The dSPACE system : a development system for fast controller implementation

Kamphuis, P.E.

Award date:
1996

Eindhoven University of Technology
Department of Electrical Engineering
Measurement and Control Group

The dSPACE system

A development system for fast controller implementation

By P.E. Kamphuis

Master of Science Thesis
carried out from March 1996 to October 1996
commissioned by prof.dr.ir. P.P.J. van den Bosch
under supervision of ir. R.J.A. Gorter and W.H.A. Hendrix

Abstract

The Measurement and Control Group at the Eindhoven University of Technology received the development system for control applications from dSPACE gmbh. This system is especially designed for rapid implementation of control structures for testing and tuning. The capability of the system is enlarged by a large set of development tools and function libraries.

The dSPACE system is bought with the objective to provide a power full system for developing high speed controllers demanding high accuracy. With the system it is possible to automatically generate code for a Simulink model of the controller. It is possible to specify the needed I/O capabilities of the dSPACE system in the Simulink model. The generated code is used as the controller application running on the digital signal processor which is the center of the hardware configuration. With the development tools it is possible to alter parameters of the controller even while its is running on the DSP. Giving the control engineer the possibility to adjust the control parameters while it is controlling the desired process. It is also possible to measure and visualize the behavior of variables in time. This data acquisition is performed in real-time at the sample frequency of the running application. The data capturing can be performed in one single capture or free running. It is also possible to perform the data acquisition based on a level triggered base of one of the signals. The capture interval can be specified around the trigger moment making it possible to perform pre- and posttriggering.

In this report the configuration of the dSPACE system is described along with the available development tools. The automatic code generation for designed controllers is also described, this covers also the structure of the generated code and the methods that can be used to extend the functionality of the automatically generated controller implementation.

The whole development process is shown using the design of a master-slave synchronization for two motors. The objective of this control task is to synchronize two load axis each driven by a motor. Both the axis must revolve with the same position displacement of the axis, this implies that both axis must have the same angular velocity. For this control task three approaches are taken, two cases with synchronous controllers and one with an asynchronous controller. The first synchronous controller is used as a reference for the performance of the two other methods. It uses the position encoders of the motors with full resolution. The second synchronous controller uses a limited resolution of the position encoder of the slave motor. The third controller is an asynchronous controller. In this solution is also a limited resolution of the slave position encoder used, but the controller is triggered by the encoder pulses of the slave encoder. This results in a control action that is asynchronous in time. The results of these three solutions for the master-slave synchronization is also presented in this report.

Contents

1 Introduction

It is nowadays quite simple to develop and test a new controller in a simulation environment of computer aided control system design tools. A disadvantage to simulation is the fact that there is always some simplification involved, because it is hard to model every dynamic of the process. Therefore it is beneficial to be able to test the controller on the real process.

In March 1996, the Measurement and Control Group of the Department of Electrical Engineering at the Eindhoven University of Technology received development system from dSPACE gmbh based on a digital signal processor (DSP). This system, hereafter called the dSPACE system, is especially designed for rapid prototyping of applications in the field of control engineering and digital signal processing. It can automatically generate code for Simulink a model to test them in a real process environment.

The advantages of the dSPACE system in developing and testing control applications should be that the control engineer does not need to worry about technical details of the hardware or how his controller must be implemented. The development system generates code for his controller and this code must behave the same as his simulated controller.

In the case that a controller must be built which cannot be generated automatically, it is necessary to have a good description of how to implement this special tasks. To be able to give such a description a good knowledge is required of complete implementation of the dSPACE system. The objective of this thesis is stated as:

Make an analysis of the dSPACE system which make it easier for future users to implement their controllers. Including a description of how to extend the capabilities of the standard system.

The hardware configuration of the dSPACE system is described in chapter 2. It is necessary to know the hardware configuration to be able to determine if the required controller fits in the available hardware, i.e., if there are enough I/O capabilities available of the correct type.

To give the engineer an overview of the tools that are available to him in the development system contains chapter 3 a description of all the software components in the dSPACE development system. In chapter 4, a walk through is given of how to make an implementation of the a Simulink controller model. This chapter includes also a full description of how the implementation is created and which tools and libraries are used for the implementation. Chapter 5 gives a description of three methods to extend the capabilities of the standard generated implementation.

In chapter 6 a master-slave synchronization of the load axis of asynchronous motors is presented which is used as a case study to test the possibilities of the dSPACE system and its development tools. It uses almost all the hardware capabilities of the dSPACE system and it uses the standard development framework extended with all three methods described in chapter 5.

Finally, in chapter 7 some conclusions and recommendations are given on the use of the dSPACE controller development system.

2 The dSPACE system

The dSPACE system provides software and hardware tools for implementing and testing real-time systems such as advanced control systems, hardware-in-the-loop simulations and other high-end signal-processing applications.

The dSPACE system is setup in a way to be suitable and extendible for a large range of control applications. This chapter describes how this system is build up, to give a better view on the possibilities of the system.

2.1 The DS1003 floating point processor board

The DS1003 floating point processor board [1] is the central part in the dSPACE system. It is specifically designed for the development of high-speed multivariable digital controllers and real-time simulators in various fields as

- robotics
- automotive
- drive electronics
- computer peripherals

and is also well suited for general digital signal processing and related tasks. Several peripheral boards provide support for a wide range of applications.

The dSPACE system together with the specialized implementation and code generator tools give the system designer a development package which facilitates rapid system implementation and verification.

2.1.1 Architectural overview

The DS1003 board is built around the Texas Instruments TMS320C40 floating point Digital Signal Processor. It contains local and global memory fast enough for zero wait state operation of the DSP. True dual port memory is included as part of the DSP's global memory to provide fast communication between host and DSP without overhead of bus arbitration.

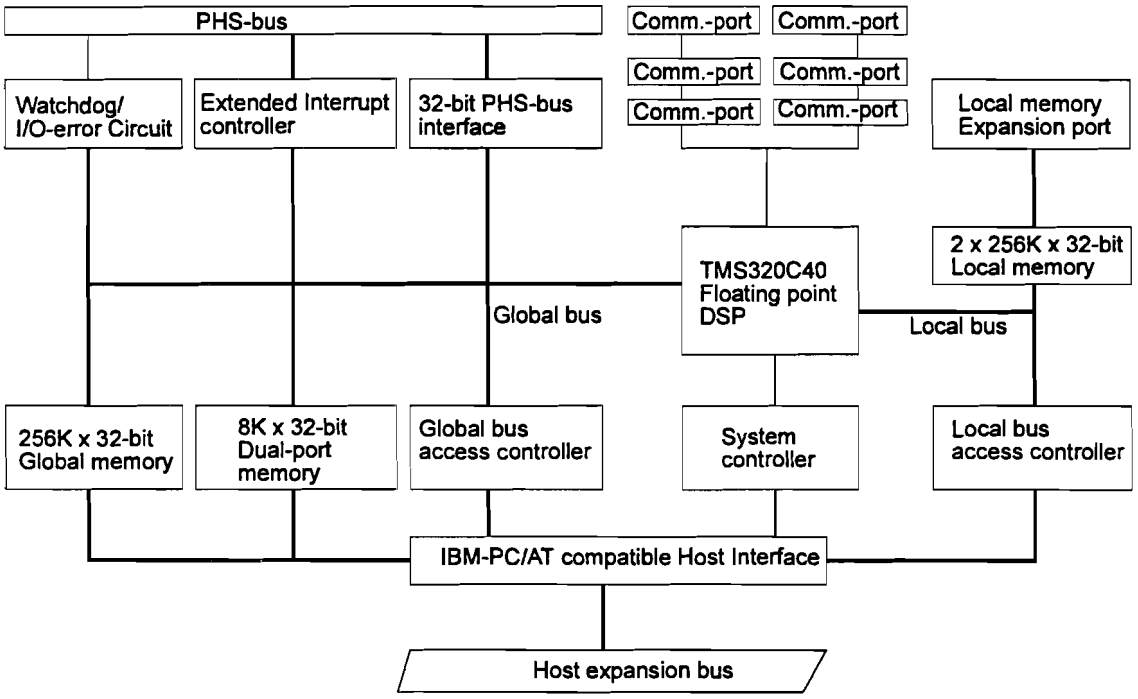

figure 2-1 : DS1003 board block diagram

The TMS320C40 supports a total memory space of 4 Giga 32-bit words containing program, data and I/O space. The external memory space is split into two sections called local and global address space, with separate data and address busses. The global memory address space usually includes the program code and the memory mapped I/O while the local memory contains data.

The DSP's internal interrupt system is expanded by a table-controlled vector interrupt unit supporting up to 64 maskable general purpose interrupts with programmable priority. All system components are accessible by the host even while the DSP is running.

To generate a reference time base the TMS320C40 is provide with two 32-bit timers. These timers can both be connected to an interrupt to be able to run cyclic tasks.

2.1.2 The host to DSP interface

The DS1003 interfaces to the host PC by a block of eight consecutive I/O ports and a 64K or 256K block of shared memory. The I/O interface is primarily used for board setup and to control the DSP while the memory interface supports program downloads and runtime data transfers. A part of the memory interface is implemented as a true dual-port RAM to optimize the runtime data transfer between the host and the DSP. Included on the DS1003 is a bi-directional interrupt port to enable synchronization between the host and DSP programs. This bi-directional interrupt port provides DSP to host and host to DSP interrupts.

2.1.3 I/O interface

The I/O interface between the host computer and the DS1003 consist of a block of eight I/O ports. The host has access to the board's control and status register through these ports.

The block of eight I/O addresses can be placed on one of the following I/O base address 0208h, 0288h, 0308h, 0318h, 0388h, 8308h

table 2-1 : I/O interface registers

offset	access	width	name	function
0	RD/WR	8	STP	setup register
1	RD		STS	status register
1	WR	8	ICR	interrupt control register
2	RD/WR	8	ASR	address select register
3	RD/WR	8	PSR	page select register
4	RD/WR	8	WTR	watchdog timing register
5	RD/WR	8	TBCA	test bus controller address register
6	RD/WR	16	TBCD	test bus controller data register

Setup register

The setup register (STP) controls the various operating modes and states of the control signals of the DS1003 ,i.e., the memory model, the watchdog operating mode and the reset signal of the TMS320C40. It can be read and written to allow the recognition of the board's state and easy changes of single bits of the register.

During initialization the STP register should first be cleared before the appropriate setup is programmed in order to switch the DS1003 into a defined state.

Status register

The status register (STS) provides information about the internal state of the DS1003. It enables the host to monitor I/O-error conditions, DSP-interrupt servicing, and the test bus controller state. It allows to read the configuration bits of the interrupt control register (ICR). The STS register can only be read.

Interrupt control register

The interrupt control register (ICR) selects the interrupt line of the host computer to be used when the DSP generates an interrupt request to the host processor. The selected interrupt line must not be used by other peripheral boards, otherwise the host will fail or the interrupt line driver of the DS1003 can be damaged. The interrupt line select bits and the interrupt enable bit are stored in latches and can be read using the STS register.

The ICR also contains two strobe bits which are used by the host for interrupting the DSP and for acknowledging host interrupts issued by the DSP. Writing a '1' to these bits is sufficient to trigger the desired operation. It is not necessary to turn it on and off.

Address select register	The address select register (ASR) is a read/write register which selects the base-address to be used for mapping the DSP's memory into the 16-Mbytes address range of the PC/AT. The SH (Small/Huge mode) bit in the STP register determines whether a 64-Kbytes or a 256-Kbytes block of memory is used. The base address of this block is selected by the contents of the ASR in 64-Kbytes increments in small mode (SH=1) and in 256-Kbytes increments in huge mode (SH=0).
Page select register	The DSP memory is split into pages of either 16K or 64K 32-bit words (small or huge mode) which are seen by the host as 64 Kbytes or 256 Kbytes pages. To select a particular page the page select register (PSR) is used together with the page select bits in the setup register (STP). The host memory interface maps the desired page in the host memory map based on the address specified in the ASR. To keep the page select operation as fast as possible the host memory interface supports only those sections of the TMS320C40 memory map which are actually used on a fully equipped DS1003. The most significant bit of the PSR determines whether the local or the global bus of the TMS320C40 will be accessed. The local bus is selected if the MSB is 0 and the global bus is selected if the MSB is 1.
Watchdog timing register	The DS1003 contains a watchdog circuit which resets the DSP if it fails to issue watchdog strobes periodically. The watchdog timing register (WTR) determines the maximum time interval between two successive watchdog strobe pulses. If this time elapses and no watchdog strobe has been received the watchdog circuit generates an I/O error signal thus resetting all peripheral boards (if programmed to do so) and the DSP.
Test bus controller	The TMS320C40 features a superset of the IEEE 1149.1 JTAG standard emulation port. This emulation port can be used for hardware testing, in-circuit emulation and software debugging. The DS1003 contains a JTAG test bus controller (TBC) allowing to directly access the TMS320C40's emulation port without the need of additional emulator hardware. Accessing the JTAG test bus controller is performed using two registers, the 8-bit test bus controller address register (TBCA) and the 16-bit test bus controller data register (TBCD).

2.1.4 Memory interface

The TMS320C40 supports two memory sections of two GWords each named global and local memory. Regarding the accessibility by the host and DSP both memory sections are completely identical, i.e., they can both be accessed by the host even while the DSP is running.

The DS1003 does not support the complete address space of the TMS320C40. Each memory section is limited to 8 MWords of address space to reduce the number of I/O operations required for page setting.

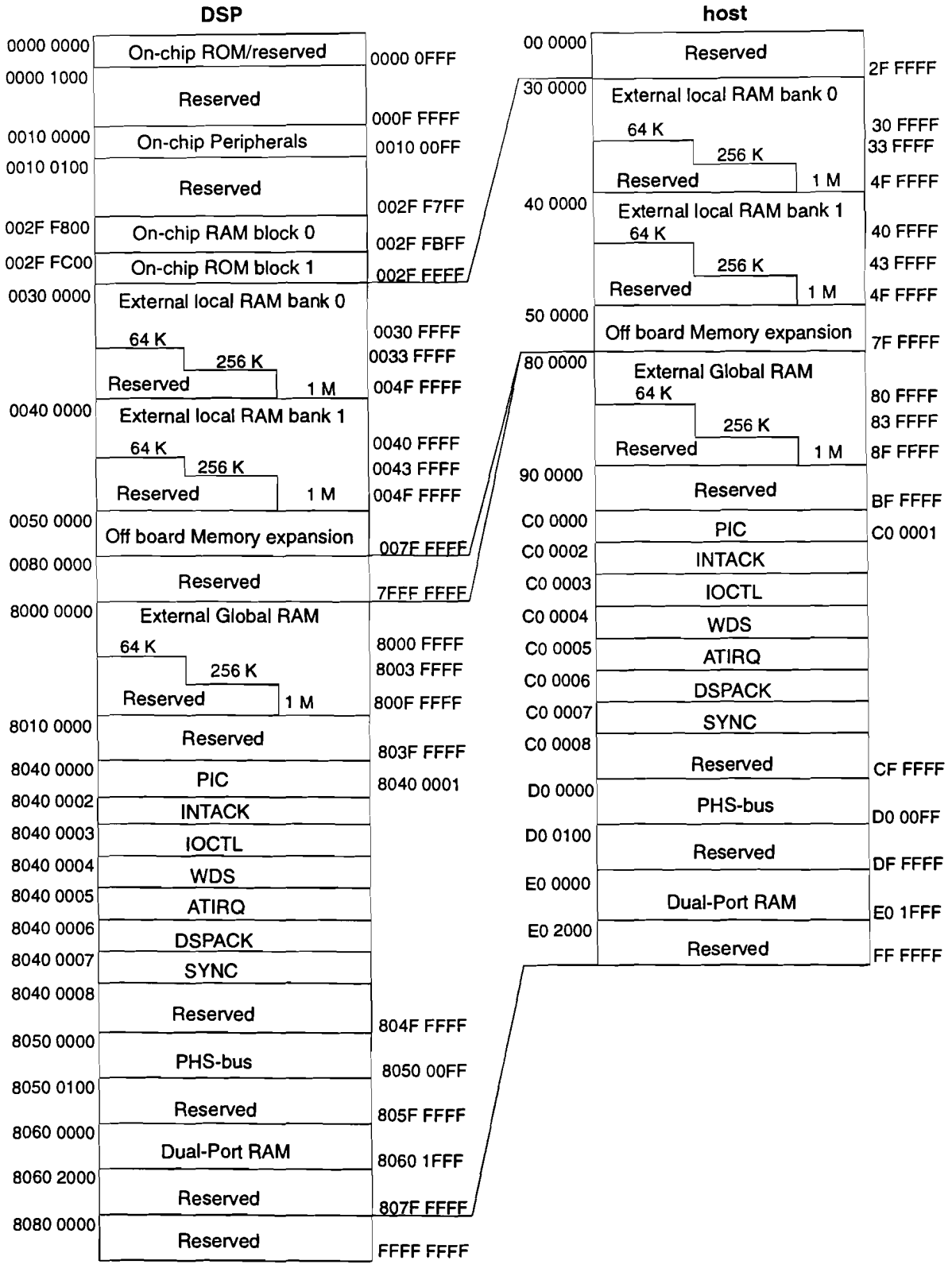

figure 2-2 : DS1003 memory layout

2.1.4.1 Local memory section

The lower 2 GWords of memory is controlled by the local bus of the TMS320C40. All memory connected to this bus is called local memory. This local memory of the DS1003 is always accessible by the DSP and host.

The TMS320C40 DSP provides an on-chip ROM containing a boot loader for autoload applications. In autoload mode the external RAM bank 0 is replaced by a byte-wide EPROM after power up. The DSP can either boot from this EPROM or from a communication port thus allowing stand-alone operation. After the booting has been completed the EPROM is turned off and bank 0 is enabled.

During program development the host controlled mode is more suitable. In this mode the host has complete control over the external DSP memory. User programs can be downloaded, monitored or altered at any time independently of the DSP even while it is running.

2.1.4.2 Global memory section

The upper 2 GWords of memory are controlled by the global bus. Memory accesses to the global section can be performed in parallel to accesses to the local section thus doubling the memory transfer bandwidth. The global bus is intended to support instruction memory and I/O devices.

2.1.4.3 Dual ported RAM

The DS1003 is equipped with 8 KWords of true dual-port memory (DPMEM). This memory has separate data and address busses to the host and DSP thus allowing data transfers without the speed penalty of single bus shared memory systems. The DPMEM is located in the TMS320C40 global memory section and is accessible by the DSP as ordinary data memory.

The dual-port memory contains arbitration logic resolving access contentions which arise when the host accesses the same location the DSP is currently accessing or vice versa. In that case the CPU starting the access later has to wait until the other one has completed its access. The overcome the slow memory access of the host the DS1003 performs the access as fast as possible and stores the required data in intermediate registers. This allows the DSP to resume its operation while the host still has not completed its transfer.

2.1.5 PHS bus

Through the PHS-bus (Peripheral High Speed bus) all the peripheral boards are connected to the DS1003. The PHS-bus is a high speed bi-directional bus allowing direct I/O operations between DSP and peripheral boards without host intervention (see figure 2-3), thus circumventing the limited bandwidth of the PC/AT bus. The PHS-bus supports 32-bit parallel transfers with a maximum transfer rate of 150 ns / 32-bit transfer (200 ns/32-bit transfer on the DS1003).

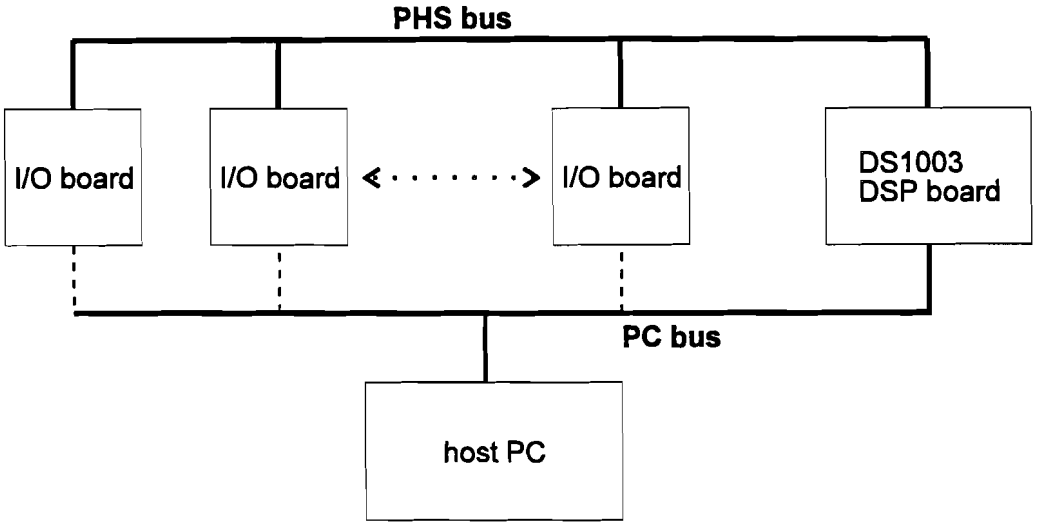

figure 2-3 : Separate PHS bus and PC/AT bus

All boards connected to the PHS-bus either act as bus-masters or bus-slaves. A bus-master contains the central processing unit of the system and controls all activities on the PHS-bus. Within the dSPACE system is the bus-master the DS1003 board. It provides the addresses and control signals required for data transfers and contains the master interrupt control unit of the extended interrupt system. Further more the DS1003 board has an interface to the host enabling the host to access the PHS-bus master.

A bus-slave contains peripherals line ADC's and DAC's or digital interfaces and is completely controlled by the master. Each slave appears on the PHS-bus as a set of registers. A bus-slave may contain a slave interrupt control unit of the extended interrupt system. A bus-slave has expect for power-supply no direct connection to the host-bus.

2.2 I/O boards

The dSPACE I/O boards are specifically designed for the implementation of high-speed multivariable digital controllers and data acquisition systems. All the I/O boards are connected to the DS1003 DSP board by the 32 bit wide PHS bus (Peripheral High Speed bus) to overcome the limited bandwidth of the host.

2.2.1 High-Resolution ADC Board DS2001

The key features of this board are

- 5 fully parallel 16-bit A/D converters
- 5 us conversion time for 16-bit resolution
- Programmable analog input voltage ranges
- Programmable ADC short-cycling

For more information on the DS2001 ADC board see the DS2001 User's Guide [2].

2.2.2 High-Resolution D/A Converter Board DS2102

The key features of this board are

- 6 fully parallel 16-bit voltage output D/A converters
- 1.3 us full scale settling time to 0.025 % (2.0 us with deglitcher enabled)
- Programmable analog output voltage ranges
- Double-buffered and write-through output modes
- Programmable output deglitcher
- Auto-zero function for critical applications

For more information on the DS2102 DAC board see the DS2102 User's Guide [3].

2.2.3 Multi-Channel ADC Board DS2003

The key features of this board are

- Two separate 16-bit A/D converters with 32 multiplexed inputs
- 32 sample & hold circuits allowing to sample all 32 inputs simultaneously
- Programmable scan table with up to 16 entries
- Hardware based conversion sequencer saves DSP time
- Two inputs are converted simultaneously
- Separately programmable analog input voltage range for each input
- Programmable ADC short-cycling for each input channel pair
- 4.2 us conversion time for 16-bit resolution
- Double buffer automatically stores converted ADC values
- Flexible trigger capabilities

For more information on the DS2003 ADC board see the DS2003 User's Guide [4].

2.2.4 Digital Waveform Capture Board DS5001

The purpose of the DS5001 is to sample digital waveforms like pwm outputs or position detector signals. These waveforms can be described as a series of rising and falling edges and the corresponding times (timestamps). The capture function of the DS5001 can be seen as an event driven transient recorder, which only stores data when at least one input signal changes. The DSP which controls the DS5001 reads the timestamps and performs any complex signal analysis the user wants to have. So the tasks are clearly separated: The DS5001 does the data acquisition and the host does analysis.

The DS5001 key features are:

- 16 identical comparators / edge detectors featuring
  - sample rate 25 ns
  - rising and/or falling edge detect programmable
  - trigger level programmable from -10V to +10V
  - input FIFO for fast bursts
- Event storage featuring
  - 512 events (timestamp + edge polarity) per input channel
  - 31 bit timestamp, timebase is common for all input channels.
    - Time stamp period is $2^{31} * 25$ ns = 53.68 s
  - Address generator supporting FIFO and LIFO like readout
  - Interrupts can be generated after a predefined number of events
- 3 additional event counters featuring
  - 32 bit width
  - each counter can be assigned to any edge detector output
  - independent from input FIFO and event storage

For more information on the DS5001 Digital Waveform Capture Board see the DS5001 User's Guide [5].

2.2.5 Expansion Box Interface Boards DS811/DS812

From the two Expansion Box Interface Boards [6] one is placed inside the host computer and the other in the expansion box. The expansion box is an extension of the PC/AT bus with its own power supply. The advantages of a separate box is that the dSPACE boards do not need to be powered by the host computer. The two expansion box interface boards provide the interface mechanism for the two busses.

note: The DS811/DS812 extended memory mapping is according to dSPACE gmbh not supported. This implies that system can only be used in the Base Memory Mapping.

2.3 The dSPACE hardware installation

For the hardware installation of the dSPACE system the defaults are used. If the system needs to be reinstalled on an other computer, it is necessary to check if these default I/O range and memory range do not conflict with any I/O boards (e.g. a ethernet card) in that computer. If necessary the memory and I/O settings of the DS811 Interface board need to be changed, as well as the I/O range setting on the DS1003 DSP board.

For the used host computer a few adjustments must be made as well. the used memory segment must be excluded for the memory manager in the config.sys. This ensures that the segment is not used as a memory segment for other PC applications. Further more the caching of the memory segment must be turned of in order to be sure that values read and written are really from the DSP memory. This is usually done in the BIOS setup of the computer, in worst case the system memory caching must be turned of completely.

Besides these settings and adjustments the software protection key (dongle) must be connected to the printer port (LPT1). With out this protection key several dSPACE software components do not work.

Refer to the Hardware Installation Guide [7] and the different hardware guide's for proper installation of all the system components.

2.4 summary

The dSPACE system provide in a system that is very suitable for control applications. The TMS320C40 digital signal processor is fast enough to be able to perform even the larger control tasks. The 32-bit timers provides the controller with a correct time base for sampling instances. The extended interrupt system of the DS1003 board provides the necessary interrupt sources to perform interrupt driven control tasks.

The host-DSP interface which gives the host full control of the DSP memory is very useful when testing controllers. It gives the possibility to update a control algorithm very fast and to monitor or change any variable in the program.

The PHS-bus provides a way to connect the peripherals to the DSP. This PHS-bus is a fast separate bus and overcomes the transfer bandwidth of the host PC-bus. While keeping the possibility for the host to access the peripherals directly.

The peripheral boards provide all the necessary I/O for control systems. The 16-bit AD- and DA-converters are fast and precise enough to be useful for any control task. Digital input possibilities are provide through the Digital Waveform Capture board. This board has capabilities to detect any waveform for signal analyses and encoder reading.

2.5 Literature

[1] DS1003 Floating-point Processor Board
User's Guide
document version 1.0
dSPACE gmbh

[2] DS2001 High-Resolution ADC Board
User's Guide
document version 1.1
dSPACE gmbh

[3] DS2102 High-Resolution DA Converter Board
User's Guide
document version 1.1
dSPACE gmbh

[4] DS2003 Multi-Channel ADC Board
User's Guide
document version 2.0
dSPACE gmbh

[5] DS5001 Digital Waveform Capture Board
User's Guide
document version 1.0
dSPACE gmbh

[6] DS811/DS812 Expansion Box Interface Boards
User's Guide
document version 2.0
dSPACE gmbh

[7] Hardware Installation Guide
for DS1002 and DS1003 Systems
document version 3.0
dSPACE gmbh

3 The dSPACE software components

One of the key features of the dSPACE system is the ability to automatically build applications. In that case the dSPACE Real-Time Interface (RTI) connects the design and analysis tool Matlab, Simulink and the Real-Time Workshop (RTW) with the dSPACE's real-time system. This gives the control engineer an integrated and ready-to-use environment for real-time applications that allows him to focus on the development and design of the controller. The software modules include automatic code generation facilities, real-time analysis tools (**TRACE**), and GUI's for interaction with the experiment (**COCKPIT**). This results in what could be called the basic development system.

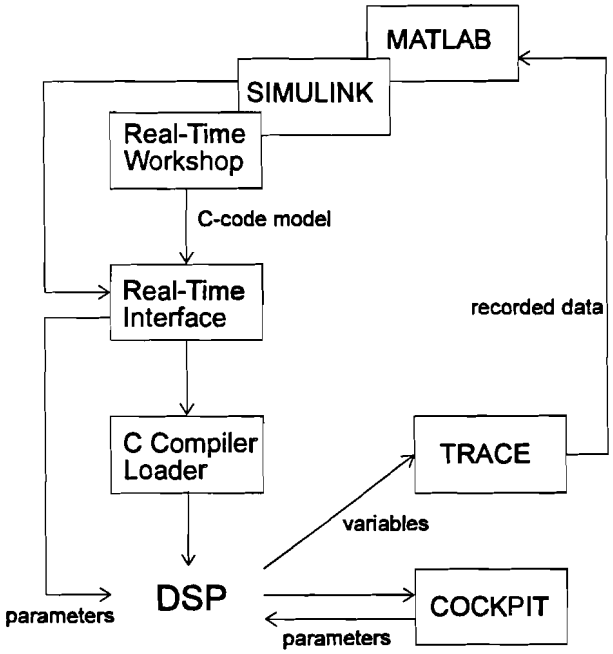

figure 3-1 : Basic development system

By using this development system the control engineer only needs to focus on the controller design. When the controller design is finished the automatic program building feature takes care of the implementation of the controller on the processor. After which the controller can be tested in a practical environment. The control engineer can read and modify variables and parameters of the controller while the program is running using the **TRACE** and **COCKPIT** tools.

To provide the control engineer with this development system several other software components are needed. The software that comes with the dSPACE system can be divided into four sections.

First the basic dSPACE software, these are programs to use and control the behaviour of the dSPACE system.

Secondly the dSPACE Windows development tools. These tools implement the basic development system and provide the control engineer with the easy to use development system.

Then there are the interface libraries for Matlab. With these interface libraries it is possible to access the DSP and the program that is running on the DSP from with in Matlab.

Finally there are some additional interface and development libraries. The interface library gives the possibility to access the DSP from user written DOS and Windows programs. While the development libraries provide additional functionality to DSP programs.

3.1 The basic dSPACE software

3.1.1 The DSP device driver

The DSP device driver is intended to coordinate access of the applications to the dSPACE signal processor board DS1003. The host interface of the DSP board executes each command immediately and is not aware of that it is accessed by different applications. The DSP device driver is a software shell that encapsulates the host interface of the DSP board. It provides its own user interface for the application programs that substitutes the board's host interface.

The device driver allows simultaneous accesses by several host applications to a single DSP board. It serializes theses accesses and suspends a host application temporarily in situations where a board access must not be interrupted by another program trying to perform an access to the same board. For this reason the device driver provides a mechanism to lock the board by an application program, that means, getting the exclusive access rights for the board.

The device driver improves safety checks because it keeps track of the DSP board's state. It makes it possible for host applications to inquire the name and the creation time of the DSP program currently running and the memory equipment for the DSP board.

3.1.1.1 The DSP device driver for MS-DOS

The DSP device driver DSPDRV.EXE is a so called Terminate and Stay Resident (TSR) for MS-DOS. It can be invoked directly from the DOS command prompt or from AUTOEXEC.BAT. Once started it stays resident in PC memory and can be used by application programs to access the DSP board.

DSPDRV.EXE is intended to work under pure DOS and not on DOS boxes of MS-Windows. For use under MS-Windows a virtual device driver must be installed.

3.1.1.2 The DSP device driver for MS-Windows

The DSP device driver VDSPDRV.386 is a virtual device driver for the 386 Enhanced Mode of MS-Windows. To install this driver for MS-Windows, it must be added to the section [386Enh] of the SYSTEM.INI file in the WINDOWS directory.

3.1.1.3 Controlling the device driver

In normal operation the DSP device driver is only used by application programs and not directly by the user. But when necessary there is a program available, called DCONT.EXE, to control the operation of the device driver.

This program has to initialize the device driver after it was loaded. The program DCONT.EXE reads the information about the DSP hardware equipment (the system setup) from a file named DSPACE.INI and passes this information to the device driver.

The program DCONT.EXE can be called with several command line options:

/a	initialize all DSP processor boards specified in the initialization file DSPACE.INI
/b board	initialize DSP processor board with name board, an entry with that name must already exist in the initialization file DSPACE.INI
/d board	delete entry for DSP processor board and all entries for the peripheral boards attached to it from the file DSPACE.INI
/l	lists all processor boards and host applications currently registered at the device driver
/r	reset the device driver
/?	display short help about available options

Initialization has to be done each time after the device driver is loaded, and before any program accesses the DSP processor board.

If only the device driver for MS-DOS is installed, the command to initialize it with DCONT.EXE can be placed in the AUTOEXEC.BAT after the command to load the device driver.

If the device driver for MS-DOS as well as the device driver for MS-Windows is installed, initializing can also be done from AUTOEXEC.BAT. The device driver for MS-Windows does not need a separate initialization because the device driver for MS-Windows and the device driver for MS-DOS exchange their data each time MS-Windows starts or exits.

If the device driver for MS-DOS is not installed, but the device driver for MS-Windows is installed, the initialization can be performed at the start of MS-Windows by placing the call to DCONT.EXE into a batch file named WINSTART.BAT. This file must be located in the WINDOWS directory. If this batch file exists, MS-Windows runs it automatically when starting. Installing the device driver for MS-Windows without the device driver for MS-DOS saves about 20 Kbytes of DOS memory.

3.1.2 The Setup Editor SED40

The setup editor program SED40 is used to create and edit the global system setup file DSPACE.INI and the application specific setup file (.stp) for hand coded programs. It also contains menu selections which serve for verifying presence of a DS1003 processor board and for displaying a list of all peripheral boards being connected to the processor board.

The setup editor utility can conflict with the normal operation of the DSP hardware system. It cannot be executed while the DSP is running. If SED40 could detect a DS1003 board due to the information found in the system setup file DSPACE.INI, it checks whether the processor is running. If it is running the user is prompted to confirm that SED40 may reset the processor. If this is not confirmed SED40 will abort.

3.1.2.1 Edit system setup

The global system setup file must be updated each time a board is added to or removed from the system or if another change to the global operating conditions is required, e.g. watchdog operation of the processor board.

The system setup file can be generated or edited, if it is already present, by selecting the appropriate choice of the main menu. The editor will supply default setup for each board it can detect in the computer. This will always be at least a processor board and then, depending on the configuration, all the peripheral boards connected to it.

3.1.2.2 Application specific range setup

The application specific setup files contain information only about the input and output channel ranges of the channels used by the particular application. This file is always needed to download an object file into the DSP.

Entering the submenu in order to create an application specific setup file requires the specification of a filename prefix for the file this information shall be written to. This prefix must be identical to the prefix of the corresponding object file containing the DSP object code. The filename suffix .stp will be supplied automatically.

3.1.3 The DS1003 processor board monitor MON40

The monitor program MON40 performs the actual loading of the system setup file and of DSP object code modules and their corresponding application specific setup files. The monitor program provides the option to control the DSP operation by accessing its RESET signal. It supports operation of the watchdog timer if this component is enabled via the system setup. MON40 also allows to access peripheral board registers by the PHS-bus and to initialize I/O-channels.

The monitor program can be used in command mode and in menu mode. The command mode is invoked if an object module is specified in the command line.

3.1.3.1 Command mode

In the command mode the monitor program evaluates the system setup file dspace.ini, the application specific setup file (.stp) and loads the appropriate setups into the board. It will load the object module, restart the DSP, and return to the DOS command level immediately. If an error is detected the DSP will not be started. This is the fastest method to execute new applications and can be used in batch files.

3.1.3.2 Menu mode

To run the monitor program in the menu mode it requires the DSP to be reset. When loading the system setup it prompts for confirmation to allow resetting the DSP if the DSP is running. If this is not confirmed the program aborts immediately.

If the system can be setup properly the monitor is ready to load application programs into the DSP's memory by selecting the appropriate menu choice.

Load object module	When the main choice to load an application program is selected, the monitor program prompts for a filename prefix. The monitor program adds the suffix .stp and expects to find a setup file with this name. Having successfully read the setup file and setup the hardware the monitor adds the extension .obj to the filename prefix and tries to open a file with the resulting name. If the object file can be read correctly it is downloaded to the DSP board's memory and may then be started by selecting 'Restart DSP' from the main menu.
Restart DSP	After an object module is successfully downloaded this option can be used to start the DSP. In case the DSP is already running, this choice will first reset the DSP and then release its RESET signal again causing the application program to restart.
Reset DSP	The DSP is put into the RESET state and remains in this state until it is either restarted by using the Restart DSP option or if an application is loaded by using the monitor in command mode.
Clear I/O channels	When the DSP is reset, the monitor can be used to initialize I/O channels and the processor board's memory.
Read/write PHS-bus	When the DSP is reset, the monitor can be used to access peripheral board registers. The addresses used are interpreted as 32-bit PHS bus addresses.
Watchdog control	If the system setup file specifies use of the processor board's watchdog the monitor will support it. It will disable the watchdog as long as the DSP is reset and reinitialize the watchdog hardware by zeroing the watchdog bit. Activating the DSP by invoking the monitor's restart option causes the monitor first to initialize the watchdog timer with the selected period value. Then it selects the appropriate mode (shutdown / retry) and enables the DSP by releasing the appropriate signals. Finally it enables the watchdog. The watchdog will become active after the DSP has issued the first watchdog re-trigger pulse by writing to the appropriate address.

3.1.4 Down40.bat

The batch file down40.bat is available to compile and download user written programs for the DSP.

3.1.5 Error check program CHKERR40

The chkerr40 program checks the DSP error variable and signals the user with an appropriate error message when necessary. The error variable is defined by dSPACE as the last address of the dual ported RAM section. This variable can be used in applications to flag an error situation in the DSP program.

3.2 The dSPACE Windows development tools

The most important development tools in the dSPACE system are Windows based applications. These applications form together a useful development system for designing and testing controllers. One is not really an application, but an extension for the code-generation capabilities of Matlab and Simulink. This is called the Real-Time Interface (RTI). The two other applications are support tools. They provide real-time data tracing and parameter tuning of a controller under development. Besides these tools, there are two Matlab interface libraries and a debugger.

3.2.1 Real-Time TRACE module

TRACE provides real-time trace capabilities for any application running on the DS1003 DSP board. TRACE allows to record and graphically display all signals and parameters represented as single-precision float or integer variables in the processor's memory. TRACE thus provides insight into any application while the data acquisition is executed in real-time by the DSP.

- TRACE features include
- free-running or level-triggered mode
- pre- and post-trigger
- distinction between float and integer variables
- distinction between static and dynamic data
- storing traced signals on disk and reloading any signals for comparisons
- variable trace capture sampling rate (downsampling)
- automatic display of the traced signals
- auto trace (automatic repetitive trace captures)
- data export in Matlab binary format or in ASCII format
- endless trace capturing without data gaps for slow processes
- freely configurable order, size, and position of plots
- multiple plot windows, each of which contains a layout which may be stored to disk and loaded again later
- presentation of signals as Yt or YX plots
- several data cursors (crosshair, deltaX, deltaY, deltaXY) for analysis
- data import and export in several data formats including an automatic experiment description and user written text

3.2.1.1 The implementation of TRACE

Full real-time capability during data acquisition is only achieved if the DSP itself is used for this task. Therefore TRACE downloads a piece of service code to the DSP board. This service code collects a set of data once for every sampling step into a buffer on the DSP board. This service routine is copied to the DSP board as soon as a new trace capture is requested by pressing the START TRACE button.

TRACE allocates a piece of DSP memory from the device driver and download its service code (figure 3-2).

The service code is only invoked if the DSP program has been prepared for use with TRACE. This means that the function service_trace(); must be added to the interrupt service routine. This function call is a C-macro defined in BRTENV.H containing a TRAPU instruction. This TRAPU instruction generates a software interrupt. While TRACE is idle, this software interrupt returns directly without any effects; as soon as a capture has been started, the TRAPU instruction calls the TRACE service code.

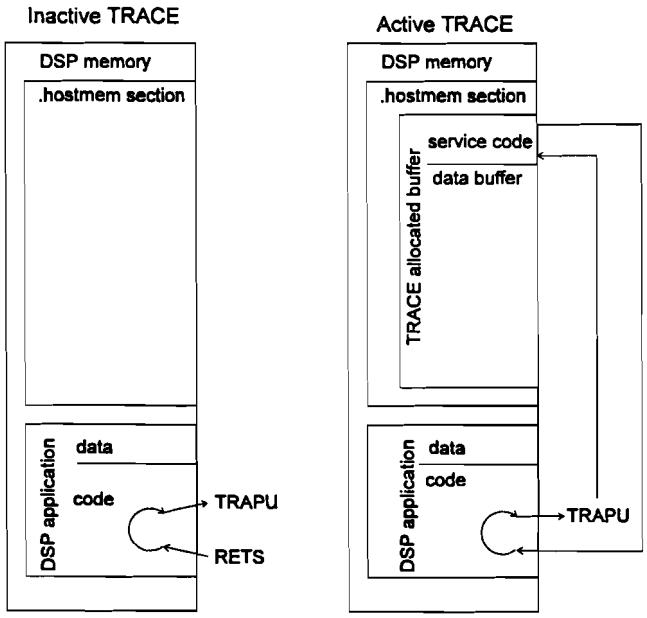

figure 3-2 : TRACE implementation on the DSP

Because the data acquisition is done by the DSP, sufficient buffer space for the collected data must be available on the DSP board. The location of the service code and the buffer is controlled by the DSP device driver. The TRACE program allocates the required DSP memory from the device driver and uses this area for data storage. If there is not enough memory left or the data allocation fails for any reason TRACE does not start a capture and displays an error message.

The device driver is responsible for the DSP memory allocation of all host applications.

3.2.1.2 TRACE input and output files

TRACE needs several input containing information about the currently running DSP program as well as some configuration files (figure 3-3). Some of these files are global files existing only once in the environment, others contain application specific information. TRACE also creates some output files to store diverse information.

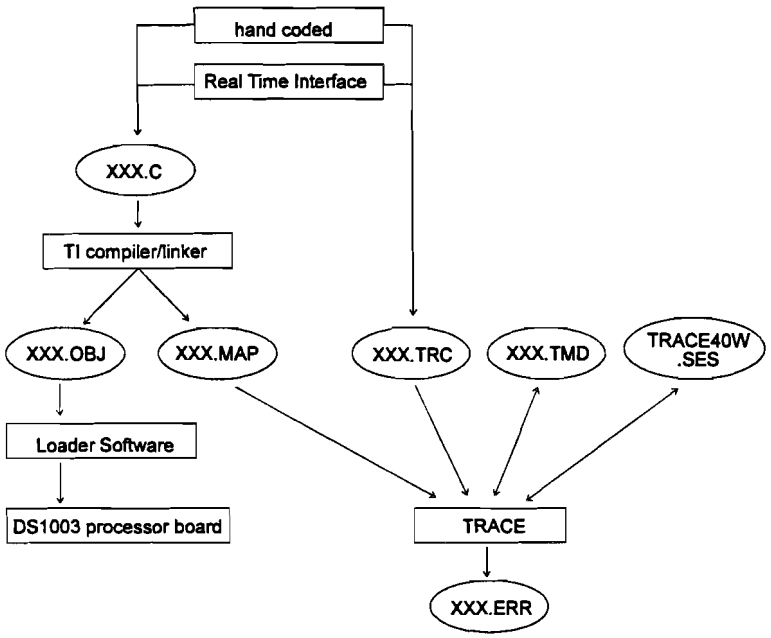

figure 3-3 : TRACE file relations

TRACE needs at least two input files specific for each DSP application. These are the trace file (.TRC) and the TI linker map file (.MAP). These files must be seen as a union, because they are interpreted together. They must exist in the same directory with the same prefix as the DSP program's object file.

The trace file describes which variables of the DSP program are intended to be traced. The addresses of these variables are retrieved from the linker map file.

If an error occurs while parsing the .TRC or the .MAP file, an error file will be created. It will be placed in the same directory and with the same prefix as the trace file, but with the extension .ERR.

3.2.2 COCKPIT Instrumental Panel

COCKPIT is an instrument panel, which provides graphical output and interactive modification of variables, for any application running on the DS1003 digital processor board. It is possible to modify or display all variables represented as single precision floating point or integer variables in the processors board's memory. Output is performed with instruments like digital displays, tachometers, or moving bars. Variables can be modified with sliders, various buttons, or numeric input from the keyboard.

The COCKPIT program is not intended to be used for real-time analysis purposes like the dSPACE TRACE module, i.e. DSP access is not performed at fixed time steps.

3.2.2.1 The implementation of COCKPIT

For accessing all memory areas of the signal processor without restrictions, some service code running on the DSP is required. This piece of service code is downloaded to the DSP board as soon as the animation mode of COCKPIT is started. It has the task to copy the contents of all required variables into a small buffer in the DSP memory. This buffer is then read as one block of data by COCKPIT. So the communication overhead between the host and the DSP board is minimized.

The device driver is responsible for the DSP memory allocation of all host applications.

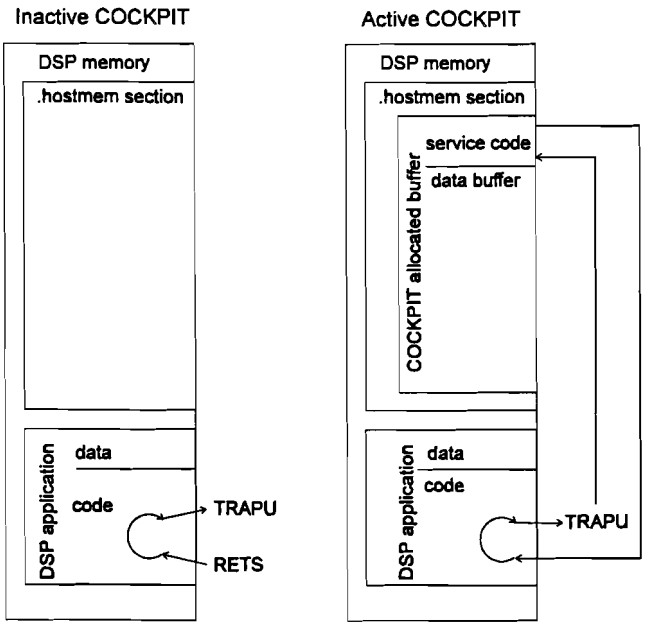

figure 3-4 : COCKPIT implementation on the DSP

3.2.2.2 TRACE input and output files for COCKPIT

The COCKPIT program needs several input containing information about the currently running DSP program as well as some configuration files (figure 3-5). Some of these files are global files existing only once in the environment, others contain application specific information. COCKPIT also creates some output files to store diverse information.

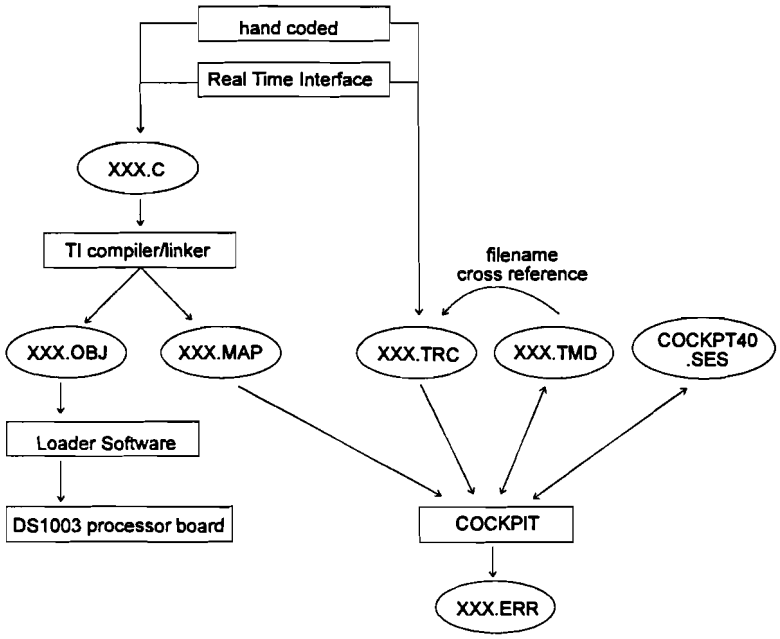

figure 3-5 : COCKPIT file relations

COCKPIT needs at least two input files specific for each DSP application. These are the trace file (.TRC) and the TI linker map file (.MAP). These files must be seen as a union, because they are interpreted together. They must exist in the same directory with the same prefix as the DSP program's object file.

The trace file describes which variables of the DSP program are intended to be traced. The addresses of these variables are retrieved from the linker map file.

If an error occurs while parsing the .TRC or the .MAP file, an error file will be created. It will be placed in the same directory and with the same prefix as the trace file, but with the extension .ERR.

3.2.3 DEBUG40W: C Source Debugger for DS1003 Boards

The DEBUG40W is the Texas Instrument C source debugger for C40 processor boards, ported to dSPACE DS1003 boards in a PC/AT running under Windows 3.1 or newer versions.

3.2.4 Real-Time Interface to Simulink

The dSPACE Real-Time Interface (RTI) connects the MathWork's development software Matlab, Simulink, and the Real-Time Workshop (RTW) with dSPACE's real-time system to form an integrated development environment for real-time applications. It provides for an automatic and seamless implementation of Simulink graphical models on dSPACE real-time hardware systems for hardware-in-the-loop simulation and controller prototyping. Any kind of Simulink model for which code can be generated by means of the RTW can be implemented by the RTI, including continuous-time, discrete-time, and hybrid systems.

3.3 Interface libraries for Matlab

3.3.1 MTRACE: Real-Time TRACE module for Matlab

The Real-Time Trace module for Matlab (MTRACE) provides real-time data capture capabilities for the dSPACE DSP processor boards.

MTRACE functions can be called directly from M-files. Within the dSPACE environment for rapid control prototyping, MTRACE can be used for fast real-time data acquisition.

MTRACE features include:

- free running or level-triggered mode
- pre- and post-trigger
- distinction between float and integer variables
- adjustable trace capture sampling rate (downsampling)
- direct data transfer to Matlab workspace
- multiple service routines per DSP application

For more information on the MTRACE library see the MTRACE User's Guide [13].

3.3.2 MLIB: Matlab-DSP Interface Library

The Matlab-DSP Interface Library (MLIB) allows access to the dSPACE DSP hardware from within Matlab's workspace.

The MLIB library provides basic functions for reading and writing data to the dSPACE processor boards. Functions for generating interrupts, setting the processor state, and getting processor status information by the host also provided.

The MLIB function are suited to modify parameters in the DSP program currently running, or to send test sequences to the DSP application. There is no need to know the addresses of DSP variables.

For more information on the MLIB library see the MLIB User's Guide [11].

3.4 Additional interface and development libraries

3.4.1 Filter Library

The Filter Library (FLIB40) comprises C functions for filter algorithms such as

- lowpass filters
- highpass filters
- bandpass filters
- bandstop filters
- allpass filters

It also includes functions for filter initialization and on-line parameter modification.

3.4.2 Signal Generator Library

The Signal Generator Library (SLIB40) comprises C functions for generating various periodic waveforms such as

- sine-wave signals
- square-wave signals
- pulse-train signals
- triangular signals
- sawtooth signals
- raw and filtered PRBS noise

It also includes functions for signal parameter initialization and on-the-fly parameter modification.

3.4.3 Host-DSP Interface Library CLIB

The CLIB library provides a set of basic functions to simplify control of the processor board and access to the DSP memory. The CLIB library is available in versions for the Microsoft C and the Borland C++ compiler, for development of either DOS or MS-Windows applications. The required version can be selected during software installation.

Refer to the Host-DSP interface Library User's Guide

[12] for further information on installation and use.

3.4.4 TextIO for DSP Boards

The TextIO library provides a set of DSP functions permitting communication of the DSP program with the host PC's screen and keyboard. In the DSP program standard C-library functions like printf() can be used to generate formatted output, which is displayed on the host PC's screen with a terminal like server program. Likewise the PC program can accept key-strokes and communicate them to the DSP which can evaluate them using scanf(). The library functions are useful for debugging of DSP programs, and for reading and writing data in cases where TRACE and COCKPIT are not sufficient. When used properly these functions are real-time capable which permits their use i.e. in hardware-in-the-loop simulation systems where a normal debugger can not be used.

The PC server program is provided in two versions. A pure DOS application TERM40 and a true Windows application WTERM40.

3.5 The dSPACE software installation

This section describes the action needed in order to install the dSPACE software components correctly on the host computer. In order to be able to use the different applications a license file (**license.dsp**) must be created in the directory **\dsp_cit\citfiles**. This file must be created by hand an filled with the license registration codes provided with the software.

note: All dSPACE application can easily be installed using the installation programs on the disks. There is just one thing to keep in mind. The Windows applications should be installed from Windows. The application with a DOS installation program should be installed from DOS. Installation ofthese applications from Windows failed.

note: The Windows application TRACE and COCKPIT require the Windows DLL file **wkwin.dll**. This DLL-file is expected in the windows system directory. In the currently installed version this DLL-file is copied in the **\dsp_cit\exe** directory. This results in a warning message each time TRACE or COCKPIT is started.

3.5.1 Minimum installation

The installation of the dSPACE system starts by installing **Base Vs 1.0**. This disk contains the dSPACE device drivers for DOS and Windows and batch files for settings and loading the device driver. Before the dSPACE boards can be used, the device driver must be loaded. Two batch files that accompany the dSPACE software take care of loading the device driver.

dspace.bat initialization of the environment settings
drvinit.bat driver initialization

For use of the dSPACE boards within Windows the following line has to be appended to the entry [386Enh] in the SYSTEM.INI file.

device=C:\dsp_cit\exe\vdspdrv.386

The device drivers and the batch files are installed in the directory **\DPS_CIT\EXE**.

In order to load applications in the DSP the DS1002/DS1003 Setup Editor and Monitor (**SED40/MON40 Vs 5.2**) must be installed. The setup editor is also needed to complete the hardware installation. It is used to check if the whole systems is installed correctly. The setup editor and monitor are installed in the directories **\DSP_CIT\EXE** and **\DSP_CIT\citfiles**.

The key part for the code generation of the dSPACE system is the Texas Instruments TMS320 C compiler / assembler / linker tool set. When the version 2.4 of the Real-Time Interface is used, release 4.6 of the compiler / assembler can be used. The Texas Instruments diskettes contain a directory DOS32. The files from these directories must be copied to the directory **\C30TOOLS** in order to install the compiler.

Before any application program can be successfully compiled and linked the Parallel Runtime Support Library PRTS40.lib must be built. (General Software Installation Guide). This can be done by running the following command in the **\c30tools** directory.

```
mk30 -v40 --h -o2 prts40.src
```

The specified options mean the following:

-v40 C40 object code
--h extract all header files contained in the source library, some of them are needed by the dSPACE standard software library
-o2 maximum optimization

These steps cover the minimal installation needed to use the dSPACE system. All the software components in the next sections only need installation when they are needed.

3.5.2 dSPACE Windows development tools

The Real-Time Interface for Simulink can be installed using the disk labeled **RTI40 Vs.2.4.1**. The installation is started by running winstall.exe from within Windows. The different components of the RTI are installed in the following directories

\dsp_cit\matlab\rti\dslib

\dsp_cit\matlab\rti\mask

\dsp_cit\matlab\rti\sfcn

\dsp_cit\matlab\rti\tools

\dsp_cit\matlab\rti\dempx0

The TRACE program is installed by running **winstall.exe** on the disk labeled **TRACE40W Vs. 3.0.1.** The TRACE application will be installed in the following directories

\dsp_cit\exe

\dsp_cit\doc

\dsp_cit\citfiles

\dsp_cit\demo1003\std

The COCKPIT program is installed by running **winstall.exe** on the disk labeled **COCKPIT40 Vs. 3.0**. The COCKPIT application will be installed in the following directories

\dsp_cit\exe

\dsp_cit\doc

\dsp_cit\citfiles

\dsp_cit\demo1003\std

The debugger can be installed from the disk labeled **DBG40W Vs. 1.0** by running **winstall.exe** from this disk. It will be installed in the next directories.

\dsp_cit\exe

\dsp_cit\citfiles

\dsp_cit\demo1003\debugger

3.5.3 Additional interface and development libraries

The CLIB Host-DSP interface library can be installed by running **install.exe** on the disk labeled **CLIB Vs 3.0** from the DOS command prompt. Running the installation program installs the interface library in the directories.

\dsp_cit\clib

\dsp_cit\demo1003\clib

The library functions for the filter and signal generator libraries are installed by running **install.exe** on the disk labeled **SFLIB40 Vs 1.1**. The libraries and demos will than be installed in the directories

\dsp_cit\c40

\dsp_cit\demo1003\sflib

In order to link these libraries into the generated program the following bold printed lines must be added to the linker command file (DS1003.LNK).

```
-u startup
-l c:\dsp_cit\c40\slib40.lib
-l c:\dsp_cit\c40\flib40.lib
-l c:\dsp_cit\ds1003.lib
```

This will make sure that, if needed, the appropriate functions are linked.

The TextIO library can be installed by running install.exe from the disk labeled **TextIO40W Vs 1.01**. The library and terminal application will than be installed in the directories

\dsp_cit\c40

\dsp_cit\exe

3.5.4 Interface libraries for Matlab

The two Matlab interface libraries MTRACE and MLIB are provided on the disk labeled **MLIB Vs 1.11** and **MTRACE40 Vs 1.01**. Both can be installed by running **winstall.exe** from the appropriate disks.

The MTRACE interface library is installed in the directories:

\dsp_cit\matlab\lmtrace

\dsp_cit\matlab\local

\dsp_cit\demo1003\mtrace

The MLIB interface library is installed in the directories:

\dsp_cit\matlab\lmlib

\dsp_cit\matlab\local

\dsp_cit\demo1003\mlib

3.5.5 Resulting directory structure

Installation of all the dSPACE software components results in the creation of the directory structure of figure 3-6.

note: This directory structure is the result of a complete installation and several updates. Therefore a directory **\dsp_cit\matlab_dslib.old** is also created with a backup of the \dsp_cit\matlab\rti\dslib directory. When the previous described installation procedure is followed this directory will not appear.

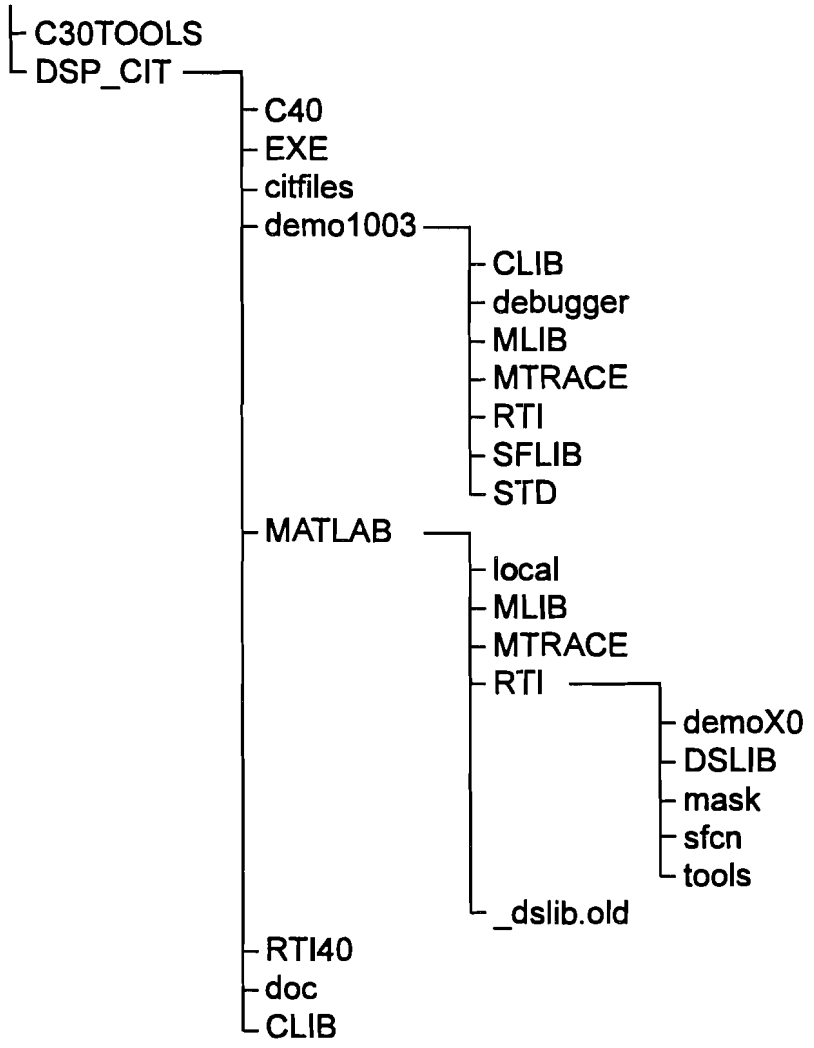

figure 3-6 : directory structure for the dSPACE software

3.6 Summary

The software provided with the dSPACE system makes a very complete development system for developing control systems. There are basic initialization programs available for setting up the dSPACE hardware and loading applications.

Function libraries are provided for developing low level applications with signal generators and filters.

The Windows development system is a complete developing environment with automatic code generation for Simulink models and tools for monitoring, testing and debugging the build applications.

There are also Matlab libraries available to gain access to the DSP application directly from Matlab.

All together a development system is available suitable for about any application that needs to be build. Especially the Windows development tools are very useful and easy to work with.

3.7 Literature

[1] DS1003 Software Environment
Reference guide
Document version 3.1
dSPACE gmbh

[2] Real-Time Interface to Simulink
Release Notes
Document version 2.4 for software version 2.4
dSPACE gmbh

[3] COCKPIT Instrument Panel
User's Guide, Reference Guide
Document version 3.1 for software version 3.0
dSPACE gmbh

[4] Real-Time TRACE Module
User's Guide, Reference Guide
Document version 3.0 for software version 3.0
dSPACE gmbh

[5] Real-Time Workshop
User's Guide
May 1994
The MathWorks Inc.

[6] Simulink 1.3
Release Notes
1994
The MathWorks Inc.

[7] Filter Library
Document version 1.1
dSPACE gmbh

[8] DS1002/DS1003 Setup Editor and Monitor
User's Guide
Document version 4.0
dSPACE gmbh

[9] TextIO for DSP boards
User's Guide
Document version 1.0
dSPACE gmbh

[10] Signal Generator Library
Document version 1.1
dSPACE gmbh

[11] Matlab DSP Interface Library MLIB
User's Guide and Reference Guide
Document version 1.1
dSPACE gmbh

[12] Host-DSP Interface Library CLIB
User's Guide
Document version 3.0
dSPACE gmbh

[13] Real-Time TRACE Module for Matlab MTRACE
User's Guide and Reference Guide
Document version 1.0
dSPACE gmbh

[14] C Source Debugger for DS1003 Boards DEBUG40W
Installation Guide
Document version 1.0
dSPACE gmbh

[15] DSP Device Driver
User's Guide
Document version 1.0
dSPACE gmbh

4 Automatic program generation

The automatic code generation feature of the dSPACE system is an extension to the code generation of the Matlab Real-Time Workshop. The Real-Time Workshop [1] can be used to generate and build real-time applications for several types of hardware. In this process of automatically building programs, several programs are involved as can be seen in figure 4-1.

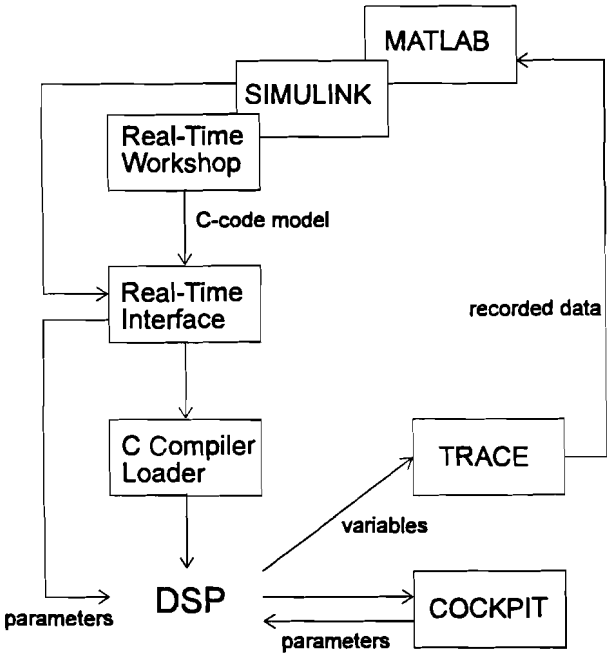

figure 4-1 : automatic code generation for the dSPACE system

A typical development and test session for real-time application comprises the following steps:

- Use Matlab and Simulink to design a real-time application and use the dSPACE I/O board library added to Simulink to specify the I/O hardware setup.
- Use the Real-Time Workshop to generate C code for the real-time application and start the Real-Time Interface. The RTI runs through all necessarily steps to prepare the application to run on the dSPACE system, and finally loads the application to the floating-point digital signal processor board.
- After the application is started on the DSP, the dSPACE TRACE tool can be started for recording time histories of variables of the real-time application. The recorded data can be transferred into Matlab for further analysis and visualization.
- After the application is started on the DSP, the dSPACE COCKPIT tool can be started for experiment control, real-time parameter tuning and signal monitoring.

This chapter summarize the whole process and the programs involved of automatic code generation. The getting started section presents a walk through of the whole path from a Simulink model to the generated code. This walk through gives a adequate knowledge of the process to start working with the dSPACE code generation tools. The Real-Time Interface is the key part of the automatic code generation process, and it is started by the Real-Time Workshop. The second section 'The Real-Time Workshop' explains the necessary parts of the Real-Time Workshop as needed by the Real-Time Interface. The third section 'The Real-Time Interface' contains a complete description of the tasks it perform.

4.1 Getting started

To illustrate the development process, the implementation of a simple controller for a ball balancing system is used. The goal of this process is to control the ball position on a rail. The angle of this rail

can be adjusted to control the direction and the speed of the ball. This ball balancing system has one input, i.e. the motor to adjust the rail, and two outputs, i.e., the ball position and the rail angle. As a model for this system a discrete state space description is used with a low sample frequency of 10 Hz.

4.1.1 Starting the development system

The automatic code generation tool of the dSPACE system is a Windows based development system. Although prior to starting windows two batch files has to been run;

dspace.bat
drvinit.bat

c:\dsp_cit\exe\dspace.bat	this batch file set up the compiler and linker environment for the development system.
c:\dsp_cit\exe\drvinit.bat	this batch file loads the device driver to and checks if the DSP is connected to the computer.

After these two batch files were run, Windows and Matlab can be started.

To be able to develop a system with Matlab for the dSPACE system, the following path must be added to the Matlab path.

c:\dsp_cit\matlab\local

note: For the DS5001 phase block there might occur a name confusion with a Matlab function in the Vision Toolbox. In that case this toolbox must be discarded from the Matlab path.

When this directory is added to the Matlab path the rest of the settings can be made by calling the m-file **dspace.m** in the c:\dsp_cit\matlab\local directory.

After these settings are made the dSPACE Simulink block library DSLIB can be opened by typing **DSLIB** from the Matlab prompt. If everything is setup correctly the following Simulink library will be opened:

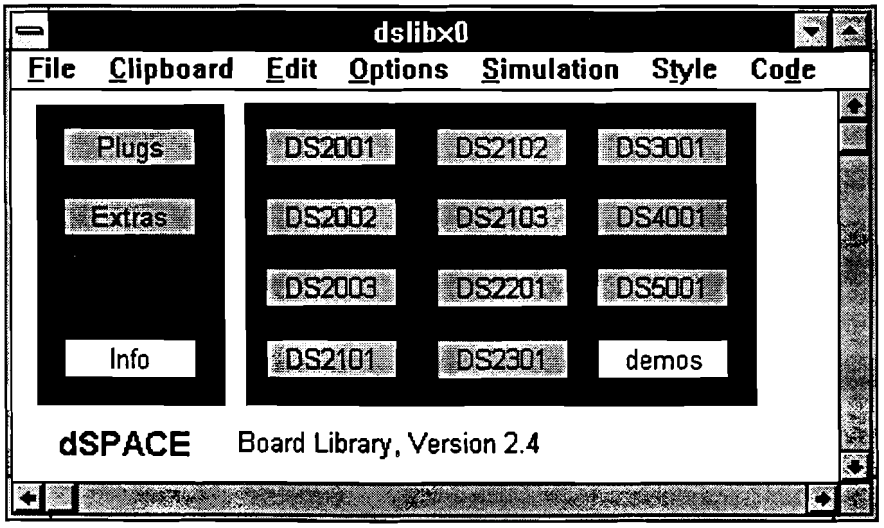

figure 4-2 : Simulink dSPACE component library DSLIB

After that, the creation of a new Simulink model can be started.

4.1.2 Generating the Simulink controller model

Different steps can be recognized in the development process. At first a Simulink model is needed of the controller for which a program is to be build. Usually a Simulink model is used to test and simulate the regulator in combination with a process model. This model can be used in Simulink to test and adjust the controller for correct results.

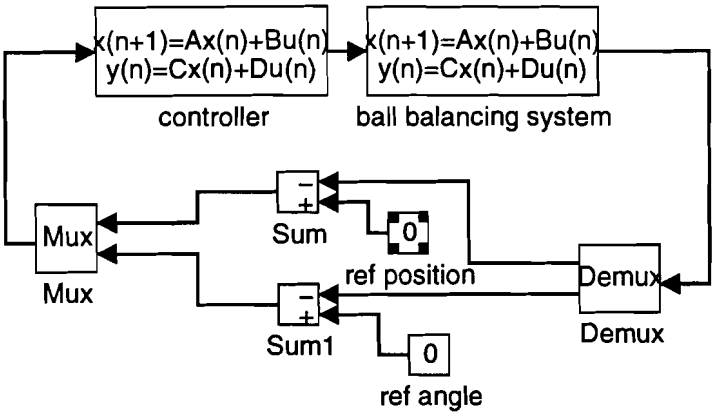

figure 4-3 : Simulink simulation model

When the controller is designed properly and the behaviour of the controlled system is tested, the controller can be implemented on the dSPACE system to get a real working controller. For the implementation the controller block only is needed. In this case the controller must be accompanied by a multiplexer block to connect the two inputs and one output of the controller. These controller inputs and outputs must be connected to the physical I/O components of the dSPACE system. For almost all the dSPACE I/O boards Simulink blocks are provided in the **DSLIB** library. This library contains Simulink blocks for all the specific I/O functions and modes of each I/O board. In the case of the current example only two AD converters and one DA converter are needed.

note : The identifiers of the I/O blocks must not be changed because they are used by the preprocessor as identifiers.

It is not sufficient to add the I/O blocks in order to get a working model for which a program is to be created. The used inputs and outputs of the I/O blocks must be connected to input and output plugs. These plugs are used in the resulting program to identify the input and output variables. The special plugs for the I/O blocks are also contained in the **DSLIB** library.

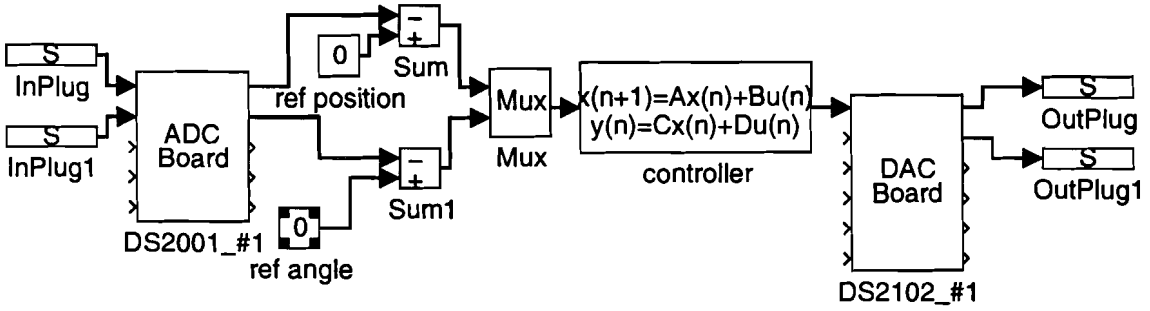

figure 4-4 : Basic controller model

The resulting model of figure 4-4 is sufficient to get a real working controller, but the inputs and outputs of controllers are usually designed to have some physical meaning. In the case of this controller the first input is the ball position in meters and the second input is the rail angle in radians. The I/O blocks on the other hand give results or expect values in the range between -1 en +1. This implies that these values must be adjusted to their physical meaning, i.e., meters or radians.

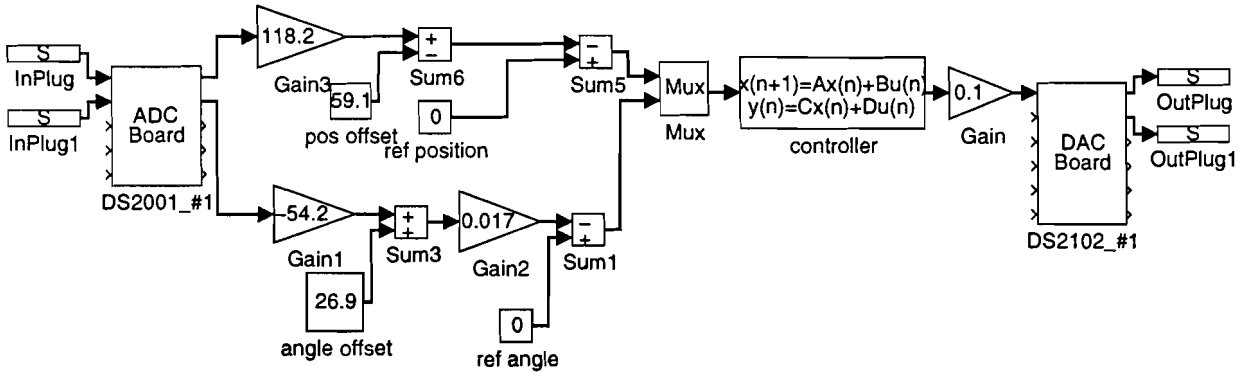

figure 4-5 : Final controller model

After all the Simulink blocks are added for the adjustment of the input and output values the resulting model can be used for automatic program building.

4.1.3 Setting up the Real-Time build options

Prior to generating the program code a few options needed to generate the correct code must be set. These options are the so called real-time options and can be set by choosing the item real-time options the Simulink CODE menu.

Choosing this menu option results in the Real-Time options dialog box.

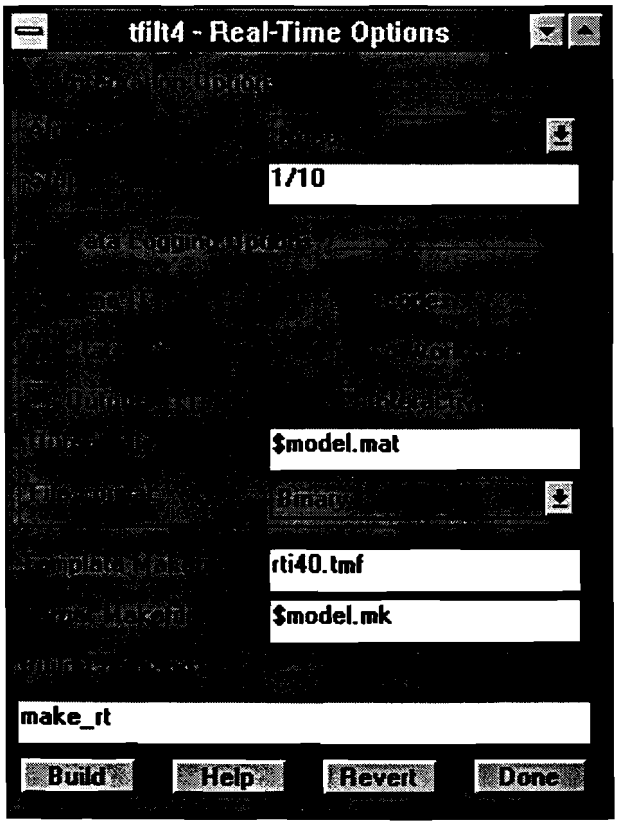

figure 4-6 : Real-Time Options menu

In this dialog box the integrating options and the step size must be set. The integration algorithm determines which integration algorithm is used in case of continuous models. The step size determines the fastest period time at which the model is evaluated. It determines the sample frequency of the AD and DA converters and the integration stepsize for numeric integration.

Further more the template makefile must be specified. The template makefile is used to build a makefile for this specific application. This specific makefile contains all the information to automatically build the desired application.

Building programs for the dSPACE system requires the template makefile option to be set to **rti40.tmf**. The **make_rt** m-file uses this template makefile to create a specific makefile for the current model. This application specific makefile gets the name **model.mk** by default, in which **model** is replaced by the current model name.

When used in combination with the Real-Time Interface, the Data Logging Options are irrelevant, because they are simply not used. The capability of real-time data logging for applications built by the RTI is provided by the dSPACE TRACE tool.

4.1.4 Automatic program building

After the real-time options are set correctly, the DSP program can be generated for the controller. The automatic program building is initiated directly from Simulink's graphical user interface. There are two ways to start the program building, the first is by choosing the **build** option in the Real-Time Options dialog box, and the second is by choosing the **Generate and Build Real-Time** option of the Code menu.

After the code is generated, it is automatically compiled into a real program. When there are no errors this program is automatically downloaded to the DSP and started.

4.1.5 Using COCKPIT to control and monitor the DSP program

The COCKPIT application from the dSPACE system can be used to monitor and change variables and parameters in the application currently running on the DSP.

COCKPIT has two modes, an edit mode and an animation mode. In the edit mode the user can define and/or edit an Instrument Panel for the application running on the DSP. In the animation mode the program interacts with the DSP board. In animation mode the variables of the currently running DSP application are read and displayed in the Instrument Panel. DSP values can be modified by user action on various controls.

When COCKPIT is started, it automatically reloads the last used Instrument Panel for a DSP application and starts in edit mode. A COCKPIT session for a newly created DSP application starts by choosing **new** from the FILE menu. This clears the current Instrument Panel and trace-file.

The trace file is a variable definition file that is automatically created for any DSP program generated with the Real-Time Interface. This file contains the specification of every block in the Simulink model from which the program is created. This file is need to monitor and edit the variables and parameters in the running application. For every new Instrument Panel this trace-file must be loaded first. This is done by choosing **Load TRACE** file from the FILE menu.

After the trace-file is loaded, the definition of the new Instrument Panel can be started. In a Instrument Panel three types of controls can be distinguished; input, output, and special. Each one is subdivided into different controls according to table 3-1.

table 4-1 : Instrument Panel controls

Input	Output	Special
ButtonArray	Alert	Data Array
Checkbutton	Bar	Frame
Incremental Input	Display	Image
Knob	Gauge	Start Executable
Numeric Input	Message Box	Static Text
OnOffButton	Multistate Alert	
Pushbutton	Sound	
Radiobuttons		
Slider		
Table Editor		

Each of these controls can be selected from the CONTROL menu. After that the control can be placed (drawn) into the Instrument Panel. After a control is placed in the panel its specific parameters must be set. This can be done by opening (double clicking the control) the control. These parameters include the two Basic Parameters for a control; the **Label** and the **Variable** of the DSP program the control gets connected to.

The label is user defined text and is used as a headline for the control. The label may be left empty, in that case there is no headline displayed for the control.

The variable specifies the program variable to which the control is assigned. The list of available variables is defined by the trace-file (.trc) that is currently loaded.

In the available variables there is a distinction between outputs of a Simulink block and parameters of a Simulink block. The block parameters are identified by a P followed by a the variable name between parentheses. Both types of variables can be used for output controls, but using a block output variable with a input control has no effect.

For every control the font, foreground color, and background color can also be specified.

The simulation framework used by the Real-Time Interface defines a few additional variables which are not part of the Simulink model. These variables are include in every trace-file generated by the RTI and can be useful for testing the designed controller. These variables distinguish in three sections

Error signaling

To signal error situations in the DSP program a variable **_error** is defined. The address of the error variable is defined by dSPACE as the last element in the dual-port RAM.

note: By specifying a defined address for the error variable is reached that this variable can be found without knowledge of the running application. This has the advantage that a stand-alone error check utility (**wchkerr** or **chkerr40**) can be used to detect errors.

The variable _error can also be displayed in the instrument panel allowing to monitor error situations.

Timing control

For measuring the execution time of the control algorithm three variables are provide.

- exec_time
- current_time
- final_time

The variable **exec_time** contains the time needed to executed the timer interrupt routine containing the model execution code. It can be used to verify the fastest possible execution rate (sample frequency) of the control algorithm.

The variable **final_time** contains the time at which the execution of the control algorithm is halted. This can be used to execute the control task for a limited time period. The variable **current_time** contains the time elapsed since the DSP application is started.

Simulation control

For simulation control two variables are available:

- sim_state
- exec_state

The variable **sim_state** is the most useful of the two. It is used as a three state variable containing 0,1 or 2. These values correspond to STOP, PAUSE and RUN. When the variable **sim_state** is 2 the control task is executed every timer interrupt. When the variable is 1 there is no execution of the control task but the internal states of the model are preserved. In the case that sim_state is 0 there is also no execution of the control task but the internal states of the model are reset to their initial values.

The **sim_state** control variable can thus be used to start, stop and reset the control task. This is very useful when testing the control task.

The second variable **exec_state** is less useful.

After the instrument panel is defined the animation mode can be started.

4.1.6 Using TRACE to measure signals

The TRACE application can be used to measure real-time behavior of any variable in the DSP application. The first thing to do before TRACE can be used is to load an application into the DSP. The first thing to do after TRACE is started is to create a new experiment and to load the trace-file of the application running on the DSP. The control panel of TRACE is shown in figure 4-7. In the control panel it is possible to select the signals to trace and to set the different trace options.

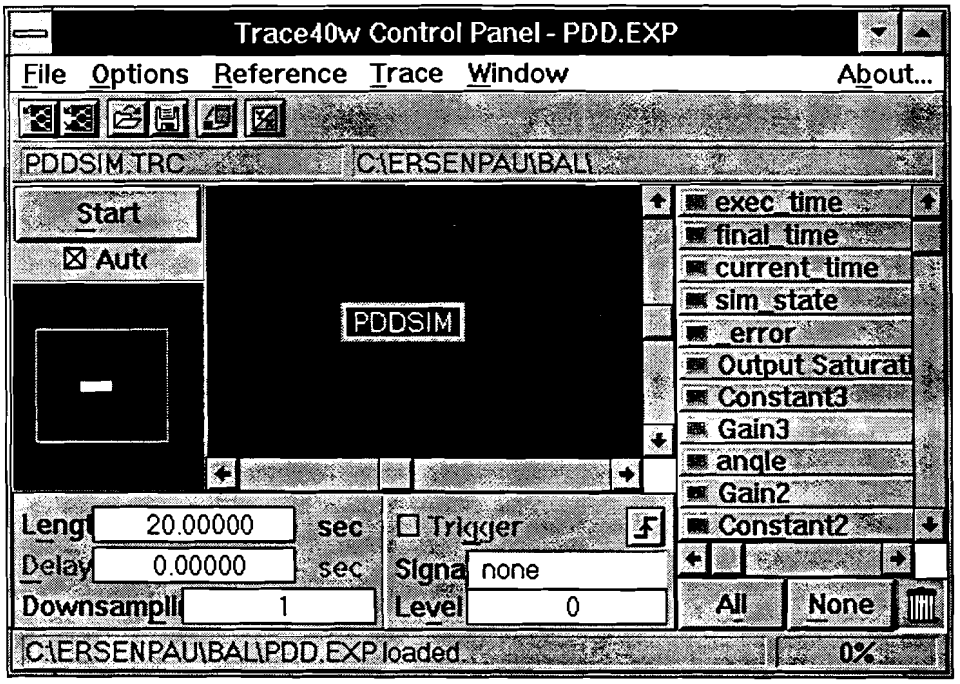

figure 4-7 : TRACE control panel

The traceable variables are listed on the right hand side of the control panel. Before a variable can be selected to trace, a plot template must be opened. This is done by choosing the 'open template' or 'new template' option in the File menu. A variable is selected for tracing selecting the variable name with the cursor. In that case a new plot frame is created in the template plot window. The template plot window has its own menu, some of the menu options affect only the current plot window, for example the option scaling of axis in the Options menu.

After selecting the desired variables for tracing, the trace capture can be started by choosing the START button in the control panel or in the plot template. After the capture finishes the data is plot into the plot windows.

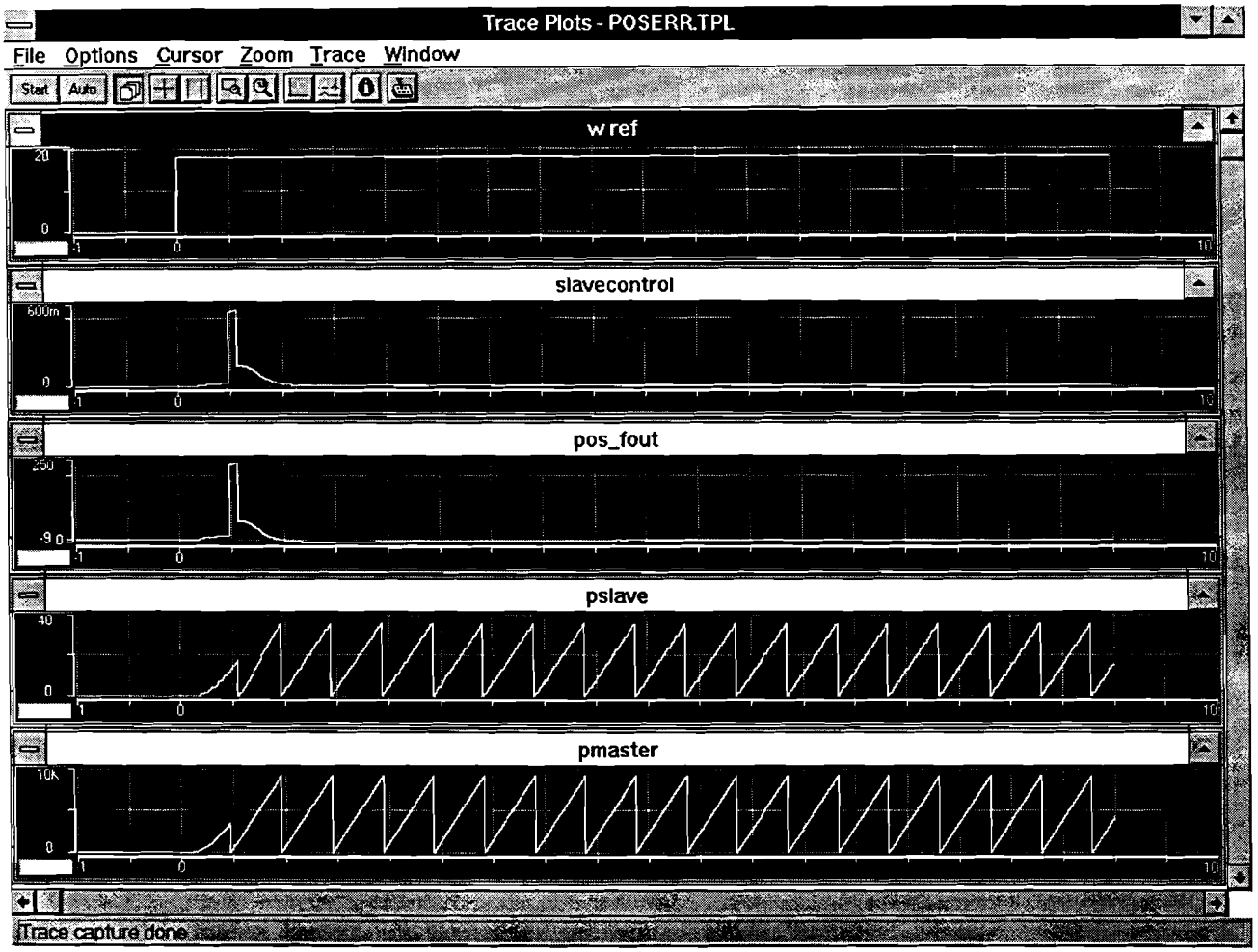

figure 4-8 : plot template window

In the TRACE control panel the behaviour of the trace data capture can be controlled. For example in the interval definition panel, (lower left corner) the time interval of the data capture can be set and a possible downsampling factor (data capture is performed with the base sample frequency of the running application). Next to the interval panel is the trigger panel. A trigger variable is chosen by selecting the square next to the variable name in the list box. The name of the variable appears in the signal box of the trigger panel. The trigger capability is turned on by selecting the trigger checkbox. It is possible to trigger on rising and falling edges with a trigger level specified by the level input. When the trigger option is turned on and the start button is selected the message 'Trigger armed' appears in the status bar at the bottom of the window. When the desired trigger event occurs, the trace capture starts. To see the traced signals prior to the trigger event, a delay can be set in the interval panel. This delay option is only active when triggering is on. The trigger mechanism is very useful to measure for example the system response to a step input or an error that occurs at unpredictable moments.

TRACE has many different ways to perform the acquisition. When start trace is chosen TRACE performs one trace capture after that the results are displayed. When the auto mode selected TRACE repeats the TRACE capture after the results of the previous trace capture are displayed. In the menu item Acquisition Mode in the Options menu some additional acquisition options can be set. This includes the option continuous mode which can be used to capture a large amount of data and store it immediately to the harddisk. This option is very useful for fast data acquisition in, for example, system identification.

Besides the described options TRACE has much more option for acquisition and data analysis. For more information on these option refer to the Real-Time TRACE module User's Guide and Reference guide [2].

4.2 The Real-Time Workshop

The Real-Time Workshop simplifies the process of building application programs. The RTW can be used to create programs for real-time and non real-time applications in a variety of host environments.

The automatic program building of the Real-Time Workshop can be initiated directly from Simulink's graphical user interface. The specific RTI settings in the Real-Time Options dialog box are used in the program building. With the correct RTI settings, the build process is started by choosing the Build button in the Real-Time Options dialog box or the Generate and Build Real-Time choice of the Code menu.

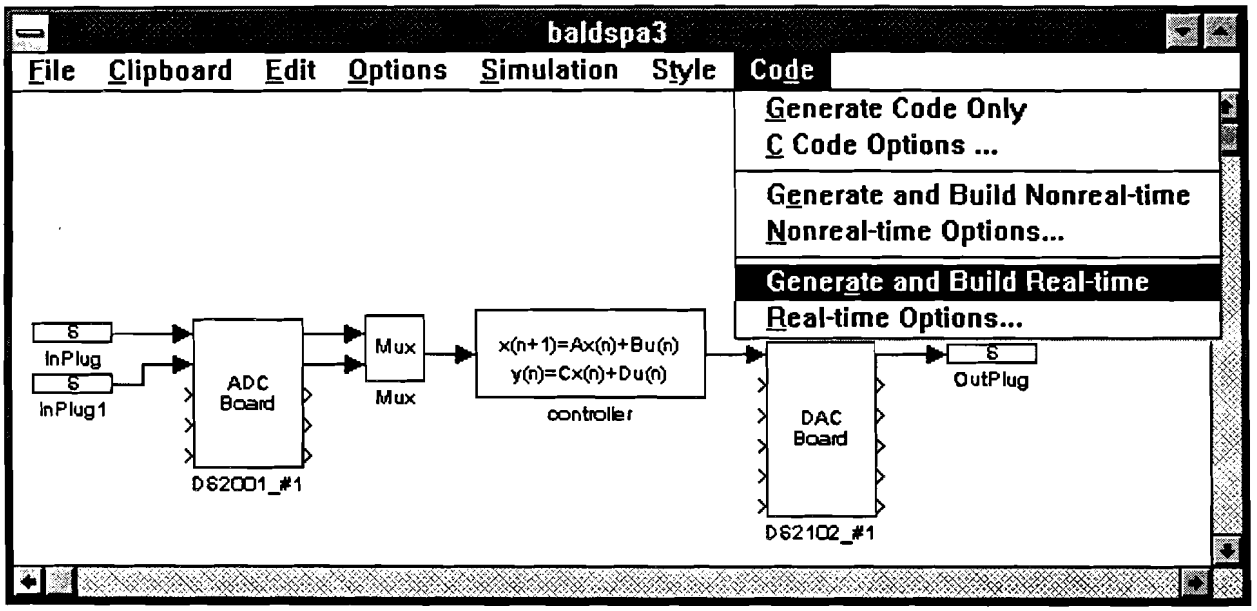

figure 4-9 : the menu option Generate and Build Real-Time

The build process consists of three main stages:

- Generating the model code
- Generating a makefile that is customized for a given build
- Invoking the make utility with the customized makefile

In the first stage C-code is generated for the Simulink model. This step results in two files model.c and model.h. The application modules and the C-code generated for the Simulink model are implemented using a common API (application program interface). This API is the same that introduced by Simulink 1.3 for S-functions.

In the second stage a makefile is generated for the model code. The makefile contains rules and dependencies to determine which code modules to compile and link. The makefile for this model is based on a template makefile. This template makefile contains information on what compiler and linker to use and about the needed libraries and source files. This template makefile is processed by a Matlab function. This processing incorporates the parameter settings from the Real-Time Options dialog box into the makefile. These parameters, in turn, determine which source files are linked and what compiler options are specified when the program is build.

In the final stage the make utility is started with the model makefile as an input file.

4.2.1 The Real-Time Options

The options in the Real-Time Options dialog box control the behavior of the Real-Time Workshop. They determine what kind of program the RTW creates and what integration method the generated program uses.

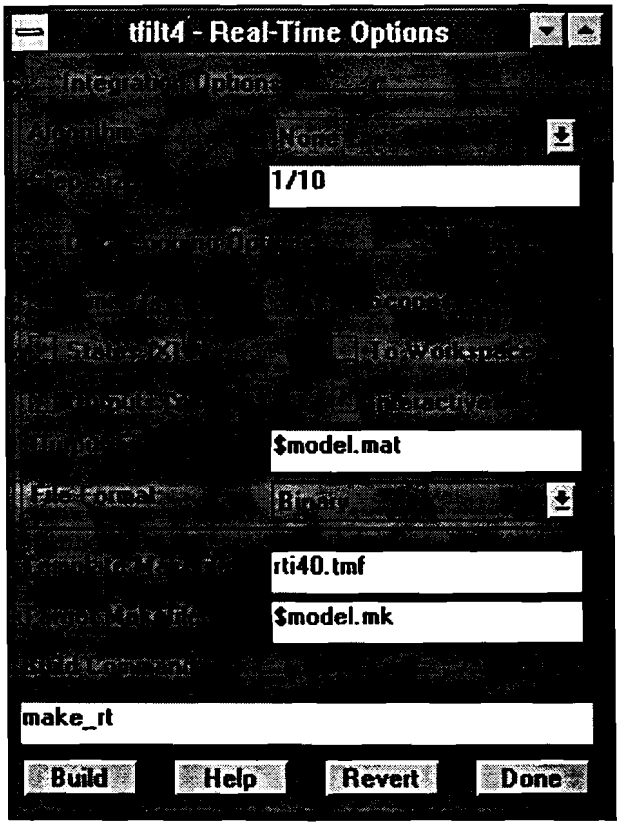

figure 4-10 : Real-Time Options menu

The Real-Time option dialog box is divide into three sections.

Integration Options

For models with continuous states, the numerical integration method can be selected in this section.

The integration algorithm can be set to one of the following options

- Euler
- RK-3
- RK-5
- Huen
- None

The default selection is the Euler integration method. The step size for the numerical integration method must be entered in the Step Size text box. For real-time applications all numerical integration methods are running with fixed step size. This is to guarantee the completion of each execution step within a fixed amount of time.

This algorithm is only used in continuous systems and hybrid systems. In discrete time systems this option is ignored and this option can be set to **None**.

Makefile options

The Real-Time Workshop uses a make utility to compile and build the final code. The makefile options determine what type of program to create and how to create it.

First of all the **Build command** option, this command should be a Matlab m-file. When the application building starts this m-file is executed. The **make_rt** m-file should be used for building real time application. The m-file uses the template makefile to create a specific makefile for the current model. This application specific makefile gets by default the name **model.mk**, in which **model** is replaced by the current model name. This is specified in the **Target makefile** option.

The template makefile to be used is specified by the **Template makefile** option. All template makefiles are similar in structure, making them easier to read and modify. They contain typical make command rules and dependencies as well as macros defining file pathnames, compiler options, etc. The macros within each template are grouped into categories so you can easily find particular definitions.

The dSPACE development system provides a template makefile for use in the automatic program building to create programs for the dSPACE system. When the dSPACE development tools are correctly installed the only thing needed to build a dSPACE program is using this makefile called **rti40.tmf**.

Data logging options

The Data Logging Options section is designed to select different simulation data to be written automatically to a Matlab MAT-file. These settings are not used in combination with the dSPACE system. The capability of real-time data logging for applications built by the RTI is provided by the dSPACE TRACE tool.

The Real-Time Workshop's program framework provides the additional source code necessary to build the model code into a complete, stand-alone program. The automatic program builder ensures the program is created with the proper modules configured in the template makefile.

4.3 The Real-Time Interface

The Real-Time Interface is a link between the C-code generated for a Simulink model and the DSP application. Basically it provides another interface level to the DSP (figure 4-11). The Real-Time Interface supplies a standard framework to run real-time simulations, with C-coded Simulink models on dSPACE processor boards.

The following modules of the RTI implement the simulation environment:

srtframe.c	This is the source code of the real-time simulation frame, i.e., the main program, the initialization routines, and the interrupt service routine.
srtframe.h	The header file that holds all standard declarations used in the simulation frame.
rti40.lib	The library holding all RTI-specific functions.
rti40lib.h	The header file corresponding to the library rti40.lib; it holds all RTI-specific data types and function prototypes.
rti40.lnk	The RTI-specific linker command file used by the Texas Instruments Linker.

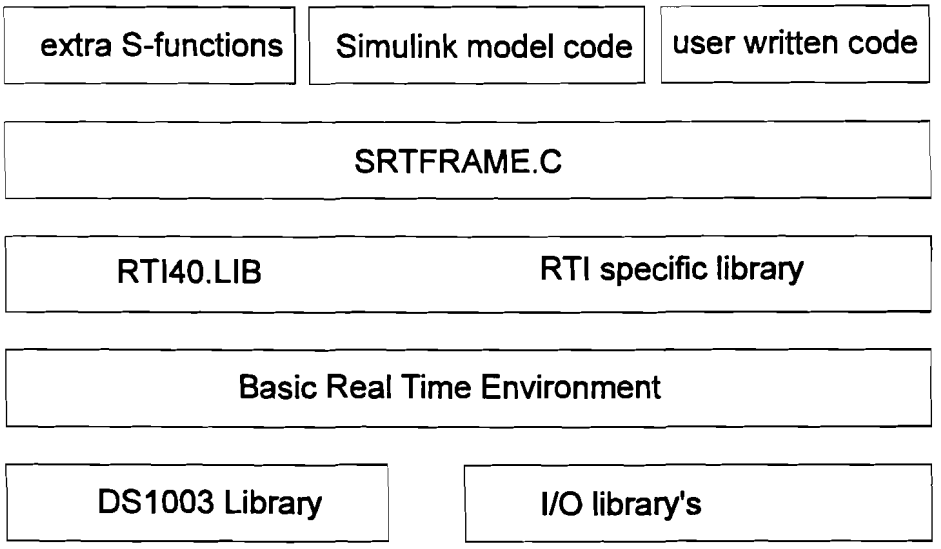

figure 4-11 : RTI development layers

Besides these main files several other files are generated and/or used. The dependencies between these files are shown in figure 4-12.

For the correct interface to the hardware the dSPACE software supplies the library ds1003.lib. Refer to the header file ds1003.h and the DS1003 software environment Reference Guide [3] for more information on this library.

At the same level it is possible to define errorcodes to signal errors in the DS1003 application. These errors signal to the error check utility (i.e., CHKERR40) through an error flag located at a defined memory location. The header file dsperror.h contains all the definitions for this mechanism.

Above this layer is the Basic Real-Time Environment available. This environment contains definitions and macros that are frequently needed in application programs. The file brtenv.h automatically includes the header files ds1003.h and dsperror.h

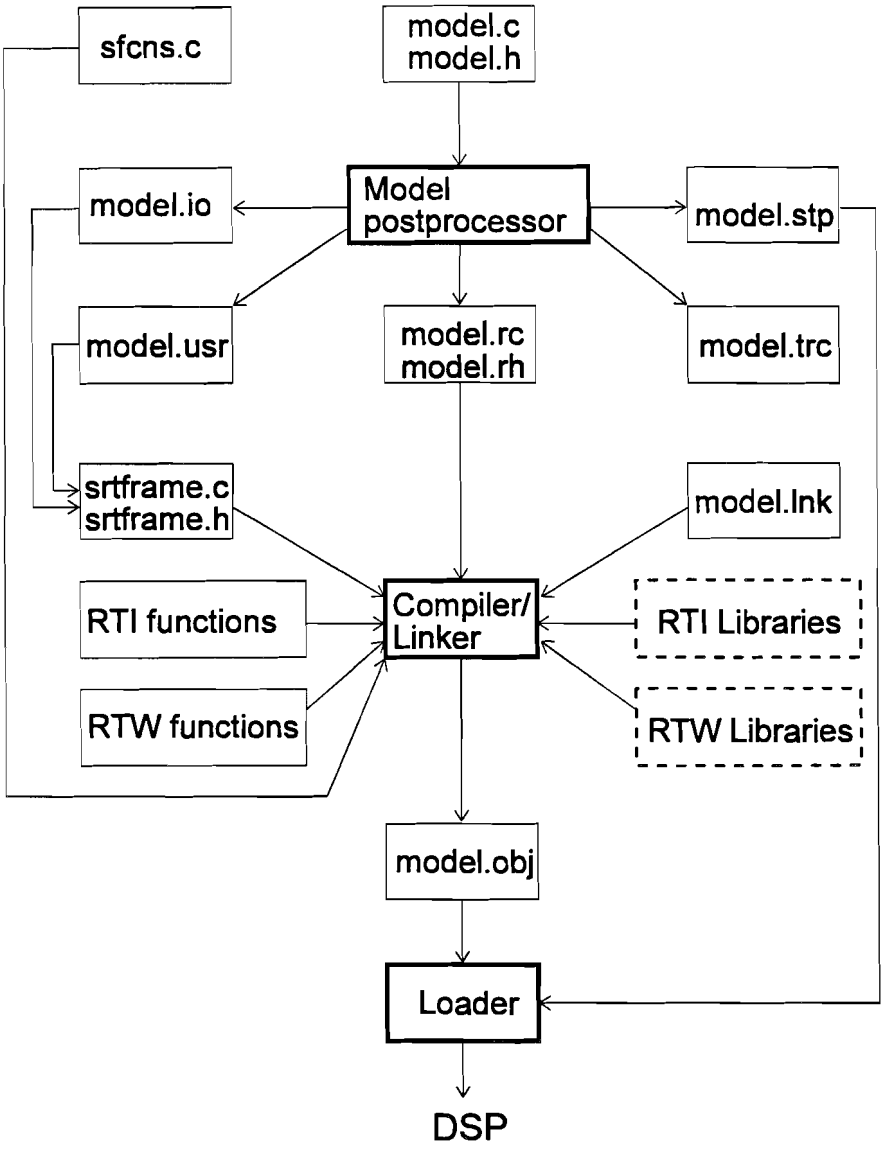

figure 4-12 : Real-Time Interface layout

The Real-Time Interface provides additional source codes and a new template makefile for the use with the Real-Time Workshop [1]. This new template makefile holds a set of macro definitions and rules describing how to get from the C-coded model files generated by the RTW to the real-time program executed on the a dSPACE hardware system. These rules describe how to compile the different source files of a model, how to link files and libraries together to form an executable image, and how to download the executable to the DSP. Dependencies between the different rules determine the sequence in which these rules need to be evaluated, and control the following steps:

1. Run the RTI Model Postprocessor ***revise*** to modify the generated C code of the model, and to generate all application specific setups and I/O function calls for the implementation on dSPACE hardware.
2. Compile the model files, source code for the S-function used, and the files of the real-time simulation environment; link all the necessary object files and RTW/RTI libraries, and generate an executable image. For this task, the Texas Instruments Compiler/Linker ***cl30*** is used.
3. Download the executable to the hardware using the ***mon40*** program.
4. Start the dSPACE error check program ***chkerr40***, which controls whether the executable was started correctly or not.

The generation of the DSP code starts with the C-code generated by the Real-Time Workshop for the Simulink model. This generated C-code consist of two code files; *model.c* and *model.h*. The file *model.c* is first processed by the postprocessor **revise**. This postprocessor takes all the dSPACE I/O definitions out of the c-file and renames the processed c-file *model.rc*. The file model.h is not processed by the postprocessor but also copied to the *model.rh*. The I/O definitions that **revise** takes out of the *model.c* file are placed in the file *model.io*.

Besides the files *model.rc*, *model.rh*, and *model.io* **revise** creates three other files. First of them is the file *model.trc*. This file contains all global defined variables of the file *model.c*. This file in combination with the *model.map* file generated by the linker/compiler, are input files for the programs **TRACE** and **COCKPIT**. The second file *model.stp* contains the setup definition of the DS1003 board and the peripheral boards needed by the model code. The last file *model.usr* is only created if it not already exists in the current directory. This file contains macro definitions for user functions that are included in the file *srtframe.c*. These macro definitions can be modified to perform the desired function.

After the model postprocessor finishes the TI compiler and linker are started. The compiler generates object code for the different input modules. The linker finally combines the object modules into one object file *model.obj*.

After the code is compiled and build, it is loaded into the DSP RAM by means of the loader utility **mon40**.

4.3.1 Configuring the template makefile

The Matlab utility **rtisetup** provides a flexible and comfortable setting of options of the template makefile and the Real-Time Options Dialog box. By using this utility it is not necessary to edit the template makefile directly.

This setup utility offers a graphical user interface with different register pages.

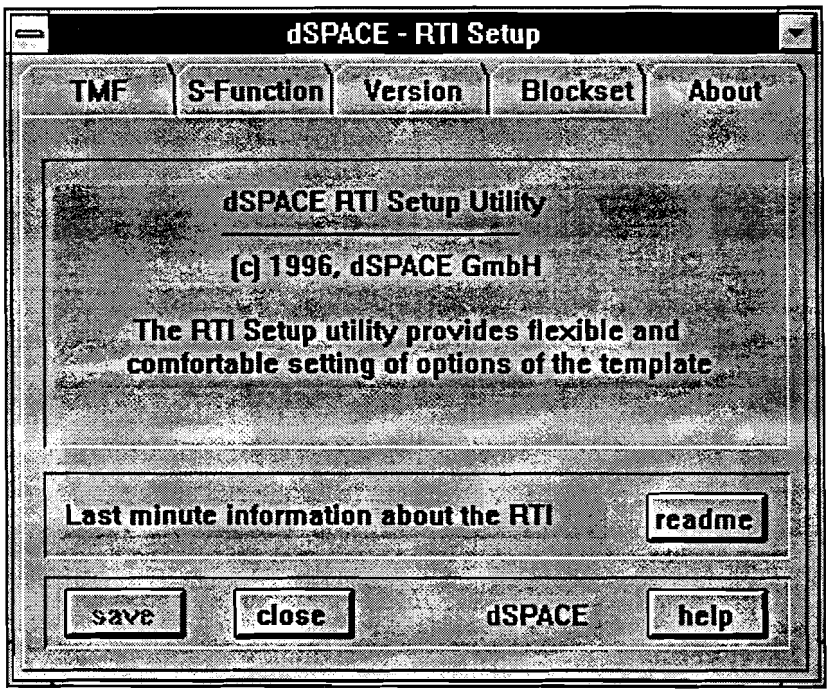

figure 4-13 : Real-Time Interface setup dialog box

TMF page	On this page options set via RTI's template makefile can be altered. It is possible to specify the template makefile it should affect. It is possible to specify if the program should be singletasking or multitasking, for which version of the TRACE the trace-file must be created. Version 3.0 of TRACE 3.0 has extended capabilities. The number (Vector Size) of outputs or parameters from one Simulink block to appear in the trace-file can be set. Further more it is possible to specify user code settings, as; additional source files, source directories, include paths, and libraries.
S-function page	On this page the directories where the C-coded S-functions and their corresponding MEX-files are found can be specified.
Version page	On this page RTI version specific information can be altered, which appears in the Real-Time Option Dialog box. It is possible to indicate a non standard template makefile to be used.
Blockset page	On this page RTI support for the blocksets displayed in the list box can be invoked.

4.4 Structure of the generated code

The Real-Time Workshop and the Real-Time Interface generate code for Simulink models. This provides a framework for implementing real-time application programs on the dSPACE DSP system. This section describes the framework and the structure of the programs created with it.

Initiating the build option in the Real-Time Workshop generates code for the Simulink model of the desired controller. This code is in the S-function format of Matlab. The function of this S-function must be called in the correct sequence in order to calculate the model. Therefore the model code is embedded into a framework which takes care of these calls.

The c-code file *srtframe.c* contains the main program of the final DSP program. This file contains the source code of the needed timer interrupt and the main program loop. The main program contains the proper initialization calls needed to initialize the S-function model code in the file model.rc.

4.4.1 The main function

The main() function in a C program is the point where program execution begins. This main routine is the basic program for a control algorithm. It takes care of initialization and servicing background tasks. It has the following layout:

```
void main()
{
       initialize();
       start_isr_t1(base_sample_time);
       /* background task */
       for (;;)
       {
              service_cockpit();
              UpdateParameters(S, &P_new, &P);
              /* simulation control */
              ...
       }
}
```

As can be seen the control program is contained in an endless loop. With basicly only two initialization calls. The call **start_isr_t1** set the timer interrupt for timer 1. This timer of the TMS320C40 is used to generate a timebase for sampling instants. The actual timer interrupt contains the code to execute the model code.

The endless loop is the background tasks of the control algorithm. It contains the **service_cockpit** call and a call to **UpdateParameters**. The **service_cockpit** call is a software interrupt which is provided with code only if COCKPIT is in animation mode. It is contained in the code to be able to use COCKPIT for application testing and monitoring. The **UpdateParameter** call changes the parameters of the Simulink data structure. This call is necessary, because COCKPIT can change parameters but can not change these parameters directly in the Simulink data structure.

The simulation control is not described here specifically, but it contains code to be able to turn on and off the execution of the Simulink model code. This is very useful when testing control application, because it is than possible to reset the model execution in case of an error. This can be controlled by adding the corresponding variable to COCKPITs instrument panel.

4.4.2 Initialization

The initialization routine performs several initialization tasks for the control program.

```
Initialize()
{
       init();
       S=ssCreateSimstruct()
       model(S)
       ssCallmdlInitializeSizes(S);
       ssInitSimstruct(S)
       InitSampleTimes(S)
       InitUpdateParameters(S)
       InitVariables();
       InitIO();
}
```

At first a call is made to the function **init** from the basic real-time environment. This function initializes the basic DS1003 hardware in order to operate correctly. After that the data structure for the S-function is created. This data structure contains all the model information like e.g.; sampletimes, number of continuous states, number of discrete states and model parameters. At first this new data structure is empty. The call to **model**, which is defined in the model.rc source file, connects all the operational functions to the data structure. These operational functions are the functions prescribed in the S-function format, which are defined in the model.rc file. After that the sizes of the state vectors and input and output vectors are initialized by a call to **mdlInitializeSizes**. Next the remaining parameters in the Simstruct data structure are initialized by a call to **ssInitSimstruct** and the sample times are initialized by a call to **InitSampleTimes**. This covers the normal initialization of the Simstruct data structure. In this case the S-function is a Simulink model, this requires some additional initialization. At first the parameters of the Simulink blocks are initialized (**InitUpdateParameters**). Secondly the initial conditions of the state vectors are initialized by a call to **InitVariables**.

After all these initialization call the Simstruct data structure is completely initialized and ready to be evaluated. Remains the initialization of the needed dSPACE I/O components. This is done by a call to **initio** (implicit called by **InitIO**). This function is defined in the by the postprocessor **revise** generated **model.io** file. It contains all the initialization of the dSPACE I/O used by **model**.

4.4.3 Interrupt service routine

When the DSP program is running, the background task is always executed. It is only interrupted by the interrupt of the timer 1. In the interrupt service routine of this interrupt the controller model is evaluated.

```
void isr_t1()
{
       begin_isr_t1(* _error);
       service_trace();

       evaluate_model(S);

       end_isr_t1();
}
```

Besides the model execution code, there is some additional code in the timer interrupt routine. First there are calls to **begin_isr_t1** and **end_isr_t1**. The function calls build up a detection scheme to be able to detect the interrupt of a previous interrupt call. This error situation is called overloading. An overload occurs if the interrupt service routine is started again while it is still active (previous call is not yet ended).

4.4.4 Controller evaluation

The function **evaluate_model** performs the execution of the control algorithm at the current sampling instant.

```
evaluate_model()
{
        input();
        usr_input();
        ssCallOutputs();              /*calculate model outputs */

        usr_output();
        output();

        ssCallmdlUpdate();            /* calculate and update discrete states */
        UpdateContinuousStates        /*calculate and update continuous states */
}
```

The control algorithm is based on the calculation of a state space system. With the current states and the new input the new output can be calculated.

$$\begin{aligned} \mathbf{y}(t) &= \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t) \qquad \text{(continuous)} \\ \mathbf{y}(k) &= \mathbf{C}\mathbf{x}(k) + \mathbf{D}\mathbf{u}(k) \qquad \text{(discrete)} \end{aligned} \tag{4-1}$$

After that the model states can be updated.

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \qquad \text{(continuous)} \\ \mathbf{x}(k+1) &= \Phi\mathbf{x}(k) + \Gamma\mathbf{u}(k) \qquad \text{(discrete)} \end{aligned} \tag{4-2}$$

This means for the control task that first the inputs must be read **(input)**. Most of the time this will be an AD conversion of the input signals.

note: In the dSPACE system AD conversion of input signals is started by the **input** function. All AD converters are started at the same time and the function waits until all conversions are finished before the input data is collected. In that case the execution time of the control algorithm is extended by the conversion time of the AD converters.

After the inputs are read the new outputs can be calculated by the S-function call **ssCallmdlOuputs**. The new output values are 'send' to the required outputs, e.g., DA converter.

The internal states of the model are update by calls to **ssCallmdlUpdate** and **UpdateContinuousStates**. The prior calculates the new states using equation 3-2 for the discrete states. The latter uses equation 3-2 to calculate the derivatives of the continuous states and updates the states using numeric integration.

note: The numeric integration method used for updating the continuous states is the method specified in the Real-Time Options dialog box of the Real-Time Workshop.

4.5 Summary

The Windows development system from the dSPACE system is an easy to use framework for designing, implementing and testing controllers. The Real-Time Interface in combination with the Real-Time Workshop makes it simple to generate working code for a Simulink model of a controller. A process that can take several days when the implementation must be performed by hand. With the advantages that the controller model can be tested in Simulink and that the code is generated for this tested controller. An implemtation can be made for any type of controller, continuous time or discrete time. The code generation takes care of the numerical integration routines needed for updating a continuous system. It is also very easy to modify the controller, simply change the Simulink model and regenerate the code.

The generated controller implementation runs on the dSPACE DSP board, giving the possibility to test the controller on real test plants. The TRACE and COCKPIT tools are very useful when testing the controller. With COCKPIT it is possible to change parameters in the model, which becomes effective immediately. With TRACE it is possible to view the time behaviour of the controller.

One disadvantage might be that the Real-Time Interface is only useful for cyclic tasks, i.e., systems with a fixed sample frequency. This covers most controllers but sometimes some external events are necessary for additional control. To perform such tasks the framework must be edited and the desired functionality added. This is also true for functions which are not standard Simulink blocks. This functionality must be added by user written code, reducing the advantages of fast controller implementation.

4.6 Literature

[1] Real-Time Workshop
User's Guide
May 1994
The MathWorks Inc.

[2] Real-Time TRACE Module
User's Guide, Reference Guide
Document version 3.0 for software version 3.0
dSPACE gmbh

[3] DS1003 Software Environment
Reference guide
Document version 3.1
dSPACE gmbh

5 Extending the generated code

It is sometimes necessary to perform actions in a controller which can not be done directly in a Simulink model. The generated code must be extended with these extra functions.

There are three ways to extend the generated code with extra functionality. The easiest way is to use the macro definitions in the generated file model.usr. This file is created when code is generated for a Simulink model and in the current directory is not yet a model.usr file available. The second way is to make an S-function for the needed functionality. An S-function is a piece of code following a defined format. This format is defined by MathWorks as a possibility to make additional functions for Matlab. The third way is to make changes to the simulation framework file srtframe.c. All three methods are discussed in this chapter.

5.1 Using the model.usr file

When code is generated for a Simulink model using the Real-Time Interface, there is also a file generated called **model.usr**. This file is only created when it does not already exists. It contains the macro definitions of five calls:

- usr_background
- usr_input
- usr_output
- usr_initialize
- usr_terminate

These function calls are made by the program code in srtframe.c at specific points. It makes it possible to add some user code to the executing program. The function **usr_background** can contain code for some extra background tasks. The functions **usr_input** and **usr_output** can be used to perform non standard input and output or some filtering of the just read input or the next output. The macro **usr_initialize** can contain some initialization code for extra variable or non standard I/O components. The **usr_terminate** is called when the model execution is stopped, it can be used to clear output of I/O component in case of an error.

In order to be able to access variable (block outputs) in the Simulink model in this file with user code some special blocks are available in the **DSLIB** library. These are special input and output blocks called DSInport and DSOutport and are found in the **extras** section of **DSLIB**. The DSInport is used to transfer the value of the user written code to the Simulink model and the DSOutport is to transfer a Simulink block output to the user code.

This process is illustrated using a very simple example. An Simulink Sine Wave block is used to generated a sinewave. This signal must be level adjusted by adding a constant value of 0.5. This can be done in Simulink by using a Sum block. However in this case both the sinewave and the constant value of 0.5 are transferred to the user code where they are added and transferred back to the Simulink model.

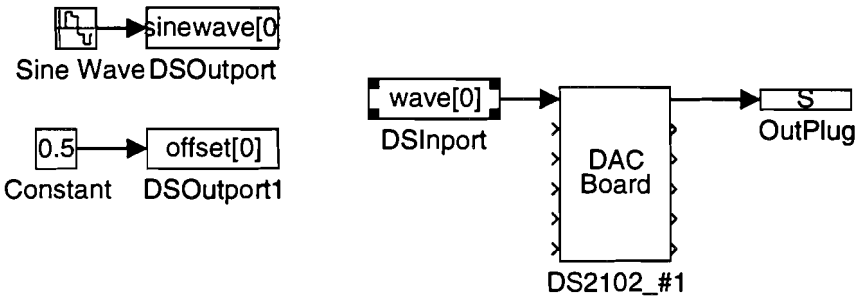

figure 5-1 : Simple example for extending the generated code.

When code is generated for this Simulink model, a model.usr file is created. In the following example of the model.usr file is the code added for the addition that is required.

```
/*********************************************************************

   Include file simpex.usr:

   Definition of macros for user defined initialization, system I/O,
   and background process code


   Copyright (c) 1993-96 by dSPACE GmbH

 *********************************************************************/
float sum,sin,off;

#define usr_initialize()                \
  sum = 0;

#define usr_input()                     C-code to Simulink \
  DSINPORT(wave,0,sum);    \

#define      usr_output()               output Simulink to C-code\
  off = DSOUTPORT(offset,0);            \
  sin = DSOUTPORT(sinewave,0);          \
  sum=sin+off;

#define      usr_background()           \

#define      usr_terminate()            \
```

The data transfer to and from the user code is performed through the variables defined in the DSInport and DSOutport blocks. These variable names are also used in the functions DSINPORT and DSOUTPORT in the user code. With the DSINPORT function the value of (in this case) *sum* is copied to the transfer variable *wave*. With the DSOUTPORT function the values of the transfer variables *sinewave* and *offset* are copied to the variable *sin* and *off*. The variable *sum* is in the macro **usr_initialize** set to zero in order to have a defined input when **usr_input** is called for the first time (this is prior to **usr_output**).

The function DSINPORT is also very useful to make a global program variable for tracing using TRACE and COCKPIT, because in TRACE and COCKPIT it is normally only possible to trace variables from the Simulink blocks. By using the DSINPORT function and the DSInport block the variable can be made available in the Simulink model and thus it can be traced.

5.2 Simulink S-function format

An S-function is a programmatic description of a dynamic system. S-functions can be written using Matlab, C, or FORTRAN. The C and FORTRAN S-functions are compiled as MEX-files using the CMEX and FMEX utilities. As with other MEX-files, they are dynamically linked into Matlab when needed. The Matlab S-functions (M-file S-functions) can run directly in Matlab.

C-code S-functions that conform to the revised Simulink 1.3 format can be integrated seamlessly with automatically generated code.

The form of an S-function is very general and can accommodate continuous, discrete, and hybrid systems. As a result any Simulink model can be described as a S-function. In fact, the Simulink C-code generator works by converting a block diagram model into the equivalent C-code S-function.

S-functions are incorporated into Simulink models by using the S-function block in the Nonlinear library. The S-function block's dialog box is used to specify the name of the underlying S-function. The Simulink masking facility can be used to add custom dialog boxes to create S-function blocks to make it easier to specify optional parameters. Refer to the Simulink User's Guide [1] for more information on using S-functions.

The advantage of using S-functions is that it is possible to build a general purpose block that can be used many times in a model with different parameters for each instance of the block in both the Simulink simulation model as well as in the dSPACE real-time program.

5.2.1 How S-functions work

Each block in a Simulink model has the following general characteristics: a set of inputs u, a set of outputs y, and a set of states x.

The state vector may consist of continuous states, discrete states, or a combination of both. The mathematical relationship between these quantities are expressed by the following equations.

$$y = f_0(t,x,u) \qquad \text{(output)}$$

$$x_{d_{k+1}} = f_u(t,x,u) \qquad \text{(update)}$$

$$\dot{x}_c = f_d(t,x,u) \qquad \text{(derivative)}$$

$$\text{where } x = \begin{bmatrix} x_c \\ x_{d_k} \end{bmatrix}$$

When a simulation runs, Simulink makes repeated calls to each block in the model, to compute its outputs, update its discrete states, or compute its derivatives. Additional calls are made at the beginning and end of a simulation to perform initialization and termination tasks.

For M-file S-functions, these different calls are denoted by the value of the flag parameter in the function call. For C-code S-functions, calls are made to individual subroutines. These subroutines must have exactly the same names as shown in table 5-1 in order to work correctly.

table 5-1 : Function names in S-functions

Simulation stage	C MEX-file S-functions	M-file S-function flag
Initialization	mdlInitializeSizes mdlInitializeSampeTimes mdlInitializeConditions	flag = 0
Computation of output	mdlOutputs	flag = 3
Major time step update	mdlUpdate	flag = 2
computation of derivatives	mdlDerivatives	flag = 1
end of simulation tasks	mdlTerminate	flag = 9

5.2.2 C-code S-functions

The application modules and the code generated for a Simulink model are implemented using a common API (application program interface). This API defines a data structure (called Simstruct) that encapsulates all data for each instance of a model. The elements in this data structure are summarized in table 4-2.

table 5-2 :Fields in Simstruct

Field	Data contained in Field
Version	Version of Simstruct
Parent	The Simstruct's parent
Root	The root level Simstruct
Sizes	Size information about the model (e.g., number of states, number of inputs and outputs, number of sample times)
Inputs	Contains additional input arguments passed to S-function
Vectors	Various vectors (e.g., block inputs and outputs, block parameter vector, work vector)
T	The simulation time. (not always used for real-time applications.)
TFinal	The final simulation time (not always used for real-time applications.)
TCount	Counter used for accurate time base
StepSize	The time interval on which to execute the model code
MinorTimesStepFlag	Flag indicating when a minor time step is occurring
Events	Timing events (e.g., interval times and skew times for discrete systems, current and next sample time)
ModelMethods	Pointers to model functions
States	Input, output, states, derivative, and discrete state vectors
Utility	Temporary storage and user definable data

This API is the same as the API introduced by Simulink 1.3 for S-functions. Using a common API for both the model code and the S-functions makes it possible to generate code that automatically incorporates any S-functions that are in the Simulink model. This simplifies the process of:

- Incorporating existing C code into the real-time program by adding it to the model as a S-function
- Interfacing the program with other engineering tools
- Connecting the program to hardware devices

The API of the Simulink 1.3 for S-function format defines many functions to act on the data structure Simstruct. These functions perform tasks varying from initialization to model execution and model termination.

Main program initialization

Initialize size information in Simstruct	mdlInitializeSizes()
Initialize sample time and offsets	mdlInitializeSampleTime()
Specify initial conditions	mdlInitializeConditions()
Add function pointers to Simstruct	model()
Initialize block information for monitoring	mdlBlockInfo()
Initialize scope block information	mdlScopeInfo()

Model Execution

Compute block and system outputs	mdlOutputs()
Update discrete state vector	mdlUpdate()
Compute derivatives for continuous models	mdlDerivatives()

Main program termination

Orderly termination at end of program	mdlTerminate()

Some of these functions are needed for the user to write an S-function:

- mdlInitializeSizes()
- mdlInitializeSampleTime()
- mdlInitializeConditions()
- mdlOutputs()
- mdlUpdate()
- mdlDerivatives()
- mdlTerminate()

In order to make it easier for the user to write S-function, Matlab provides a template containing the seven functions. This template is called **sfuntmpl.c** and is found in the directory **\MATWIN\SIMULINK\SRC**. This directory contains also some example S-functions.

5.3 Extending the srtframe.c

The third way to extend the capabilities of a generated application is to edit the srtframe.c directly. Usually this is only necessary when the required functionality can not be added with one of the other methods. This is for example the case when interrupt driven I/O is required. Normally the timer interrupt provide by srtframe.c is sufficient for cyclic control systems, i.e., all state space systems. This is not sufficient when for example rate varying pulses must be counted. In that case an extra interrupt routine is needed triggered by the incoming pulses [see chapter 6].

In normal operation the Real-Time Interface retrieves the srtframe.c from the **\DSP_CIT\RTI40**. This version of the srtframe.c functions like a template. If it is copied to the directory where the model resides, this version of the srtframe.c is used by the RTI. The local version can than be edited without affecting the other dSPACE applications based on the srtframe.c

note : Editing the srtframe.c in the \DSP_CIT\RTI40 directory effects all applications depending on the global version of srtframe.c

Editing the srtframe.c can for example be used to add an PHS interrupt task [see chapter 2] to the application. These PHS interrupts are not supported by the Real-Time Interface. An example of the srtframe.c with an extension for using the DS5001 PHS bus interrupt is provide with the PHS Interrupt library.

5.4 Summary

The three methods provide great flexibility for adding extra functionality to the generated code for a Simulink model.

The first method, extending the model.usr file, is useful for small additions to the Simulink model. The S-functions have the advantages that they can be used for simulation in Simulink if they are compiled as MEX-files. The S-functions perform their tasks also on a cyclic time base just like the Simulink model code. (The S-functions are executed by the Simulink model code). The method with the most freedom for the programmer is extending the srtframe.c. The srtframe.c can also be used as a basis for an application with Simulink providing some extra model code for easy to perform tasks.

5.5 Literature

[1] Simulink
User's Guide
1993
The MathWorks Inc.

6 Master Slave motor synchronization

The company Buhrs in Zaandam develops production lines for automatic packing of magazines and papers. Different operations are performed in each stage of these production lines. All the stages in the production lines are currently synchronized by one long mechanical axis. This mechanical axis gives difficulties in servicing and adjusting these production lines. Buhrs wants to know if it is possible to replace the mechanical axis by an electrical axis. The main notion for this electrical axis is to give each stage in the production line its own driving motor and synchronize all the motors in the production line. This results in a master slave synchronization, one motor (the master) becomes a reference while all other motors (the slaves) must be synchronized with the master.

6.1 The process description

To develop and test the master slave motor synchronization a test plant consisting of two motors with frequency inverters is provided by Buhrs.

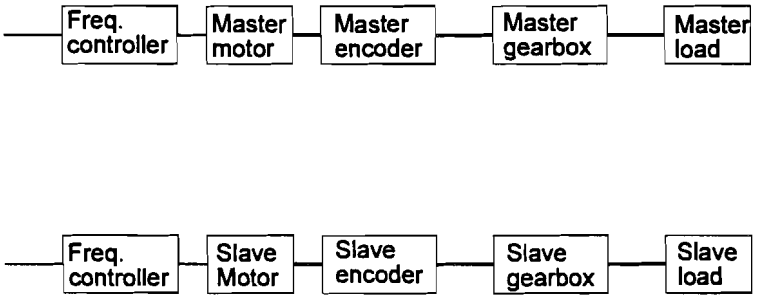

figure 6-1 : Available test plant

In this test plant several components are observed, first of all the two asynchronous motors. Both of these three phase motors are driven by three voltages supplied by a frequency inverter. The frequency inverter generate sines based on the input voltage. Each motor is supplied with an encoder generating 512 pulses per revolution of the motor axis. These encoders define the position of the motor axis. The motor axis is connected to the load axis of motor by a transmission.

The load axis of both motors are supplied with a pulse sensor. These sensors generate one pulse per revolution of the load axis. These pulses mark the product time cycle T_c of each stage in the production line. The timing of each stage to the next stage comes very strict, too much difference in the start of the product cycle of each stage can result in a pile up in the production line. It is however possible to have a phase difference between the start of the product cycle in different stages.

The master-slave synchronization is solved using the dSPACE system as the development system. The environmental model of figure 6-2. shows the interconnection between the test plant and the DSP.

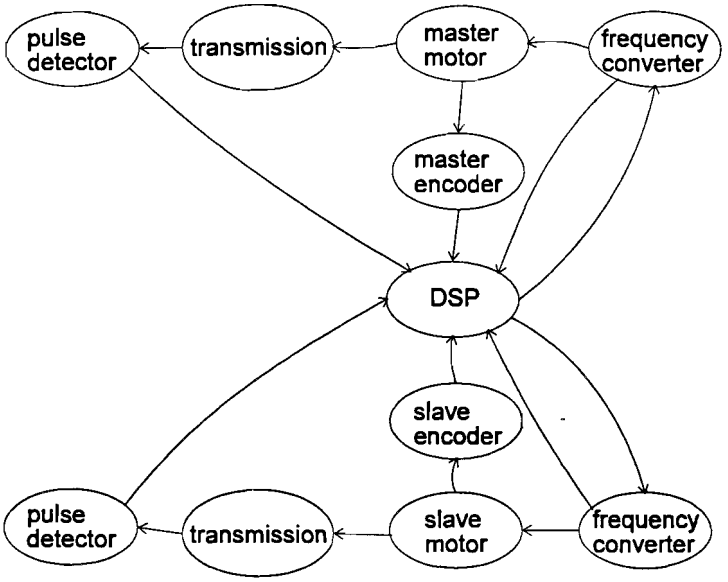

figure 6-2 : Environmental model

6.2 Definitions

In order to describe the process behaviour the following definitions are used.

Master axis

ω_{mm} angular velocity of the master motor axis (rad/sec)

ω_m angular velocity of the master load axis (rad/sec)

θ_{mm} position of the master motor axis (rad)

θ_m position of the master load axis (rad)

n_m master transmission

p_m pulse count master encoder

Slave axis

ω_{sm} angular velocity of the slave motor axis (rad/sec)

ω_s angular velocity of the slave load axis (rad/sec)

θ_{sm} position of the slave motor axis (rad)

θ_s position of the slave axis (rad)

n_s slave transmission

p_s pulse count slave encoder

General

ω_{ref} angular velocity reference

T_{cm} product time cycle master (sec)

T_{cs} product time cycle slave (sec)

t_{sm} start of product cycle master (sec)

t_{ss} start of product cycle slave (sec)

φ phase difference between t_{sm} and t_{ss} (rad)

6.2.1 The asynchronous motor

A three phase AC motor is operated by applying three phase shifted sinusoidal voltages to the motor. The motor velocity is directly related to the frequency of these sine waves. When this reference frequency is denoted by ω_{ref} the motor can be simply modeled like figure 6-3

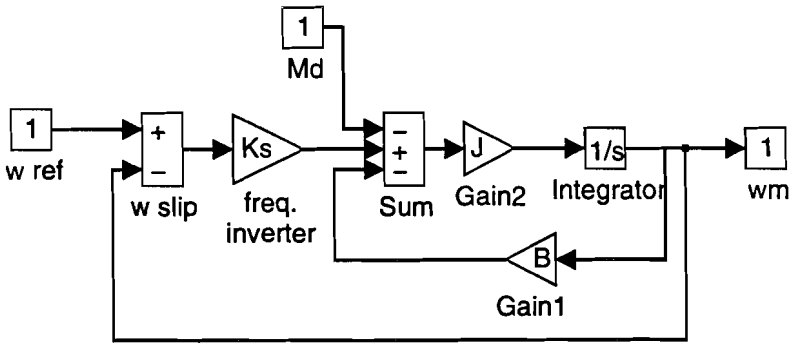

figure 6-3 : first order motor model

Based on this model the next system equation for the motor can be derived.

$$\omega_m(s) = \frac{K_s/J}{s + \frac{K_s+B}{J}}\omega_{ref}(s) - \frac{1/J}{s + \frac{K_s+B}{J}}M_d(s) \qquad (6\text{-}3)$$

Because there is no information available on the parameters K_s, J and B, the motor model will be determined by system identification techniques.

6.2.2 The frequency inverter

The frequency inverter converts the input value into the three motor input signals. In order not to produce to much slip in the motor, the frequency inverter includes a rate limiter. This rate limiter limits the rate at which the motor frequency can change.

The conversion of the frequency inverter is given in table 6-1.

table 6-1 : conversion frequency converter

U_{in}	ω_{ref}
0 .. 10 V	0 .. 70.4 Hz 0 .. 442.34 rad/sec

The rate limiter changes the frequency from minimum to maximum in 2 seconds when a step input of 10 V is applied. Smaller steps inputs result in equally smaller rise times. This implies a maximum change in the applied motor frequency of:

$$\frac{\partial f}{\partial t} < 35{,}2 \text{ Hz} \qquad (6\text{-}4)$$

The frequency inverter is a non linear component in the control loop. This gives rise to certain difficulties when controlling the process. Therefore special measures must be taken to take this non-linearity into account. To simplify this problem the frequency converter has an additional output. This output is a signal equal to the limited frequency profile that is applied to the motor by the inverter. The output signal of the frequency inverter is a little attenuated when compared to the input voltage. The maximum frequency applied to the motor corresponds to 7.57 V.

6.3 Problem analysis

The control problem can be described as:

Synchronize the slave motor with the master motor in such a way that the start of the product cycle time of the master equals the start of the product cycle time of the slave. The start of the product cycle time is indicated by the one pulse on the load axis. A defined phase shift between the two is allowed. This phase difference is hard to describe as a time difference because it depends then on the velocity of the motors. Therefore it is described using the motor positions which are taken relatively to the product cycle start times.

$$\theta_s = \theta_m + \varphi \qquad (6\text{-}5)$$

This synchronization must be achieved using a minimal pulse count p_s for the slave encoder.

From the process description some velocity relations can be derived

$$\omega_m = \frac{\omega_{mm}}{n_m} \qquad (6\text{-}6)$$

$$\omega_s = \frac{\omega_{ss}}{n_s} \qquad (6\text{-}7)$$

and from the process cycle time the desired master velocity can be calculated

$$\omega_m = \frac{2\pi}{T_{cm}} \qquad (6\text{-}8)$$

The position encoders are connected to the motor axis. The motor axis makes n_m (n_s) revolutions compared to one revolution of the load axis. Therefore θ_m (θ_s) can be measured in stepsizes according to equation 6-9.

$$\Delta\theta_m = \frac{2\pi}{n_m p_m}$$
$$\Delta\theta_s = \frac{2\pi}{n_s p_s} \qquad (6\text{-}9)$$

The maximum rotation frequency of the motor is 70.4 Hz.

6.4 Master-slave synchronization

The master slave synchronization is a synchronization of the position of the load axis. So the goal is to reduce the position error between the master and slave axis.

The motor position can be detected with the resolution of the encoder stepsize according equation 6-9. This resolution is the limitation on the accuracy of the position measurement. The detected position θ_d of the master and slave axis can be up $\Delta\theta$ to smaller than the real position θ_r.

$$\theta_d \le \theta_r < \theta_d + \Delta\theta \qquad (6\text{-}10)$$

The fact that only a positive error is made in the position detection implies that the position error between the master and slave can be detected with an accuracy according equation 6-11.

$$-\Delta\theta_s < \theta_m - \theta_s < +\Delta\theta_m \qquad (6\text{-}11)$$

From equation eqn follows that decreasing the number of slave encoder pulses decreases the accuracy of the position measurement error. Only when θ_d equals θ_r the position error can be determined with an accuracy of $+\Delta\theta$. From this observation follows the idea to control the slave in an asynchronous way at the time instants that the detected slave position θ_{ds} equals the real slave position θ_{rs}.

For the solution of the problem three different approaches are taken.

- Synchronous controller and maximum resolution of the slave encoder
- Synchronous controller with minimal resolution of the slave encoder
- Asynchronous controller with minimal resolution of the slave encoder

The synchronous solution is the normal control application with fixed sample frequency. At each sampling instant the position error between the master and the slave is calculated. The goal of the control action is to force the error to zero. If this error is kept zero in steady state (or zero averaged in time) the master and slave velocity will also be the same. If the position error correction is performed by the slave controller, the master controller can be regulated in velocity. This results in the possible control application architecture of figure 6-4. This application structure is the basis for all three proposed solutions.

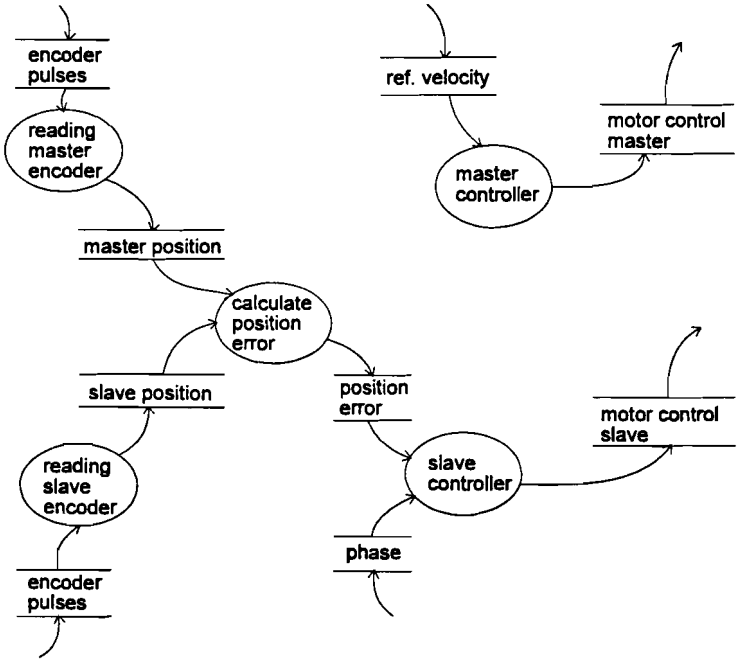

figure 6-4 : Synchronous solution

The synchronous controller is tested in two different situations, first with a full resolution slave encoder and secondly with smaller resolution slave encoder. The asynchronous controller is basically the same except that the slave controller is triggered by the pulses from the slave position. This implies that the control action is calculated asynchronous in time.

6.5 The encoder reading and pulse detection

The reading of the position encoders is performed by the DS5001 Digital Waveform Capture Board. It is possible to connect up to 8 incremental encoders to the DWC. All logic necessary to determine the direction of rotation has been moved to the hardware, thus reducing the software requirements to the minimum of counting a variable up or down.

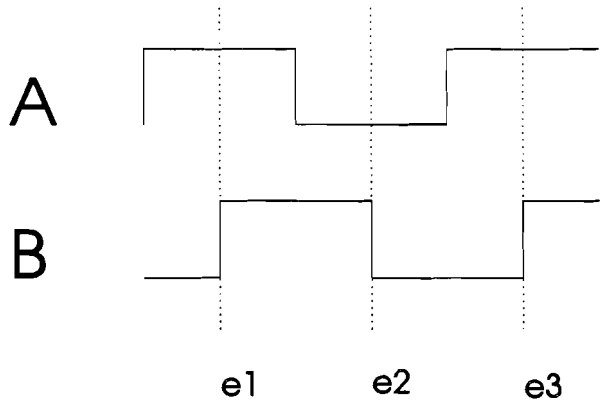

figure 6-5 : Position encoder signals A and B (counterclockwise)

An incremental encoder outputs two square wave signals (channel A and B) which have a phase offset of +90° or -90° depending on the direction of rotation.

Clockwise rotation is indicated by

- rising edge at channel B and low level at channel A
- falling edge at channel B and high level at channel A

Counterclockwise rotation is indicated by

- rising edge at channel B and high level at channel A
- falling edge at channel B and low level at channel A

Thus the encoder movements can be described as:

- movement of any edge on channel B
- direction =(level channel A) XOR (level channel B) after edge at channel B

The 16 channels of the DWC can be used in eight pairs for the incremental encoder feature (CH 1/2, 3/4, 5/6, ...). Each of these channel pairs can be programmed as incremental encoders. The direction bit with timestamp is stored in channel A (CH 1, 3, 5, ..) and the edge detection is stored in channel B (CH 2, 4, 6,...).

The additional software only has the increment or decrement a counter based on the direction bit in the event buffer of channel A.

note : There are two events detected on B during one pulse period of A this doubles the resolution of the encoder.

Besides the two channels A en B the encoder gives one additional signal Z. This signal gives one pulse when the encoder completed one revolution. This signal can be used to reset the position counters to zero to get a angular position counter.

The DS5001 DWC board has an another useful feature. It is possible to generate an interrupt from the DWC board to the DSP after an event buffer received a definable count of events. This is an advantages for the reset signals Z because if the reset pulse is received the DSP can respond immediately by resetting the position counters. If the reset pulse was detected by polling some count pulses might have occurred after the reset pulse has occurred but before it is detected.

These interrupt mechanism can also be used to emulate encoders with less pulses per revolution. If the event count to generate an interrupt is set for channel A or B of the encoder reading, some pulses can be skipped and by that reducing the number of counted pulses.

There is one disadvantages in this interrupt mechanism, all event buffers generated the same interrupt to the DSP. There is a register on the DWC board containing the channel number that generated the interrupt. When two interrupts occur at almost the same time only one of them is registered as causing the interrupt. This means that when an interrupt occurs the software has to poll if other channels also need servicing.

Besides the pulse detection and the interrupt mechanism the DS5001 DWC board contains three 32 bit counters which can be used to count the number of events occurred on a channel. These counters can be used as position counters because the motors can turn only in one direction. That means that there is no need to pay attention to the direction bit in channel A. Each event means just that the motor has moved one position.

Besides the two encoders with their reset pulses there are also the pulse detectors for detecting a revolution of the load axis of both motors. These pulse detectors are also connected to the DS5001 DWC board resulting in the channel usage according table 5-2.

table 6-2 : channel setup for Digital Waveform Capture board

DWC channel	input function
1-2	master encoder A+B
3-4	slave encoder A+B
5	master encoder Z
6	slave encoder Z
7	pulse detector master load axis
8	pulse detector slave load axis

6.6 The velocity observer

The normal operation to obtain the angular velocity ω from the position signal θ_m is to differentiate the position signal. In discrete time the difference equation 6-12 describes the differentiator. The disadvantages of this way to calculate the velocity is the noise sensitivity. A small error in the position measurement results in a difference in the velocity of the error times the sample frequency.

$$\omega_m(k) = \frac{1}{T}\{\theta_m(k) - \theta_m(k-1)\} \qquad (6\text{-}12)$$

To circumvent this problem a new approach is chosen by using a velocity observer (figure 6-6). The notion of this velocity observer is to track the motor position. If the predicted motor position and the real motor position θ_m are equal than the input of the last integrator equals the velocity ω_m of the motor. The disadvantages of the signal v as output is that there is a direct feed through in the system, i.e., the signal v is still sensitive to high frequency noise. This can be solved by choosing ω_m as the system output. If there is no error between the predicted position and the real position ω_m equals v.

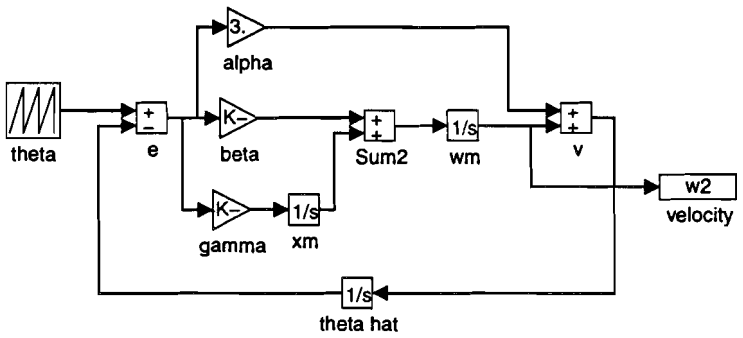

figure 6-6 : velocity observer

The observer can be described by the state space description of equation 6-13. With the state ω_m as output.

$$\begin{aligned}\dot{\hat{\theta}}_m &= \omega_m + \alpha\theta_m - \alpha\hat{\theta}_m \\ \dot{\omega}_m &= x_m + \beta\theta_m - \beta\hat{\theta}_m \\ \dot{x}_m &= \gamma\theta_m - \gamma\hat{\theta}_m\end{aligned} \qquad (6\text{-}13)$$

To determine the gain values α, β and γ the transfer function (equation 6-14) is more useful

$$H_{obsrv}(s) = \frac{\omega_m}{\theta_m} = \frac{s(s\beta + \alpha\beta + \gamma - \alpha\beta)}{s^3 + s^2\alpha + s\beta + \gamma} = \frac{s(s\beta + \gamma)}{s^3 + s^2\alpha + s\beta + \gamma} \qquad (6\text{-}14)$$

From equation 6-14 it can be seen that the observer behaves like a true differentiator for low frequencies. This equation can be rewritten in a more convenient way of equation 6-15.

$$H_{obsrv}(s) = \frac{\omega_m}{\theta_m} = \frac{s(s[1 + 2\varsigma k]\omega_n^2 + k\omega_n^3)}{(s + k\omega_n)(s^2 + 2\varsigma\omega_n s + \omega_n^2)} \qquad (6\text{-}15)$$

In this case the gain values α, β and γ can be expressed in k, ω_n and ζ

$$\alpha = (2\zeta + k)\omega_n$$
$$\beta = (1 + 2\zeta k)\omega_n^2 \qquad (6\text{-}16)$$
$$\gamma = k\omega_n^3$$

The design parameters k, ω_n and ζ determine the bandwidth for the frequency at which the velocity changes can be tracked.

$$v(t) = \hat{V}\sin(\omega t)$$
$$\theta(t) = \int v(t)dt = -\frac{\hat{V}}{\omega}\cos(\omega t) \qquad (6\text{-}17)$$

Equation 6-15 can now be used to created an overall frequency plot of the transfer function (figure 6-7).

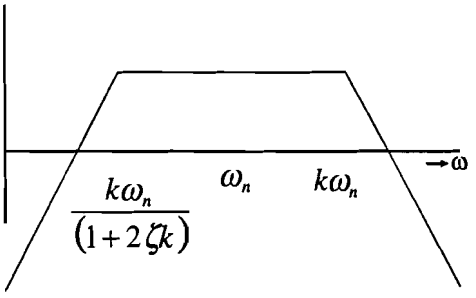

figure 6-7 : Desired Bode diagram velocity observer

The shape of the Bode diagram has the advantages that the observer behaves like a differentiator for low frequencies and also reduces the influence of high frequency components (quantization noise).

To test the behaviour of the observer the following parameters are used

$\omega_n = 18$

$k = 2$

$\zeta = 0.75$

The results in a theoretic Bode diagram for the observer with a passband of two octaves centered at ω_n. The true Bode diagram is shown in figure 6-8.

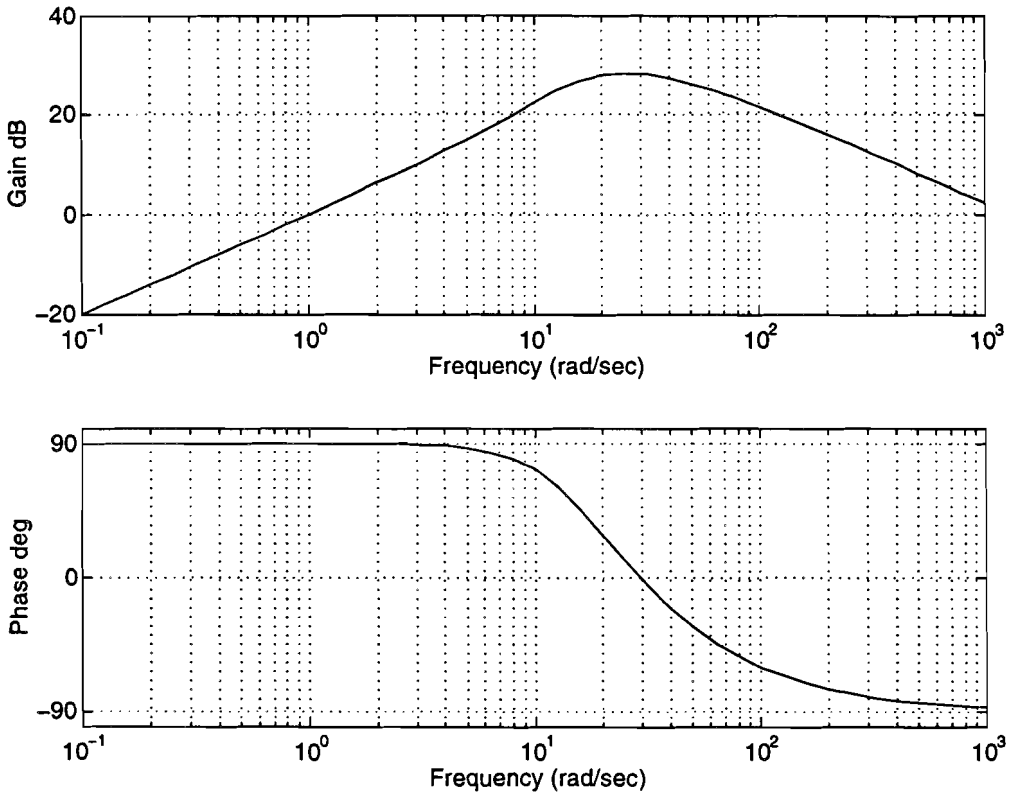

figure 6-8 : Bode diagram velocity observer

The behaviour of the observer is tested with two input signals. The first is a sine wave which is interpreted as a velocity reference ω_{ref}. The second uses a pulse with a limited rise time, which can be compared to the output of the frequency inverter. Both input signals are integrated to get a reference position which is the input to the observer. The integration time is chosen to be fixed at 0.0005 sec, this is done to make a good comparison to the discrete time case.

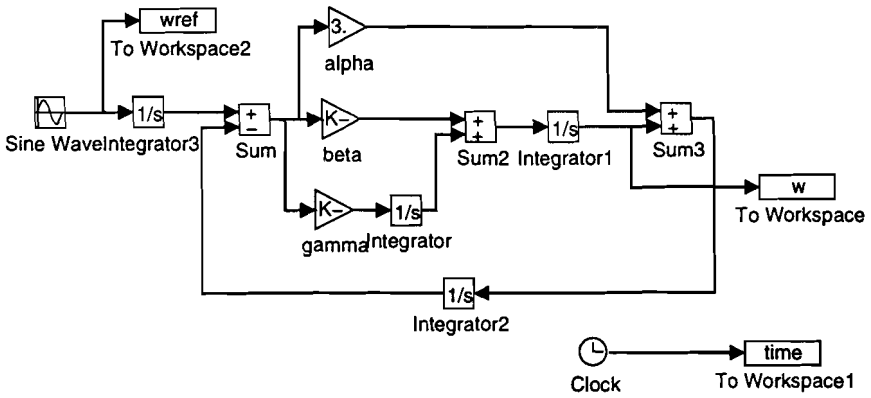

figure 6-9 : Simulation model for the continuous velocity observer

The first test using the sine wave as input signal, taken for ω_n is 18 and 50 rad/sec. Both are performed for four different input frequencies ω_{ref} is 1, 18, 50 and 100 rad/sec.

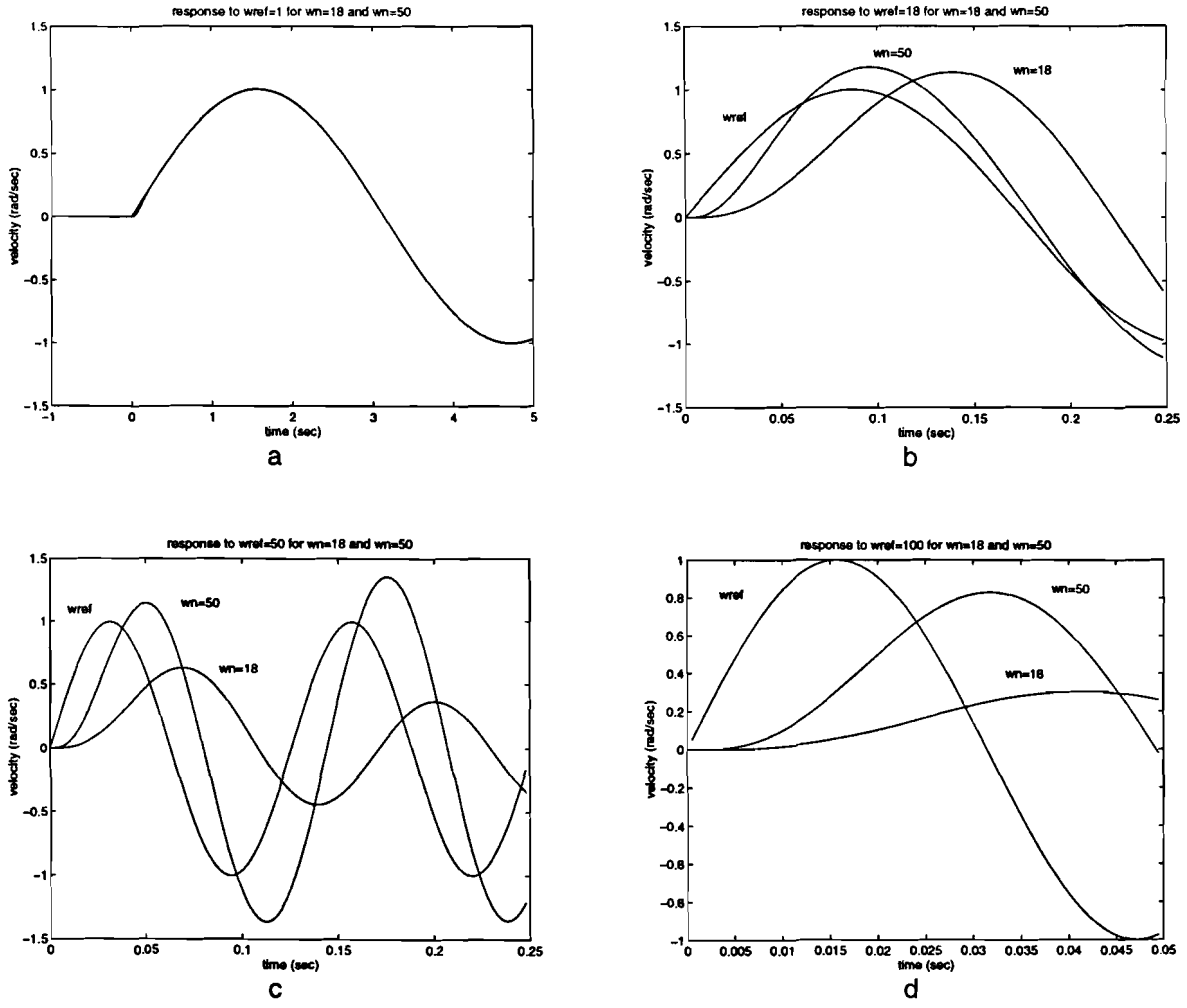

figure 6-10 : observer response to input signals of different frequency
(a) ω_{ref} = 1 rad/sec (b) ω_{ref} 0= 50 rad/sec (c) ω_{ref} = 18 rad/sec (d) ω_{ref} = 100 rad/sec

From the results a few observations can be made. First the different frequencies are traced by the observer, but all with their respective attenuation and phase shift. Secondly a higher ω_n results in a faster responds (less phase lag). The best choice for ω_n depends on the application, if there are fast velocity changes expected or not.

For the second test an input signal is used that rises from 0 to 50 rad/sec in two seconds. This corresponds to the maximum angular velocity of the load axis of the motor. The rise time from two seconds is the limitation of the frequency converter. The results are shows in figure 6-11.

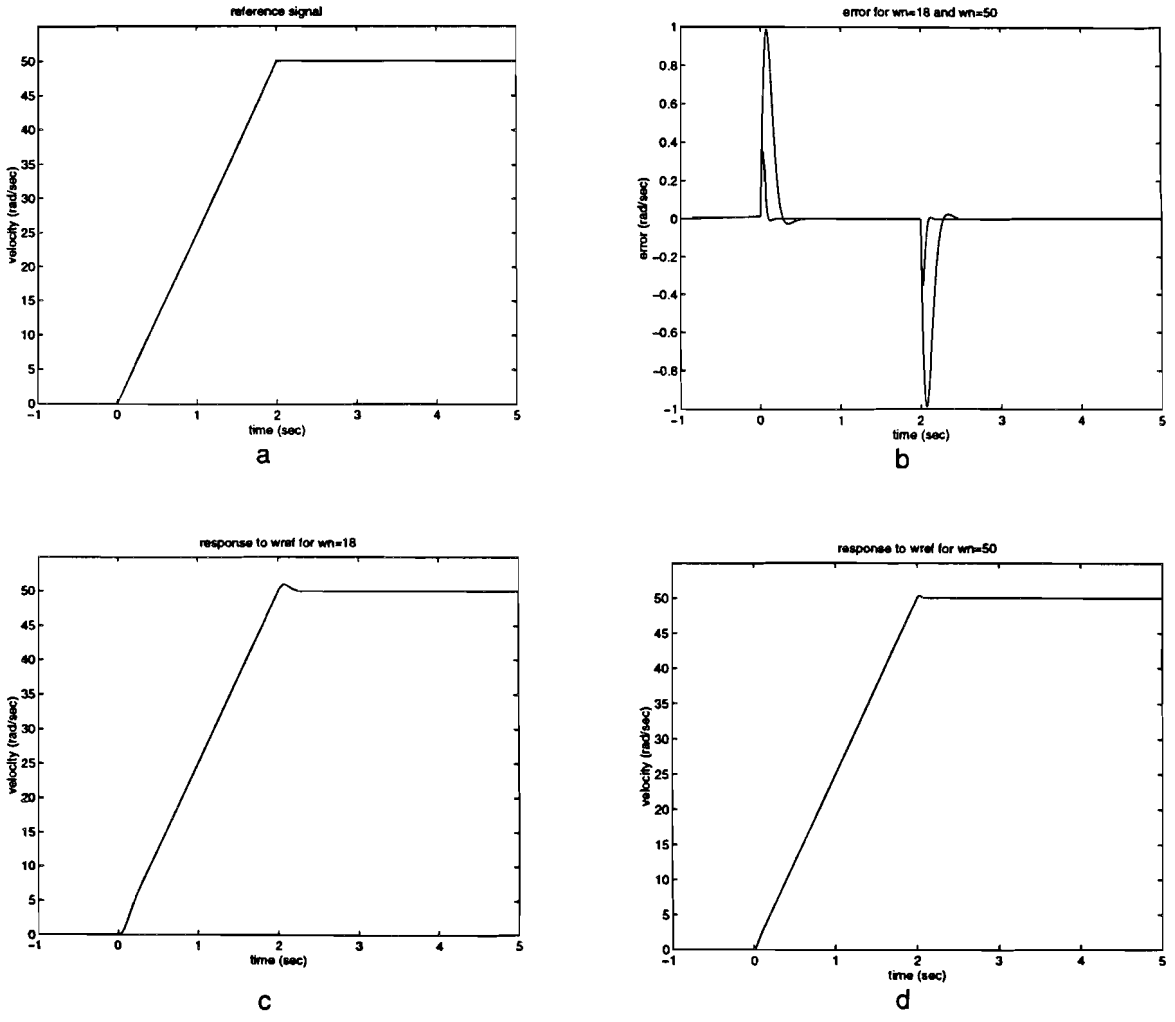

figure 6-11 : (a) ω reference. (b) error between the ω_{ref} and observer output.
(c) observer output for ω_n = 18 rad/sec. (d) observer output for ω_n = 50 rad/sec.

These results show, as expected, that a larger ω_n results in a faster and more accurate responds. Note that the observer is able to track the ramp shape position signal with zero steady state error.

6.6.1.1 Implementation on a digital computer

In order to implement this continuous observer model on a digital computer, it is necessary to approximate the integrals that appear in the control law. There are a few different approximations that can be used. The most commonly used are forward difference, backward difference and Tustin's approximation. Of these three the Tustin approximation is the most accurate.

$$s \rightarrow \frac{1}{h}\{1-q^{-1}\} = \frac{1}{h}\left\{\frac{q-1}{q}\right\} \qquad \text{Backward difference}$$

$$s \rightarrow \frac{1}{h}\left\{\frac{1-q^{-1}}{q^{-1}}\right\} = \frac{1}{h}\{q-1\} \qquad \text{Forward difference (Euler)}$$

$$s \rightarrow \frac{2}{h}\left\{\frac{1-q^{-1}}{1+q^{-1}}\right\} = \frac{2}{h}\left\{\frac{q-1}{q+1}\right\} \qquad \text{Tustin's approximation}$$

In these approximations is h the sampletime. The disadvantage of the backward difference and the Tustin's approximation is that these approximations introduce a direct feed through. This causes an algebraic loop in the closed loop path of the observer causing a calculation problem. This implies that at least one of the integral approximations in the closed loop must be a forward difference.

Two approximations are compared here. The first uses the forward difference approximation for all integral terms. The second uses the Tustin's approximation except for Integrator2 (calculation of $\partial\theta$ from ω_m). For this integrator the forward difference is used to prevent an algebraic loop.

The first approximation can easily formulated in a state space description (state two is the output).

$$\begin{pmatrix} x_m(k+1) \\ \omega_m(k+1) \\ \hat{\theta}_m(k+1) \end{pmatrix} = \begin{pmatrix} 1 & 0 & -h\gamma \\ h & 1 & -h\beta \\ 0 & h & 1-h\alpha \end{pmatrix} \begin{pmatrix} x_m(k) \\ \omega_m(k) \\ \hat{\theta}_m(k) \end{pmatrix} + \begin{pmatrix} h\gamma \\ h\beta \\ h\alpha \end{pmatrix} \theta_m(k) \qquad (6\text{-}18)$$

The second approximation can not directly be transformed into a state space description. Therefore the difference equations are given.

$$\begin{aligned} x_m[k] &= x_m[k-1] + \tfrac{h}{2}\{\gamma\theta_m[k-1] - \gamma\hat{\theta}_m[k-1] + \gamma\theta_m[k] - \gamma\hat{\theta}_m[k]\} \\ \omega_m[k] &= \omega_m[k-1] + \tfrac{h}{2}\{x_m[k-1] + \beta\theta_m[k-1] - \beta\hat{\theta}_m[k-1] + x_m[k] + \beta\theta_m[k] - \beta\hat{\theta}_m[k]\} \\ \hat{\theta}_m[k] &= \hat{\theta}_m[k-1] + h\{\omega_m[k-1] + \alpha\theta_m[k-1] - \alpha\hat{\theta}_m[k-1]\} \end{aligned} \qquad (6\text{-}19)$$

To compare these two approximations with each other and with the continuous case the previous test with an input signal that rises from 0 to 50 rad/sec in two seconds is used. The tests are performed for sample frequencies of 2 kHz (h=0.0005 sec) and 500 Hz (h=0.002 sec).

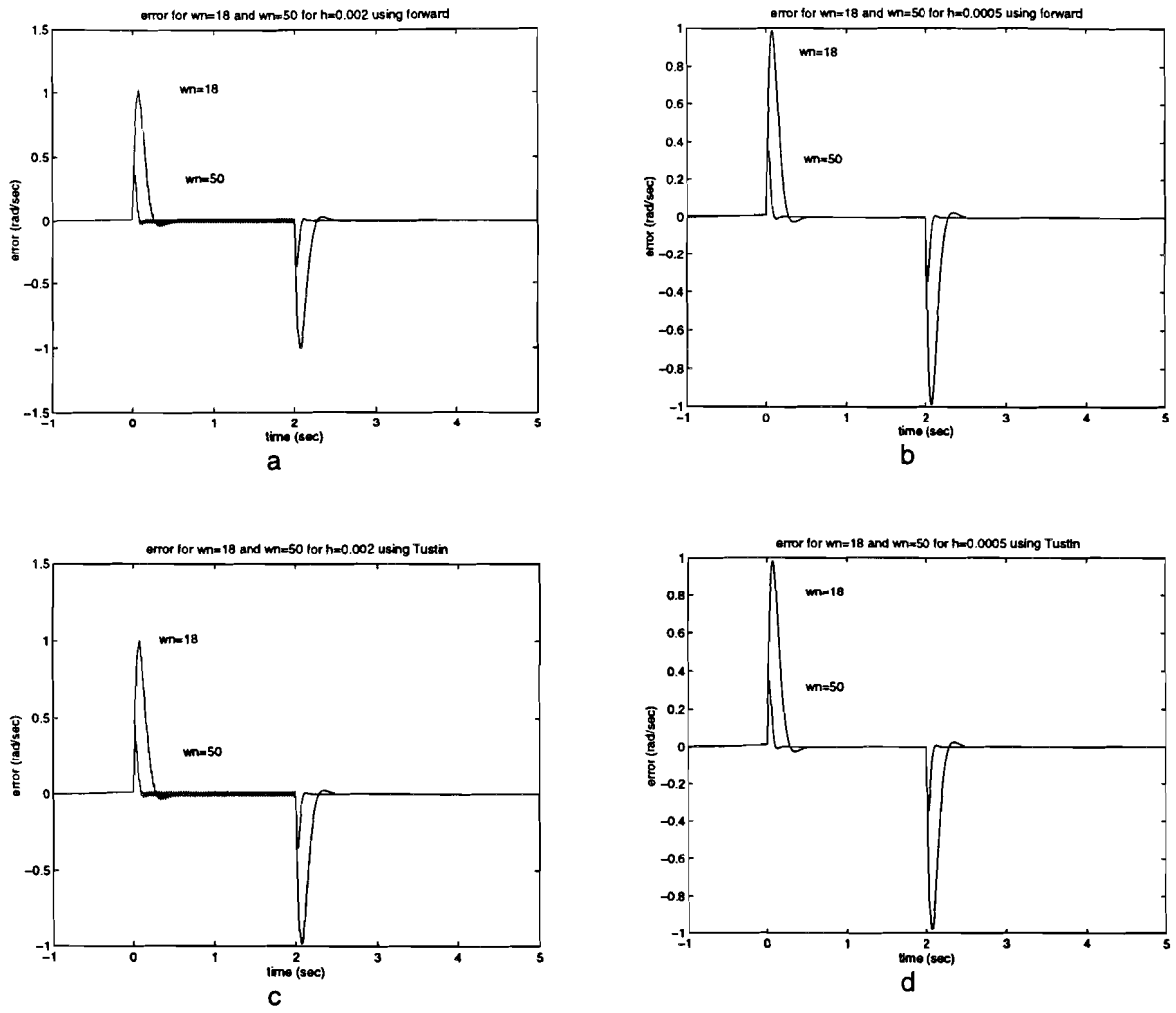

figure 6-12 : error in the response for the discrete observer for different sample frequencies (a) and (c) with 2 kHz sample frequency and (b) and (d) with 500 Hz sample frequency

The encoder currently used in the simulations has a continuous changing position. In the real world a position encoder subdivides the position steps based on the amount of encoder pulses. This effect is simulated by adding a quantization block after the position (theta) integrator. For the quantization level the position update of one encoder pulse is used (in radians).

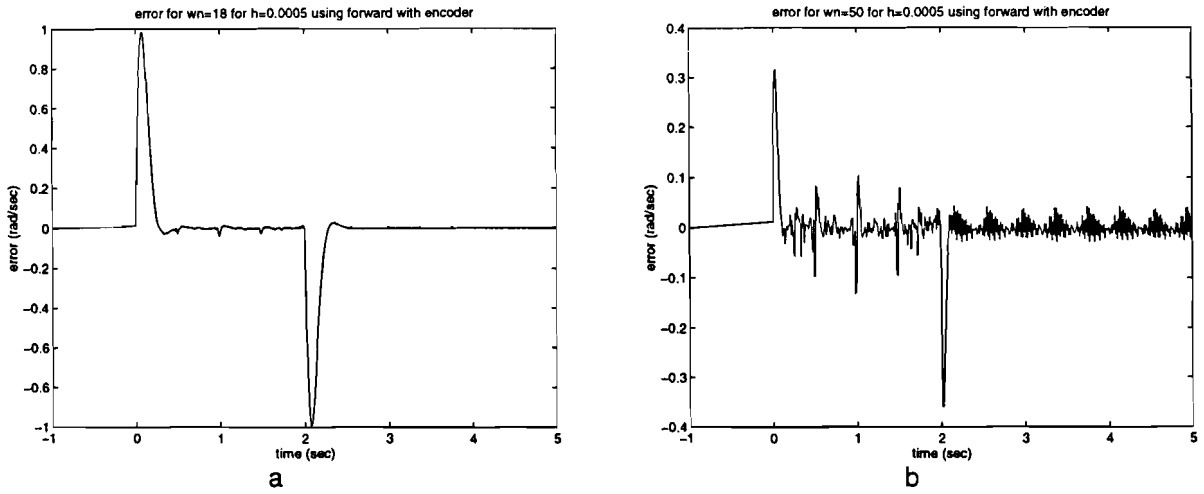

figure 6-13 : error in the observer responses using a discrete encoder with f_s 2 kHz.

When figure 6-13 b is compared to figure 6-13 a it is clear that increasing the ω_n results, as expected, in a response that is more sensitive to high frequency noise. Choosing the parameter ω_n is therefore a trade off between noise sensitivity and tracking error.

6.6.1.2 Implementation of the velocity observer

The velocity observer is implemented as an Simulink S-function. This has the advantages that it can be used for simulations as well. The S-function source code for the velocity observer is include in Appendix B. It can be used as a guide for writing other S-functions.

6.7 Motor identification

To design the controller for the motor velocity and synchronization it would be easy to have access to a model of the motor. Therefore an estimation of the motor model is made using identification techniques.

A general approach to estimate a model for a process is to apply an input signal to the process and measure its response. An estimation of the model of a LTI system can be made by measuring the response and compare this with the response simulated with the model. Good identification results are obtained by using an input signal containing every frequency. A white noise signal has a flat spectrum, that is it contains every frequency with equal energy.

When the test signal for identification is generated by computer than a PRBNS (Pseudo Random Binary Noise Signal) is used instate of the white noise. The spectrum of a PRBNS signal is only flat for a limited frequency range. This is sufficient for identification if this range is wider than the process bandwidth. The basis for a PRBNS signal is a random binary sequence, only the amplitude of this sequence is adjust to contain enough energy for the process. This spectrum of the signal is flat only for frequencies up to the maximum changing rate of the signal, i.e., the sample frequency at which the signal is generated. The Simulink model of figure 6-14 is used to acquire the needed data for identification.

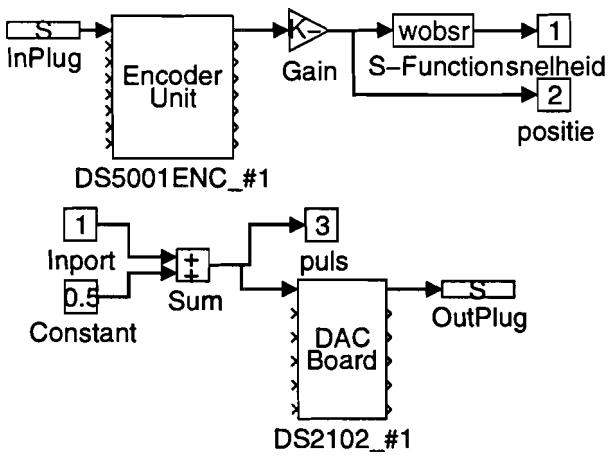

figure 6-14 : Simulink model used for identification

The binary noise generator from the signal generator library is used to generate the noise sequence. The call to this library function is include in the model.usr file. This gave the simplest way to include this small extension.

```
/*********************************************************************

   Include file meetmod.usr:

   Definition of macros for user defined initialization, system I/O,
   and background process code


   Copyright (c) 1993-96 by dSPACE GmbH

 *********************************************************************/
#include <slib40.h>
#include <flib40.h>

noise_parm_t nst;

#define      usr_initialize()                                  \
  noise_init(4,&nst);


#define      usr_input()                                       \
 u[0]=noise(&nst)/10


#define      usr_output()                                      \


#define      usr_background()                                  \


#define      usr_terminate()                                          \
        ds2102 (0x00000090, 0x00000001, 0);          \
```

The data acquisition is performed using TRACE with immediate data transfer to disk. The data acquisition is performed during a period of 30 seconds with a sampling frequency of 2 kHz. It should be noted that a model is estimated for the motor including the velocity observer. That is the output of the frequency converter is used as the input signal for the identification and the velocity output of the observer is used as the output signal.

The identification is performed with a parametric OE model [1]. This results in the transfer function of equation 6-18. This is a transfer function with as input the reference of the applied motor frequency and as output the angular velocity of the motor.

$$H_m(q) = \frac{0.479633q - 0.465873}{q^2 - 1.989419q + 0.989627} \qquad (6\text{-}20)$$

The response of the estimated model and the measured response is shown in figure 6-15.

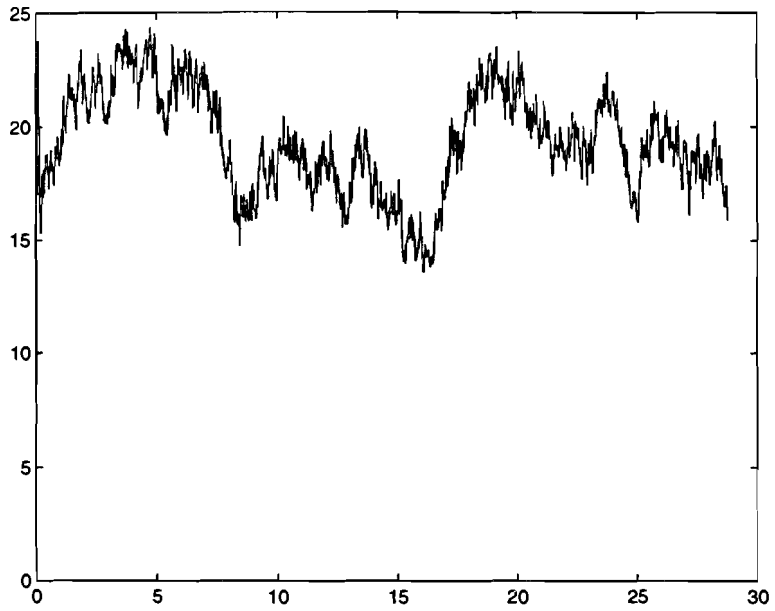

figure 6-15 : The estimated model response and the measured response

The step response of the estimated model is shown in figure 6-16. This response shows some overshoot which can be ascribed to the velocity observer.

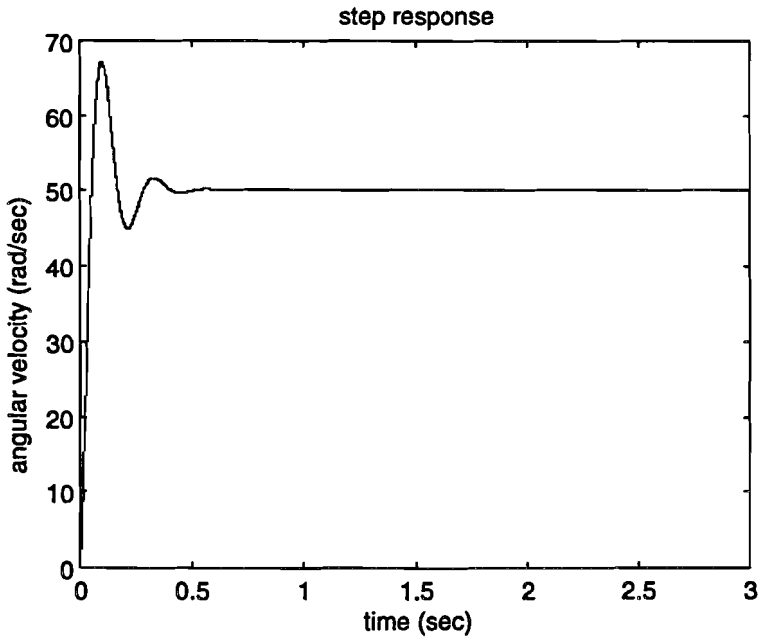

figure 6-16 : Step response of the estimated process.

The estimated model is used for simulations and for parameter tuning of the used PID controllers.

6.8 The PID controller

The PID controller is one of the most common control algorithms. It originates from the early analog controllers, but is nowadays mostly implemented on digital computers. The most common form of the PID controller is given in equation 6-21. This form is used in many text books.

$$H(s) = \left\{K_p + \frac{1}{\tau_i s} + \tau_d s\right\} \qquad (6\text{-}21)$$

In this controller form all terms act on the error input. This error signal is the difference between the reference input and the process output. This reference input will normally be constant. It will thus not contribute to the derivative term. However when the derivative term will change drastically when the reference is changed. For this reason it is common in practice to apply the derivative action only to the process input. The continuous derivative-of-output PID controller form is described by equation

$$U(s) = K\left\{E(s) + \frac{1}{\tau_i s}E(s) - \frac{s\tau_d}{1 + s\frac{\tau_d}{N}}Y(s)\right\} \qquad (6\text{-}22)$$

This controller is made discrete by using an Euler approximation of the integral part and a backward difference approximation of the derivative part. This results in

$$u(kh) = K\left\{e(kh) + \frac{h}{\tau_i(q-1)}e(kh) - \frac{\tau_d}{(h+\frac{\tau_d}{N})}\frac{q-1}{q-\frac{\tau_d}{Nh+\tau_d}}y(kh)\right\} \qquad (6\text{-}23)$$

Where u is the controller output, y is de process output, w is de reference signal, and e=w-y is de process tracking error. The controller parameters are the proportional gain K, the integral time constant (or reset time) τ_i, and the derivative time constant τ_d. The high frequency gain N is usually set between 7 and 10. Finally h is the sampling period of the controller.

The controller equation 6-23 can be rewritten in a common input-output form as

$$R(q)u(kh) = T(q)w(kh) - S(q)y(kh) \qquad (6\text{-}24)$$

To streamline the notation T_i, T_d, and γ are introduced.

$$T_i = \frac{h}{\tau_i}$$

$$T_d = \frac{\tau_d}{(h+\frac{\tau_d}{N})} \qquad (6\text{-}25)$$

$$\gamma = \frac{\tau_d}{(Nh+\tau_d)}$$

This simplifies equation 6-23 to

$$u(kh) = K\left\{e(kh) + \frac{T_i}{(q-1)}e(kh) - \frac{T_d(q-1)}{(q-\gamma)}y(kh)\right\} \qquad (6\text{-}26)$$

$$(q-1)(q-\gamma)u(kh) = K\left\{(q-1)(q-\gamma)e(kh) + T_i(q-\gamma)e(kh) - T_d(q-1)(q-1)y(kh)\right\} \qquad (6\text{-}27)$$

$$\begin{aligned}(q-1)(q-\gamma)u(kh) &= K(q-1+T_i)(q-\gamma)w(kh) \\ &\quad - K\left((q-1+T_i)(q-\gamma) + T_d(q-1)(q-1)\right)y(kh)\end{aligned} \qquad (6\text{-}28)$$

Equation 6-28 is in the common input-output form. So the controller polynomials become

$$R(q) = (q-1)(q-\gamma)$$
$$T(q) = K\left(q^2 + (T_i - 1 - \gamma)q - (T_i - 1)\gamma\right) \quad (6\text{-}29)$$
$$S(q) = K\left(\left(1 + T_d\right)q^2 + (T_i - 1 - \gamma - 2T_d)q + \left(T_d - \gamma T_i + \gamma\right)\right)$$

The closed loop system of the common control form is

$$y(kh) = \frac{TB}{AR + BS} w(kh) \quad (6\text{-}30)$$

In which the process model is

$$H_m(q) = \frac{B(q)}{A(q)} = \frac{0.479633q - 0.465873}{q^2 - 1.989419q + 0.989627} \quad (6\text{-}31)$$

So the closed loop process transfer function is

$$H_C(q) = \frac{TB}{AR + BS} \quad (6\text{-}32)$$

6.8.1 Discrete time pole placement

To find the controller parameters the pole placement technique is used. The goal of the pole placement technique is to place the poles of the closed loop transfer function at desired locations. This technique is described in [2], this general case has the disadvantages that there is no guarantee that the found polynomials comply with the PID polynomials. In order to be able to use the general PID controller a slightly different approach is taken. This approach is described in [3], except that there the T polynomial of the PID controller is omitted.

The pole placement technique specific for PID controllers is only suitable for systems up to second order. Continuous systems of second order can be described with three parameters. By using a PID controller, which also has three parameters, it is possible to arbitrarily place the three poles of the closed loop system. Sampling continuous systems of second order result in a discrete process description of also second order. (The estimated process model in the previous section has this form).

$$H_m(q) = \frac{B(q)}{A(q)} = \frac{b_1 q + b_2}{q^2 + a_1 q + a_2} \quad (6\text{-}33)$$

The desired closed loop process is described by the transfer function

$$H_d(q) = \frac{B_d(q)}{A_d(q)} \quad (6\text{-}34)$$

With pole placement it is possible to derive the controller polynomials which result in the desired process transfer function. The controller polynomials are the solution to equation.

$$H_d(q) = H_C(q) \quad (6\text{-}35)$$

Using equation (6-32) and (6-34) the next equation must be solved

$$AR + BS = A_d$$
$$TB = B_d \quad (6\text{-}36)$$

Using the process model and the controller polynomials in the general form

$$R(q) = (q-1)(q+r_1)$$
$$T(q) = t_0q^2 + t_1q + t_2 \qquad (6\text{-}37)$$
$$S(q) = s_0q^2 + s_1q + s_2$$

The characteristic equation becomes

$$(q^2 + a_1q + a_2)(q-1)(q+r_1) + (b_1q + b_2)(s_0q^2 + s_1q + s_2) = 0$$
$$q^4 + (a_1 + r_1 - 1 + b_1s_0)q^3 + (a_2 + a_1r_1 - a_1 - r_1 + b_1s_1 + b_2s_0)q^2 + \qquad (6\text{-}38)$$
$$(a_2r_1 - a_2 - r_1a_1 + b_1s_2 + b_2s_1)q + (b_2s_2 - a_2r_1) = 0$$

This characteristic equation must equal the characteristic equation of the desired closed loop system. For the desired characteristic equation the polynomial of equation 6-39 is used (taken from [2]).

$$A_{d=}(q+\beta)^2(q^2 + p_1q + p_2) = 0$$
$$A_{d=}q^4 + (2\beta + p_1)q^3 + (\beta^2 + 2\beta p_1 + p_2)q^2 + (\beta^2 p_1 + 2\beta p_2)q + \beta^2 p_2 = 0$$

where

$$\beta = -e^{-\alpha\omega h} \qquad (6\text{-}39)$$
$$p_1 = -2e^{-\varsigma\omega h}\cos\left(\omega h\sqrt{1-\varsigma^2}\right)$$
$$p_2 = e^{-2\varsigma\omega h}$$

This corresponds to a fourth-order system having two dominant poles with relative damping (ζ) and frequency (ω), and two real poles located in $-\alpha\omega$. This discrete time polynomial is constructed by specifying the continuous poles and using pole mapping to retrieve the polynomial.

The parameters for the polynomials can be solved by equating 6-38 and 6-39. This results in the general solution matrix:

$$\begin{pmatrix} 1 & b_1 & 0 & 0 \\ a_1 - 1 & b_2 & b_1 & 0 \\ a_2 - a_1 & 0 & b_2 & b_1 \\ -a_2 & 0 & 0 & b_2 \end{pmatrix} \begin{pmatrix} r_1 \\ s_0 \\ s_1 \\ s_2 \end{pmatrix} = \begin{pmatrix} 1 + 2\beta + p_1 - a_1 \\ a_1 + \beta^2 + 2\beta p_1 + p_2 - a_2 \\ a_2 + p_1\beta^2 + 2\beta p_2 \\ p_2\beta^2 \end{pmatrix} \qquad (6\text{-}40)$$

From the equations 6-37 and 6-29 the solution matrix for K, T_i and T_d (r_1=-γ) can be derived.

$$\begin{pmatrix} 0 & -1 & s_0 \\ -1 & 2 & s_1 \\ -r_1 & =1 & s_2 \end{pmatrix} \begin{pmatrix} Ti \\ Td \\ 1/K \end{pmatrix} = \begin{pmatrix} 1 \\ r_1 - 1 \\ -r_1 \end{pmatrix} \qquad (6\text{-}41)$$

In this derivation the polynomial T is omitted, but for a PID controller it is necessary. The T polynomial is determined completely by the currently found parameters (equation 6-42). This implies that there is no possibility to specify the zeros of the closed loop transfer function. So there could originate a transfer function with undesired behaviour like a non minimum phase transfer function.

$$T(q) = K\left(q^2 + (T_i - 1 - \gamma)q - (T_i - 1)\gamma\right) \qquad (6\text{-}42)$$

Using equation 6-25 the continuous time PID parameters τ_i, τ_d and N can be retrieved.

$$\tau_i = \frac{h}{T_i}$$
$$\tau_d = T_d h\left(1-\frac{\gamma}{\gamma-1}\right) \qquad (6\text{-}43)$$
$$N = T_d\left(1-\frac{\gamma-1}{\gamma}\right)$$

6.8.2 Anti wind-up techniques

All process are submitted to constraints. For instance a controller works in a limited range of 0-10 V, a motor driven actuator has a limited speed. Such constraints are usually referred to as plant input limitations. The most common types of limitations are magnitude and rate limitation. The magnitude and rate limitations can be described by the following two equations:

$$u^r = \begin{cases} u_{max} & ; u > u_{max} \\ u & ; u_{min} \le u \le u_{max} \\ u_{min} & ; u < u_{min} \end{cases} \qquad (6\text{-}44)$$

$$\frac{\partial u^r}{\partial t} = \begin{cases} & ; \frac{\partial u}{\partial t} > v_{max} \\ v_{max} & \\ v & ; v_{min} \le \frac{\partial u}{\partial t} \le v_{max} \\ v_{min} & \\ & ; \frac{\partial u}{\partial t} < v_{min} \end{cases} \qquad (6\text{-}45)$$

As a result of these limitations, the real plant input is temporarily different (saturation) from the controller output. In that case the system performs operates in open loop since the feedback path is broken. If the controller is unstable this may give severe consequences. A PID controller is a typical example of a controller that may cause instability or poor transient output during saturation.

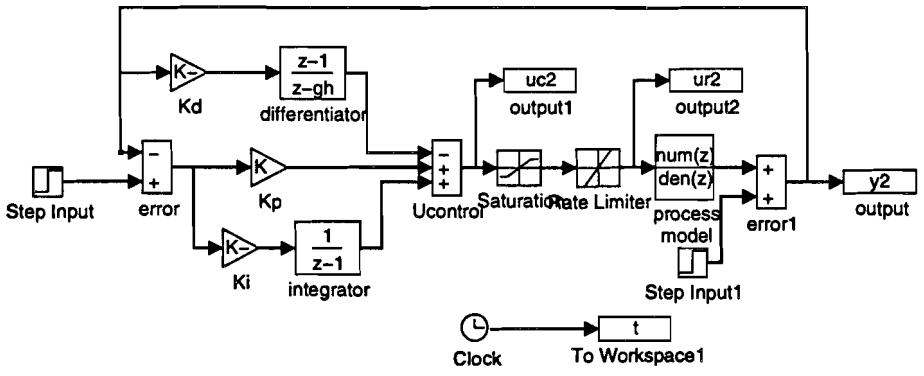

figure 6-17 : Limited closed loop system

For PID controllers the wind-up effect can be described as follows. A step change in the reference w causes a jump in the controller output u due to the proportional gain K. If in that case the actuator saturates at its limit, the process input u^r becomes smaller (considering a positive jump and K>0) than u, and the process output y is slower than in the unlimited case. Due to the slower y, the error signal e decreases slowly. The integral term increases much more than in the unlimited case and

becomes large. When y approaches w, u still remains saturated due to the large integral term. u starts decreasing after the error has been negative for sufficiently long time. This leads to a large overshoot and a large settling time.

To illustrate this phenomenon a simulation is used. For the simulation the estimated process model is used. While the frequency inverter and the DA converter are used as the limitations. The DA-converter gives some magnitude limitation of u_{max}=1 and u_{min}=0. In the dSPACE system these values correspond to with maximum and minimum output voltage of the DA-converters. The rate limiter of the frequency inverter is included with v_{max} = 0.5 sec^{-1} and v_{min}= -0.5 sec^{-1}. This is also a scaled version of the true limitations. This results in the simulation model of figure 6-17.

The response of the unconstrained process to a step reference starting at 1 sec and a distortion start at 3 seconds, is shown in figure 6-18.

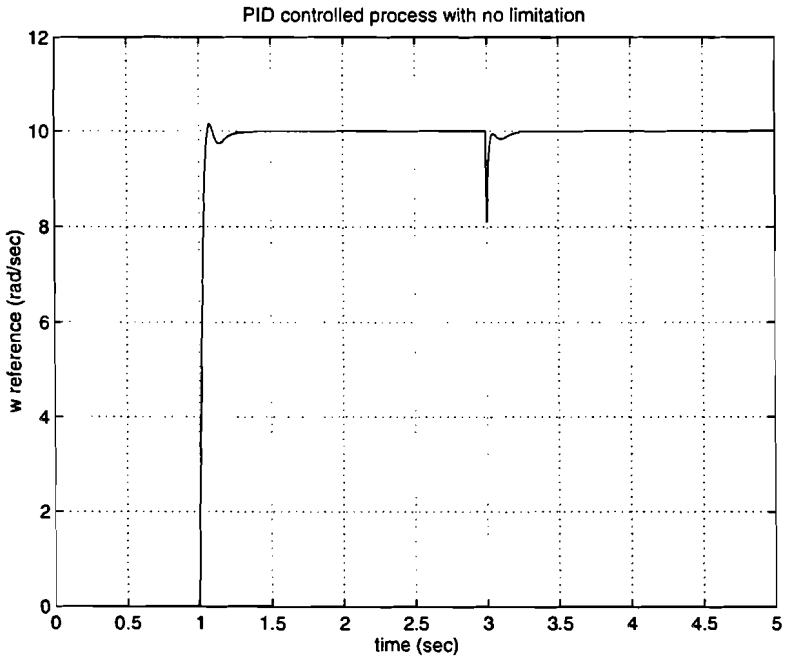

figure 6-18 : Simulation results for a PID controller with unlimited plant input

Figure 6-19 shows the response of the limited process. The large overshoot is very clear when the response is compared to figure 6-18.

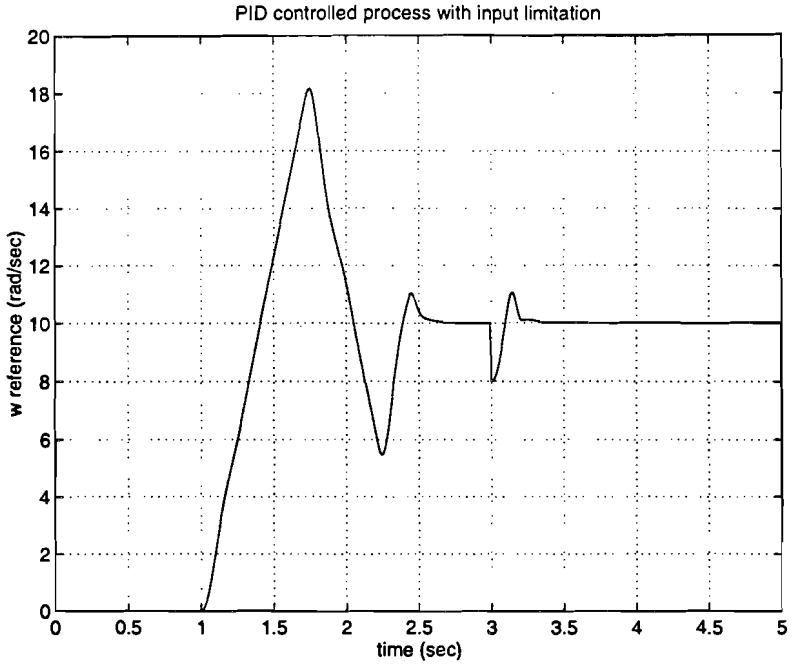

figure 6-19 : Simulation results for a PID controller with limited plant input

As an anti wind-up technique is the so called conditioning technique [4] used. In the case of input saturation the controller output is unequal to the process input. Here the internal state of the controller is not correct. In the conditioning technique the internal state is corrected by adjusting the reference input of the controller.

The response of controlled process using a PID controller with the anti wind-up scheme is shown in figure 6-20. When the results are compared to the previous result the improvement is obvious.

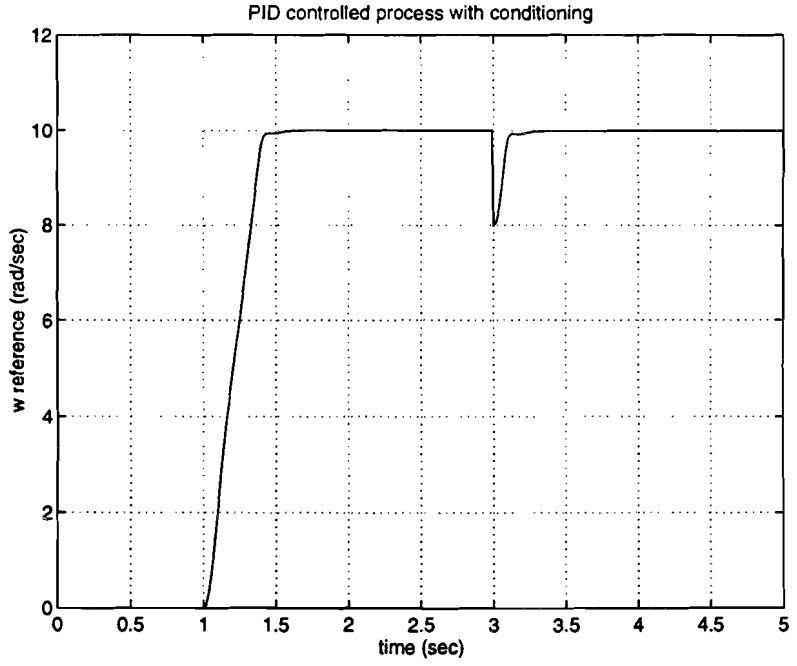

figure 6-20 : Simulation results for a PID controller with Anti Windup on a limited plant input

6.9 Master motor controller

For the master motor the angular velocity must be controlled. As a controller for he master motor a PID controller with an anti wind-up scheme is used. The parameters for the PID controller are determined using the discrete pole placement technique as described in the previous section.

This results in the following PID parameters

K =	0.0083
Ti =	0.0049
Td =	10.19
γ =	0.9573
T =	0.0005

These are the parameters for the discrete version of the PID controller (equation 6-26). The simulation results of this controller on the estimated model are shown in figure 6-20. Figure 6-21 shows the results of the PID controller on the real process.

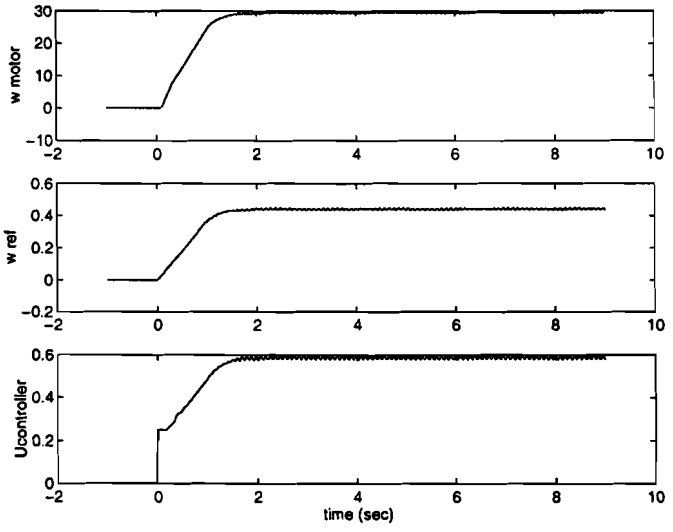

figure 6-21 : Response of the master motor using a PID controller tuned by pole placement

The calculated PID parameters result in a oscillatory response of the process. (In the figure 6-21 is this hard to see). This response is not useable in this application because the master control signal is used as a feed forward for the slave motor. The oscillatory effect indicates that the proportional gain is probably too high. Therefore the gain is a little adjusted. The parameter Td is probably a little high to. The parameters for the PID controller are therefore adjusted to.

K =	0.005
Ti =	0.005
Td =	1
γ =	0.1
T =	0.0005

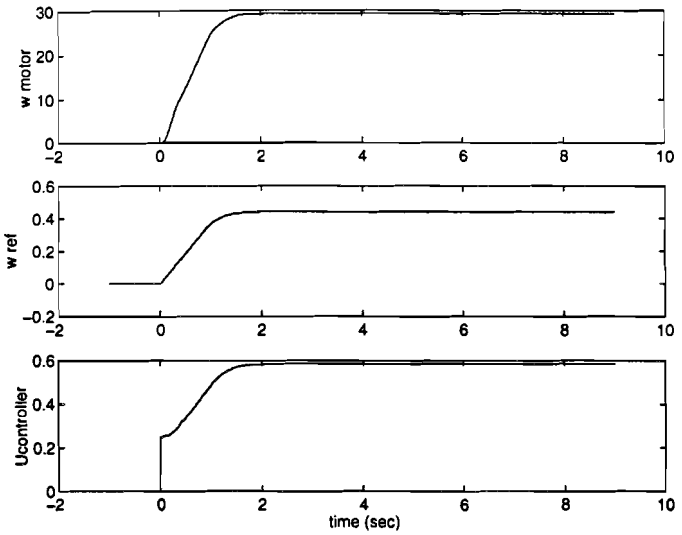

figure 6-22 : Response of the master motor using a PID controller with adjusted parameters

This results in a PID controller which can be used in the master slave synchronization application, because there is no noise feed to the slave motor in this case.

6.10 Slave motor controller

The slave controller is implemented in three different ways. The distinction between these implementation is on how the controller is implemented. In the synchronous case a simple PID controller is implemented in Simulink. In the asynchronous case is the same PID controller implemented in the interrupt routine triggered by the pulses of the slave encoder.

6.10.1 Synchronous implementation

There are two different situations tested with the synchronous implementation. The first uses the full slave encoder resolution to detect the position error between the master and the slave load axis. In this case two of the hardware counters of the DS5001 board are used to count the slave position and the master position. An interrupt routine is used to detect the one pulse of the load axis. The interrupt generated by the load pulse resets the corresponding counter. The difference between the master and slave position is used as input to the PID controller. The Simulink model contains the PID controller and the reference input for the master controller and is shown in figure 6-23. In this model also two velocity observers are included for reference. These observers are not part of the control loop.

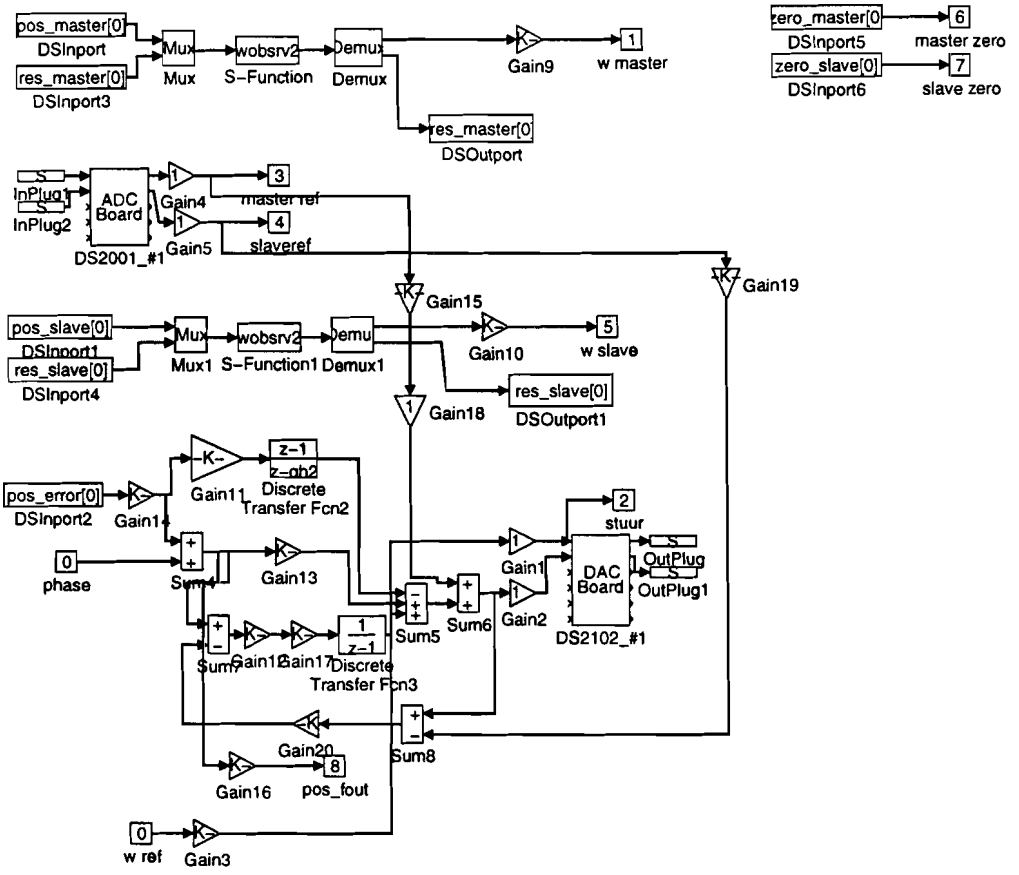

figure 6-23 : Simulink model for synchronous master-slave controller

6.10.2 Results

The control structure of figure is 6-23 is used with only the P and I action of the controller. The results of this controller to a step change in the angular velocity of the master are shown in figure 6-24. These results are used to verify the results of the other implementations.

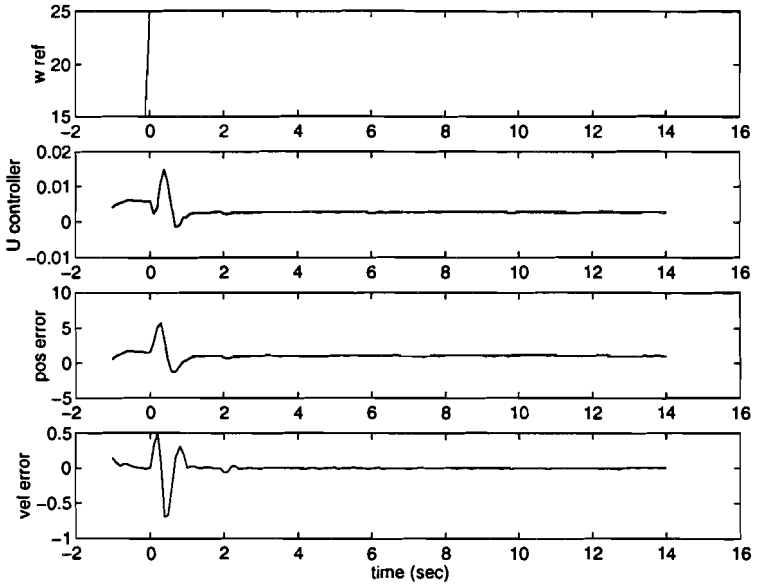

figure 6-24 : Results of the synchronous controller with full slave encoder resolution

The second synchronous controller uses only 16 pulses of the slave encoder resolution. This is done by using the interrupt feature of the DS5001 board which can generate an interrupt after a definable number of events. The full event resolution of the encoders is 1024 pulses/revolution. The event interrupt generator is therefore programmed to generated an interrupt after 64 events (pulses of the slave encoder). The interrupt routine contains the position counter for the position of the slave load axis.

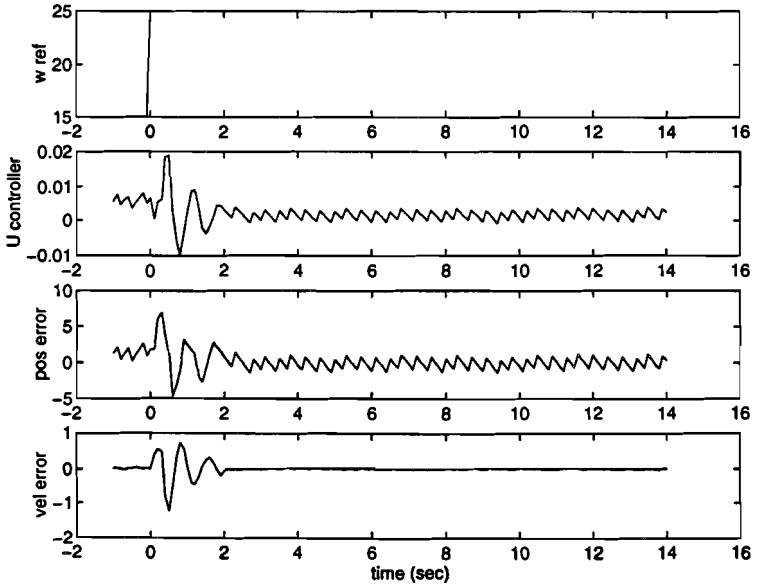

figure 6-25 : Results of the synchronous controller with a slave encoder resolution of 16 pulses/revolution

The signals presented in the results have the following dimensions

ω_{ref}	Hz
U controller	1/10 Volt
position error	degrees
velocity error	rad/sec

From the results of the slave controller with a minimal resolution of 16 pulses/revolution is clear that the position error starts to increase. This saw tooth like error signal is due to the fact that a change of the master position increases the position error immediately, while only once every 64 master pulses the slave position is adjusted.

In both cases of the synchronous controller a PI controller is used with the following parameters.

K =	0.1
Ti =	0.01
T =	0.0005

These parameters are determined by manual tuning.

6.10.3 Asynchronous implementation

The asynchronous controller uses an interrupt given by the encoder pulses of the slave encoder, at these time instants the control action is calculated (i.e. asynchronous in time). The complete asynchronous controller is implemented in the interrupt routine of the DS5001 board. Only the summation of the slave control signal and the feed forward from the master is added in the Simulink model. The advantage is that in that case the control signal is changed with a sampling frequency of 2 kHz. So the feed forward is almost applied continuously. The source code of the asynchronous controller is included in appendix A.

For testing the asynchronous solution an encoder of 4 pulses per slave motor revolution is used. With a manually tuned PI controller (K = 0.12 and Ti = 0.06). This tuning can be performed easily.

As a velocity reference for the master motor two speeds are used, 15 Hz and 30 Hz resulting in an axis velocity of 10.6 rad/sec and 21.4 rad/sec.

For each of the two velocities four tests are performed

- a step change of 10 Hz in the master motor velocity.
- an attenuation of the feed forward of the master motor to the slave
- a positive phase shift
- a negative phase shift

The results of these tests are shown on the next pages.

6.10.4 Results

The signals presented in the results have the following dimensions:

ω_{ref}	Hz
U feedforward	1/10 Volt
U slave controller	1/10 Volt
position error	degrees
velocity error	rad/sec
phase	degrees

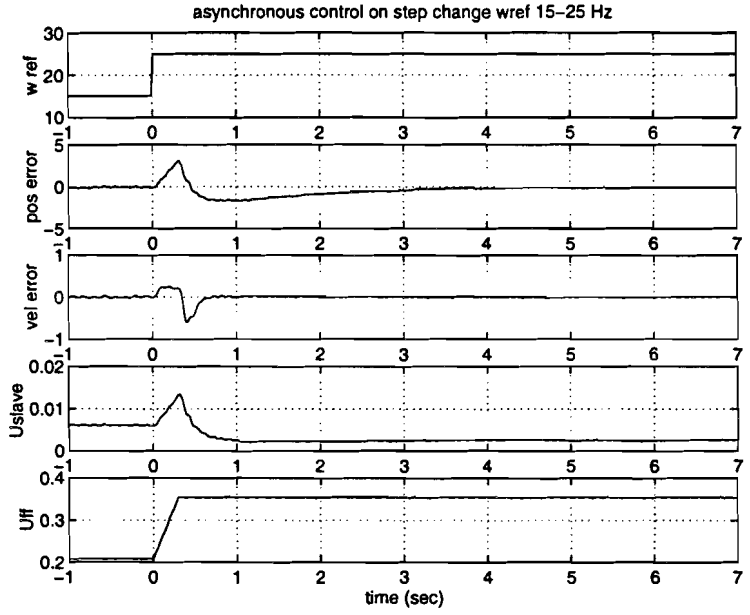

figure 6-26 : results of asynchronous control on step change in ω_{ref} from 15 to 25 Hz

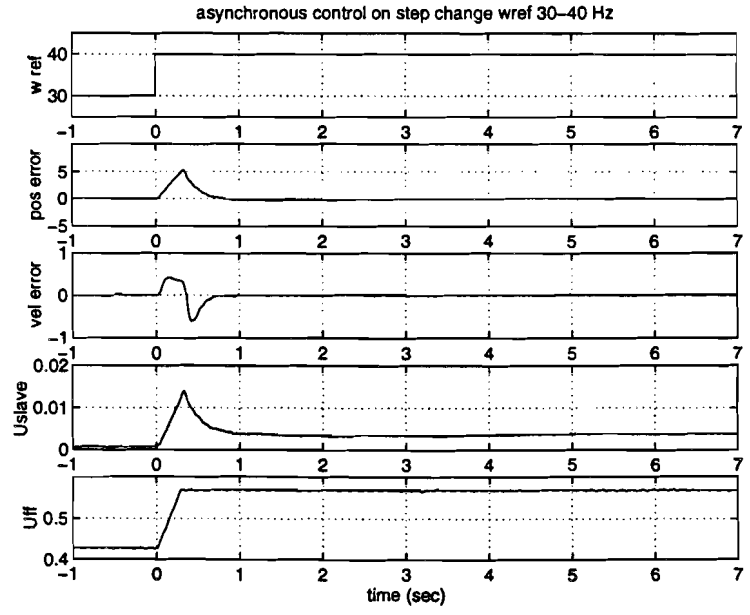

figure 6-27 : results of asynchronous control on step change in ω_{ref} from 30 to 40 Hz

The results of figure 6-26 and 6-27 show smoother responses of the error signals between the master and slave positions and velocity than in the synchronous case (compare to figures 6-24 and 6-25). From figure 6-26 and 6-27 some peculiar observations can be made. First the settling time in figure 6-26 is about 4 seconds and in figure 6-27 this is about 1 second. A higher angular velocity of the motor results in a faster settling time, because the sampling time (the time between two encoder pulses in the asynchronous case) is not used in the asynchronous controller. The is synchronous controller with the full resolution slave encoder has a faster settling time than the asynchronous controller.

Secondly it looks in the asynchronous case that there is no steady state error in the position error. This is not expected because the controller is a tracking the position of the master motor. To get a zero steady state error in these tracking controllers a double integrator is needed. In the used controllers just one integrator (I-action) is used, that is the reason that the synchronous controller has a steady state position error (figures 6-24 and 6-25).

The results of the first test of the asynchronous controller are so much better than in the synchronous case with a small slave encoder resolution. Therefore some additional measurements are performed. In the following test is the feedforward signal from the master motor changed.

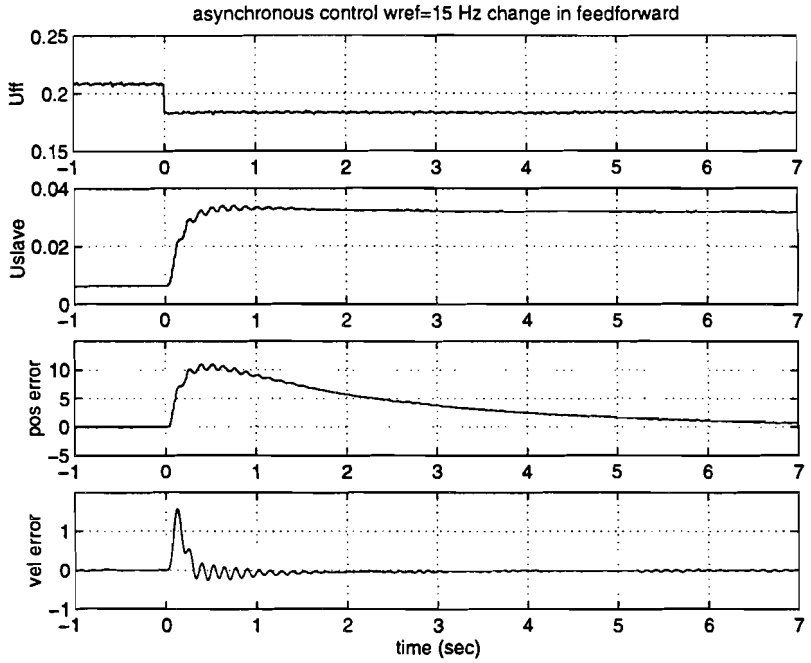

figure 6-28 : results of asynchronous control on step change in master feed forward at ω_{ref} = 15 Hz

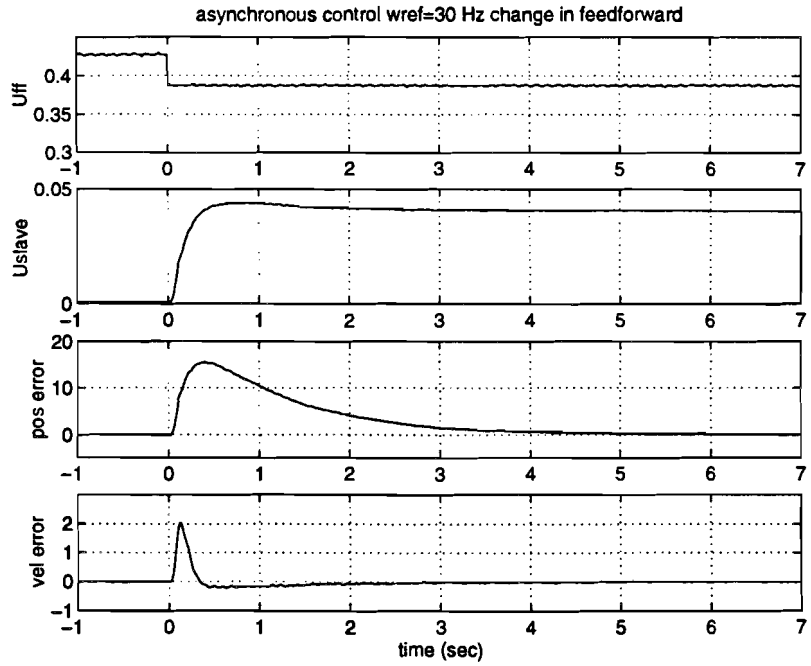

figure 6-29 : results of asynchronous control on step change in master feed forward at ω_{ref} = 30 Hz

Changing the feedforward can be interpreted as an external load applied to the slave motor. The slave controller must compensate the position error when the feedforward is decreased. In this case the feedforward is changed with a step, which is not expected in practice. The results in the figures 6-28 and 6-29 show fast adjustment of the position error in both cases. Figure 6-28 shows some distortion with a frequency of about 10 Hz on the control signal of the slave controller. This looks like the same distortion as in the first test of the PID controller for the master motor

(figure 6-21). If the feedforward signal is observed more closely, the same frequency can be found. This is probably caused by the frequency inverter.

Finally a test is performed by changing the phase input. This input determines the phase difference between the start of the production cycle of the master load axis and the slave load axis. This test is performed using a positive and a negative phase shift. These test are performed while the motors are running. In practice, the phase shift will probably be set before the motors are started.

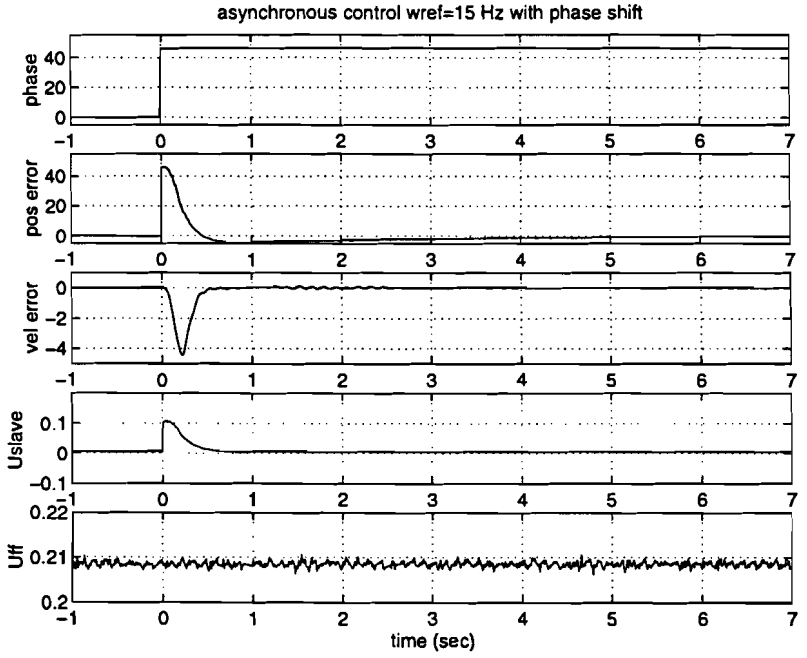

figure 6-30 : results of asynchronous control on positive phase shift with ω_{ref} = 15 Hz

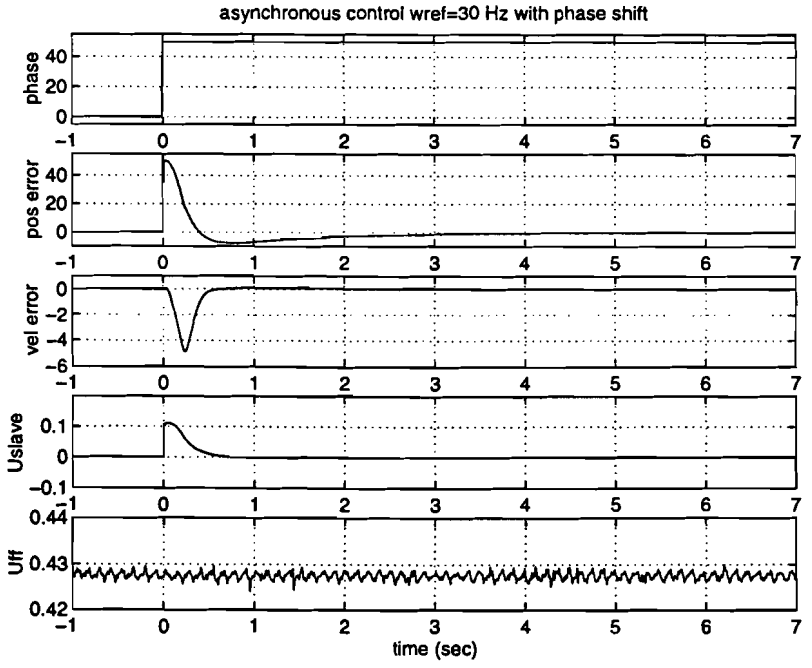

figure 6-31 : results of asynchronous control on positive phase shift with ω_{ref} = 30 Hz

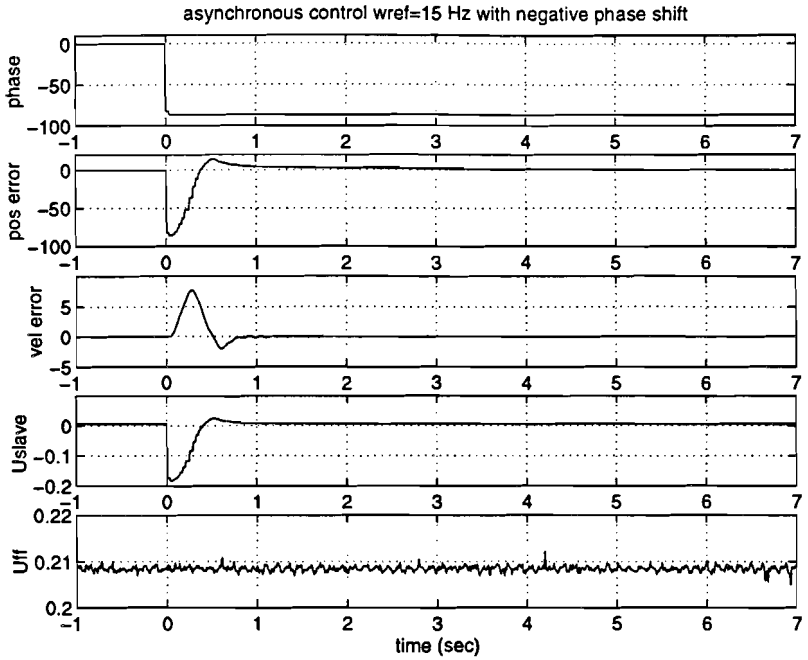

figure 6-32 : results of asynchronous control on negative phase shift with ω_{ref} = 15 Hz

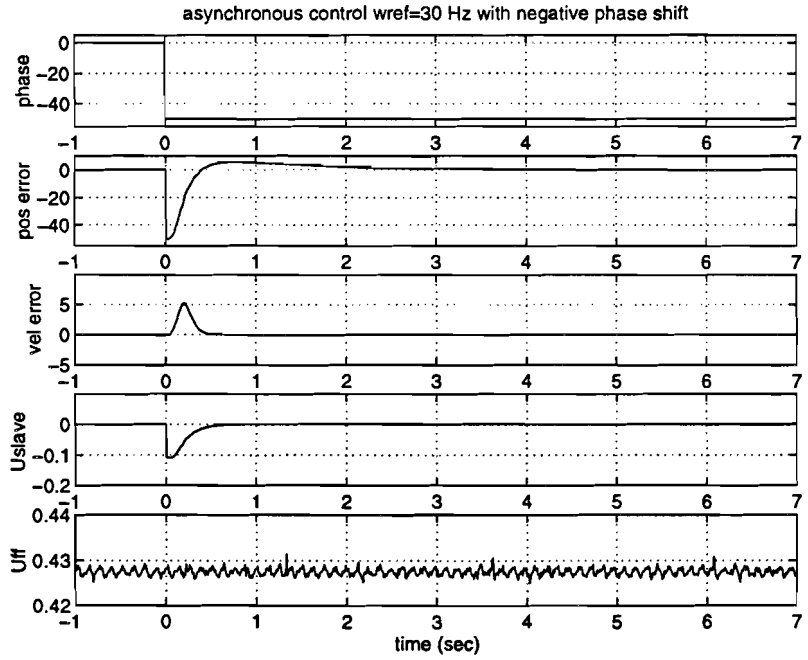

figure 6-33 : results of asynchronous control on negative phase shift with ω_{ref} = 30 Hz

The figures 6-30 to 6-33 show good responses the phase shifts for both velocities. This implies that phase errors can be adjusted while the motors are running. In these result the feedforward signal from the master frequency converter is shown more clearly. The 10 Hz distortion is also seen here. This might be corrected by using an anti aliasing filter for the AD converter that reads the feedforward. In the test environment there is no filter used for that.

From the results of the asynchronous slave controller is clear that the results are much better than the synchronous case with the low encoder resolution. The results of the asynchronous controller are even better than the result of the synchronous controller with full slave encoder resolution for faster velocities. This is probably due to the fact that a better tuning is found for the asynchronous controller.

6.10.5 Comments on the implementations

The current implementation of the controllers for the master-slave synchronization is done in an application running on one DSP. In this application two velocity observers are used to retrieve the motor velocity. This velocity is only used for reference now, although it could be used as input in a controller for the velocity of the master motor. These observers are executed at a sample frequency of 2 kHz occupying much of the execution of the DSP.

In a final solution the control can be split into a controller for the master motor and a controller for the slave motor. This master controller should control the velocity of the master motor and provide the position of the master load axis as a reference to the slave controllers.

The slave controllers need as inputs the position of the master axis and a feed forward of the control signal applied to the master motor. The required phase shift in the start of the product cycle times between the master and the slave could also be treated as an input. Most of the slave controller can be implemented in cheap hardware. A detection mechanism for the encoder signals with a counter to count up the encoder pulse. The pulse detector of the master load axis can be used as direct reset for this hardware counter. The control algorithm of the PI(D) controller can be implemented on a simple processor which must have an external interrupt.

6.11 Summary

For the master-slave synchronization three solutions are implemented and tested.

First a synchronous solution using the full resolution of the slave encoder. This solution is used as a reference for comparing the results of the other solutions. This solution works well, but the used PID controller is very difficult to tune (due to the D-action which was implemented incorrectly). The results with a manually tuned PI controller showed a steady state position error less than 2° on the load axis.

The second solution is also a synchronous solution, except that a slave encoder is used with only 16 pulses. As expected is the tuning of the PI controller more difficult. Although a controller setting is found which gives a steady state error of less than 2.5° on the load axis. But it show very clearly the effect of the large position steps of the slave encoder.

The third solution implemented and tested is an asynchronous controller for the slave motor. In this case a slave encoder is used with a resolution of 4 pulses per revolution of the motor. This solution works very well. The tuning of the used PI controller is very easy and there is hardly any adjustment needed when the velocity of the motor is increased.

The steady state position error of the synchronization is less than 1° and the error when the master velocity is changed is much less than in the synchronous case with the 16 pulses slave encoder. Even changes in the required phase between the two load axis are corrected fast.

These results make the asynchronous solution very suitable for Buhrs.

The combination of the dSPACE hardware and development tools and the developed software for the controllers is very suitable to test other combinations.

It should be noted that on changing the reference input of the master the position error between the master and slave increases drastically. This is due to the fact that in that case the control signal of the master controller saturates. This saturated control signal is applied to the slave motor as a feed forward leaving no room for the slave controller to adjust because the feed forward input to the slave is almost saturated. This problem can be reduced by using a stronger limitation on the master control signal, leaving room for the slave controller to adjust.

6.12 Literature

[1] Bosch van den P.P.J.; A.C. van der Klauw
Modeling, Identification and Simulation of Dynamical Systems
CRC Press, 1994
Boca Raton

[2] Aström K.J.; T. Hägglund
Automatic Tuning of PID Controllers
Instrument Society of America, 1988

[3] Aström K.J.; B. Wittenmark
Computer Controlled Systems
Prentice Hall, Englewood Cliffs, N.J., 1984

[4] Peng Y.; D. Vrancic; R. Hanus
Anti-Windup, Bumpless, and Conditioned Transfer Techniques for PID controllers
In: Control Systems
IEEE Control Systems Society
Volume 16, Number 4, August 1996

7 Conclusions and recommendations

The conclusions and recommendations must be split in two parts. The first part about the dSPACE development system and the second part about the motor synchronization.

The design process of the controller for the motor synchronization shows that the dSPACE system is a very useful development system. The tools supplied are very convenient when testing a designed controller. The three different ways to extend the normal functionality of the dSPACE system are easy to use. The hardware of the dSPACE system has enough power to be useful for many control applications.

Some recommendations for the dSPACE system are; A setup program and monitor program for the dSPACE I/O boards. It is currently not possible to monitor the configuration of the I/O boards, which can be useful when writing specific code. A second tool that can be useful is an extension of the simulation framework. This framework does not support interrupts. It could be extended with section that makes it easy for an engineer to add some interrupt code.

The master-slave synchronization shows very good results with an asynchronous controller, especially when a low resolution slave encoder is used. The velocity observer used in this application gives a useful observer to estimate the velocity from a position signal. The used PID controllers show far better control performances because of the anti windup method used.

In the master-slave synchronization control application are some points that need some more study. First of all the algorithm must be tested with an one pulse slave encoder resolution. Secondly the some other types of controllers must be tried to see if an even better performance is possible. Maybe an automatic tuning algorithm for the controllers could be added.

Appendix A Source code asynchronous controller

This appendix contains the full source code of **srtframe.c** for the asynchronous controller for the master-slave synchronization. It is include to give a full reference of this extended version of the standard **srtframe.c**. The basis for this version of the srtframe.c is the version supplied as an example with the **PHS bus interface library** from dSPACE gmbh.

Also included in this appendix is the file **async.usr**. This file contains the definitions to performs the data transfer between the framework **srtframe.c** and the Simulink model code. Transferring global variables of the framework to the Simulink model code makes the available for use with TRACE and COCKPIT.

Appendix A.1 Source code srtframe.c

```
/*****************************************************************************

   Real-time simulation routines for the dSPACE Real-time Interface to
   SIMULINK. This file holds the main program for simulation, the
   functions for the initialization and the timer interrupt service
   routine.

   Modified version to support PHS-bus interrupts.
   Timer 1 is started on DS1003 boards to allow execution time
   measurement even for pure interrupt-driven models.

   dSPACE 08/22/1996

   Copyright (c) 1992 - 1996 by dSPACE GmbH

   $RCSfile: srtframe.c $ $Revision: 1.17 $ $Date: 1996/04/23 13:45:05 $

   Extended with a asynchronous controller for the Master-Slave
   synchronization. The DS5001 interrupt routine contains the
   code for detecting the load axis pulse for the master and
   slave axis and the position counter for the slave axis.
   The position counter for the master axis is done by
   a hardware counter on the DS5001 board.
 *****************************************************************************/

/* DS5001 channel definitiions */
#define MASTER_ENC 1
#define SLAVE_ENC 3
#define MASTER_RESET 5
#define SLAVE_RESET 7
#define MASTER_ZERO 6
#define SLAVE_ZERO 8

#define MASTER_PULS 0          /* pulses detected before interupt */
#define SLAVE_PULS 256         /* pulses detected before interupt */
#define Nmaster 1024           /* pulse resolution of master encoder */
#define Nslave 4               /* pulse resolution of slave encoder */

#define pi 3.14159265

/*
** The following global variables are used
** service routine isr_external().
*/

long Master_Reset = 0;      /* booleans for reset positions */
long Slave_Reset = 0;
long Master_Zero = 0;
long Slave_Zero = 0;

long Counter_Master = 0;    /* motor position counters */
long Counter_Slave = 0;     /* currently hardware counters */

long Slave_Position = 0;    /* slave axis position counter*/
```

```
long Master_Position = 0;  /* master axis position counter (hardware) */
long Master_Revolve = 0;   /* counter master axis revolution */

float h = 0;               /* step size for PID controller */
float Ti = 0;              /* discrete Ti parameter */
float Td = 0;              /* discrete Td parameter */
float gh = 0;              /* discrete gamma parameter */
float K = 0;               /* K gain parameter */
float ti = 0;              /* continuous time ti */
float td = 0;              /* continuous time td */
float N = 0;               /* continuous time N */
long Tk = 0;               /* timestamp current slave pulse */
long Tkm = 0;              /* timestamp previous slave pulse */
float ek = 0;              /* postion error */
float ekm = 0;             /* postion error k-1 for I action */
float phase = 0;           /* phase difference load axis */
float ik = 0;              /* integral action */
float dk = 0;              /* derivative action */
float ukm = 0;             /* controller output k-1 */
float pk = 0;              /* proportional action */
float ykm = 0;             /* position error k-1 */
float urkm = 0;            /* real output k-1 */
float urk = 0;             /* real output k */
float yk = 0;              /* current position error master-slave */
float uk = 0;              /* slave controller output */
float uff = 0;             /* feedforward from master */
long NextPulse = 1;        /* alternating variable for detection samplerate */


/* standard definitions */

#define disable_isr()                                          \
   disable_isr_t1()

#define enable_isr()                                           \
   enable_isr_t1()

#define start_isr(stepsize)                                    \
   start_isr_t1(stepsize)

#define isr_main()                                             \
   isr_t1()



/*
** The following two defines specify the external interrupt source.
** You are free to modify these defines for your application.
*/
#define EXTINT_BOARD_BASE   DS5001_1_BASE        /* DS5001 board  */
#define EXTINT_SLAVE_INT    0                    /* IR0 interrupt */


#define __PHSINT_MIXED__


/* ------------------------------------------------------------------
   standard declarations
   ------------------------------------------------------------------ */

#define TRUE  1
#define FALSE 0

/* model variables */
double  *u, *y, *x, *dx, *xk, *P;

/* parameter array for external simulation */
double  *P_new;

/* variable to signal a reset of the model
   to the initial conditions; used for
   encoder indexing                      */
```

```
int set_init_conditions = FALSE;

/* time control variables */
double  *current_time, *final_time,
         exec_time = 0.0;

/* counter variable to measure
   the execution time          */
unsigned long _count;

/* variables associated with the sample times */
int numSampleTimes, rateIdentity = FALSE,
    **RateIDs, numRateIDs = 0,
    SampleTimeStartIndex  = 1;

/* variable to signal the task number
   for which an overload occurred     */
int error_no = 0;

/* ISR preempted flag */
int preempted = FALSE;

#define TSTART  0.0
#define TFINAL  1.0E+10

/* simulation control */
#define STOP    0
#define PAUSE   1
#define RUN     2

#ifndef SSTATE
#define SSTATE  RUN
#endif

/* set the simulation state */
volatile int sim_state = SSTATE;

/* ISR execution state */
volatile int exec_state = PAUSE;

#define MAX_SAMPLE_TIMES (16)
static int overrun_flag[MAX_SAMPLE_TIMES];

#include <srtframe.h>

/*--------------------------------------------------------------------------*/

#ifdef __PHSINT_ONLY__

/* definitions for complete PHS-bus interrupt driven model */
/* use timer interrupt handler for PHS-bus interrupts */

#undef disable_isr
#undef enable_isr
#undef start_isr
#undef isr_main

#include <phsint.h>                    /* interface to PHSbus interrupts */

#define disable_isr()                                          \
   disable_phs_int(EXTINT_BOARD_BASE, EXTINT_SLAVE_INT)

#define enable_isr()                                           \
   enable_phs_int(EXTINT_BOARD_BASE, EXTINT_SLAVE_INT)

#define start_isr(dummy)                                       \
{                                                              \
   int phs_ret;                                                \
                                                               \
   phs_ret = install_phs_int_vector(EXTINT_BOARD_BASE,         \
                EXTINT_SLAVE_INT, isr_external);               \
   if (phs_ret != PHSINT_NO_ERROR)                             \
```

```
   {                                                          \
     *_error = phs_ret;          /* signal error to host */  \
     exit(0);                                                 \
   }                                                          \
   time_start(1);                        \
   global_enable();                      \
}

#define isr_main()                                            \
   isr_external()

#endif


/*---------------------------------------------------------------------
  Definitions the PHSINT modes.
  --------------------------------------------------------------------- */

#ifdef __PHSINT_MIXED__

/* Definitions for mixed timer and PHS-bus interrupt driven model. */
/* Provide your own initialization code and PHS-bus interrupt handler below. */


/*** Register address offsets of the DS5001 board ***/

#define DS5001_FALLEN_OFFS  0x00
#define DS5001_RISEEN_OFFS  0x01
#define DS5001_COMP_OFFS    0x02
#define DS5001_INCR_OFFS    0x02
#define DS5001_GATE_OFFS    0x03
#define DS5001_EVENT_OFFS   0x04
#define DS5001_INTCH_OFFS   0x05
#define DS5001_WRDAC_OFFS   0x05
#define DS5001_ICU1_OFFS    0x06
#define DS5001_ICU2_OFFS    0x07
#define DS5001_RADR_OFFS    0x08
#define DS5001_MODE_OFFS    0x09
#define DS5001_INTSEL_OFFS  0x0A
#define DS5001_COUNTR_OFFS  0x0B
#define DS5001_POINTR_OFFS  0x0C
#define DS5001_LENGTH_OFFS  0x0D
#define DS5001_INTLEN_OFFS  0x0E
#define DS5001_IDENT_OFFS   0x0F
#define DS5001_RESET_OFFS   0x0F
#define TTL_LEVEL 1.45


/*** Initializations of the DS5001 registers ***/

void ds5001_phsint_init (long base)
{
    volatile long *fallen = (long *) (PHS_BUS_BASE + base + DS5001_FALLEN_OFFS);
    volatile long *riseen = (long *) (PHS_BUS_BASE + base + DS5001_RISEEN_OFFS);
    volatile long *incr   = (long *) (PHS_BUS_BASE + base + DS5001_INCR_OFFS);
    volatile long *gate   = (long *) (PHS_BUS_BASE + base + DS5001_GATE_OFFS);
    volatile long *intch  = (long *) (PHS_BUS_BASE + base + DS5001_INTCH_OFFS);
    volatile long *wrdac  = (long *) (PHS_BUS_BASE + base + DS5001_WRDAC_OFFS);
    volatile long *mode   = (long *) (PHS_BUS_BASE + base + DS5001_MODE_OFFS);
    volatile long *countr = (long *) (PHS_BUS_BASE + base + DS5001_COUNTR_OFFS);
    volatile long *pointr = (long *) (PHS_BUS_BASE + base + DS5001_POINTR_OFFS);
    volatile long *length = (long *) (PHS_BUS_BASE + base + DS5001_LENGTH_OFFS);
    volatile long *intlen = (long *) (PHS_BUS_BASE + base + DS5001_INTLEN_OFFS);
    volatile long *reset  = (long *) (PHS_BUS_BASE + base + DS5001_RESET_OFFS);

    int error;
    /*** add your initializations for the DS5001 here ***/
    ds5001_init(base);
    /* intialize master and slave encoder mode */
    error = ds5001_enc_init(base,MASTER_ENC,TTL_LEVEL,TTL_LEVEL);
    error = ds5001_enc_init(base,SLAVE_ENC,TTL_LEVEL,TTL_LEVEL);
```

```
    /* setup interrupt generation for encoders */
    *gate = *gate&~DS5001_GATESL;      /* disable all inputs for initialization
*/
    *intch;
    *mode = (*mode & ~DS5001_SELECT)| (MASTER_ENC-1);  /* select master encoder
*/
    *length=0;
    *intlen = MASTER_PULS;              /* interrupt 0 after MASTER_PULS events
*/
    *mode = (*mode & ~DS5001_SELECT)| (SLAVE_ENC-1);    /* select slave encoder
*/
    *length=0;
    *intlen = SLAVE_PULS;               /* interrupt 0 after SLAVE_PULS events
*/

    /* intialize master encoder reset interrupt */
    *mode=(*mode & ~DS5001_SELECT)| (MASTER_RESET-1);
    *length=0;
    *wrdac= volt_to_int(TTL_LEVEL);       /* TTL level */
    *intlen = 1;                          /* interrupt 0 after 1 events     */
    *riseen|=(1<<(MASTER_RESET-1));       /* enable rising edge detection   */
    *fallen&=~(1<<(MASTER_RESET-1));      /* disable falling edge detection */

    /* intialize slave encoder reset interrupt */
    *mode = (*mode & ~DS5001_SELECT)| (SLAVE_RESET-1);
    *length=0;
    *wrdac= volt_to_int(TTL_LEVEL);       /* TTL level                      */
    *intlen = 1;                          /* interrupt 0 after 1 events     */
    *riseen|=(1<<(SLAVE_RESET-1));        /* enable rising edge detection   */
    *fallen&=~(1<<(SLAVE_RESET-1));       /* disable falling edge detection */

    /* intialize load axis master pulse interrupt*/
    *mode = (*mode & ~DS5001_SELECT)|(MASTER_ZERO-1);
    *length=0;
    *wrdac= volt_to_int(TTL_LEVEL);
    *intlen = 1;                          /* interrupt 0 after 1 events     */
    *riseen|=(1<<(MASTER_ZERO-1));        /* enable rising edge detection   */
    *fallen&=~(1<<(MASTER_ZERO-1));       /* disable falling edge detection */

    /* intialize load axis slave pulse interrupt*/
    *mode = (*mode & ~DS5001_SELECT)| (SLAVE_ZERO-1);
    *length=0;
    *wrdac= volt_to_int(TTL_LEVEL);
    *intlen = 1;                           /* interrupt 0 after 1 events     */
    *riseen|=(1<<(SLAVE_ZERO-1));          /* enable rising edge detection   */
    *fallen&=~(1<<(SLAVE_ZERO-1));         /* disable falling edge detection */

    *reset = 0;                           /* clear input FIFOs */

    /* clear event counters by dummy read on high addr */
    *mode = (*mode & ~DS5001_SELECT)| 4;
    *countr;
    *mode = (*mode & ~DS5001_SELECT)| 5;
    *countr;
    *mode = (*mode & ~DS5001_SELECT)| 6;
    *countr;

    /* clear pending FIFOINT and DPRINT */
    *intch;

    /* connect hardware event counters to channels */
    *gate = (*gate&~DS5001_CSEL)|0x0020;
            /* counterA to channel 1  */
            /* counterB to channel 1  */
            /* counterC to channel 3  */

    /* enable all inputs, set all event */
    *gate = (*gate & ~DS5001_GATESL)|DS5001_ALWAYS;
}


/***  PHSbus interrupt handler ***/
```

```
void isr_external()
{
    long channel;
    /*** add your external interrupt service code here ***/

    /* DS5001 PHS bus addresses */
    volatile long *mode   = (long *) (EXTINT_BOARD_BASE + PHS_BUS_BASE +
DS5001_MODE_OFFS);
    volatile long *length = (long *) (EXTINT_BOARD_BASE + PHS_BUS_BASE +
DS5001_LENGTH_OFFS);
    volatile long *intch  = (long *) (EXTINT_BOARD_BASE + PHS_BUS_BASE +
DS5001_INTCH_OFFS);
    volatile long *countr = (long *) (EXTINT_BOARD_BASE + PHS_BUS_BASE +
DS5001_COUNTR_OFFS);
    volatile long *event = (long *) (EXTINT_BOARD_BASE + PHS_BUS_BASE +
DS5001_EVENT_OFFS);
    volatile long *radr = (long *) (EXTINT_BOARD_BASE + PHS_BUS_BASE +
DS5001_RADR_OFFS);
    volatile long *pointr = (long *) (EXTINT_BOARD_BASE + PHS_BUS_BASE +
DS5001_POINTR_OFFS);

    /* reset the LENGTH register of the channel that generated this interrupt  */
    /* this can be without checking because all channels must be polled anyway */
    *intch;

    /* reset master motor position */
    *mode = (*mode & ~DS5001_SELECT)| (MASTER_RESET-1);
    if(*length>=1)
    {
      if(*length>1) *_error=MASTER_RESET;
      *length=0;
      /* clear and read event counter A */
      *mode = (*mode & ~DS5001_SELECT)| 4;
      /* keep reset value for velocity observer */
      Master_Reset=*countr;
    }

    /* reset slave motor position */
    *mode = (*mode & ~DS5001_SELECT)| (SLAVE_RESET-1);
    if(*length>=1)
    {
      if(*length>1) *_error=SLAVE_RESET;
      *length=0;
      /* clear and read event counter C */
      *mode = (*mode & ~DS5001_SELECT)| 6;
      /* keep reset value for velocity observer */
      Slave_Reset=*countr;
    }

    /* reset master load axis position */
    *mode = (*mode & ~DS5001_SELECT)| (MASTER_ZERO-1);
    if(*length>=1)
    {
      if(*length>1) *_error=MASTER_ZERO;
      *length=0;
      /* set boolean for visualization with TRACE */
      Master_Zero=1;
      /* clear and read event counter B */
      *mode = (*mode & ~DS5001_SELECT)| 5;  /* clear event counters */
      *countr;
      /* count axis revolution */
      if (Master_Revolve<1) Master_Revolve++;
    }

    /* reset slave axis position */
    *mode = (*mode & ~DS5001_SELECT)| (SLAVE_ZERO-1);
    if(*length>=1)
    {
      if(*length>1) *_error=SLAVE_ZERO;
      *length=0;
      /* reset slave position counter */
```

```
      Slave_Position=0;
      /* count axis revolution compared to master axis */
      if (Master_Revolve>-1) Master_Revolve--;
      /* set boolean for visualization with TRACE */
      Slave_Zero=1;
    }

    /* interrupt for pulses slave encoder */
    /* triggers the control task */
    *mode = (*mode & ~DS5001_SELECT)| (SLAVE_ENC-1);
    if(*length>=SLAVE_PULS)
    {
      start_ds2001 (0x00000000);
      /* flip boolean for sample frequency visualization */
      NextPulse=-1*NextPulse;
      /* count slave position */
      Slave_Position++;
      *length=0;
      /* retrieve sample time */
      *radr=*pointr;
      Tk=*event;
      h=((Tk-Tkm)&DS5001_TIME)/DS5001_OSC_CLK;
      Tkm=Tk;
      /* possible calculation of PID parameters */
     /*  Ti=h/ti;
         Td=td/(h+td/N);
         gh=td/(N*h+td);  */
      /* read real slave control signal */
      urk = ds2001 (0x00000000, 0x00000002)/0.757;
      /* read master position and calculate master slave position error */
      yk = Slave_Position*SLAVE_PULS;
      *mode=(*mode&~DS5001_SELECT) | 1;   /* select counter B */
      yk=yk-(*countr+Master_Revolve*Nmaster*9);
      /* calculate position error */
      ek=phase+yk;                    /* calculate current error (in positions)*/
      ek=ek*(2*pi/(9*Nmaster));       /* convert error to radians */
      /* PID controller */
      dk=gh*dk+K*Td*(yk-ykm)*(2*pi/(9*Nmaster));    /* D action */
      pk=K*ek;                                      /* P action */
      /* conditional integration of I action */
      if(((ik > 0.5) && (ek>0)) || ((ik< -0.5) && (ek<0))) { ik=ik; }
      else {ik=ik+K*Ti*ekm;}
      uk=(pk-dk)+ik;
      ekm=ek;                          /* save old error for I action */
      if((uk<0) && (abs(uk)>uff)) {uk=-0.8*uff;};
      urkm=uk+uff;
    }
}

#undef disable_isr
#undef enable_isr
#undef start_isr


#include <phsint.h>                    /* interface to PHSbus interrupts */

#define disable_isr()                                          \
{                                                              \
   disable_isr_tl();                                           \
   disable_phs_int(EXTINT_BOARD_BASE, EXTINT_SLAVE_INT);       \
}

#define enable_isr()                                           \
{                                                              \
   enable_isr_tl();                                            \
   enable_phs_int(EXTINT_BOARD_BASE, EXTINT_SLAVE_INT);        \
}

#define start_isr(dummy)                                       \
{                                                              \
   int phs_ret;                                                \
   phs_ret = install_phs_int_vector(EXTINT_BOARD_BASE,         \
```

```
                EXTINT_SLAVE_INT, isr_external);          \
   if (phs_ret != PHSINT_NO_ERROR)                        \
   {                                                      \
     *_error = phs_ret;          /* signal error to host */ \
     exit(0);                                             \
   }                                                      \
   start_isr_t1(dummy);                                   \
/*   enable_isr();     */                                      \
}

#endif


/* ---------------------------------------------------------------------
   function for model evaluation
   --------------------------------------------------------------------- */

#include <rt_sim.h>

extern void rtUpdateContinuousStates (double *y, double *x, double *u,
                                      SimStruct *S, int id);


__INLINE void evaluate_model (SimStruct *S, int id)

{
   /* variable defines for user code */
   int i,sr,mr;
   volatile long *mode   = (long *) (EXTINT_BOARD_BASE + PHS_BUS_BASE +
DS5001_MODE_OFFS);
   volatile long *length = (long *) (EXTINT_BOARD_BASE + PHS_BUS_BASE +
DS5001_LENGTH_OFFS);
   volatile long *intch  = (long *) (EXTINT_BOARD_BASE + PHS_BUS_BASE +
DS5001_INTCH_OFFS);
   volatile long *countr = (long *) (EXTINT_BOARD_BASE + PHS_BUS_BASE +
DS5001_COUNTR_OFFS);

   read_started_input();              /* macro to read inputs          */
   usr_input();                       /* macro for user defined input  */

   ssCallmdlOutputs(y, x, u, S, id);  /* evaluate outputs */
   usr_output();                      /* macro for user defined output */
   output();                          /* macro to write outputs */

   ssCallmdlUpdate(x, u, S, id);      /* call discrete update function */

   if (ssIsContinuousTask(S, id)) {   /* only continuous task updates states */
#ifdef MULTITASKING
     for (i=0; i < numRateIDs; i++)  /* reset task IDs to deal with integration
*/
       *RateIDs[i] = 1;              /* methods with order > 1                */
#endif /* MULTITASKING */
     rtUpdateContinuousStates(y, x, u, S, id);
   }
}


/* ---------------------------------------------------------------------
   interrupt service routine for timer 1
   --------------------------------------------------------------------- */

int initConditions();

void isr_main()
{
   char event_flag[MAX_SAMPLE_TIMES];
   int  multiple_hits, i, j;

   /* get current timer count */
   _count = count_timer(1);

   /* simulation control */
```

```
  if (exec_state != RUN) {
    exec_time = time_elapsed(1, _count);
    service_trace();                          /* TRACE  service routine */
    service_mtrace("0");                      /* MTRACE service routine */
    return;
  }
  else if ( ssGetT(S) > ssGetTFinal(S) ) {
    sim_state = STOP;
    return;
  }

  start_input();                        /* macro to start input drivers   */
  read_input();                         /* macro to read all inputs which
                                           do not employ a start function */

  /* check whether the service routine has been preempted */
  preempted = FALSE;
  for (j=0; j< numSampleTimes; j++)
    preempted += overrun_flag[j];
  if(preempted)
    exec_time = ssGetStepSize(S);

  /* if requested reset the model to the initial conditions */
  if (set_init_conditions) {
    if (!ssIsMinorTimeStep(S)) {
      initConditions();
      set_init_conditions = FALSE;
    }
  }

  if (overrun_flag[0]++) {
    disable_isr();
    *_error   = OVERLOADED;     /* sampling too fast for the fastest rate */
    error_no  = 0;
    sim_state = STOP;
    return;
  }
  else
    global_enable();

  /* handle the sample hit event flags */
  multiple_hits = FALSE;                        /* reset multiple hits flag */
  if (numSampleTimes > 1) {
    rtUpdateDiscreteEvents(S);                          /* update the flags */
#ifdef MULTITASKING
    for (i = 1; i < numSampleTimes; i++) {              /* buffer the flags */
      event_flag[i] = ssIsSampleHitEvent(S, i, -1);
      if (event_flag[i]) {
        if (overrun_flag[i]++) {
          disable_isr();
          *_error   = OVERLOADED;                   /* Sampling too fast   */
          error_no  = i;                            /* for sample time "i" */
          sim_state = STOP;
          global_disable();
          return;
        }
        multiple_hits = TRUE;         /* multiple sample hits encountered */
      }
    }
    if (rateIdentity) {                  /* handle rate identity situation */
      for (i=0; i < numRateIDs; i++)
        *RateIDs[i] = 0;
      overrun_flag[1]--;
    }
#endif /* MULTITASKING */
  }

  /* run the fastest rate of the model and the I/O */
  evaluate_model(S, 0);
  service_trace();                          /* call TRACE  service routine */
  service_mtrace("0");                      /* call MTRACE service routine */
  ssIncrementT(S);
```

```
#ifdef MULTITASKING

   if ((numSampleTimes == 1) || (!multiple_hits)) {
     global_disable();
     overrun_flag[0]--;
   }
   else {
     overrun_flag[0]--;

     /* run the model for any other sample times */
     for (i = SampleTimeStartIndex; i < numSampleTimes; i++) {
       if (event_flag[i]) {
         rtStepSim(S, i);
         if (i==(numSampleTimes-1))
           global_disable();
         overrun_flag[i]--;
       }
     }
     global_disable();
   }

#else /* SINGLETASKING */

   global_disable();
   overrun_flag[0]--;

#endif /* MULTITASKING */

   /* execution time measurement */
   exec_time = time_elapsed(1, _count);
   if (intpt_flag_timer1())
     exec_time = ssGetStepSize(S);

}


/* ---------------------------------------------------------------------
   main function
   --------------------------------------------------------------------- */

void initialize();
int  initVariables();

void main()
{
   int last_sim_state = sim_state;

   /* initializations */

   *_error = UNINITIALIZED;
   initialize();
   if (*_error != UNINITIALIZED)
     exit;
   else
     *_error = NO_ERROR;

   /* get number of sample times */
   numSampleTimes = ssGetNumSampleTimes(S);

#ifdef __PHSINT_MIXED__
   /* (re-)initialize some registers of the DS5001 board */
   ds5001_phsint_init(EXTINT_BOARD_BASE);
#endif

   /* start interrupt service routine for timer1 */
   start_isr(ssGetStepSize(S));

   /* background task */

   for (;;)
```

```
  {

    if (*_error != NO_ERROR)
      sim_state = STOP;

    service_cockpit();
    UpdateParameters(S, &P_new, &P);

    switch (sim_state)
    {
      case RUN:
        if (last_sim_state == STOP)
        {  /* initialize and restart the simulation */
           initVariables();
           *_error = NO_ERROR;
           enable_isr();          /* to enable timer interrupts after
                                     an overload situation occurred   */
        }
        exec_state = RUN;
        last_sim_state = RUN;
        break;

      case STOP:
        exec_state = PAUSE;
        if (last_sim_state != STOP)
        {
          ssCallmdlTerminate(S);
          usr_terminate();        /* macro for user defined actions
                                     after the process has terminated */
        }
        last_sim_state = STOP;
        break;

      case PAUSE:
        exec_state = PAUSE;
        if (last_sim_state == STOP)
           sim_state = STOP;
        last_sim_state = PAUSE;
        break;

      default:
        exec_state = PAUSE;
        last_sim_state = sim_state;
        break;
    }


    background();                 /* macro for generated background task */
    usr_background();             /* macro for user defined background
                                     task */
  }
}


/* ---------------------------------------------------------------------
   initialization functions
   --------------------------------------------------------------------- */

/* initialization function for the initialize conditions */

int initConditions()
{
   int  i;
   SimStruct *s;
   double    Tcount0;

   /* set global time variables */
   Tcount0 = floor(TSTART/ssGetStepSize(S));
   S->Tcount=Tcount0;
   ssSetT(S, S->Tcount * ssGetStepSize(S));
   ssSetTFinal(S, TFINAL);
```

```
   /* set task-specific time variables */
   for (i = 0; i < ssGetNumSampleTimes(S); i++)
     ssSetPresentTimeEvent(S, i, S->Tcount * ssGetStepSize(S));

   /* set time variables for S-functions */
   for (i = 0; i < ssGetNumSFunctions(S); i++) {
     int sti;
     s = ssGetSFunction(S, i);
     Tcount0 = floor(ssGetT(S)/ssGetStepSize(s));
     s->Tcount=Tcount0;
     ssSetT(s, s->Tcount * ssGetStepSize(s));
     ssSetTFinal(s, TFINAL);
     for (sti = 0; sti < ssGetNumSampleTimes(s); sti++)
       ssSetPresentTimeEvent(s, sti, s->Tcount * ssGetStepSize(s));
   }

   /* set frame-specific time variables */
   current_time = &ssGetT(S);
   final_time   = &ssGetTFinal(S);

   /* set init conditions for the model */
   {
     int error_flag;
     error_flag = *_error;
     ssCallmdlInitializeConditions(ssGetX(S),S);
     if (*_error == INIT_FCNGEN_ERROR)
       return (int) INIT_FCNGEN_ERROR;
     else
       *_error = error_flag;
   }

   /* initialization of user defined code */
   usr_initialize();

   return (int) 0;
}


/* ------------------------------------------------------------------------ */

/* initialization function for the model variables */

int initVariables()
{
   /* local error variable */
   int error;
   int i;

   /* let the model variables point to the corresponding
      SIMULINK variables of S                              */
   u  = ssGetU(S);
   y  = ssGetY(S);
   x  = ssGetX(S);
   dx = ssGetdX(S);
   xk = &x[ssGetNumContStates(S)];

   /* set inputs to zero */
   for (i=0; i< (ssGetNumInputs(S)); i++)
     u[i] = 0.0;

   /* set outputs to zero */
   for (i=0; i< (ssGetNumOutputs(S)); i++)
     y[i] = 0.0;

   /* set block outputs to zero */
   for (i=0; i< (ssGetNumBlockIO(S)); i++)
     B[i] = 0.0;

   /* set init conditions for the simulation */
   error = initConditions();
   if (error)
     return error;
```

```
   /* initialize overrun flags of the scheduler */
   memset(overrun_flag, 0, sizeof(overrun_flag));

   return (int) 0;
}


/* ---------------------------------------------------------------------- */

/* initialization function for the I/O drivers */

int initIO ()
{
   /* local error variable used during the initialization  */
   int error = 0;

   /* macro for initialization of the I/O drivers          */
   init_io();

   /* initialization was successful if we get here, i.e.
      no error was detected in the code of init_io(), and
      thus no premature return occurred                    */
   return (int) 0;
}


/* ---------------------------------------------------------------------- */


#ifdef MULTITASKING

  /* function to detect rate identity in nested S-functions */
  void CheckSfcnRateIdentity(SimStruct *S)
  {
    int i;

    /* fastest task is continuous */
    if (ssGetSampleTimeEvent(S, 0) == 0.0) {
      if (ssGetNumSampleTimes(S) > 1) {
        if (ssGetSampleTimeTaskID(S,1) == 1) {
          /* get pointer to the task ID index 1 */
          RateIDs[numRateIDs++]=(ssGetSampleTimeTaskIDPtr(S))+1;
        }
      }
    }
    /* fastest task is time-discrete */
    else if (ssGetSampleTimeTaskID(S,0) == 1) {
      /* get pointer to the task ID index 0 */
      RateIDs[numRateIDs++]=ssGetSampleTimeTaskIDPtr(S);
    }

    /* recurse for nested S-functions */
    for (i = 0; i < ssGetNumSFunctions(S); i++) {
      CheckSfcnRateIdentity(ssGetSFunction(S,i));
    }
  }


  /* function to detect rate identity in the root model */
  int CheckRateIdentity (SimStruct *S)
  {
    int i;

    /* is the model continuous; if not, return */
    if (ssGetSampleTimeEvent(S, 0) != 0.0)
      return 0;

    /* is there more than one sample rate; if not, return */
    if (ssGetNumSampleTimes(S) <= 1)
      return 0;
```

```
    /* does a rate identity exist, else return */
    if (ssGetSampleTimeEvent(S, 1) == ssGetStepSize(S)) {
      rateIdentity = TRUE;
      SampleTimeStartIndex = 2;
    }
    else
      return 0;

    /* allocate memory for the book-keeping vector */
    RateIDs = (int **) malloc (128 * sizeof(int *));
    if (RateIDs == NULL) {
      return 1;
    }

    /* get pointer to the task ID index 1 */
    RateIDs[numRateIDs++]=(ssGetSampleTimeTaskIDPtr(S))+1;

    /* adjust the flag for task 1 in the PerTaskSampleHits array */
    (S)->Root->Events.PerTaskSampleHits[0]=1;

    /* look for rate identity in nested S-functions */
    for (i = 0; i < ssGetNumSFunctions(S); i++) {
      CheckSfcnRateIdentity(ssGetSFunction(S,i));
    }

    return 0;

  }
#endif /* MULTITASKING */


/* ------------------------------------------------------------------------ */


/* main initialization function */

void initialize ()
{

   /* local error variable */
   int error;

   init();  /* basic hardware initialization */

   /* create SIMULINK data structure */
   S = ssCreateSimStruct(NULL);
   if (S == NULL) {
     error_message(CREATE_SIMSTRUCT_ERROR);
     return;
   }
   MODEL(S);
   ssCallmdlInitializeSizes(S);
   error = ssInitSimStruct(S);
   if (error != 0) {
     error_message(INIT_SIMSTRUCT_ERROR);
     return;
   }

   /* initialize and check sample time events */
   if (ssGetNumSampleTimes(S) == 0)
     ssSetStepSize(S, (double) DT);
   else if (ssGetSampleTimeEvent(S, 0) == 0.0)
     ssSetStepSize(S, (double) DT);
   else
     ssSetStepSize(S, (double) ssGetSampleTimeEvent(S, 0));
   ssInitSampleTimeTaskIDs(S);
   if (!rtCheckSampleTimes(S)) {
     error_message(SAMPLE_TIMES_ERROR);
     return;
   }
```

```
#ifdef MULTITASKING
   /* workaround to let the countinuous task and the
      first discrete task execute at the same time
      if they own the same rate                          */
   error = CheckRateIdentity(S);
   if (error) {
     error_message(RATEID_ALLOC_ERROR);
     return;
   }
#endif /* MULTITASKING */

   /* initialize SIMULINK parameters */
   P = ssGetBlockParam(S);
   error = InitUpdateParameters(S, &P_new);
   if (error) {
     error_message(INIT_PARAM_ERROR);
     return;
   }

   /* initialize SIMULINK states and signal vectors */
   error = initVariables();
   if (error == INIT_FCNGEN_ERROR) {
     error_message(INIT_FCNGEN_ERROR);
     return;
   }

   /* initialization of the I/O drivers */
   error = initIO();
   if (error) {
     error_message(error);
     return;
   }

}
```

Appendix A.1 Source code async.usr

```
/*********************************************************************

   Include file async.usr:

   Definition of macros for user defined initialization, system I/O,
   and background process code

   Adjusted for data transfer between interrupt routine and Simulink model
   By P.E. Kamphuis
   Copyright (c) 1993-96 by dSPACE GmbH

 *********************************************************************/

#define usr_initialize()                          \


#define usr_input()                               \
    disable_isr();                                \
    *mode=(*mode & ~DS5001_SELECT)| 0;            \
    DSINPORT(pos_master,0,(float)*countr);        \
    *mode=(*mode & ~DS5001_SELECT)| 2;            \
    DSINPORT(pos_slave,0,(float)*countr);         \
    *mode=(*mode & ~DS5001_SELECT)| 1;            \
    DSINPORT(posmaster,0,(float)*countr);         \
    DSINPORT(res_master,0,(float)Master_Reset);   \
    enable_isr();                                 \
    DSINPORT(res_slave,0,(float)Slave_Reset);     \
    DSINPORT(zero_master,0,(float)Master_Zero);   \
    DSINPORT(zero_slave,0,(float)Slave_Zero);     \
    DSINPORT(posslave,0,(float)Slave_Position);   \
    DSINPORT(pos_error,0,(float)ek);              \
    DSINPORT(ia,0,(float)ik);                     \
    DSINPORT(da,0,(float)dk);                     \
```

```
    DSINPORT(pa,0,(float)pk);                    \
    DSINPORT(stuur,0,(float)uk);                 \
    DSINPORT(uterug,0,(float)urk);               \
    DSINPORT(interupt,0,(float)NextPulse);       \
    DSINPORT(samplet,0,(float)h);                \
    Master_Zero=0;                               \
    Slave_Zero=0;                                \


#define usr_output()                             \
   mr = (int) DSOUTPORT(res_master2,0);          \
   sr = (int) DSOUTPORT(res_slave2,0);           \
   if(mr==2) Master_Reset = 0;                   \
   if(sr==2) Slave_Reset = 0;                    \
   phase = DSOUTPORT(set_phase,0);               \
   /* transfer PID paramters */                  \
   uff = DSOUTPORT(set_ff,0);                    \
   Ti = DSOUTPORT(set_ti,0);                     \
   Td = DSOUTPORT(set_td,0);                     \
   N = DSOUTPORT(set_gh,0);                      \
   K = DSOUTPORT(set_K,0);                       \

#define usr_background()                         \


#define usr_terminate()                          \
```

Appendix B Source code velocity observer

This appendix contains the full source code of the S-function for the velocity observer as used for reference in the master-slave motor synchronization.

```
/*
 * w observer S-function source file.
 * discrete velocity observer
 *  Copyright (c) 1996 TU Eindhoven
 *  P.E.Kamphuis
 *  All Rights Reserved
 */

/*
 * The following #define is used to specify the name of your S-Function.
 * You should change the define to include the name of your S-function.
 */

#define S_FUNCTION_NAME wobsrv2

/*
 * Need to include simstruc.h for the definition of the SimStruct and
 * its associated macro definitions.
 */

#include "simstruc.h"

/* define input arguments */
#define WN ssGetArg(S,0)
#define TS ssGetArg(S,1)
#define Ns ssGetArg(S,2)

/*
 * mdlInitializeSizes - initialize the sizes array
 *
 * The sizes array is used by SIMULINK to determine the S-function block's
 * characteristics (number of inputs, outputs, states, etc.).
 */

static void mdlInitializeSizes(SimStruct *S)
{
  ssSetNumContStates(     S, 0);    /* number of continuous states */
  ssSetNumDiscStates(     S, 6);    /* number of discrete states */
  ssSetNumInputs(         S, 2);    /* number of inputs */
  ssSetNumOutputs(        S, 2);    /* number of outputs */
  ssSetDirectFeedThrough(S, 0);    /* direct feedthrough flag */
  ssSetNumSampleTimes(    S, 1);    /* number of sample times */
  ssSetNumInputArgs(      S, 3);    /* number of input arguments */
  ssSetNumRWork(          S, 11);   /* number of real work vector elements */
  ssSetNumIWork(          S, 0);    /* number of integer work vector elements */
  ssSetNumPWork(          S, 0);    /* number of pointer work vector elements */
}

/*
 * mdlInitializeSampleTimes - initialize the sample times array
 *
 * This function is used to specify the sample time(s) for your S-function.
 * If your S-function is continuous, you must specify a sample time of 0.0.
 * Sample times must be registered in ascending order.  If your S-function
 * is to acquire the sample time of the block that is driving it, you must
 * specify the sample time to be INHERITED_SAMPLE_TIME.
 */

static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTimeEvent(S, 0, mxGetPr(TS)[0]);
    ssSetOffsetTimeEvent(S, 0, 0.0);

    /*
```

```
     * SET OTHER SAMPLE TIMES AND OFFSETS HERE
    */
}

/*
 * mdlInitializeConditions - initialize the states
 *
 * In this function, you should initialize the continuous and discrete
 * states for your S-function block.  The initial states are placed
 * in the x0 variable.  You can also perform any other initialization
 * activities that your S-function may require.
 */

static void mdlInitializeConditions(double *x0, SimStruct *S)
{
    int i,nStates;
    double Wn,T;
    double *r = ssGetRWork(S);

    Wn=mxGetPr(WN)[0];
    T=mxGetPr(TS)[0];

    for(i=0;i < ssGetNumRWork(S);i++) r[i]=0.0;
    r[5]=T;               /* T */
    r[6]=T/2;             /* T/2 */
    r[7]=3.5*Wn;          /* Kp=3.5*Wn */
    r[8]=4*Wn*Wn;         /* Ki=4*Wn*Wn */
    r[9]=Wn*Wn*Wn*r[5];   /* K3*T/2=2*Wn*Wn*Wn*T/2 */
    r[10]=mxGetPr(Ns)[0]; /* aantal pulsen per omwenteling */

    nStates=ssGetNumDiscStates(S);
    for(i=0;i<nStates;i++) *x0++=0.0;
}

/*
 * mdlOutputs - compute the outputs
 *
 * In this function, you compute the outputs of your S-function
 * block.  The outputs are placed in the y variable.
 */

static void mdlOutputs(double *y, double *x, double *u,
                       SimStruct *S, int tid)
{
    double *r = ssGetRWork(S);
    int k;
    double temp;
    temp=x[0];
    y[1]=0;               /* reset modulo boolean */
    if(u[1]!=0)
    {
        temp=temp-r[10];
        y[1]=2;
    }
    r[0]=u[0]-temp;                   /* e[k]=th[k]-thh[k] */
    r[1]=x[2]+r[9]*(r[0]+x[1]);       /* f[k]=f[k-1]+k3T/2*(e[k]+e[k-1]) */
    r[2]=r[1]+r[8]*r[0];              /* v[k]=f[k]+ki*e[k] */
    r[3]=x[4]+r[6]*(r[2]+x[3]);       /* w[k]=w[k-1]+T/2*(v[k]+v[k-1]) */
    r[4]=temp+r[5]*(r[7]*r[0]+r[3]);  /* thh[k+1]=thh[k]+T*(Kp*e[k]+w[k]) */
    y[0]=r[3];
}

/*
 * mdlUpdate - perform action at major integration time step
 *
 * This function is called once for every major integration time step.
 * Discrete states are typically updated here, but this function is useful
 * for performing any tasks that should only take place once per integration
 * step.
 */

static void mdlUpdate(double *x, double *u, SimStruct *S, int tid)
```

```
{
  double *r = ssGetRWork(S);

  x[0]=r[4];
  x[1]=r[0];
  x[2]=r[1];
  x[3]=r[2];
  x[4]=r[3];
}

/*
 * mdlDerivatives - compute the derivatives
 *
 * In this function, you compute the S-function block's derivatives.
 * The derivatives are placed in the dx variable.
 */

static void mdlDerivatives(double *dx, double *x, double *u, SimStruct *S, int
tid)
{
}

/*
 * mdlTerminate - called when the simulation is terminated.
 *
 * In this function, you should perform any actions that are necessary
 * at the termination of a simulation.  For example, if memory was allocated
 * in mdlInitializeConditions, this is the place to free it.
 */

static void mdlTerminate(SimStruct *S)
{

}

#ifdef MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```