbeat();
printf("\n");
if (stopflag != 0) {
    printf("SRI error - no DSP hartbeat detected !\n");
} else {
    printf("SRI - Yo! DSP alive and kicking !\n");
} 

PressEnter();

else /* COMLOAD == 1 */
```

57
APPENDIX F. APPLICATION FOR HOST: SRIMAIN01.C

```c
verbose = 1;
nbytes = 0;
result = DSPLoadFile("sri.c40", verbose, &nbytes);
printf("nbytes=%d\n", nbytes);
stopflag = mpm_WaitForHartBeat();
printf("n");
if (stopflag != 0) {
    printf("SRI error - no DSP hartbeat detected !\n");
} else {
    printf("SRI - Yo! DSP alive and kicking !\n");
}
PressEnter();
}
```

```c
for (loopcount = 0; loopcount < 1000; loopcount++) {
    FlushStdOut();
    stopflag = 0;
    /* wait for DSP processing, copy of input data should be in memory C/D.
     * PFR bit is set by timer of SRI.
     */
    do {
        csrval = GetLongPB(Base + MPM_CSR);
    } while ((csrval & PFR) == 0);
    PutLongPB(Base + MPM_CSR, PFR | MPM_MODE_3); /* let 'm rip !!!! */
    ResetTime();
    t-timeDelay('e');
    /***************************************************************************/
     */
    for (loopcount = 0; loopcount < 1000; loopcount++) {
        FlushStdOut();
        stopflag = 0;
        /* wait for DSP processing, copy of input data should be in memory C/D.
         * PFR bit is set by timer of SRI.
         */
        do {
            csrval = GetLongPB(Base + MPM_CSR);
        } while ((csrval & PFR) == 0);
        PutLongPB(Base + MPM_CSR, PFR | MPM_MODE_3); /* clear PFR bit for next round */
        /* DONE bit MUST (!) have been set by writing OPR bit in DSP CSR */
        do {
            pswval = GetWordPB(Base + MPM_PSW);
        } while ((pswval & DONE) == 0);
        if (pswval & ERR) {
            errval = GetWordPB(Base + MPM_ERRID);
            printf("SRI error - ERR bit in PSW is set\n");
            printf("Error Identification Register: Ox%04x\n", errval);
        }
        read enc values
        pbGetLong(Base+0x32,&enc_datal);
        enc1=(signed long)enc_datal;
        pbGetLong(QUADRATURE_COUNTER_CHANNEL_0+4096,&enc_data0);
        enc0=(signed long)enc_data0;
        //
        pbGetLong(QUADRATURE_COUNTER_CHANNEL_1+4096,&enc_datal);
        enc1=(signed long)enc_datal;
        //
        t=GetTime();
        printf("t=%f \n",t);
        //
        update qad channels
        if (qadUpdate(!t)==0) {
            printf("TDeSACS error: ADC timeout, RTA stopped.\n");
            return 1;
        }
    }
```
APPENDIX F. APPLICATION FOR HOST: SRMAIN01.C

qref=vref't;
vel0=(enc0-enc0)/dt;

w=kpr*(qref-enc0)+kdv*(vref-vel0);

write dac values

u=0.07;
dac-write(u,0,l);

in DIRECTDAC mode 2nd dac actually writes value to dac...
dac-write(0.0,1,l);

nlongstoread = GetLongPB(Base + MEM_PBLEN_CD) + 1;

printf("SR1 - reading %lu longwords from MEM\n",nlongstoread);

err = *mpm_ReadMemoryData(sripos,srivel,nlongstoread);

err += mpm_ReadMemoryData(enc0,enc1,nlongstoread);

exit qad
qad-exit();

close log file
fclose(fp);

result = ClosePb();
if (result != PB_OK) {
   printf("SR1 error - closing of TIODACS bus failed\n");
}

printf("SR1 - stopping MEM / SRI demo application\n");