Controlling the H-drive

Controlling the H-Drive

A.F. Rovers

February 2002
DCT Report nr. 2002.12

TU/e – Practical Traineeship Report

Coaching:
Dr. Ir. M.J.G. van de Molengraft

Eindhoven University of Technology
Faculty of Mechanical Engineering
Dynamics and Control Technology Group

Contents

Preface

This report is written to finalize my traineeship in the Control Systems Technology group at the Eindhoven University of Technology. The traineeship had two goals:

1. Getting the H-Drive in the laboratory of the group operational.
2. Studying an alterative to the present zero-search method that is based on a vibrational procedure designed by R. Beijenberg. The method to be designed could also use prior knowledge of the system in order to obtain a faster and accurate zero-determination.

This report first describes the working of LiMMS systems like the H-Drive and the stages that are needed to align the drive. Next the controllers and the safety-layer are studied in detail. At the end of the report, an alternative zero-search procedure based on a Kalman-filter is explored.

During the traineeship the final reports of Antoine Verweij and S.G.M. Hendriks proved to be an useful source of information.

Chapter 1

Introduction to the H-Drive

The first chapter contains an introduction to the H-Drive. First the principle of the Linear Motion Motor System that drives the H-Drive, is explained. After that, relevant system parameters are identified and the various sensors attached to the robot are described.

1.1 Principle of the Linear Motion Motor System (LiMMS)

Linear Motion Motor Systems (LiMMS) are composed of two parts, a stator and translator, as depicted in figure 1.1. The stator consists of a set of permanent magnets that are placed on a metal strip with a constant pitch. The translator contains a set of iron-coils that act like electromagnets when a current is supplied. Attracting and repelling forces between the magnets of the stator and translator result in a thrust force that sets the system in movement. To derive the equations of motion for the complete LiMMS, the resulting force on one coil is determined first.

Using the definitions of current and electromagnetic field in combination with Faraday's law of induction yields:

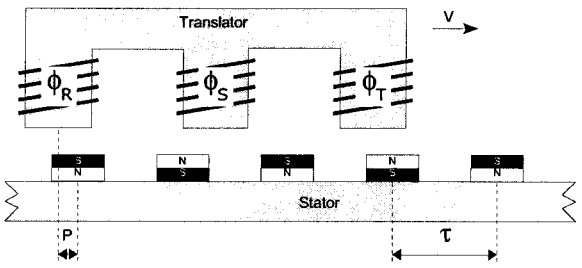

Figure 1.1: Three phase synchronous permanent-magnet linear motor

$F_{ph,n}$	force of one phase (for coil number n)	[N]
n	index of coil	[-]
K_{ph}	motor constant for one phase	[N/A]
$i(t)$	current through one coil	[A]
$\widehat{I}$	amplitude of current through the coils	[A]
$\psi(t)$	coupled flux	[Vs]
$\widehat{\psi}$	amplitude of the coupled flux	[Vs]
p	position offset of the pole shoe with respect to the permanent magnets (before initialization, converted from [m] to [rad])	[rad]
τ	magnet pitch (distance between N-S pole)	[m]
φ	current angle offset	[rad]
x	position of the LiMMS	[m]

Table 1.1: Symbols and variables

$$F_{ph,n} = -\frac{d\psi}{dx} i_n \tag{1.1}$$

The coupled flux varies with position according to equation 1.2. The resulting horizontal force will be zero (either repelling or attracting) when the coil is positioned exactly above a magnet. The force will be maximal when the coil is halfway between two magnets.

$$\psi_n = \widehat{\psi} \cos(\frac{\pi x}{\tau} + p_n) \tag{1.2}$$

The force $F_{ph,n}$ on one coil n also varies with the current that is supplied to the coil. When deriving the equation of movement for the whole stator further on in this section, it will be seen that it is useful to make the current position dependent, with a user controllable offset φ_n.

$$i_n = \widehat{I} \cos(\frac{\pi x}{\tau} + \varphi_n) \tag{1.3}$$

Combining above equations results in equation 1.4 with $K_{ph,n} = \frac{\pi}{\tau}\widehat{\psi}$.

$$F_{ph,n} = \frac{1}{2}\widehat{I} K_{ph,n} \left\{ \cos(p_n - \varphi_n) - \cos(\frac{2\pi x}{\tau} + p_n + \varphi_n) \right\} \tag{1.4}$$

The LiMMS is constructed such that the difference in position offset, converted from meters to radians[1], between two adjacent coils is $\frac{2}{3}\pi$. In a three-phase motor the phases of the currents to the different coils are also shifted $\frac{2}{3}\pi$ radians:

$$\begin{array}{lll} \text{phase 0:} & p_{ph,0} = p & \varphi_{ph,0} = \varphi \\ \text{phase 1:} & p_{ph,1} = p + \frac{2}{3}\pi & \varphi_{ph,1} = \varphi + \frac{2}{3}\pi \\ \text{phase 2:} & p_{ph,2} = p + \frac{4}{3}\pi & \varphi_{ph,2} = \varphi + \frac{4}{3}\pi \end{array} \tag{1.5}$$

[1] Conversion from meters to radians: $p_{[rad]} := p_{[m]} \cdot \pi/\tau$

Using equation 1.4 and the settings from 1.5 yield:

$$\begin{array}{ll} \text{phase 0:} & F_{ph,0} = \frac{1}{2}\widehat{I}K_{ph}\left\{\cos(p-\varphi) - \cos(\frac{2\pi x}{\tau} + p + \varphi)\right\} \\ \text{phase 1:} & F_{ph,1} = \frac{1}{2}\widehat{I}K_{ph}\left\{\cos(p-\varphi) - \cos(\frac{2\pi x}{\tau} + p + \varphi + \frac{4}{3}\pi)\right\} \\ \text{phase 2:} & F_{ph,2} = \frac{1}{2}\widehat{I}K_{ph}\left\{\cos(p-\varphi) - \cos(\frac{2\pi x}{\tau} + p + \varphi + \frac{8}{3}\pi)\right\} \end{array}$$

The total resulting horizontal thrust force is equal to the sum of the forces on the individual coils.

$$F_{ph} = \sum_{n=0}^{2} F_{ph,n} = \frac{3}{2}\widehat{I}K_{ph,n}\cos(p-\varphi) \tag{1.6}$$

Equation 1.6 shows that by applying a position-dependent current, as described in equation 1.3, the thrust force has been made position-independent. The efficiency of the LiMMS is maximal when the offset φ of the current is equal to the initial position offset p.

Equation 1.7 shows the resulting system equation.

$$m\ddot{x}(t) = F_{ph} - F_{disturbances} \tag{1.7}$$

The disturbances are caused by:

- Cogging force, caused by the attraction between the permanent magnets and the iron cores of the LiMMS. This force is always present and tries to align the iron cores to a stable position of the translator. The effect can be modeled with a sinusoidal function with a period depending on the magnets.
- Reluctance force, caused by the varying self-inductance of the windings of the coils of the translator. This effect results in a position and velocity dependent force ripple that can only be modeled when the accuracy of the placement and magnetic tolerance of the separate magnets are known. This requires a detailed analysis of the LiMMS.
- Friction in the ball bearings between the translator and the guiding rails.

1.2 Setup of the H-Drive

The H-Drive comprises of two parallel Y-axes. The X-axis is connected to the translator that moves along the Y-axes. It is possible to attach different devices, like a Z-axis, to the translator of the X-axis. Figure 1.2 shows a schematic view of the H-Drive and the used system of coordinates.

A number of sensors is used to report special states of the H-Drive:

- end-of-stroke sensor (eos) (figure 1.3.a): two micro-switches are attached to every translator. The switches get activated when the translator bumps to the spring at the end of the axes.

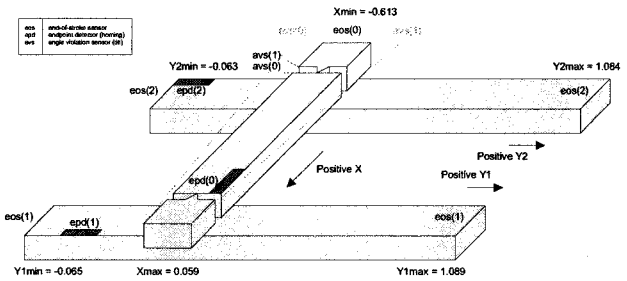

Figure 1.2: System of coordinates and location of sensors

- end-position detector (epd) (figure 1.3.b): every axis contains one metal strip. The edge of the strip coincides with the absolute origin of the coordinate system of that axis. An inductive sensor at the translator (epd) is used to detect the presence of the metal strip.

- angle violation sensor (avs) (figure 1.3.c): one set of inductive sensors guard the tilt of the X-axis with respect to the Y-axes. Initially it was assumed that sensor avs(0) gets activated when the angle gets too big in one direction and sensor avs(1) gets activated by an angle violation in the other direction. Pending the project accurate measurements showed that sensor avs(0) gets activated when the X-axis is perpendicular to the Y-axis. Only avs(1) detects an angle violation. At the opposite side of the location of the avs-sensors a connection point for two more avs-sensors is available. It looks like these sensors should be attached, but got lost during the modifications that have been made to the H-Drive to make it suitable for research.

The current for the coils is supplied by a current amplifier (figure 1.3.d).

The Controller for the H-Drive can be build in the MATLAB/Simulink system. This model is compiled and loaded into the PowerPC processor that runs independent from the PC on the dSPACE DS1130 board that takes care of the interfacing between the PC, the controller in the PowerPC processor and the hardware belonging to the H-Drive.

1.3 System identification

The transfer-functions of the system contain a lot of useful information. To determine the transfer-function[2] the system is moved at constant velocity, also called jogging, when injecting noise n on input I (figure 1.4). This is done to eliminate non-linear and position effects like cogging and friction. A weak controller $C(s)$ is used to make the translator track the desired trajectory r needed for jogging.

[2]Special thank go to Aart-Jan van der Voort for determining the transfer functions and giving permission to publish the results in this report.

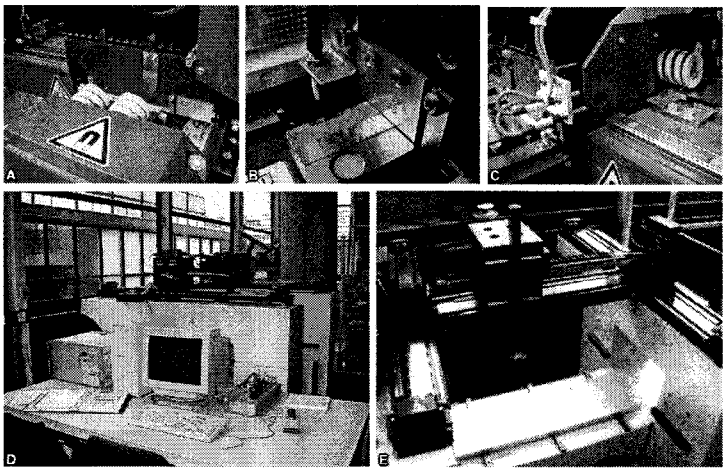

Figure 1.3: Overview of the H-Drive system: (A) EOS-sensor Y1-Axis, (B) EPD-sensor X-Axis, (C) AVS-sensor, (D) Setup with current amplifier, PC, dSPACE box and H-Drive, (E) H-Drive

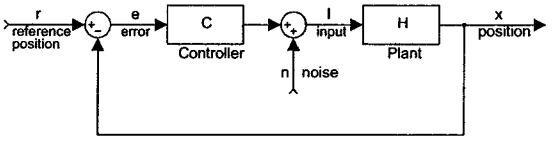

Figure 1.4: Block scheme used to determine transfer function.

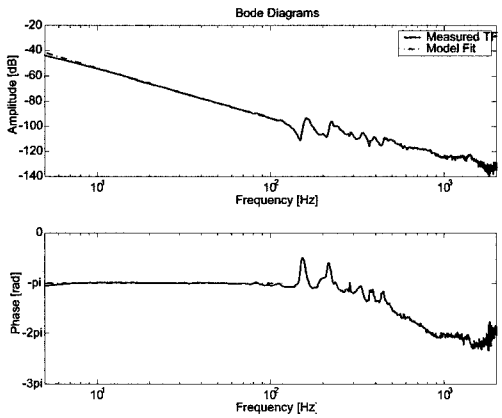

Figure 1.5: Transfer function of X-axis

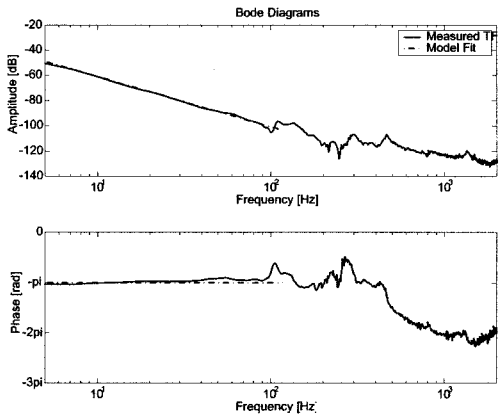

Figure 1.6: Transfer function of Y-axis

The noise n and current-input $\widehat{I}$ are measured and the transfer function $S(s)$ between these two signals is determined. Studying figure 1.4 shows that $S(s)$ can be expressed as 1.8.

$$S(s) = \frac{1}{1 + H(s)C(s)} \tag{1.8}$$

Using 1.8 the transfer-function of the H-Drive is calculated. The resulting transfer functions are displayed in figure 1.5 and 1.6.

It can been seen that the system behaves like a multi-DOF mass. Until the first mass is decoupled at the first resonance frequency, the system behaves like a single mass as can be seen in the bode-plot (gain decreases 40dB per decade if frequency is expressed in rad/sec, phase is -180 degrees). This is in accordance with theory..

The first resonance of the X-axis occurs at 161 Hz. The first resonance of the Y-axis at 110 Hz

Ignoring disturbances the equation of motion can be written as 1.9 ($p = \varphi$).

$$\begin{aligned} \ddot{x}(t) &= \frac{\frac{3}{2}K_{ph}}{m}\widehat{I} = \alpha\widehat{I} \\ H(s) &= \frac{\widehat{I}(s)}{X(s)} = \frac{1}{\alpha s^2} \end{aligned} \tag{1.9}$$

By fitting 1.9 on the first part of the transfer function where the system behaves like a single mass, parameter α can be determined.

The motor constant was calculated by S.G.H Hendriks[3] as follows. The translator is an electromechanical transducer which uses a magnetic field. Equation 1.10 shows a typical model of such a transducer.

$$\begin{aligned} U &= Ri - \frac{\partial\psi(x,i)}{\partial t} \\ &= Ri - \left(\frac{\partial\psi}{\partial i}\frac{\partial i}{\partial t} + \frac{\partial\psi}{\partial x}\frac{\partial x}{\partial t}\right) \\ &= Ri - L\frac{\partial i}{\partial t} + K_m\frac{\partial x}{\partial t} \end{aligned} \tag{1.10}$$

When moving the translator by hand the current i will be zero, but a voltage U will be generated because of the self-inductance (equation 1.11).

$$U = K_m\dot{x}(t) \tag{1.11}$$

By integrating the induced voltage over time, the change of the coupled flux can be calculated as showed in 1.12 and 1.13.

$$E = \frac{\partial\psi}{\partial x}\frac{\partial x}{\partial t} \tag{1.12}$$

$$\begin{aligned} \int_{t_1}^{t_2} E dt &= \int_{x(t_1)}^{x(t_2)} \frac{\partial\psi}{\partial x} dx \\ &= \int_{x(t_1)}^{x(t_2)} \partial\psi \\ &= \psi(x(t_2)) - \psi(x(t_1)) \end{aligned} \tag{1.13}$$

$E_{ph/ph}$ is the induced voltage measured between two of the three coils. When measuring the translator position x and the induced voltage $E_{ph/ph}$ during movement, the integrated $E_{ph/ph}(\psi_{ph/ph})$ can be plotted as a function of position, as depicted in figure 1.7. In figure 1.7 it can be seen that the magnet pitch is $\tau = 12\ mm$ and the amplitude of the coupled flux over two phases is $\widehat{\psi}(x)_{ph/ph} = 0.33\ Vs$.

[3]Thesis Report No. 2000.37 - Iterative Learning Control on the H-drive, S.G.H. Hendriks, Eindhoven University of Technology, Department of Mechanical Engineering, 20 november 2000

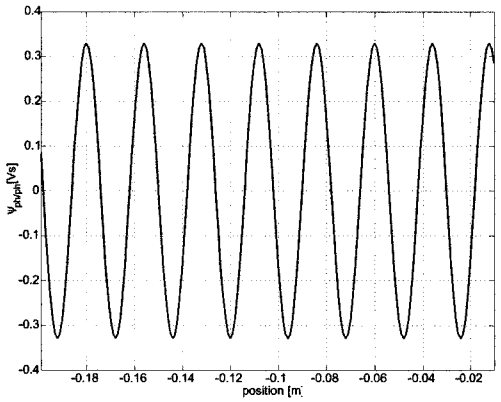

Figure 1.7: Measured $\psi_{ph/ph}$ as funtion of translator position x

The current through the coils is shifted 120° degrees in phase. Using equation 1.2 the measured flux $\psi(x)_{ph/ph}$ can be calculated.

$$\begin{aligned} \psi(x)_{ph/ph} &= \psi(x_2) - \psi(x_1) \\ &= \widehat{\psi}\cos(\frac{\pi x}{\tau} + p + 120°) - \widehat{\psi}\cos(\frac{\pi x}{\tau} + p) \\ &= \widehat{\psi}(x)\sqrt{3}\cos(\frac{\pi x}{\tau} + phase) \\ &= \widehat{\psi}(x)_{ph/ph}\cos(\frac{\pi x}{\tau} + phase) \end{aligned} \tag{1.14}$$

Using the definition of K_{ph} (the motor constant over one phase) and equation 1.14 yields 1.15.

$$\begin{aligned} K_{ph} &= -\frac{\partial\psi}{\partial x} = -\frac{1}{\sqrt{3}}\frac{\partial\psi}{\partial x} \\ &= \frac{\pi}{\tau\sqrt{3}}\widehat{\psi}(x)_{ph/ph}\sin(\frac{\pi x}{\tau} + phase) \\ \widehat{K}_{ph} &= \frac{\pi}{\tau\sqrt{3}}\widehat{\psi}(x)_{ph/ph} \end{aligned} \tag{1.15}$$

Therefore the motor constant K_{ph} can be calculated using the measurement depicted in figure 1.7: $K_{ph} = 49.6\ N/A$

Antoine Verweij[4] determined the coulomb and viscous friction of a permanent magnet linear motor by moving the translator with several constant velocities. The force needed during a constant speed movement can be calculated by multiplying the motor current with the motor constant. By plotting a trend line through the measurement results the coulomb friction and damping can be determined. These experiments have not been done on the H-drive yet, but figure 1.8 depicts how the resulting friction-speed graph could look like.

[4]Thesis Report No. EPE 2000.02 - Control of a permanent magnet linear motor with dSPACE and MATLAB/Simulink, Antoine Verweij, Eindhoven University of Technology, Department of Electrical Engineering, november 2000

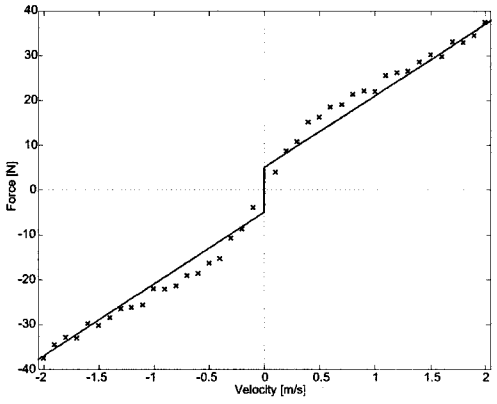

Figure 1.8: Coulomb and Viscous friction as funtion of translator velocity v. (The figure shows no real measurement data, but a result that can be expected).

Parameter	**X-axis**	**Y-axes**
First resonance [Hz]	161	110
Motor constant K_{Ph} [N/A]	49,6	49,6
Parameter α (eq. 1.9) [N/(AKg)]	8,08	3,60
Lumped mass [Kg]	9,21	20,6
Pitch between two magnets [m]	0,012	0,012
Coulom friction [N]	5,0*	5,0*
Viscous friction [Ns/m]	16*	16*

Table 1.2: Parameters of H-Drive (* = based on report of Antoine Verweij, not measured or verified by experiments)

The most important results of the system identification are listed in table 1.2.

Chapter 2

Zero-search procedure

In this chapter the original zero-search procedure as based on a patent of R. Beijenberg, is described. First the notion of an equilibrium point is explained. Subsequently the idea behind the vibration procedure is disclosed and finally a schematic view of the procedure is showed.

2.1 Stable and unstable equilibrium

The aim of the zero-search procedure is to find the initially unknown value $(p - \varphi)$ with the objective to use commutation to keep the motor constant at its maximal level (see equation 1.6). Figure 2.1 shows the resulting horizontal thrust force as function of the current angle offset φ and position offset p (FPP-Curve: Force Position Phase).

As can be seen in the in the figure, there are two points at which the force is equal to zero. Only one of these points is a stable equilibrium. This can be demonstrated with a simple virtual experiment.

Assume that at a certain fixed current angle φ the translator is positioned such that $(p - \varphi) = 45°$ and the current amplitude $\widehat{I}$ is positive. According to figure 2.1 the thrust force is positive, which causes the translator to move in the positive direction: p increases. If p increases, $(p - \varphi)$ increases and the thrust force decreases until the equilibrium at 90° is reached.

If the translator is positioned such that $(p-\varphi) = 135°$ and the current amplitude $\widehat{I}$ is positive, the resulting thrust force will be negative and the translator

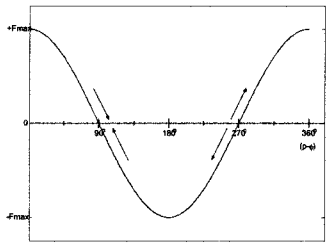

Figure 2.1: Force as function of translator position and phase of current

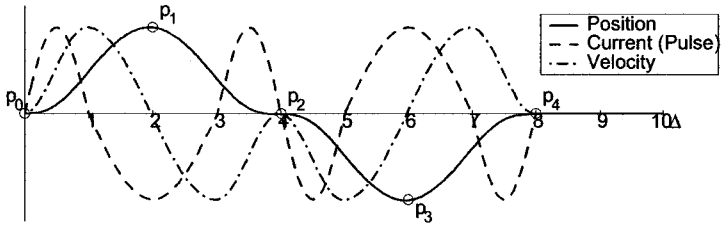

Figure 2.2: One single vibration pulse: current, velocity and position.

will also move towards the equilibrium point at 90°.

However, if the translator is positioned near $(p - \varphi) = 270°$ a small disturbance will cause the translator to drift away from the unstable equilibrium point at 270°

2.2 Vibration pulses

Theoretically, it is possible to move the translator to the position of the stable equilibrium by merely supplying the coils with a current. However, this method is not used in practice because:

- The translator could make a sudden uncontrolled movement.
- If the translator starts at the unstable equilibrium point $(p - \varphi) = 270°$, the translator would not move at all and the stable equilibrium will never be reached.

To get around these difficulties the zero-search procedure makes use of a vibration method which moves the translator as little as possible. This is achieved by supplying a pulsing vibration current to the coils which causes a vibrating movement: a movement in positive direction is followed immediately by a movement in negative direction (figure 2.2). The total pulse endures 10Δ and comprises of 6 sine parts with periods of Δ and 2Δ, followed by a pause of 2Δ. By doing so the net displacement after a vibration pulse will be approximately zero.

To keep the movement during the vibration small, the duration of the pulse[1] is kept small (typical: $\Delta = 1...10\ msec$).

During a vibration pulse offset φ is kept constant. The amount of movement, and the direction of the movement provide information about the current position on the FPP-Curve. The total amount of movement is calculated using equation 2.1. By changing $\widehat{I}$ and φ in between successive pulses, it is possible to reduce the movement to a minimum, as described in the next section.

[1]The duration of Δ can be set by changing define `DELTA` in `HD_V4_Movetest.c`

$$\begin{aligned} RESULT &= displacement_{(0\to1)} + displacement_{(1\to2)} \\ &\quad + displacement_{(2\to3)} + displacement_{(3\to4)} \\ &= (p_1 - p_0) - (p_2 - p_1) - (p_3 - p_2) + (p_4 - p_3) \\ &= -p_0 + 2p_1 - 2p_3 + p_4 \end{aligned} \tag{2.1}$$

The position is measured by an incremental encoder. Therefore a certain amount of movement is required. Interference between the LiMMS and other vibrating parts of the H-Drive also require a minimum amount of movement, called the detection level[2], before a result is reliable enough to adjust the value φ for the next pulse (typical: $detectionlevel = 10...20\mu m$).

2.3 Zero-search procedure

Figure 2.3 shows the flow-chart of the vibration procedure that is used to find the stable equilibrium.

Before the actual zero-search procedure is started a movetest is executed to make sure there is sufficient movement to retrieve useful information about the current position on the FPP-Curve. The movetest does not just serve as a self-test for the LiMMS, but also guarantees that the zero-search procedure finds the stable equilibrium.

The movetest is labelled "successful " if the total movement caused by one vibration pulse exceeds the detection level during three successive pulses. If the total movement stays beneath the detection level, the amplitude I is increased and phase φ is shifted by 90° until the test succeeds. An error message is raised when the desired detection level even can not even be reached with the maximum allowed current.

After a successful movetest, the procedure continues with the actual zero-search procedure that tries to find the stable equilibrium point on the FPP-Curve: the point where even the highest allowed amplitude of the current does not cause a significant movement. To be able to provide a position independent motor constant during each vibration period, current offset φ is kept constant and commutation is activated.

After each vibration the resulting movement is compared with the movement of the previous period and the estimated position φ is shifted an angle $d\varphi$ more towards the equilibrium point that is looked for. Pending the iteration process the step-size $d\varphi$ of the shift is made smaller. The following scheme is used to find the zero-point:

- If the resulting movement stays under the detection level, the amplitude of the current is increased (without changing φ). If the maximum amplitude is reached, the stable equilibrium is found and the zero-search procedure is stopped.
- If the resulting movement exceeds the detection level, the estimate φ is shifted to try to reduce the movement during the next vibration pulse.

[2]The detection level can be set by changing the define `DETECTION_LEVEL` in `HD_V4_Movetest.c`

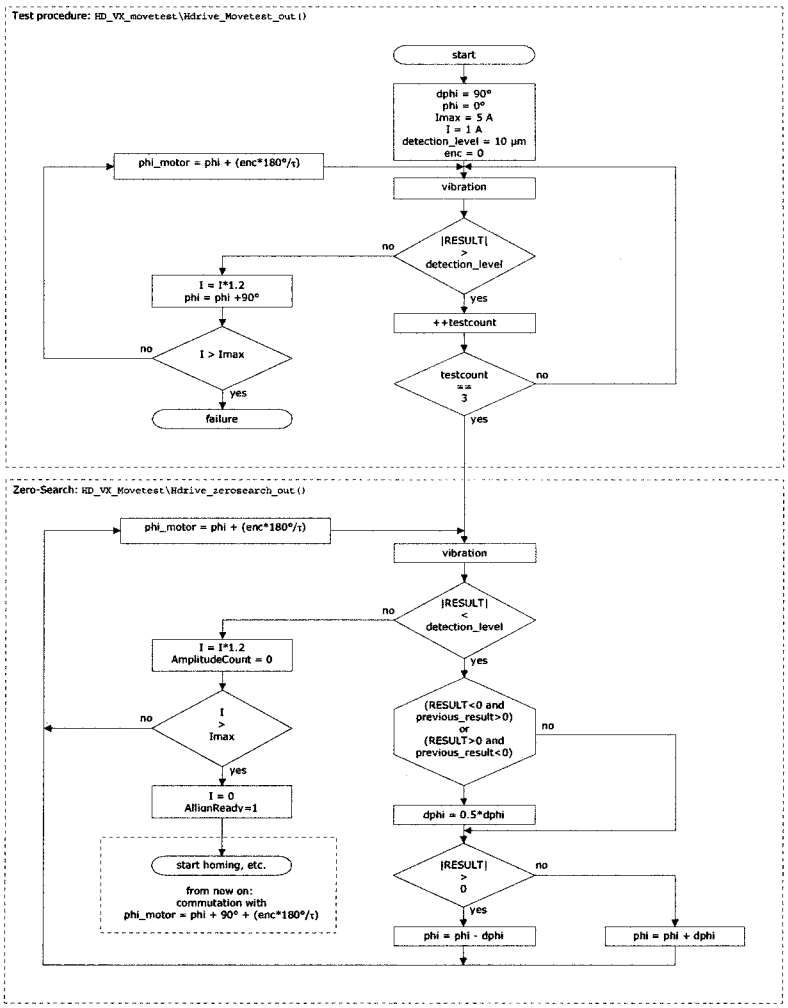

Figure 2.3: Flow chart of zero-search procedure

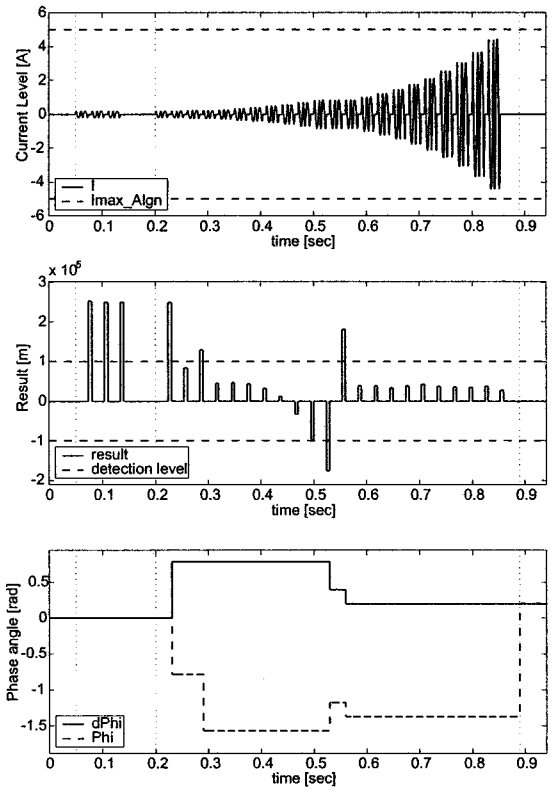

Figure 2.4: Signals during zero-search procedure

- If the sign of the current $RESULT$ is equal to the sign of the previous $RESULT$, the equilibrium point on the FPP curve was not passed during last iteration step. Therefore estimate φ is shifted an angle $d\varphi$ in the correct direction, without changing step-size $d\varphi$.
- If the sign of the current $RESULT$ differs from the sign of the previous $RESULT$, this points out that the equilibrium point on the FPP curve was passed during last iteration step. Therefore step-size $d\varphi$ is reduced and estimate φ is shifted $d\varphi$ in the correct direction.

Finally a phase of 90° is added to φ to emerge at the position where the FPP-curve has value +1, which results in a overall motor constant of $\frac{3}{2}K_{ph,i}$ (equation 1.6). Figure 2.4 shows measured signals from a random zero-search procedure.

When the axis moves too far[3] away from its initial position during zero-searching, the procedure is stopped and an error-message is issued.

The H-Drive becomes unstable when the zero-search procedure is started when the homing sensor is activated. This problem does not occur in simulations

[3]The maximum allowed drift during zero-searching can be set by changing the value of define `ZERO_MAX_DRIFT` in `HD_V4_HDrive.c`

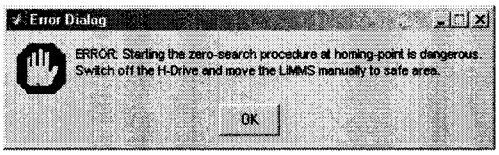

Figure 2.5: Error message caused by activated EPD-Sensor.

and is therefore probably caused by the hardware, not by errors in the H-Drive software. Because this problem could not be solved pending the period of the internship, the vibration procedure is interrupted by the software when the EPD-Sensor gets activated and an error message is issued (figure 2.5). The system would not move until the axes have been manually moved to a safe position. When moving the axes, the H-Drive has to be turned off to prevent dangerous situations to occur.

Chapter 3

Initialization of the system

This chapter describes the different stages of the initialization that are needed before the system can be operated by a user:

- Before the zero-search procedure is started a movetest is performed to ensure sufficient movement is present to start the zero-search procedure.
- In the zero-search procedure the offset p between the translator and the permanent magnets is determined. At the end of this stage it is possible to keep the motor-constant maximal by applying commutation.
- The Y-axes will be aligned in a straight position so the system can be moved without a dangerous tilt that can damage the system.
- Find the origin of the system's coordinate system by homing the three axes. After this procedure the absolute positions of the axes are known.
- Move the robot to the start position.
- Pass control to user. The user defined controller is embedded in a safety-layer that runs to protect the H-Drive for dangerous situations.

Above stages are studied in more detail in the following sections

3.1 Movetest and Zero-Search

The theory of the zero-search procedure is described in detail in chapter 2.

Experiments on the H-Drive showed that executing the vibration-based alignment procedure on one axis hardly interferes with the alignment procedure on another axis[1]. Therefore it is allowed to align the axes simultaneously to achieve the shortest initialization time possible. When the zero-search of one axis is completed, the axis waits till the zero-searches for all axes are completed before continuing to the next stage.

[1]The maximum measured displacement of one axis, when aligning one of the other axes, was 1 μm during one complete set of vibration pulses.

When the axis moves too far[2] away from its initial position during zero-searching, the procedure is stopped and an error-message is issued.

For inexplicable reasons the H-Drive becomes unstable when the zero-search is started with an activated homing sensor. Therefore an error message is issued when the user tries to start the initialization at the homing point. The system will not move until the axes have been moved manually to a safe position.

After the zero-search procedure a commutation algorithm[3] is used to keep the motor constant at its maximum value to achieve a constant, maximum efficiency.

3.2 Alignment of Y-axes

After the zero-search the X-axis has to be set in a position perpendicular to the Y-Axes before moving to the homing point to prevent the axis from wedging, caused by a tilt between the axes. Because the position of the axes is not known until the homing point has been found, it is not possible to align the Y-axes by making use of the encoder-outputs.

Therefore the avs-sensors are used to align the Y-axes. There are two procedures[4] build into the software that can be used to align the Y-axes:

- The first method keeps the translator of the Y1-axis on its initial position and moves the Y2-axis in positive direction till avs(1) is activated and stores the position of avs(1). Next the Y2-axis is moved in the negative direction till avs(0) gets activated. The position of avs(1) gets stored. Finally a PID-Controller moves the Y2-axis to the position exactly in the middle between the both avs-sensors.
  This method should be used when both avs-sensors signal a tilt in a different direction.

- The second method keeps the translator of the Y1-axis on its initial position and moves the Y2-axis in positive direction till avs(1) is activated and stores the position of avs(1). Next the Y2-axis is moved in the negative direction till avs(0) gets just activated.
  This method should be used when avs(1) detects a tilt in one direction and avs(0) detects the perpendicular position.

As long as the second set of avs-sensors has not been installed on the H-Drive the second method is used.

3.3 Homing

In the next stage all axes are moved simultaneously at a constant low velocity[5] until the homing point is detected by the epd-sensor. After this the translator

[2]The maximum allowed drift during zero-searching can be set by changing the value of define `ZERO_MAX_DRIFT` in `HD_V4_HDrive.c`

[3]The direction of the commutation (in the same or opposite direction of the coordinate system) can be set in `mdlInitializeConditions()` of `HD_V4_HDrive.c`

[4]By defining `ALIGN_Y_TO_CENTRE` in `HD_V4_HDrive.c` the first method is used, by undefining `ALIGN_Y_TO_CENTRE` the second method is used.

[5]The homing-speed can be set in function `mdlInitializeConditions()` of `HD_V4_HDrive.c`

	Value
Rise time	3.5 msec
Settling time	3.5 msec
Overshoot	1.2 percent
Bandwidth (for $m = 12$)	107 Hz
Gain margin	∞
Phase margin	91 degrees at 107 Hz

Table 3.1: Characteristics of PID-Controller for moving

stops smoothly according to a third degree setpoint and waits till all translators arrive at their homing point. Next all translators move with a very low speed (one third of the original homing-speed) till the epd is no longer detected. The global origin of the coordinate system is known and the position-value is reset in software.

3.4 Moving

A PID-Controller with a bandwidth of 20 Hz is used to make the translator follow the desired profile with a small error. The controller is tuned in such a way that the same settings[6] can be used for the X-axis and Y-axes. The Bode and Nyquist diagrams[7] are depicted in figure 3.1, 3.2 and 3.3. Table 3.1 summarizes the most important characteristics[8] of the controller. Because the gain stays in the vicinity of 0 dB for frequencies well beyond the bandwidth of the system, a relative low bandwidth is chosen to prevent problems caused by the resonance frequencies of the system.

REMARK: *When designing the controllers for the H-Drive software the mass of the three LIMMS were estimated to be 12 Kg. Later on, when the software was finished, the real masses were estimated using the system identification as described in section 1.3. The controllers in the software have not been re-tuned for the new masses. However, the bode-plots for these systems are depicted in this report to show their performance and prove stability.*

3.5 User control

The user is not allowed to have full control over the H-Drive. A safety-layer protects the system from dangerous situations. This layer is studied in detail in the next chapter.

[6]The parameters of the PID-Controller can be set in function `mdlInitializeConditions()` of `HD_V4_HDrive.c`

[7]The plots from figure 3.1, 3.2 and 3.3 are based on simulations, not on real measurements. Because sampling takes places at a relatively high frequency, lag-times caused by the discretizing the controller can be ignored.

[8]The rise time is here defined as the time it takes the system to reach the vincity of it's settling point (90% of desired end-point), the settling-time is the time it takes the system to stay in the vincity of it's settling point (10% deviation of desired value) and the overshoot is the maximum amount the system overhoots it's final value (expressed as a percentage).

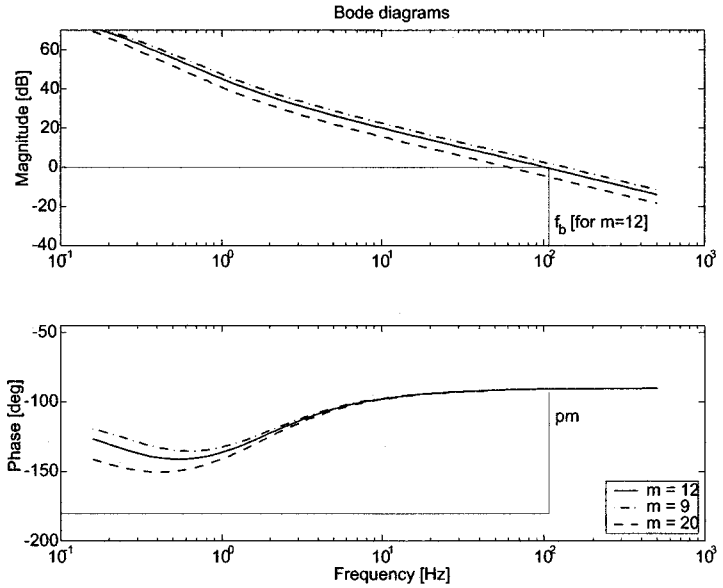

Figure 3.1: PID Controller - Bode plot of open loop system (from I to x)

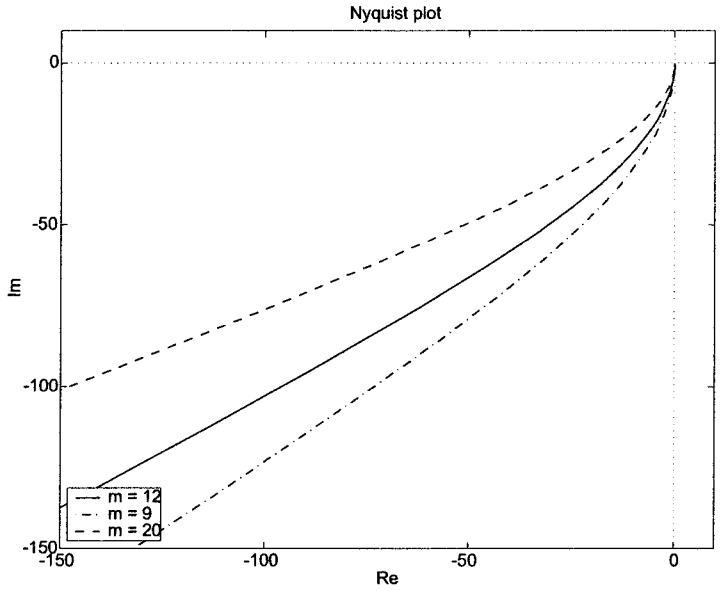

Figure 3.2: PID Controller - Nyquist plot of open loop system

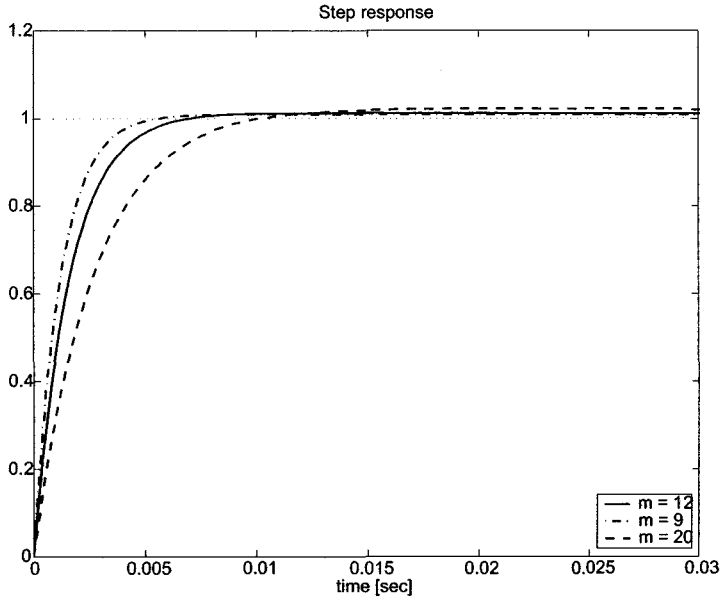

Figure 3.3: PID Controller - Step response of system (digitally controlled at 5 kHz)

Chapter 4

Safety layer

To protect the H-Drive form dangerous events during operation, the system is safeguarded by a safety layer that takes over control from the user when necessary. The different features of the safety layer are explained in this chapter.

4.1 Components of the safety-layer

Because several violations can occur simultaneously for the same axis, a priority scheme is used to solve the most hazardous problems first. Situations with a lower priority are not looked at until all problems with higher priority have been solved. In the current implementation of the safety-layer problems are solved in the following order:

- When the emergency-button is pressed the H-Drive is brought to standstill as fast as possible.
- A tilt-protection procedure prevents the axis from wedging, caused by a tilt between the axes
- A software based airbag is implemented to prevent the LiMMS from hitting the end strokes.
- The speed of the LiMMS is limited by a velocity-brake.
- The current that is send from the current amplifier to the H-Drive is limited to 4 Ampere during regular use. Some components of the safety-layer are allowed to shift the limit to a higher value temporary when necessary[1].

There exists one exception to the priority scheme: because calculating the output of the safety-controllers and sending the current actually to the motor is done at different stages in the code, clipping can occur simultaneously with other actions from the safety-layer. After calculating the outputs needed for the airbag or velocity-brake, the current send to the motor is clipped to a safe level.

[1]The 4 Ampère limit is referred to as `IMax_Low` and can be changed in `HD_V4_HDrive.c`. The 8 Ampère limit that is used to prevent destructive situations is referred to as `IMax_High`.

4.2 Airbag

The LiMMS is protected from bumping against the end-strokes by the use of a software safety-layer. The user is only allowed to control the LiMMS in the safe area (figure 4.1, L_{SAFE}). When the LiMMS enters the airbag protected region L_{AB}, a software based airbag takes over control. Because the only aim of the airbag is to brake the movement and push the LiMMS back into the safe area, a simple PD-controller can be used to perform this task.

Figure 4.1: Safety airbag

As a rule of thumb, when using a digital controller, the sampling frequency [Hz] of the controller should be bigger than 30 times the bandwidth of the closed loop system. Employing a sampling frequency of 5kHz requires a bandwidth smaller than 167 Hz. By tuning the PD-controller to form a critically damped closed loop, the fastest possible response is obtained (see figure 4.4). The bode and Nyquist diagrams in figure 4.2 and 4.3 show that the closed loop system is stable. Settings: $P = 100$, $D = 25$.

During the airbag operation the maximum allowed current level is temporary increased. Afterwards the H-Drive is turned off.

The minimally required thickness of the required airbag is calculated by using standard kinematic equations.

The maximum velocity of the LiMMS is limited to 1 m/s by the velocity brake (see next section). It is plausible to assume that the position error (distance to nearest point in safe area) and velocity cause the controller to send out a current near the maximum allowed current level $\widehat{I}_{max}$. Using equation 1.7 and ignoring disturbances yields the following equation ($\alpha_1 = \frac{3}{2}\frac{K_{ph}}{m}$):

$$\ddot{x}(t) = \alpha_1 \widehat{I}_{\max} \tag{4.1}$$

The time that is needed to come to a standstill:

$$t_{AB} = \frac{v_{\max}}{\alpha_1 \widehat{I}_{\max}} \tag{4.2}$$

Combining 4.1, 4.2 and $x(t_{AB}) = x_o + v_o t_{AB} + \frac{1}{2}\ddot{x}t_{AB}^2$ gives the minimal thickness of the airbag.

$$L_{AB,\min} = \frac{1}{2}\frac{v_{\max}^2}{\alpha_1 \widehat{I}_{\max}} \tag{4.3}$$

With $v_{max} = 1$, $\alpha 1x - a_{1,x-axis} = 8.08$, $\alpha_{1,Y-axes} = 3.60$ and $I_{\max} = 4$, a safety layer of 15 mm is sufficient for the X-axis and 35 mm satisfies for the Y-axes.

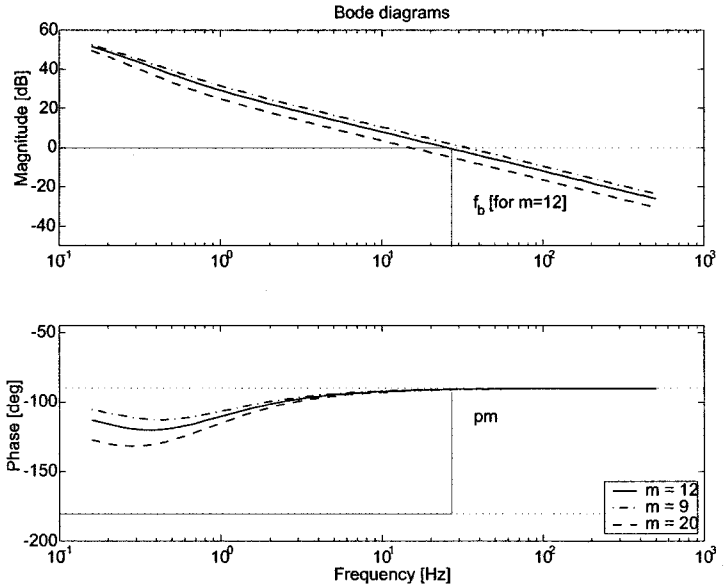

Figure 4.2: Airbag - Bode plot of open loop system (from I to x)

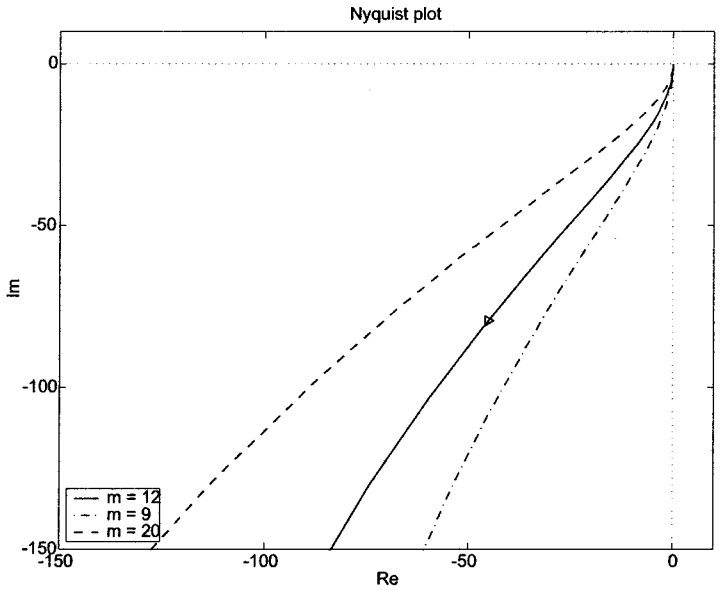

Figure 4.3: Airbag - Nyquist plot of open loop system

	X-axis	Y-axes
Penetration:		
Estimated	15 mm	35 mm
Measured	13 mm	31 mm
Settings:		
minpos	-0.60 m	-0.05 m
maxpos	+0.05 m	+1.05 m
margin	+0.03 m	+0.05 m
Result:		
Usable area:	-0.57...+0.02 m	0.00...1.00 m

Table 4.1: Settings of the airbag

The airbag of the X-axis has been tested experimentally by applying a constant current between -4A and +4A to the LiMMS and measuring the position after entering the airbag. The airbag of the Y-axes has been tested by applying a current to the Y1-axis and using a PID-Controller and feedforward to make the Y2-axis follow the Y1-axis as good as possible. Table 4.1 shows the maximum penetration and the final settings of the airbag[2].

Because the airbag-procedure is meant to prevent a destructive event, the maximum current limit is increased to `IMax_High`.

4.3 Velocity brake

The original velocity limitation consisted of a simple procedure that sends a current to the motor with an amplitude of minus two times the velocity at that moment.

In the new version of the H-Drive software a P-Controller is used to control the velocity back to a safe level when needed. The bandwidth of the open loop system has to stay beneath 167 rad/sec when running the controller at 5 kHz.

Setting[3] $P = 25$ does the job (see figures 4.2, 4.6 and 4.7).

When the velocity reaches a safe level, control is passed back to the user. A high velocity is not destructive. Therefore the current limit is kept at `IMax_Low`.

4.4 Tilt protection

The tilt protection comes into operation when the difference between the Y1 and Y2 position becomes too big[4].

The procedure brakes the X-axis and Y-axis by using the P-Controller of the velocity brake (section 4.3). A slave-controller uses the PID controller of the homing/moving procedure (section 3.3) to eliminate the difference between the coordinates of the Y-axes. The difference reduces according to a third-degree setpoint[5], by constructing the Y2-reference from the current Y1-state (position

[2]These settings can be changed in function `mdlInitializeConditions()` of `HD_V4_HDrive.c`

[3]These settings can be changed in function `mdlInitializeConditions()` of `HD_V4_HDrive.c`

[4]These settings can be changed by setting `maxangle` in function `mdlInitializeConditions()` of `HD_V4_HDrive.c`

[5]Initialized in `safety_angle_viol_init()` and calculated in `safety_angle_viol()` in `HD_V4_Safety.c`

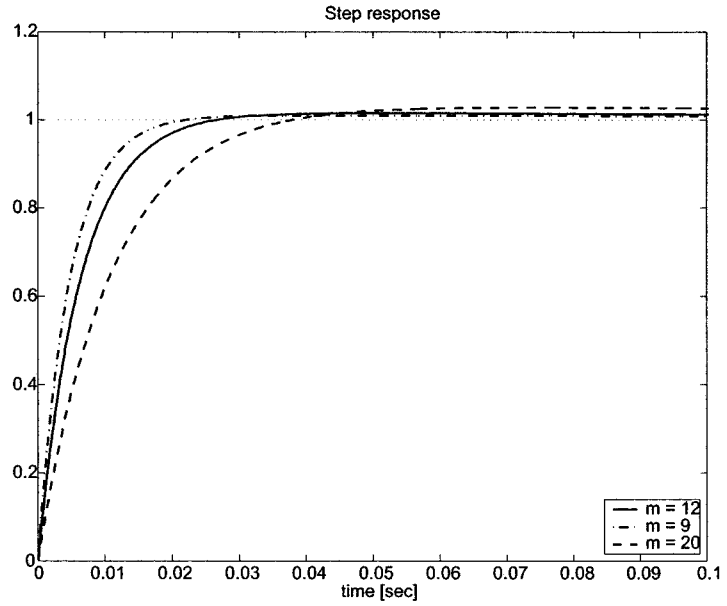

Figure 4.4: Airbag - Step response of system (digitally controlled at 5 kHz)

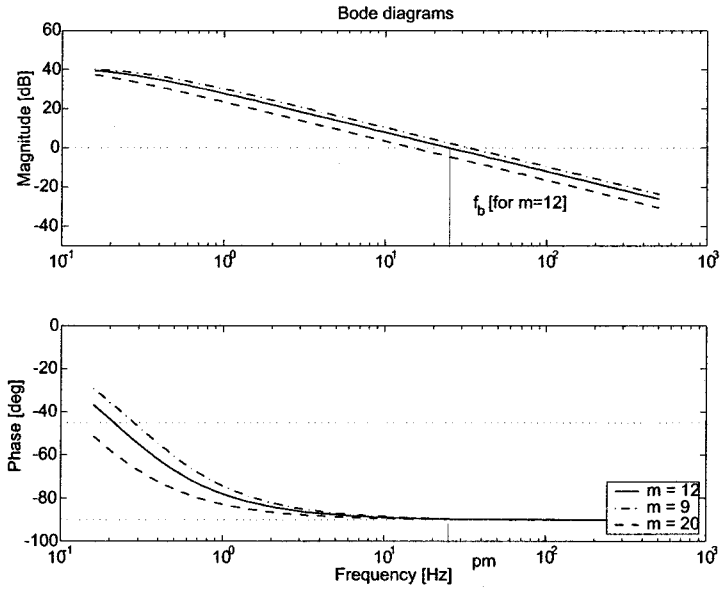

Figure 4.5: Velocity - Bode plot of open loop system (from I to x ')

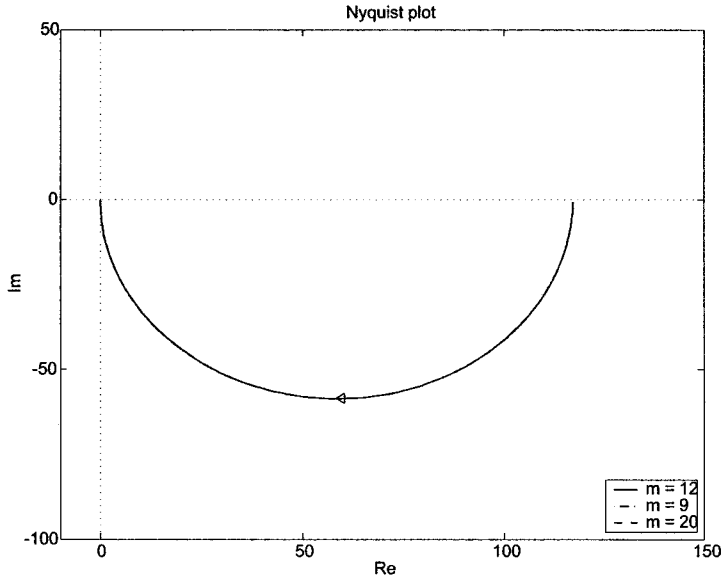

Figure 4.6: Velocity - Nyquist plot of open loop system

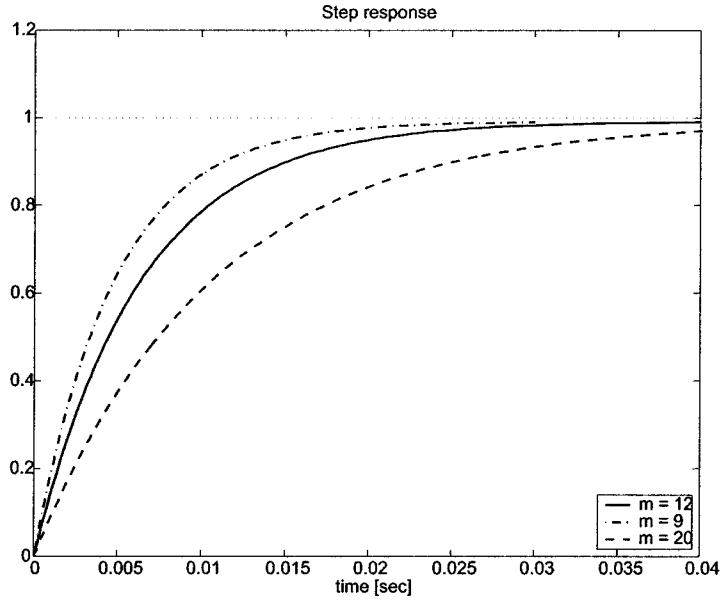

Figure 4.7: Velocity - Step response of system (digitally controlled at 5 kHz)

and velocity) superposed with the values from the setpoint.

To assure that the Y2-axis is able to follow the Y1-axis, the current of the Y1 axis is limited to 80% of the maximum current level. In this manner the PID-Controller of the Y2 axis has enough play to reduce the error.

Physically a difference of 30 mm is allowed, but a safety-margin is needed to absorb the effects of overshoot when tilt occurs when the Y-axes are moving with a significant difference in velocity. Therefore the tilt-protection is activated when the difference between the Y coordinates becomes bigger than 20 mm.

Control is not given back to the user when the tilt is gone.

4.5 Emergency stop

The emergency-button that is standard installed on the robot only disconnects the power when engaged[6]. Therefore this procedure is inapt to interfere in an emergency situation.

A second emergency button is connected to the dSPACE box. When this button is pressed, the H-Drive is brought to a standstill as fast as possible by the H-Drive control software.

The procedure of the emergency stop is very similar to the procedure of the tilt protection: brake X-axis and Y1-axis by making use of the controller of the velocity-brake, use a slave-controller to make the Y2-axis follow the Y1-axis and limit the current of the master-controllers to 80% of the maximum current-level to make sure the slave-controller can do its job. Because the emergency procedure is build to protect the system and user from very dangerous situations the maximum current level is set to `Imax_High`.

Afterwards the system is switched off so the translators can be moved with a minimal amount of resistance.

[6]When only the power is disconnected form the LiMMS, the axes keep moving untill the friction brings the system to a standstill or when the LiMMS bumps to one of the end-points. This can result in serious damage and dangerous situations.

Chapter 5

Implementation in Simulink

At the beginning of the project, only the X-axis was controlled by version 3.0 of the H-drive software written by S.G.H. Hendriks[1] (based on software from M.J.G. van de Molengraft en Antoine Verweij). Unfortunately this version of the code turned out to be inapt to expand to a three axes controller. Further more, the code was set up in a disorderly manner. Therefore, a new implementation of the code has been developed with a accessory Simulink interface. Version 4.0 of the H-Drive software is available in two different versions that can be used with MATLAB/Simulink:

- H-Drive Model: simulation model of the H-Drive, including initialization procedure, safety layer, sensors and 3D animated view of the system.
- H-Drive Hardware: Simulink block to control the real H-Drive system.

The Simulink blocks for Simulation and Hardware control are fully compatible and can be interchanged without adapting the surrounding model. Both blocks use the same C code for the initialization and protection of the H-Drive. In the next sections the blocks are discussed in more detail.

5.1 Introduction

The H-Drive Simulink block has the following input and output ports (also see appendix B.1):

- *In_I* (vector 3): vector with currents that have to be applied to the different axes as defined by the user, defined in the order: [current X, current Y1, current Y2]. The software in the H-Drive block protects the system from overcurrent.
- *In_Start* (scalar) : a value "1 " on this input-port switches the H-Drive on and starts the initialization procedure. Value "0 " turns the system off[2].

[1]Thesis Report No. 2000.37 - Iterative Learning Control on the H-drive, S.G.H. Hendriks, Eindhoven University of Technology, Department of Mechanical Engineering, 20 november 2000

[2]When using version 4.0 of the H-Drive software, it's sometimes safer to use the emergency button to stop the system. Section 4.5 explains why.

(Sub)code	State	Description
	During Initialization:	
8	Waiting for start	Waiting for signal to start (In_Start)
11	Test	Movetest of vibration procedure
12	Zero search	Executing zero-search procedure
13	Y-Align	Aligning Y-axes
14	Homing	Homing
15	Moving	Moving axes to starting position
7	Aligning failed	Zero-search procedure failed
	During Operation:	
..0..	Ready	LiMMS is operating ok
..2..	End of stroke	Eos-sensor activated (hit end-stop)
..3..	Position violation	LiMMS entered airbag region
..4..	Velocity violation	Moving too fast
..5..	Current violation	Overcurrent
..6..	Angle violation	Angle between Y- and X-axis too big
	Emergency:	
17	Emergency stop	Emergency button was hit

Table 5.1: State indications for the H-Drive

- *Out_Pos* (vector 3): Position of the H-Drive with respect to the global system of coordinates. The position of the LiMMS is not known until the homing-procedure of the initialization is finished. Therefore the output of this port can be set[3] to 0 during initialization.

- *Out_Time* (scalar): time since initialization. This time has to be used in controllers. The global Simulink-time returns the time that elapsed since turning the robot on.

- *Out_State* (scalar): status of the H-Drive (see table 5.1). During the initialization the status-variable shows the global status of the overall system. After the initialization the status variable shows the separate states of the three axes. Example: state 350 means: position violation (3) for X-axis, overcurrent (5) for Y1-axis, operating OK (0) for Y2-axis.
  Leading zeros are not displayed, so state 50 means: X-axis OK (leading zero, not showed), overcurrent (5) for Y1 axis and Y2 axis OK (0).

- *Out_CtrEn* (scalar): the output of this port is 1 when the user is given permission to control the H-Drive and 0 when the software controls the H-Drive (initialization / safety layer). Therefore this output can be used to enable a subsystem that contains a user defined controller.

- *Out_I2Drive* (vector 3): contrary to the input signal *In_I* this vector shows the currents that have actually been send to the LiMMS. Actions of the safety layer are therefore visible in this signal.

[3]Setting definine `SHOW_POS` in `HD_V4_HDrive.c` to zero, output of position and velocity are supressed during initialization.

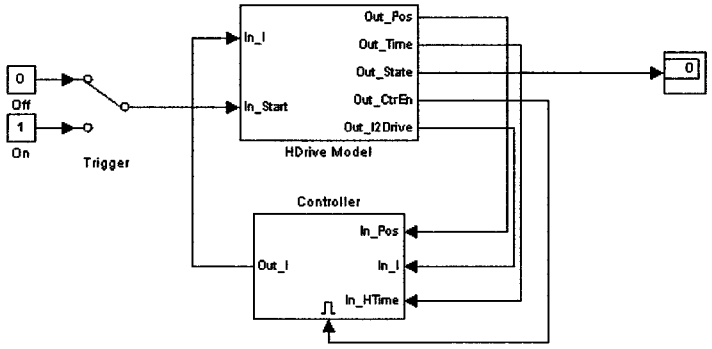

Figure 5.1: HD_V4_Model.mdl

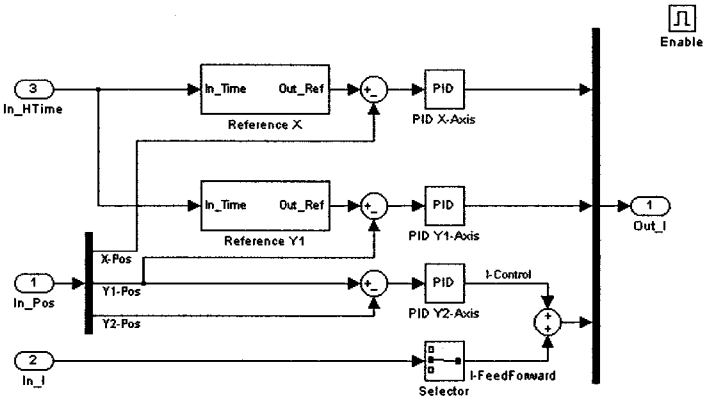

Figure 5.2: HD_V4_Model \ Controller (Example)

Figure 5.1 shows a typical Simulink model to control the H-Drive. The model consists of the H Drive Model[4] and a controller.

Figure 5.2 depicts a possible setup for a controller, using the I/O ports of the H-Drive block to control the plant.

Under the mask of the Simulink block a C based S-function takes care for initializing and protecting the H-Drive. This S-function is either connected to a software-model of the plant or the real-life hardware. Externally the masked subsystems for simulation and hardware control look exactly the same and are directly interchangeable.

5.2 Simulation

The simulation version of the software has the following features:

[4]The model to control the real-life system looks exactly the same, with the exception that the Hdrive block is called "HDrive Hard".

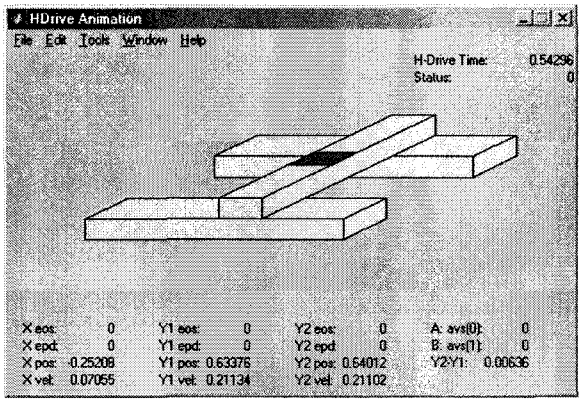

Figure 5.3: HD_V4_Model \ HDrive Model

- Modelled plant, based on equation 1.7 . The friction is modelled using a constant coulomb friction component and a viscous component that is linear with the velocity of the LiMMS. Because of the second order model, higher order effects like resonances are not visible in simulation. Cogging and reluctance forces are not modelled. (figure B.4)

- Simulated sensor-signals: EOS, EPD, AVS (see figure 1.2 and 1.3 for sensors and sensor locations).

- Animated H-Drive (figure 5.3).

Before the simulation can be started, the simulation parameters have to be defined from the MATLAB command window. This can be done by running the M-file `HD_Parameters`, which defines the plant parameters (table 1.2), sensor positions (figure 1.2) and sets some parameters for the animated view.

Figure 5.3 shows the animated view of the H-Drive, that automatically pops up when the simulation is started (if variable `AnimShow` is set to `1` in the MATLAB workspace). The window contains the following information:

- State of the sensors: end-of-stroke (EOS), end-position detector (EPD) and angle violation sensor (AVS). Value `0` means the sensor is not activated, value `1` means the sensor is activated.

- Position (pos) and velocity (vel) of the LiMMS.

- Difference between the Y2 and Y1 coordinate. This is a measure for the tilt of the Y-axes with respect to the X-axis.

- Status of the H-Drive.

- H-Drive time: time elapsed since the initialization.

5.3 Interfacing with the H-Drive

The Simulink block that is used to control the real-life plant, looks the same like the block that is used for the simulation. Internally the H-Drive model from the simulation is replaced by connections that lead to the dSPACE DS1130 board that takes care for the interfacing between the PC and the H-Drive. After designing a controller in MATLAB/Simulink and selecting an appropriate sampling frequency (default: 5 kHz), the model can be compiled using the Realtime Workshop compiler (RTW Build). Hereafter the system can be controlled from ControlDesk. Because the real-life plant is used, it is not necessary to run the `HD_Parameters` script.

Under some (dangerous) situations the H-Drive is automatically turned off by the safety-layer. Afterwards it is not always possible to restart the system by triggering the signal *In_Start*. Resetting the software in the PPC processor (from within ControlDesk) or switching the current amplifier on and off solves this problem.

5.4 Simulink S-Function

The code that is used to control the H-Drive is included in appendix B.

Chapter 6

Kalman-based zero-search

The zero-search procedure presented in chapter 2 only considers the direction of the displacement, caused by a pulsing current amplitude. By not looking at the amount of displacement, valuable information gets lost. This chapter presents the idea of using a Kalman filter to estimate the position offset.

6.1 Introduction

Before starting the design of the new zero-search procedure, the performance and accuracy of the Kalman filter is investigated.

Appendix A.1.1 and A.1.2 contain an elaborate description of the extended Kalman filter that will be used. In the first instance, a simple system model without friction is used (equation 1.6 and 1.7 in section 1.1:

$$\begin{aligned} F_{ph} &= \sum_{i=0}^{2} F_{ph,i} = \frac{3}{2}\hat{I}K_{ph,i}\cos(p-\varphi) \\ m\ddot{x}(t) &= F_{ph} - F_{disturbances} \end{aligned} \tag{6.1}$$

Ignoring disturbances like friction forces results in:

$$\ddot{x} = \alpha_1 u_1 \cos(\alpha_2 - u_2) \tag{6.2}$$

with the following groups of parameters to be estimated:

$$\begin{aligned} \alpha_1 &= \frac{3}{2}\frac{K_{ph}}{m} \\ \alpha_2 &= p \end{aligned} \tag{6.3}$$

Figure 6.1 shows the change of some important parameters of the Kalman filter when the settings of table 6.1 are used in a simulation environment. Before the output of the modelled plant is lead into the Kalman filter, some noise is added. Moreover, the output of the model is discretized to simulate the behavior of the encoder.

Discription	Value
Input	
Current amplitude $\widehat{I}$	$u_1 = sin(2\pi t/30e^{-3})$
Phase φ of current	$u_2 = \pi t/30e^{-3}$
Filter	
Model	$x" = \alpha_1 u_1 \cos(\alpha_2 - u_2)$
Extended state	$\overrightarrow{x} = \begin{bmatrix} x & \dot{x} & \alpha_1 & \alpha_2 \end{bmatrix}^T$
Initial estimate	$\overrightarrow{x_0} = \begin{bmatrix} 0 & 0 & (0.75 + 0.5\,\mathrm{rand}(1))\alpha_1 & 0 \end{bmatrix}^T$
Measurement noise	$R = (10^{-6})^2$
Modelling noise	$diag(Q) = \begin{bmatrix} (10^{-6})^2 & (10^{-3})^2 & (10^{-3})^2 & (10^{-3})^2 \end{bmatrix}^T$
Variance matrix	$diag(P_0) = \begin{bmatrix} (10^{-6})^2 & (10^{-4})^2 & (1)^2 & (10)^2 \end{bmatrix}^T$.

Table 6.1: Settings Kalman filter depicted in figure 6.1

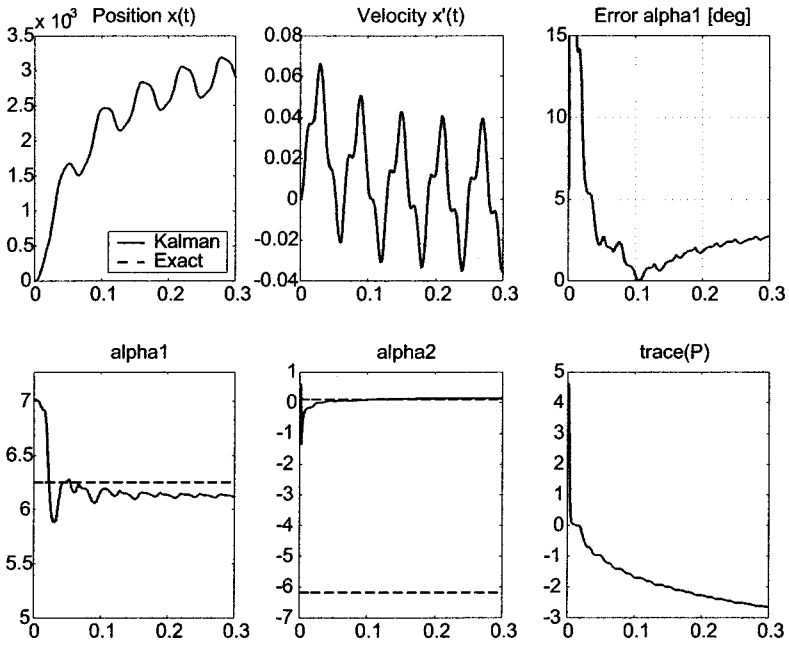

Figure 6.1: Signals from simulations with Kalman filter

The first simulations showed that it should be possible to achieve an accuracy of 5 degrees when determining the position offset of the LiMMS. This is slightly worse than the accuracy of the vibration method of chapter 2, which is 3 degrees. The variations in changing motor gain[1] caused by this error are 0.4%. By tuning and extending the Kalman filter, the accuracy of the zero search method can probably be increased.

Another requirement of the zero-search is the possibility to perform the algorithm at an apparently fixed position (in the uncontrolled example of figure 6.1 the drift form the starting point is several millimeters, which is unacceptable). This might be achieved by implementing a controller (section 6.3) that takes care the LiMMS follows a predefined setpoint with a very small amplitude.

Persistent excitating signals on the inputs u_1 and u_2 are required to assure correct operation of the Kalman filter (section 6.2). From this point of view a more complex setpoint like $x_{ref}(t) = amplitude \cdot \sin^3(t)$ would be a better choice. However, due to friction and cogging its undesirable to have periods with zero velocity. As a first test the setpoint[2] consists of a simple sine with an amplitude `REF_AMPLITUDE` of 50 μm, period `REF_DELTA` of 0.1 sec and phase offset `REF_PHI_OFFSET` of 0 rad.

During the first experiments on the real H-Drive, it became obvious that friction and cogging play an important part. Therefore the effect of friction is included in the system model, that will be used in the Kalman filter (equation 6.4 , 6.5 and 6.1). Appendix A.1.1 and A.1.2 contain an elaborate description of the extended Kalman filter that will be used in the next sections.

$$\ddot{x} = \alpha_1 u_1 \cos(\alpha_2 - u_2) - \alpha_3 sign(\dot{x}) \tag{6.4}$$

$$\begin{array}{ll} \alpha_1 = \frac{3}{2}\frac{K_{ph}}{m} & u_1 = \widehat{I} \\ \alpha_2 = p & u_2 = \varphi \\ \alpha_3 = \frac{c}{m} & \end{array} \tag{6.5}$$

6.2 Input signals

Looking at equation 6.4 it can be seen that the system has two inputs to control only one output. This means the control is over-determined. The following strategy is used to make sure both signals u_1 and u_2 are persistently excitating (which is required for the Kalman filter to operate correctly):

- u_1: Amplitude $\widehat{I}$ of the current is used as control-input to make the system follow the desired setpoint. Section 6.3 goes further into the matter of the controller and its difficulties.
- u_2: Phase φ of the current follows a predefined profile (see later on). The profile is initially chosen such that the parameters of the Kalman filter

[1] $gain\_error = \frac{K_{ph,real} - K_{ph,attained}}{K_{ph,real}} = \frac{(\cos(zero) - \cos(zero \pm error))}{\cos(zero)} * 100\% = (1 - \cos(error)) * 100\% = 0.4\%$

[2] The setpoint can be modified by changing the mentioned parameters `REF_AMPLITUDE`, `REF_DELTA` and `REF_PHI_OFFSET` or adapting function `Hdrive_zerosearch_getsetpoints()` in `HD_V5_Phi_Est.c`

can be estimated quickly. In course of time the profile is set such that the setpoint can be followed more accurately.

As long as position offset p (and coherent with p parameter α_2) is not known exactly, the gain of the cosine-term in equation 6.4 can vary between -1 and $+1$, which makes it almost impossible to control the system. The uncertainty with respect to the cosine makes it very difficult to choose a correct gain for the controller. Moreover, the uncertainty of the sign of the cosine can cause the controller to become unstable. To minimize the effect of these problems the desired form of signal u_2 is set up as follows (figure 6.2):

- Stage 1: Signal u_2 is chosen such that gain $\cos(\alpha_2 - u_2)$ in the system equation varies between -1 and $+1$. This enormous fluctuation guarantees that the gain is at least once in a period big enough to cause a movement of the LiMMS. During the first stage a rough estimate of the parameters is obtained quickly, so the sign and gain of the cosine-term in the system equation are roughly known and controlling the LiMMS is more easily.
  Reference signal during this stage: $\varphi_{ref} = \frac{2\pi t}{REF\_DELTA}$
  Phase send to current amplifier: $u_2 = -\varphi_{ref}$
  Resulting cosine term in system: $cos(\alpha_{2,\text{real}} - u_2) = \cos(p_{\text{real}} - \varphi_{ref})$
  The desired signal is shifted by offset p which is physically present in the system and can not be compensated as long as no good estimate of α_2 is available.

- Stage 2: Transition between stage 1 and 3. See later on.

- Stage 3: During the third stage $\cos(\alpha_2 - u_2)$ fluctuates between $+0.5$ and $+1.0$, so the gain has a rather constant value and sign, which makes it possible to make the LiMMS follow the setpoint, while there is still sufficient excitation for improving the estimates of the unknown parameters α_i.
  Because the cosine-term has to stay between $+0.5$ and $+1.0$ by controlling u_2, the random offset p that was present in stage 1 has to be compensated for. Without the compensation the cosine will fluctuate between two random boundaries. Moreover, the phase of the reference signal itself can be adjusted to get a better transition between the different stages.
  Reference signal: $\varphi_{ref} = 1.0472 \sin(\frac{\pi t}{REF\_DELTA} + p_{offset})$
  Phase send to current amplifier[3]: $u_2 = -\varphi_{ref} + p_{estimate}$
  Resulting cosine term in system: $cos(\alpha_{2,\text{real}} - u_2) = \cos(p_{\text{real}} - p_{estimate} + \varphi_{ref}) = \cos(\varphi_{ref} + error_p)$
  To get the cosine terms of stages 1 and 3 in phase, p_{offset} is set to $p_{offset} = \frac{p_{estimate}}{2}$. It should be noticed the random phase offset that was present in the cosine-term during the first stage has been eliminated largely in the third stage.

- Stage 2: The second stage guarantees a smooth transition between stage 1 and 3. The amplitude range of the cosine is gradually shifted from $[-1.0, +1.0]$ to $[+0.5, +1.0]$ and the offset correction gets activated. A

[3]While φ_{ref} is the phase-angle that is desired to be experienced by the system, the phase angle that is actually present is $\varphi_{observed} = p - u_2$.

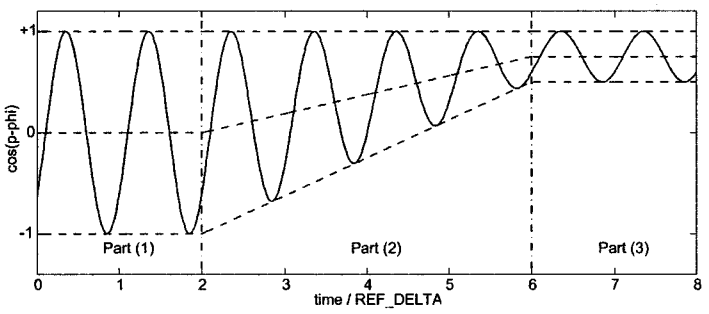

Figure 6.2: $\cos(p-\varphi) = \cos(\alpha_{2,\text{real}} - u_2)$ as function of time.

transition function $g(t)$, that varies between 0 and 1 linear in time, is used to gradually switch between stage 1 and 3.
Transition function: $g(t) = \frac{(t-t_{start,stage2})}{(t_{end,stage2}-t_{start,stage2})}$
Amplitude of reference signal: $amp(t) = \pi(1-g(t)) + 1.0472g(t)$
Reference signal: $\varphi_{ref} = amp(t)\sin(\frac{\pi t}{REF\_DELTA} + p_{offset}g(t))$
Phase send to current amplifier: $u_2 = -\varphi_{ref} + p_{estimate}g(t)$
Resulting cosine term in system: $cos(\alpha_{2,\text{real}} - u_2) = \cos(p_{\text{real}} - p_{estimate}g(t) + \varphi_{ref}) = \cos(\varphi_{ref} + (1-g(t))p_{estimate} + error_p)$

Figure 6.2 shows the ideal shape of the resulting cosine term of equation 6.4 when using the defined setpoint. In practice the transition during the second stage is not always as gradual as depicted in figure 6.2 because of the rather poor[4] offset compensation that is used. In the experimental version of the Kalman-based zero-search method the first two stages comprises a fixed number of periods[5]. The third stage is finished after a pre-defined time.

6.3 Controller

As mentioned in the previous section, the gain of the system defined as $\alpha_1 \cos(\alpha_2 - u_2)$ in equation 6.4 , is not know accurately until the zero-search procedure has been finished. Initially the sign as well as the order of the order of the gain are unknown. Designing a stable controller is therefore a rather complicated task.

To suppress these problems a PD-Controller is used, which output is suppressed at moments when the value or sign of $\cos(\alpha_2 - u_2)$ is not know well.

- After initializing the filter, parameters α_1 and α_2 are unknown. Therefore the gain of the PD-Controller is suppressed by applying an extra time-dependent gain[6] with amplitude $\frac{2}{\pi}\arctan(REF\_ATC\_TIME \cdot t)$ that

[4]Because of the lack of offset-compensation at the start of the second stage, it's not possible to guarantee that the mean of the cosine has the same course as depicted in figure 6.2. The direct, unfiltered, offset-compensation with $p_{estimate}$ later on can result in a capricious graph.

[5]The numbers of periods (of `REF_DELTA` seconds) during the first stage can be modified by adapting define `REF_PHIEXC_INIT` in `HD_V5_Phi_Est.c` while `REF_PHIEXC_STOP` defines the number of periods during the second stage. Stage 3 finishes after `ZERO_SEARCHTIME` seconds.

[6]The supression of the PD controller can be modified by adapting define `REF_ATC_TIME` or adapting function `Hdrive_zerosearch_controlout()` in `HD_V5_Phi_Est.c`

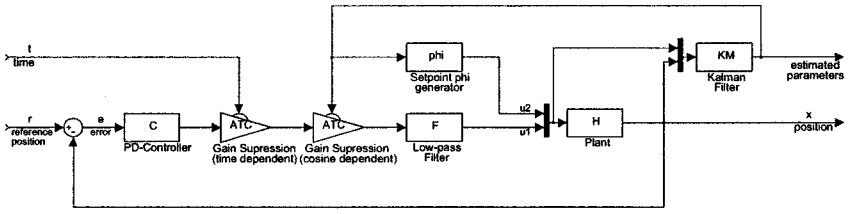

Figure 6.3: System setup during the Kalman-based zero-search procedure.

limits the output of the controller as long as a rough estimate of α_1 and α_2 is not available.

- When $\cos(\alpha_2 - u_2) \approx 0$ a paradoxical situation comes into existence. Because of the low system gain, the PD controller needs to have a high gain to be able to control the system.
  However, because of errors in the estimation of α_2 the cosine might have a bigger gain as expected. The cosine might even have a different sign as expected when $\cos(\alpha_2 - u_2)$ is near its zero. In view of stability a high controller gain is therefor not desirable.
  Because stability is more important that being able to follow the setpoint with a minimal error, the gain of the controller is lowered near $\cos(\alpha_2 - u_2) = 0$ by applying an extra gain[7] of with amplitude $\frac{2}{\pi}\arctan(REF\_ATC\_COS \cdot \cos(\alpha_2 - u_2)_{estimation})$. This extra gain also takes care for the sign-correction of the controller output when the system-gain changes sign.

Applying the extra gains mentioned above has exactly the same effect as adapting the P and D gain of the controller simultaneously with the same factor. From this point of view dynamic scheduling the P and D gain might be a more elegant solution. By changing the P and D gain independently it is possible to have more control over the systems response.

The output of the controller is filtered by a first order low-pass filter[8] that suppresses frequency contents above 50 Hz to prevent the eigen frequencies of the systems from being excited.

Figure 6.3 shows a schematic view of the controlled system.

Simulations with several controller settings (different bandwidths and gain/phase-margins) showed that the system becomes often unstable and the setpoint can hardly be followed. One of the sources of error is the impact of friction when making small movements (see section 6.4). To reduce the influence of friction, a friction-compensating feedforward was added. The feedforward had no positive effect at all. Friction plays especially a dominant role when the LiMMS has to be brought in movement from standstill during the first moments of the zero-search procedure. However, the system parameters that are needed for the friction compensation are initially unknown, so compensation is virtually

[7]The supression of the PD controller can be modified by adapting define `REF_ATC_COS` or adapting function `Hdrive_zerosearch_controlout()` in `HD_V5_Phi_Est.c`

[8]The first order filter, defined in the discretized z-domain, can be modified by adapting defines `ZERO_FILTER_GAIN` and `ZERO_FILTER_POLE` or adapting function `Hdrive_zerosearch_controlout()` in `HD_V5_Phi_Est.c`

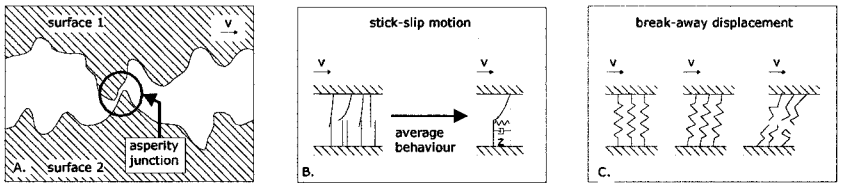

Figure 6.4: Physical interpretation of the LuGre friction model

Variable	Discription
	Equations
$F_{friction} = \sigma_0 z + \sigma_1 \dot{z} + \sigma_2 \dot{x}$	Tangential friction force
$\dot{z} = \dot{x} - \frac{\lvert\dot{x}\rvert}{g(\dot{x})} z$	Average deflection of the bristles
$\sigma_0 g(\dot{x}) = F_c + (F_s - F_c)\, e^{-\left(\frac{\dot{x}}{v_s}\right)}$	Stribeck curve
	Variables
x	Relative velocity between the two surfaces
z	Average bristle deflection
$g(\dot{x})$	Stribeck curve for steady-state velocities
v_s	Stribeck velocity
F_s	Static friction (Stribeck curve)
F_c	Coulomb friction (Stribeck curve)
σ_0	Bristle stiffness
σ_1	Bristle damping
σ_2	Viscous damping coefficient

Table 6.2: Equations and variables of the LuGre friction model

impossible when it is most needed. With a little error in the estimation of parameter α_2 system gain $\alpha_1 \cos(\alpha_2 - u_2)$ might have a different sign as expected, which results in a friction compensation that works in the wrong direction and makes the system unstable. The same problems arise when implementing a mass feedforward.

6.4 Improvements

The reliability of the zero-search procedure presented in this chapter is low. It takes a good deal of improvements before the Kalman-based procedure can surpass the vibration method with respect to stability, speed and noise-level. The following items should be considered:

- When following the prescribed setpoint for the position, with its small displacements and low velocity, friction plays an important role. During the motion the system has several periods in which the system is in stick-slip[9] mode and equation of motion (6.4) is no longer valid. A more elaborate model is needed to describe the physical situation. Figure 6.4 shows what

[9]While being in stick-slip mode, the sliding surface sticks to the non-sliding surfce untill the force exceeds the break-away force. After moving a little the surface gets stuck again, waiting till it breaks away again. This results in a jerky motion: stick-slip.

is physically going on.
At microscopical level, the tops of the profile of the sliding surfaces form asperities that can deform when subjected to a driving force (figure 6.4.a). Contacting asperities act as small stiff springs with dampers, giving rise to microscopic displacements (stick) and return forces. If the displacement becomes too large, the junctions break. During the deformations the asperities act as bristles that can be modeled as springs and dampers with an average deflection z (figure 6.4.b). At high velocities the asperities break and equation 6.4, that describes the motion of a driven mass subjected to friction, is valid again (figure 6.4.c).
The LuGre friction model[10] from table 6.2 can be used to describe these phenomena.

- Adding friction- and mass compensation in the feed-forward makes it more easy to follow the desired setpoint. However, this is very difficult because the most important parameter, position offset p, is initially completely unknown as long as relative position encoders are used to determine the position, instead of absolute encoders so it is impossible to use results of previous zero-search procedures.

- The current controller can be improved by independently scaling P and D instead of directly changing the gain of the controller by using gain-reduction blocks as shown in figure 6.3. By changing P and D independently it is possible to tune the behavior of the controller for the various stages of the zero-stage procedure.

- Initially it is very difficult to make the LiMMS follow the desired trajectory. When the LiMMS moves into the wrong direction during the very first period of the setpoint, the sign of the system gain is apparently different as expected. Therefore the estimate of the offset could be shifted 180 degrees. Moreover the sign of the setpoint[11] could be shifted to make it more easily for the controller to move the error that arose because of the bad estimate of the system gain.

- The phase u_2 send to the LiMMS is currently determined by correcting the defined setpoint with the estimated offset calculated by the Kalman filter as described in section 6.2 (see also figure 6.5.a). Although the estimated offset p is provided by the Kalman filter, the resulting signal u_2 is initially rather capricious. Therefore adding a extra controller to make the observed phase angle[12] follow the setpoint for φ_{ref} might be worth the consideration (figure 6.5.b) . This controller might also improve the stability during the zero-search procedure.

- Instead of making the LiMMS follow a desired trajectory, a completely different strategy can be used in which an arbitrary input u_1 supplied.

[10] Journal publication: R.H.A. Hensen, M.J.G. van de Molengraft, M. Steinbuch, Frequency domain identification of dynamic friction model parameters, in Proc. 3rd IEEE Int. Conf. on Control Theory and Applications; Editors: Jianliang Wang, Pretoria, South Africa, 167-171, (2001)

[11] The direction of the setpoint can be flipped by changing the sign of the global variable `refdir(i)`. Although this feature is present in version 5 of the software, it is not used yet.

[12] See equation 1.6, 6.4: $\varphi_{obeserved} = p - u_2$

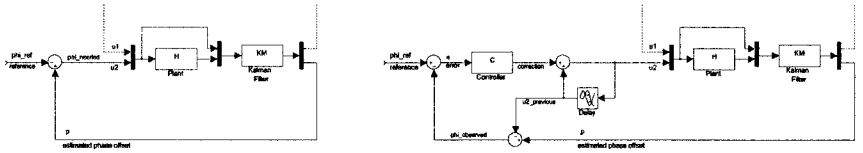

Figure 6.5: Proposal: (a) current setup (b) setup with extra u_2 controller

When the LiMMS drifts too far away from its initial starting-point action is taken to change the direction of moving (e.g. by changing sign u_1). This setup does not need a controller to limit the amount of movement of the LiMMS.

- The trace of variance matrix P of the Kalman filter is a measure for the amount of uncertainty in the parameter estimates. Therefore $trace(P)$ should be used as variable to change the controller gains instead of time t.
  $Trace(P)$ should also be included in the condition to finish the zero-search stage and continue with homing.

When improving the Kalman-based zero-search procedure one should use Version 4.0 of the H-Drive software as starting point (Appendix B).
Version 5.ξ only contains experimental code to make the Kalman-based method operational. The safety and alignment procedures are not as sophisticated as in Version 4.0

Chapter 7

Conclusion and recommendations

Getting the H-Drive operational turned out to be quite a job. The single axis code that was already available at the beginning of the project[1] turned out be unsuitable to control all axis of the robot: the code was difficult to expand, set up in a disorderly manner and besides contained several bugs.

Version 4.0 of the software solves this problem. Using the defines from section B.2 the most important characteristics of the initialization and the safety-layer can be changed in an easy manner, while the code-description from section B.3 can be used as a guide to apply more profounding changes.

The various controllers that are used during the initialization of the system and in the safety-layer are based on old estimates of the system parameters. The performance of the safety-layer can be increased by tuning the controllers, using the data resulting from the system identification that is described in section 1.3.

Further, it might be wise to add a neat stop-procedure to stop the H-Drive. When the current is disconnected from the LiMMS, the axes keep moving until the friction becomes too big or the LiMMS bumps to the end-point. At the moment the H-Drive can therefore only be turned of in a safe way by pressing the emergency switch (see footnote section 5.1) or generating a setpoint-profile to bring the H-Drive to a standstill.

The experiments with a Kalman-based zero-search procedure yielded no useful results. One could ask oneself if implementing another zero-search procedure is desirable at all, for the vibrational methods works accurate and proved to be quite stable. Moreover, the time it takes to find the zero is small with regard to the total time needed to initialize the system.

However, when replacing the present zero-search procedure with a Kalman-based one, the recommendations from section 6.4 should be taken into account.

Finally, an attempt was made to bring a lot of useful information on the H-Drive together in this report. Hopefully the report will therefore a good starting

[1]Thesis Report No. 2000.37 - Iterative Learning Control on the H-drive, S.G.H. Hendriks, Eindhoven University of Technology, Department of Mechanical Engineering, 20 november 2000

point for future research...

Appendix A

Digital Kalman filter

This appendix summarizes the equations of the digital extended Kalman filter that is used in version V5.ξ of the H-Drive software that is described in chapter 6 and appendix D.

A.1 Equations of the Kalman filter

A.1.1 General digital extended Kalman filter

The calculation scheme for a digital extended Kalman filter is as follows:

1. Time update: estimate $\vec{x}_{k+1}$ based on the previous state $\vec{x}_k$:
   $\vec{x}_{k+1|k} = \vec{f}(\vec{x}_k, \vec{u}_k) + Q$

2. Calculate Jacobian matrix F_{k+1} of $\vec{f}(x_{k+1|k})$ and H_{k+1} of $\vec{h}(x_{k+1|k})$:
   $F_{ij} = \partial f_i / \partial x_j$
   $H_{ij} = \partial h_i / \partial x_j$

3. Calculate variance matrix P and gain matrix K:
   $P_{k+1|k} = F_{k+1} P_k F_{k+1}^T + Q$
   $K_{k+1} = P_{k+1|k} H_{k+1}^T [H_{k+1} P_{k+1|k} H_{k+1}^T + R]^{-1}$

4. Measurement update:
   $\vec{z}_{k+1} = result\ of\ measurement$
   $\vec{x}_{k+1} = \vec{x}_{k+1|k} + K_{k+1}[\vec{z}_{k+1} - \vec{h}_{k+1}(\vec{z}_{k+1|k})]$

5. Update variance matrix P:
   $P_{k+1} = [I - K_{k+1} H_{k+1}] P_{k+1|k}$

The Kalman filter uses system model $\vec{f}()$ to estimate the state-vector $\vec{x}_{k+1|k}$ of the next period. This estimate is then improved by comparing measurement results $\vec{z}_k$ with results that are predicted by model $\vec{h}()$. Noise matrices Q and R indicate the magnitude of model uncertainties and measurement noise. The value of error $(\vec{z}_k - \vec{h}_k)$ and the magnitude of uncertainties (Q, R, and P) determine how much time-update $\vec{x}_{k+1|k}$ is changed by measurement results.

Parameter	Description
$\vec{x}_k$	State after step k (final estimate)
$\vec{x}_{k+1\|k}$	Intermediate state-estimate for step $k+1$ based on information from step k
$\vec{f}(x_k)$	Function that describes the relation between successive states and inputs
F_k	Jacobian matrix of $\vec{f}(x)$ at step k
$\vec{z}_k$	Measurement vector at step k
$\vec{h}(x_k)$	Function that describes the relation between measurement vector $\vec{z}_k$ and state $\vec{x}_k$
H_k	Jacobian matrix of $\vec{h}(x)$ at step k
P_k	Covariance matrix at step k
$P_{k+1\|k}$	Intermediate covariance matric estimate for step $k+1$ based on information from step k
K_k	Gain matrix at step k
I	Identity matrix
Q	Process noise matrix
R	Measurement noise matrix

Table A.1: Variables used in the Kalman filter

Before using the Kalman filter, it is necessary to choose initial values for state estimate $\vec{x}_0$ and variance matrix P_0. Considering the statistic (normal) distribution of the possible values of the state variables, $\vec{x}_0$ contains the mean values and P_0 is a diagonal matrix with variances on the diagonal.

A.1.2 Kalman filter for H-Drive

Using the equations of the previous subsection, a digital Kalman filter for the H-Drive is designed.

In section 1.1 the following equations of motion were derived (equation 1.6 and 1.7):

$F_{ph} = \sum_{i=0}^{2} F_{ph,i} = \frac{3}{2}\widehat{I}K_{ph,i}\cos(p-\varphi)$

$m\ddot{x}(t) = F_{ph} - F_{disturbances}$

Taking coulomb friction into account as the only disturbance force results in:

$\ddot{x} = \alpha_1 u_1 \cos(\alpha_2 - u_2) - \alpha_3 sign(\dot{x})$

with parameters:

$\alpha_1 = \frac{3}{2}\frac{K_{ph}}{m}$

$\alpha_2 = p$

$\alpha_3 = \frac{c}{m}$

and inputs:

$u_1 = \widehat{I}$

$u_2 = \varphi$

The sign of the coulomb friction can be determined by comparing successive encoder measurements. Therefore $sign(\dot{x})$ is not calculated based on state $\vec{x}_{k+1}$, but is an externally determined parameter:

$sign(\dot{x})$

The state vector consists of position and velocity, extended with the unknown (constant) parameters that are to be determined:

$\vec{x} = \begin{bmatrix} x & \dot{x} & \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix}^T$

Only the position is measured:

$\vec{z}_{k+1} = \textit{measured position at time-step } _{K+1} = \vec{h}(\vec{x}_k) + R$

Using these equations, the vectors and matrices for the Kalman filter are:

$$\vec{x} = \begin{bmatrix} x & \dot{x} & \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix}^T$$

$$\vec{x}_{k+1} = \vec{f}(\vec{x}_k, \vec{u}_k) + Q$$

$$= \begin{bmatrix} x + \dot{x}dt \\ \dot{x} + \ddot{x}dt \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$$

$$= \begin{bmatrix} x_k(1) + x_k(2)dt \\ x_k(2) + [x_k(3)u(1)\cos(x_k(4) - u(2)) - x_k(5)sign(\dot{x})]dt \\ x_k(3) \\ x_k(4) \\ x_k(5) \end{bmatrix}$$

$$F_{k+1} = \begin{bmatrix} \partial f_1/\partial x_1 & \cdots & \partial f_1/\partial x_5 \\ \vdots & \ddots & \vdots \\ \partial f_5/\partial x_1 & \cdots & \partial f_5/\partial x_5 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & dt & 0 & 0 & 0 \\ 0 & 1 & u(1)\cos(x_k(4) - u(2))dt & -x_k(3)u(1)\sin(x_k(4) - u(2))dt & -sign(\dot{x})dt \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\vec{z}_{k+1} = \vec{h}(\vec{x}_k, \vec{u}_k) + R$$

$$= [x]$$

$$H_{k+1} = \begin{bmatrix} \partial h_1/\partial x_1 & \cdots & \partial h_1/\partial x_5 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The resolution of the encoder is $1\mu m$, therefore the variance of the measurement error is 10^{-12}. Setting R to 10^{-13} yields the best results.

Vector $\vec{x}_o$ contains an initial estimate of the parameters:
$\vec{x}_o = \begin{bmatrix} x(t_0) & 0 & 6.25 & \pi/2 & 0.5 \end{bmatrix}^T$ with uncertainty (variance)
$diag(P) = \begin{bmatrix} 10^{-12} & 10^{-8} & 1 & 1 & 0.1 \end{bmatrix}^T$.

Matrix Q is a diagonal matrix with model uncertainties on its diagonal. The equation to estimate the position of the next time-step is almost perfect, therefore is Q_{11} is very small. On the other hand, the modelling of the friction is rather poor, therefore Q_{55} is big. Studying all parameters results in $diag(Q) = \begin{bmatrix} 10^{-14} & 10^{-8} & 10^{-6} & 10^{-8} & 10^{-4} \end{bmatrix}^T$. The interdependency of the elements of Q determines which parameters are fitted the best[1].

[1] A large value of Q_{ii} allows a big deviation between the measured state and the estimated

A.1.3 Implementation of the digital Kalman filter

To minimize the processor load during operation, the Kalman filter is rewritten to minimize the number of operations needed to update the state. This is done by:

- Converting matrix equations to a set of simple algebraic equations, so time-consuming that are needed to do matrix multiplications can be avoided.
- Taking into account the symmetry of the variance matrix P: only the upper triangular part has to be calculated.
- Isolating terms that occur more than once in the equation: assign a variable to these terms that has to be calculated only once.

With these measures the Kalman filter can even operate real-time at high sample frequencies. The resulting C code is included in file `HD_V5_Phi_Est.c` of Version 5.ξ of the H-Drive software set. More information can be found in appendix D.

A.2 Performance

Section 6.1 shows an application of the digital Kalman filter and the attained performance.

state variable i. The state variable changes slowly because of deviations and is therefore not sensitive to temporary disturbaces (e.g. caused by modelling errors)

Appendix B

C Code for the excitation method (Software V4.0)

This appendix contains an introduction to Version 4.0 of the H-Drive software: operation of the Simulink blocks including a description of the I/O ports and error messages, parameters in the C software that are used to change important settings like controller-parameters and safety margins, an overview of the code and the source-code itself.

B.1 Simulink

Figure B.1 shows the most important blocks that are part of version 4 of the H-Drive software set:

- Simulation Interface (A): block for simulating the H-Drive. The block contains a model of the H-Drive (C), the simulator (D) and the control block (E). More information: section 5.1 and 5.2 and Appendix table B.1. Figure 5.1 shows a what is under the mask of the simulation interlace.
- Hardware Interface (B): block for interfacing with the real-life H-Drive. The block contains a model of the H-Drive (C), the simulator (D) and the control block (E). More information: section 5.1 and B.2 and Appendix table B.1. Figure B.2 shows a view under the mask of the hardware interface.
- Simulation Model (C): 3-Axis of the H-Drive, including sensors (section 1.2) and emergency switch. Both the real position of the LiMMS and the position output of the decoder (with random origin) are available at the output. More information: Appendix table B.2. Figure B.4 shows the Simulink model.
- H-Drive Animator (D): block for generating an animated view of the H-Dirve. More information: section 5.1, 5.2 and Appendix table B.3.
- Control block with software (E): block with initialization procedures, safety-layer and other software. This block forms the core of the simulation and hardware interface. More information: Appendix table B.4.

The following pages contain views of the several components, views under the masks and tables with descriptions of the I/O ports.

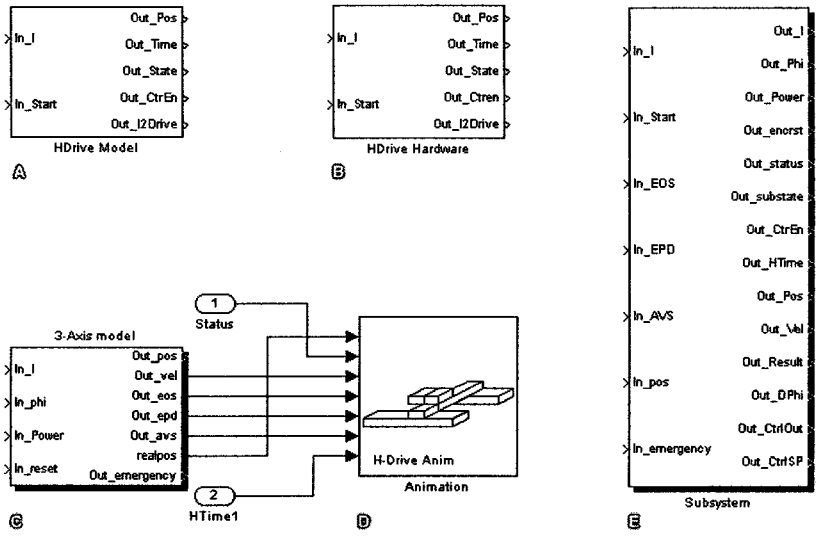

Figure B.1: Simulink blocks belonging to version 4 of the H-Drive software set: (A) Simulation Interface, (B) Hardware Interface, (C) Simulation Model, (D) H-Drive Animator, (E) Control block with software

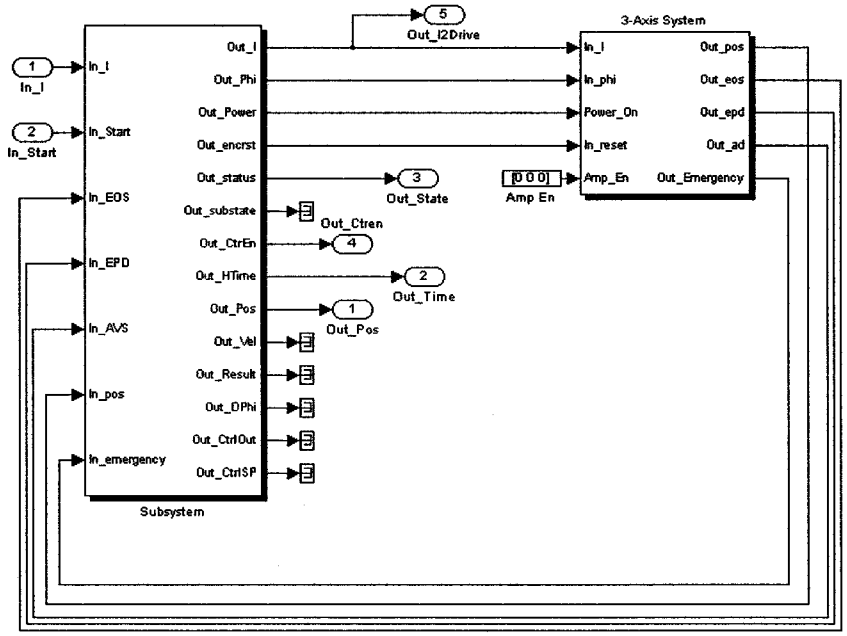

Figure B.2: Inside hardware interface (B): HD_V4_Hardware \ HDrive Hard

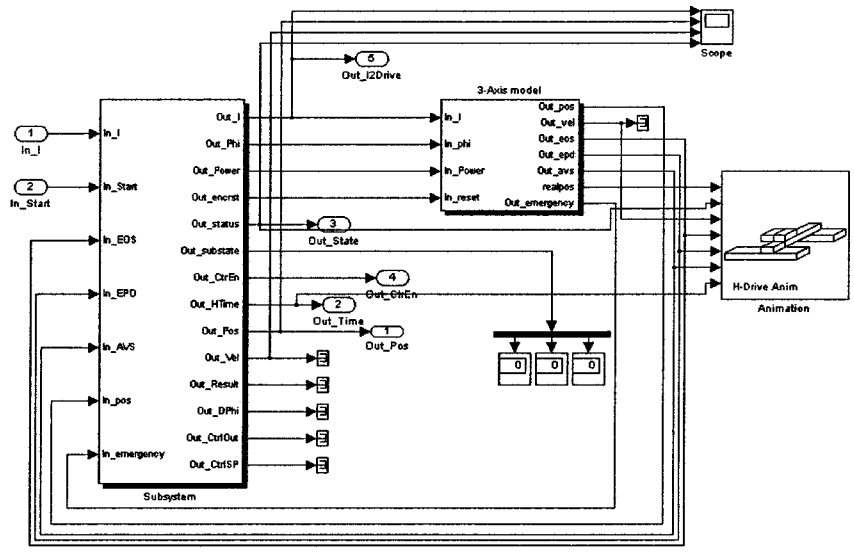

Figure B.3: Inside simulation interface (A): HD_V4_Model \ HDrive Model

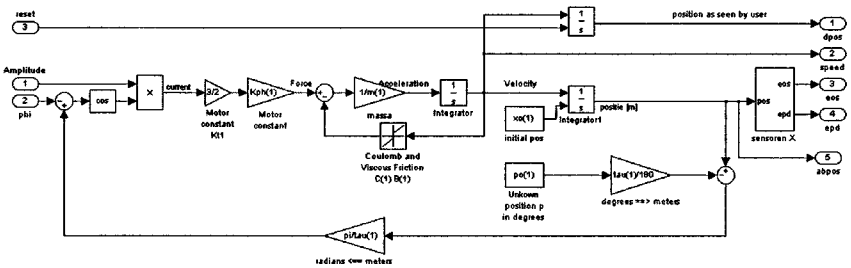

Figure B.4: Inside Model: HD_V4_Model \ HDrive Model \ 3-Axis Model \ X

Port	Width	Description
Input:		
In_I	3	Current to be send to the 3 axes (before passing safety layer)
In_Start	1	Switch H-Drive On (1) / Off (0)
Output		
Out_Pos	3	Position of the LiMMS for 3 axes
Out_time	1	Time since initialization procedure finished
Out_State	1	State of the H-Drive (see table B.5)
Out_CtrEn	1	Enable (1) or disable (0) external controller defined by user
Out_I2Drive	3	Current actually send to the 3 axes

Table B.1: Simulation Interface (A) and Hardware Interface (B)

Port	Width	Description
Input:		
In_I	3	Amplitude of current to be send to the 3 axes
In_Phi	3	Phase of current to be send to the 3 axes
In_Power		Switch H-Drive On (1) / Off (0)
In_Reset		Reset position encoder
Output		
Out_Pos	3	Position of the LiMMS for 3 axes (encoder output)
Out_Vel	3	Position of the LiMMS for 3 axes (exact, no estimate)
Out_eos	3	Output of EOS-sensor for 3 axes
Out_epd	3	Output of EPD-sensor for 3 axes
Out_avs	3	Output of AVS-sensors
realpos		Position of the LiMMS for 3 axes (with respect to orgin)
Out_emergency	3	Output of Emergency button

Table B.2: 3-Axis Model (C)

Port	Width	Description
Input		
Port 1	3	Position of the LiMMS for 3 axes (with respect to orgin)
Port 2	1	State of the H-Drive (see table B.5)
Port 3	3	Velocity of the LiMMS for 3 axes
Port 4	3	Output of EOS-sensor for 3 axes
Port 5	3	Output of EPD-sensor for 3 axes
Port 6	2	Output of AVS-sensors
Port 7	1	Time since initialization procedure finished

Table B.3: H-Drive Animator (D)

Port	Width	Description
Input:		
In_I	3	Current to be send to the 3 axes (before passing safety layer)
In_Start	1	Switch H-Drive On (1) / Off (0) (before passing safety layer)
In_EOS	3	Connection to hardware: EOS-sensor
In_EPD	3	Connection to hardware: EPD-sensor
In_AVS	2	Connection to hardware: AVS-sensor
In_Pos	3	Connection to hardware: Position encoder
In_Emergency	1	Connection to hardware: Emergency button
Output		
Out_I	3	Connection to hardware: Current actually send to H-Drive
Out_Phi	3	Connection to hardware: Phase of current send to H-Drive
Out_Power	1	Connection to hardware: Switch H-Drive On (1) / Off (0)
Out_encrst	3	Connection to hardware: Reset position encoder
Out_status	1	State of the H-Drive (see table B.5)
Out_Substate	3	States of the separate axes
Out_CtrEn	1	Enable (1) or disable (0) external controller, defined by user
Out_HTime	1	Time since initialization procedure finished
Out_Pos	3	Position of the LiMMS for 3 axes
Out_Vel	3	Velocity estimate of the LiMMS for 3 axes
Out_Result	3	Debug information: value of variable *RESULT* (zero search)
Out_DPhi	3	Debug informatio: value of variable *DPhi* (zero search)
Out_CtrlOut	3	Debug information: output of internal PID Controller
Out_CtrlSP	3	Debug information: output of setpoint generator

Table B.4: Control block with software (E)

(Sub)code	State	Description
	During Initialization:	
8	Waiting for start	Waiting for signal to start (In_Start)
11	Test	Movetest of vibration procedure
12	Zero search	Executing zero-search procedure
13	Y-Align	Aligning Y-axes
14	Homing	Homing
15	Moving	Moving axes to starting position
7	Aligning failed	Zero-search procedure failed
	During Operation:	
..0..	Ready	LiMMS is operating ok
..2..	End of stroke	Eos-sensor activated (hit end-stop)
..3..	Position violation	LiMMS entered airbag region
..4..	Velocity violation	Moving to fast
..5..	Current violation	Overcurrent
..6..	Angle violation	Angle between Y- and X-axis too big
	Emergency:	
17	Emergency stop	Emergency button was hit

Table B.5: State indications for the H-Drive

B.2 Important Parameters

Several settings of the H-Drive can be changed by setting parameters in the software. The following tables show the most important variables together with their position in the software and links to sections in this report where the variable is described in more detail. For convenience the parameters have been grouped according their use: system settings, zero search procedure, initialization and safety layer.

B.2.1 System Settings

Parameters with respect to system settings: number of operational axes, show position, velocity observer, magnet pitch and current settings.

`noa`

Source file:	HD_V4_HDrive.c
Location / Type:	Define
Unit:	[-]
Default value:	3
See also:	
Discription:	Number of axes: 3=operate all axes, 1=operate X-axis only.
WARNING:	Be very carefull when chaning this parameter. Memory allocation and safety operations are only designed for 3 axes (X,Y1,Y2) or only the X-axis being operational. Before using another configuration, a lot of procedures have to be modified.

SHOWPOS

Source file:	HD_V4_HDrive.c
Location / Type:	Define
Unit:	[-]
Default value:	1.0
See also:	Section 5.1
Discription:	1.0 = Show position and velocity during initialization, 0.0 = Suppress output of position and velocity during initialization..

KGAIN

Source file:	HD_V4_PID.c
Location / Type:	Define
Unit:	
Default value:	100
See also:	
Discription:	Gain of internal velocity reconstruction filter (used by the safety later for monitoring the velocity of the axes and the D-action for the controllers during initialization)

TAU

Source file:	HD_V4_Motor.c
Location / Type:	Define
Unit:	[m]
Default value:	0.012
See also:	Section 1.1, 1.3
Discription:	Magnet pitch of the LiMMS (distance between the North and South poles of the magnets in the rails)

Imax_Low

Source file:	HD_V4_HDrive.c
Location / Type:	Define
Unit:	[A]
Default value:	4.0
See also:	Section 4.1
Discription:	Maximum allowed current during normal use

Imax_High	
Source file:	HD_V4_HDrive.c
Location / Type:	Define
Unit:	[A]
Default value:	8.0
See also:	Section 4.1
Discription:	Maximum allowed current during critical operations performed by the safety layer.

B.2.2 Parameters zero search procedure

Parameters with respect to zero search procedure: current settings, detection level, pulse-period and maximum drift.

Imax_Algn	
Source file:	HD_V4_HDrive.c
Location / Type:	Define
Unit:	[A]
Default value:	5.0
See also:	Section 2.3
Discription:	Maximum allowed current during zero search procedure.

DETECTION_LEVEL	
Source file:	HD_V4_Movetest.c
Location / Type:	Define
Unit:	[m]
Default value:	10e-6
See also:	Section 2.2, 2.3
Discription:	Detection level of zero-search procedure.

DELTA	
Source file:	HD_V4_Movetest.c
Location / Type:	Define
Unit:	[sec]
Default value:	3e-3
See also:	Section 2.2
Discription:	Duration of Δ in vibration pulse.

ZERO_MAX_DRIFT	
Source file:	HD_V4_HDrive.c
Location / Type:	Define
Unit:	[]
Default value:	5e-4
See also:	Section 2.3, 3.1
Discription:	Maximum allowed drift from starting point during zero-search procedure.

B.2.3 Parameters initialization

Paramters with respect to initialization: method of Y-alignment, alignment speed, homing speed, PID-controller moving stage and end-position after initialization.

ALIGN_Y_TO_CENTRE	
Source file:	HD_V4_HDrive.c
Location / Type:	Define
Unit:	[#define / #undef]
Default value:	#undef
See also:	Section 3.2
Discription:	Choose procedure for aligning X-axis with respect to Y-axes by defining/undefining this parameter. Read section 3.2 for more information.

vyalgn	
Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	[m/sec]
Default value:	0.03
See also:	Section 3.2
Discription:	Maximum speed of aligning X-axis with respect to Y-axes

YALGN_TREST	
Source file:	HD_V4_HDrive.c
Location / Type:	Define
Unit:	[sec]
Default value:	0.2
See also:	
Discription:	Delay between centering Y-axes and starting the homing procedure.

`vh(i)`

Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	[m/sec]
Default value:	+0.1 / -0.1 / -0.1 respectively for X / Y1 / Y2
See also:	Section 3.3
Discription:	Homing speed of axis of axis i

`p_move(i)`

Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	[m/sec]
Default value:	1000 (for all axes)
See also:	Section 3.3, 3.4, Figure 3.1
Discription:	PID-Controller for moving axes: P-action for axis i

`i_move(i)`

Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	[m/sec]
Default value:	100 (for all axes)
See also:	Section 3.3, 3.4, Figure 3.1
Discription:	PID-Controller for moving axes: I-action for axis i

`d_move(i)`

Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	[m/sec]
Default value:	100 (for all axes)
See also:	Section 3.3, 3.4, Figure 3.1
Discription:	PID-Controller for moving axes: D-action for axis i

target(i)	
Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	[m]
Default value:	-0.3 / +0.5 / +0.5 respectively for X / Y1 / Y2
See also:	Section 3.4
Discription:	Final position of axis i after initializing

TREST	
Source file:	HD_V4_HDrive.c
Location / Type:	Define
Unit:	[sec]
Default value:	1.0
See also:	
Discription:	Delay between end of moving procedure and enabling external controller

B.2.4 Parameters safety layer

Parameters with respect to safety layer: airbag parameters, margins, PD-controller airbag, P-controller velocity brake, maximum speed.

maxpos(i)	
Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	[m]
Default value:	+0.5 / +1.05 / +1.05
See also:	Section 4.2, Table 4.1
Discription:	Airbag boundaries: maximum coordinate for axis i (physically allowed).

minpos(i)	
Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	[m]
Default value:	-0.60 / -0.05 / -0.05
See also:	Section 4.2, Table 4.1
Discription:	Airbag boundaries: minimum coordinate for axis i (physically allowed).

`margin(i)`

Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	[m]
Default value:	0.03 / 0.05 / 0.05
See also:	Section 4.2, Table 4.1
Discription:	Thickness of airbag for axis i

`p_airbag(i)`

Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	
Default value:	100 / 100 / 100
See also:	Section 4.2, Figure 4.2
Discription:	PD-Controller for airbag: P-action for axis i

`d_airbag(i)`

Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	
Default value:	25 / 25 / 25
See also:	Section 4.2, Figure 4.2
Discription:	PD-Controller for airbag: D-action for axis i

`maxspeed(i)`

Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	
Default value:	1.0 / 1.0 / 1.0
See also:	Section 4.3
Discription:	Maximum allowed speed for axis i

`p_vel_brake(i)`

Source file:	HD_V4_HDrive.c
Location / Type:	mdlInitializeConditions()
Unit:	
Default value:	25 / 25 / 25
See also:	Section 4.3, Figure 4.5
Discription:	P-Controller for velocity brake: P-action for axis i

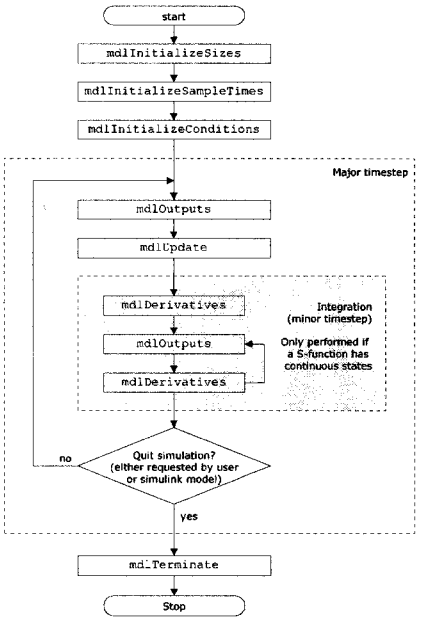

Figure B.5: Flow chart of a typical Simulink program

B.3 Description of the code

This section contains a description of the code, summarized per source-file. For an overview of the most important defines and variables one is referred to section B.2.

To make a custom-made component interact with MATLAB/Simulink, the component should have a set of standard functions that can be called by Simulink. Figure B.5 shows the order in which these functions are called:

- `mdlInitializeSizes`: set width of I/O ports, allocate memory for work vectors, etc.
- `mdlInitializeSampleTimes`: tell the Simulink simulator at what sample-times the model should be evaluated. The choice of sample-times should be compatible with the integration method[1] that is used by Simulink.
- `mdlInitializeConditions`: initialize variables that are used within the code.
- `mdlOutputs`: calculate outputs of the component.
- `mdlUpdate`: update (discrete) states of the component.

[1]Because the H-Drive component will be used with dSPACE / RealTime-Workshop, the integration method of Simulink is set to Euler, with a fixed step. The code for the H-Drive can therefore use continuous sample times.

- `mdlDerivatives`: calculates the derivatives of the states. This function is only used during minor (iterating) time steps. This code is therefore not used by the Real Time Workshop.
- `mdlTerminate`: end of simulation tasks. Above functions are located in the main-file `HD_V4_Hdrive.c`. The control-functions use a set of supporting functions that are described in the next sub-sections.

B.3.1 HD_V4_HDrive.c

This is the main-file that ties together the other files and sets some important defines.
The main file also contains all compulsory Simulink functions that were mentioned in the introduction of this section (figure B.5).

- Defines and includes
- function `mdlInitializeSizes()`
  Compulsory Simulink function, needed for setting I/O-ports, work-vectors, etc.
- function `mdlInitializeSampleTimes()`
  Compulsory Simulink function, needed for setting sample times (continuous) and time-offset (0 sec).
- function `mdlInitializeConditions()`
  Compulsory Simulink function, needed for initialization of some variables:
  - Initialize globals in the work-space for re-entrancy
  - Initialize variables for the main-loop (e.g. airbag parameters, controller settings, etc)
  - Initialize the H-Drive-API by calling `HD_V4_Vapi\Hdrive_initialize()`
  - Initialize PID-controller by calling `HD_V4_PID\pid_ini()` (also needed for velocity observer)
- function `mdlOutputs()`
  x: continuous states, y: outputs, uPtrs: inputs.
  Before the different stages of the H-Drive are looked at, the function first checks if the user pressed the emergency-button. It his is true and the system is in its `TEST` or `ZERO_SEARCH` mode the power is immediately switched off, because it is not possible to use a controller to bring the system to standstill as quick as possible with the yet unknown system-gain. Otherwise the `EMERGENCY_STOP_SYSTEM` cases is used to stop the system as quick as possible.
  At every stage the needed motor-currents have to be calculated and send to the current amplifier using `HD_V4_Motor\send_motor_command()`, according to the current state the system is in:
  - `WAITING_FOR_START`
    Initialize variables needed during the zero-search procedure, set motor-currents to zero, switch current amplifiers on.

- TEST
  Test if there is enough movement to start the zero-search procedure (using HD_V4_Movetest\Hdrive_movetest_out()). If the displacement is bigger than the DETECTION_LEVEL during three successive tests, the test succeeds and the zero-search procedure is started. If the test fails, the amplitude of the reference current is increased and the phase is shifted 90 degrees until the test succeeds, or the amplitude exceeds the maximum allowed level (followed by stage ALIGNING_FAILED).
- ZERO_SEARCH
  Execute the zero-search procedure by calling HD_V4_Movetest\Hdrive_zerosearch_out() (that is, search the phase at which the system moves less than DETECTION_LEVEL when even the maximum current amplitude is applied)
- Y_ALIGN
  Align the Y-axes with respect to the X-axis using HD_V4_Vapi\Hdrive_yalign_out() to prevent the axes from wedging during the homing procedure.
- HOMING
  Home LiMMS using HD_V4_Vapi\Hdrive_homing_out()
- MOVING
  Move LiMMS to the desired location using HD_V4_Vapi\Hdrive_moving_out() before control is handed over to the user.
- POS_VIOLATION***
  I_VIOLATION***
  These states do no longer exist in Version 4 of the software. Safety-operations are carried out per axis in state READY. Assigning a global state for all axis is not possible, because axes can be in different states.
- END_OF_STROKE
  Power is switched off to prevent further damage.
- VEL_VIOLATION
  This state brakes the LiMMS when the velocity becomes too big during the zero-search procedure. This state would not get activated in practice because the velocity cannot exceed the maximum limit during alignment, because the safety-layer already cancels the zero-search after a relatively small drift ZERO_MAX_DRIFT.
- ALIGNING_FAILED
  The power is switched off because the zero-search procedure failed.
- READY
  After a successfully initialization the system comes into the READY-state in which control is handed over to the user. To protect the system from dangerous situations, a safety layer is used.
  Function HD_V4_PID\pidout() is used to estimate the velocity, after which HD_V4_Safety\safety_Check() is used to check for possible violations. Violations are returned by setting the correct global variables (per axis):

* `pos_violation` is set whenever the LiMMS enters the airbag-region.
  Function `HD_V4_Safety\safety_pos_airbag()` is used to send the LiMMS back into the safe area.
* `vel_violation` is set when the velocity of the LiMMS becomes too high.
  Function `HD_V4_Safety\safety_vel_airbag()` is used to slow down the LiMMS.
* `ang_violation` is set when the angle between the x- and y-axes becomes too big function `HD_V4_Safety\safety_angle_viol()` is used to speed down the LiMMS and align the Y-axes with respect to each other after initializing the safety-operation with `HD_V4_Safety\safety_angle_viol_ini()`.

Two other violations remain. These are checked at another place in the code:

* The end-of-stroke sensor gets activated when the LiMMS hits one of the springs located at the end of the axes, near This situation is checked for in `HD_V4_HDrive\mdlUpdate()`
* Overcurrent protection is included in `HD_V4_Motor\send_motor_command()`

During this stage the time since initialization is also calculated and fed back to the user.

- `ANG_VIOLATION`
  When the angle between the x- and y-axes becomes too big, this stage can be used to brake the speed of all axes, while at the same time Y2 is pulled towards Y1 to correct the angle-violation.
  Function `HD_V4_Safety\safety_angle_viol()` is used to achieve this task.
- `EMERGENCY_STOP_SYSTEM`
  This stage uses HD_V4_Safety\safety_direct_stop() to bring the H-Drive to standstill as quick as possible. When the velocity of all axes are approximately zero, the current-amplifier is switched off.

- function `mdlUpdate()`
  x: continuous states, y: outputs, uPtrs: inputs
  Function `mdlOutputs()` only generates the outputs to be send to the current amplifier as described in the previous item. An extra controlling mechanism is needed to switch between the different stages at the right moment. This is exactly the purpose of the `mdlUpdate()` function.
  Before looking at the different cases, the functions first determines whether or not the software switch `IN_START_ALGN` is turned on.

  - `WAITING_FOR_START`
    Wait until the user gives the sign to start the alignment procedure. Jump to state `TEST` to start alignment.
  - `TEST`
    Test if the movement-test of the zero-search procedure from the `mdlOuput()` function generated sufficient movement to proceed with

the zero-search procedure.
After a successful test the state changes to ZERO_SEARCH.
Erroneous situations result in ALLIGNING_FAILD (movement-test failed) or END_OF_STROKE (end stroke hit).

- ZERO_SEARCH
  This stages controls the zero-stage procedure.
  Possible failures:
  * The zero-search procedure as described in section 2.3 is not able to find the zero: results in ALLIGNING_FAILD
  * The LiMMS drifts to far away from its starting point because of too heavy vibrations or an unstable point: resulting in ALLIGNING_FAILD
  * The end of stroke is hit: results in END_OF_STROKE

  When the zero-search succeeds, PHI is incremented by 90 degrees to get gain $cos(p - \varphi) = 1$ and the state changes to HOMING or Y_ALIGN, depending on the number of operational axes:
  * 1-Dimensional (only X-axis used): switch to HOMING-state and initialize the first part of homing [2] by calling HD_V4_Vapi\Hdrive_start_homing().
  * 3-Dimensional (all axes in use); switch to Y_ALIGN to align the Y2-axis with respect to the Y1-axis. Initialize the alignment by calling HD_V4_Vapi\Hdrive_yalign_restart(). Sub-state YALGN_STATE is set to YALGN_FIND_AVS_B to indicate that the program is looking for the location of AVS-sensor B[3].
- ALLIGNING_FAILD
  After an alignment error the system can not be used anymore. Therefore this state can only change to a worse situation: an activated end-of-stoke sensor (state END_OF_STROKE).
- Y_ALIGN
  During the first part of the Y-Alignment (YALGN_FIND_AVS_B) HD_V4_Vapi\Hdrive_yalignsynchronize() is used to detect the state of sensor avs(1). When this sensor gets activated for the first time, YALGN_STATE is set to YALGN_FIND_AVS_A and HD_V4_Vapi\Hdrive_yalign_restart() is called to store the position of the sensor and calculate the jogging-parameters for the next part. What happens during the next part, depends on the aligning-mode as described in section 3.2:
  * Method 1 (ALIGN_Y_TO_CENTRE)
    During the second part of the Y-Alignment (YALGN_FIND_AVS_B) HD_V4_Vapi\Hdrive_yalignsynchronize() is used to detect the state of sensor avs(0). When this sensor gets activated for the first time, YALGN_STATE is set to YALGN_CENTRE and HD_V4_Vapi\Hdrive_yalign_restart() is called to store the position of the sensor and calculate the trajectory-parameters for

[2] First part of homing: move toward the origin of the axis untill the homing sensor gets activated for the first time.
[3] In some parts of this report sensor A is called avs(0) and sensor B is called avs(1)

the next part.
During the third part of the Y-Alignment (YALGN_FIND_AVS_B) the Y2 axis is moved to the centre between avs(0) and avs(1) using a PID-controller. HD_V4_Vapi\Hdrive_yalignsynchronize() is used to detect when the desired position gets reached. Finally sub-state YALGN_STATE is set to READY, while the global and substate() are set to HOMING.
HD_V4_Vapi\Hdrive_start_homing() is called to initialize the homing-parameters (jogging profile).

* Method 2 (Not ALIGN_Y_TO_CENTRE)
During the second part of the Y-Alignment (YALGN_FIND_AVS_B) HD_V4_Vapi\Hdrive_yalignsynchronize() is used to detect the state of sensor avs(0). When this sensor gets activated for the first time, YALGN_STATE is set to READY, while global state and subsate() are set to HOMING and
HD_V4_Vapi\Hdrive_start_homing() is called to initialize the homing-parameters (jogging profile).

During the alignment the state of the end-of-stroke sensor is monitored and state END_OF_STROKE gets activated when necessary.

- HOMING
During the first part of the homing-procedure, the LiMMS moved until the homing sensor gets activated for the first time
(HD_V4_Vapi\Hdrive_synchronize() evaluates to true with option HdriveHOME1).
At the end of the first part the jogging speed is lowered to get a more accurate measurement. To change the direction of movement, the sign of the jogging velocity is changed.
HD_V4_Vapi\Hdrive_start_homing() is called to initialize the jogging parameters for the second part.
During the second part of the homing-procedure the LiMMS is moved at a lower speed to get a more accurate measurement that indicates the location of the homing-markers.
When HD_V4_Vapi\Hdrive_synchronize() evaluates to true with option HdriveHOME2 the homing procedure is ready. The state switches to MOVING and HD_V4_Vapi\Hdrive_start_moving() is called to calculate the parameters for the trajectory needed for the moving procedure.
During homing the state of the end-of-stroke sensor is monitored and state END_OF_STROKE gets activated when necessary.

- MOVING
Function HD_V4_Vapi\Hdrive_synchronize(MOVING) is used to detect when the moving-procedure is finished.
When all axes have reached the desired position, the initialization of the H-Drive is finished: the state is set to READY, the external controller gets activated and the combined status of the individual axes is showed instead of the global initialization status.
During moving the state of the end-of-stroke sensor is monitored and state END_OF_STROKE gets activated when necessary.

  - VEL_VIOLATION
    Overspeed violations are handled per axis in the READY-state. The global VEL_VIOLATION state can be used during the alignment procedure. Because recovery from overspeed during alignment is not possible, VEL_VIOLATION can only change to state END_OF_STROKE.
  - READY
    The READY state only checks for position and angle violation. (state END_OF_STROKE and ANG_VIOLATION respectively)

- function mdlDerivatives()
  This function is implemented in Simulink to provide the derivatives of the state-variables. The derivatives dx[] are calculated in this manner:

  - WAITING_FOR_START
    dx[*status_id*] is set to 0.0
  - TEST and ZERO_SEARCH
    Empty case, dx[] not altered.
  - Y_ALIGN
    Function HD_V4_Vapi\Hdrive_yalign_dif() is used to calculate the derivatives.
  - HOMING
    Function HD_V4_Vapi\Hdrive_homing_dif() is used to calculate the derivatives.
  - MOVING
    Function HD_V4_Vapi\Hdrive_moving_dif() is used to calculate the derivatives.
  - READY
    Function HD_V4_Vapi\Hdrive_ready_dif() is used to calculate the derivatives.
  - ALLIGNING_FAILD, VEL_VIOLATION, POS_VIOLATION, ANG_VIOLATION and EMERGENCY_STOP_SYSTEM
    Function HD_V4_Vapi\Hdrive_violation_dif() is used to calculate the derivatives.

- function mdlTerminate()
  Empty function, but compulsory for Simulink.

- Trailer code to define the file as MATLAB MEX-file.

B.3.2 HD_V4_HDrive.h

Defines to identify the several stages and synchronization stages.

B.3.3 HD_V4_IOPorts.h

Overview of I/O ports and accessory defines (see also table B.4 on page 57)

B.3.4 HD_V4_Work.c

This file contains code to allocate global memory for re-entrancy.
During a Simulink simulation, the MEX-program is called repeatedly. Usually, global variables disappear when a computer program terminates. This means that it would not be possible to use the value of a declared variable in a next program call (read: next iteration step / time step). Therefore MATLAB implemented the possibility of re-entrancy.

- function `rwrk_init(int *pivar, int *pidx, int nrw, SimStuct *S)`
  - This function allocates re-entrancy memory for a set of `nrw` variables.
  - `pivar` stores the index of the current variable-set (the first set gets index `0`, the second set gets index `1`, etc).
  - `pidx` stores the index of the first variable in the current variable set. See example for more information.
  - The `SimStruct` tells MATLAB which Simulink block will use the re-entrancy variable

  Example:
  You want to create the following structure: the first set of variables contains 3 variables, the second one contains a single value and the third set contains 3 values.
  - During the first function call you use `nrw=3`, in the second call `nrw=1`, in the third `nrw=3`
  - After the first call value `0` is stored at address `*pivar`, the second call stores `1` at the specified address `*pivar` and the third call stores `2`.
  - After the first call value `0` is stored at address `*pidx`, the second call stores `3` at the specified address `*pivar` (during the first call 3 variables were created, so the first variable of the second set will be at position 4), and the third call stores `4` (during the first call 3 variables were created, the second call declared 1 variable, so the first variable of the third set will be at position 5).

- function `rwrk_init_all()`
  Generates all function calls to `rwrk_init_all()` that are needed to generate the data-structure for the variables used by the H-Drive Program V4.0

B.3.5 HD_V4_Movetest.c

The Movetest-file contains the functions and parameters needed for the zero-search procedure that is described in chapter 2. The flowchart, depicted in figure 2.3, shows a comprehensive overview of the code.

- defines to set settings of procedure: vibration period and detection level.

- function `Hdrive_movetest_out(real_T *u, real_T *pos, Simstruct *S)`
  The function tests whether there is enough movement to start with the zero-search procedure when using the current settings of the current-amplitude and phase-angle. If the displacement is bigger than the detection-level during three successive measurements, the zero-search is started. However, when the displacement is too small, the reference current is increased and the phase-angle is shifted 90 degrees.

- function `Hdrive_zerosearch_out(real_T *u, real_T *pos, Simstruct *S)`
  Function to align the H-Drive, that is: finding the phase at which the movement of the axis stays beneath the detection level, even when applying the maximum allowed current.

In above functions `*u` is a pointer to an array to store the controller outputs, `*pos` is a pointer to an array with the positions of the axes of the H-Drive and `*S` contains the current SimStruct.

B.3.6 HD_V4_Safety.c

This file comprises the following functions that are used to create a safety-layer to protect the H-Drive from dangerous situations:

- function `safety_check(real_T *u, real_T *pos, real_T *vel, Simstruct *S)`
  This function is used by the main-file to check for different safety-violations: velocity, position and angle violation[4]. When a violation is detected, the sub-state and according violation-parameter of the current axis are set. Overcurrent is not detected by this function[5].

- function `safety_pos_airbag(real_T *u, real_T *pos, real_T *vel, Simstruct *S)`
  This function uses a PD-controller[6] to push the system back into the safe-area when boundary trespassing occurs (see section 4.2 and figure 4.1 at page 24). After trespassing the function can be used to brake the movement to zero-velocity.

- function `safety_vel_airbag(real_T *u, real_T *pos, real_T *vel, int_T i Simstruct *S)`
  The velocity-brake (section 4.3) consists of a simple P-controller[7] that controls the velocity of axis `i` to its maximum allowed value.

- function `safety_angle_viol_ini(real_T *t, real_T *pos, Simstruct *S)`
  This function calls `HD_V4_Jog\p2p_ini()` to create the trajectory that is needed by `safety_angle_viol()` to correct the angle-violation.

[4]Position and angle violations are only reported when the initialization has finished and the exact position of the H-Drive is known.

[5]Overcurrent is detected when sending the current to the motor: `HD_V4_Motor\send_motor_command()`

[6]Settings PD-Controller (per axis): `p_airbag()`, `d_airbag()`

[7]Settings PD-Controller (per axis): `p_vel_brake()`

- function `safety_angle_viol(real_T *u, real_T t, real_T *pos, real_T *vel, cons real_T *xc, Simstruct *S)`
  The following actions are taken to prevent damage, caused by a tilt:
  - Brake X-axis to zero velocity using a P-controller[8].
  - Brake Y1-axis to zero velocity using a P-controller. The maximum control-current is limited to 80% of the maximum allowed value to make sure that the contoller for Y2 can follow Y1.
  - Use a PID-controller[9] to control the Y2-axis to remove the tilt between the Y-axes by making the difference between the Y-coordinates follow the profile calculated by `safety_angle_viol_ini()`.

- function `safety_direct_stop(real_T *u, real_T *pos, real_T *vel, cons real_T *xc, Simstruct *S)`
  This function brings the H-Drive to a standstill as fast as possible. The function is similar to `safety_angle_viol()` with the difference that the controller for Y2 follows the Y1 coordinate (instead of $Y1 + profile$).

In above functions `*u` is a pointer to an array to store the controller outputs, `*pos` is a pointer to an array with the positions of the axes of the H-Drive, `*vel` is a pointer to an array with the velocities, `*xc` is a pointer to an array with the state-variables and `*S` contains the current SimStruct.

B.3.7 HD_V4_Motor.c

Motor.c contains the code to drive the motors of the H-Drive:

- define `TAU`:
  Defines the magnet pitch. This value is needed for the commutation.

- function `send_motor_command(real_T *u, real_T *pos, Simstruct *S)`
  First, this function determines the maximum current level for the current state. In case of an emergency, the current can be increased to prevent damage.
  After this the function sends the (if necessary clipped) current to the output, and applies commutation to keep the motor-constant at its maximum level. (see section 1.1)

In above function `*u` is a pointer to an array to store the controller outputs, `*pos` is a pointer to an array with the positions of the axes of the H-Drive and `*S` contains the current SimStruct.

B.3.8 HD_V4_PID.c

Implementation of a simple PID-Controller.

- Defines: gain `KGAIN` of the velocity-observer

[8] Parameter P is set to `p_vel_brake()`
[9] Settings PID-Controller (per axis): `p_move()`, `i_move()`, `d_move()`

- function `pid_ini(real_T *xc, real_T *pos)`
  Initialize state-variables needed by the controller: set `xc[i]` to the current position and reset the integrated position-error `xc[i+4]`.

- function `pid_dif(real_T *dxcdt, const real_T *xc, real_T *pos, real_T *qref)`
  Function to calculate the derivatives of the state variables. This function is inherited by some functions that are used in `mdlDerivatives()` in the main-file.

- function `pid_out(real_T *u, real_T *vel, const real_T *xc, real_T *pos, real_T *qref, real_T *vref, Simstruct *S)`
  This function estimates the velocity, using the same observer as used in `pid_dif()` and returns the output of the PID-controller[10].

In above functions `*u` is a pointer to an array to store the controller outputs, `*pos` is a pointer to an array with the positions of the axes of the H-Drive, `*vel` is a pointer to an array with the velocities, `*xc` is a pointer to an array with the state-variables and `*S` contains the current SimStruct.

B.3.9 HD_V4_Jog.c

This file contains functions to generate a third order setpoint profile as depicted in figure B.6. Two types of setpoint profiles can be used:

- Jogging: the movement is started using `jog_ini()` and continues at a constant velocity until `jog_stop()` is called to stop movement in a gentle way. This function is used to generate the setpoint for the movement during homing and alignment of the Y-axes.

- Point-to-point movement: the complete trajectory is calculated using `p2p_ini()`, based on the desired start- and end-position, and some other parameters.

The complete set of functions comprises the following components:

- Defines with the memory-locations of the set-point parameters

- function `jog_ini(real_T xstart, real_T tstart, real_T vdes, real_T tdes, real_T maxjerk, int_T i, Simstruct *S)`
  This function calculates the parameters for a setpoint profile to generate a jogging-movement. By setting t_4 to a very big value, the system keeps moving at a constant velocity. The movement can be stopped by calling the `jog_stop()`-function that calculates the parameters for the fourth period and further.
  Arguments of the function:

  - `xstart, tstart`: starting position and starting time of the setpoint
  - `vdes, tdes`: desired jogging speed and time to reach v_{des}
  - `maxjerk`: maximum jerk during the acceleration phase

[10]Settings PID-Controller (per axis): `p_move()`, `i_move()`, `d_move()`

- i: axis id

Table B.6 shows how the parameters for the third order setpoint from figure B.6 can be calculated.

- function `jog_get(real_T *qref, real_T *vref, real_T *aref, real_T t, int_T i, Simstruct *S)`
  This function calculates the current reference values (position, speed, velocity) using the setpoint-parameters for axis `i` at time `t`.
  Take care: the locations to store the reference values `qref`, `vref` and `aref` are pointers to the memory-locations to store the value for axis `i`, not a pointer to an array to store the reference values of all axis.
  For example:
  `qref[]`, `vref[]` and `aref[]` are three arrays that have to be used to store the setpoint-values for the X, Y1 and Y2 axis.
  The following call should be used to get the current setpoint profiles for the Y2-axis (axis id: 2):
  `jog_get(&qref[2], &vref[2], &aref[2], time, 2, S)`

- function `jog_stop(real_T t, int_T i, Simstruct *S)`
  This function ends the jogging-movement for axis `i`

- function `jog_satus(real_T t, int_T i, real_T tset, Simstruct *S)`
  This function can be used to monitor the status of the jogging movement. Value 1 is returned when the jogging-movement ends (that means when time `t` is bigger than `t7 + tset`)
  `tset` is the settiling time: an extra pause that is added to the profile to make sure that the system has reached its desired end-position and vibrations damped out.

- function `p2p_ini(real_T xstart, real_T tstart, real_T xend, real_T vdes, real_T tdes, real_T maxjerk, int_T i, Simstruct *S)`
  This function calculates the parameters for a set-point profile to generate a point-to-point movement. The function is nothing more than a combination of `jog_ini()` and `jog_stop()`.
  The value of t_4 is chosen such the end-position of the setpoint profile is equal to the desired position `xend`[11].

- function `p2p_get(real_T *qref, real_T *vref, real_T *aref, real_T t, int_T i, Simstruct *S)`
  This function calculates the current reference values (position, speed, velocity) using the setpoint-parameters for axis `i` at time `t`. Because jogging and moving use the same equations, this function is nothing but a copy of `jog_get()`.

- function `p2p_status(real_T t, int_T i, real_T tset, Simstruct *S)`

[11] Because the time at which the period with constant velocity v_{des} ends is not know in jogging-mode, t_4 contains the time at which jog_stop() was called. However, when defining a point-to-point movement, t_4 can be caculated beforehand. A correct value of t_4 can cause a trajectory that started at position x_{start}, ends at position x_{des}.

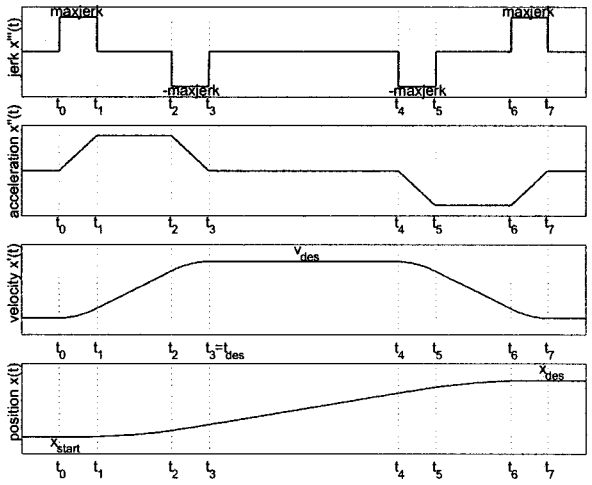

Figure B.6: Trajectory as genereated by HD_V4_Jog.c

This function can be used to signal the status of the point-to-point movement. The function is merely a copy of `jog_status()`.

B.3.10 HD_V4_Vapi.c

The H-Drive API contains a set of useful functions to control H-Drive specific tasks like homing, moving and aligning the y-axes:

- `Hdrive_initialize(real_T *pos, SimStruct *S)`
  This function initializes the H-Drive: reset parameters for the zero-search procedure, reset the encoders, etc.

- `Hdive_start_homing(real_T *pos, real_T t, SimStruct *S)`
  Initialize the homing procedure for all axes simultaneously: initialize the jogging-profile using `HD_V4_Jog\jog_ini()` and initalize the PID-controller using `HD_V4_PID\pid_ini()`.
  The jogging profile is not initialized here when the end-point-detector is already activated, because this means that the first part of the homing procedure (roughly searching the epd-detector) can be skipped.

- `Hdrive_homing_dif(real_T *dx, real_T *x, real_t *pos, real_T t, SimStruct *S)`
  This function calculates the derivatives of state xc during the homing-state using `HD_V4_Jog\jog_get()` to get the reference position and `HD_V4_PID\pid_dif()` to calculate the derivatives using the actual and desired positions.

- `Hdrive_homing_out(real_T *u, real_T *vel, real_T *x, real_T *pos, real_T t, SimStruct *S)`
  This function calculates the controller output during the homing-stage.

t_0	Just initial conditions	$t_0 = t_{start}$ $a_0 = a(t=t_0) = 0$ $v_0 = v(t=t_0) = 0$ $s_0 = s(t=t_0) = x_{start}$
$t_0 \rightarrow t_1$	$j(t) = jerk$ $a(t) = jerk \cdot (t-t_0)$ $v(t) = \frac{1}{2} jerk \cdot (t-t_0)^2$ $s(t) = \frac{1}{6} jerk \cdot (t-t_0)^3$	$a_1 = a(t=t_1) = jerk \cdot \delta$ $v_1 = \frac{1}{2} jerk \cdot \delta^2$ $s_1 = \frac{1}{6} jerk \cdot \delta^3$
$t_1 \rightarrow t_2$	$j(t) = 0$ $a(t) = a_1 = jerk \cdot \delta$ $v(t) = v_1 + a_1 \cdot (t-t_1)$ $s(t) = s_1 + v_1 \cdot t$ $+\frac{1}{2} a_2 \cdot (t-t_1)^2$	$a_2 = a(t=t_2) = a_1$ $v_2 = v(t=t_2)$ $= v_1 + a_1 \gamma$ $= v_1 + a_1 (t_{des} - 2\delta)$ $= jerk \cdot \delta \cdot (t_{des} - \frac{3}{2}\delta)$ $v_2 = v(t=t_2) = \frac{1}{2} jerk \cdot \delta^2$ $s_2 = s(t=t_2) = \frac{1}{6} jerk \cdot \delta^3$ $+\frac{1}{2} jerk \cdot \delta^2 \gamma + \frac{1}{2} jerk \cdot \gamma^2 \delta$
$t_2 \rightarrow t_3$	$j(t) = -jerk$ $a(t) = a_2 - jerk \cdot (t-t_2)$ $v(t) = v_2 + a_2 \cdot (t-t_2)$ $-\frac{1}{2} jerk \cdot (t-t_2)^2$ $s(t) = s_2 + v_2 \cdot t$ $+\frac{1}{2} a_2 \cdot (t-t_2)^2$ $-\frac{1}{6} jerk \cdot (t-t_0)^3$	$a_3 = a(t=t_3) = 0$ $v_3 = v(t=t_3)$ $= v2 + a2\delta - \frac{1}{2} jerk \cdot \delta^2$ $= -jerk \cdot \delta^2 + t_{des} \cdot jerk \cdot \delta$ $= v_{des}$ $s_3 = s(t=t_3) = ...$
$t_3 \rightarrow ...$	Etc.	Etc.

Table B.6: Some equations for determining the jogging parameters

HD_V4_jog_get() is used to get the setpoint, while HD_V4_PID\pid_out() is used to calculate the controller output.

- Hdrive_start_moving(real_T *pos, real_T t, SimStruct *S)
  Initialize the moving procedure for all axes simultaneously: initialize the setpoint-profile using HD_V4_Jog\p2p_ini() and initialize the PID-controller using HD_V4_PID\pid_ini()

- Hdrive_moving_dif(real_T *dx, real_T *x, real_T *pos, real_T t, SimStruct *S)
  This function calculates the derivatives of state xc during the moving-state, using HD_V4_Jog\p2p_get() to get the reference position and HD_V4_PID\pid_dif() to calculate the derivatives using the actual and desired positions.

- Hdrive_moving_out(real_T *u, real_T *vel, real_T *x, real_T *pos, real_T t, SimStruct *S)
  This function calculates the controller-output during the moving-stage. HD_V4_p2p_get() is used to get the setpoint, while HD_V4_PID\pid_out() is used to calculate the controller output.

- Hdrive_synchronize(real_T *pos, real_T *tar, real_T t, int_T sync_id, SimStruct *S)
  This function is used to control the course of the homing and moving procedure:

  - When sync_ID is set to HdriveHOME1, this means that the H-Drive is performing the first part of the homing-procedure: roughly seeking the position of the epd-detector.
    The first time an epd-detector is detected for a particular axis, the jogging profile for that axis is stopped using HD_V4_Jog\jog_stop(). The function returns value 0 until all axes detected the epd.
  - When sync_ID is set to HdriveHOME2, this means that the H-Drive is performing the second part of the homing-procedure: accurately seeking the position of the epd-detector.
    When the epd-sensor gets de-activated for the first-time, the position of that axis is stored in the variable Epd_Marker(i) to indicate the exact position of the endpoint-detector. Further more HD_V4_Jog\jog_stop)_. is called to stop the jogging-movement of that axis.
    The function returns value 0 until the epd-position is known for all axes. At that point the position of the H-Drive is reset so that position zero, coincides with the position of the epd-marker. Also, the current position is transformed with respect to the new origin and the PID-controller gets resetted.
  - When sync_ID is set to HdriveMOVE, the function return 0 until the moving-procedure has finished. The PID-controller gets initialized and output OUT_CTRL_EN is set to 1, so the user-defined controller gets enabled.

- `Hdrive_ready_dif(real_T *dx, real_T *x, real_T *pos, real_T t)`
  This function calculates the derivatives of state xc during the ready-state using `HD_V4_PID\pid_dif()`.

- `Hdrive_violation_dif(real_T *dx, real_T *x, real_T *pos, real_T t)`
  This function calculates the derivatives of state xc during a violation-state using `HD_V4_PID\pid_dif()`.

- `Hdrive_yalign_restart(real_T *pos, real_T t, int_T contr_id, SimStruct *S)`
  This function is used to initialize the sub-states of the y-align procedure: initialize the jogging- or point-to-point setpoint and initialize the PID-controller.

- `Hdrive_yalign_out(real_T *u, real_T *vel, real_T *x, real_T *pos, real_T t, int_T contr_id, SimStruct *S)`
  This function calculates the controller-output during the y–align stage. `HD_V4_PID\pid_out()` is used to calculate the controller output, while `HD_V4_Jog\jog_get()` or `HD_V4_Jog\p2p_get()` is used to get the current setpoint (depending on the sub-state of the y-alignment).

- `Hdrive_yalign_synchronize(real_T t, int_T sync_id, SimStruct *S)`
  This function is used to control the course of the y-align procedure:

  - When `sync_ID` is set to `YALGN_FIND_AVS_B`, the functions returns value 0 as long as avs-sensor B is not seen.
  - When `sync_ID` is set to `YALGN_FIND_AVS_A`, the functions returns value 0 as long as avs-sensor A is not seen.
  - When `sync_ID` is set to `YALGN_CENTRE`, the functions returns value 0 as long as the final aligning position of the Y-axes has not been reached.

- `Hdrive_yalign_dif(real_T *dx, real_T *x, real_T *pos, real_T t, int_T contr_id, SimStruct *S)`
  This function[12] calculates the derivatives of state xc during the moving-state, using `HD_V4_Jog\p2p_get()` or `HD_V4_Jog\p2p_get()` to get the reference position (depending on the sub-state of the y-alignment) and `HD_V4_PID\pid_dif()` to calculate the derivatives using the actual and desired positions.

In above function `*u` is a pointer to an array to store the controller outputs, `*pos` and `*vel` are pointers to an array with the positions and velocities of the axes of the H-Drive `*x` and `*xc` point to the states and an array to store the derivatives of the states, `t` holds the current time and `*S` contains the current SimStruct.

[12]Input argument `*tar` is no longer used by this function and can therefore be removed in future releases.

B.4 Source code

B.4.1 HD_V4_HDrive.c

```
/*
     AligningHdrive V4.0 (Matlab 5.3 version)

     (c) Loy Rovers, 2001
     (c) Stef Hendriks, 2000

     last update: March 26, 2001

     This program is based on the VRS software of Rene' van de Molengraft
     and on the aligning software of Antoine Verweij

     The allignmentpulses are based on a sinus
*/



// ------------------------------------------------------------------------------
// Setup
// ------------------------------------------------------------------------------


#define S_FUNCTION_NAME  HD_V4_HDrive
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"
#include <math.h>

#define SHOWPOS                   1.0
//      1=show position and velocity during initialisation

#define pi                        3.14159265358979323846264338327 95
#define noa                       3
//      The number of axis that are operational
//      noa=1 -> axis 0=X-axis
//      noa=3 -> axis 0=X-axis, 1=Y1-axis, 2=Y2-axis

#define NINPUTS                   20
#define NOUTPUTS                  34
#define NSTATES                   7

#define NRWRK                     246
//      10*1+42*3 for HD_V4_HDrive + 37*3 for HD_V4_Jog.c
#define NIWRK                     89
//      1 for HD_V4_HDrive.c + 37 for HD_V4_Jog.c

#define Imax_Algn                 5
//      maximum current during alignment[A]
#define Imax_Low                  4
//      maximum current during normal use [A]
#define Imax_High                 8
//      maximum current during critical safety-procedures [A]

#define TREST                     1
//      delay between end of moving procedure en enabeling ext. controller [sec]
#define YALGN_TREST               0.2
//      delay between end of centring y-axes and homing [sec]
```

```
//      *** Warning: In the current setup of the H-Drive one of the Tilt-sensors
//      ** seems to be missing. The assumption that sensor AVS_A detects tilt in
//      * one direction, and B detects tilt in the other direction isn't correct
//
//      Sensor AVS_B seems to detect tilt, Sensor AVS_A gets activated when the
//      X-axis is perpendicular to the Y-axis.
//      The following define makes it possible to switch between the possibility
//      to (1) set the Y-axis in such a position that is in the middle between
//      the position where the avs-sensors get activated or (2) set the y-axis
//      in the position where AVS_B gets just activated
//
//      #define ALIGN_Y_TO_CENTRE   -> activates option (1)
//      #undef  ALIGN_Y_TO_CENTRE   -> activates option (2)

#undef  ALIGN_Y_TO_CENTRE


// ------------------------------------------------------------------------
// Control
// ------------------------------------------------------------------------


#define START_PULSE(element)        prwrk[piwrk[0]+element]
//

#define STATUS                      prwrk[piwrk[1]]
//      Global status of the H-Drive (See HD_V4_HDrive.h)
#define substate(element)           prwrk[piwrk[2]+element]
//      Local status per axis (See HD_V4_HDrive.h)
#define SHOWCOMBINED                prwrk[piwrk[3]]
//      0=Show Global state, 1=Show combined state

#define YALGN_STATE                 prwrk[piwrk[4]]
//      sub-state of aligning y-axes
#define HOMING1_READY(element)      prwrk[piwrk[5]+element]
//      1=homing phase 1 ready
#define HOMING1_ALL_READY           prwrk[piwrk[6]]
//      1=homing phase 1 ready for ALL axes

#define Hdrive_pos(element)         prwrk[piwrk[7]+element]
//      Position of H-Drive (corrected with pos_reset)
#define Hdrive_vel(element)         prwrk[piwrk[8]+element]
//      Velocity of H-Drive

#define us(element)                 prwrk[piwrk[9]+element]
//      Control output
#define PHI(element)                prwrk[piwrk[10]+element]
//      The Angle wich should be determined
#define Hdrive_result               prwrk[piwrk[11]]

#define Hdrive_time                 prwrk[piwrk[12]]
//      Current time

#define pos_reset(element)          prwrk[piwrk[13]+element]
//       position at major reset
#define posreset_yalgn(element)     prwrk[piwrk[14]+element]
//      position at virtual reset (aligning y-axes)

#define epd(element)                prwrk[piwrk[15]+element]
//      1=epd sensor activated (and recognized by controller)
```

```
#define p_move(element)            prwrk[piwrk[16]+element]
//      PID-Action (Aligning Y-axes, Homing, Moving)
#define d_move(element)            prwrk[piwrk[17]+element]
//      PID-Action (Aligning Y-axes, Homing, Moving)
#define i_move(element)            prwrk[piwrk[18]+element]
//      PID-Action (Aligning Y-axes, Homing, Moving)

#define triggertime                prwrk[piwrk[19]]
//      1=system has been triggered for start (and recognized by controller)

#define vyalgn                     prwrk[piwrk[20]]
//      speed of y-alignment m/s
#define vh(element)                prwrk[piwrk[21]+element]
//      The homing speed m/s
#define target(element)            prwrk[piwrk[22]+element]
//      Target (point to move to after initialization)
#define commdir(element)           prwrk[piwrk[23]+element]
//      Directions of commutation



// ------------------------------------------------------------------------
// Zero-search procedure (vibration)
// ------------------------------------------------------------------------


#define ALLIGN_READY(element)      prwrk[piwrk[24]+element]
//      Vibration: Zero-search ready
#define Iref(element)              prwrk[piwrk[25]+element]
//      Vibration: Reference current for the alligning procedure
#define DPHI(element)              prwrk[piwrk[26]+element]
//      Vibration: Maximal Step size of the angle change
#define DISC_STEP(element)         prwrk[piwrk[27]+element]
//      Vibration: To count the discrete steps

#define AMPLITUDE_COUNT(element)   prwrk[piwrk[28]+element]
//      Vibration: Max numbers of changing the angle without changing amplitude

#define RESULT(element)            prwrk[piwrk[29]+element]
//      Vibration: Total movement per cicle
#define PREVIOUS_RESULT(element)   prwrk[piwrk[30]+element]
//      Vibration: Previous value of RESULT()
#define temp0(element)             prwrk[piwrk[31]+element]
//      Vibration: Memory for displacement after a pulse
#define temp1(element)             prwrk[piwrk[32]+element]
//      Vibration: Memory for displacement after a pulse
#define temp2(element)             prwrk[piwrk[33]+element]
//      Vibration: Memory for displacement after a pulse
#define temp3(element)             prwrk[piwrk[34]+element]
//      Vibration: Memory for displacement after a pulse
#define temp4(element)             prwrk[piwrk[35]+element]
//      Vibration: Memory for displacement after a pulse

#define TEST_COUNT(element)        prwrk[piwrk[36]+element]
//      Move Test: The number of how many test cicles should succeeed
#define TEST_FAULT(element)        prwrk[piwrk[37]+element]
//      Move Test: No detections of movement



// ------------------------------------------------------------------------
```

```
// Homing Procedure
// ------------------------------------------------------------------------------


#define Epd_Marker(element)        prwrk[piwrk[38]+element]
//      Temporary storage to store position of homing point
//      (Only needed until encoder is reset)



// ------------------------------------------------------------------------------
// Safety-Layer
// ------------------------------------------------------------------------------


#define ZERO_MAX_DRIFT             5.0e-4
//      Maximum drift during zero-search (bigger -> stop procedure)

#define maxout(element)            prwrk[piwrk[39]+element]
//      Safety Layer: maximum allowed current (see HD_V4_Motor.c)

#define p_airbag(element)          prwrk[piwrk[40]+element]
//      Safety Layer: PD-action (Airbag)
#define d_airbag(element)          prwrk[piwrk[41]+element]
//      Safety Layer: PD-action (Airbag)

#define maxpos(element)            prwrk[piwrk[42]+element]
//      Safety Layer: maximum allowed position
#define minpos(element)            prwrk[piwrk[43]+element]
//      Safety Layer: minimum allowed position
#define margin(element)            prwrk[piwrk[44]+element]
//      Safety Layer: extra safety margins (for user)

#define p_vel_brake(element)       prwrk[piwrk[45]+element]
//      Safety Layer: gain for braking at overspeed
#define maxspeed(element)          prwrk[piwrk[46]+element]
//      Safety Layer: maximum allowed speed

#define maxangle                   prwrk[piwrk[47]]
//      afety Layer: maximum angle of y-axis (Delta (Y2-Y1) in [m])

#define pos_violation(element)     prwrk[piwrk[48]+element]
//      Safety Layer: 0=OK, 1=position violation of axis
#define vel_violation(element)     prwrk[piwrk[49]+element]
//      Safety Layer: 0=OK, 1=overspeed
#define ang_violation              prwrk[piwrk[50]]
//      Safety Layer: 0=OK, 1=angle violation (between Y-axes)
#define i_violation(element)       prwrk[piwrk[51]+element]
//      Safety Layer: 0=OK, 1=current too high



// ------------------------------------------------------------------------------
// Headers and includes
// ------------------------------------------------------------------------------


#include "HD_V4_HDrive.h"
#include "HD_V4_Work.c"
#include "HD_V4_Jog.c"
#include "HD_V4_pid.c"
#include "HD_V4_Vapi.c"
```

```
#include "HD_V4_Safety.c"
#include "HD_V4_Motor.c"
#include "HD_V4_Movetest.c"


// ------------------------------------------------------------------------
// COMMENT ON INPUTS AND OUTPUTS
// ------------------------------------------------------------------------

#include "HD_V4_IOPorts.h"



// ------------------------------------------------------------------------
// mdlInitializeSizes()
// ------------------------------------------------------------------------



static void mdlInitializeSizes(SimStruct *S)
{
     /*ssSetNumSFcnParams(S, 0);
     if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) return;*/

     ssSetNumContStates(S, NSTATES);
     ssSetNumDiscStates(S, 0);

     if (!ssSetNumInputPorts(S, 1)) return;
     ssSetInputPortWidth(S, 0, NINPUTS);
     ssSetInputPortDirectFeedThrough(S, 0, 0);

     if (!ssSetNumOutputPorts(S,1)) return;
     ssSetOutputPortWidth(S, 0, NOUTPUTS);

     ssSetNumSampleTimes(S, 1);
     ssSetNumRWork(S, NRWRK);
     ssSetNumIWork(S, NIWRK);
     ssSetNumPWork(S, 0);
     ssSetNumModes(S, 0);
     ssSetNumNonsampledZCs(S, 0);
}



// ------------------------------------------------------------------------
// mdlInitializeSampleTimes()
// ------------------------------------------------------------------------



static void mdlInitializeSampleTimes(SimStruct *S)
{
     ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
     ssSetOffsetTime(S, 0, 0.0);
}
```

```
// ------------------------------------------------------------------------
// mdlInitializeConditions()
// ------------------------------------------------------------------------



#define MDL_INITIALIZE_CONDITIONS


static void mdlInitializeConditions(SimStruct *S)
{
     //double  step_size = ssGetStepSize(S);
     int_T     *piwrk    = ssGetIWork(S);
     real_T    *prwrk    = ssGetRWork(S);
     real_T    *x        = ssGetContStates(S);
     int_T     i;


//   Initialize the global variables in the work space

     rwrk_init_all(S);
     STATUS=(int_T) WAITING_FOR_START;
     for (i=0;i<noa;i++) {
       substate(i)=(int_T) WAITING_FOR_START;
     }

     Hdrive_time=0.0;

     for (i=0;i<noa;i++) {
       us(i)=0.0;
       START_PULSE(i)  =1;
       ALLIGN_READY(i) =0;
     }

//   Aligning Y-axes: speed of aligning [m/sec]
     vyalgn          = 0.03;

//   Homing: Homing speed [m/sec]
     vh(0)           = +0.1;    // X-axis
     vh(1)           = -0.1;    // Y1-axis
     vh(2)           = -0.1;    // Y2-axis

//   Moving: Position after initializing
     target(0)       = -0.3;    // X-axis
     target(1)       = +0.5;    // Y1-axis
     target(2)       = +0.5;    // Y2-axis
//   Moving: PID-Controller for moving to end-position
     p_move(0)       =  1000.0; // X-axis
     i_move(0)       =  100;    //
     d_move(0)       =  100;    //
     p_move(1)       =  1000.0; // Y1-axis
     i_move(1)       =  100;    //
     d_move(1)       =  100;    //
     p_move(2)       =  1000.0; // Y2-axis
     i_move(2)       =  100;    //
     d_move(2)       =  100;    //
```

```
//   Safety Layer: airbag boundaries
     maxpos(0)      = +0.05;   // X-axis
     minpos(0)      = -0.60;   //
     margin(0)      =  0.03;   //
     maxpos(1)      = +1.05;   // Y1-axis
     minpos(1)      = -0.05;   //
     margin(1)      =  0.05;   //
     maxpos(2)      = +1.05;   // Y2-axis
     minpos(2)      = -0.05;   //
     margin(2)      =  0.05;   //
//   Safety Layer: PD-Controller for airbag
     p_airbag(0)    =  100.0;  // X-axis
     d_airbag(0)    =  25;     //
     p_airbag(1)    =  100.0;  // Y1-axis
     d_airbag(1)    =  25;     //
     p_airbag(2)    =  100.0;  // Y2-axis
     d_airbag(2)    =  25;     //
//   Safety Layer: Maximum speed [m/sec]
     maxspeed(0)    =  1.0;    // X-axis
     maxspeed(1)    =  1.0;    // Y1-axis
     maxspeed(2)    =  1.0;    // Y2-axis
//   Safety Layer: P-Controller for velocity brake
     p_vel_brake(0) =  25;     // X-axis
     p_vel_brake(1) =  25;     // Y1-axis
     p_vel_brake(2) =  25;     // Y2-axis
//   Safety Layer: Maximum difference in position between Y1 and Y2 axis
     maxangle       = 0.020;   //
//   Safety Layer: Clipping parameters
//   See HD_V4_Motor.c

//   Overall: Direction of commutation
     commdir(0)     = +1.0;    // X-axis
     commdir(1)     = -1.0;    // Y1-axis
     commdir(2)     = -1.0;    // Y2-axis


//   initialize Hdrive
     Hdrive_initialize(&Hdrive_pos(0),S);

//   controller initial condition
     pid_ini(x,&Hdrive_pos(0));
}



// ------------------------------------------------------------------------
// mdlOutputs()
// ------------------------------------------------------------------------



static void mdlOutputs(SimStruct *S, int_T tid)
{
     real_T             *x     = ssGetContStates(S);
     real_T             *yPtrs = ssGetOutputPortRealSignal(S,0);
     InputRealPtrsType  uPtrs = ssGetInputPortRealSignalPtrs(S,0);

     int_T     *piwrk    = ssGetIWork(S);
     real_T    *prwrk    = ssGetRWork(S);
```

```
    int_T     istat,i,statout,all_vel_zero;
    double    time;


//  current time
    time=ssGetT(S);

//  current position
    for (i=0;i<noa;i++) {
      Hdrive_pos(i)=IN_POS(i)-pos_reset(0+i);
      OUT_SUBSTATE(i)=substate(i);
    }

//  Calculate output-status variable
//  READY->status per axis, combined in one number
    if ((int)SHOWCOMBINED==1){
       statout=0;
       for(i=0;i<noa;i++) {
         statout=10*statout+substate(i);
       }
    }
    else{
       statout=STATUS;
    }
    OUT_STATUS=statout;


//  +-------------------------------------------------+
//  | ***              EMERGENCY STOP?              *** |
//  +-------------------------------------------------+



    if ((int)IN_EMERGENCY==1) {
       istat=(int)STATUS;
//     Zero_Search: Unable to control system -> switch off
//     current
       switch (istat) {

              case TEST:
              case ZERO_SEARCH:
                   STATUS=EMERGENCY_STOP_SYSTEM;
                   OUT_POWER=0;
              break;

              default:
                   STATUS=EMERGENCY_STOP_SYSTEM;
              break;
       }
    }



    istat=(int)STATUS;
    switch (istat) {

//  +-------------------------------------------------+
//  | *** STILL INITILALIZING: SYNCHRONISED CONRTOL *** |
//  +-------------------------------------------------+
```

```
    case WAITING_FOR_START:

        for (i=0;i<noa;i++) {

          AMPLITUDE_COUNT(i)=  0;
          DISC_STEP(i)      =  0;

          TEST_COUNT(i)     =  0;
          TEST_FAULT(i)     =  0;

          DPHI(i)           =  0.25*pi;
          PHI(i)            =  0;
          RESULT(i)         =  0;
          PREVIOUS_RESULT(i)=  0;
          Iref(i)           =  0.2;
          temp0(i)          =  0;
          temp1(i)          =  0;
          temp2(i)          =  0;
          temp3(i)          =  0;
          temp4(i)          =  0;
        }
        triggertime         =  0;
        SHOWCOMBINED        =  0;

//      switch off motor commands
        for (i=0;i<noa;i++) {
          us(i)=0.0;
          START_PULSE(i)  =1;
          ALLIGN_READY(i) =0;
        }

//      zeroise outputs except for status out and enable signal (end of etroke)
        for (i=0;i<NOUTPUTS;i++) {
          yPtrs[i]=0.0;
        }
        OUT_POWER=1;      // Enable power supply (end of stroke)
        OUT_STATUS=(real_T) WAITING_FOR_START;
        for(i=0;i<noa;i++) {
          substate(i)=(int_T) WAITING_FOR_START;
        }
        Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),S);
    break;


    case TEST:
        /* This case test if there is enough movement before starting the
        zero-search When there is not enough movement the Reference Amperes
        will be scaled up and the angle is increased with pi/2. The movement
        should be higher then the DETECTION_LEVEL, this should be tested
        three times. After this the zero-search can be started */

        Hdrive_movetest_out(&us(0),&Hdrive_pos(0),S);
        Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),S);

        for (i=0;i<noa;i++) {
          OUT_POS(i)=SHOWPOS*Hdrive_pos(i); // position not known yet
          OUT_VEL(i)=SHOWPOS*Hdrive_vel(i); // yields zero at this stage (?)
        }
    break;


    case ZERO_SEARCH:
```

```
        /* The zero search case is a iterative way of finding the zero point
        of the motor.At this zero angle the motor will not move at any current
      . The drive should move more than the detection level. */

        Hdrive_zerosearch_out(&us(0),&Hdrive_pos(0), S);
        Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),S);

        for (i=0;i<noa;i++) {
          OUT_POS(i)=SHOWPOS*Hdrive_pos(i); // position not known yet
          OUT_VEL(i)=SHOWPOS*Hdrive_vel(i); // yields zero at this stage (?)
        }
    break;

    case Y_ALIGN:
        Hdrive_yalign_out(&us(0),&Hdrive_vel(0),x,&Hdrive_pos(0),time,
          YALGN_STATE,S);
        Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),S);
        for (i=0;i<noa;i++) {
          OUT_POS(i)=SHOWPOS*Hdrive_pos(i); // position not known yet
          OUT_VEL(i)=SHOWPOS*Hdrive_vel(i); // yields zero at this stage (?)
        }
    break;


    case HOMING:
        Hdrive_homing_out(&us(0),&Hdrive_vel(0),x,&Hdrive_pos(0),time,S);
        Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),S);

        for (i=0;i<noa;i++) {
          OUT_POS(i)=SHOWPOS*Hdrive_pos(i); // position not known yet
          OUT_VEL(i)=SHOWPOS*Hdrive_vel(i); // yields zero at this stage (?)
        }
    break;


    case MOVING:
        Hdrive_moving_out(&us(0),&Hdrive_vel(0),x,&Hdrive_pos(0),time,S);
        Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),S);

        for (i=0;i<noa;i++) {
          OUT_POS(i)=SHOWPOS*Hdrive_pos(i);
          OUT_VEL(i)=SHOWPOS*Hdrive_vel(i); // yields zero at this stage (?)
        }
    break;


/*  -> Doesn't occur during initialization stages
    case POS_VIOLATION:
        // to get velocity
        pid_out(&us(0),&Hdrive_vel(0),x,&Hdrive_pos(0),&Hdrive_pos(0),
          &Hdrive_vel(0),S);
        Hdrive_result=(real_T) airbag(&us(0),&Hdrive_pos(0),&Hdrive_vel(0),
          1,S);
        // outputs
        for (i=0;i<noa;i++) {
          OUT_POS(i)=Hdrive_pos(i);
          OUT_VEL(i)=Hdrive_vel(i);
        }
        Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),S);
    break;
*/
```

```
//   case I_VIOLATION:
//   break;


     case END_OF_STROKE:
          // outputs
          for (i=0;i<noa;i++) {
            OUT_POS(i)=SHOWPOS*Hdrive_pos(i);
            OUT_VEL(i)=SHOWPOS*Hdrive_vel(i);//geeft elke keer nul
          }

          OUT_POWER=0;  // Power off
     break;



     case ALLIGNING_FAILD:
// -> New code:
          // to get velocity
          pid_out(&us(0),&Hdrive_vel(0),x,&Hdrive_pos(0),&Hdrive_pos(0),
            &Hdrive_vel(0),S);

          for (i=0;i<noa;i++) {
            OUT_POS(i)=SHOWPOS*Hdrive_pos(i);
            OUT_VEL(i)=SHOWPOS*Hdrive_vel(i);
            us(i)=0.0;  // No current
          }
          Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),
            S);          // set current to 0
          OUT_POWER=0;  // Turn Power off

/* ->Old code: just a brake action:
          // to get velocity
          pid_out(&us(0),&Hdrive_vel(0),x,&Hdrive_pos(0),&Hdrive_pos(0),
            &Hdrive_vel(0),S);
          for (i=0;i<noa;i++) {
            OUT_POS(i)=SHOWPOS*Hdrive_pos(i);
            OUT_VEL(i)=SHOWPOS*Hdrive_vel(i);
          }
          for (i=0;i<noa;i++) {
            us(i)=-2*Hdrive_vel(i);
          }
          Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),S);
*/
     break;


// This state is only used for overspeed during alignment

     case VEL_VIOLATION:
          for (i=0;i<noa;i++) {
            OUT_POS(i)=SHOWPOS*Hdrive_pos(i);
            OUT_VEL(i)=SHOWPOS*Hdrive_vel(i);
          }

          pid_out(&us(0),&Hdrive_vel(0),x,&Hdrive_pos(0),&Hdrive_pos(0),
            &Hdrive_vel(0),S);//to get velocity

          for (i=0;i<noa;i++) {
            us(i)=-2*Hdrive_vel(i);
          }
```

```
        Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),S);
    break;


//  +-------------------------------------------------+
//  | ***      OPERATING: CONRTOL PER AXIS          *** |
//  +-------------------------------------------------+

    case READY:

        // triggering time for feed-forward
        if (IN_START_ALGN==1 && triggertime==0){
          Hdrive_time=time;
          triggertime=1;
        }

        if (triggertime==1) {
          OUT_HTIME=time-Hdrive_time;
        }
        else {
          OUT_HTIME=time;
        }

        // to get velocity
        pid_out(&us(0),&Hdrive_vel(0),x,&Hdrive_pos(0),&Hdrive_pos(0),
          &Hdrive_vel(0),S);


        // *** SAFETY CHECK PER AXIS

        for (i=0;i<noa;i++) {

          // ---> pass user inputs
          us(i)=U(0+i);

          // ---> Check for possible violations
          safety_check(&us(0),&Hdrive_pos(0),&Hdrive_vel(0),S);

          // ---> End of stroke

          // Checked in mdlUpdate(), special state gets activated

          // ---> Position Violation

          if ((int_T)pos_violation(i)==1) {
            // Airbag action
            safety_pos_airbag(&us(0),&Hdrive_pos(0),&Hdrive_vel(0),i,S);
          }

          // ---> Velocity Violation

          else if ((int_T)vel_violation(i)==1) {
             // brake
             safety_vel_airbag(&us(0),&Hdrive_pos(0),&Hdrive_vel(0),i,S);
          }

        }

        // *** GLOBAL SAFETY-CHECK

        // ---> Angle Violation
```

```
        if ((int_T)ang_violation==1) {
          safety_angle_viol_ini(time,&Hdrive_pos(0), S);
          safety_angle_viol(&us(0),time,&Hdrive_pos(0),&Hdrive_vel(0),x,S);
        }

        // ---> outputs
        for (i=0;i<noa;i++) {
          OUT_POS(i)=Hdrive_pos(i);
          OUT_VEL(i)=Hdrive_vel(i);
        }

        Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),S);
    break;


    case ANG_VIOLATION:
         // to get velocity
        pid_out(&us(0),&Hdrive_vel(0),x,&Hdrive_pos(0),&Hdrive_pos(0),
          &Hdrive_vel(0),S);

        // Brake axes and pull Y2 towards Y1
        safety_angle_viol(&us(0),time,&Hdrive_pos(0),&Hdrive_vel(0),x,S);

        for (i=0;i<noa;i++) {
          OUT_POS(i)=SHOWPOS*Hdrive_pos(i);
          OUT_VEL(i)=SHOWPOS*Hdrive_vel(i);
        }
        Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),
          S);          // set current

        //OUT_POWER=0;  // Turn Power off
    break;


//  +-------------------------------------------------+
//  | ***              EMERGENCY STOP?              *** |
//  +-------------------------------------------------+


    case EMERGENCY_STOP_SYSTEM:
        all_vel_zero=1;
        // to get velocity
        pid_out(&us(0),&Hdrive_vel(0),x,&Hdrive_pos(0),&Hdrive_pos(0),
          &Hdrive_vel(0),S);

        // Calculate outputs
        safety_direct_stop(&us(0),&Hdrive_pos(0),&Hdrive_vel(0),x,S);

        for (i=0;i<noa;i++) {
          // Switch off current when velocity becomes realy small
          if (fabs(Hdrive_vel(i))>0.001) {
             all_vel_zero=0;
          }
        }

        // velocity sufficient low to switch off power
        if ((int)all_vel_zero==1) {
           OUT_POWER=0;
        }


        for (i=0;i<noa;i++) {
```

```
            OUT_POS(i)=SHOWPOS*Hdrive_pos(i);
            OUT_VEL(i)=SHOWPOS*Hdrive_vel(i);
          }
          Hdrive_result=(real_T) send_motor_command(&us(0),&Hdrive_pos(0),
            S);          // set current
      break;
    }
}



// ------------------------------------------------------------------------
// mdlUpdate()
// ------------------------------------------------------------------------



#define MDL_UPDATE
#if defined(MDL_UPDATE)


static void mdlUpdate(SimStruct *S, int_T tid)
{
     real_T    *x    = ssGetContStates(S);
     InputRealPtrsType    uPtrs = ssGetInputPortRealSignalPtrs(S,0);

     int_T     *piwrk    = ssGetIWork(S);
     real_T    *prwrk    = ssGetRWork(S);
     int_T     istat,i,count;
     real_T    time;


//   check for stop...
     if (IN_START_ALGN==0) {
//     reinitialize Hdrive
       Hdrive_initialize(&Hdrive_pos(0),S);
       pid_ini(x,&Hdrive_pos(0));
       STATUS=WAITING_FOR_START;
     }

//   current time
     time=ssGetT(S);

     istat=(int_T) STATUS;
     switch (istat) {


//   +--------------------------------------------------+
//   | *** STILL INITILALIZING: SYNCHRONISED CONRTOL *** |
//   +--------------------------------------------------+


     case WAITING_FOR_START:
          if (IN_START_ALGN==1) {
//          start allignment...
            STATUS=(int_T)TEST;
            for (i=0;i<noa;i++) {
              substate(i)=(int_T)TEST;
            }
          }
```

```
break;


case TEST:
     // NB. 3 loops -> State gets the error-value of the error whith the
     // highest priority

     // Check which axis are ready with testing
     count=0;
     for (i=0;i<noa;i++) {
       if ((int_T)TEST_COUNT(i)==3) {
         substate(i)=(int_T) ZERO_SEARCH;
         count++;
       }
     }
     // all axes ready whith testing -> goto stage ZERO_STAGE
     if (count==noa){
       STATUS=(int_T) ZERO_SEARCH;
     }

     // Alignment-procedure failed for one or more axes -> goto state
     // ALLIGNING_FAILD
     for (i=0;i<noa; i++) {
       if ((int_T)TEST_FAULT(i)==1) {
         STATUS=(int_T) ALLIGNING_FAILD;
         substate(i)=(int_T) ALLIGNING_FAILD;
       }
     }

     // One or more eos-sensors activated -> goto state END_OF_STROKE
     for (i=0;i<noa;i++) {
       if ((IN_EOS(i)==1)) {                    //end of stroke
         STATUS=(int_T) END_OF_STROKE;
         substate(i)=(int_T) END_OF_STROKE;
       }
     }
break;


case ZERO_SEARCH:
     // Alignment-procedure failed for one or more axes -> goto state
     // ALLIGNING_FAILD
     for (i=0;i<noa; i++) {
       if (AMPLITUDE_COUNT(i)>=50) {
         STATUS=(int_T) ALLIGNING_FAILD;
         substate(i)=(int_T) ALLIGNING_FAILD;
       }
     }

     // Check which axis are ready with aligning
     count=0;
     for (i=0;i<noa;i++) {
       if (ALLIGN_READY(i)==1) {
         substate(i)=(int_T) HOMING;
         count++;
       }
     }
     // all axes ready whith testing -> goto stage ZERO_STAGE
     if (count==noa){
       // Calculate phase-offset
       for (i=0;i<noa;i++) {
         PHI(i)=PHI(i)+pi/2;
```

```
        }
        // Next stae: align Y2 axis with respect to Y1 acis or go homing?
        if (noa<3) { // Just one axis, no y-alignment needed
          STATUS=(int_T) HOMING;
          Hdrive_start_homing(&Hdrive_pos(0),time,S);
        }
        else {
          STATUS=(int_T) Y_ALIGN;
          YALGN_STATE=(int_T) YALGN_FIND_AVS_B;
          Hdrive_yalign_restart(&Hdrive_pos(0),time,YALGN_STATE,S);
        }
      }

      // Check if axis have drifted to far away from starting point
      for (i=0;i<noa;i++) {
        if (fabs(Hdrive_pos(i))>ZERO_MAX_DRIFT) {
          substate(i)=(int_T) ALLIGNING_FAILD;
          STATUS=(int_T) ALLIGNING_FAILD;
        }
      }

      // One or more eos-sensors activated -> goto state END_OF_STROKE
      for (i=0;i<noa;i++) {
        if ((IN_EOS(i)==1)) {                    // end of stroke
          STATUS=(int_T) END_OF_STROKE;
          substate(i)=(int_T) END_OF_STROKE;
        }
      }
break;


case ALLIGNING_FAILD:
      // One or more eos-sensors activated -> goto state END_OF_STROKE
      for (i=0;i<noa;i++) {
        if ((IN_EOS(i)==1)) {                    // end of stroke
          STATUS=(int_T) END_OF_STROKE;
          substate(i)=(int_T) END_OF_STROKE;
        }
      }
break;


case Y_ALIGN:

      // Still busy with finding Angle_Violation_A Sensor?
      if (YALGN_STATE==YALGN_FIND_AVS_B) {
        if(Hdrive_yalign_synchronise(time,YALGN_FIND_AVS_B,S)){
          YALGN_STATE=(int_T) YALGN_FIND_AVS_A;
          Hdrive_yalign_restart(&Hdrive_pos(0),time,YALGN_FIND_AVS_A,S);
        }
      }

      // *** Warning: In the current setup of the H-Drive one of the
      // *** Tilt-sensors seems to be missing. The assumption that
      // *** sensor AVS_A detects tilt in one direction, and B detects
      // *** tilt in the other direction isn't correct.
      // *** Sensor AVS_B seems to detect tilt, Sensor AVS_A gets
      // *** activated when the X-axis is perpendicular to the
      // *** Y-axis.
      // *** The following define makes it possible to switch between
      // *** the possibility to (1) set the Y-axis in such a position
      // *** that is in the middle between the position where the avs-
```

```
          // *** sensors get activated or (2) set the y-axis in the position
          // *** where AVS_B gets just activated

#ifdef    ALIGN_Y_TO_CENTRE

          // Still busy with finding Angle_Violation_B Sensor?
          if (YALGN_STATE==YALGN_FIND_AVS_A){
            if(Hdrive_yalign_synchronise(time,YALGN_FIND_AVS_A,S)){
              YALGN_STATE=(int_T) YALGN_CENTRE;
              Hdrive_yalign_restart(&Hdrive_pos(0),time,YALGN_CENTRE,S);
            }
          }

          // Still busy with moving to centre?
          if (YALGN_STATE==YALGN_CENTRE){
            if(Hdrive_yalign_synchronise(time,YALGN_CENTRE,S)){
              YALGN_STATE=(int_T) READY;
              STATUS=(int_T) HOMING;
              for(i=0;i<noa;i++) {
                substate(i)=(int_T) HOMING;
              }
              Hdrive_start_homing(&Hdrive_pos(0),time,S);
            }
          }

#else

          // Still busy with finding Angle_Violation_B Sensor?
          if (YALGN_STATE==YALGN_FIND_AVS_A){
            if(Hdrive_yalign_synchronise(time,YALGN_FIND_AVS_A,S)){
              YALGN_STATE=(int_T) READY;
              STATUS=(int_T) HOMING;
              for(i=0;i<noa;i++) {
                substate(i)=(int_T) HOMING;
              }
              Hdrive_start_homing(&Hdrive_pos(0),time,S);
            }
          }

#endif

          // One or more eos-sensors activated -> goto state END_OF_STROKE
          for (i=0;i<noa;i++) {
            if ((IN_EOS(i)==1)) {                    // end of stroke
              STATUS=(int_T) END_OF_STROKE;
              substate(i)=(int_T) END_OF_STROKE;
            }
          }

     break;


     case HOMING:
          // Not all axis ready with stage Hdrive_HOME1 ?
          if (HOMING1_ALL_READY==0){
            if (Hdrive_synchronize(&Hdrive_pos(0),&target(0),time,HdriveHOME1,
              S)) {
              // All axis ready with stage HdriveHOME1
              for (i=0;i<noa;i++) {
                vh(i)=-vh(i)/3;
              }
              Hdrive_start_homing(&Hdrive_pos(0),time,S);
```

```
        }
      }

      // All axis ready with stage HDriveHOME1 ?
      if (HOMING1_ALL_READY==1){
        if(Hdrive_synchronize(&Hdrive_pos(0),&target(0),time,HdriveHOME2,
          S)) {
          // All axis ready with stage HDriveHOME2
          for (i=0;i<noa;i++) {
            substate(i)=(int_T) MOVING;
            vh(i)=-vh(i)*3;
          }
          STATUS=MOVING;
          Hdrive_start_moving(&Hdrive_pos(0),time,S);
        }
      }


      // One or more eos-sensors activated -> goto state END_OF_STROKE
      for (i=0;i<noa;i++) {
        if ((IN_EOS(i)==1)) {                    // end of stroke
          STATUS=(int_T) END_OF_STROKE;
          substate(i)=(int_T) END_OF_STROKE;
        }
      }
    break;


    case MOVING:
      // All axes ready with stage HdriveMOVE ?
      if (Hdrive_synchronize(&Hdrive_pos(0),&target(0),time,HdriveMOVE,
        S)) {
        for (i=0;i<noa;i++) {
          substate(i)=(int_T) READY;
        }
        STATUS=READY;
        SHOWCOMBINED=1;
      }

      // One or more eos-sensors activated -> goto state END_OF_STROKE
      for (i=0;i<noa;i++) {
        if ((IN_EOS(i)==1)) {                    // end of stroke
          STATUS=(int_T) END_OF_STROKE;
          substate(i)=(int_T) END_OF_STROKE;
        }
      }
    break;


    case VEL_VIOLATION:
      // One or more eos-sensors activated -> goto state END_OF_STROKE
      for (i=0;i<noa;i++) {
        if ((IN_EOS(i)==1)) {                    //end of stroke
          STATUS=(int_T) END_OF_STROKE;
          substate(i)=(int_T) END_OF_STROKE;
        }
      }
    break;

/*  Doesn't occur during initiazing states
    case POS_VIOLATION:
      // One or more eos-sensors activated -> goto state END_OF_STROKE
```

```
          for (i=0;i<noa;i++) {
            if ((IN_EOS(i)==1)) {                    // end of stroke
              STATUS=(int_T) END_OF_STROKE;
              substate(i)=(int_T) END_OF_STROKE;
            }
          }
      break;

*/



//   +--------------------------------------------------+
//   | ***      OPERATING: CONRTOL PER AXIS          *** |
//   +--------------------------------------------------+


      case READY:

          // One or more eos-sensors activated -> goto state END_OF_STROKE
          for (i=0;i<noa;i++) {
            if ((IN_EOS(i)==1)) {                    // end of stroke
              STATUS=(int_T) END_OF_STROKE;
              substate(i)=(int_T) END_OF_STROKE;
            }
          }

          if ((int_T)ang_violation==1) {
            STATUS=(int_T) ANG_VIOLATION;
          }

         // Substates are determined in mdlOutputs()
         // if (fabs(Hdrive_vel(0))>2.1)       // only for one axis
         //   STATUS=(int_T) OVERSPEED;        // overspeed protection > 2,1m/s


      break;


      case ANG_VIOLATION:
      break;


      case END_OF_STROKE:
      break;

      case EMERGENCY_STOP_SYSTEM:
      break;

     }
}


#endif



// ------------------------------------------------------------------------
// mdlDerivatives
// ------------------------------------------------------------------------
```

```
#define MDL_DERIVATIVES
#if defined(MDL_DERIVATIVES)


static void mdlDerivatives(SimStruct *S)
{
     real_T    *dx  = ssGetdX(S);
     real_T    *x   = ssGetContStates(S);
     int_T     *piwrk = ssGetIWork(S);
     real_T    *prwrk = ssGetRWork(S);
     int_T     i,istat;
     real_T    time;


//   current time
     time=ssGetT(S);

     istat=(int_T) STATUS;
     switch (istat) {


     case WAITING_FOR_START:
          for (i=0;i<NSTATES;i++) {
            dx[i]=0.0;
          }
     break;


     case TEST:
     break;


     case ZERO_SEARCH:
     break;

     case Y_ALIGN:
          Hdrive_yalign_dif(dx,x,&Hdrive_pos(0),time,YALGN_STATE,S);
     break;

     case HOMING:
          Hdrive_homing_dif(dx,x,&Hdrive_pos(0),time,S);
     break;


     case MOVING:
          Hdrive_moving_dif(dx,x,&Hdrive_pos(0),time,S);
     break;


     case READY:
          Hdrive_ready_dif(dx,x,&Hdrive_pos(0),time);
     break;

     case ALLIGNING_FAILD:
          Hdrive_violation_dif(dx,x,&Hdrive_pos(0),time);
     break;
```

```
     case VEL_VIOLATION:
          Hdrive_violation_dif(dx,x,&Hdrive_pos(0),time);
     break;


     case POS_VIOLATION:
          Hdrive_violation_dif(dx,x,&Hdrive_pos(0),time);
     break;

     case ANG_VIOLATION:
          Hdrive_violation_dif(dx,x,&Hdrive_pos(0),time);
     break;

     case EMERGENCY_STOP_SYSTEM:
          Hdrive_violation_dif(dx,x,&Hdrive_pos(0),time);
     break;

     }
}


#endif


// ---------------------------------------------------------------------------
// mdlTerminate()
// ---------------------------------------------------------------------------



static void mdlTerminate(SimStruct *S)
{
}



// ---------------------------------------------------------------------------
// Trailer code
// ---------------------------------------------------------------------------



#ifdef  MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfun.h"
#endif
```

B.4.2 HD_V4_HDrive.h

```
// ------------------------------------------------------------------------
// status values
// ------------------------------------------------------------------------


// During initialization

#define WAITING_FOR_START          8
#define TEST                       11
#define ZERO_SEARCH                12
#define Y_ALIGN                    13
#define HOMING                     14
#define MOVING                     15

#define ALLIGNING_FAILD            7

// During operation:

#define READY                      0
#define END_OF_STROKE              2

#define POS_VIOLATION              3
#define VEL_VIOLATION              4
#define I_VIOLATION                5

#define ANG_VIOLATION              6    // Angle of y-axis too big

// Stop System:

#define EMERGENCY_STOP_SYSTEM      17


// ------------------------------------------------------------------------
// synchronize values (homing)
// ------------------------------------------------------------------------


#define HdriveHOME1                10
#define HdriveHOME2                11
#define HdriveMOVE                 12


// ------------------------------------------------------------------------
// synchronize values (aligning y-axis to set angle to zero)
// ------------------------------------------------------------------------


#define YALGN_FIND_AVS_B           10
#define YALGN_FIND_AVS_A           11
#define YALGN_CENTRE               12


// ------------------------------------------------------------------------
// clipping values (OBSOLETE)
// ------------------------------------------------------------------------


// #define X_CLIP                 -1.0
// #define Y1_CLIP                -1.0
// #define Y2_CLIP                -1.0
```

B.4.3 HD_V4_IOPorts.h

```
// ------------------------------------------------------------------------
// HDrive I/O-Ports
// ------------------------------------------------------------------------



#define U(element)    (*uPtrs[element])
#define Y(element)    (yPtrs[element])



// Number of Axis (noa):
//
//   0: X-axis
//   1: Y1-axis
//   2: Y2-axis
//
// Input Signals
//
//   U(0)....U(2)   :    Users controller input.... currents [A]
//   U(3)           :    start signal allignment
//   U(4)....U(6)   :    end of stroke sensor 1: switch on
//   U(7)....U(9)   :    Home sensor (EPD), 1: home found
//   U(10)...U(11)  ;    Angle violation between Y-axes
//   U(12)...U(14)  :    phi offset, default 0
//   U(15)...U(17)  :    position of the Hdrive for the three axis[m]
//   U(18)          :    trigger feedforward signal/ Hdrive time
//   U(19)          :    Emergency brake
//
// Output channels
//
//   y[0]...y[2]     :   amplitude current for the three axis [A]
//   y[3]...y[5]     :   Angle PHI for the three axis
//   y[6]            :   Enable power supply
//   y[7]...y[9]     :   Reset Encoders
//
//   y[10]           :   Status
//   y[11]...y[13]   :   Substates
//
//   y[14]           :   Enable controller after moving
//   y[15]           :   Hdrive time after Moving
//
//   y[16]...y[18]   :   After Moving for the three axis
//   y[19]...y[21]   :   velocity for the three axis
//
//   y[22]...y[24]   :   Result
//   y[25]...y[27]   :   DPHI
//   y[28]...y[30]   :   Controller output for homing/moving
//   y[31]...y[33]   :   Controller output for homing/moving


                                                //
#define IN_I(element)          U(0+element)     //   U(0)....U(2)
#define IN_START_ALGN          U(3)             //   U(3)
#define IN_EOS(element)        U(4+element)     //   U(4)....U(6)
#define IN_EPD(element)        U(7+element)     //   U(7)....U(9)
#define IN_AVS(element)        U(10+element)    //   U(10)...U(11)
#define IN_PHI_OFST(element)   U(12+element)    //   U(12)...U(14)
#define IN_POS(element)        U(15+element)    //   U(15)...U(17)
#define IN_TRIG                U(18)            //   U(18)
```

```
#define IN_EMERGENCY          U(19)           //   U(19)
                                              //
                                              // Output channels
                                              //
#define OUT_I(element)        Y(0+element)    //   y[0]...y[2]
#define OUT_PHI(element)      Y(3+element)    //   y[3]...y[5]
#define OUT_POWER             Y(6)            //   y[6]
#define OUT_RST_ENC(element)  Y(7+element)    //   y[7]...y[9]
                                              //
#define OUT_STATUS            Y(10)           //   y[10]
#define OUT_SUBSTATE(element) Y(11+element)   //   y[11]...y[13]
                                              //
#define OUT_CTRL_EN           Y(14)           //   y[14]
#define OUT_HTIME             Y(15)           //   y[15]
                                              //
#define OUT_POS(element)      Y(16+element)   //   y[16]...y[18]
#define OUT_VEL(element)      Y(19+element)   //   y[19]...y[21]
                                              //
#define OUT_RESULT(element)   Y(22+element)   //   y[22]...y[24]
#define OUT_DPHI(element)     Y(25+element)   //   y[25]...y[27]
#define OUT_CTRL_OUT(element) Y(28+element)   //   y[28]...y[30]
#define OUT_CTRL_SP(element)  Y(31+element)   //   y[31]...y[33]
```

B.4.4 HD_V4_Work.c

```
// init globals in work-space for re-entrancy

#define JOG_IDX 52


// ------------------------------------------------------------------------
// rwrk_init_var()
// ------------------------------------------------------------------------




int rwrk_init_var(int *pivar, int *pidx, int nrw, SimStruct *S)
{
     int_T     *piwrk = ssGetIWork(S);

     piwrk[pivar[0]] = pidx[0];
     pivar[0]++;
     pidx[0] = pidx[0]+nrw;

     return 1;
}



// ------------------------------------------------------------------------
// rwek_init_all()
// ------------------------------------------------------------------------



int rwrk_init_all(SimStruct *S)
{
     int  ivar,idx;

     ivar = 0;
     idx = 0;


     // ************************************************************************
     // * HD_V4_Hdrive.c                                                       *
     // ************************************************************************



     // ------------------------------------------------------------------------
     // Control
     // ------------------------------------------------------------------------


     rwrk_init_var(&ivar,&idx,3,S);    // START_PULSE(element)

     rwrk_init_var(&ivar,&idx,1,S);    // STATUS
     rwrk_init_var(&ivar,&idx,3,S);    // substate(element)
     rwrk_init_var(&ivar,&idx,1,S);    // SHOWCOMBINED

     rwrk_init_var(&ivar,&idx,1,S);    // YALGN_STATE
```

```
rwrk_init_var(&ivar,&idx,3,S);    // HOMING1_READY(element)
rwrk_init_var(&ivar,&idx,1,S);    // HOMING1_ALL_READY

rwrk_init_var(&ivar,&idx,3,S);    // Hdrive_pos(element)
rwrk_init_var(&ivar,&idx,3,S);    // Hdrive_vel(element)

rwrk_init_var(&ivar,&idx,3,S);    // us(element)
rwrk_init_var(&ivar,&idx,3,S);    // PHI(element)
rwrk_init_var(&ivar,&idx,1,S);    // Hdrive_result

rwrk_init_var(&ivar,&idx,1,S);    // Hdrive_time

rwrk_init_var(&ivar,&idx,3,S);    // pos_reset(element)
rwrk_init_var(&ivar,&idx,3,S);    // posreset_yalgn(element)

rwrk_init_var(&ivar,&idx,3,S);    // epd(element)

rwrk_init_var(&ivar,&idx,3,S);    // p_move(element)
rwrk_init_var(&ivar,&idx,3,S);    // d_move(element)
rwrk_init_var(&ivar,&idx,3,S);    // i_move(element)

rwrk_init_var(&ivar,&idx,1,S);    // triggertime

rwrk_init_var(&ivar,&idx,1,S);    // vyalgn
rwrk_init_var(&ivar,&idx,3,S);    // vh(element)
rwrk_init_var(&ivar,&idx,3,S);    // target(element)
rwrk_init_var(&ivar,&idx,3,S);    // commdir(element)


// ------------------------------------------------------------------------
// Zero-search procedure (vibration)
// ------------------------------------------------------------------------


rwrk_init_var(&ivar,&idx,3,S);    // ALLIGN_READY(element)
rwrk_init_var(&ivar,&idx,3,S);    // Iref(element)
rwrk_init_var(&ivar,&idx,3,S);    // DPHI(element)
rwrk_init_var(&ivar,&idx,3,S);    // DISC_STEP(element)

rwrk_init_var(&ivar,&idx,3,S);    // AMPLITUDE_COUNT(element)

rwrk_init_var(&ivar,&idx,3,S);    // RESULT(element)
rwrk_init_var(&ivar,&idx,3,S);    // PREVIOUS_RESULT(element)
rwrk_init_var(&ivar,&idx,3,S);    // temp0(element)
rwrk_init_var(&ivar,&idx,3,S);    // temp1(element)
rwrk_init_var(&ivar,&idx,3,S);    // temp2(element)
rwrk_init_var(&ivar,&idx,3,S);    // temp3(element)
rwrk_init_var(&ivar,&idx,3,S);    // temp4(element)

rwrk_init_var(&ivar,&idx,3,S);    // TEST_COUNT(element)
rwrk_init_var(&ivar,&idx,3,S);    // TEST_FAULT(element)


// ------------------------------------------------------------------------
// Homing Procedure
// ------------------------------------------------------------------------


rwrk_init_var(&ivar,&idx,3,S);    // Epd_Marker(element)
```

```
// ------------------------------------------------------------------------
// Safety-Layer
// ------------------------------------------------------------------------


//                                  // ZERO_MAX_DRIFT

rwrk_init_var(&ivar,&idx,3,S);      // maxout(element)

rwrk_init_var(&ivar,&idx,3,S);      // p_airbag(element)
rwrk_init_var(&ivar,&idx,3,S);      // d_airbag(element)

rwrk_init_var(&ivar,&idx,3,S);      // maxpos(element)
rwrk_init_var(&ivar,&idx,3,S);      // minpos(element)
rwrk_init_var(&ivar,&idx,3,S);      // margin(element)

rwrk_init_var(&ivar,&idx,3,S);      // p_vel_brake(element)
rwrk_init_var(&ivar,&idx,3,S);      // maxspeed(element)

rwrk_init_var(&ivar,&idx,1,S);      // maxangle

rwrk_init_var(&ivar,&idx,3,S);      // pos_violation(element)
rwrk_init_var(&ivar,&idx,3,S);      // vel_violation(element)
rwrk_init_var(&ivar,&idx,1,S);      // ang_violation
rwrk_init_var(&ivar,&idx,3,S);      // i_violation(element)


// ************************************************************************
// * HD_V4_Jog.c                                                          *
// ************************************************************************


rwrk_init_var(&ivar,&idx,3,S);      // t0(element)
rwrk_init_var(&ivar,&idx,3,S);      // t1(element)
rwrk_init_var(&ivar,&idx,3,S);      // t2(element)
rwrk_init_var(&ivar,&idx,3,S);      // t3(element)
rwrk_init_var(&ivar,&idx,3,S);      // t4(element)
rwrk_init_var(&ivar,&idx,3,S);      // t5(element)
rwrk_init_var(&ivar,&idx,3,S);      // t6(element)
rwrk_init_var(&ivar,&idx,3,S);      // t7(element)
rwrk_init_var(&ivar,&idx,3,S);      // s0(element)
rwrk_init_var(&ivar,&idx,3,S);      // s1(element)
rwrk_init_var(&ivar,&idx,3,S);      // s2(element)
rwrk_init_var(&ivar,&idx,3,S);      // s3(element)
rwrk_init_var(&ivar,&idx,3,S);      // s4(element)
rwrk_init_var(&ivar,&idx,3,S);      // s5(element)
rwrk_init_var(&ivar,&idx,3,S);      // s6(element)
rwrk_init_var(&ivar,&idx,3,S);      // s7(element)
rwrk_init_var(&ivar,&idx,3,S);      // v0(element)
rwrk_init_var(&ivar,&idx,3,S);      // v1(element)
rwrk_init_var(&ivar,&idx,3,S);      // v2(element)
rwrk_init_var(&ivar,&idx,3,S);      // v3(element)
rwrk_init_var(&ivar,&idx,3,S);      // v4(element)
rwrk_init_var(&ivar,&idx,3,S);      // v5(element)
rwrk_init_var(&ivar,&idx,3,S);      // v6(element)
rwrk_init_var(&ivar,&idx,3,S);      // v7(element)
rwrk_init_var(&ivar,&idx,3,S);      // a0(element)
rwrk_init_var(&ivar,&idx,3,S);      // a1(element)
rwrk_init_var(&ivar,&idx,3,S);      // a2(element)
```

```
    rwrk_init_var(&ivar,&idx,3,S);    // a3(element)
    rwrk_init_var(&ivar,&idx,3,S);    // a4(element)
    rwrk_init_var(&ivar,&idx,3,S);    // a5(element)
    rwrk_init_var(&ivar,&idx,3,S);    // a6(element)
    rwrk_init_var(&ivar,&idx,3,S);    // a7(element)
    rwrk_init_var(&ivar,&idx,3,S);    // delta(element)
    rwrk_init_var(&ivar,&idx,3,S);    // gamma(element)
    rwrk_init_var(&ivar,&idx,3,S);    // jerk(element)
    rwrk_init_var(&ivar,&idx,3,S);    // idir(element)
    rwrk_init_var(&ivar,&idx,3,S);    // xstrt(element)


    return 1;
}
```

B.4.5 HD_V4_Movetest.c

```
#include "HD_V4_IOPorts.h"


#define DELTA                       3E-3
//      Vibration: period of pulse
#define vibration_step              (DELTA/ssGetStepSize(S))
//      Vibration: number of samples per part of a pulse
#define DETECTION_LEVEL             10E-6
//      Vibration: detection level [m]



// ------------------------------------------------------------------------
// Hdrive_movetest_out()
// ------------------------------------------------------------------------



int_T Hdrive_movetest_out(real_T *u,real_T *pos, SimStruct *S)
{
     int_T     i;
     int_T     *piwrk    = ssGetIWork(S);
     real_T    *prwrk    = ssGetRWork(S);
     real_T    *yPtrs    = ssGetOutputPortRealSignal(S,0);
     InputRealPtrsType   uPtrs = ssGetInputPortRealSignalPtrs(S,0);
     real_T     DeltaT   = ssGetStepSize(S); //sample time


     if ssIsSampleHit(S,0,tid){
       for (i=0;i<noa;i++) {
         if (IN_EPD(i)==1) {
            ssSetErrorStatus(S,
              "ERROR: Starting the zero-search procedure at homing-point is "
              "dangerous. Switch off the H-Drive and move the LiMMS manually "
              "to safe area.");
            TEST_FAULT(i)=1;
          }

         if((int_T)substate(i)==TEST) {   // Only do this when axis is still
                                          // in test-mode
           OUT_RST_ENC(i)=0;              //  reset encoder

           if (START_PULSE(i) ==  1){     // The motor will be excited by a
                                          // serie of short pulses
             u[i]=Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
             OUT_RST_ENC(i)=1;           // reset encoder
             START_PULSE(i)=0;
           }
           else if (DISC_STEP(i)< ( 1*vibration_step)) {
             u[i]=Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
           }
           else if (DISC_STEP(i)< ( 2*vibration_step)) {
             u[i]=Iref(i)*sin((pi/(DELTA*2))*DeltaT*DISC_STEP(i)+(pi/2));
           }
           else if (DISC_STEP(i)==( 2*vibration_step)){
             temp1(i)=pos[i];
             u[i]=Iref(i)*sin((pi/(DELTA*2))*DeltaT*DISC_STEP(i)+(pi/2));
           }
           else if (DISC_STEP(i)< ( 3*vibration_step)) {
```

```
            u[i]=Iref(i)*sin((pi/(DELTA*2))*DeltaT*DISC_STEP(i)+(pi/2));
          }
          else if (DISC_STEP(i)< ( 4*vibration_step)) {
            u[i]=-Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
          }
          else if (DISC_STEP(i)==( 4*vibration_step)){
            temp2(i)=pos[i];
            u[i]=-Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
          }
          else if (DISC_STEP(i)< ( 5*vibration_step))
            u[i]=-Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
          else if (DISC_STEP(i)< ( 6*vibration_step))
            u[i]=-Iref(i)*sin((pi/(DELTA*2))*DeltaT*DISC_STEP(i)+(pi/2));
          else if (DISC_STEP(i)==( 6*vibration_step)){
            temp3(i)=pos[i];
            u[i]=-Iref(i)*sin((pi/(DELTA*2))*DeltaT*DISC_STEP(i)+(pi/2));
          }
          else if (DISC_STEP(i)< ( 7*vibration_step))
            u[i]=-Iref(i)*sin((pi/(DELTA*2))*DeltaT*DISC_STEP(i)+(pi/2));
          else if (DISC_STEP(i)< ( 8*vibration_step))
            u[i]=Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
          else if (DISC_STEP(i)==( 8*vibration_step)){
            temp4(i)=pos[i];
            RESULT(i)=-(temp0(i)-temp1(i))+(temp1(i)-temp2(i))
                      +(temp2(i)-temp3(i))-(temp3(i)-temp4(i));
            OUT_RESULT(i)=RESULT(i);
            u[i]=0;
          }
          else if (DISC_STEP(i)< (10*vibration_step))
            u[i]=0;
          else if (DISC_STEP(i)==(10*vibration_step)){
            temp0(i)=pos[i];
            OUT_RESULT(i)=0;
            if ((RESULT(i)>=-DETECTION_LEVEL)&&(RESULT(i)<=DETECTION_LEVEL))
              {Iref(i)=1.2*Iref(i);
              PHI(i)=PHI(i)+(pi/2);
              TEST_COUNT(i)=0;
              if (Iref(i)>Imax_Algn) {
                Iref(i)=0;
                TEST_FAULT(i)=1;
              }
            }
            else {
              ++TEST_COUNT(i);
              TEST_FAULT(i)=0;
            }
            DISC_STEP(i)=0;
            u[i]=Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
            OUT_RST_ENC(i)=0;  // reset encoder
          }

          DISC_STEP(i)++;

        }
        else { // axis was already done with testing-stage
          u[i]=0;
        }
      }
    }

    return 1;
}
```

```
// ------------------------------------------------------------------------
// Hdrive_zerosearch_out()
// ------------------------------------------------------------------------



int_T Hdrive_zerosearch_out(real_T *u,real_T *pos, SimStruct *S)
{
     int_T     i;
     int_T     *piwrk    = ssGetIWork(S);
     real_T    *prwrk    = ssGetRWork(S);
     real_T    *yPtrs        = ssGetOutputPortRealSignal(S,0);
     real_T    DeltaT    = ssGetStepSize(S); //sample time


     if ssIsSampleHit(S,0,tid){
       for (i=0;i<noa;i++) {
         if ((int_T)substate(i)==ZERO_SEARCH) {  // Only do this when axis is
                                                 // still in zero-search-mode

           if (DISC_STEP(i)< ( 1*vibration_step))
             u[i]=Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
           else if (DISC_STEP(i)< ( 2*vibration_step))
              u[i]=Iref(i)*sin((pi/(DELTA*2))*DeltaT*DISC_STEP(i)+(pi/2));
           else if (DISC_STEP(i)==( 2*vibration_step)){
              temp1(i)=pos[i];
              u[i]=Iref(i)*sin((pi/(DELTA*2))*DeltaT*DISC_STEP(i)+(pi/2));}
           else if (DISC_STEP(i)< ( 3*vibration_step))
             u[i]=Iref(i)*sin((pi/(DELTA*2))*DeltaT*DISC_STEP(i)+(pi/2));
           else if (DISC_STEP(i)< ( 4*vibration_step))
             u[i]=-Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
           else if (DISC_STEP(i)==( 4*vibration_step)){
             temp2(i)=pos[i];
             u[i]=-Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
           }
           else if (DISC_STEP(i)< ( 5*vibration_step))
             u[i]=-Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
           else if (DISC_STEP(i)< ( 6*vibration_step))
             u[i]=-Iref(i)*sin((pi/(DELTA*2))*DeltaT*DISC_STEP(i)+(pi/2));
           else if (DISC_STEP(i)==( 6*vibration_step)) {
            temp3(i)=pos[i];
             u[i]=-Iref(i)*sin((pi/(DELTA*2))*DeltaT*DISC_STEP(i)+(pi/2));
           }
           else if (DISC_STEP(i)< ( 7*vibration_step))
             u[i]=-Iref(i)*sin((pi/(DELTA*2))*DeltaT*DISC_STEP(i)+(pi/2));
           else if (DISC_STEP(i)< ( 8*vibration_step))
             u[i]=Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
           else if (DISC_STEP(i)==( 8*vibration_step)) {
             temp4(i)=pos[i];
             RESULT(i)=-(temp0(i)-temp1(i))+(temp1(i)-temp2(i))
                      +(temp2(i)-temp3(i))-(temp3(i)-temp4(i));
             OUT_RESULT(i)=RESULT(i);
             u[i]=0;}
           else if (DISC_STEP(i)< (10*vibration_step))
             u[i]=0;
           else if (DISC_STEP(i)==(10*vibration_step))
```

```
            {temp0(0)=pos[i];
            if ((RESULT(i)>=-DETECTION_LEVEL)&&(RESULT(i)<=DETECTION_LEVEL))
              {Iref(i)=1.2*Iref(i);
              if (Iref(i)>Imax_Algn)
                {Iref(i)=0;
                ALLIGN_READY(i)=1;
              }
              AMPLITUDE_COUNT(i)=0;
            }
            else
              {AMPLITUDE_COUNT(i)++;            // Same amplitude count
              if (((PREVIOUS_RESULT(i)>0)&&(RESULT(i)<0)) ||
                 ((PREVIOUS_RESULT(i)<0)&&(RESULT(i)>0)))
                DPHI(i)=DPHI(i)*0.5;
                PREVIOUS_RESULT(i)=RESULT(i);
              if (RESULT(i)>0)
                PHI(i)=PHI(i)-DPHI(i);
              else
                PHI(i)=PHI(i)+DPHI(i);
            }
            OUT_DPHI(i)=DPHI(i);
            OUT_RESULT(i)=0;                    // signal result zero
            DISC_STEP(i)=0;
            u[i]=Iref(i)*sin((pi/DELTA)*DeltaT*DISC_STEP(i));
          }
          DISC_STEP(i)++;
        }
        else { // axis was already done with testing-stage
          u[i]=0;
        }
      }
    }

    return 1;
}
```

B.4.6 HD_V4_Safety.c

```
/*
    SAFETY AIRBAG
*/



// ------------------------------------------------------------------------
// safety_check()
// ------------------------------------------------------------------------



void safety_check(real_T *u, real_T *pos, real_T *vel, SimStruct *S)
{

    int_T *piwrk=ssGetIWork(S);
    real_T *prwrk=ssGetRWork(S);

    int_T i;

    // *** SAFETY CHECK PER AXIS

    for (i=0;i<noa;i++) {

      // All OK
      if (STATUS==READY)
        substate(i)=READY;

      // Check for speed violation
      if (fabs(vel[i])>=maxspeed(i)) {
        vel_violation(i)=1;
        substate(i)=VEL_VIOLATION;
      }
      else {
        vel_violation(i)=0;
      }

      // Check for boundary tresspassing (only when initialisation has been
      // completed!)
      if (((pos[i]<=minpos(i)+margin(i))||(pos[i]>=maxpos(i)-margin(i))) &&
         ((int)STATUS==READY)) {
        pos_violation(i)=1;
        substate(i)=POS_VIOLATION;
      }
      else {
        pos_violation(i)=0;
      }

      // Check for too large current
      // Not in safety-layer, but in motor.c
      // Why? Compensating other violations can result in extra high
      // current levels that are not known at this point in het code
      // (compensation will be calculated in functions like
      // safety_pos_airbag())
    }

    // *** GLOBAL SAFETY-CHECK

    // Check for tilt Y-axes (only when initialisation has been
```

```
    // completed and all axes are operational)
    if (((int)STATUS==READY) && (noa>=3) && (fabs(pos[2]-pos[1])>maxangle)) {
      ang_violation=1;
      substate(1)=ANG_VIOLATION;
      substate(2)=ANG_VIOLATION;
    }
    else {
      ang_violation=0;
    }
}



// ------------------------------------------------------------------------
// safety_pos_airbag()
// ------------------------------------------------------------------------



void safety_pos_airbag(real_T *u, real_T *pos, real_T *vel, int_T i,
  SimStruct *S)
{
    int_T *piwrk=ssGetIWork(S);
    real_T *prwrk=ssGetRWork(S);

//  compensate boundary tresspassing: PD-control to border of safe area
    if (pos[i]<=minpos(i)+margin(i)) {
      u[i]=p_airbag(i)*(minpos(i)+margin(i)-pos[i])-d_airbag(i)*vel[i];
    }
    else if (pos[i]>=maxpos(i)-margin(i)) {
      u[i]=p_airbag(i)*(maxpos(i)-margin(i)-pos[i])-d_airbag(i)*vel[i];
    }
    else {
      u[i]=-d_airbag(i)*vel[i]; // tresspassing just compensated...
    }
}



// ------------------------------------------------------------------------
// safety_vel_airbag()
// ------------------------------------------------------------------------



void safety_vel_airbag(real_T *u, real_T *pos, real_T *vel, int_T i, SimStruct *S)
{
    int_T *piwrk=ssGetIWork(S);
    real_T *prwrk=ssGetRWork(S);
    real_T desired_vel;

//  compensate overspeed: brake

//  Original method: brake by applying a current that is -2 times the current
//  velocity (desired_vele=0)
//  u[i]=-p_vel_brake(i)*vel[i];

//  New method: control to maximum veocity
```

```
    desired_vel=maxspeed(i)*fabs(vel[i])/vel[i];
    u[i]=p_vel_brake(i)*(desired_vel-vel[i]);
}



// ------------------------------------------------------------------------
// safety_angle_viol()
// ------------------------------------------------------------------------



void safety_angle_viol_ini(real_T t, real_T *pos, SimStruct *S)
{
//   int_T *piwrk=ssGetIWork(S);
//   real_T *prwrk=ssGetRWork(S);
     real_T xstart,xend,tdes,vdes;

//   Generate profile that describes the distance between Y2 and Y1 in time
//   p2p_ini(real_T xstart, real_T tstart, real_T xend, real_T vdes,
//   real_T tdes, real_T maxjerk, int_T i, SimStruct *S)

     xstart=pos[2]-pos[1];   // initial delta
     xend=0;                 // final delta
     vdes=(xend-xstart)*5;   // speed (delta=0within 1/5-th of a second)
     tdes=0.1;               // time to reach vdes

     p2p_ini(xstart,t,xend,vdes,tdes,100,2,S);
}



void safety_angle_viol(real_T *u, real_T t, real_T *pos, real_T *vel,
  const real_T *xc, SimStruct *S)
{
     int_T *piwrk=ssGetIWork(S);
     real_T *prwrk=ssGetRWork(S);
     real_T desired_vel,y2xref,y2vref,qref[1],vref[1],aref[1];

     desired_vel=0.0;

//   X-axis: Brakte to zero velocity
     u[0]=p_vel_brake(0)*(desired_vel-vel[0]);
//   Y1-axis: Brake to zero-velocity
     u[1]=p_vel_brake(1)*(desired_vel-vel[1]);
     // Make sure u[1] doesn't get to big so Y2 can follow Y1
     if (fabs(u[1])>0.8*Imax_Low) {
       u[1]=0.8*Imax_Low*(u[1]/fabs(u[1]));
     }
//   Y2-axis: Follow Y1 as good as possible
     p2p_get(&qref[0],&vref[0],&aref[0],t,2,S);  // (Y2-Y1)-profile
     y2xref=pos[1]+qref[0];
     y2vref=0; //vel[1]+vref[0];
     u[2]=p_move(2)*(y2xref-pos[2])+d_move(2)*(y2vref-vel[2])+i_move(2)*xc[2+4];
}
```

```
// ------------------------------------------------------------------------------
// safety_direct_stop()
// ------------------------------------------------------------------------------



void safety_direct_stop(real_T *u, real_T *pos, real_T *vel, const real_T *xc,
  SimStruct *S)
{
     int_T *piwrk=ssGetIWork(S);
     real_T *prwrk=ssGetRWork(S);
     real_T desired_vel;
     int_T i;

     for(i=0;i<noa;i++) {
       // X, Y1 -> Brake to zero
       desired_vel=0.0;
       if (i<2) {
         u[i]=p_vel_brake(i)*(desired_vel-vel[i]);
         if (fabs(u[i])>0.8*Imax_High) {
           u[i]=0.8*Imax_High*(u[i]/fabs(u[i]));
          }
       }
       else {
         // Y2 -> Follow Y1-axis
         u[i]=p_move(2)*(pos[1]-pos[2])+d_move(2)*(vel[1]-vel[2])+i_move(2)*xc[2+4];
       }
     }
}
```

B.4.7 HD_V4_Motor.c

```
/*
     send motor commands
*/

#include "HD_V4_IOPorts.h"
#define  TAU 0.012              // Magnet Pitch (distance between N-S Pole) [m]



// ------------------------------------------------------------------------------
// send_motor_command()
// ------------------------------------------------------------------------------



int_T send_motor_command(real_T *u, real_T *pos, SimStruct *S)
{
     int_T     i,iclip;

     int_T     *piwrk = ssGetIWork(S);
     real_T    *prwrk = ssGetRWork(S);
     real_T    *yPtrs = ssGetOutputPortRealSignal(S,0);


//   clip motor command if higher than maxout
     iclip=1;

     for (i=0;i<noa;i++){

       // Determine maximum current level
       if (((int_T)pos_violation(i)==1) ||
          (((int_T)STATUS==EMERGENCY_STOP_SYSTEM))) {
         maxout(i)=Imax_High;             // Position violation ->
       }                                   // Larger current allowed
       else {
         maxout(i)=Imax_Low;              // Normal operation ->
       }                                   // normal current level

       // Check for too large current
       if (fabs(u[i])>maxout(i)) {
         u[i]=maxout(i)*u[i]/fabs(u[i]);
         i_violation(i)=1;
         iclip=(int_T) I_VIOLATION;
         if ((int_T)substate(i)==READY)   // NO other violations?
           substate(i)=I_VIOLATION;
       }
       else
         i_violation(i)=0;

       // Outputs
       OUT_I(i)=u[i];
       OUT_PHI(i)=PHI(i)+commdir(i)*((pos[i]+pos_reset(i))*pi/TAU);
     }

     return iclip;
}
```

B.4.8 HD_V4_PID.c

```
// simple pid-controller



// ------------------------------------------------------------------------
// Defines and includes
// ------------------------------------------------------------------------



#define KGAIN                       100
#include "HD_V4_IOPorts.h"



// ------------------------------------------------------------------------
// pid_ini()
// ------------------------------------------------------------------------



void pid_ini(real_T *xc, real_T *pos)
{
     int_T i;

     for (i=0;i<noa;i++) {
       xc[i]=pos[i];
       xc[i+4]=0.0;
     }
}



// ------------------------------------------------------------------------
// pid_dif()
// ------------------------------------------------------------------------



void pid_dif(real_T *dxcdt, const real_T *xc, real_T *pos, real_T *qref)
{
     int_T i;

//   xc[0...3] represent filtered positions

     for (i=0;i<noa;i++) {
       dxcdt[i]=KGAIN*(pos[i]-xc[i]);
       dxcdt[i+4]=qref[i]-pos[i];
     }
}



// ------------------------------------------------------------------------
```

```
// pid_out()
// -----------------------------------------------------------------------------



void pid_out(real_T *u, real_T *vel, const real_T *xc, real_T *pos,
  real_T *qref, real_T *vref, SimStruct *S)
{
     int_T i;
     real_T    *yPtrs = ssGetOutputPortRealSignal(S,0);
     int_T     *piwrk = ssGetIWork(S);
     real_T    *prwrk = ssGetRWork(S);

     for (i=0;i<noa;i++) {

//     velocity estimates
       vel[i]=KGAIN*(pos[i]-xc[i]);

//     simple PID-controller
       u[i]=p_move(i)*(qref[i]-pos[i])+d_move(i)*(vref[i]-vel[i])
           +i_move(i)*xc[i+4];
       OUT_CTRL_OUT(i)=u[i];
     }
}
```

B.4.9 HD_V4_Jog.c

```
// jogging and point-to-point motion based on third-degree setpoint profile
// Rene' van de Molengraft, April, 14th, 2000
// saves settings for four different axes
// time t is the absolute time



// ------------------------------------------------------------------------------
// Defines and includes
// ------------------------------------------------------------------------------



// shortcuts for jogging variables in workspace
// #define JOG_IDX                  See HD_Work.c for starting index
#define t0(element)                 prwrk[piwrk[JOG_IDX+0]+element]
#define t1(element)                 prwrk[piwrk[JOG_IDX+1]+element]
#define t2(element)                 prwrk[piwrk[JOG_IDX+2]+element]
#define t3(element)                 prwrk[piwrk[JOG_IDX+3]+element]
#define t4(element)                 prwrk[piwrk[JOG_IDX+4]+element]
#define t5(element)                 prwrk[piwrk[JOG_IDX+5]+element]
#define t6(element)                 prwrk[piwrk[JOG_IDX+6]+element]
#define t7(element)                 prwrk[piwrk[JOG_IDX+7]+element]
#define s0(element)                 prwrk[piwrk[JOG_IDX+8]+element]
#define s1(element)                 prwrk[piwrk[JOG_IDX+9]+element]
#define s2(element)                 prwrk[piwrk[JOG_IDX+10]+element]
#define s3(element)                 prwrk[piwrk[JOG_IDX+11]+element]
#define s4(element)                 prwrk[piwrk[JOG_IDX+12]+element]
#define s5(element)                 prwrk[piwrk[JOG_IDX+13]+element]
#define s6(element)                 prwrk[piwrk[JOG_IDX+14]+element]
#define s7(element)                 prwrk[piwrk[JOG_IDX+15]+element]
#define v0(element)                 prwrk[piwrk[JOG_IDX+16]+element]
#define v1(element)                 prwrk[piwrk[JOG_IDX+17]+element]
#define v2(element)                 prwrk[piwrk[JOG_IDX+18]+element]
#define v3(element)                 prwrk[piwrk[JOG_IDX+19]+element]
#define v4(element)                 prwrk[piwrk[JOG_IDX+20]+element]
#define v5(element)                 prwrk[piwrk[JOG_IDX+21]+element]
#define v6(element)                 prwrk[piwrk[JOG_IDX+22]+element]
#define v7(element)                 prwrk[piwrk[JOG_IDX+23]+element]
#define a0(element)                 prwrk[piwrk[JOG_IDX+24]+element]
#define a1(element)                 prwrk[piwrk[JOG_IDX+25]+element]
#define a2(element)                 prwrk[piwrk[JOG_IDX+26]+element]
#define a3(element)                 prwrk[piwrk[JOG_IDX+27]+element]
#define a4(element)                 prwrk[piwrk[JOG_IDX+28]+element]
#define a5(element)                 prwrk[piwrk[JOG_IDX+29]+element]
#define a6(element)                 prwrk[piwrk[JOG_IDX+30]+element]
#define a7(element)                 prwrk[piwrk[JOG_IDX+31]+element]
#define delta(element)              prwrk[piwrk[JOG_IDX+32]+element]
#define gamma(element)              prwrk[piwrk[JOG_IDX+33]+element]
#define jerk(element)               prwrk[piwrk[JOG_IDX+34]+element]
#define idir(element)               prwrk[piwrk[JOG_IDX+35]+element]
#define xstrt(element)              prwrk[piwrk[JOG_IDX+36]+element]



// ------------------------------------------------------------------------------
// jog_ini()
// ------------------------------------------------------------------------------
```

```
void jog_ini(real_T xstart, real_T tstart, real_T vdes, real_T tdes,
  real_T maxjerk, int_T i, SimStruct *S)
{
     /*
     input arguments
          xstart    : start position
          tstart    : start time
          vdes      : desired jogging speed
          tdes      : time to reach vdes
          maxjerk   : jerk in acceleration phase
          i         : axis id
     */

     real_T    det;
     int_T     *piwrk=ssGetIWork(S);
     real_T    *prwrk=ssGetRWork(S);

     // vdes en jerk positive, idir contains the direction of the movement

     if (vdes>=0.0) {
       idir(i)=1.0;
     } else {
       idir(i)=-1.0;
       vdes=-vdes;
     }

     if (maxjerk>=0.0) {
       jerk(i)=maxjerk;
     } else {
       jerk(i)=-maxjerk;
     }

//   compute jerk period delta
     det=tdes*tdes*jerk(i)*jerk(i)-4.0*jerk(i)*vdes;
     if (det<0) {
       ssSetErrorStatus(S,
         "JOG_INI: vdes cannot be reached (increase jerk and/or tdes).");
     } else {
       delta(i)=(tdes*jerk(i)-sqrt(det))/(2.0*jerk(i));

//     compute acceleration period gamma

       gamma(i)=tdes-2.0*delta(i);

//     compute switching times

       t0(i)=tstart;
       t1(i)=t0(i)+delta(i);
       t2(i)=t1(i)+gamma(i);
       t3(i)=t2(i)+delta(i);

//     t4, t5, t6 and t7 equal infinity at startup

       t4(i)=100000.0;
       t5(i)=100000.0;
       t6(i)=100000.0;
       t7(i)=100000.0;
```

```
//      compute reference values at switching times

        xstrt(i)=xstart;
        s0(i)=0.0;

        a1(i)=jerk(i)*(t1(i)-t0(i));
        v1(i)=0.5*jerk(i)*(t1(i)-t0(i))*(t1(i)-t0(i));
        s1(i)=s0(i)+jerk(i)*(t1(i)-t0(i))*(t1(i)-t0(i))*(t1(i)-t0(i))/6.0;

        a2(i)=a1(i);
        v2(i)=v1(i)+a1(i)*(t2(i)-t1(i));
        s2(i)=s1(i)+v1(i)*(t2(i)-t1(i))+0.5*a1(i)*(t2(i)-t1(i))*(t2(i)-t1(i));

        a3(i)=a2(i)-jerk(i)*(t3(i)-t2(i));
        v3(i)=v2(i)+a2(i)*(t3(i)-t2(i))-0.5*jerk(i)*(t3(i)-t2(i))*(t3(i)-t2(i));
        s3(i)=s2(i)+v2(i)*(t3(i)-t2(i))+0.5*a2(i)*(t3(i)-t2(i))*(t3(i)-t2(i))
             -jerk(i)*(t3(i)-t2(i))*(t3(i)-t2(i))*(t3(i)-t2(i))/6.0;
      }
}



// ------------------------------------------------------------------------------
// jog_get()
// ------------------------------------------------------------------------------



void jog_get(real_T *qref, real_T *vref, real_T *aref, real_T t, int_T i,
  SimStruct *S)
{

//   i : axis id

     int_T     *piwrk = ssGetIWork(S);
     real_T    *prwrk = ssGetRWork(S);

     // NOTE:
     // - QUESTION: Is it an error that index 0 is used in all statemets
     //   below instead of index i?
     // - ANSWER: No! The statement that calls the jog_get() function uses
     //   the correct index as argumnt (for example: v[i] as input), which
     //   results that in jog_get() element i of the orinal array is visible
     //   at position [0] in the function being called
     if (t<=t0(i)) {
       aref[0]=0;
       vref[0]=0;
       qref[0]=s0(i);
     } else if (t<=t1(i)) {
       aref[0]=jerk(i)*(t-t0(i));
       vref[0]=0.5*jerk(i)*(t-t0(i))*(t-t0(i));
       qref[0]=s0(i)+jerk(i)*(t-t0(i))*(t-t0(i))*(t-t0(i))/6.0;
     } else if (t<=t2(i)) {
       aref[0]=a1(i);
       vref[0]=v1(i)+a1(i)*(t-t1(i));
       qref[0]=s1(i)+v1(i)*(t-t1(i))+0.5*a1(i)*(t-t1(i))*(t-t1(i));
     } else if (t<=t3(i)) {
       aref[0]=a2(i)-jerk(i)*(t-t2(i));
       vref[0]=v2(i)+a2(i)*(t-t2(i))-0.5*jerk(i)*(t-t2(i))*(t-t2(i));
       qref[0]=s2(i)+v2(i)*(t-t2(i))+0.5*a2(i)*(t-t2(i))*(t-t2(i))
```

```
            -jerk(i)*(t-t2(i))*(t-t2(i))*(t-t2(i))/6.0;
    } else if (t<=t4(i)) {
      aref[0]=0;
      vref[0]=v3(i);
      qref[0]=s3(i)+v3(i)*(t-t3(i));
    } else if (t<=t5(i)) {
      aref[0]=-jerk(i)*(t-t4(i));
      vref[0]=v4(i)-0.5*jerk(i)*(t-t4(i))*(t-t4(i));
      qref[0]=s4(i)+v4(i)*(t-t4(i))-jerk(i)*(t-t4(i))*(t-t4(i))*(t-t4(i))/6.0;
    } else if (t<=t6(i)) {
      aref[0]=a5(i);
      vref[0]=v5(i)+a5(i)*(t-t5(i));
      qref[0]=s5(i)+v5(i)*(t-t5(i))+0.5*a5(i)*(t-t5(i))*(t-t5(i));
    } else if (t<=t7(i)) {
      aref[0]=a6(i)+jerk(i)*(t-t6(i));
      vref[0]=v6(i)+a6(i)*(t-t6(i))+0.5*jerk(i)*(t-t6(i))*(t-t6(i));
      qref[0]=s6(i)+v6(i)*(t-t6(i))+0.5*a6(i)*(t-t6(i))*(t-t6(i))
            +jerk(i)*(t-t6(i))*(t-t6(i))*(t-t6(i))/6.0;
    } else {
      aref[0]=0.0;
      vref[0]=0.0;
      qref[0]=s7(i);
    }

    if (idir(i)==-1.0) {
      aref[0]=-aref[0];
      vref[0]=-vref[0];
      qref[0]=-qref[0];
    }

    qref[0]=qref[0]+xstrt(i);
}



// ------------------------------------------------------------------------
// jog_stop()
// ------------------------------------------------------------------------



void jog_stop(real_T t, int_T i, SimStruct *S)
{
//   i : axis id

    int_T     *piwrk = ssGetIWork(S);
    real_T    *prwrk = ssGetRWork(S);

    t4(i)=t;
    t5(i)=t4(i)+delta(i);
    t6(i)=t5(i)+gamma(i);
    t7(i)=t6(i)+delta(i);

    a4(i)=0.0;
    v4(i)=v3(i);
    s4(i)=s3(i)+v3(i)*(t4(i)-t3(i));

    a5(i)=-jerk(i)*(t5(i)-t4(i));
    v5(i)=v4(i)-0.5*jerk(i)*(t5(i)-t4(i))*(t5(i)-t4(i));
    s5(i)=s4(i)+v4(i)*(t5(i)-t4(i))-jerk(i)*(t5(i)-t4(i))*(t5(i)-t4(i))*(t5(i)
```

```
          -t4(i))/6.0;

     a6(i)=a5(i);
     v6(i)=v5(i)+a5(i)*(t6(i)-t5(i));
     s6(i)=s5(i)+v5(i)*(t6(i)-t5(i))+0.5*a5(i)*(t6(i)-t5(i))*(t6(i)-t5(i));

     a7(i)=a6(i)+jerk(i)*(t7(i)-t6(i));
     v7(i)=v6(i)+a6(i)*(t7(i)-t6(i))+0.5*jerk(i)*(t7(i)-t6(i))*(t7(i)-t6(i));
     s7(i)=s6(i)+v6(i)*(t7(i)-t6(i))+0.5*a6(i)*(t7(i)-t6(i))*(t7(i)-t6(i))
          +jerk(i)*(t7(i)-t6(i))*(t7(i)-t6(i))*(t7(i)-t6(i))/6.0;
}



// ------------------------------------------------------------------------
// jog_status()
// ------------------------------------------------------------------------



int_T jog_status(real_T t, int_T i,real_T tset, SimStruct *S)
{
//   i : axis id

     int_T *piwrk=ssGetIWork(S);
     real_T *prwrk=ssGetRWork(S);

     if (t>t7(i)+tset) {
       return 1;
     } else {
       return 0;
     }
}



// ------------------------------------------------------------------------
// p2p_ini()
// ------------------------------------------------------------------------



void p2p_ini(real_T xstart, real_T tstart, real_T xend, real_T vdes,
  real_T tdes, real_T maxjerk, int_T i, SimStruct *S)
{
     /*
     input arguments
          xstart    : start position
          tstart    : start time
          xend      : end position
          vdes      : desired jogging speed
          tdes      : time to reach vdes
          maxjerk   : jerk in acceleration phase
          i         : axis id
     */

     real_T    det,disp;
```

```
    int_T     *piwrk = ssGetIWork(S);
    real_T    *prwrk = ssGetRWork(S);

    disp=xend-xstart;
    if (disp<0.0) {
      disp=-disp;
    }

    if (vdes>=0.0) {
      idir(i)=1.0;
    } else {
      idir(i)=-1.0;
      vdes=-vdes;
    }

    if (maxjerk>=0.0) {
      jerk(i)=maxjerk;
    } else {
      jerk(i)=-maxjerk;
    }

//  compute jerk period delta

    det=tdes*tdes*jerk(i)*jerk(i)-4.0*jerk(i)*vdes;
    if (det<0) {
      ssSetErrorStatus(S,
        "P2P_INI: vdes cannot be reached (increase jerk and/or tdes).");
    } else {
      delta(i)=(tdes*jerk(i)-sqrt(det))/(2.0*jerk(i));

//    compute acceleration period gamma

      gamma(i)=tdes-2.0*delta(i);

//    compute switching times

      t0(i)=tstart;
      t1(i)=t0(i)+delta(i);
      t2(i)=t1(i)+gamma(i);
      t3(i)=t2(i)+delta(i);

//    compute t4

      t4(i)=t3(i)+(disp-2.0*jerk(i)*delta(i)*delta(i)*delta(i)
           -3.0*jerk(i)*delta(i)*delta(i)*gamma(i)
           -jerk(i)*delta(i)*gamma(i)*gamma(i))/vdes;
      if (t4(i)<t3(i)) {
        ssSetErrorStatus(S,
          "P2P_INI: vdes too high for displacement (decrease vdes).");
      } else {
        t5(i)=t4(i)+delta(i);
        t6(i)=t5(i)+gamma(i);
        t7(i)=t6(i)+delta(i);

//      compute reference values at switching times

        xstrt(i)=xstart;
        s0(i)=0.0;

        a1(i)=jerk(i)*(t1(i)-t0(i));
        v1(i)=0.5*jerk(i)*(t1(i)-t0(i))*(t1(i)-t0(i));
        s1(i)=s0(i)+jerk(i)*(t1(i)-t0(i))*(t1(i)-t0(i))*(t1(i)-t0(i))/6.0;
```

```
        a2(i)=a1(i);
        v2(i)=v1(i)+a1(i)*(t2(i)-t1(i));
        s2(i)=s1(i)+v1(i)*(t2(i)-t1(i))+0.5*a1(i)*(t2(i)-t1(i))*(t2(i)-t1(i));

        a3(i)=a2(i)-jerk(i)*(t3(i)-t2(i));
        v3(i)=v2(i)+a2(i)*(t3(i)-t2(i))-0.5*jerk(i)*(t3(i)-t2(i))*(t3(i)-t2(i));
        s3(i)=s2(i)+v2(i)*(t3(i)-t2(i))+0.5*a2(i)*(t3(i)-t2(i))*(t3(i)-t2(i))
             -jerk(i)*(t3(i)-t2(i))*(t3(i)-t2(i))*(t3(i)-t2(i))/6.0;

        a4(i)=0.0;
        v4(i)=v3(i);
        s4(i)=s3(i)+v3(i)*(t4(i)-t3(i));

        a5(i)=-jerk(i)*(t5(i)-t4(i));
        v5(i)=v4(i)-0.5*jerk(i)*(t5(i)-t4(i))*(t5(i)-t4(i));
        s5(i)=s4(i)+v4(i)*(t5(i)-t4(i))
             -jerk(i)*(t5(i)-t4(i))*(t5(i)-t4(i))*(t5(i)-t4(i))/6.0;

        a6(i)=a5(i);
        v6(i)=v5(i)+a5(i)*(t6(i)-t5(i));
        s6(i)=s5(i)+v5(i)*(t6(i)-t5(i))+0.5*a5(i)*(t6(i)-t5(i))*(t6(i)-t5(i));

        a7(i)=a6(i)+jerk(i)*(t7(i)-t6(i));
        v7(i)=v6(i)+a6(i)*(t7(i)-t6(i))+0.5*jerk(i)*(t7(i)-t6(i))*(t7(i)-t6(i));
        s7(i)=s6(i)+v6(i)*(t7(i)-t6(i))+0.5*a6(i)*(t7(i)-t6(i))*(t7(i)-t6(i))
             +jerk(i)*(t7(i)-t6(i))*(t7(i)-t6(i))*(t7(i)-t6(i))/6.0;
      }
    }
}



// ------------------------------------------------------------------------
// p2p_get()
// ------------------------------------------------------------------------



void p2p_get(real_T *qref, real_T *vref, real_T *aref, real_T t, int_T i,
  SimStruct *S)
{
    jog_get(qref,vref,aref,t,i,S);
}



// ------------------------------------------------------------------------
// p2p_status()
// ------------------------------------------------------------------------



int_T p2p_status(real_T t, int_T i,real_T tset, SimStruct *S)
{
    return jog_status(t,i,tset,S);
}
```

B.4.10 HD_V4_Vapi.c

```
#include "HD_V4_IOPorts.h"



// ------------------------------------------------------------------------
// Hdrive_initialize()
// ------------------------------------------------------------------------



int_T Hdrive_initialize(real_T *pos, SimStruct *S)
{
     int_T     i;
     int_T     *piwrk = ssGetIWork(S);
     real_T    *prwrk = ssGetRWork(S);
     real_T    *yPtrs = ssGetOutputPortRealSignal(S,0);


     for (i=0;i<noa;i++) {
       epd(i)=0.0;
       pos[i]=0.0;
       pos_reset(i)=0.0;                // reset
       HOMING1_READY(i)=0;
       OUT_RST_ENC(i)=1;

       AMPLITUDE_COUNT(i)= 0;
       DISC_STEP(i)      = 0;
       TEST_COUNT(i)     = 0;
       TEST_FAULT(i)     = 0;

       DPHI(i)           = 0.25*pi;
       PHI(i)            = 0;
       RESULT(i)         = 0;
       Iref(i)           = 0.2;
       PREVIOUS_RESULT(i)= 0;

       temp0(i)          = 0;
       temp1(i)          = 0;
       temp2(i)          = 0;
       temp3(i)          = 0;
       temp4(i)          = 0;

       pos_violation(i)  = 0;
       vel_violation(i)  = 0;
       i_violation(i)    = 0;
     }
     ang_violation       = 0;

     triggertime         = 0;
     HOMING1_ALL_READY   = 0;
     YALGN_STATE         = 0;

     return 1;
}



// ------------------------------------------------------------------------
```

```
// Hdrive_start_homing()
// ------------------------------------------------------------------------------



int_T Hdrive_start_homing(real_T *pos, real_T t, SimStruct *S)
{
     int_T     i;
     real_T    *x=ssGetContStates(S);
     InputRealPtrsType   uPtrs = ssGetInputPortRealSignalPtrs(S,0);
     int_T     *piwrk = ssGetIWork(S);
     real_T    *prwrk = ssGetRWork(S);


//   initialize jogging parameters

     for (i=0;i<noa;i++) {
//     as long as epd is not seen...
       epd(i)=IN_EPD(i);
       if (epd(i)==1.0 && (HOMING1_READY(i)==0)) {
         break;
       }
       else{
         jog_ini(pos[i],t,vh(i),0.1,100.0,i,S);
       }
     }


//   initialize controller states
     pid_ini(x,pos);

     return 1;
}



// ------------------------------------------------------------------------------
// Hdrive_homing_dif()
// ------------------------------------------------------------------------------



int_T Hdrive_homing_dif(real_T *dx, real_T *x, real_T *pos, real_T t,
  SimStruct *S)
{
     int_T     i;
     real_T    qref[noa],vref[noa],aref[noa];

     real_T    *yPtrs = ssGetOutputPortRealSignal(S,0);


     for (i=0;i<noa;i++) {
       jog_get(&qref[i],&vref[i],&aref[i],t,i,S);
     }

     //pid_dif(real_T *dxcdt, const real_T *xc, real_T *pos, real_T *qref)
     pid_dif(dx,x,pos,qref);

     for (i=0;i<noa;i++) {
```

```
        OUT_CTRL_SP(i)=qref[i];
    }

    return 1;
}



// ------------------------------------------------------------------------
// Hdrive_homing_out()
// ------------------------------------------------------------------------



int_T Hdrive_homing_out(real_T *u, real_T *vel, real_T *x, real_T *pos,
  real_T t, SimStruct *S)
{
    int_T     i;
    real_T    qref[noa],vref[noa],aref[noa];
    real_T    *yPtrs = ssGetOutputPortRealSignal(S,0);


    for (i=0;i<noa;i++) {
      jog_get(&qref[i],&vref[i],&aref[i],t,i,S);
    }
    // pid_out(real_T *u, real_T *vel, const real_T *xc, real_T *pos,
    //   real_T *qref, real_T *vref, SimStruct *S)
    pid_out(u,vel,x,pos,qref,vref,S);

    for (i=0;i<noa;i++) {
       OUT_CTRL_SP(i)=qref[i];
    }

    return 1;
}



// ------------------------------------------------------------------------
// Hdrive_start_moving()
// ------------------------------------------------------------------------



int_T Hdrive_start_moving(real_T *pos, real_T t, SimStruct *S)
{
    int_T     i;
    real_T    *x = ssGetContStates(S);

    int_T     *piwrk = ssGetIWork(S);
    real_T    *prwrk = ssGetRWork(S);


    for (i=0;i<noa;i++) {
    // p2p_ini(real_T xstart, real_T tstart, real_T xend, real_T vdes,
    // real_T tdes, real_T maxjerk, int_T i, SimStruct *S)
    // Original code: p2p_ini(pos[i],t,target(i),-2*vh(i),0.1,100.0,i,S);
       p2p_ini(pos[i],t,target(i),-2*vh(i),0.25,100.0,i,S);
```

```
    }

//  initialize controller states
    pid_ini(x,pos);

    return 1;
}



// ------------------------------------------------------------------------
// Hdrive_moving_dif()
// ------------------------------------------------------------------------



int_T Hdrive_moving_dif(real_T *dx, real_T *x, real_T *pos, real_T t,
  SimStruct *S)
{
    int_T     i;
    real_T    qref[noa],vref[noa],aref[noa];


    for (i=0;i<noa;i++) {
      p2p_get(&qref[i],&vref[i],&aref[i],t,i,S);
    }

    //pid_dif(real_T *dxcdt, const real_T *xc, real_T *pos, real_T *qref)
    pid_dif(dx,x,pos,qref); // V3.0 of software: control_dif(dx,x,pos,qref);

    return 1;
}



// ------------------------------------------------------------------------
// Hdrive_moving_out()
// ------------------------------------------------------------------------



int_T Hdrive_moving_out(real_T *u, real_T *vel, real_T *x, real_T *pos,
  real_T t, SimStruct *S)
{
    int_T     i;
    real_T    qref[noa],vref[noa],aref[noa];

    real_T    *yPtrs = ssGetOutputPortRealSignal(S,0);


    for (i=0;i<noa;i++) {
      p2p_get(&qref[i],&vref[i],&aref[i],t,i,S);
    }

    // pid_out(real_T *u, real_T *vel, const real_T *xc, real_T *pos,
    //   real_T *qref, real_T *vref, SimStruct *S)
    pid_out(u,vel,x,pos,qref,vref,S); //V3.0 soft.: control_out(u,x,pos,qref,S);
```

```
    for (i=0;i<noa;i++) {
       OUT_CTRL_SP(i)=qref[i];
    }

    return 1;
}



// ------------------------------------------------------------------------
// Hdrive_synchronize()
// ------------------------------------------------------------------------



int_T Hdrive_synchronize(real_T *pos, real_T *tar, real_T t, int_T sync_id,
  SimStruct *S)
{
    int_T     i,iret,count;

    InputRealPtrsType   uPtrs  = ssGetInputPortRealSignalPtrs(S,0);
    real_T              *yPtrs = ssGetOutputPortRealSignal(S,0);
    real_T              *x     = ssGetContStates(S);
    int_T               *piwrk = ssGetIWork(S);
    real_T              *prwrk = ssGetRWork(S);


    iret=0;


    switch (sync_id) {


    case HdriveHOME1:
         for (i=0;i<noa;i++) {
//         as long as epd is not seen...
           if (epd(i)==0.0) {
             epd(i)=IN_EPD(i);
             if (epd(i)==1.0) {
               jog_stop(t,i,S);
             }
           }
         }

         count=0;
         for (i=0;i<noa;i++) {
           if ((jog_status(t,i,0.1,S))==1) {
             count++;
             HOMING1_READY(i)=1;
           }
         }

         if (count==noa) {
           HOMING1_ALL_READY=1;
           iret=1;
         }
         else {
           iret=0;
         }
```

```
    break;


    case HdriveHOME2:              // Second part of the homing procedure....
         for (i=0;i<noa;i++) {
           // as long as epd is seen...
           if (epd(i)==1.0) {
             epd(i)=IN_EPD(i);
             if (epd(i)==0.0) {
               // EPD Found -> Store position and brake movement
               Epd_Marker(i)=pos[i];
               jog_stop(t,i,S);
             }
           }
         }


         count=0;
         for (i=0;i<noa;i++) {
           if (jog_status(t,i,1,S)==1) {
             count++;
           }
         }

         if (count==noa) {
           // temporary variable to store overhoot with respect to epd
           real_T epd_overshoot[3] = {0.0, 0.0, 0.0};
           // reset positions and determine overshoot
           for (i=0;i<noa;i++) {
             // Reset position, using the location of the EPD
             // sensor, determined earlier
             epd_overshoot[i]=pos[i]-Epd_Marker(i);
             pos_reset(i)=Epd_Marker(i);
             pos[i]=epd_overshoot[i];
           }
//         reinitialize differentiator states
           pid_ini(x,pos); // V3.0 of software: control_ini(x);
           iret=1;
         }
         else {
           iret=0;
         }
    break;


    case HdriveMOVE:
         count=0;
         for (i=0;i<noa;i++) {
           if (p2p_status(t,i,TREST,S)==1) {
             count++;
           }
         }

         if (count==noa) {
           iret=1;
           pid_ini(x,pos);
           OUT_CTRL_EN=1;          // enable external controller
         }
         else {
           iret=0;
         }
    break;
```

```
    }

    return iret;
}



// ------------------------------------------------------------------------
// Hdrive_ready_dif()
// ------------------------------------------------------------------------



int_T Hdrive_ready_dif(real_T *dx, real_T *x, real_T *pos, real_T t)
{
    pid_dif(dx,x,pos,pos);

    return 1;
}



// ------------------------------------------------------------------------
// Hdrive_violation_dif()
// ------------------------------------------------------------------------



int_T Hdrive_violation_dif(real_T *dx, real_T *x, real_T *pos, real_T t)
{
//  we still need velocity estimates for airbag...
    pid_dif(dx,x,pos,pos);

    return 1;
}



// ------------------------------------------------------------------------
// Hdrive_yalign_restart()
// ------------------------------------------------------------------------

// Start alignment of y2 axis with respect to y1 axis or go to nwxt phase

int_T Hdrive_yalign_restart(real_T *pos, real_T t, int_T contr_id,
  SimStruct *S)
{
    int_T     i;
    real_T    *x=ssGetContStates(S);
//  InputRealPtrsType   uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    int_T     *piwrk = ssGetIWork(S);
    real_T    *prwrk = ssGetRWork(S);
//  real_T    *yPtrs = ssGetOutputPortRealSignal(S,0);

    switch (contr_id) {
```

```
      case YALGN_FIND_AVS_B:
           // store position of second avs-sensor
           for (i=0;i<noa;i++) {
             posreset_yalgn(i)=pos[i];
           }
           // initialize jogging parameters
           jog_ini(pos[2],t,+vyalgn,0.125,100.0,2,S);
      break;

      case YALGN_FIND_AVS_A:
           // store position of first avs-sensor
           posreset_yalgn(2)=pos[2];
           // initialize jogging parameters
           jog_ini(pos[2],t,-vyalgn,0.125,100.0,2,S);
      break;

      case YALGN_CENTRE:
           // Move to centre:
           // Target: half between sensor AVS_A and AVS_B
           // AVS_A: posreset_yalgn(2) (due to "reset" at that position)
           // Target: (Position(AVS_A)+Position(AVS_B))/2 =
           //        = (posreset_yalgn(2)+pos[2])/2
           // p2p_ini(real_T xstart, real_T tstart, real_T xend, real_T vdes,
           //   real_T tdes, real_T maxjerk, int_T i, SimStruct *S)
           p2p_ini(pos[2],t,(posreset_yalgn(2)+pos[2])/2,+vyalgn,0.25,100.0,2,S);
      break;
    }

//  initialize controller states
    pid_ini(x,pos);

    return 1;
}



// ------------------------------------------------------------------------
// Hdrive_yalign_out()
// ------------------------------------------------------------------------



int_T Hdrive_yalign_out(real_T *u, real_T *vel, real_T *x, real_T *pos,
  real_T t, int_T contr_id, SimStruct *S)
{
    int_T     i;
    real_T    qref[noa],vref[noa],aref[noa];
    int_T     *piwrk = ssGetIWork(S);
    real_T    *prwrk = ssGetRWork(S);
    real_T    *yPtrs = ssGetOutputPortRealSignal(S,0);

    switch (contr_id) {

      case YALGN_FIND_AVS_B:
           // Get parameters to achieve desired speed
           jog_get(&qref[2],&vref[2],&aref[2],t,2,S);
      break;

      case YALGN_FIND_AVS_A:
```

```
            // Get parameters to achieve desired speed
            jog_get(&qref[2],&vref[2],&aref[2],t,2,S);
        break;

        case YALGN_CENTRE:
            // Get parameters to achive desired position
            p2p_get(&qref[2],&vref[2],&aref[2],t,2,S);
        break;
      }

      qref[0]=posreset_yalgn(0); vref[0]=0.0; aref[0]=0.0;
      qref[1]=posreset_yalgn(1); vref[1]=0.0; aref[1]=0.0;

      //pid_out(real_T *u, real_T *vel, const real_T *xc, real_T *pos,
      //  real_T *qref, real_T *vref, SimStruct *S)
      pid_out(u,vel,x,pos,qref,vref,S);
      // suppres output of amplifiers of x and y1 axis
      //u[0]=0.0; // automatisch omdat x en y1 in positie blijven!
      //u[1]=0.0;

      for (i=0;i<noa;i++) {
         OUT_CTRL_SP(i)=qref[i];
      }

      return 1;
}



// ------------------------------------------------------------------------------
// Hdrive_yalign_synchronize()
// ------------------------------------------------------------------------------



int_T Hdrive_yalign_synchronise(real_T t, int_T sync_id, SimStruct *S)
{
      int_T     iret;

      InputRealPtrsType   uPtrs  = ssGetInputPortRealSignalPtrs(S,0);
//      real_T              *yPtrs = ssGetOutputPortRealSignal(S,0);
//      real_T              *x     = ssGetContStates(S);
//      int_T               *piwrk = ssGetIWork(S);
//      real_T              *prwrk = ssGetRWork(S);


      iret=0;


      switch (sync_id) {


        case YALGN_FIND_AVS_B:
//          as long as avs_b is not seen
            if (IN_AVS(1)==0.0){
              iret=0;
            }
            else {
              iret=1;
            }
```

```
      break;

      case YALGN_FIND_AVS_A:
//          as long as avs_a is not seen
            if (IN_AVS(0)==0.0){
              iret=0;
            }
            else {
              iret=1;
            }
      break;


      case YALGN_CENTRE:
//          as long as Y2 axis is nit centres with respect to Y1
            if (!(p2p_status(t,2,YALGN_TREST,S)==1)) {
              iret=0;
            }
            else {
              iret=1;
            }
      break;

    }

    return iret;
}



// ------------------------------------------------------------------------
// Hdrive_yalign_dif()
// ------------------------------------------------------------------------



int_T Hdrive_yalign_dif(real_T *dx, real_T *x, real_T *pos, real_T t,
  int_T contr_id, SimStruct *S)
{
//   int_T     i;
     real_T    qref[noa],vref[noa],aref[noa];

//   real_T    *yPtrs = ssGetOutputPortRealSignal(S,0);

     switch (contr_id) {

       case YALGN_FIND_AVS_B:
       case YALGN_FIND_AVS_A:
            // Get parameters to achieve desired speed
            jog_get(&qref[2],&vref[2],&aref[2],t,2,S);
       break;

       case YALGN_CENTRE:
            // Get parameters to achive desired position
            p2p_get(&qref[2],&vref[2],&aref[2],t,2,S);
       break;
     }

     // keep X and Y1 axis in position
     qref[0]=0.0; vref[0]=0.0; aref[0]=0.0;
```

```
    qref[1]=0.0; vref[1]=0.0; aref[1]=0.0;

    //pid_dif(real_T *dxcdt, const real_T *xc, real_T *pos, real_T *qref)
    pid_dif(dx,x,pos,qref);

    return 1;

}
```

Appendix C

Hardware

This appendix shows which Inputs/Outputs of dSPACE are used to connect the H-Drive hardware with the software.

Remark with respect to table C.2: Phase T is generated automatically in the current amplifier, based on phase R and S: $cos(R) + cos(S) + cos(T) = 1$

Pin	Axis	Signal	Description
Incl	X	EncX	Output encoder X
Inc3	Y1	EncY1	Output encoder Y1
Inc5	Y2	EncY2	Output encoder Y2

Table C.1: dSPACE - Encoder inputs

Pin	Axis	Signal	Description
		X-Axis	
DACH1	X	XRsetP	Current phase R for LiMMS
DACH2	X	XSsetP	Current phase S for LiMMS
		Y1-Axis	
DACH3	Y1	Y1RsetP	Current phase R for LiMMS
DACH4	Y1	Y1SsetP	Current phase S for LiMMS
		Y2-Axis	
DACH5	Y2	Y2RsetP	Current phase R for LiMMS
DACH6	Y2	Y2SsetP	Current phase S for LiMMS

Table C.2: dSPACE - DA Converters

Pin	Axis	Description	Setting
		Power	
IO0	X	Power off	0 = Power on
IO1	X	Current amplifier enable	0 = Amplifier on
IO2	Y1	Power off	0 = Power on
IO3	Y1	Current amplifier enable	0 = Amplifier on
IO4	Y2	Power off	0 = Power on
IO5	Y2	Current amplifier enable	0 = Amplifier on
		Sensors X-axis	
IO8	X	Overcurrent	Not used
IO9	X	Home EPD-sensor	1 = Sensor activated
IO10	X	EOS-sensor	1 = Sensor activated
IO11	X	OT-Sensor (Over Temperature)	Not activated
		Sensors Y1-axis	
IO12	Y1	Overcurrent	Not used
IO13	Y1	Home EPD-sensor	1 = Sensor activated
IO14	Y1	EOS-sensor	1 = Sensor activated
IO15	Y1	OT-Sensor (Over Temperature)	Not activated
		Sensors Y2-axis	
IO18	Y2	Overcurrent	Not used
IO19	Y2	Home EPD-sensor	1 = Sensor activated
IO20	Y2	EOS-sensor	1 = Sensor activated
IO21	Y2	OT-Sensor (Over Temperature)	Not activated
		Tilt sensors	
IO22		AVS-A Sensor	1 = Sensor activated
IO23		AVS-B Sensor	1 = Sensor activated
		Emergency Button	
IO16		Emergency button	1 = Button hit

Table C.3: dSPACE - Digital IO

Appendix D

C Code for the Kalman method (Software V5.ξ)

This appendix contains a short introduction to the experimental code that uses a Kalman-based zero-search procedure.

D.1 Description of the code

Version 5.ξ of the H-Drive software is based on the code of an old alpha-release of version 4 of the H-drive software and contains merely experimental code to get the Kalman-filter zero-search operational. The code resembles the code that is described in detail in section B.3, but differs at some points:

- Version 5 is not as sophisticated as version 4: there is no code present to handle an emergency stop. Moreover, tilt of the axis is only detected, not corrected for.
- `HD_V5_Work.c` contains the same code as version 4, but now allocates memory for the V5 variables.
- `HD_V4_Movetest.c` with code for a vibrational zero-search procedure is replaced by file `HD_V5_Phi.c` that contains experimental code for a Kalman-filter and a reference generator as described chapter 6.
- `HD_V5_Safety.c` does not support an emergency stop. Also the code for correcting a tilt of the axes during movement is absent.
- The homing-procedure from `HD_V5_Vapi.c` stores the detected epd-position at the end of the sub-state when the epd-sensors of all axes have been detected. Because the axes might move a little after detecting the epd-sensor, this method is not as accurate as the method that is used in `HD_V4_Vapi.c` where the epd-position is stored in a temporary variable as soon as it has been detected for the first time.

When improving the Kalman-filter, the final release of Version 4.0 of the H-drive software should be used.

D.2 Source code

The source code is included on the CD-ROM that is supplied with this report.