Eindhoven University of Technology

MASTER

Design and realization of a real-time environment on a DSP-PC system

Mathijssen, S.G.A.

Award date:
1995

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

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

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

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

DESIGN AND REALIZATION OF A REAL-TIME ENVIRONMENT ON A DSP-PC SYSTEM

By S.G.A. Mathijssen

Master of Science Thesis
carried out from March 1995 to October 1995
commissioned by prof.dr.ir. P.P.J. van den Bosch
under supervision of ir. R.J.A. Gorter and ir. J.W.J.J. Beckers

The Department of Electrical Engineering of the Eindhoven University of Technology accepts no responsibility for the contents of M.Sc.Theses or reports on practical training periods.

Abstract

The Measurement and Control Group at the Eindhoven University of Technology received a DSP-PC system from the Delft University of Technology in February 1995. Digital Signal Processors (DSP's) have fast parallel processing capabilities and are very suitable for real-time purposes. Hence the Measurement and Control Group wants to measure and control processes in real time by using the DSP-PC system.

Therefore, the design objective is to develop and realize a real-time environment on the system in order to easily measure and control processes in real time. More specifically, an environment must be designed and realized on the DSP-PC system so that the user can implement his algorithm on the system without knowing the hardware configuration. This means that the user is not interested in how data acquisition is done or how data is logged to the hard disk.

In this report the design and realization of the real-time environment on the DSP-PC system is desribed. A small literature study is done on aspects and problems involving real-time systems. The Ward & Mellor method, a structured development method for real-time systems, is used for software design. The software design is implemented on the system by using the C language. A simple PID controller is implemented to show the usage of the real-time environment.

From literature study it is clear that a distinction must be made between foreground tasks and background tasks. Foreground tasks must be executed within a short time of the occurence of the interrupt, while background tasks should run when no interrupts are being processed. Therefore, background tasks have been implemented on the PC and foreground tasks have been implemented on the DSP.

Because tasks with hard time constraints have been implemented on the DSP, the environment is a hard real-time environment. However, tasks with soft time constraints (except hard disk logging) have been implemented on the PC and in that case the environment is a soft real-time environment.

The Ward & Mellor method appears to be suitable for software design of real-time systems. The transformation schemes that are produced by the method have been implemented on the DSP and the PC. In this way, a real-time environment has been implemented which is capable of logging data to the hard disk, displaying data and signals on screen, setting parameters

and enabling a debug mode.

A PID algorithm is implemented on the DSP to show the usage of the real-time environment. If no data is read from the input channels or written to the output channels this algorithm can be performed in 10.6 μsec. If data is read from the input channels and written to the output channels the performing of the algorithm takes 12.5 μsec (i.e. real time).

Contents

1 Introduction **6**

2 Aspects of real-time systems **8**

2.1 Definition of a real-time system 8
2.2 Classification of real-time systems 9
2.2.1 Clock-based systems 12
2.2.2 Event-based systems 12
2.2.3 Interactive systems 13
2.3 The role of time in real-time systems 13
2.3.1 Timeliness 13
2.3.2 Simultaneousness 14
2.3.3 Predictability 15
2.3.4 Dependability 15
2.4 Foreground/background modules 15
2.5 Synchronization in real-time systems 19
2.5.1 Scheduling foreground/background tasks 19
2.5.2 Problems concerning synchronization 20
2.6 Summary 23

3 Description of the DSP-PC system **24**

3.1 Description of the TMS320C30 PC Processor Board 25
3.1.1 General overview of the system 25
3.1.2 The TMS320C30 PC Processor Board 26
3.2 Description of the 16 bit Stereo Interface Boards 30
3.2.1 Outline description 30
3.2.2 Interface to the DSP 31
3.2.3 Conversion timing 31
3.3 The Digital Signal Processor TMS320C30 32
3.3.1 General description of Digital Signal Processors 32
3.3.2 General description of the TMS320C30 33
3.4 Description of the PC system 36
3.5 General purpose processor versus DSP 37
3.6 Summary 38

4 The Ward & Mellor method **39**

4.1 Description of the Ward & Mellor method 39

4.1.1 The static view 40
4.1.2 The event view 41
4.2 Summary 44

5 Software design of the real-time environment 45
5.1 System's requirements 45
5.2 General description of the system 46
5.3 Detailed description of the system 47
5.4 Summary 56

6 Software implementation of the real-time environment 57
6.1 Implementation of the communication PC <-> DSP 57
6.1.1 Data transfer from DSP to PC 57
6.1.2 Data transfer from PC to DSP 61
6.2 Implementation of the Ward & Mellor transformations 62
6.3 Implementation of function `main()` (file *pcmain.c*) 63
6.4 Implementation of the debug mode 65
6.5 System's performance 67
6.6 Summary 70

7 Example: a simple PID controller 71
7.1 Description of the PID controller 71
7.2 Implementation of the basic PID algorithm 73
7.3 Implementation of the PID controller 73
7.3.1 Plain implementation 73
7.3.2 Implementation of the PID controller using HD logging 78
7.3.3 Implementation of the PID controller using the debug mode 80
7.4 Summary 87

8 Conclusions and recommendations 88
8.1 Conclusions 88
8.2 Recommendations 89

A Hardware configuration of the DSP-PC system 93
A.1 Description of the TMS320C30 PC Processor Board 93
A.1.1 Memory 94
A.1.2 User prototyping area 96
A.1.3 Parallel expansion (DSPLINK) 96
A.1.4 Data transfers from the DSP to the PC 98
A.1.5 Data transfers from the PC to the DSP 98
A.2 Description of the 16 bit Stereo Interface Boards 99
A.2.1 Outline description 99
A.2.2 Interface to the DSP 100
A.2.3 Conversion timing 101
A.3 The Digital Signal Processor TMS320C30 103
A.3.1 General description 103
A.3.2 Memory organization 104

A.3.3 Interrupts 106
A.3.4 Timers 110
A.4 Status of the present hardware 111

B Directory structure and files 120
B.1 Directory structure of disk C 120
B.2 Some important batch files 124
B.3 Getting started 124

Chapter 1

Introduction

Nowadays, Digital Signal Processors (DSP's) are very popular and they can be found in many application areas. With their special architecture and instruction set, these devices can execute millions of instructions per second. Therefore, DSP's are very suitable for extremely fast processing of large volumes of data.

Real-time systems are closely related to DSP's. Real-time systems are systems that must produce correct responses within a definite time limit. Should computer responses exceed these time bounds then performance degradation and/or malfunction results. The need for and the advantages of real-time systems are more often recognized by engineers. Both DSP's and real-time systems are used in applications where speed of execution is very important. Hence they form the perfect combination.

The Measurement and Control Group of the Department of Electrical Engineering at the Eindhoven University of Technology has become aware of the advantages of real-time systems and DSP's. In February 1995, the Measurement and Control Group received a DSP-PC system from the Delft University of Technology. There, this system was used for the implementation of real-time state and parameter estimation of an asynchronous machine. The aim of the Measurement and Control Group is to use the system for measuring and controlling signals in real time.

The user is often not interested in the hardware configuration of a computer control system. He just wants to know how to implement his algorithm on the system without worrying about the technical details. Hence the general design objective is formulated as:

Design and realize a real-time environment on the DSP-PC system in order to measure and control signals in real time. The environment must be as flexible as possible, which means that every arbitrary algorithm should be implemented on the system without knowing the technical details.

The real-time environment must be capable of logging data to the hard disk, displaying data

and signals on the screen and setting parameters. Furthermore, a simple debug facility must be implemented to walk through the algorithm step by step, i.e. each time a key is pressed the algorithm must be performed once. The environment must be implemented using the C language.

The design and implementation of the real-time environment is presented in this report. Some knowledge of real-time systems is necessary to successfully perform the objective. Therefore, this report begins with an outline of the results of a small literature study on aspects and problems involving real-time systems. In chapter 3 the DSP-PC hardware configuration is described in broad outlines. The Ward & Mellor method is used for software design. This method is a structured development method for real-time systems and is explained in chapter 4. In chapter 5 is described how the Ward & Mellor method is applied on the real-time environment. Chapter 6 covers the whole implementation of the real-time environment. Here, the Ward & Mellor transformations are implemented. Additionally, the results of eight measurements are given to get an impression of the system's performance. The C language is used for the implementation.

Because the user needs to know how to implement his algorithm on the system a simple PID controller is implemented to illustrate the usage of the real-time environment. This is described in chapter 7. Finally, in chapter 8 some conclusions and recommendations for further work are given.

Chapter 2

Aspects of real-time systems

Some knowledge of both hardware and software design and construction of real-time control systems is needed to design a real-time environment on the DSP-PC system. Therefore, this chapter describes the results of a small literature study on real-time systems. Most techniques and problems listed in this chapter concern large real-time systems and they are only mentioned to get an impression of what real-time systems are about. When some kind of technique for the implementation of the real-time environment is used, this is explicitly indicated.

2.1 Definition of a real-time system

Several definitions of real-time systems are found in literature. Some definitions are hard to understand, while others are easy to interpret. However, concerning the content all definitions are identical.

Most computer control systems are those which must response within a specified time, because otherwise loss of control and possibly loss of many lives will be dealt with (e.g. aircraft engine control systems). For this kind of systems [Coo 91] offers the following clear definition:

> "Real-time systems are those which must produce correct responses within a definite time limit. Should computer responses exceed these time bounds then performance degradation and/or malfunction results."

2.2 Classification of real-time systems

A generalized picture of a computer control system is shown in figure 2.1 [Ben 94]. The input devices plus the input software gather the information needed to create an *input image* of the plant. The input image is a snapshot of the status of the plant and this snapshot is renewed at specified intervals. Where the external information is in analog form, the process of obtaining the snapshot will involve digitization (A/D conversion) as well as sampling.

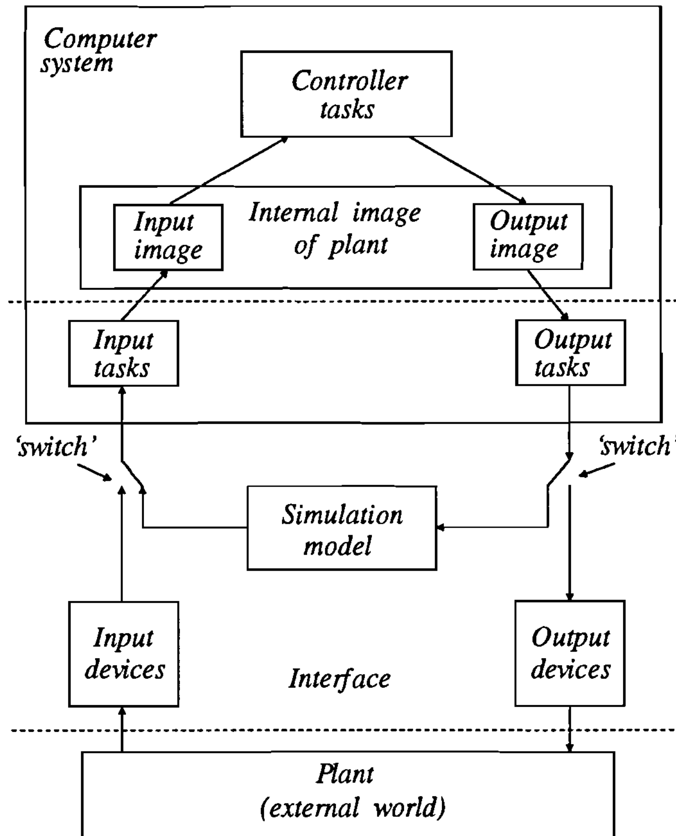


Figure 2.1: *Generalized computer control system.*

The *output image* represents the current set of outputs generated by the control calculations. The output image will be updated periodically by the control tasks. It is the job of the output tasks to transfer data contained in the output image to the plant (D/A conversion). Thus the control software can be considered as operating on an input image (or model) of the plant to produce the output image.

The blocks between the dashed lines form the *interface* [KKZ 88]. It consists of all points of influence in either direction and bounds the scope of responsibility of the system. What is and what is not part of the interface depends on relationships that have to be maintained by the system. The interface is all the environment "sees" of the system and vice versa. The characteristics of the interface depend strongly on the environment. The elements of the interface consist of state variables or of events. On the one hand there are events generated

by the environment and state variables manipulated by the environment, while on the other hand there are events produced by the system and state variables set by the system.

Furthermore, in a control system a mechanism for debugging must be present. Thus there must be a kind of switch for switching between the external world and the simulation model. This is depicted in figure 2.1.

The simulation model has two modes of debugging:

- code debugging
- "cold practising"

Through code debugging the source code of programs is debugged, while through "cold practising" the model is simulated. However, debugging of real-time programs is a difficult problem, because the computer and the real world to which it is relating are operating asynchronously [AT 90]. This means that it is often not possible to duplicate the situations that are causing problems. This is in distinct contrast to debugging a computational program. In a computational program all the action takes place in the computer, which is a strictly sequential machine and therefore will always duplicate its action exactly. Before entering the real-time phase of operation the computational part of the program has to be correct. The computational part can be checked by operating the program with another program that simulates the real-time part of the system.

The heart of the computer control system is formed by the *Controller tasks* block. This block must take care of all tasks that are performed in the system. The objective of the project is to design a real-time environment on a DSP-PC system in order to provide the user to implement his algorithm on the system. The user is not interested in how data acquisition is done or how and where variables are stored in memory. Hence the objective of the project is satisfied by designing the *Controller tasks* block.

As can be seen in figure 2.1, software tasks are divided into three major areas:

- plant input tasks
- plant output tasks
- control tasks

This means that the communication with the operator is treated as part of the plant input and plant output tasks. However, in many applications communication will extend beyond simple indicators and switches. Plant engineers, plant managers and machine operators, for example, will require detailed information on all aspects of the operation of the plant. Therefore, the model depicted in figure 2.1 must be enlarged. This leads to model 2.2. The tasks (i.e. plant input, control, plant output and communication) may be carried out sequentially, concurrently (more than one processor perform various operations at the same time) or pseudo-concurrently (a single processor performs full parallel operation). The plant

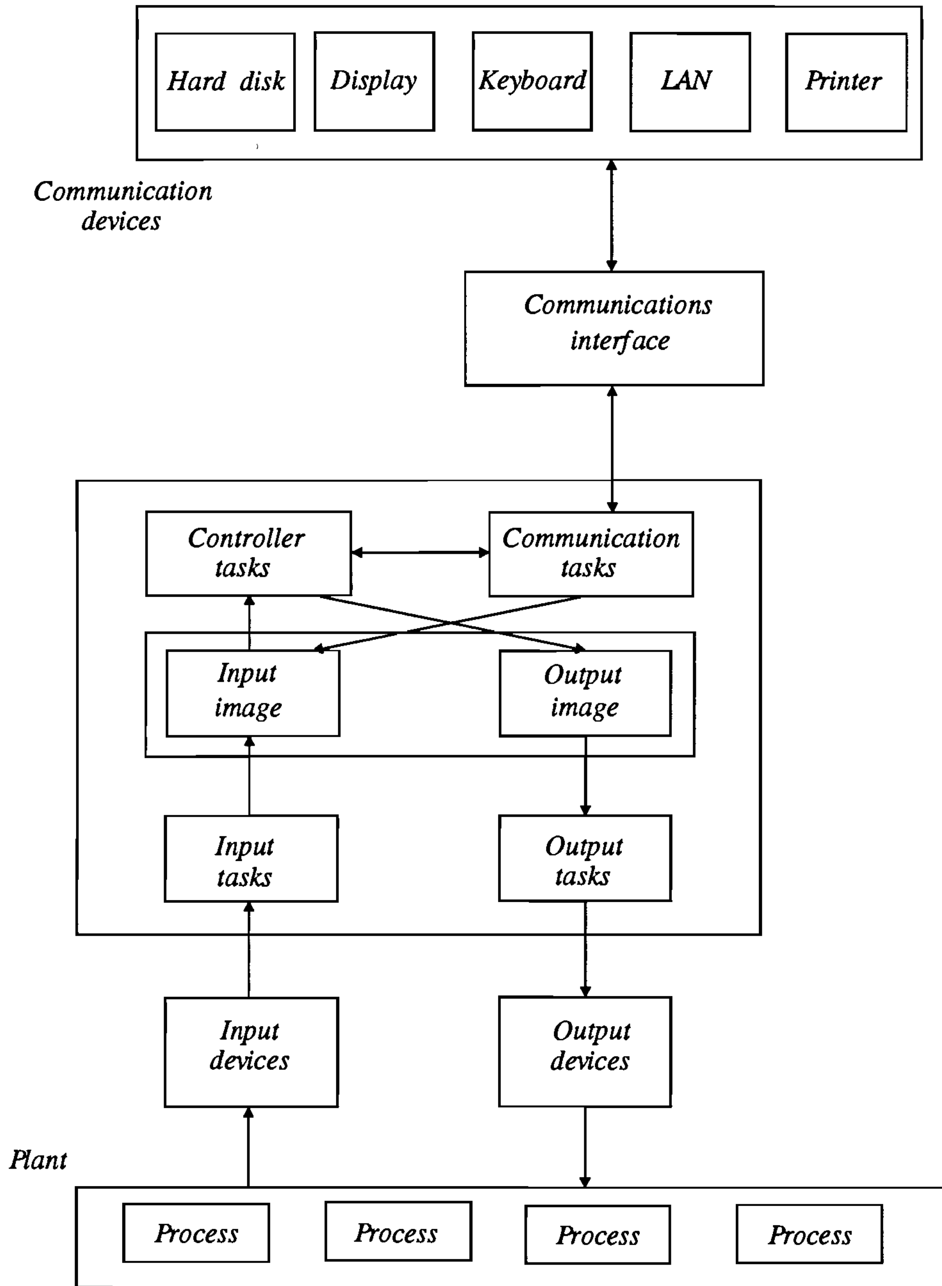


Figure 2.2: *Computer control system showing communication tasks.*

input, plant output and communication tasks shown in figure 2.2 have one feature in common: they are connected by physical devices to processes which are external to the computer. These external processes all operate in their own time-scales and the computer is said to operate in real time if actions carried out in the computer relate to the time-scales of the external processes. When the external processes and the internal actions are synchronized, a real-time system can be divided into three categories: clock-based systems, event-based systems and interactive systems.

2.2.1 Clock-based systems

A process plant operates in real time and has plant time constants. For feedback control the required sampling rate will be dependent on the time constant of the process to be controlled. The shorter the time constant of the process, the faster the required sampling rate. Hence the computer, which is used to control the plant, must be synchronized to real time. Furthermore, the computer must be able to carry out all the required operations (measurement, control and actuation) within each sampling interval.

The completion of the operations within the specified time is dependent on the number of operations to be performed and the speed of the computer. Synchronization is usually obtained by adding a clock (a "real-time" clock) to the computer. A signal from this clock can be used to interrupt the operations of the computer at some predetermined fixed time intervals.

Clock-based tasks are typically referred to as *cyclic* or *periodic* tasks where the terms can imply either that the task is to run once per time period T or is to run at exactly T unit intervals. A different name for clock-based is *state-processing* [KKZ 88]. State variables form one of the two elements of the interface and they can be continuous or discrete. Essential is, that a state variable is persistently present. Throughout its existence a state variable has a value, representing the state, which changes with the progression of time. Some variables get their values from processes operating in the environment, while other variables are set by the system.

The discrete perception of a state variable leads to a series of samples. In order to give a faithful picture of the environment, the samples must be taken sufficiently often, otherwise aliasing may occur. The DSP-PC system, for example, is a clock-based system.

2.2.2 Event-based systems

There are many systems where actions have to be performed not at particular times or time intervals, but in response to some event like the closure of a switch. Such systems are the so-called *event-based* systems. They normally use interrupts to inform the computer that action is required. Some small, simple systems may use polling. Polling means that the computer periodically asks (polls) the various sensors to see if action is required.

A different name for event-based is *event-processing.* Events form one of the two elements of

the interface. If the difference between sequentially ordered events is insufficient, the system may see one event instead of two or may not recognize the actual ordering of events in time. This leads to non-determinacy with respect to events that occur close to one another. Therefore, event-based tasks are frequently referred to as *aperiodic* tasks.

2.2.3 Interactive systems

Interactive systems probably represents the largest class of real-time systems. The real time requirement is usually expressed in terms of the average response time not exceeding a specified value. Although this type of system seems similar to the event-based system, it is different in that it responds at a time determined by the internal state of the computer. All interactive programs are driven by external events: the distinction which has to be made is that in a real-time program the external event, although limited to a certain class, cannot be predicted from the current state of the program.

2.3 The role of time in real-time systems

To clarify the definition according to [Coo 91] (see section 2.1), real-time systems can be characterized by the following requirements [HH 91]:

- timeliness
- simultaneousness
- predictability
- dependability

Figure 2.3 gives an overview of the requirements of a real-time system. The following sections describe the various requirements.

2.3.1 Timeliness

Two requirements on the timing of system behaviour specific to real-time systems can be distinguished:

- **Relative time.** If an action has to occur within a given interval of time relative to the occurence of an event, this is specified by giving a maximum duration between action and event.
- **Absolute time.** If an action has to occur at fixed points in time, the time intervals at which some action has to take place are specified with accuracy relative to a chosen distance between two points in time (the width of a time interval).

In connection with the timeliness requirement real-time systems are divided into two categories: *hard* and *soft real-time systems.* In a soft real-time environment there are only soft deadlines and if a deadline is exceeded the result can still be used, but its value for the system as a whole is less (e.g. position information in a navigation system). So soft real-time systems are systems for which an occasional failure to meet a deadline does not cover the correctness of the system.

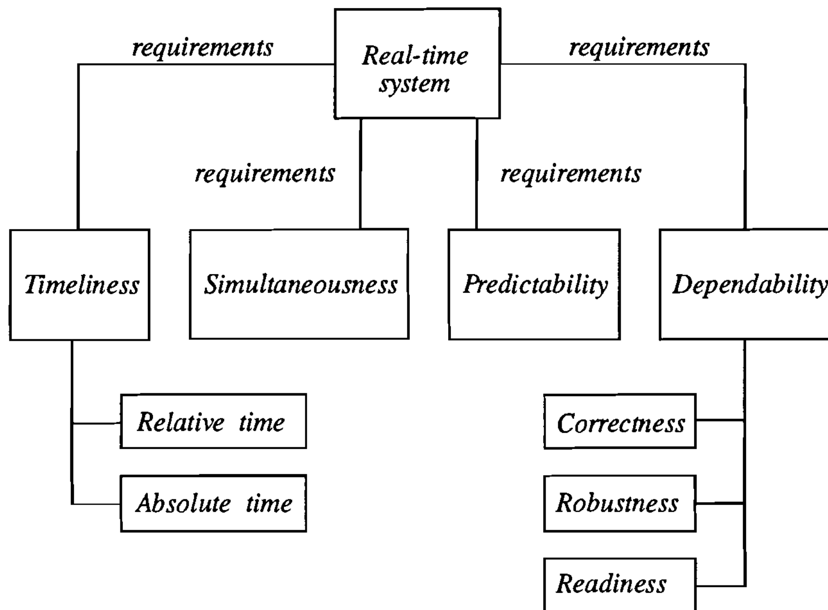


Figure 2.3: *Requirements of real-time systems.*

In a hard real-time environment violation of a deadline may yield useless results in the ideal situation, but it can result in disruption of a control process or even a catastrophe at worst. Thus hard real-time systems are systems that must satisfy the deadlines on each and every occasion.

Tasks that are running on the DSP all have hard time constraints and therefore the real-time environment is a hard real-time environment. However, tasks that are running on the PC (except the functions for hard disk logging) all have soft time constraints and in that case the real-time environment is a soft real-time environment.

2.3.2 Simultaneousness

A system may receive inputs from two or more independent sources and has to respond to each of them within given time limits. Because durations between different event-response pairs may overlap in time, simultaneous processing must be applied to satisfy this requirement. This can be achieved by true parallelism (more processors) or by pseudo-parallelism (multiprogramming).

2.3.3 Predictability

The actual behaviour of a system has to be within limits that can be derived from the specification. The output of the system must be fully *predictable* with respect to the input from the environment. Non-deterministic choice of resources is undesirable, as it may lead to unpredictable waiting times and resource allocations. Also performance enhancing features like virtual memory or caches introduce some degree of unpredictability, so they must be avoided.

2.3.4 Dependability

Dependability can be detailed by several more specific requirements:

- **Correctness.** It is the requirement that the functional behaviour of a system satisfies the specification. For real-time systems there is a strong relationship between correctness and performance: it may be as incorrect for a real-time program to produce the wrong result as to produce the right result at the wrong time.
  Performance studies are concerned with issues such as the time for execution of a program in a particular implementation, the effect on the execution time of the availability of computing resources, etc. [JG 89]. Performance studies take into account implementation details such as the number of processors in the machine and the structure and bandwidth of its communication network. Program verification systems, on the other hand, usually abstract away from implementation considerations and concentrate on program structure.
- **Robustness.** It is the requirement that the system remains in a predictable state even if the environment does not correspond to the specification (e.g. inputs are not within a given range) or when a part of the system (a peripheral device) fails.
- **Readiness.** Non-terminating processes provide for an on-going interaction with the environment. Termination of these processes, for instance as a result of failure, cannot be tolerated. Thus readiness is a requirement for real-time systems.

2.4 Foreground/background modules

There are obvious advantages (less module interaction, less tight time constraints) if the modules with hard time constraints can be separated from and handled independently of the modules with soft time constraints or no time constraints [Ben 94]. The modules with hard time constraints, i.e. those which are the most closely coupled to the external processes, are performed in the so called *foreground* and the modules with soft constraints (or no constraints) are performed in the *background*. The foreground modules (tasks) have a higher priority than the background tasks and some mechanism must be provided to enable a foreground task to interrupt a background task. The foreground task should be written as an interrupt routine and the background task as a standard program.

The flowcharts of a foreground/background approach are given in figure 2.4 and figure 2.5 respectively.

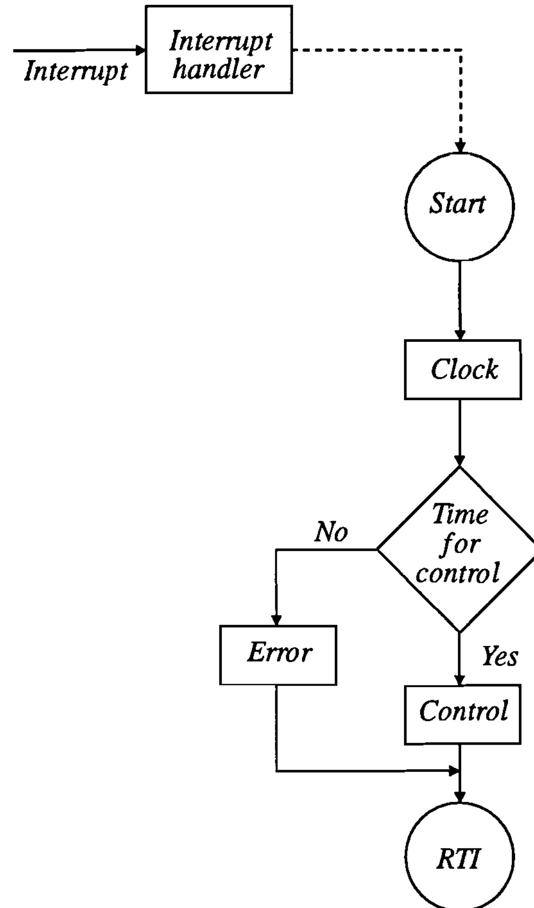


Figure 2.4: *Foreground approach.*

Foreground tasks are executed in direct response to hardware interrupts [AT 90]. They are typically very short so that they will not interfere with other foreground tasks. Most foreground tasks are time critical, i.e. they must be executed within a short time of the occurence of the interrupt to avoid loss of data or other malfunctions. Thus each time a foreground task must be performed, it should be checked if there is time left for control. If there is not, an error must be produced. Remark: "RTI" stands for **R**e**T**urn from **I**nterrupt. However, in the case of the DSP-PC system no check is done if there is time left to perform the algorithm, because it is assumed that the DSP is fast enough to perform the algorithm. In the most critical cases, a foreground task with all other interrupts disabled should be run to assure that it will be completed in time. Foreground tasks are responsible for managing the input and output data streams, doing whatever buffering and processing is necessary. As such, they represent a set of parallel processes. Each one of them can be implemented in a separate processor or each one of them can be implemented in the same processor and use the interrupt hardware

for scheduling their operation.

Background tasks run when no interrupts are being processed. When an interrupt occurs, whatever background task is in progress, is suspended while the interrupt is processed and is resumed when the foreground task finishes.

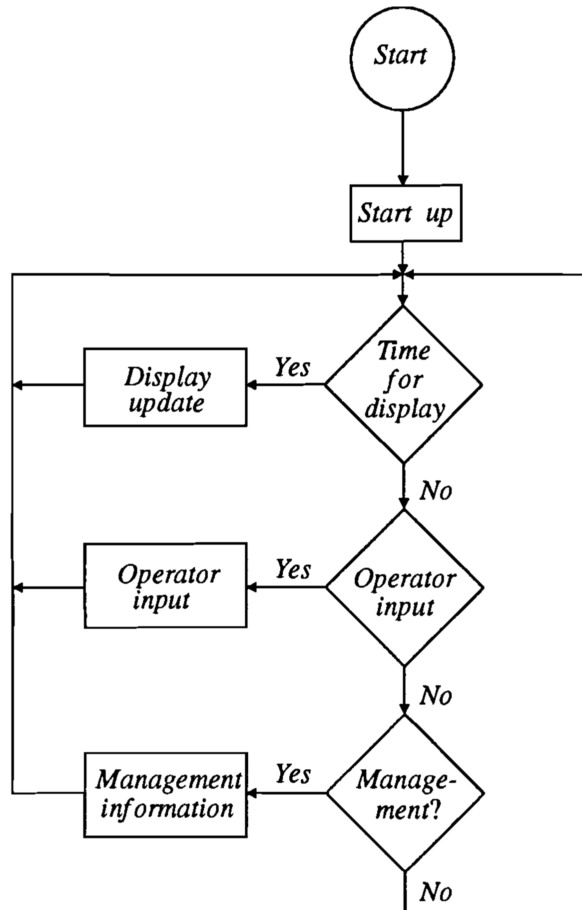


Figure 2.5: *Background approach.*

Other than the loss of time, the background task is not affected by the operation of the foreground. Communication between the foreground and background takes place through data buffers and flag variables to signal conditions that initiate or terminate activities. The *Display update* module in figure 2.5 can be divided into three modules: display messages, display selected settings and display process values. The *Operator input* module consists of setting and selecting parameters for analog output, setting of the sample frequency and setting of offset and scaling of the input signals. Finally, the *Management information* provides information on parameters that are typical for some kind of process, e.g. the average demanded temperature of an hot-air blower. These modules and there condition flags are used for the implementation of the real-time environment and are described more detailed later in

this report. However, in the real-time environment it is not checked if there is time left for display.

Although the foreground/background approach separates the control structure of the foreground and background modules, the modules are still linked through the data structure as shown in figure 2.6.

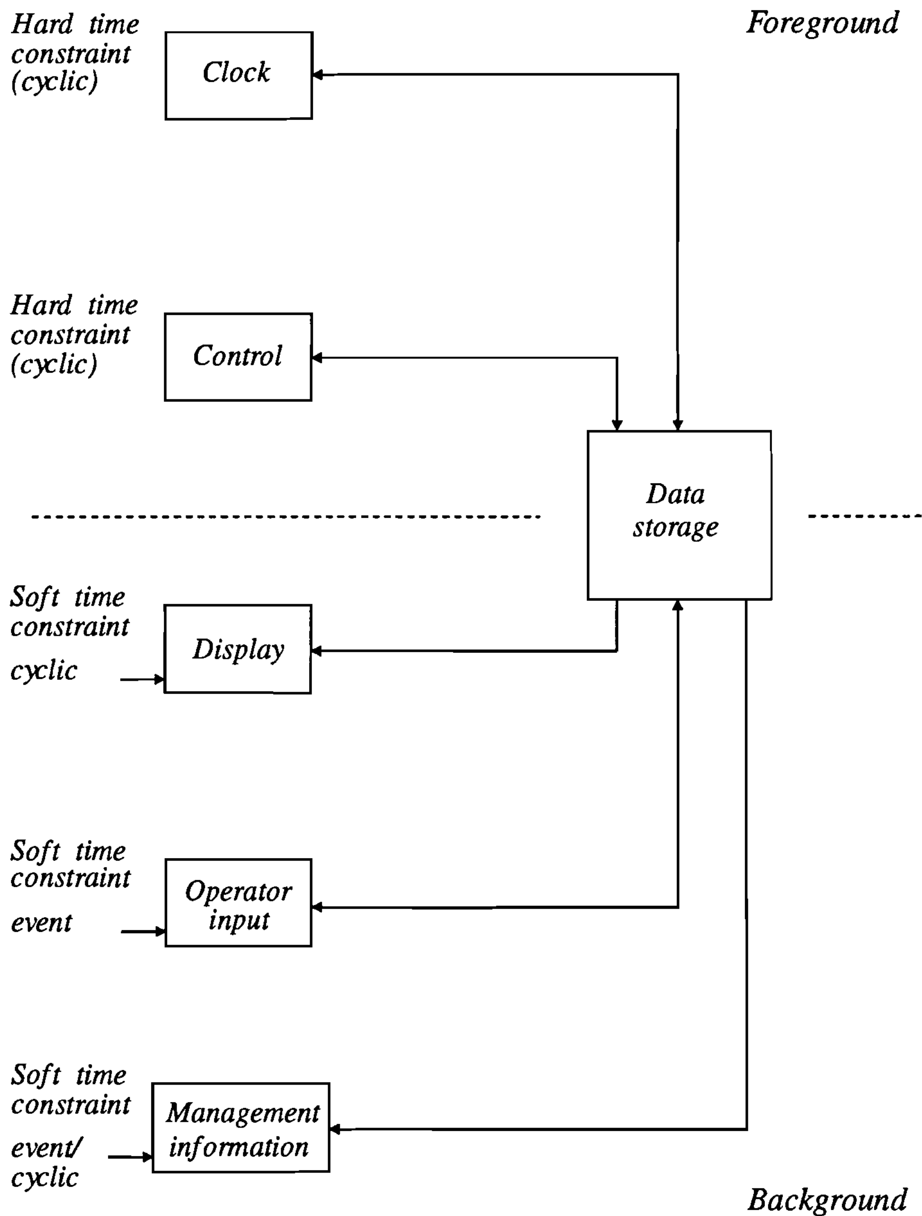


Figure 2.6: *Software modules for a foreground/background system.*

The linkage occurs because they share data variables. In a single program there is no difficulty in controlling access to the shared variables since only one module (task) is active at any one time, whereas in the foreground/background system tasks may be used in parallel, i.e. one foreground module and one background module may be active at the same time. The variables between the control, display and operator input modules can be shared without any difficulty, since only one module writes to any given variable. The operator input module writes the controller parameters and setpoint variables, the clock module writes to the date and time variables and the control module writes to the plant data variables. There is, however, one precaution which must be taken: the input from the operator must be buffered and only transferred to the shared storage when it has been verified.

To understand the reasons for buffering, consider what would happen if, when a new value is entered, it is stored directly in the shared data areas. Suppose a controller is operating with controller parameters $P_1 = 10$, $P_2 = 5$ and $P_3 = 6$ and it is decided that the new values of the control parameters should be $P_1 = 20$, $P_2 = 3$ and $P_3 = 0.5$. As soon as the new value of P_1 is entered the controller begins to operate with $P_1 = 20$, $P_2 = 5$ and $P_3 = 6$, i.e. neither with the old nor with the new values.

2.5 Synchronization in real-time systems

A concurrent real-time program usually has a number of cyclic (periodic) and sporadic (demand based) processes (see sections 2.2.1 and 2.2.2): the cyclic processes need to be executed at regular intervals and the sporadic processes on demand, but with some minimum time between successive demands [JG 89]. Without much loss of generality, all the processes can be considered to be cyclic if the sporadic processes are treated as requiring repeated execution with the specified minimum interval between executions. Let a program have n such processes, each process P_i making a request for C_i units of processor time every T_i time units. In a hard real-time system, a valid schedule must guarantee that, for some implementation, each process receives its processing requirement C_i within the interval T_i. An implementation in which this is achieved is called *feasible*. The repetition time T_i of process P_i is determined by the frequency with which some external process sends its events, while the computation time C_i depends on the time taken to complete the processing requirement for the process, i.e. on the speed of the processor.

If a real-time program consists of processes that interact closely with each other, decisions about their scheduling will be constrained by the needs of synchronization, for example, because processes that send messages must be executed before processes that receive the messages. If a process can undertake one of several actions and the choice between them is non-deterministic, scheduling the processes is considerably more difficult.

2.5.1 Scheduling foreground/background tasks

The parallelism that is obtained from foreground/background tasking results in the need for a separation of concerns [FP 88]. The need to share computing and data resources and

perform simultaneously a variety of loosely related tasks while meeting real-time deadlines often results in code in which conceptually different functions are interweaved. Therefore, the various tasks needs to be synchronized. A mechanism that can be used for synchronization between the foreground tasks and the background tasks is the *scheduler.*

The scheduler is the bridge between the foreground and background world. Its purpose is to control the operation of the background, but it can run as part of either the background or the foreground. It runs in response to any event that requires that a decision is made about which background tasks should be running.

When a function or task is running in the background, it can be interrupted by an external interrupt which will cause a foreground task to run. When the foreground task finishes its work the background function will continue from where it left off. However, once that background function begins, no other background function can run until it is done. If the scheduler is part of the background then it cannot run until the current background task is done. Each background task is running as a function called by the scheduler, so when it is done it returns to the scheduler. At that time the scheduler checks its task queue to see if any other tasks are ready to run. If any are, it will run the first one on the queue after setting the timer. If no tasks are ready, it can stay in a test loop, checking the time, until a task is ready to run.

The primary advantage to background scheduling is that it is easy to program and requires minimal interaction with the computer hardware. There are, however, several limitations that must be taken into account if it is to be used successfully. These mostly come from the property that once a background task is started, no other background task can be run until it finishes. The first consequence of this is that all of the background tasks must have the same priority. A task with a high priority would imply that, when its run time came up, it would be able to interrupt the execution of any lower priority task that were currently running. With background scheduling, if a low-priority task were to start just before the time for a high-priority task, the high-priority task would have to wait for the low-priority task to finish before it could start.

2.5.2 Problems concerning synchronization

In most real-time programs, processes are not independent and schedules must take account of interprocess synchronization and communication and the need for resource allocation. What is needed here is *mutual exclusion* between processes.

A multi-tasking operating system is able to share resources between several concurrently active tasks. However, this does not imply the resources can be used simultaneously. The use of some resources is restricted to only one task at a time. This restriction must be made for resources such as input and output devices, otherwise there is a danger that input intended for one task could get corrupted by input for another task. Once a task has access to a device driver, all other tasks are excluded until the task that has access relinquishes it. This is known as *mutual exclusion.*

The need for mutual exclusion can be satisfied by using several methods:

- **Semaphores.** The binary semaphore is a condition flag which records whether or not a resource is available. If, for a binary semaphore s , s = 1, then the resource is available and the task may proceed. If, however, s = 0, then the resource is unavailable and the task must wait. To avoid the processor wasting time while a task is waiting for a resource to become available, a queue is associated with each semaphore. This queue is used for suspending the running of a task when the processor is waiting and for recording that the task is waiting for a particular semaphore.
- **Monitors.** A monitor is a set of procedures that provide access to data or to a device. The procedures are combined inside a module that has the special property that only one task at a time can be actively executing a monitor procedure. It can be thought of as providing some kind of "fence" around critical data. The operations that can be performed on the data are moved inside the fence as well as the data itself. The user task thus communicates with the monitor rather than directly with the resource.

Another problem which can arise when using multi-tasking is known as *deadlock* [Lis 79]. Suppose task A has acquired exclusive use of resource X and now requests resource Y, but between A acquiring X and requesting Y, task B has obtained exclusive use of Y and has requested use of X. Neither task can proceed since A is holding X and waiting for Y and B is holding Y and waiting for X. The system is said to be deadlocked. Figure 2.7 shows an example of a *state graph* in which the nodes are resources and an arc between nodes A and B implies that there exists a process which holds resource A and is requesting resource B. In figure 2.7 resources A, B and D are involved in deadlock. These resources cause the deadlock situation and as a result resource C is also in deadlock.

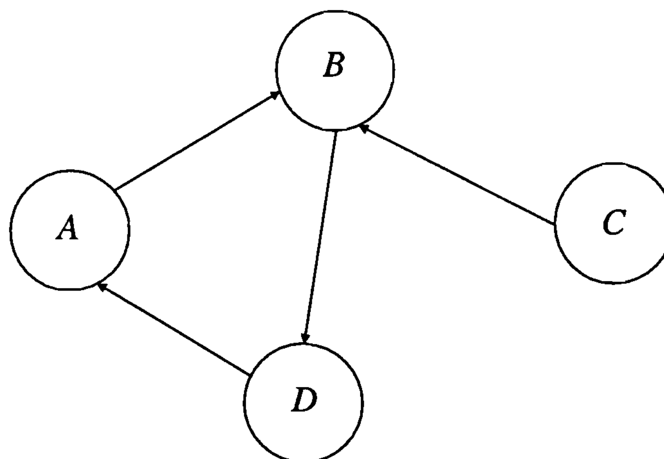


Figure 2.7: *Example state graph.*

In general the necessary and sufficient conditions for deadlock are as follows:

1. The resources involved are unshareable.
2. Processes hold the resources they have already been allocated while waiting for new ones.
3. Resources cannot be pre-empted while being used.
4. A circular chain of processes exists such that each process holds resources which are currently being requested by the next process in the chain.

The problem of deadlock can be solved by adopting one of the following strategies:

1. Prevent deadlock by ensuring at all times that at least one of the four conditions above does not hold.
2. Detect deadlock when it occurs and then try to recover.
3. Avoid deadlock by suitable anticipatory action.

A third problem which can arise while synchronizating processes is known as *code sharing.* Suppose two tasks use the sub-routine S. If task A is using the sub-routine, but before it finishes some event occurs which causes a re-scheduling of the tasks and task B runs and uses the sub-routine, then when a return is made to task A, although it will begin to use sub-routine S again at the correct place, the values of locally held data will have been changed and will reflect the information processed within the sub-routine by task B. Task A and B are said to be *code sharing.*

Two methods can be used to overcome the problem of code sharing:

- **Serially re-usable code.** The sub-routine is written in such a way that the value of any local variable on entry to the routine has no effect on the actions of the routine. Some form of lock mechanism is placed at the beginning of the routine such that if any task is already using the routine, the calling task will not be allowed entry until the task which is using the routine unlocks it.
- **Re-entrant code.** If the sub-routine can be coded such that it does not hold within it any data, i.e. it is purely code (any intermediate results are stored in the calling task or in a stack associated with the task) then the sub-routine is said to be *re-entrant.* Re-entrant routines can be shared between several tasks since they contain no data relevant to a particular task and hence can be stopped and restarted at a different point in the routine without any loss of information.

The DSP-PC system is not a multi-tasking operating system. Only the interrupt service routine (data acquisition, algorithm) is running as a foreground task. This means that resources like the hard disk and the input and output devices need not to be shared by several concurrently active tasks. Hence situations like deadlock and code sharing are not found in the implementation of the real-time environment.

2.6 Summary

In this chapter the results of a small literature study on real-time systems are given. A real-time system is defined as a system that must produce correct responses within a definite time limit, because otherwise performance degradation or malfunction results.

Three classes of real-time systems are distinguished. Clock-based system use the internal clock of the computer for synchronization and each task is to run once per time period T or is to run at exactly T unit intervals. The DSP-PC system is an example of such a system. In event-based systems actions have to be performed not at particular times or time intervals, but in response to some event. Interactive systems seem similar to event-based systems, but they are different in that they respond at a time determined by the internal state of the computer.

Real-time systems are characterized by four requirements namely timeliness, simultaneousness, predictability and dependability. In connection with the timeliness requirement real-time systems are divided into two catogories. Soft real-time systems are systems for which an occasional failure to meet a deadline does not cover the correctness of the system. Hard real-time systems are systems that must satisfy the deadlines on each and every occasion. Tasks that run on the DSP have hard time constraints and therefore the real-time environment is a hard real-time environment. However, tasks that run on the PC (except the functions for hard disk logging) have soft time constraints and in that case the real-time environment is a soft real-time environment.

Tasks with hard time constraints run in the so-called foreground, while tasks with soft time constraints run in the so-called background. Background tasks run when no interrupts are being processed, while foreground tasks must be executed within a short time of the occurence of the interrupt to avoid loss of data or other malfunctions. Schedulers are used to control the operation of the background. They form the bridges between the foreground and background modules and they can be found in large multi-tasking operating systems as well as in microprocessors.

Resources like the hard disk and the input and output devices need not to be shared by several concurrently active tasks. Therefore, in the design and realization of the real-time environment no attention is paid to mutual exclusion, deadlock and code sharing.

Chapter 3

Description of the DSP-PC system

In the past all manipulation of signals, like filtering of unwanted components from an input signal and extraction of information from a signal, was performed using analogue circuits and techniques. All analogue designs are dependent on sensitive components, i.e. resistor and capacitor values are never completely accurate, but their values change as a function of time, voltage and temperature. Furthermore, analogue designs using passive components have become more complicated. The solution to these analogue problems is digital signal processing (DSP).

DSP's are very fast microprocessors/microcomputers specially designed to execute the calculation of DSP algorithms. With their advanced architecture, parallel processing capabilities and dedicated DSP instruction set, these devices can execute millions of instructions per second (MIPS). One of the most often used instructions in DSP applications is multiply and accumulate (MAC). Through this MAC capability the implementation of complex DSP algorithms on a small silicon chip is possible, for which in the past a minicomputer or an array processor was necessary (an array processor is a parallel processor with a matrix-like structured organization for extremely fast processing of large volumes of data).

In the last ten years DSP's have become more popular. Now they can be found in many application areas. The most well known application areas are:

- **General purpose DSP**: digital filters, convolution, fast Fourier transforms, correlation, adaptive filtering.
- **Voice/speech**: speech recognition, speech validation, speech synthesis.
- **Graphics/imaging**: robot vision, pattern recognition, 3-D rotation, image enhancement.

- **Control and regulation**: servo control, robot control, motor control.

- **Telecommunications**: echo cancelling, data and speech encryption, FAX, digital speech interpolation.

The DSP-PC system uses the TMS320C30 DSP chip from Texas Instruments. This chip is implemented on a board that is manufactured by Loughborough Sound Images (LSI). The following sections describe the most important features of the board. A more detailed description can be found in appendix A.

3.1 Description of the TMS320C30 PC Processor Board

3.1.1 General overview of the system

The DSP chip is implemented on a board that is manufactured by LSI [PB 91]. The board uses a special bus to connect I/O modules to. This bus is called DSPLINK (or expansion bus or secondary bus; the three names indicates one and the same bus) and is described in section 3.1.2. The system uses three I/O cards with two D/A-converters and two A/D-converters each. A general overview of the complete system is depicted in figure 3.1. Unless

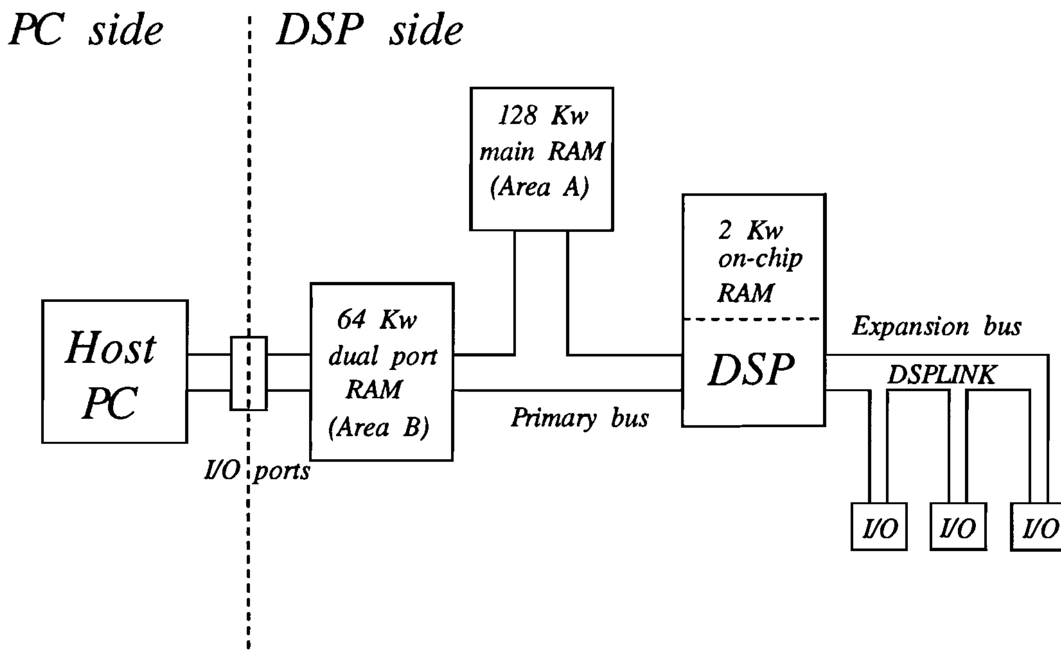


Figure 3.1: *Overview of the DSP–PC system.*

otherwise specified a memory word means a 32 bit word.

3.1.2 The TMS320C30 PC Processor Board

The TMS320C30 board consists of a digital signal processor TMS320C30 from Texas Instruments, DSPLINK (parallel expansion), memory expansion, prototyping area, two serial ports and several links (jumpers). The DSPLINK is connected to three 16 bit Stereo Interface Boards from LSI. These boards are described in section 3.2.

The board is suitable only for the PC AT and compatibles, as it uses the full 16 bit data interface. Access to memory passes through the dual porting hardware on the TMS320C30 board (see figure 3.1). The TMS320C30 dual port interface includes an address counter for block transfers. Furthermore, it includes hardware to transfer data between the 16 bit AT bus and the 32 bit DSP bus. Interrupts from the PC to the TMS320C30 and vice-versa are supported.

The board is designed to operate without the need to interrupt the PC. It is supplied with no interrupt facility enabled. However, in some circumstances one might need to generate an interrupt to the PC. This can be done using one of the on-board links. One can generate one of six interrupts (IRQ 3, 4, 5, 6, 7 or 9). Between the DSP and the PC there is a physical interrupt line. The various parts of the board are described in the next part of this section.

Memory

On the board there are 24 memory chips installed. Each chip is a CMOS 64K x 4 device and is static RAM. The chip is configured to operate in microprocessor mode, placing the interrupt locations in external memory and disabling the internal RAM at these locations. Two internal RAM memory blocks (1K word long) operate at the full processor speed with zero wait-states. Two memory areas are provided off-chip to supplement the 2K word RAM provided on-chip:

- **Area A** is divided into two areas. One 64K bank is configured with zero wait-state devices. External memory uses a hardware wait-state generator (on the board, but external to the TMS320C30 chip) to accomodate the different access times required by the various memory areas. Each area is 64K words long (see the block *"128 Kw main RAM"* in figure 3.1). Wait-states are set by on-board links to suit the speed of memory device used. The area is accessed from the PC by cycle stealing from the DSP if they both try to access this area simultaneously.

- **Area B** is used for transfer of data between the PC and the DSP (see the block *"64 Kw dual port RAM"* in figure 3.1). Both 64K words of one wait-state memory and 16K x 4 devices can be fitted. Wait-states and memory size are set by on-board links. Area B is true dual ported memory in that it may be accessed without halting the chip. Therefore, many of the library functions include a parameter which identifies the type of memory to be accessed. Although the PC can read and write other memory areas on the board, it incurs more overhead because the DSP chip must be held during PC accesses.

A memory expansion connector is available to provide the user to access the unused external

memory space on the primary TMS320C30 bus. Only daughter boards, such as 32 bit I/O boards, can be connected.

Parallel expansion (DSPLINK)

A parallel expansion system is provided as a memory-mapped peripheral area. The DSPLINK interface supports 16 bit parallel transfers at a maximum rate of 5 Million 16 bit words/sec, including software overhead. A more typical "Peak Software Transfer Rate" is 3.33 Million words/sec. The actual data transfer rate over the DSPLINK depends on two things:

1. The rate at which the master can read or write data.
2. The rate at which the slave can be read from or written to.

In general, most standard DSPLINK master boards automatically insert one processor wait-state between accesses. In the present configuration the DSPLINK master board inserts no wait-state. Increasing throughput by eliminating DSPLINK wait-states does not come without a penalty. One must be extremely careful in his design to meet very stringent timing requirements. The actual throughput that can be get depends on where the data is going after the DSP reads it in from the DSPLINK or where the data comes from before the DSP puts it out to the DSPLINK. Transfers over the DSPLINK use two wait-states to achieve a 180 nsec transfer cycle. 8K secondary bus memory locations (in the memory of the TMS320C30 board) are available and multiple slave boards can be accomodated.

There are 16 I/O locations. Each I/O location occupies a 32 bit word in the TMS320C30's memory space (memory-mapped I/O), but only the most significant 16 bits are used (the DSPLINK is only 16 data bits wide). When reading or writing from or to the I/O cards, a conversion must be done. Figure 3.2 shows what happens if 16 bits are read from the I/O cards. The convention is that the 16 most significant bits are read. Thus when reading 16 bits

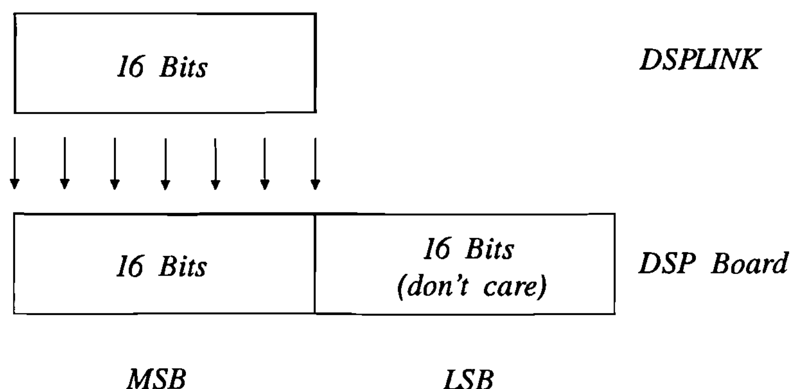


Figure 3.2: *Reading 16 bits from the I/O cards.*

from the DSPLINK, these bits will be placed in the most significant part (MSB) of a 32 bit word. The 16 least significant bits of this word contain garbage. Hence the 16 bits in the most significant part need to be shifted to the right (to the least significant part) and the most significant part must be filled with zeros. In this way, a 32 bit word is formed and the

binary value of this word is the same as the binary value of the 16 bits from the DSPLINK. When writing to the I/O cards the opposite must be done.

Data transfers from the DSP to the PC

Data is transferred to/from the TMS320C30 board via the 32 bit Data Registers. Once the address of the data to be accessed has been set up, two 16 bit transfers over the PC I/O bus are required to complete the 32 bit access.

From the DSP point of view it is a matter of *memory-mapped I/O*, while from the PC point of view it is a matter of *I/O mapped I/O*. Figure 3.3 shows the difference between I/O mapped I/O and memory-mapped I/O. In the case of I/O mapped I/O there are two spaces: an I/O space and a memory space. The term I/O space refers to a group of data registers that are addressed separately from the locations in a memory space. The two spaces are separately accessed by using different instructions when writing the software. The I/O

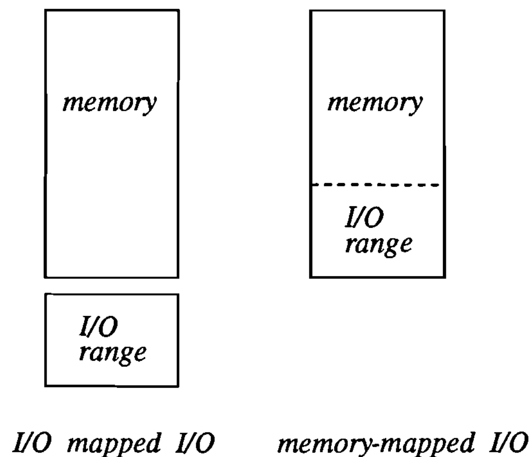


Figure 3.3: *Difference between I/O mapped I/O and memory-mapped I/O.*

space has several control lines like I/O-read and I/O-write. However, in the case of memory-mapped I/O the I/O space is part of the memory itself. Thus the memory range is divided into memory locations and I/O space. In fact every memory location in the I/O space is an I/O port. The separate ports are connected with the databus through addressdecoders.

Data transfers from the PC to the DSP

The interfacing with the PC is established through a number of I/O ports on the PC (I/O mapped I/O), as is mentioned in the previous section . Through these ports the dual ported RAM (DP RAM) can be accessed and several states of the DSP can be controlled.

LSI supplies with the DSP processor board a library of functions to make interfacing to the DSP board easier. This library uses the DP RAM. The functions in the library are written in the C language. They can be used for loading programs and data to and from the DSP board.

The functions in the library can be divided into three groups:

1. Code loading functions
2. DSP control functions
3. Data transfer functions

The first group consists of functions to load the COFF (the output file that the linker produces is in the so-called **C**ommon **O**bject **F**ile **F**ormat). Command line arguments can also be passed to the DSP.

The second group contains all functions that are necessary to control the operation of the DSP. This means functions for selecting the right board, resetting, halting and releasing the processor.

The third group is the main part of the available functions and deals with transferring data to and from the board. Data can be read or written as single values or as blocks of data. One argument of the functions decides where the data is located. In the case of the DP RAM, no extra actions are necessary, but if data must be read or write to or from the main RAM, the DSP must be halted by using the HOLD/HOLD Acknowledge of the TMS320C30. The PC then needs access to the bus to which the DSP is also connected. Only one system can access a bus at a time. A more detailed description of the PC processor board can be found in appendix A.

Interrupt-handling

Figure 3.4 shows what happens if an interrupt occurs. For clearity not all lines and buses are drawn, but only the most important ones. These are the DSPLINK, the primary bus and the ISA bus. The ISA bus forms the connection between the PC and the DSP board. One part of this bus is connected with the DP RAM, while the other part of the bus contains lines that are connected with other parts of the DSP board, like IRQ5. For clearity, only IRQ5 is drawn.

The timer of the first I/O card is used to synchronously trigger all conversions. The timer signal is used for external triggering on the second and third board. The End of Conversion (EOC) generates an interrupt on the DSPLINK. Only one of the three boards is used to generate an interrupt. The DSPLINK uses INT1 (see appendix A, table A.10) for interrupting the DSP. Detection of this interrupt is done by checking the EOC bit in the Status Register of the I/O card (see appendix A, table A.7). In this manner more cards could use this interrupt at the same time. The interrupt generated by the EOC activates an Interrupt Service Routine (C-ISR) on the DSP. In this routine data is used by an algorithm. The C-ISR contains a special command, which interrupts the PC (IRQ5). The output of the algorithm in the C-ISR is set in the DP RAM, just like the parameter variables. The Interrupt Service Routine (ISR) on the PC now knows that data has been set in the DP RAM. This data is read through I/O ports on the PC.

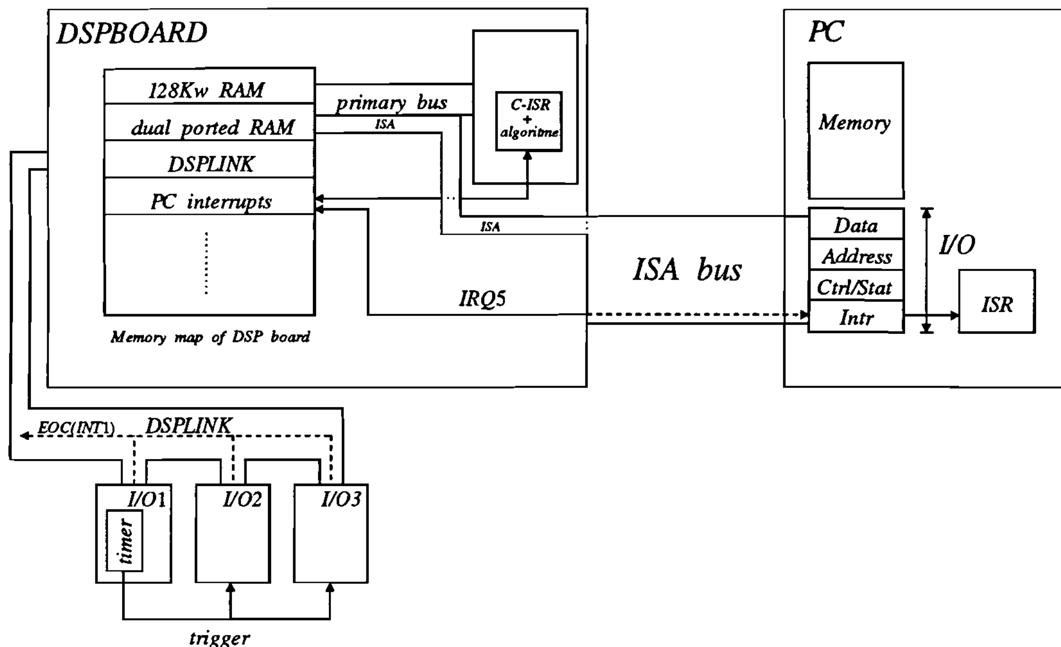


Figure 3.4: *Schematic overview of interrupt-handling.*

3.2 Description of the 16 bit Stereo Interface Boards

3.2.1 Outline description

The 16 bit Stereo Interface Board (SIB) is designed to connect directly to the TMS320C30 PC Processor Board [SIB 91]. The boards consist of two A/D converters and two D/A converters each. Each analog input channel consists of a Programmable Gain input amplifier, a 4th order lowpass filter, a Sample and Hold amplifier and a fast A/D converter giving a 16 bit resolution with 14 bit linearity. The timer, filters and gain amplifiers can all be configured through jumpers and resistor packs (the resistor packs should be used for the filters to obtain the desired cut-off frequency).

Figure 3.5 shows a part of one I/O board (one channel). The differential input signal is first fed through a Programmable Gain amplifier (pga) to support a range of peak to peak input signal levels. It is then passed into a low pass filter and a Sample and Hold amplifier. The output signal of the Sample and Hold will follow its input signal until a trigger pulse is received by the A/D converter. This trigger will enable the signal to be captured and held during the analog-to-digital conversion process. The resulting 16 bit word is clocked out of the A/D converter serial output into a Shift/Storage Register. At this point the sampled data in the Shift Register can be read by the host DSP for processing. Any processed data may also be written back into the Shift Register before the next trigger pulse is received. The Shift Register data is clocked out to the D/A converter when the next data sample is clocked

in. The resulting analog signal is then filtered before being seen at the channel output.

In addition to the two analog channels the board contains a programmable sample rate timer, Control and Status Registers and circuitry to interface the board to the DSPLINK connector. Furthermore, each board has two 1 bit digital input and output channels to be used for synchronization.

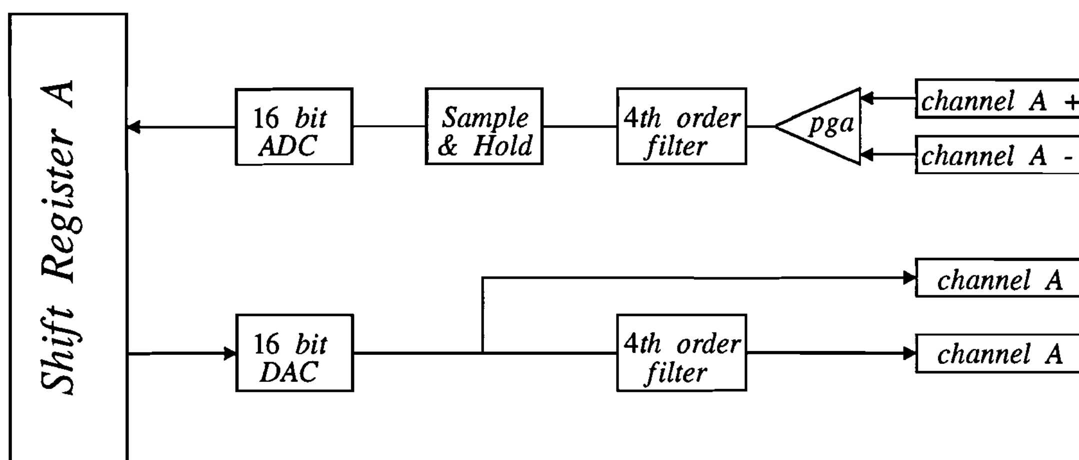


Figure 3.5: *Schematic overview of one I/O board (one channel).*

3.2.2 Interface to the DSP

The board occupies four locations in the I/O addressing range of the DSPLINK master processor board (memory mapped I/O). The conversion can be triggered in three ways. The first way is with the timer on the SIB, the second way is with software through a register and the third way is with an external trigger. The timer of the first I/O card triggers the other cards, as is mentioned above.

3.2.3 Conversion timing

If the on-board timer is used the maximum sampling frequency that can be achieved on a single channel is 157 KHz. This limit is determined by the time taken to serially transfer 16 bits into the Shift Register for transfer over the DSPLINK. If an external trigger is used the pulse width of the trigger pulse will also affect this limit.

Sampling rates up to 300 KHz can be achieved by feeding the input signal to both channels and sampling on each channel alternately. The success of the method is dependent on the sampling rate being such that the next trigger pulse occurs before the end of conversion on one channel. A more detailed description of the Stereo Interface Boards can be found in appendix A.

3.3 The Digital Signal Processor TMS320C30

3.3.1 General description of Digital Signal Processors

As was mentioned before, DSP's have become more popular in the last ten years and now they can be found in many application areas. Why are DSP's so popular, in other words, what is the difference between general purpose processors (e.g. Intel 486) and digital signal processors ?

Most general purpose computers have a so-called *Von Neumann architecture.* This architecture has the following properties:

- There is no significant difference between instructions and data.
- The bit pattern of an instruction is divided into an operation field and an operand field.
- The same memory can be used for storage of data and instructions.

Performing of a program is done in the following four steps:

1. instruction fetch (IF)
2. instruction decode (ID)
3. operand fetch (OF)
4. execute (EX)

These steps are performed sequentially, i.e. starting with an IF and ending with an EX. Therefore, general purpose computers are said to have a *non-pipelined processor.* This is depicted in figure 3.6. In contrast to general purpose computers DSP's don't have a Von Neumann architecture, but a *Harvard architecture.*

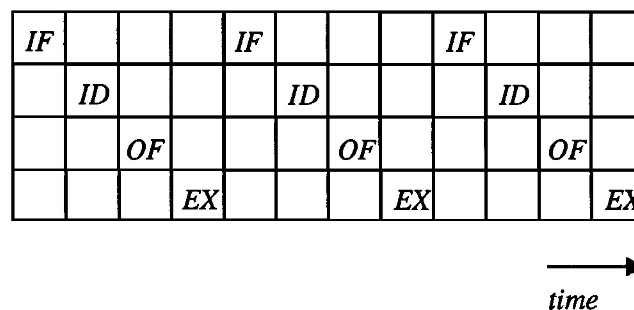


Figure 3.6: *A non-pipelined processor.*

The Harvard architecture has the following properties:

- Data and instructions are stored in different memories.
- These memories each have their own bus connected to the CPU.

The separate buses provide paralellism in instruction fetch and execution. The goal is to have very fast processing. DSP's are said to be *pipelined processors*. This is depicted in figure 3.7. From this figure it is clear that when the processor is decoding the instruction in the second stage, it already fetches the next instruction. A pipelined processor has the

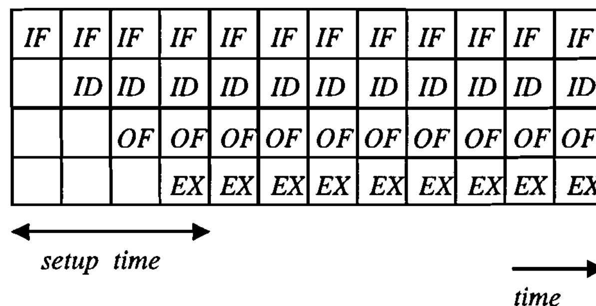


Figure 3.7: *A pipelined processor (4 stages).*

following disadvantages:

- The maximum speed up is only reached after some setup time.
- Decoding an instruction can sometimes cost a lot of time, because the resulting pipeline has to wait.
- Bus conflicts and memory conflicts occur when performing IF, OF and EX.

However, the Harvard architecture prevents conflicts between IF and OF (or between IF and EX) and therefore the pipeline architecture of the CPU is succesfully used in DSP's.

3.3.2 General description of the TMS320C30

The TMS320C30 internal busing and special digital signal processing instruction set provide speed and flexibility [TI 89]. This combination produces a processor capable of executing up to 33 MFLOPS (million floating-point operations per second). The TMS320C30 optimizes speed by implementing functions in hardware that other processors implement through software or microcode.

The TMS320C30 can perform parallel multiply and ALU operations on integer or floating-point data in a single cycle. The processor also possesses a general-purpose register file, program cache, dedicated auxiliary register arithmetic units (ARAU), internal dual-access

memories, one Direct Memory Access (DMA) channel supporting concurrent I/O and a short machine-cycle time. High performance and ease of use are achieved through greater parallelism, greater accuracy and general-purpose features.

Some key features of the TMS320C30 are listed below.

- 60 nsec single-cycle instruction execution time (33 MFLOPS, 16.7 MIPS (million instructions per second))
- one 4K x 32 bit single-cycle dual-access on-chip ROM block
- two 1K x 32 bit single-cycle dual-access on-chip RAM blocks
- 64 x 32 bit instruction cache
- 32 bit instruction and data words, 24 bit addresses
- on-chip DMA controller for concurrent I/O and CPU operation
- two 32 bit timers

Pipeline structure

The TMS320C30 pipeline structure has five major units:

1. *Fetch unit*: fetches the instruction words from memory and updates the program counter.
2. *Decode unit*: decodes the instruction word and performs address generation.
3. *Read unit*: if required, reads the operands from memory.
4. *Execute unit*: if required, reads the operands from the register file, performs the necessary operation and if needed writes results to the register file.
5. *DMA channel*: reads and writes memory.

Priorities have been assigned to each of the functional units. The priorities from the highest to lowest are:

- execute (highest)
- read
- decode
- fetch
- DMA (lowest)

When processing of an instruction is ready to pass to the next higher pipeline level, but that level is not ready to accept a new input, a pipeline conflict occurs. In this case, the lower priority unit waits until the higher priority unit completes its currently executing function.

Memory organization

The memory organization of the DSP chip is depicted in figure 3.8. The total memory space

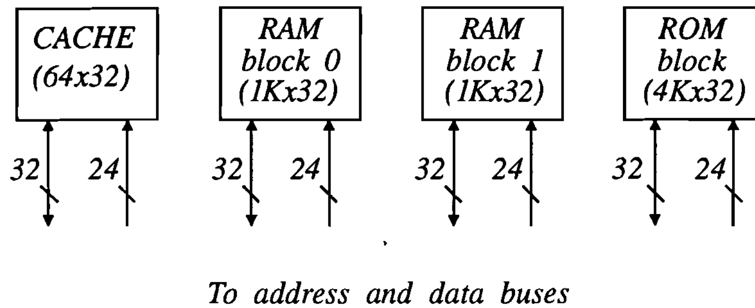


Figure 3.8: *Memory organization of the TMS320C30.*

of the TMS320C30 is 16M (million) 32 bit words. Program, data and I/O space are contained within this 16M word address space, thus allowing tables, coefficients, program code or data to be stored in either RAM or ROM. In this way, memory usage can be maximized and memory space allocated as desired.

The ROM block is 4K x 32 bit. Each RAM and ROM block is capable of supporting two accesses in a single cycle. The separate program buses, data buses and DMA buses allow for parallel program fetches, data reads and writes and DMA operations. In this way the CPU is able to access two data values in one RAM block and perform an external program fetch in parallel with the DMA loading another RAM block, all within a single cycle.

A 64 x 32 bit instruction cache stores often repeated sections of code. This greatly reduces the number of off-chip accesses necessary and allows code to be stored off-chip in slower, lower-cost memories.

The memory map is dependent upon whether the processor is running in the microprocessor mode or the microcomputer mode. In microprocessor mode, the 4K on-chip ROM is not mapped into the TMS320C30 memory map. In the present configuration the microprocessor mode is used.

Timers

The TMS320C30 timer modules are general-purpose 32 bit timer/event counters with two signalling modes and internal or external clocking. The timer modules can be used to signal to the processor or the external world at specified intervals or to count external events. With an internal clock, the timer can be used to signal an external A/D converter to start a conversion or it can interrupt the DMA controller to begin a data transfer. With an external input, the timer can count external events and interrupt the CPU after a specified number of events. Available to each timer is an I/O pin that can be used either as an input clock to

the timer, an output clock signal or a general-purpose I/O pin.

Three memory-mapped registers are used by each timer:

- Global Control Register
- Period Register
- Counter Register

The Global Control Register determines the operating mode of the timer, monitors the timer status and controls the function of the I/O pin of the timer. The Period Register specifies the timer's signalling frequency. The Counter Register contains the current value of the incrementing counter. The counter is set to zero whenever its value equals that in the Period Register. Appendix A gives a more detailed description of the TMS320C30 chip.

3.4 Description of the PC system

The heart of the PC system is a 486 DX2 processor from Intel. The Intel 486 DX2 microprocessor is a fully compatible member of the Intel 486 family. It has a 32 bit architecture with on-chip memory management, floating-point and cache memory units. On-chip cache memory allows frequently used data and code to be stored on-chip reducing accesses to the external bus. A clock doubler has been added to speed up internal operations to twice that of an Intel 486 DX microprocessor running with the same bus clock. Thus the i486 DX2 internally doubles the supplied clock signal: a 33 MHz i486 DX2 internally becomes a 66 MHz i486 DX. All internal units of the CPU such as, for example, the ALU, decoding unit, floating-point unit, segmentation unit and access to the 8 Kbyte cache run at twice the speed. With the 33 MHz i486 DX2 this leads to 66 MHz. Only the bus unit runs from the external clock.

The i486 has a very flexible bus in view of its size and cycle behaviour. The data bus can be operated with a width of 8, 16 and 32 bits. In the present configuration a 16 bits ISA bus is used for connection with the DSP board. The bus width can be defined separately and independently for every cycle and need not be predefined at the time of system design. Thus the i486 can cooperate flexibly with memories and peripherals of various widths. The data transfer rate of the data bus is about 8 Mb/sec [Sja 90]. The bus clock-frequency is 7.159 MHz.

For real-time applications the so-called *interrupt latency* is very important. The interrupt latency indicates the time lag between a device's request for service and the actual servicing of that request. Maximum interrupt latency should be expected during multiplication, division, variable bit shift and rotate, etc. The interrupt latency for the ISA bus is about 1μsec - 5μsec [Phi 89].

3.5 General purpose processor versus DSP

In this section a Pentium processor (66 MHz) and a TMS320C30 processor are compared. The TMS320C30 processor is a so-called RISC processor. RISC stands for **R**educed **I**nstruction **S**et **C**omputer [Mes 94]. Such processors have a significantly reduced instruction set. In contrast, CISC (**C**omplex **I**nstruction **S**et **C**omputer) processors have an extensive instruction set of more than 300 machine instructions, complex addressing schemes and micro-encoding of the processor instructions. In a CISC processor, like the Pentium processor, a decoding unit is present which decodes a machine instruction into micro-instructions and transmits them to a micro-code queue. In a RISC processor, on the other hand, the machine instructions are directly executable and don't need to be decoded into a sequence of micro-instructions. Thus a RISC processor operates faster than a CISC processor.

The Pentium processor delivers over 100 MIPS at its initial frequency, 66 MHz [NAG 93]. The address bus is 32 bits wide and the data bus is 64 bits wide. The bus can support up to 528 Mb/sec of data throughput. Data can be transferred in 8, 16, 32 or 64 bit units. The Pentium processor is a dual issue super-scalar CPU, meaning that it can prefetch, decode, execute and write-back two instructions in one clock cycle.

Table 3.1 shows the properties of both processors.

Table 3.1: *Important differences and similarities between a Pentium and a DSP*

TMS320C30	**Pentium (66 MHz)**
Harvard architecture	Von Neumann architecture
RISC	CISC
32 bit (parallel)	64 bit
floating-point	floating-point
16.7 MIPS	100 MIPS
2 instructions/clock cycle	2 instructions/clock cycle

The Pentium processor seems to be faster than a DSP (64 bit, 100 MIPS). However, a Pentium processor is a CISC processor which cannot perform parallel operations (Von Neumann architecture). Moreover, for multi-tasking purposes it is recommended to use more than one processor. Thus if the real-time environment is implemented on a single Pentium processor this will result in a degradation of performance because of pseudo multi-tasking (one processor). Hence a DSP processor together with a general purpose processor (i486, Pentium) is preferred to a single Pentium processor.

3.6 Summary

In this chapter a brief description is given of the hardware configuration of the DSP-PC system. In broad outlines, the system can be divided into three parts: the I/O cards, the TMS320C30 PC Processor Board (DSP board) and the PC.

The I/O cards are connected with the DSP board through a special bus. This bus is called DSPLINK and is designed for 16 bits parallel transfers. When reading from the I/O cards a 16 bits shift to the right must be performed. When writing to the I/O cards the opposite has to be done. The timer of the first I/O card triggers the other cards. In this way a maximum sampling frequency of 157 KHz can be achieved.

On the DSP board there are three memory locations: a 64 Kw DP RAM, a 128 Kw main RAM and a 2 Kw on-chip RAM. The 64 Kw DP RAM is used for communication with the PC. From the DSP point of view it is a matter of memory-mapped I/O, while from the PC point of view it is a matter I/O mapped I/O. A library of functions can be used to make interfacing to the DSP board easier. When the main RAM is used the DSP must be halted, resulting in degradation of performance. The TMS320C30 chip from Texas Instruments forms the heart of the DSP board. Due to its so-called Harvard architecture, this chip can perform parallel multiply and ALU operations on integer or floating-point data in a single cycle.

The PC system has an i486 processor running at 66 MHz. This processor has a 32 bit architecture with on-chip memory management, floating-point and cache memory units.

A Pentium processor and the TMS320C30 processor are compared. Although a Pentium processor seems to be faster than a DSP, the latter one is more suitable for acquiring and processing floating-point data because of its Harvard structure.

Chapter 4

The Ward & Mellor method

Now that aspects of real-time systems and the hardware configuration of the system are well-known, a start is made with the software design. There are several techniques for analysis and design of real-time systems [Ver 90]:

- Gane & Sarson
- Ward & Mellor
- Hatley & Pirbhai
- Systems Engineering Methodology
- Yourdon

The Ward & Mellor method is used, because this is probably the most well known technique for designing real-time systems and as a result there is a lot of literature available on this method. Furthermore, this method allows modeling of control and data as well as modeling of time-continuous and time-discrete behavior. Both modeling aspects play an important role in the modeling of real-time systems.

4.1 Description of the Ward & Mellor method

The Ward & Mellor method is a structured development method for real-time systems [WM 85]. A modeling language which has to be adequate for real-time systems must be capable of separating the *data* needed by the process from the *control* that actually makes the process operate. An adequate systems modeling language must allow modeling of control as well as data.

Another aspect which plays an important role in the modeling of real-time systems is the behavior (*time-continuous* or *time-discrete*) of the system. The distinction between time-continuous and time-discrete behavior is closely related to the distinction between data and control. In order to model the time-related complexities of a real-time system, an adequate modeling language must also be able to distinguish time-continuous and time-discrete behaviors and must be able to model the interactions between the two. Because the Ward & Mellor method provides with both the distinction between control and data and the distinction between time-continuous and time-discrete behavior, the method is suitable for modeling real-time systems.

The transformation scheme according to Ward & Mellor models a system as an active entity, i.e. as a network of activities that accept and produce data and control messages. It denotes graphically the layout of the transformations that operate on flows that cross the system boundary and it is the active part of the system that responds to events that occur in the environment. In the next sections only the conventions that are used for modelling the system are described.

4.1.1 The static view

Consider the processing applied to an encrypted message to produce a message in plain text. The associated transformation scheme is shown in figure 4.1. The notation shows inputs and

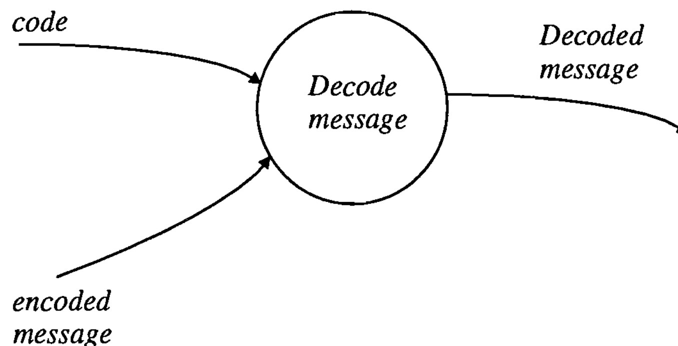


Figure 4.1: *Basic notation of a transformation scheme.*

outputs as labeled arrows (*flows*) and represents the work, which has to be done to produce the outputs, as a labeled circle (*transformation*). The time-continuous behavior is denoted by an arrow with a double arrowhead, while the time-discrete behavior is denoted by an arrow with one arrowhead. The system is decomposed hierarchically from one circle and some arrows (general description) into many circles and many arrows (detailed description).

Stores are represented as two parallel lines. In figure 4.2 *System status* is a store. Although stores are typically sets, a single item which is operated on by various transformations could also be shown as a store. The store notation is thus used to represent an item or set that is operated on by a group of transformations, but whose basic character remains unchanged.

The conventions for connecting transformations to stores are related to the *net flow* between

them in the following ways: the flow connecting the store and the transformation is not labeled; it only represents availability of the store to the transformation. An arrow head pointing from a store to a transformation means that the transformation uses something from the store. An arrow head pointing from a transformation to a store means that the contents of that store is changed. A bi-directional arrow between a transformation and a store means that both preceding characteristics apply.

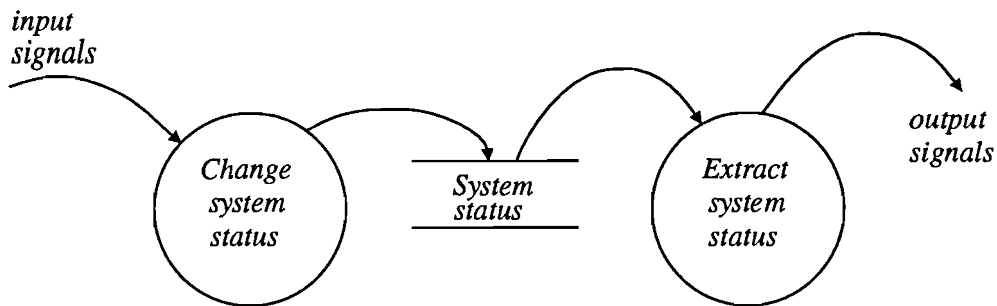


Figure 4.2: *System status as a store.*

There is an important distinction between a pair of transformations connected by a flow and a pair connected through a store. On the left-hand side of figure 4.3, there is a *causal connection* between transformations X and Y. The production of an output by X causes Y to operate and so X and Y must be synchronized, i.e. when the result of the first transformation comes available the second one starts. On the right-hand side of figure 4.3, there is no causal connection. The production of output by X has no immediate effect on Y.

4.1.2 The event view

In the dynamic view of a system, a time-discrete data flow has two distinct characters. It represents both the contents of the data and the occurence of the flow as an input or output at a specific point in time. Most real-time systems contain flows that have no contents; they are simply signals that indicate that something has happened or give a command. Such flows are called *event flows* and they are represented as dotted arrows. A transformation that accepts only event flows as inputs and produces only event flows as outputs is called a *control transformation.* It is represented as a dotted circle (see figure 4.4).

The behavior of a control transformation can be desribed by a *state-transition diagram.* In figure 4.4 a single switch controls the lamp so that the lamp will turn on or off. In this way, there are two behaviors (on and off). The components shown in the transition diagram are the state, represented by an rectangle; the transition, represented by an arrow; the transition condition, shown adjacent to the transition above the line and the transition action, shown adjacent to the transition below the line.

A *state* represents an externally observable mode of behavior. The name of the state is the

name of the behavior exhibited by the system. Each state represents an unique status of the control transformation's memory and the transformations can be in only one state at a given time. Initial states are shown by a transition arrow pointing into the designated state with no source state. The initial state should be thought of as the behavior of the system before any transitions have occured.

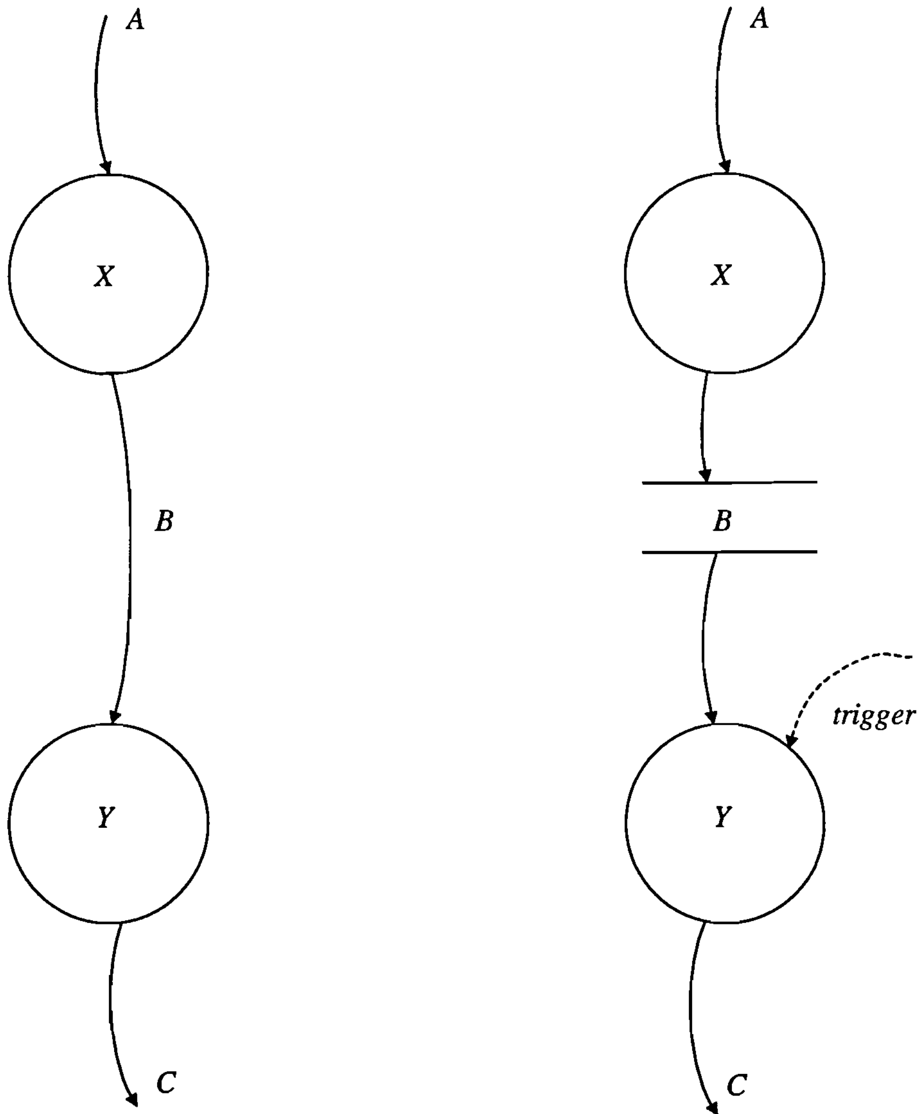


Figure 4.3: *Causal and non-causal interfaces.*

A final state has transition arrows entering, but has no arrows to other states. *Transitions* represent the movement from one state to another. *Conditions* cause the system to make a transition. Each condition is identified with an input event flow that signals that the condition has occured. *Actions* are taken as the transition occurs. An action is a single indivisible activity which is identified with output event flow(s). Furthermore, the information of a

transition diagram may also be represented by constructing a *state-transition table* where rows are conditions, columns represent the current state and the intersection of a row and a column defines the new state produced when the condition occurs in the current state. The principal advantage of this scheme is that one is forced to consider all possible conditions for each state. *Action tables* serve as a companion technique to denote the actions. For the transition diagram in figure 4.4 both tables are listed in tables 4.1 and 4.2.

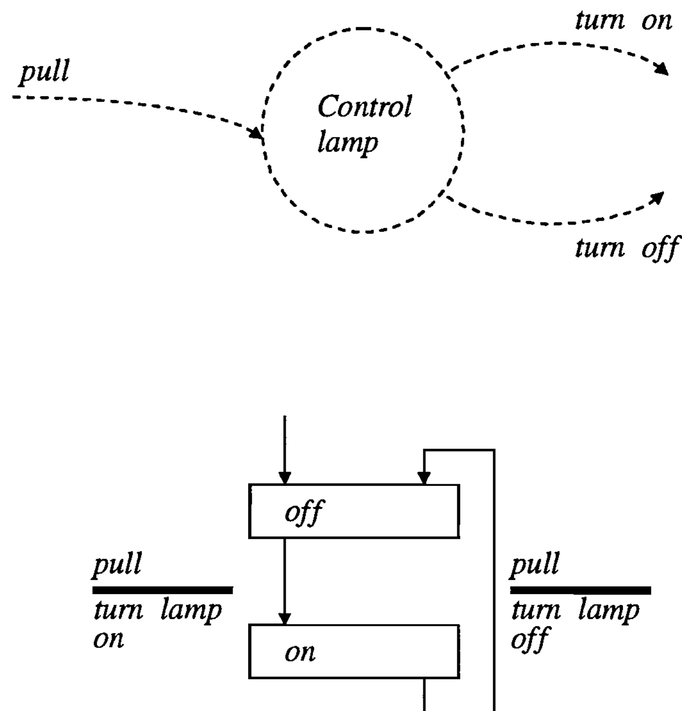


Figure 4.4: *Control transformation with a transition diagram.*

Table 4.1: *Transition table for the transition diagram in figure 4.4*

	off	**on**
pull	on	off

Table 4.2: *Action table for the transition diagram in figure 4.4*

	off	**on**
pull	turn lamp on	turn lamp off

Before a system can be defined using the items that were described, the following rules must be taken into account:

- There is no data sink, i.e. every transformation has an output.
- Conservation of data: the inputs should be sufficient to produce the outputs.
- If a data transformation has more than one input or if it is composed of more elements, then all inputs and elements have to be present for the transformation to take place.

4.2 Summary

The Ward & Mellor method is described as a structured development method for designing real-time systems. The method is suitable for real-time systems, because it provides with both the distinction between control and data and the distinction between time-continuous and time-discrete behavior.

In the static view a transformation scheme consists of labeled arrows (flows) and a labeled circle (transformation), denoting the work that has to be done to produce the outputs. When a pair of transformations is connected through a store there is a non-causal connection between the transformations. If, on the other hand, a pair of transformations is connected by a flow then there is a causal relationship between both transformations. The system is decomposed hierarchically from one circle and some arrows into many circles and many arrows.

In the dynamic view of the system event flows are represented as dotted arrows. Transformations that accepts only event flows as inputs and produces only event flows as outputs are called control transformations. These transformations are represented as dotted circles. Their behavior can be described by a state-transition diagram. The information of a transition diagram is represented by constructing a state-transition table and an action table.

Chapter 5

Software design of the real-time environment

5.1 System's requirements

Before a start is made with the software design an outline is given of the requirements of the real-time environment. The design objective was formulated as:

Design and realize a real-time environment on the DSP-PC system in order to measure and control signals in real time. The environment must be as flexible as possible, which means that every arbitrary algorithm should be implemented on the system without knowing the technical details.

The system's requirements can be summarized as follows:

- Log data to the hard disk.
- Display signals on screen (graphic mode).
- Display variables on screen (text mode).
- Display settings (settable parameters, selected input and output channels).
- Set sample frequency.
- Set algorithm parameters.
- Process elemental commands like start, quit and reset.
- Perform a simple debug facility.

In chapter 2 some aspects and techniques concerning real-time systems were discussed. The aspects and techniques that are used for the implementation of the real-time environment can be summarized as follows:

- The design objective is satisfied by designing the *Controller tasks* of figure 2.1.
- The DSP-PC system is a clock-based system meaning that synchronization is obtained by using the internal clock of the DSP processor.
- Foreground tasks are implemented on the DSP, while background tasks are implemented on the PC.
- Tasks that run on the DSP all have hard time constraints and thus the real-time environment is a hard real-time environment. However, tasks that run on the PC have soft time constraints and in that case the real-time environment is a soft real-time environment.
- Resources like the hard disk and the input and output devices need not to be shared by several concurrently active tasks. Therefore, in the design and realization of the real-time environment no attention is paid to mutual exclusion, deadlock and code sharing.

5.2 General description of the system

The scheme in figure 5.1 covers the whole system. This highest level of the model is called the *context scheme* and is given in figure 5.1. The sample frequency, the input channel(s) to

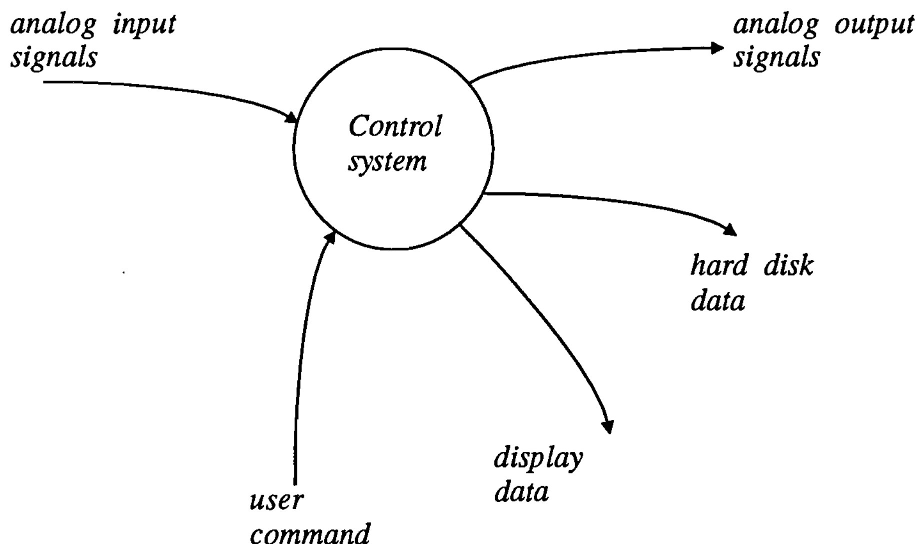


Figure 5.1: *Context scheme of the system.*

be read from, the output channel(s) to be write to, the signals to be displayed on screen, the signals to be logged to hard disk and the parameters are set by the user. The *Control system* transformation reads data from the input channels, performs the algorithm, converts the

selected parameters to analog output signals, logs data to the hard disk and displays the selected signals on screen.

5.3 Detailed description of the system

The *Control system* can be described more detailed as is shown in figure 5.2.

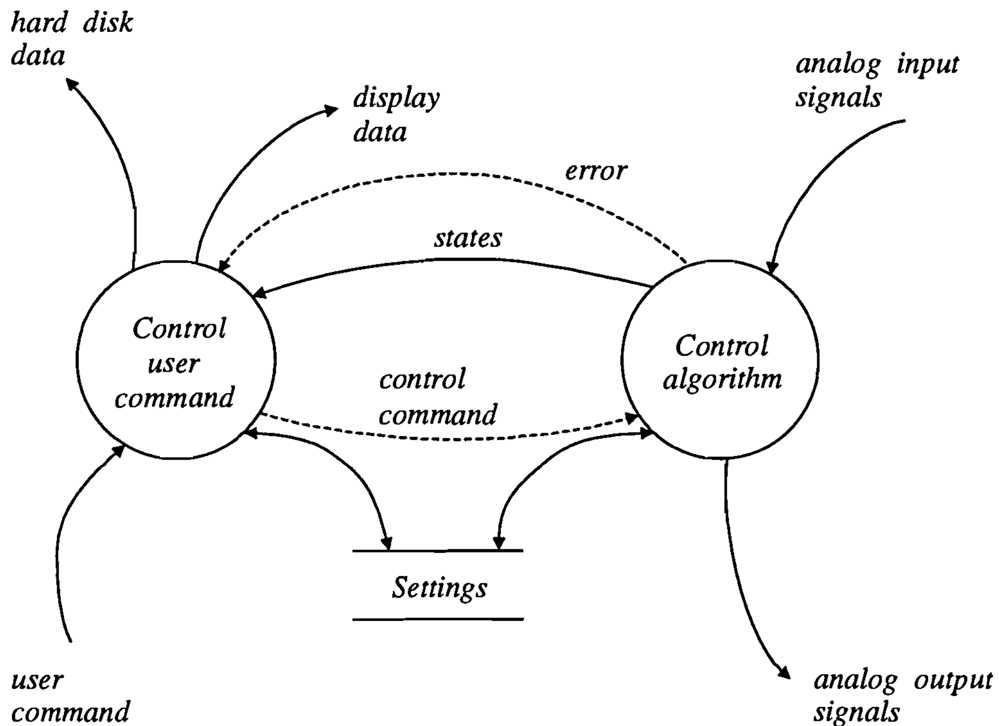


Figure 5.2: *Description of the system at level one.*

The *control command* flow contains commands like start, stop and reset. A reset is a stop immediately followed by a start. These commands can be set by the user and are controlled by the *Control user command* transformation. The *Settings* store contains the sample frequency, definitions of the input and output channels to be used, definitions of the signals to be displayed, scaling of the axes for the signals to be displayed and the settable parameters. A store must be used, because the production of output by the *Control user command* transformation has no immediate effect on the *Control algorithm* transformation (see section 4.1.1).

The *states* flow contains data which has to be saved to the hard disk or displayed on screen. This flow does not have a store, because storing of data starts as soon as data is available. Thus the production of output of the *Control algorithm* transformation has an immediate effect on the *Control user command* transformation. When a start command is given by the user, the *Control algorithm* transformation converts the analog input signals into analog

output signals. The algorithm uses the contents of the *Settings* store. When the algorithm is performed, data is logged to the hard disk and displayed on screen. If an error occurs, this is displayed on screen. By dividing the *Control system* transformation in figure 5.1 into two transformations a distinction is made between the PC side and the DSP side. The *Control user command* transformation controls tasks which are running in the background, while the *Control algorithm* transformation controls the algorithm, which is running in the foreground. Thus the *Control user command* transformation refers to the PC side and the *Control algorithm* refers to the DSP side.

Figure 5.3 is a decomposition of the *Control algorithm* transformation, while figure 5.4 is a decomposition of the *Control user command* transformation. Because the communication between the DSP and the PC takes place through the DP RAM both figures have the same stores. These three stores form the *Settings* store of figure 5.2.

The parameters that can be set by the user are stored in the *Settable parameters* store. The *Selection* store contains information on signals to be displayed, signals to be logged to hard disk and signals to be converted to analog output signals.

When all settings have been done by the user, the *Process user commands* transformation executes the user commands. The *Control tasks* transformation executes the control commands that are get from the *Process user command* transformation. As is mentioned in section 4.1.2 the behavior of a control transformation can be described through a state-transition diagram and a transition and action table. The state-transition diagram and transition and action table for the *Control tasks* transformation are given in figure 5.5 and table 5.1 and 5.2 respectively. When the *Timer* transformation is started by the *Control tasks* transformation, it reads the sample frequency from the *Sample frequency* store. Then the *Convert signals* transformation is triggered. This transformation converts the analog input signals into digital signals. When the *Perform algorithm* transformation is started by the *Control tasks* transformation, the parameters that are set by the user are read from the *Settable parameters* store. With these parameters and data from the input signals the algorithm is performed. When the *Timer* transformation triggers the *Convert selected parameters* transformation, the selected parameters are converted into analog output signals.

A more detailed description of the *Control tasks* transformation is depicted in figure 5.6. The function of the transformation is split in accepting and executing the control command for the timer and the algorithm. The *Control algorithm* transformation is responsible for starting and stopping the algorithm. When an error occurs during the performing of the algorithm, this is reported to the user interface. The *Control timer* transformation is responsible for starting and stopping the timer. The *Put hard disk data* transformation is responsible for logging the selected states to the hard disk. The state-transition diagrams, transition tables and action tables for the transformations in figure 5.6 are given in figure 5.7 and tables 5.3 through 5.7.

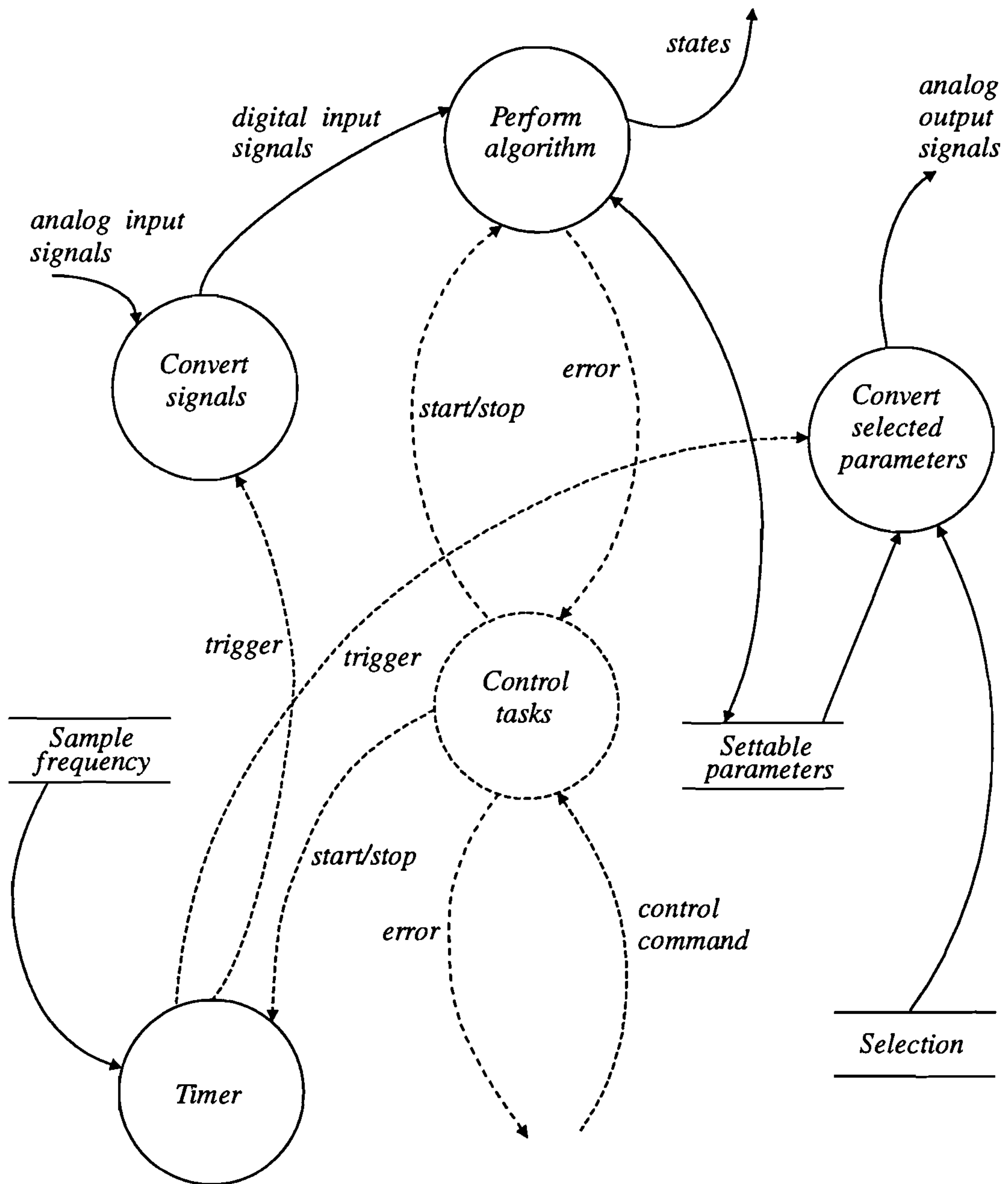


Figure 5.3: *Decomposition of the Control algorithm transformation.*

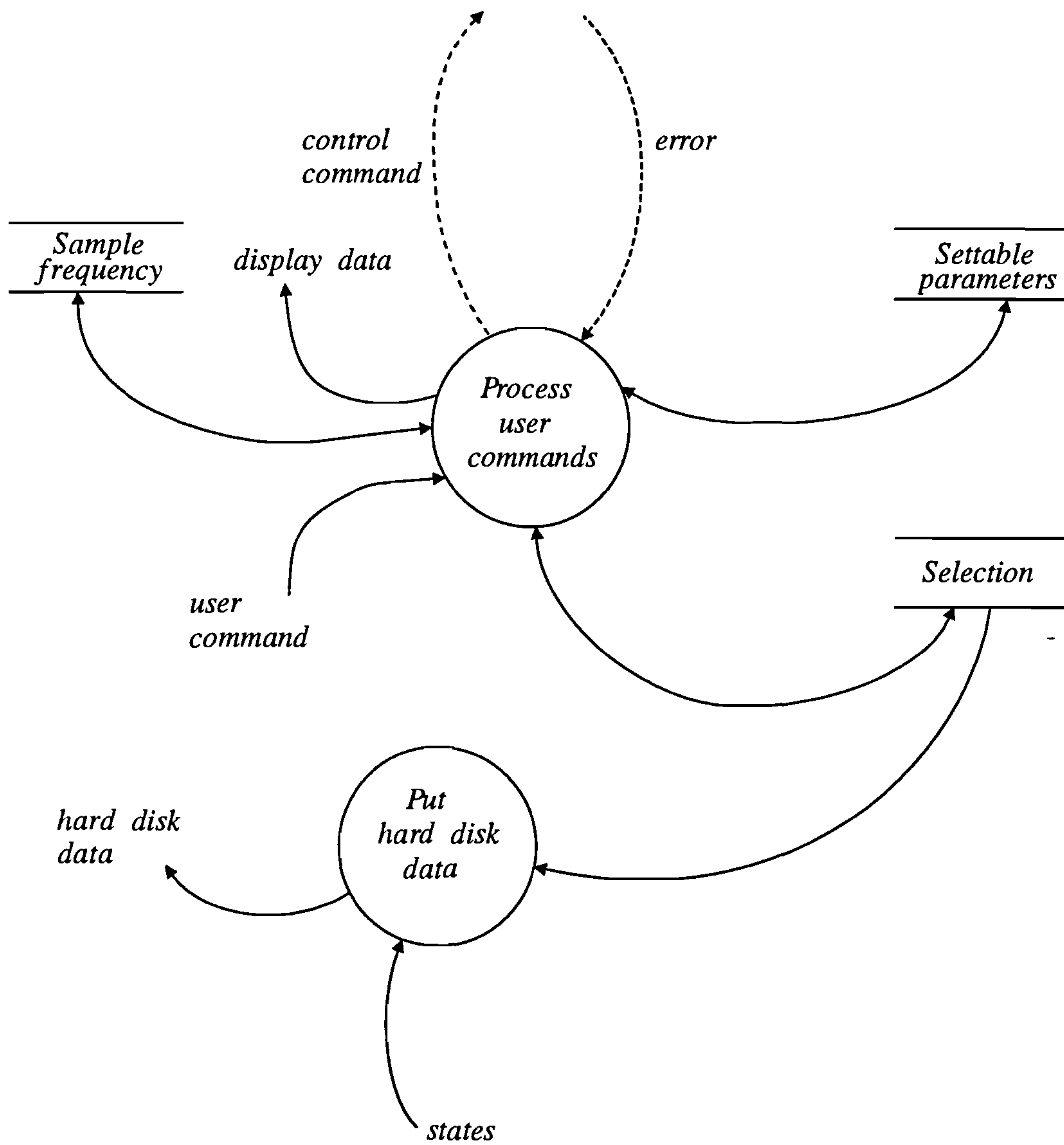


Figure 5.4: *Decomposition of the Control user command transformation.*

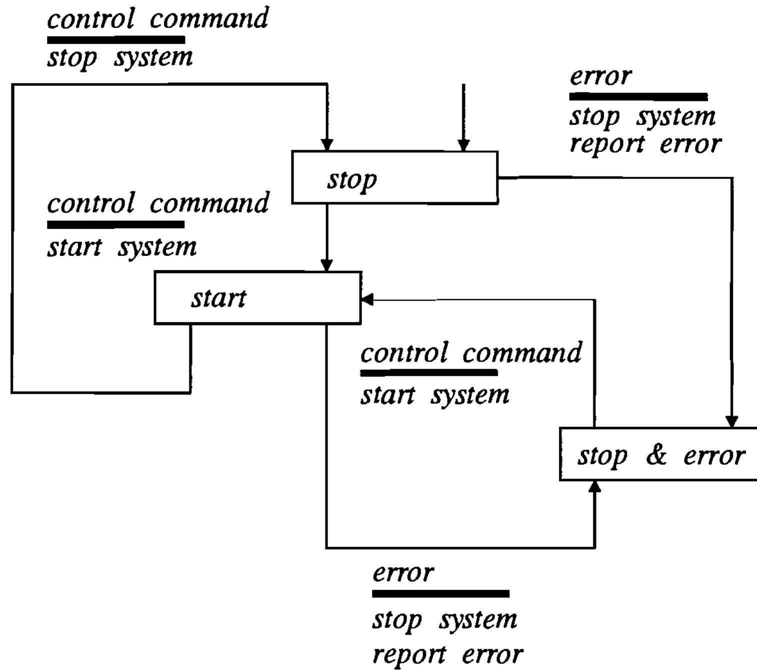


Figure 5.5: *State-transition diagram for the Control tasks transformation.*

Table 5.1: *Transition table for the Control tasks transformation*

	start	**stop**	**stop & error**
control command	stop	start	start
error	stop & error	stop & error	

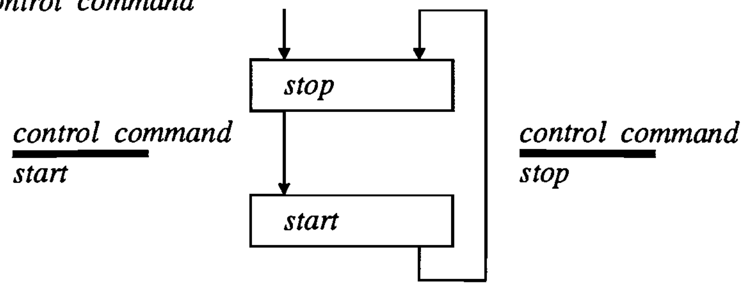
Remark: a blank box in the transition table means that for a particular condition no transition occurs for a state (ignore transition). However, in the implementation this transition must be taken into account, because otherwise malfunction of the system may occur. For example, if the system is in the *stop & error* state an error condition has no effect on this state (the system remains in the *stop & error* state). However, the system must be told that no transition is needed.

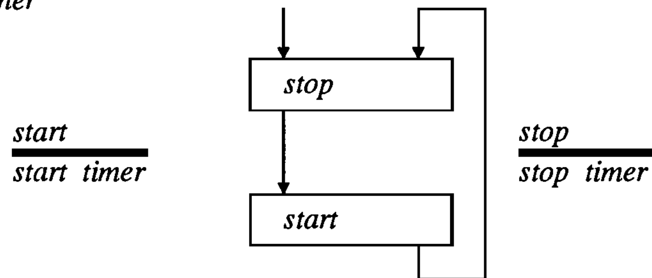
Table 5.2: *Action table for the Control tasks transformation*

	start	**stop**	**stop & error**
control command	stop system	start system	start system
error	stop system & report error	stop system & report error	

Remark: a blank box in the acion table means that for a particular condition no action occurs for a state (ignore action). However, in the implementation this action must be taken into account, because otherwise malfunction of the system may occur.

Figure 5.6: *Detailed view of the Control tasks transformation.*





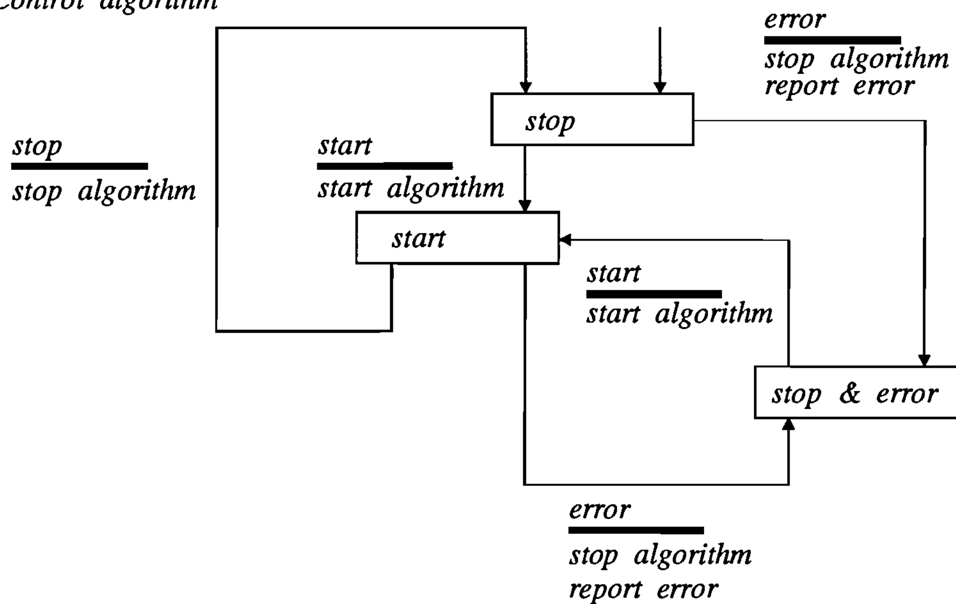


Figure 5.7: *State-transition diagrams for the transformations in figure 5.6.*

Table 5.3: *Transition table for the Execute control command transformation*

	start	**stop**
control command	stop	start

Remark: the action table for the *Execute control command* transformation is the same as its transition table.

Table 5.4: *Transition table for the Control timer transformation*

	start	**stop**
start		start
stop	stop	

Table 5.5: *Action table for the Control timer transformation*

	start	**stop**
start		start timer
stop	stop timer	

Table 5.6: *Transition table for the Control algorithm transformation*

	start	**stop**	**stop & error**
start		start	start
stop	stop		
error	stop & error	stop & error	

Table 5.7: *Action table for the Control algorithm transformation*

	start	**stop**	**stop & error**
start		start algorithm	start algorithm
stop	stop algorithm		
error	stop algorithm & report error	stop algorithm & report error	

A detailed view of the *Process user command* transformation is depicted in figure 5.8.

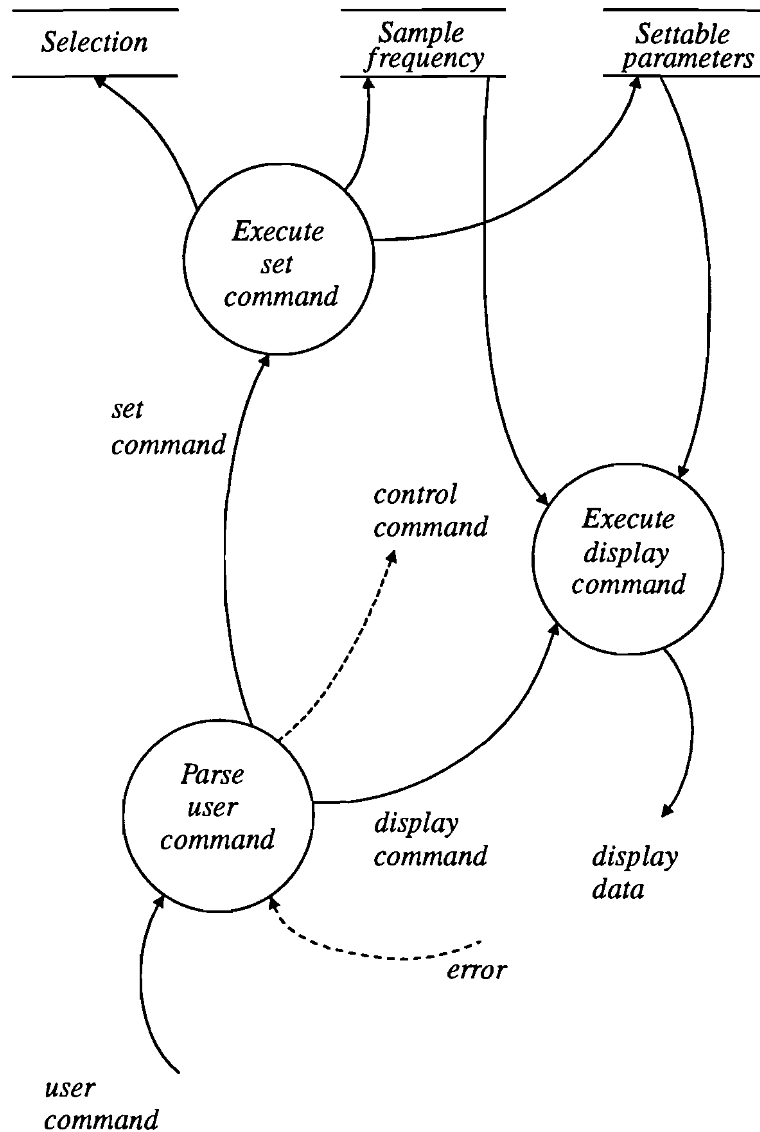


Figure 5.8: *Detailed view of the Process user command transformation.*

The *Parse user command* transformation processes and redirects the commands that are given by the user. The commands are divided into three parts: control, set and display. These parts refer to the three modules in section 2.4.

Control commands are:

- start the system
- quit the system
- reset the system
- halt the system

Set commands are:

- set parameters
- set scaling of axes for signals that must be displayed on screen
- set sample frequency

Display commands are:

- show (settable) parameters
- show help screen
- show error message
- show settings

The *Execute display command* transformation diplays the errors and settings (settable parameters, selected input and output channels, sample frequency, etc.) on screen. The *Execute set command* transformation executes the set commands that are listed above.

5.4 Summary

In this chapter the Ward & Mellor method is applied on the DSP-PC system. By dividing the context scheme into two transformations a distinction is made between the PC side and the DSP side. On the PC the user interface must be implemented, while on the DSP the algorithm must run. Thus the background tasks has to be implemented on the PC, while the foreground tasks must run on the DSP.

The two transformations at level one are further decomposed, resulting in two different schemes with the same stores. The *Control tasks* transformation (a control transformation) of one scheme and the *Process user command* transformation of the other scheme are the only two transformations that are further decomposed. Because the *Control tasks* transformation is a control transformation, its behavior and the behavior of the control transformations of its decomposition are described through state-transition diagrams and transition and action tables.

Chapter 6

Software implementation of the real-time environment

6.1 Implementation of the communication PC <-> DSP

6.1.1 Data transfer from DSP to PC

The fact that a PC is used as host means that all or part of the user command processing should be placed on the PC and that the interface between PC and DSP must be used to communicate. This would mean that all data must be placed in the DP RAM. For the variables that should be set from the PC, like the sample frequency and commands for the DSP, this does not have to be a problem. However, this can be a problem for the parameter variables. When these are placed in the DP RAM they have to share the data and address bus with the program instructions. Therefore, variables are placed in the on-chip RAM, which allows double access in one instruction cycle. This also prevents bus conflicts with the instruction fetch. Thus to allow the best performance the program instructions are placed in a memory area separate from the program data. In this way, the parallel instruction and data fetch can be carried out, increasing processing speed.

The algorithm is placed in the main RAM of the DSP board (see figure 6.1). Data is divided into three parts:

1. constants
2. variables
3. states

Constants are parameters that don't change during the performing of the algorithm. Variables

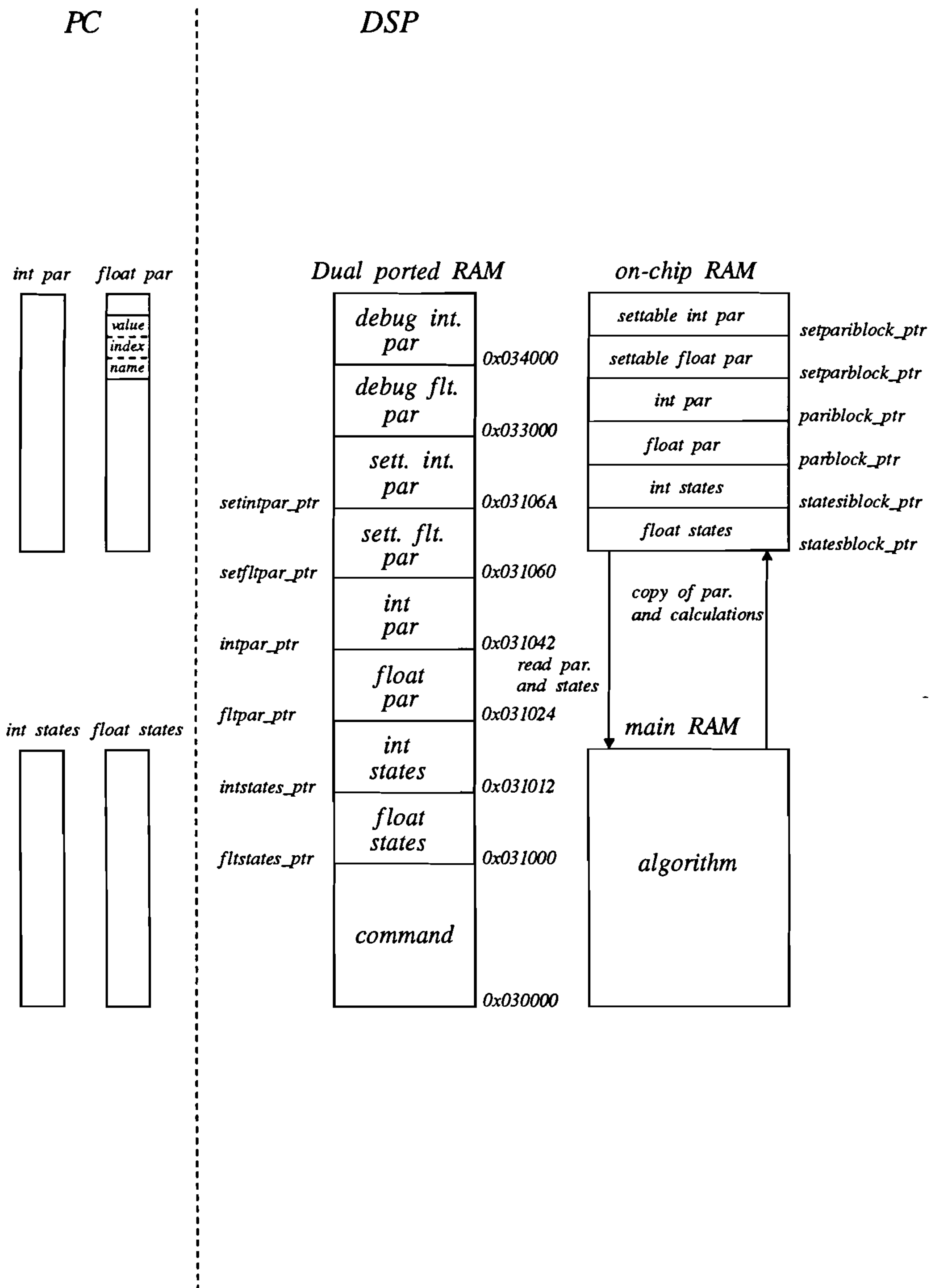


Figure 6.1: *Schematic overview of organization of data.*

are parameters that can be set from the PC (settable parameters), thus those parameters may change during the process. Finally, states are signals that vary each time instance. For example, in equation

$$x[k+1] = Ax[k] + Bu[k]$$
$$y[k] = Cx[k+1] + D$$

u, x and y are states, while A, B, C and D are parameters (constants or variables). States are considered as given variables that can not be set. States and settable parameters are divided into two parts: floats and integers. The reason for this is that floats are treated different from integers.

The on-chip RAM is organized as is depicted in figure 6.1. From this figure it is clear that the on-chip RAM is divided into six parts:

1. float states
2. integer states
3. float parameters
4. integer parameters
5. settable float parameters
6. settable integer parameters

The organization of the DP RAM is also depicted in figure 6.1. From this figure it is clear that the DP RAM is divided into nine parts:

1. command
2. float states
3. integer states
4. float parameters
5. integer parameters
6. settable float parameters
7. settable integer parameters
8. debug float parameters
9. debug integer parameters

The command part is used for transferring the commands that are given by the user (like start, quit and reset), the sample frequency, the acknowledgement signal, the error signal, the pc interrupt counter, the DSP interrupt counter, etc. The command part is divided into the following addresses:

- `error_adr`
- `command_adr` (start, quit, reset)
- `ack_adr` (acknowledgement)
- `samp_freq_adr` (sample frequency)
- `ad_adr` (six ad channels)
- `da_adr` (six da channels)
- `pcint_adr` (pc interrupt counter)
- `dspint_adr` (DSP interrupt counter)
- `ad_status_adr` (status (on/off) of the ad channels)
- `da_status_adr` (status (on/off) of the da channels)
- `debug_status_adr` (status (on/off) of the debug mode)
- `flag_adr` (flag for controlling the debug mode)
- `debug_index_adr` (index for the debug mode)

The definitions of the addresses are defined in files *dspmain.h* and *pcmain.h.* The offset values are defined in file *defs.h.* The definitions of the addresses in file *dspmain.h* are consistent with the ones in file *pcmain.h.* The debug parameters are explained in section 6.4.

Initialization

When initializing, states and non-settable parameters are copied from the on-chip RAM to the DP RAM. This is done in function `main()` in file *dspmain.c30*, which is running on the DSP (see appendix B for a detailed listing of all directories and their files). A description is given of how float states are copied from the on-chip RAM to the DP RAM. Integer states, float parameters and integer parameters are treated in the same way.

The definitions of the float states are listed in file *defs.h.* Here, they are defined as `statesblock_ptr[x]`, where `x` denotes an integer ("ptr" stands for "pointer"). In function `main()` in file *dspmain.c30* the `malloc` function is used to allocate a part of the on-chip RAM for the float states. In file *defs.h* the size of the space to be allocated is defined. In the present configuration free space is allocated for 18 float states, 18 integer states, 30 float parameters and 30 integer parameters.

An address to pointer conversion takes place by using
`fltstates_ptr = (float *)statesblock_adr` ("adr" stands for "address").
`Statesblock_adr` is the address of the float states block in the DP RAM (see figure 6.1). The float states are copied from the on-chip RAM to the DP RAM by using
`fltstates_ptr[x] = statesblock_ptr[x]`. The constants and parameters that not have to be displayed on screen remain in the on-chip RAM.

Run-time

States and parameters are read from the DP RAM by using the library of functions that Loughborough Sound Images supply with the DSP board (see section 3.1.2). This library is listed in file *30librar.c* [PB 91]. Float and integer parameters are displayed on screen by using functions `RdBlkFlt()` and `RdBlkInt()` from the library. This is done in function `main()` in file *pcmain.c*. The float and integer parameters are displayed on screen by calling function `show_algorithm_variables()` in *pcmain.h*. A description is given of how float states are displayed on screen. Integer states, float parameters and integer parameters are treated in the same way.

Float states are placed in the DP RAM at address `statesblock_adr` (see files *dspmain.h* and *pcmain.h*). Through function
`RdBlkFlt (statesblock_adr, DUAL, fltstatesindex, fltstates)` in file *pcmain.c* a block of float states is read from the DP RAM at address `statesblock_adr`. The size of this block is `fltstatesindex`, which is defined in file *pcmain.h*. In the present configuration 18 float states, 18 integer states, 30 float parameters and 30 integer parameters can be displayed on screen.

The block of float states is placed in array `fltstates`. Then the contents of `fltstates` is copied to a structure `fltstatesarray`. This structure has three fields: a name, an index and a value. It is initialized in function `init_data()` at the end of file *pcmain.c*. In this way each float state is given a name and index which is consistent with the ones defined in *defs.h*.

6.1.2 Data transfer from PC to DSP

Initialization

Settable parameters are treated differently. A description is given of how settable float parameters are dealt with. Settable integer parameters are treated in the same way.

The definitions of the settable float parameters are listed in file *defs.h*. In the main function of *dspmain.c30* the `malloc` function is used to allocate a part of the on-chip RAM for the settable float parameters. In file *defs.h* the size of the space to be allocated is defined. In the present configuration free space is allocated for 10 settable float parameters and 10 settable integer parameters.

An address to pointer conversion takes place by using
`setfltpar_ptr = (float *)setfltpar_adr`. The settable float parameters are initialized by copying their initial values from the PC to the DSP. Thus the initial values are placed in the DP RAM and copied from the DP RAM to the on-chip RAM by using
`setparblock_ptr[x] = setfltpar_ptr[x]`.

Run-time

Settable float parameters are initialized by copying their initial values from the PC to the DSP. The settable float parameters are initialized in function `init_sett_par()` in file

pcmain.c. Just like the float states a structure is used for the settable float parameters. This structure is defined `setfltpararray` and has three fields: a name, an index and a value. The contents of this structure is copied to array `setfltpar`. This array is placed in the DP RAM at address `setfltpar_adr` by using function
`WrBlkFlt (setfltpar_adr, DUAL, setfltparindex, setfltpar)`. The size of this block is `setfltparindex` and is defined in file *pcmain.h.* In the present configuration 10 float parameters and 10 integer parameters can be set by the user.

When the user wants to change the value of a float parameter a call is made to function `float_par()` in file *pcmain.c.* The values of the settable float parameters in the DP RAM are then displayed on screen. When the user has set a new value, function `PutFloat()` puts the new value of a float parameter in the DP RAM. In the same way integer parameters can be set, but now a call to function `int_par()` is made, where function `PutInt()` is used.

6.2 Implementation of the Ward & Mellor transformations

The *Convert signals* transformation is implemented in function `c_int01()` in file *dspmain.c30.* The ANSI C compiler on the DSP supplies a special set of functions to implement interrupt routines. They are called `c_intxx()`, where `xx` can be a number between 00 and 99. These functions automatically save the context on the stack.

As is mentioned in section 3.1.2, 16 bits need to be shifted to the right when reading from the I/O cards. Therefore, the transformation is implemented as `ad_ptr[x] = *ADCx >> 16`, where `x` is an integer between zero and five.

The *Convert selected parameters* transformation is implemented in function `c_int01()` in file *dspmain.c30.* As is mentioned in section 3.1.2, 16 bits need to be shifted to the left when writing to the I/O cards. Therefore, the transformation is implemented as
`*ADCx = ((int)da_ptr[x]) << 16`, where `x` is an integer between zero and five.

The *Perform algorithm* transformation is also implemented in function `c_int01()`. The source code of the algorithm must be placed between the code for reading from the input channels and writing to the output channels. In the next chapter an example is given of a PID controller to show how an algorithm must be implemented.

The *Timer* transformation is implemented through two functions: `starttimer()` and `stoptimer()`. These functions are listed in file *userctrl.c30* and they start and stop timer 1 of the DSP chip.

The *Put hard disk data* transformation is implemented through five functions: `handler()`, `close_fifo()`, `flush_fifo()`, `open_fifo()` and `write_fifo()`. These functions are listed in file *pcmain.c.* A FIFO buffer is used to log data to the hard disk. Before logging to the hard disk is performed, verify and buffering are disabled by using functions `setverify()` and `setbuf()` in file *pcmain.c.* The FIFO buffer is well described in [BP 95]. In the next chapter is explained how data must be logged to the hard disk.

The *Process user commands* transformation is implemented on both the PC and the DSP. On the PC the transformation is implemented through the `if (kbhit())`-loop. In this loop the commands given by the user are checked. If a command like start, quit or reset is given, then a particular value representing the command is put at address `command_adr` in the DP RAM. On the DSP these commands are then processed in function `process_cmd()` in file *userctrl.c30*. This function makes calls to assembler functions `start_int()` and `stop_int()` in file *dsp.asm*. These assembler functions are responsible for starting and stopping the interrupts.

If the user wants to change the sample frequency a call is made to function `sample_freq()` in file *pcmain.c*. The new sample frequency is put at address `samp_freq_adr` in the DP RAM. Function `process_cmd()` in file *userctrl.c30* is then processed.

Other commands that can be given by the user are showing a help menu, setting parameters, scaling the x-axes for displaying signals and switching between text mode and graphic mode. When the user switches to graphic mode functions from file *grgui.c* are used.

The *Control tasks* transformation is implemented on both the PC and the DSP. The control commands that can be given are start, quit, halt and reset. As is mentioned above these commands are implemented through the `if (kbhit())`-loop on the PC and function `process_cmd()` on the DSP. Again calls to the assembler functions `start_int()` and `stop_int()` are made. Starting and stopping of the timer is done through functions `starttimer()` and `stoptimer()` as is described for the implementation of the *Timer* transformation.

6.3 Implementation of function main() (file *pcmain.c*)

Because of its large size, function `main()` of file *pcmain.c* is described in pseudo-code.

```
main()
{
  define variables;
  show opening window;
  load DSP code to board;
  SelectBoard(0x290);  /* select base address of DSP board */
  Reset();             /* reset DSP board */
  Hold();              /* hold DSP board */
  init_sett_par();     /* initialize settable parameters */
  init_data();         /* initialize all variables as defined in defs.h */
  get debug status from DP RAM;  /* is debug mode enabled ? */
  open user interface in text mode;
```

```
if (hard disk log is on) {

  check if file data.bin already exists;  /* data is logged to file
                                              data.bin */
  if (answer = 'yes') exit;
  else open_fifo();
}

if (debug mode is on) {

  check if debug parameters have to be read from disk;
  if (answer = 'yes') {

    read data from hard disk;
    place data in DP RAM;
  }
  if (answer = 'no') continue;
}
display initial sample frequency;
if (debug mode is off) start system;

while(1) {  /* main (endless) loop */

  if (error) {

    show error message;
    if (hard disk log is on) close_fifo();
    quit system;
  }
  display status ad and da channels (on/off);
  display status hard disk logging (on/off);
  display states;
  if (debug mode is on) get correct debug parameters from DP RAM;

  if (kbhit) {  /* check user commands */

    case 'h': help();         /* show help menu */
    case 'i': intpol = TRUE;  /* enable interpolation in graphic mode */
    case 'k': hold_disp = TRUE;  /* hold display in graphic mode */
    case 's': sample_freq();  /* set sample frequency */
    case 'l': graph = TRUE;   /* switch to graphic mode */
    case 'x': scale_x_axes(); /* scale x-axes in graphic mode */
    case 'g': start system;
    case 'q': quit system;
    case 'a': halt system;
    case 'r': reset system;
    case 'f': show_fifo_variables();
```

```
      case 'v': show_algorithm_variables();
      case 'p': float_par();     /* set float parameters */
      case 'n': int_par();       /* set integer parameters */
      case ' ': debug_mode();    /* perform algorithm one time */
    }
  }
}
```

6.4 Implementation of the debug mode

The idea behind debugging is to walk through the algorithm step by step to see if the algorithm produces the right output ("cold debugging"; see section 2.2). "Step by step" means that each time a key is pressed the algorithm is performed exactly one time with new values (debug inputs). These values are called *debug floats* and *debug integers* and are placed in the DP RAM.

The debug mode is based on the flow chart depicted in figure 6.2. The debug mode is implemented through function `debug_mode()` in file *pcmain.c*. The "switch" in figure 2.1 of section 2.2 is performed by defining a variable `debug_mode` in file *defs.h*. If this variable is defined the debug mode is active, if it is not the debug mode is non-active.

If the system is started in debug mode the DSP processor is halted. Debug floats and debug integers are set to zero. These parameters simulates the input channels and can be seen as virtual input ports. Because there is no need to perform the algorithm in real-time all debug parameters are placed in the DP RAM and not in the on-chip RAM.

A debug index indicates a particular address of a debug parameter in the DP RAM. The contents of each address is a value which the algorithm must perform in the next step. When the space bar is not pressed, the DSP processor will be halted. Otherwise, a flag is set to one by using the statement `PutInt (flag_adr, DUAL, 1)` and the DSP processor is released. Until the flag is set to zero on the DSP, the algorithm is performed with a new debug parameter. When the flag is set to zero at the end of the algorithm, the DSP is halted, the debug index is increased by one and the PC is waiting for the space bar to be pressed.

There are two ways of performing the debug mode. The first one is to place the debug parameters in the DP RAM by defining them in file *defs.h*. Here, they are defined as `*(debugflt_adr + x)` and `*(debugint_adr + x)`, where `x` indicates the debug index. The second one is to read an input signal from the I/O cards and to log it to the hard disk. The logged data is an exact representation of the input signal and represents the debug parameters. The data is then placed in the DP RAM by reading the data from the hard disk and using functions `WrBlkFlt (debugflt_adr, DUAL, max_debugflt, debugflt)` and `WrBlkInt (debugint_adr, DUAL, max_debugint, debugint)` in file *pcmain.c*. In this way a block of debug parameters is put in the DP RAM. The variables `max_debugflt` and `max_debugint` are defined in file *pcmain.h*. These variables indicate the number of debug parameters which must be written to the DP RAM. When the variables are read from the

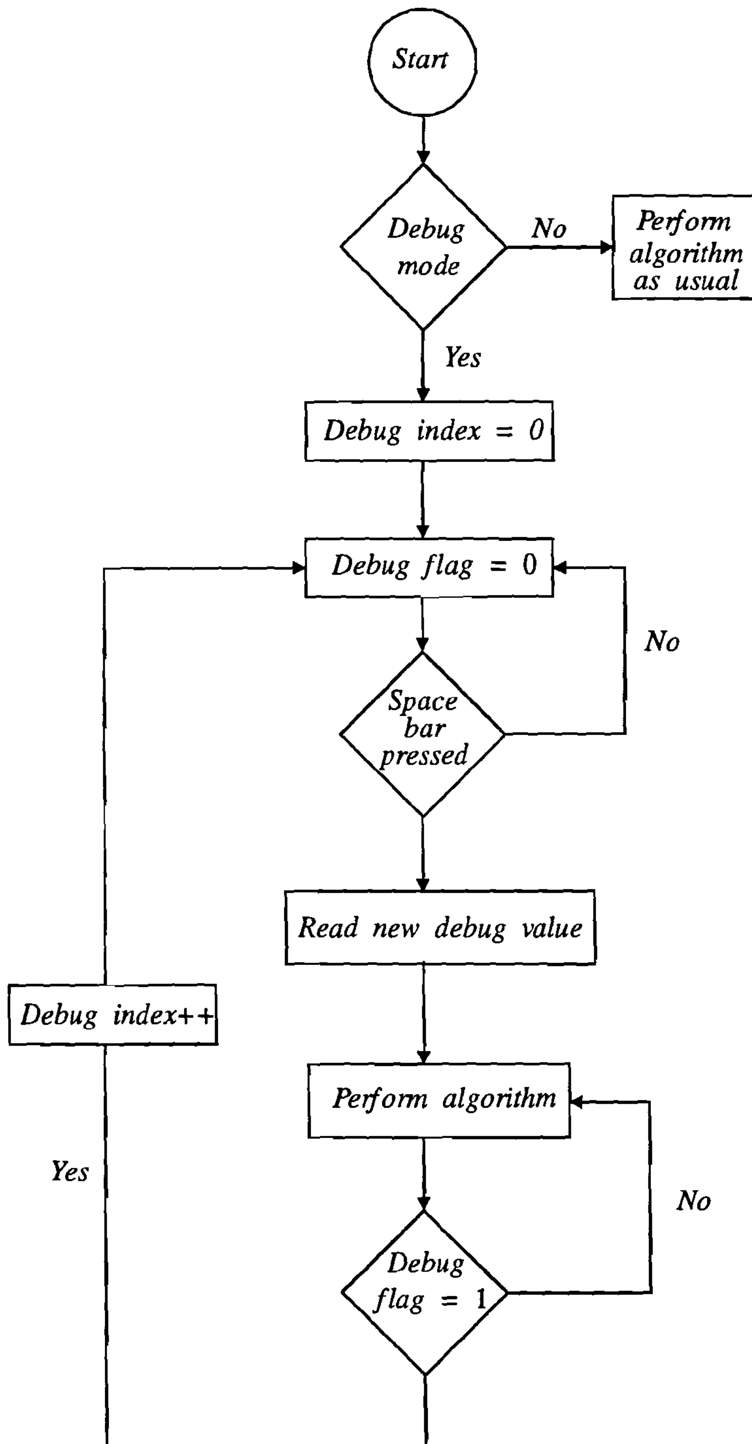


Figure 6.2: *Flow chart of debug mode.*

hard disk they are placed in arrays `debugflt` and `debugint` respectively. In the next chapter both ways are applied on a PID controller.

6.5 System's performance

Eight measurements are done to show the performance of the system:

1. exclusively read data from an input channel
2. exclusively write data to an output channel
3. read data from an input channel and immediately write data to an output channel
4. send data from DSP to PC (data throughput)
5. log data to hard disk
6. display data on screen
7. measure maximum sample frequency without logging
8. measure maximum sample frequency with logging

1. Exclusively read data from an input channel.

Data is exclusively read from one, two and six input channels without performing an algorithm or writing data to output channels. The number of clock-cycles on the DSP is measured. This number indicates the time needed for reading data from an input channel. Each clock-cycle is 60 nsec. The results are listed in table 6.1.

Table 6.1: *Results of measurement one*

# input channels	# clock-cycles	time (nsec)
1	14	840
2	24	1440
6	64	3840

2. Exclusively write data to an output channel.

Data is exclusively written to one, two and six output channels without performing an algorithm or reading data from input channels. The number of clock-cycles on the DSP is measured. This number indicates the time needed for writing data to an output channel. Each clock-cycle is 60 nsec. The results are listed in table 6.2.

Table 6.2: *Results of measurement two*

# output channels	# clock-cycles	time (nsec)
1	14	840
2	28	1680
6	76	4560

3. Read data from an input channel and immediately write data to an output channel.

Data is read from one, two and six input channels and the data is immediately written to output channels without performing an algorithm. The number of clock-cycles on the DSP is measured. This number indicates the time needed for reading data from an input channel and writing data to six output channel without the interference of an algorithm. Each clock-cycle is 60 nsec. The results are listed in table 6.3.

Table 6.3: *Results of measurement three*

# input channels	# clock-cycles	time (nsec)
1	86	5160
2	96	5760
6	136	8160

4. Send data from DSP to PC.

Data is read from an input channel and sent from DSP to PC without performing an algorithm. All measurements are performed with a sample frequency f_s of 50 KHz. The average number of cycles is measured as follows.

When de sample frequency f_s is x KHz then every $\frac{1}{x}$ milliseconds the number of cycles is increased by one. The initial number of cycles is zero. Each time 10000 I/O actions are performed. One I/O action stands for reading a 16 bit or a 32 bit word from the DP RAM (this depends on the kind of function that is used for reading data from the DP RAM). The performing of 10000 I/O actions is done 20 times. For each measurement the average number of cycles is calculated by adding all numbers of cycles and dividing the result by 20. Thus the time for performing one I/O action takes (# average cycles $\times \frac{1}{x}$)/10000 seconds.

Four measurements are done: one for `GetInt (cycles, DUAL)` with Int is a 16 bit integer, one for `GetInt (cycles, DUAL)` with Int is a 32 bit integer, one for `Get32Bit (cycles ,DUAL)` and one for `GetFloat (cycles, DUAL)`. The measurements are performed by exclusively reading the DP RAM, i.e. without reading the 128 Kw main RAM and the 2 Kw on-chip RAM. The results are listed in table 6.4. Each cycle is 60 nsec.

Table 6.4: *Results of measurement four*

instruction	# average cycles	time (μsec)	Kb/sec
GetInt (16 bit)	4412	8.8	221
GetInt (32 bit)	5246	10.5	186
Get32Bit	5357	10.7	182
GetFloat	6845	13.7	143

5. Log data to hard disk.

Normally data is written to the hard disk as text (ascii). In that case, each character is 8 bits. Therefore, data is logged binary to the hard disk to increase the performance. This means that each character is always 16 bits. Suppose i is an integer with $i = 1532$. If this integer is written to the hard disk as text this will cost 4 x 8 bits. If this integer is written to the hard disk as a binary value, this will only cost 16 bits.

The data throughput to the hard disk is measured by using the clock-cycles on the DSP. No use is made of the interrupt line from the DSP to the PC. The sample frequency is 10 KHz. No algorithm is performed during the measurement. During 10 seconds data is written to a file (*test.bin*) on the hard disk. In that case the file has a size of about 1.5 Mbyte. Thus the performance is about150 Kbyte/sec.

6. Display data on screen.

This measurement is performed by filling a totally black screen with 640*480 white pixels (the standard C function `putpixel()` is used) with no algorithm enabled. This takes 3.041 seconds, thus in one second 101019 pixels can be displayed on screen.

7. Measure maximum sample frequency without logging.

Data is read from the input channels and sent to the output channels without performing an algorithm and without logging to the hard disk. The results of the measurement of the maximum sample frequency are listed in table 6.5.

Table 6.5: *Results of measurement seven*

# input channels	# output channels	max f_s (KHz)
1	0	88
1	1	83
1	6	64
6	0	70
6	1	66
6	6	53

8. Measure maximum sample frequency with logging.

Data is read from the input channels and logged to the hard disk while performing a PID algorithm. The maximum sample frequency that can be used in this configuration is listed in table 6.6.

Table 6.6: *Results of measurement eight*

# input channels	# output channels	max f_s (KHz)
0	1	2.2
1	0	2.3
1	1	1.9
0	6	1.8
6	0	2.1
6	6	1.1

6.6 Summary

In this chapter the organization of data in the DP RAM and the on-chip RAM is described. Data is divided into three categories. Constants are parameters that don't change during the performing of the algorithm. Variables are parameters that can be set from the PC (settable parameters) and may change during the process. States are signals that vary each time instance.

States and non-settable parameters are copied from the on-chip RAM to the DP RAM. On the PC, these states and parameters are given correct names by using struct definitions and they are initialized on the DSP. Settable parameters are copied from the DP RAM to the on-chip RAM and are initialized on the PC by also using struct definitions.

Furthermore, the implementation of the Ward & Mellor transformations from the previous chapter is described. Some transformations are implemented on the DSP (*Convert signals*, *Convert selected parameters*, *Perform algorithm* and *Timer*), some on the PC (*Put hard disk data*) and some on both PC and DSP (*Process user commands* and *Control tasks*).

A debug mode is performed to walk through the algorithm step by step, i.e. each time a key is pressed the algorithm is performed exactly one time with new inputs. When the debug mode is enabled the input channels are replaced by virtual input channels. These virtual input channels are defined by debug parameters and can be used in two ways to debug the algorithm. The first one is to define the debug parameters in file *defs.h* on the DSP. The second one is to log data from an input channel to the hard disk in order to get an exact representation of the input signal. The data is put in the DP RAM.

Finally, eight measurements are done to get an impression of the system's performance.

Chapter 7

Example: a simple PID controller

7.1 Description of the PID controller

The PID control algorithm has the general form

$$m(t) = K_p[e(t) + \frac{1}{T_i}\int_0^t e(t)dt + T_d\frac{de(t)}{dt}] \tag{7.1}$$

where $e(t) = r(t) - c(t)$ and $c(t)$ is the measured variable, $r(t)$ is the reference value and $e(t)$ is the error. K_p is the overall controller gain, T_i is the integral action time and T_d is the derivative action time.

The ratio between the control signal and the error signal can be adjusted using the proportionality constant K_p. Choosing the value of K_p involves a compromise: a high value of K_p gives a small steady-state error and a fast response, but the response will be oscillatory and may be unacceptable in many applications. A low value gives a slow response and a large steady-state error. By adding the integral action term the steady-state error can be reduced to zero since the integral term integrates the error signal with respect to time. For a given error value the rate at which the integral term increases is determined by the integral action time T_i. The major advantage of incorporating an integral term arises from the fact that it compensates for steady-state disturbances that occur in the process being controlled (zero steady-state error for constant reference signals).

For a few processes which are subjected to sudden disturbances the addition of the derivative term can give improved performance. Because derivative action produces a control signal that is related to the rate of change of the error signal, it anticipates the error and hence acts to reduce the error that would otherwise arise from the disturbance.

The differential equation 7.1 is the time domain representation of the controller. The equiv-

alent frequency domain representation is

$$G_c(s) = \frac{M(s)}{E(s)} = K_p\left(1 + \frac{1}{T_i s} + T_d s\right) \tag{7.2}$$

In the frequency domain the overall system of controller and plant can be represented by a block diagram as is depicted in figure 7.1. Both the time domain and frequency domain representations are continuous representations. To implement the controller using a digital algorithm a conversion has to be done from a continuous to a discrete representation of the controller.

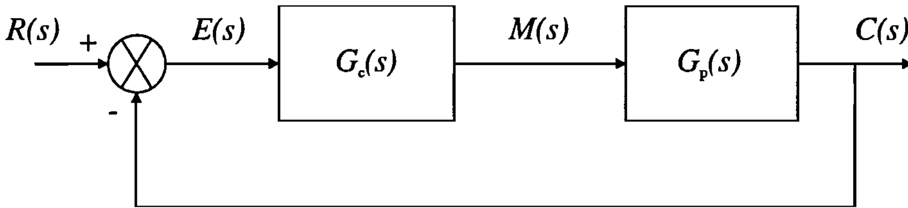


Figure 7.1: *General form of a control system in continuous form.*

There are several methods of doing this. The simplest is to use first-order finite differences. Considering the time domain version of the controller (eq. 7.1) the differential and integral terms are replaced by their discrete equivalents by using the relationships

$$\frac{df}{dt}|_k = \frac{f_k - f_{k-1}}{\Delta t}, \quad \int e(t)dt = \sum_{k=1}^{n} e_k \Delta t \tag{7.3}$$

and hence equation 7.1 becomes

$$m(n) = K_p\left[T_d\left(\frac{e(n) - e(n-1)}{\Delta t}\right) + e(n) + \frac{1}{T_i}\sum_{k=1}^{n} e_k \Delta t\right] \tag{7.4}$$

where $m(n)$ represents the value of m at some time interval $n\Delta t$ where n is an integer.

By introducing new parameters as follows:

$$\begin{aligned} K_i &= K_p(\Delta t/T_i) \\ K_d &= K_p(T_d/\Delta t) \end{aligned}$$

where $T_s = \Delta t =$ the sampling interval, equation 7.4 can be expressed as an algorithm of the form

$$\begin{aligned} s(n) &= s(n-1) + e(n) \\ m(n) &= K_p e(n) + K_i s(n) + K_d[e(n) - e(n-1)] \end{aligned} \tag{7.5}$$

where $s(n) =$ the sum of the errors taken over the interval 0 to nT_s.

7.2 Implementation of the basic PID algorithm

Writing the code to implement the algorithm given in equation 7.5 is a simple job. The basic code statements for the ideal PID controller are:

```
sn = sn + en;
mn = Kp * en + Ki * sn + Kd * (en - enOld);
enOld = en;
```

However, in practical applications the value of the manipulated variable $m(n)$ is limited by physical constraints. A regulator with integral action is an input-output unstable system. Its unstable mode can give rise to difficulties under certain circumstances. *Integrator windup* can occur if the output saturates and the controller continues to integrate the error. The output of the integrator can then assume very large values and it can take a long time to get it back to a normal value again. This problem is automatically avoided when *stop summation* is used. The stop summation technique gives a better response if the integral term is unfrozen once the sign of the error changes. The sign of the error will change before the output comes out of saturation. The code statements for the stop summation technique is then given by:

```
StopSummation = ((mn > mnmax)&&(en > 0.0))||((mn < mnmin)&&(en < 0.0));
if(!(StopSummation)) sn += en;
mn = Kp * en + Ki * sn + Kd * (en - enOld);
enOld = en;
```

The implementation of the PID controller with the stop summation technique is described in the next section.

7.3 Implementation of the PID controller

7.3.1 Plain implementation

In this section the plain implementation of the PID controller from the previous section is described. "Plain" means without debugging and without logging to the hard disk. The block diagram of the control system is depicted in figure 7.2. A square wave with a magnitude of 1 V is used as input signal.

1. Place the source code of the algorithm in function `c_int01()` in file *dspmain.c30* between the `#ifndef debug_mode` and `#endif` statements. The debug mode must be disabled by placing variable `debug_mode` in file *defs.h* between comment statements.

File *dspmain.c30*:

```
/***************************************************
 * your algorithm source code should be placed here *
 ***************************************************/

#ifndef debug_mode
*debug_status_adr = 0;
...
#endif
```

File *defs.h*:

```
/* Definition of debug mode. */

/*
#define debug_mode ON
*/
```

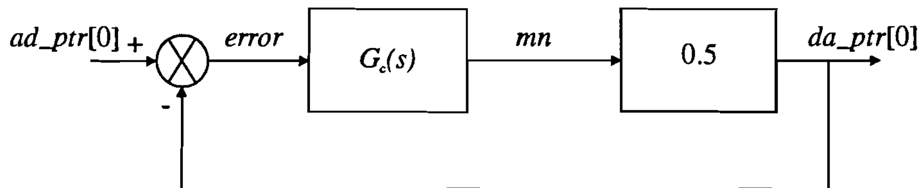


Figure 7.2: *General form of a control system in continuous form.*

2. Begin the algorithm with `*debug_status_adr = 0`(see 1). In this way the PC knows whether or not the system must operate in debug mode.
3. Define the variables which must be displayed "non-stop" on screen as states (floats or integers) in file *defs.h*. In the case of the PID algorithm these variables are `mn`, `sum`, `error` and `error_old` and they are defined as float states by using `statesblock_ptr[x]`. File *defs.h*:

```
/****************
 * float states *
 ****************/

#define mn         statesblock_ptr[0]
#define sum        statesblock_ptr[1]
#define error      statesblock_ptr[2]
#define error_old  statesblock_ptr[3]
...
```

4. Initialize these states in function `Init_vars()` in file *dspmain.c30*.
   File *dspmain.c30*:

```
/****************
 * float states *
 ****************/

mn        = 0.000;
sum       = 0.000;
error     = 0.000;
error_old = 0.000;
```

5. Give the float states correct names on the PC by initializing the float states structure in function `init_data()` in file *pcmain.c*. After the last name a "-" sign must be given, because otherwise garbage will be displayed on screen.
   File *pcmain.c*:

```
/****************
 * float states *
 ****************/

fltstatesarray[0].name_ptr = strdup("mn");   fltstatesarray[0].index = 0;
fltstatesarray[1].name_ptr = strdup("sum");  fltstatesarray[1].index = 1;
fltstatesarray[2].name_ptr = strdup("err");  fltstatesarray[2].index = 2;
fltstatesarray[3].name_ptr = strdup("eold"); fltstatesarray[3].index = 3;
fltstatesarray[4].name_ptr = strdup("-");    fltstatesarray[4].index = 4;
...
```

6. Define the variables that not have to be displayed non-stop on screen as parameters (floats or integers) in file *defs.h*. In the case of the PID algorithm these parameters are `mn_min` and `mn_max` and they are defined as integer parameters by using `pariblock_ptr[x]`.
   File *defs.h*:

```
/**********************
 * integer parameters *
 **********************/

#define mn_min pariblock_ptr[0]
#define mn_max pariblock_ptr[1]
...
```

7. Initialize these parameters in function `Init_vars()`in file *dspmain.c30*.
   File *dspmain.c30*:

```
/**********************
 * integer parameters *
 **********************/

mn_min = -32768;  /* minimum of -3 volt */
mn_max = 32768;   /* maximum of 3 volt  */
```

8. Give the integer parameters correct names on the PC by initializing the integer parameters structure in function `init_data()` in file *pcmain.c*. After the last name a "-" sign must be given, because otherwise garbage will be displayed on screen.
   File *pcmain.c*:

```
/**********************
 * integer parameters *
 **********************/

intpararray[0].name_ptr = strdup("mn_min");  intpararray[0].index = 0;
intpararray[1].name_ptr = strdup("mn_max");  intpararray[1].index = 1;
intpararray[2].name_ptr = strdup("-");       intpararray[2].index = 2;
...
```

9. Define the variables that must be set by the user as settable parameters(floats or integers) in file *defs.h*. In the case of the PID algorithm these parameters are `kp`, `ki` and `kd` and they are defined as settable float parameters by using `setparblock_ptr[x]`.
   File *defs.h*:

```
/*****************************
 * settable float parameters *
 *****************************/

#define kp setparblock_ptr[0]
#define ki setparblock_ptr[1]
#define kd setparblock_ptr[2]
...
```

10. Give the settable float parameters correct names on the PC by initializing the settable float parameters structure in function `init_data()`in file *pcmain.c*. After the last name a "-" sign must be given, because otherwise garbage will be displayed on screen.
    File *pcmain.c*:

```
/*****************************
 * settable float parameters *
 *****************************/

setfltpararray[0].setname_ptr = strdup("kp");  setfltpararray[0].index = 0;
setfltpararray[1].setname_ptr = strdup("ki");  setfltpararray[1].index = 1;
setfltpararray[2].setname_ptr = strdup("kd");  setfltpararray[2].index = 2;
setfltpararray[3].setname_ptr = strdup("-");   setfltpararray[3].index = 3;
...
```

11. Give the settable float parameters correct initial values in function `init_sett_par()` in file *pcmain.c*.
    File *pcmain.c*:

```
/*************************************
 * initial settable float parameters *
 *************************************/

setfltpararray[0].value = 1.000000;
setfltpararray[1].value = 0.000000;
setfltpararray[2].value = 0.000000;
...
```

12. In order to give the states and parameters their initial values after a reset, they must also be defined in function `reset()` in file *userctrl.c30*.

13. Because one input channel (`ad_ptr[0]`) and two output channels (`da_ptr[0]` and `da_ptr[1]`) are used `ad_0`, `da_0` and `da_1` in file *defs.h* must be defined, while the other channels must be defined as comment.

```
/* Definitions of the ad and da channels that are to be used. */

#ifndef debug_mode
#define ad_0   TRUE
/*
#define ad_1   TRUE
#define ad_2   TRUE
#define ad_3   TRUE
#define ad_4   TRUE
```

```
#define ad_5   TRUE
*/

#define da_0   TRUE
#define da_1   TRUE
/*
#define da_2   TRUE
#define da_3   TRUE
#define da_4   TRUE
#define da_5   TRUE
*/
#endif
```

14. If input channels four or five must be displayed in graphic mode the variables `signal_1` through `signal_4` in file *pcmain.h* must be changed. If, for example, input channel five in stead of input channel two must be displayed then file *pcmain.h* changes in:

```
/* Definitions of signals that have to be displayed. */

#define signal_1  0  /* display input channel 0 */
#define signal_2  5  /* display input channel 5 */
#define signal_3  2  /* display input channel 2 */
#define signal_4  3  /* display input channel 3 */
```

15. Define variable `HDLOG` in file *pcmain.h* as FALSE.

16. Variable `dsp_int_time` in file *pcmain.c* returns the number of cycles on the DSP. One cycle is 60 nsec. This variable is automatically displayed on screen (text mode) and gives an indication of the performance of the algorithm (including reading and writing from and to the I/O channels).

7.3.2 Implementation of the PID controller using HD logging

The only difference with the plain implementation from the previous section is that variable `HDLOG` in file *pcmain.h* is now to be defined as TRUE. Suppose the first two output signals have to be logged to the hard disk. Variables `log_da0` and `log_da1` in file *pcmain.h* must then be defined, while the other channels have to be defined as comment by placing their definitions between comment statements.

In file *dspmain.c30* state `mn` is written to the first output channel (`da_ptr[0] = mn`) and state `error` is written to the second output channel (`da_ptr[1] = error`). In this way states can be logged, because these two output channels are logged to the hard disk by defining `log_da0` and `log_da1` in file *pcmain.h* (see above).

Data is written as binary values to the hard disk (see section 6.5). The system must be told in what format data must be written to file *data.bin*. The available formats are floats, integers, long integers and characters. The formats are defined in file *data.frm*, where an *f* represents a float, an *i* represents an integer, an *l* represents a long integer and a *c* represents a character. Table 7.1 shows how many *i*'s file *data.frm* must contain in order to log data correctly to the hard disk (data from the I/O cards is read as integers).

Table 7.1: *Relationship between # input channels, # output channels and # i's*

# ad channels	# da channels	# i's
0	1	2
0	2	4
0	3	6
0	4	8 (max.)
1	0	1
1	1	3
1	2	5
1	3	7
1	4	9 (max.)
2	0	2
2	1	4
2	2	6
2	3	8 (max.)
3	0	3
3	1	5
3	2	7
3	3	9 (max.)
4	0	4
5	0	5
6	0	6

Thus if one input signal must be logged to the hard disk, *data.frm* must contain one *i* and if three input channels and two output channels have to be logged, *data.frm* must contain seven *i*'s (if the signal consists of integers).

Data is written to file *data.bin*. When the log procedure has been finished this file contains binary values. By typing *convert data.bin*, data of *data.bin* is converted into file *data.log* (in the DOS shell). The contents of this file can be read by simply typing *read data.log.*

The steps that must be taken can be summarized as follows:

1. Define variable `HDLOG` in file *pcmain.h* as TRUE.
2. Define the signals which have to be logged to the hard disk in file *pcmain.h.*

3. Follow steps 1 through 16 (except step 15) of the previous section.
4. Modify file *data.frm* as is described above.
5. Run file *pcmain.c.*
6. Quit the system if enough data is logged to the hard disk.
7. Go to the DOS shell.
8. Convert file *data.bin* into file *data.log* by typing *convert data.bin.*
9. Read the contents of file *data.log* by typing *read data.log.*

Algorithm results

Data from three channels (`ad_ptr[0]`, `da_ptr[0]` and `da_ptr[1]`) are logged to the hard disk. Four measurements are done:

1. K_p = 1.0, K_i = 0.0 and K_d = 0.0
2. K_p = 1.0, K_i = 10000 and K_d = 0.0
3. K_p = 1.0, K_i = 0.0 and K_d = 0.001
4. K_p = -0.5, K_i = 5000 and K_d = 0.001

During the measurements is f_s = 1 KHz and the input signal is a square wave with a frequency of 50 Hz and a magnitude of 1 V. The data is implemented in Matlab and the results are listed in figures 7.3 through 7.10. The dashed signal is the input channel (square wave). Only two periods (40 data points) are implemented.

Performing of the algorithm without reading or writing from or to the I/O channels takes 10.6 μsec. When the I/O channels are used the algorithm can be performed in 12.5 μsec.

7.3.3 Implementation of the PID controller using the debug mode

As was mentioned in section 6.4 debugging can be done in two ways.

1. Debugging by defining debug inputs in file *defs.h*

The debug mode is enabled by defining `debug_mode` in file *defs.h.* The ADC has a full operational range of +/- 3.0 V (see appendix A). This means that data points may have a value between -2^{15} and 2^{15}, where -2^{15} (-32768) is -3.0 V and 2^{15} (32768) is 3.0 V. A square wave with a frequency of 50 Hz and a magnitude of 1 V can be simulated by 20 debug integer parameters: 10 integers have a value of 10923 (1.0 V) and 10 integers have a value of -10923 (-1.0 V). These parameters are defined in file *defs.h* and are initialized in file *dspmain.c30* and in function `reset()` in file *userctrl.c30.*

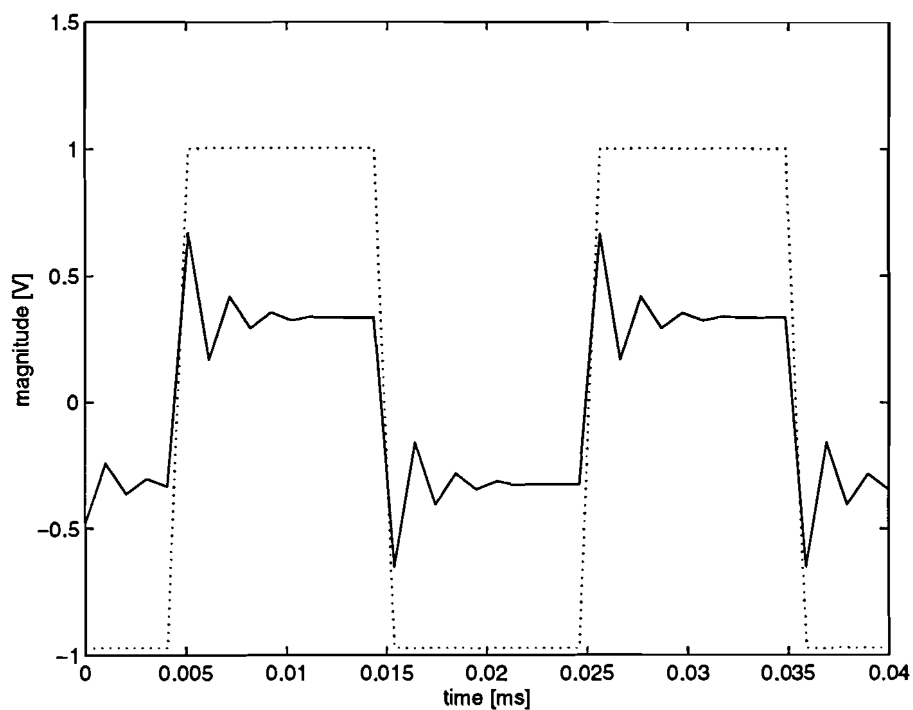


Figure 7.3: *Input signal versus mn (measurement 1).*

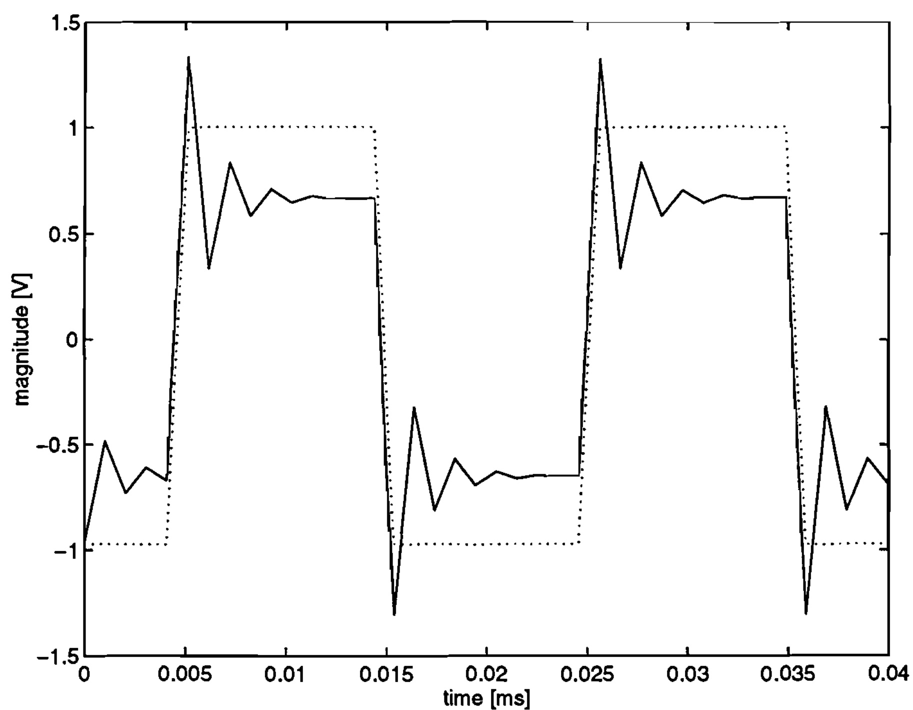


Figure 7.4: *Input signal versus error (measurement 1).*

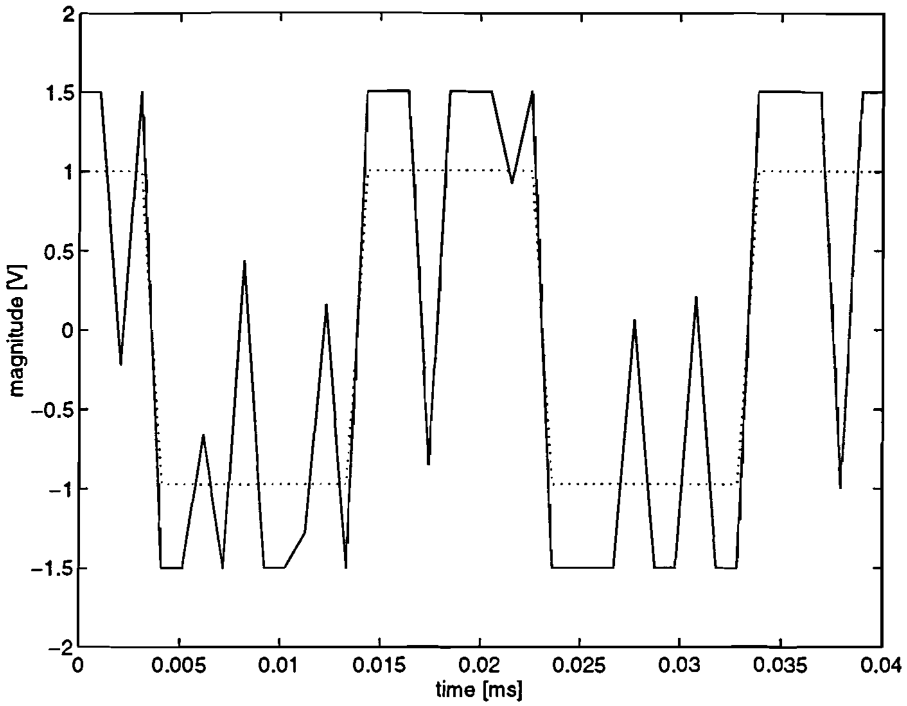


Figure 7.5: *Input signal versus mn (measurement 2).*

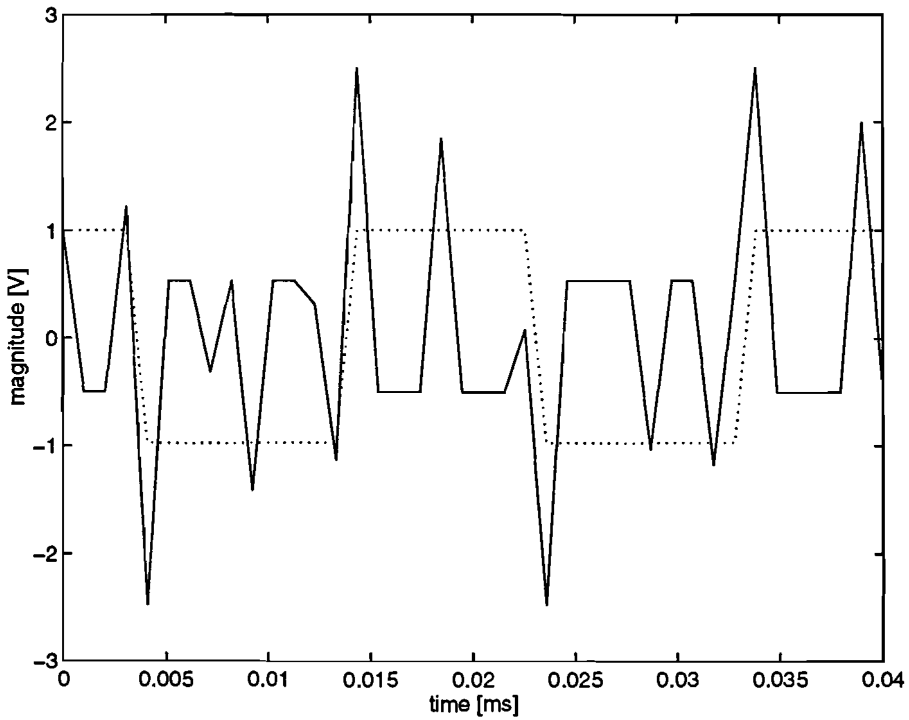


Figure 7.6: *Input signal versus error (measurement 2).*

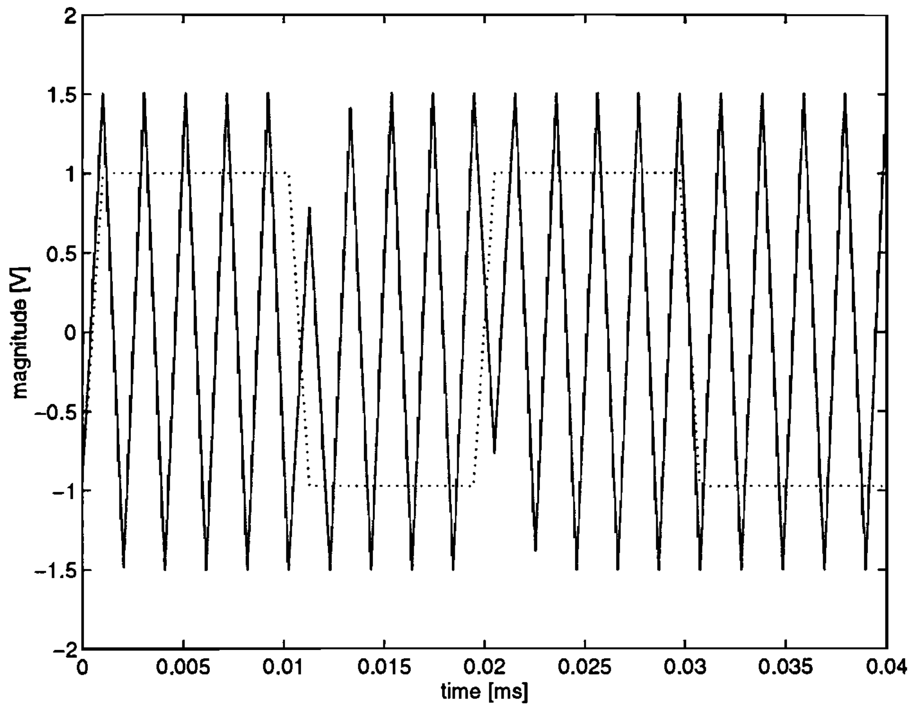


Figure 7.7: *Input signal versus mn (measurement 3).*

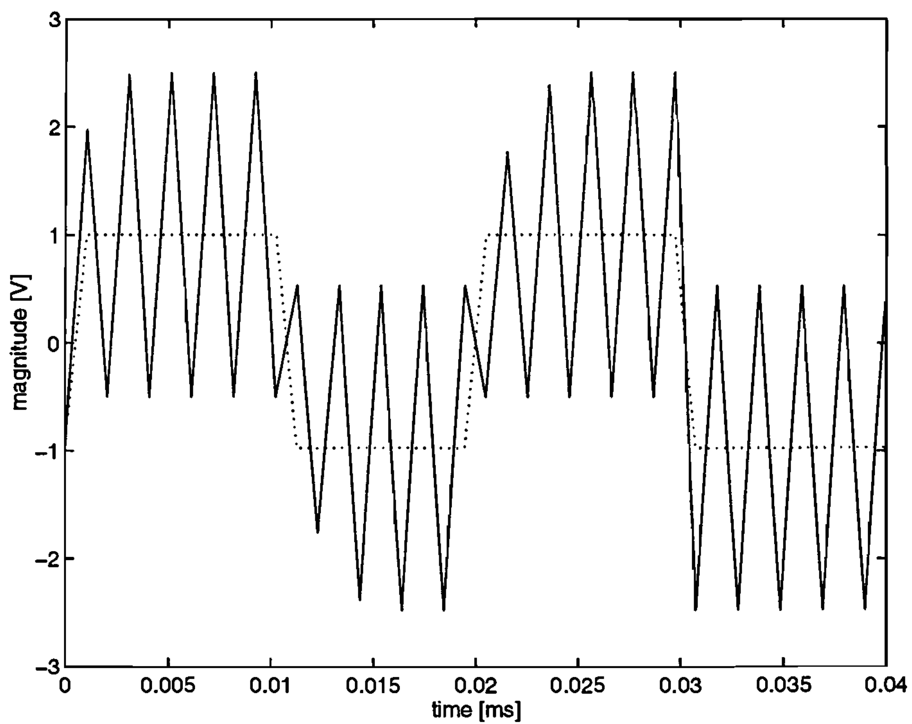


Figure 7.8: *Input signal versus error (measurement 3).*

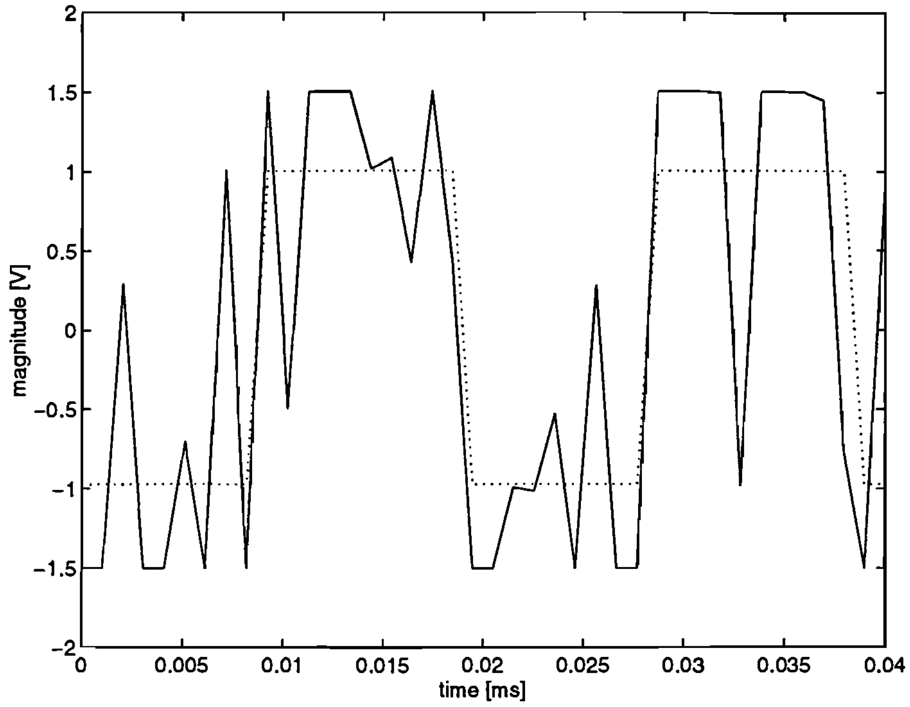


Figure 7.9: *Input signal versus mn (measurement 4).*

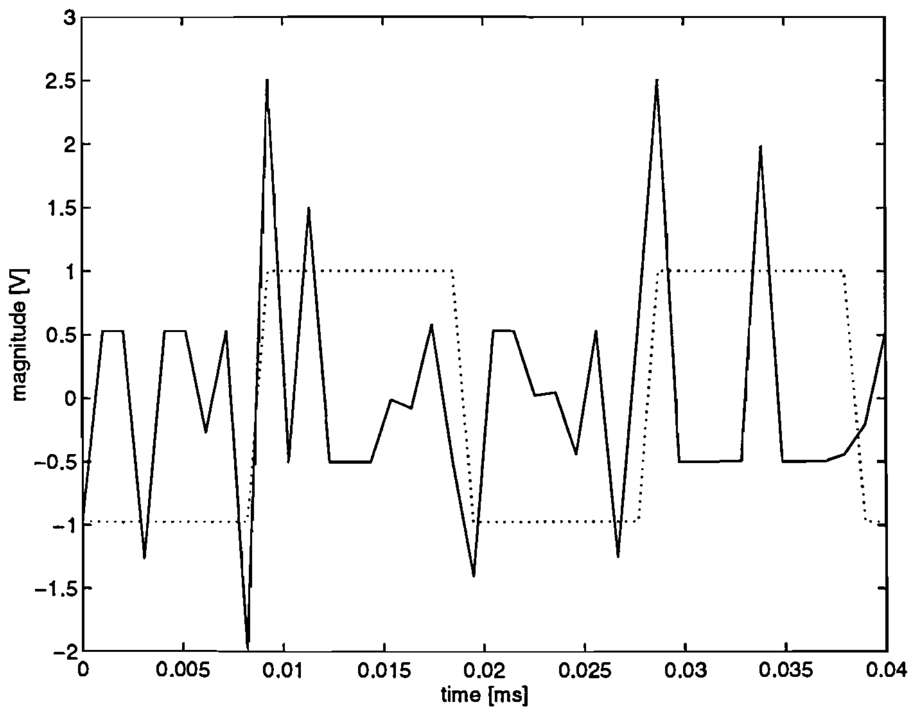


Figure 7.10: *Input signal versus error (measurement 4).*

Because the input channels are now replaced by virtual input channels (simulated by the debug integer parameters), the variable `error` of the algorithm must be calculated with the debug integer parameters. Every time the space bar is pressed the debug index is increased by one and the next debug parameter is read from the DP RAM. Remark: in the present configuration 26 debug float parameters and 26 debug integer parameters can be defined. If this amount appears to be too small, more definitions must be added in *defs.h*.

The steps described in section 7.3.1 must be followed to implement this debug option. The source code used for the algorithm in section 7.3.1 can also be used for the implementation of the algorithm in debug mode. However, there are a few differences that must be taken into account:

1. Place the source code of the algorithm in function `c_int01()` in file *dspmain.c30* between the `#ifdef debug_mode` and `#endif` statements. The debug mode must be enabled by uncommenting variable `debug_mode` in file *defs.h*.
   File *dspmain.c30*:

```
/***************************************************
 * your algorithm source code should be placed here *
 ***************************************************/

#ifdef debug_mode
*debug_status_adr = 0;
...
*flag_adr = 0;
*ack_adr  = 1;
#endif
```

   File *defs.h*:

```
/* Definition of debug mode. */

#define debug_mode  ON
```

2. Begin the algorithm with `*debug_status_adr = 1` and end it with `*flag_adr = 0` and `*ack_adr = 1` (see above). In this way the PC knows that the system must operate in debug mode and that the DSP processor must be halted when the algorithm is performed once.

3. Define the variables which must simulate the input signal(s) as debug parameters in file *defs.h*. In the case of the PID algorithm these variables are `ad0` through `ad19` (20 data points represent one period if f_s is 1000 Hz and the frequency of the function generator is 50 Hz) and they are defined as debug integers by using `*(debugint_adr + x)` where `x` is an integer between 0 and 19.

File *defs.h*:

```
/**********************************
 * integer parameters in debug mode *
 **********************************/

#define ad0  *debugint_adr
#define ad1  *(debugint_adr + 1)
#define ad2  *(debugint_adr + 2)
...
#define ad19  *(debugint_adr + 19)
...
```

4. Variable `ad_ptr[0]` used in the algorithm in section 7.3.1 must be replaced by `*(debugint_adr + (*debug_index_adr))`. Variable `debug_index_adr` is increased each time the space bar is pressed.
   Without debugging (file *dspmain.c30*):

```
...
error = ad_ptr[0] - da_ptr[0];
...
```

   With debugging (file *dspmain.c30*):

```
...
error = (*(debugint_adr + (*debug_index_adr))) - da_ptr[0];
...
```

2. Debugging by logging an input signal to hard disk

In the previous option the user simulates an input signal by defining correct values for the debug parameters. For a square wave this is not a problem (the only problem is the large amount of debug parameters that have to be defined when a small sample frequency is used), but it is a problem for signals that are difficult to simulate. For this kind of signals *adc.exe* reads an input signal from the I/O cards and writes data to file *ad_data.bin*. However, before executing *adc.exe* the type of format must be defined as is explained in the previous option (use *ad_data.frm*). At the end of conversion file *ad_data.bin* needs to be converted into file *ad_data.log* by using *convert ad_data.bin*.

When *pcmain.c* is run in debug mode file *ad_data.log* is read. Data from this file is then placed at address `debugflt_adr` or `debugint_adr` in the DP RAM by using functions `WrBlkFlt()` and `WrBlkInt()` (see section 6.4).

File *pcmain.h* contains definitions for the maximum debug float parameters and debug integer parameters. The values of `max_debugflt` and `max_debugint` depend on the amount of data

points which are needed for one period of the input signal (see previous section). Remark: if this option is to be used, the definitions of the debug parameters in *defsh.h* have to be placed between comment signs.

The steps that must be taken can be summarized as follows:

1. Remove the debug parameters in file *defs.h* as comment.
2. Follow step one, two and four of the previous option.
3. Define the maximum number of debug parameters in file *pcmain.h*. For example, if f_s is 1000 Hz and the frequency of the function generator is 50 Hz then one period of a signal consists of 20 data points. Thus if the debug mode must be performed with two periods, the maximum number of debug parameters (`max_debugflt` and `max_debugint`) must be 40.
4. Modify file *ad_data.frm* as is described in section 7.3.2 (use table 7.1).
5. Use *adc.exe* to log data to the hard disk. Convert *ad_data.bin* by typing *convert ad_data.bin* (in the DOS shell).
6. Run file *pcmain.c*.

7.4 Summary

An algorithm for an ideal PID controller is obtained. However, in practical applications integrator windup can occur. Therefore, a stop summation technique is presented to solve the problem of reset windup or integrator saturation. This technique is used in a PID algorithm to show the usage of the real-time environment. The algorithm is performed in three ways: plain, (i.e. without debugging and without logging to hard disk), with logging to hard disk and with debugging.

The algorithm can be performed in 10.6 μsec if the I/O channels are not used. If data is read from and written to the I/O channels, performing of the algorithm takes 12.5 μsec. Data which is logged to the hard disk is implemented in Matlab. The debug mode is used in two ways: debugging by defining debug inputs in file *defs.h* and debugging by logging an input signal to the hard disk.

Chapter 8

Conclusions and recommendations

8.1 Conclusions

A real-time environment has been designed and realized on a DSP-PC system. Through the environment the user is able to perform his algorithm on the system. The environment is capable of reading and writing from or to I/O cards, logging data to the hard disk, displaying data and signals on screen and setting parameters. Furthermore, a simple debug facility has been performed in order to perform the algorithm one time by pressing a key.

The real-time environment has been performed in three stages: a literature study (to get some knowledge of real-time systems), software design and software implementation (realization). The conclusions made during those steps can be summarized as follows.

- A literature study has been done on real-time systems. Several definitions of real-time systems can be found in literature, but concerning the content they are identical. A real-time system can be defined as a system that must produce correct responses within a definite time limit, because otherwise performance or malfunction results. At this point a discussion could be made whether or not the environment runs in real time or not. Tasks that run on the DSP have hard time constraints and therefore the real-time environment is a hard real-time environment. However, tasks that run on the PC (except logging to the hard disk) have soft time constraints and in that case the environment is a soft real-time environment.

- Most aspects and problems concerning real-time systems are not applied to the DSP-PC system. However, it is useful to divide the tasks that must run on the system into two categories. Tasks with hard time constraints must run in the foreground, while tasks with soft time constraints or no time constraints must run in the background. Background tasks must run when no interrupts are being processed, while foreground tasks must be executed within a short time of the occurence of the interrupt. Therefore,

background tasks must run on the PC and foreground tasks must run on the DSP. In this way a clear distinction between tasks with different priorities is obtained.

- The Ward & Mellor method appears to be a structured development method for real-time systems. Through this method the software design for the real-time environment has been graphically described. By dividing the context scheme into two transformations a distinction between the DSP and the PC has been made.

- The implementation of the real-time environment has been performed in the C language. Data acquisition, processing of user commands and performing of the algorithm are implemented on the DSP. States and non-settable parameters are initialized on the DSP by copying their values from the on-chip RAM to the DP RAM. Commands that can be given by the user are processed on the PC. Settable parameters are initialized on the PC by placing their values in the DP RAM. These parameters are then copied from the DP RAM to the on-chip RAM.

- A PID controller has been implemented to illustrate the usage of the real-time environment. If no data is read from or written to the I/O cards the performing of the PID algorithm takes 10.6 μsec. Performing the algorithm while using the I/O channels takes 12.5 μsec (i.e. real time).

8.2 Recommendations

Although the system works properly most times, there are some problems that arise in particular cases.

- The maximum frequency of a signal which can be displayed in graphic mode is about 20 Hz. When the sample frequency increases disturbances arise in the signal which is displayed on screen, because the whole screen is updated every sample. This problem may be solved by using the implementation of updating the screen as is done in MACS. Here, only the new value is put on screen, while the rest of the screen remains unchanged.

- As is clear from the performance measurements, logging to the hard disk results in a degradation of the system's performance. Because there is enough space left in the DP RAM it is recommended to place the outputs of the algorithm in the DP RAM before logging to the hard disk. If enough data is stored in the DP RAM the data-acquisition can be stopped and logging to the hard disk can be started. In this way, logging to the hard disk becomes a background task.

- Resetting the system sometimes leads to an incorrect initial state of the system. It is assumed that some functions in the assembler routine (*dsp.asm*) does not work properly.

- Although the debug facility can be used to perform the algorithm one time ("step by step"), it may be implemented more efficiently. In the present configuration the algorithm needs to be modified when using the debug mode. This can be avoided by defining six addresses in the DP RAM representing variables `ad_ptr[0]` through `ad_ptr[5]`. Each time the space bar is pressed six values must be read from a file. These

values represent the six inputs and must be copied to the six addresses in the DP RAM. In this way, the algorithm needs not to be modified, but the variables `ad_ptr[0]` through `ad_ptr[5]` can still be used. In debug mode the algorithm then reads the inputs from the DP RAM.

Additionally, some further work can be done:

- Adapt the SIMULINK Real-Time Workshop software for the DSP-PC system. The Real-Time Workshop is an automatic C language code generation environment for SIMULINK. It produces C code directly from SIMULINK graphical models and automatically builds programs that can be run in real-time in a variety of environments. The real-time code can be compiled on the PC using the TMS320C30 cross compiler. The object file is then downloaded to the DSP board, which is manufactured by dSPACE. It executes on the DSP and interfaces with external hardware via the A/D and D/A converters installed on the board. Thus both DSP boards use the TMS320C30 chip, but they are produced by different manufacturers. In the case of the LSI board the I/O ports are external to the board while the dSPACE board has on-board I/O ports.
- Use the system in conjunction with SPOX. SPOX is a DSP operating system specifically designed to meet the needs of high-end DSP microprocessors. The capabilities provided by SPOX fall into four broad categories: real-time multi- tasking, I/O and communications, a C standard library and DSP application functions.
- At the time this report was written a digital interface circuit was built. The working of this circuit must be tested. If the circuit works properly software must be written to read and write data from and to the circuit when using the real-time environment.

Bibliography

[AT 90] Auslander, D.M. and C.H. Tham
Real-time Software for Control: Program Examples in C
Prentice-Hall, Inc., 1990.

[Ben 94] Bennett, S.
Real-time Computer Control, an Introduction 2nd ed.
London: Prentice Hall International (UK) Limited, 1994.

[BP 95] Brands, J.W.M. and M.J. Peerenboom
Het Ontwerp en Implementatie van een Besturing voor Real-Time Data-acquisitie Hardware
HTS afstudeerverslag, 1995.

[Coo 91] Cooling, J.E.
Software Design for Real-Time Systems
London: Chapman & Hall, 1991.

[DIN 85] *Informationsverarbeitung*
October 1985.

[FP 88] Faulk, S.R. and D.L. Parnas
On Synchronization in Hard-Real-Time Systems
Communications of the ACM, Vol. 31 (1988), No. 3, p. 274 - 287.

[HH 91] Hoogeboom, B. and W.A. Halang
The Concept of Time in Software Engineering for Real Time Systems
In: Software Engineering for Real time Systems. Third International Conference, Cirencester (UK), 16 - 18 Sept. 1991.
London: IEE Conference Publication, No. 344, p. 156 - 163.

[JG 89] Joseph, M. and A. Goswami
Formal Description of Realtime Systems: a Review
Information and Software Technology, Vol. 31 (1989), No.2, p. 67 - 76.

[KKZ 88] Koymans, R. and R. Kuiper, E. Zijlstra
Paradigms for Real-Time Systems
In: Formal Techniques in Real-time and Fault-Tolerant Systems. Proc. of a Symposium, Warwick (UK), 22 - 23 September 1988. Ed. by M. Joseph.
Berlin: Springer, 1988. No. 331, p. 159 - 174.

[Lis 79] Lister, A.M.
Fundamentals of Operating Systems 2nd. ed.
London: Macmillan, 1979.

[Mes 94] Messmer, H.P.
The Indispensable PC Hardware Book
Addison-Wesley, 1994.

[NAG 93] Novitsky, J. and M. Azimi, R. Ghaznavi
Optimizing Systems Performance Based on Pentium Processors
COMPCON Spring '93, p. 63 - 72.
IEEE Comput. Soc. Press.

[PB 91] Loughborough Sound Images Ltd.
TMS320C30 Processor Board User Guide
Version 1.01, 1991.

[Phi 89] Philips, B.
Pick The Right Bus for PC-Based Data Acquisition
Electronic Design, April 27, 1989, p. 85 - 94.

[She 91] Shepard, T. and J.A.M. Gagné
A Pre-run-time Scheduling Algorithm for Hard Real-Time Systems
IEEE Transactions on Software Engineering, Vol. 17 (1991), No. 7, p. 669 - 677.

[SIB 91] Loughborough Sound Images Ltd.
16 Bit Stereo Interface Board User Manual
Issue 3, 1991.

[Sja 90] Sjauw En Wa, P.
Het PC Handboek, Vraagbaak voor PC Apparatuur
Amsterdam: Addison-Wesley Nederland, 1990.

[TI 89] Texas Instruments
Third-generation TMS320 User's Guide
1989.

[Ver 90] Vermeir, J.P.
Methodisch Ontwerpen van Programmatuur voor Real-Time Besturingssytemen
Afstudeerverslag, TUE, Vakgroep Meten en Regelen, 15 oktober 1990.

[WM 85] Ward, P.T. and S.J. Mellor
Structured Development for Real-Time Systems, Volume 1: Introduction & Tools
New York: Yourdon Inc., 1985.

Appendix A

Hardware configuration of the DSP-PC system

This appendix contains a detailed description of the hardware configuration of the system.

A.1 Description of the TMS320C30 PC Processor Board

The TMS320C30 board is a so-called processor board. The board consists of a digital signal processor TMS320C30 from Texas Instruments, DSPLINK (parallel expansion), memory expansion, prototyping area, two serial ports and several links (jumpers). The DSPLINK is connected to three 16 bit Stereo Interface Boards from LSI. These boards are described in section A.2.

The board is suitable only for the PC AT and compatibles, as it uses the full 16 bit interface. Access to memory passes through the dual porting hardware on the TMS320C30 board. The TMS320C30 dual port interface includes an address counter for block transfers. Furthermore, it includes hardware to transfer data between the 16 bit AT bus and the 32 bit DSP bus. Interrupts from the PC to the TMS320C30 and vice-versa are supported.

The base address (the eight most significant address lines) of the block of I/O ports at the PC side is specified by link LK2, which allows a possible 256 locations. A few recommended base addresses are 290h (default), 390h, 690h and 790h. These recommended addresses avoid the I/O ports used by most of the "standard" expansion cards found within PCs.

The TMS320C30 PC Interface Registers are mapped as is listed in table A.1. Only word accesses to this interface are permissable. In accesses to the board when the TMS320C30 has been held, the full 24 bit address is used. When the board has not been held, only the least

Table A.1: *I/O ports for the PC interface in the PC memory map of the TMS320C30 board*

ADDRESS	REGISTER	FUNCTION
BASE + 0 BASE + 2	DATA(L) DATA(H)	32 bit Data Register
BASE + 4 BASE + 6	ADDRESS(L) ADDRESS(H)	24 bit Address Register
BASE + 8	CTRL/STAT	16 bit Control/Status Register
BASE + C	INTR	Interrupt Register

significant 16 bits are used in the 64K word dual port area. If only the dual port area is to be used the Address Register at BASE + 6 need not be updated before each access. By setting bits in the Control Register (BASE + 8) the address can be made to count up or down after each address.

The board is designed to operate without the need to interrupt the PC. It is supplied with no interrupt facility enabled. However, in some circumstances one might need to generate an interrupt to the PC. This can be done using LK5. One can generate one of six interrupts (IRQ 3, 4, 5, 6, 7 or 9). Between the DSP and the PC there is a physical interrupt line. The various parts of the board are described in the next part of this section.

A.1.1 Memory

On the board there are 24 memory chips installed. Each chip is a 64K x 4 device, has 24 pins and is static RAM (CMOS). The maximum access time of the 16 devices that form the 128Kw main RAM is 25 nsec. The maximum access time of the 8 devices that form the 64Kw DP RAM is 15 nsec. The maximum access time is defined as the time interval between the address becoming valid and the valid data appearing at the output with the device in the read mode and the chip enable active. The memory map of the TMS320C30 development and applications board is given in table A.2. The chip is configured to operate in microprocessor mode (see section A.3.2),placing the interrupt locations in external memory and disabling the internal RAM at these locations. Two internal RAM memory blocks (1K word long) operate at the full processor speed with zero wait-states. Two memory areas are provided off-chip to supplement the 2K word RAM provided on-chip:

- *Area A* is divided into two areas. One 64K bank is configured with zero wait-state devices. External memory uses a hardware wait-state generator (on the board, but external to the TMS320C30 chip) to accomodate the different access times required by the various memory areas. Each area is 64K words long. Wait-states are set by on-board links to suit the speed of memory device used (see section A.4).
  For correct operation, memory area A **must always** be configured via the Primary Bus Control Register (at address 808064h) to respond only to externally controlled

wait-states to allow external hold requests and have 64K word bank sizes. The area is accessed from the PC by cycle stealing from the DSP if they both try to access this area simultaneously.
After a reset of the TMS320C30, the Primary Bus Control Register defaults to an incorrect state so that programs should set up this register by writing 800h to address 808064h. The Secondary Bus Control Register should also be set up by writing 0h to address 808060h (the secondary bus is the same as DSPLINK).

Table A.2: *Memory map of the TMS320C30 board*

BANK	SIZE (WORDS)	WAIT-STATES	ADDRESS
0 (Area A)	64K	0/1	000000h to 00FFFFh
1 (Area A)	64K	0/1	010000h to 01FFFFh
3 (Area B)	16K or 64K	0/1	030000h to 03FFFFh
Memory Expansion Connector	7936K	User defined	400000h to 7FFFFFh
DSPLINK	8K	2	800000h to 801FFFh
Reserved	8K		802000h to 803FFFh
Prototyping and PC interrupts	8K	2	804000h to 805FFFh
Reserved	8K		806000h to 807FFFh
On-chip peripherals	6K	Internal	808000h to 8097FFh
RAM 0	1K	Internal	809800h to 809BFFh
RAM 1	1K	Internal	809C00h to 809FFFh
Memory Expansion Connector	8152K	User defined	80A000h to FFFFFFh

- *Area B* is used for transfer of data between the PC and the DSP. Both 64K words of one wait-state memory and 16K x 4 devices can be fitted. Wait-states and memory size are set by on-board links (see section A.4).
  Area B is true dual ported memory in that it may be accessed without halting the chip. Therefore, many of the library functions include a parameter which identifies the type of memory to be accessed. Although the PC can read and write other memory areas on the board, it incurs more overhead because the DSP chip must be held during PC accesses.

A memory expansion connector is available to provide the user to access the unused external memory space on the primary TMS320C30 bus. Only daughter boards, such as 32 bit I/O boards, can be connected.

A.1.2 User prototyping area

On the board there is some place left for the user to construct his own custom circuitry. In this way, user specific digital or analog interface circuitry can be added to produce a single board solution with the possibility of moving to a custom printed circuit card at some later stage. Around the periphery of the prototyping area and elsewhere connections can be found for a number of address and data and control lines, mainly originating directly from the TMS320C30 itself.

A.1.3 Parallel expansion (DSPLINK)

A parallel expansion system is provided as a memory-mapped peripheral area. The DSPLINK interface supports 16 bit parallel transfers at a maximum rate of 5 Million 16 bit words/sec, including software overhead. A more typical "Peak Software Transfer Rate" is 3.33 Million words/sec. The actual data transfer rate over the DSPLINK depends on two things:

1. The rate at which the master can read or write data.
2. The rate at which the slave can be read from or written to.

In general, most standard DSPLINK master boards automatically insert one processor wait-state between accesses. In the present configuration the DSPLINK master board inserts no wait-state. Increasing throughput by eliminating DSPLINK wait-states does not come without a penalty. One must be extremely careful in his design to meet very stringent timing requirements. The actual throughput one can get depends on where the data is going after the DSP reads it in from DSPLINK or where the data comes from before the DSP puts it out to DSPLINK. Transfers over DSPLINK use two wait-states to achieve a 180 nsec transfer cycle. 8K secondary bus memory locations are available and multiple slave boards can be accomodated.

The DSPLINK expansion ports are mapped between address 800000h and 801FFFh. Thirteen address lines (A0 to A12) provide access to 8K I/O locations. Each location occupies a 32 bit word in the TMS320C30's memory space (memory-mapped I/O), but only the most significant 16 bits are used (DSPLINK is only 16 data bits wide). When reading from the I/O cards, the 16 bits that are read have to be converted to 32 bits. The 16 bits have to be shifted to the right, which means that they have to be shifted to the least significant part. The most significant part is filled with zeros. In this way, a 32 bit word is formed. When writing to the I/O cards the opposite has to be done (the bits have to be shifted to the left). All boards that support the DSPLINK standard interface fall into one of two categories:

1. DSPLINK master boards
2. DSPLINK slave boards

Master, in this context, means that the DSP chip on the processor board controls the activity on DSPLINK. Slaves, on the other hand, are controlled by the master. Transfers between the

master and slave can be polled or interrupt driven. In polled mode, the master periodically checks a Status Register on the slave board to see if an action should be taken. In interrupt driven mode, the slave asserts an interrupt into the DSP chip on the master board to initiate an action.

All of the DSPLINK master boards contain a DSP chip. There are two types of master boards:

1. System boards
2. Processor boards

Figure A.1 shows the timing specifications for the standard DSPLINK master boards.

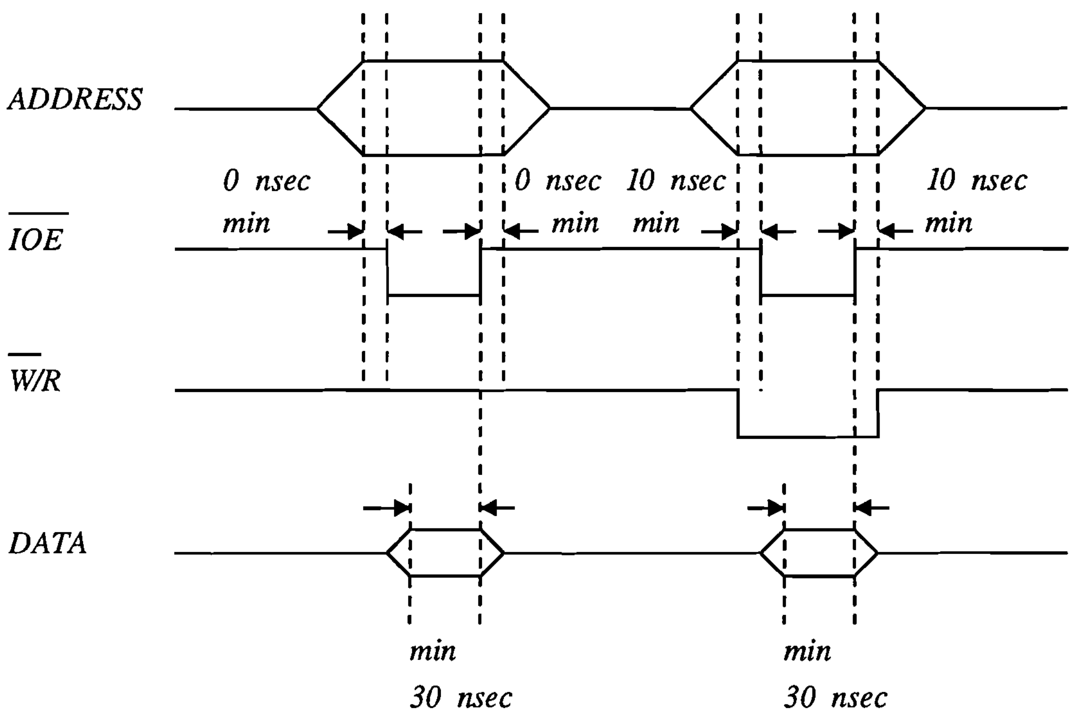


Figure A.1: *Timing specifications for DSPLINK*

System boards contain a DSP chip, memory and one or more on-board analog I/O channels. Processor boards do not have on-board analog I/O channels, thus in the present configuration the DSP board is a processor board.

The $\overline{W}$/R signal ($\overline{W}$ indicates a negative-true signal) and the address lines will be valid a minimum of 10 nsec before the $\overline{IOE}$ strobe (I/O enable) goes low on write cycles. On read cycles, $\overline{IOE}$ strobe changes at the same time as the address lines. Data to be read or data being written should be valid a minimum of 30 nsec before $\overline{IOE}$ goes high at the end of the

access. Data, $\overline{\text{W}}$/R and address lines are held a minimum of 10 nsec after this event. In general, the $\overline{\text{IOE}}$ strobe will be low for not less than 150 nsec for a port access.

A.1.4 Data transfers from the DSP to the PC

Data is transferred to/from the TMS320C30 board via the 32 bit Data Registers. Once the address of the data to be accessed has been set up, two 16 bit transfers over the PC I/O bus are required to complete the 32 bit access.

From the DSP point of view it is a matter of memory-mapped I/O, while from the PC point of view it is a matter of I/O mapped I/O. The principle of memory-mapped I/O and I/O mapped I/O has already been explained in section 3.1.2.

To read data from the board, the least significant 16 bit word is read first from the Data Register at address BASE + 0. This causes a 32 bit data value to be read from the TMS320C30 on-board memory and to be latched into the Data Registers in the interface hardware. The upper 16 bit word is then read from address BASE + 2.

The interrupt to the PC is set by the DSP writing to address 804004h. The DSP can check on the status of this interrupt by checking bit 31 of address 804005h. If this bit is set, the PC has not yet acknowledged the interrupt.

To clear the interrupt, one must read from the PC Interface Interrupt Register at address BASE + C. The interrupt status can be obtained from bit 6 of the PC Interface Status Register at address BASE + 8 (see table A.1). **Note:** before attempting to interrupt or signal the PC from the DSP, the interrupt to the PC should always be cleared by the user by reading address BASE + C.

A.1.5 Data transfers from the PC to the DSP

As mentioned in the previous part, the interfacing with the PC is established through a number of I/O ports on the PC (I/O mapped I/O). Through these ports the DP RAM can be accessed and several states of the DSP can be controlled. The ports are listed in table A.1, which indicates the I/O space.

To write data to the board, the least significant 16 bit word is first written to the register at BASE + 0. The most significant 16 bit word is then transferred to the register at address BASE + 2. The action of writing this location initiates the 32 bit transfer into the TMS320C30 on-board memory.

If the HOLD bit in the Control Register at address BASE + 8 has been asserted (active low) and a HOLDA valid signal received back (i.e. bit 1 of Status Register at address BASE + 8 is low) then the PC data transfers can access any populated memory area on the board. However, if the HOLDA signal is not active then only the dual ported area (Area B) of the board can be accessed. The HOLDA signal from the TMS320C30 affects the data transfer

mechanism to both the dual ported area (Area B) and the local memory (Area A). The bits in the Control Register and in the Status Register are listed in table A.3 and in table A.4 respectively.

Table A.3: *Bits in the Control Register of the TMS320C30 Board*

BIT	NAME	FUNCTION
0	RESET	Generates a reset to the processor. Active: high.
1	/HOLD	When active, this signal will halt the TMS320C30. Active: low.
2	/CNTRENB	Used to enable the interface address counter. Low: enables counter.
3	CNTRDIR	Sets interface address counter to count up (0) or down (1).
4	FLAG IN	General-purpose flag to the TMS320C30. Can be used to synchronize program operation or data transfers.

Note: if a HOLD or UNHOLD to the processor is initiated, then a valid acknowledge signal must be detected before attempting a data transfer.

To interrupt the DSP from the PC, one must write to the PC Interface Interrupt Register (table A.1) at address BASE + C. The status of this interrupt is determined from bit 5 of the PC Interface Status Register. This interrupt is cleared by the DSP reading from 804004h.

A.2 Description of the 16 bit Stereo Interface Boards

A.2.1 Outline description

The 16 bit Stereo Interface Board (SIB) is designed to connect directly to the TMS320C30 PC Processor Board. The boards consist of two A/D converters and two D/A converters each. Each analog input channel consists of a Programmable Gain input amplifier, a 4th order lowpass filter, a Sample and Hold amplifier and a fast A/D converter giving a 16 bit resolution with 14 bit linearity. The timer, filters and gain amplifiers can all be configured through jumpers and resistor packs (the resistor packs should be used for the filters to obtain the desired cut-off frequency).

The differential input signal is first fed through a Programmable Gain amplifier to support a range of peak to peak input signal levels. It is then passed into a low pass filter and a Sample and Hold amplifier. The output signal of the Sample and Hold will follow its input signal

until a trigger pulse is received by the A/D converter. This trigger will enable the signal to be captured and held during the analog-to-digital conversion process. The resulting 16 bit word is clocked out of the A/D converter serial output into a Shift/Storage Register. At this point the sampled data in the Shift Register can be read by the host DSP for processing. Any processed data may also be written back into the Shift Register before the next trigger pulse is received. The Shift Register data is clocked out to the D/A converter when the next data sample is clocked in. The resulting analog signal is then filtered before being seen at the channel output.

Table A.4: *Bits in the Status Register of the TMS320C30 board*

BIT	NAME	FUNCTION
0	/RESET	Determines status of the processor RESET pin. Active: low.
1	/HOLDA	This bit is low when the DSP has acknowledged that it has halted.
2		Not used.
3	FLAG OUT	Can be used for synchronization of programs or transfers.
4	FLAG IN	Status of the flag to the DSP. Set and reset via the Control Register.
5	INT PC	Status of the interrupt to the PC from the TMS320C30 board.
6	INTDSP	Status of interrupt to the DSP from the PC.
7-15		Not used.

In addition to the two analog channels the board contains a programmable sample rate timer, Control and Status Registers and circuitry to interface the board to the DSPLINK connector. Furthermore, each board has two 1 bit digital input and output channels. These channels can be used for synchronization.

A.2.2 Interface to the DSP

The board occupies four locations in the I/O addressing range of the DSPLINK master processor board. The registers of one I/O board are shown in table A.5. **Note**: The BASE address is **not** the same as the BASE address of thePC processor board. The BASE address of the SIB can be configured through LK4(see section A.4). Reading Shift Register A will clear the end of conversion (EOC) flag for channel A. Reading Shift Register B will clear the EOC flag for channel B.

The conversion can be triggered in three ways. The first way is with the timer on the SIB, the second way is with software through a register and the third way is with an external trigger.

Table A.5: *Registers of the SIB*

ADDRESS	WRITE	READ
BASE + 0	Control Register	Status Register
BASE + 1	Timer Register	Not useful
BASE + 2	Shift Register A	Shift Register A
BASE + 3	Shift Register B	Shift Register B

The bits in the Control Register and in the Status Register are listed in table A.6 and in table A.7 respectively. **Note**: the EOC flags will go high approximately 200 nsec before data becomes valid in the Shift Register. To avoid reading false data this must be allowed in the program if an exceptionally fast polling or interrupt service routine is used.

Table A.6: *Bits in the Control Register of the SIB*

BIT	NAME	FUNCTION
0	S/WARE TRIGGER	A 1 to 0 transistion will start a conversion.
1		Not used.
2	BIT OUT PIN 11	Can be used to control external circuitry.
3	BIT OUT PIN 12	See bit 2.
4	ENABLE CHANNEL A	If this bit is set to 0, then conversions on channel A are inhibited.
5	ENABLE CHANNEL B	See bit 4.
6	EOC INT ENABLE A	End of conversion on channel A sets a flag in the Status Register. If this bit is a 1, then an interrupt is also generated. Setting this bit to a 0 masks any end of conversion interrupt from channel A.
7	EOC INT ENABLE B	See bit 6.

A.2.3 Conversion timing

Referring to figure A.2 the timing of a conversion cycle on either channel can be seen. Control Register bit 4 is set to 1 to start a conversion on channel A when a trigger signal is received. A falling edge on the trigger signal initiates the cycle. It causes the Sample and Hold amplifier to enter the hold mode until the end of conversion which takes 4.3 μsec. The process of converting is indicated by bit 4 of the Status Register for channel A. This bit is 1 during conversion and 0 otherwise. The EOC is signalled to the DSP by a transition from 0 to 1 of a flag in the Status Register. Using the on-board timer the maximum sampling frequency that can be used on a single channel is 157 KHz. This limit is determined by the time taken

Table A.7: *Bits in the Status Register of the SIB*

BIT	NAME	FUNCTION
0, 1	S/WARE TRIGGER	These two status bits reflect the state of bit 0 in the Control Register.
2	BIT IN PIN 4	This bit is controlled from an external input on pin 4. If this pin is not connected, then bit 2 will appear as logic 1 because of a pull-up resistor.
3	BIT IN PIN 5	See bit 2.
4	STATUS CHANNEL A	This bit goes high at the start of a conversion on channel A and remains high throughout the conversion period. The conversion time is approximately 4 μsec.
5	STATUS CHANNEL B	See bit 4.
6	EOC FLAG CHANNEL A	The end of a conversion on channel A sets this flag. The flag is reset low by reading Shift Register A.
7	EOC FLAG CHANNEL B	See bit 6.

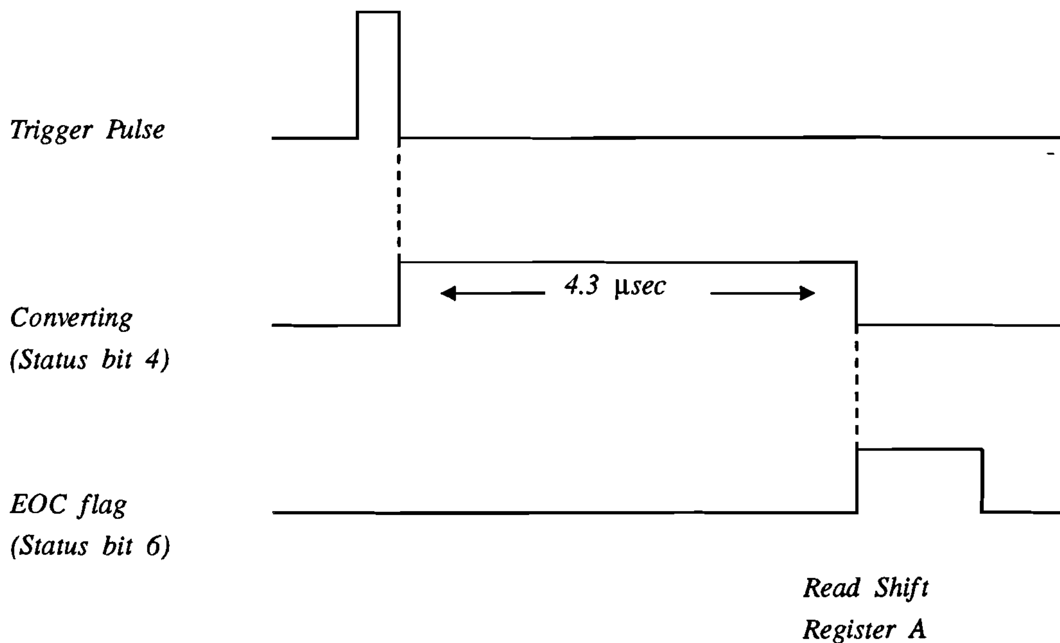


Figure A.2: *Timing of a conversion cycle of one I/O port*

to serially transfer 16 bits into the Shift Register for transfer over DSPLINK. If an external trigger is used the pulse width of the trigger pulse will also affect this limit.

Sampling rates up to 300 KHz can be achieved by feeding the input signal to both channels and sampling on each channel alternately. The success of the method is dependent on the

sampling rate being such that the next trigger pulse occurs before the end of conversion on one channel.

The external trigger pulse should have a maximum pulse width duration of 150 nsec and minimum period of 6.5 μsec to achieve a sampling rate of 153 KHz on one channel. If the pulse duration is any longer then the maximum sampling frequency will be reduced.

A.3 The Digital Signal Processor TMS320C30

A.3.1 General description

The TMS320C30 internal busing and special digital signal processing instruction set provide speed and flexibility. This combination produces a processor capable of executing up to 33 MFLOPS (million floating-point operations per second). The TMS320C30 optimizes speed by implementing functions in hardware that other processors implement through software or microcode.

The TMS320C30 can perform parallel multiply and ALU operations on integer or floating-point data in a single cycle. The processor also possesses a general-purpose register file, program cache, dedicated auxiliary register arithmetic units (ARAU), internal dual-access memories, one Direct Memory Access (DMA) channel supporting concurrent I/O and a short machine-cycle time. High performance and ease of use are achieved through greater parallelism, greater accuracy and general-purpose features.

Some key features of the TMS320C30 are listed below.

- 60 nsec single-cycle instruction execution time (33 MFLOPS, 16.7 MIPS (million instructions per second))
- one 4K x 32 bit single-cycle dual-access on-chip ROM block
- two 1K x 32 bit single-cycle dual-access on-chip RAM blocks
- 64 x 32 bit instruction cache
- 32 bit instruction and data words, 24 bit addresses
- on-chip DMA controller for concurrent I/O and CPU operation
- two 32 bit timers

A.3.2 Memory organization

The memory organization of the DSP chip is depicted in figure A.3.

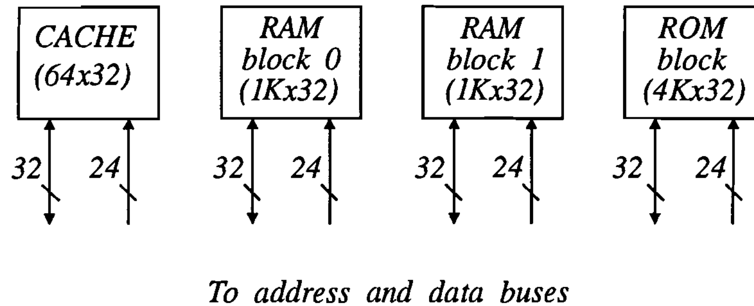


Figure A.3: *Memory organization of the TMS320C30*

The total memory space of the TMS320C30 is 16M (million) 32 bit words. Program, data and I/O space are contained within this 16M word address space, thus allowing tables, coefficients, program code or data to be stored in either RAM or ROM. In this way, memory usage can be maximized and memory space allocated as desired.

The ROM block is 4K x 32 bit. Each RAM and ROM block is capable of supporting two accesses in a single cycle. The separate program buses, data buses and DMA buses allow for parallel program fetches, data reads and writes and DMA operations. In this way the CPU can access two data values in one RAM block and perform an external program fetch in parallel with the DMA loading another RAM block, all within a single cycle. Within the ROM block there are 192 locations for interrupt vectors, trap vectors and a reserved space.

A 64 x 32 bit instruction cache stores often repeated sections of code. This greatly reduces the number of off-chip accesses necessary and allows code to be stored off-chip in slower, lower-cost memories. Three bits are provided in the CPU Status Register to control the clear, enable or freeze of the cache (see table A.12). The cache can operate in a completely automatic fashion without the need for user intervention.

The memory map is dependent upon whether the processor is running in the microprocessor mode or the microcomputer mode. In microprocessor mode the 4K on-chip ROM is not mapped into the TMS320C30 memory map. In microcomputer mode the 4K on-chip ROM is mapped into locations 0h through 0FFFh. In the present configuration the microprocessor mode is used. Table A.8 shows the memory maps of both modes.

Table A.8: *Memory maps of microprocessor mode and microcomputer mode*

LOCATION	FUNCTION	LOCATION	FUNCTION
0h	Interrupt locations and reserved (192)	0h	Interrupt locations and reserved (192)
C0h	External STRB active	C0h	ROM (internal)
		1000h	External STRB active
800000h	Expansion bus MSTRB active (8K)	800000h	Expansion bus MSTRB active (8K)
802000h	Reserved (8K)	802000h	Reserved (8K)
804000h	Expansion bus IOSTRB active (8K)	804000h	Expansion bus IOSTRB active (8K)
806000h	Reserved	806000h	Reserved (8K)
808000h	Peripheral bus memory-mapped registers (internal) (6K)	808000h	Peripheral bus memory-mapped registers (internal) (6k)
809800h	RAM block 0 (1K) (internal)	809800h	RAM block 0 (1K) (internal)
809C00h	RAM block 1 (1K) (internal)	809C00h	RAM block 1 (1K) (internal)
80A000h	External STRB active	80A000h	External STRB active
	Microprocessor mode		Microcomputer mode

The memory-mapped peripheral registers are located starting at address 808000h. The peripheral bus memory map is shown in table A.9. Each peripheral occupies a 16 word region of the memory map.

Table A.9: *Peripheral bus memory map*

LOCATION	FUNCTION
808000h	DMA controller registers (16)
808010h	Reserved (16)
808020h	Timer 0 registers (16)
808030h	Timer 1 registers (16)
808040h	Serial port 0 registers (16)
808050h	Serial port 1 registers (16)
808060h	Primary and expansion port registers (16)
808070h	Reserved

A.3.3 Interrupts

The TMS320C30 supports multiple internal and external interrupts which can be used for a variety of applications. External interrupts are synchronized internally. Once synchronized, the interrupt input will set the corresponding Interrupt Flag (IF) Register bit if the interrupt is active. Table A.10 and table A.11 show the IF Register and Interrupt Enable (IE) Register respectively.

Table A.10: *CPU Interrupt Flag Register*

BIT	NAME	FUNCTION
0	INT0	External interrupt 0 flag
1	INT1	External interrupt 1 flag
2	INT2	External interrupt 2 flag
3	INT3	External interrupt 3 flag
4	XINT0	Serial port 0 transmit interrupt flag
5	RINT0	Serial port 0 receive interrupt flag
6	XINT1	Serial port 1 transmit interrupt flag
7	RINT1	Serial port 1 receive interrupt flag
8	TINT0	Timer 0 interrupt flag
9	TINT1	Timer 1 interrupt flag
10	DINT0	DMA channel interrupt flag
11-31	Reserved	Value undefined

External interrupts can be effectively either edge- or level-triggered, depending on the duration of the low level on the interrupt input. If the interrupt is held low for between one and three cycles, then only one interrupt is recognized. If the interrupt is held low three or more cycles, more than one interrupt may be recognized depending on how rapidly interrupts are serviced.

When a particular interrupt is processed by the CPU or DMA controller, the corresponding interrupt flag bit is cleared by the internal interrupt acknowledge signal. The IF Register bits may be read and written under software control. Writing a 1 to an IF Register bit sets the associated interrupt flag to 1. Similiarly, writing a 0 resets the corresponding interrupt flag to 0. In this way, all interrupts may be triggered and/or cleared through software.

Internal interrupts operate in a similar manner. The bit in the IF Register corresponding to an internal interrupt may be read and written through software. Writing a 1 sets the interrupt latch and writing a 0 clears it.

The CPU Global Interrupt Enable (GIE) bit, located in the CPU Status Register (see table A.12), controls all CPU interrupts. All DMA interrupts are controlled by the DMA GIE bit, which is not dependent upon the GIE bit in the Status Register and is local to the DMA. The DMA GIE bit is not directly accessible through software.

Table A.11: *CPU/DMA Interrupt Enable Register*

BIT	NAME	FUNCTION
0	EINT0	Enable external interrupt 0 (CPU)
1	EINT1	Enable external interrupt 1 (CPU)
2	EINT2	Enable external interrupt 2 (CPU)
3	EINT3	Enable external interrupt 3 (CPU)
4	EXINT0	Enable serial port 0 transmit interrupt (CPU)
5	ERINT0	Enable serial port 0 receive interrupt (CPU)
6	EXINT1	Enable serial port 1 transmit interrupt (CPU)
7	ERINT1	Enable serial port 1 receive interrupt (CPU)
8	ETINT0	Enable timer 0 interrupt (CPU)
9	ETINT1	Enable timer 1 interrupt (CPU)
10	EDINT	Enable DMA controller interrupt (CPU)
11-15	Reserved	Value undefined
16	EINT0	Enable external interrupt 0 (DMA)
17	EINT1	Enable external interrupt 1 (DMA)
18	EINT2	Enable external interrupt 2 (DMA)
19	EINT3	Enable external interrupt 3 (DMA)
20	EXINT0	Enable serial port 0 transmit interrupt (DMA)
21	ERINT0	Enable serial port 0 receive interrupt (DMA)
22	EXINT1	Enable serial port 1 transmit interrupt (DMA)
23	ERINT1	Enable serial port 1 receive interrupt (DMA)
24	ETINT0	Enable timer 0 interrupt (DMA)
25	ETINT1	Enable timer 1 interrupt (DMA)
26	EDINT	Enable DMA controller interrupt (DMA)
27-32	Reserved	Value undefined

Table A.12: *Bits in the Status Register of the DSP*

BIT	NAME	FUNCTION
0	C	Carry flag
1	V	Overflow flag
2	Z	Zero flag
3	N	Negative flag
4	UF	Floating-point underflow flag
5	LV	Latched overflow flag
6	LUF	Latched floating-point underflow flag
7	OVM	Overflow mode flag. If OVM = 0, then the overflow mode is turned off.
8	RM	Repeat mode flag
9	Reserved	Read as 0
10	CF	Cache freeze. When CF = 1 the cache is frozen.
11	CE	Cache enable. CE = 1 enables the cache.
12	CC	Cache clear. CC = 1 invalidates all entries in the cache.
13	GIE	Global Interrupt Enable. If GIE = 1 the CPU responds to an enabled interrupt.
14-15	Reserved	Read as 0.
16-31	Reserved	Value undefined.

The CPU controls all prioritization of interrupts. The interrupts are automatically prioritized according to the order of table A.13, with INT0 having the highest priority. If the DMA is not using interrupts for synchronization of transfers, it will not be affected by the processing of the CPU interrupts. If the CPU is involved in a pipeline conflict, it will not respond to the interrupts until that conflict is resolved. A delayed branch for instance is completed first. This means a maximum delay of four cycles (instruction fetch, instruction decode, operand fetch and execute) or 240 nsec. Then the address of the instruction that would have been fetched subsequently is pushed on the stack, the interrupts are disabled and the vector address belonging to the occurring interrupt is loaded in the PC and execution continues.

When it is desired that an interrupt can be broken off to allow processing of another interrupt, then the interrupt should be enabled through the corresponding bit in the IE Register and the global interrupts should be enabled in the Status Register.

The most important interrupt to be used with this application is the one that is generated by an EOC of the I/O card on the DSPLINK. The DSPLINK uses INT1 for interrupting the DSP. Detection of this interrupt is done by checking the EOC bit in the Status Register of the I/O card (see table A.7). It is therefore possible to interrupt the CPU and DMA

Table A.13: *Reset and interrupt vector locations of the DSP*

RESET OR INTERRUPT	VECTOR LOCATION	FUNCTION
RESET	0h	External reset signal input on the RESET pin.
INT0	1h	External interrupt input on the INT0 pin.
INT1	2h	External interrupt input on the INT1 pin.
INT2	3h	External interrupt input on the INT2 pin.
INT3	4h	External interrupt input on the INT3 pin.
XINT0	5h	Internal interrupt generated when serial port 0 transmit buffer is empty.
RINT0	6h	Internal interrupt generated when serial port 0 receive buffer is full.
XINT1	7h	Internal interrupt generated when serial port 1 transmit buffer is empty.
RINT1	8h	Internal interrupt generated when serial port 1 receive buffer is full.
TINT0	9h	Internal interrupt generated by timer 0.
TINT1	0Ah	Internal interrupt generated by timer 1.
DINT	0Bh	Internal interrupt generated by DMA controller.

simultaneously with the same or different interrupts and synchronize their activities. Since the DMA and CPU share the same set of interrupt flags, the DMA may clear an interrupt flag before the CPU can respond to it.

A.3.4 Timers

The TMS320C30 timer modules are general-purpose 32 bit timer/event counters with two signalling modes and internal or external clocking. The timer modules can be used to signal to the processor or the external world at specified intervals or to count external events. With an internal clock, the timer can be used to signal an external A/D converter to start a conversion or it can interrupt the DMA controller to begin a data transfer. With an external input, the timer can count external events and interrupt the CPU after a specified number of events. Available to each timer is an I/O pin that can be used either as an input clock to the timer, an output clock signal or a general-purpose I/O pin.

Three memory-mapped registers are used by each timer:

- Global Control Register
- Period Register
- Counter Register

The Global Control Register determines the operating mode of the timer, monitors the timer status and controls the function of the I/O pin of the timer. The Period Register specifies the timer's signalling frequency. The Counter Register contains the current value of the incrementing counter. The counter is set to zero whenever its value equals that in the Period Register. The memory map for the timer modules is shown in table A.14.

Table A.14: *Memory-mapped timer locations of the DSP*

ADDRESS TIMER 0	ADDRESS TIMER 1	REGISTER
808020h	808030h	Timer Global Control Register
808021h to 808023h	808031h to 808033h	Reseved
808024h	808034h	Timer Counter Register
808025h to 808027h	808035h to 808037h	Reserved
808028h	808038h	Timer Period Register
808029h to 80802Fh	808039h to 80803Fh	Reserved

A.4 Status of the present hardware

Table A.15 shows the present status of the processor board. Banks 0 and 1 contain area A, while bank 3 contains area B (see table A.2). In the tables A.17 through A.22 the present status of the 16 bit Stereo Interface Boards is listed. The tables in this section need to be updated every time changes are made to the system. Remark: the first SIB is the upmost board in the system.

The resistor packs of the Stereo Interface Boards were changed on 16-08-1995. The old resistor packs contained resistors of 8.2 KΩ, while the new resistor packs contain resistors of 15 KΩ, resulting in a cut-off frequency f_c = 4 KHz (the formula for calculating f_c can be found in [SIB 91]).

Table A.15: *Present status of the TMS320C30 PC Processor Board*

DESCRIPTION	DEFAULT	STATUS	DATE
LK1: Bank 3 memory size selection. A = 64K, B = 16K	A	A	12-05-1995
LK2: I/O base address selection. A= A11, B = A10, C = A9, D = A8, E = A7, F = A6, G = A5, H = A4	A, B, D, F, G (290h)	A, B, D, F, G (290h)	12-05-1995
LK3: Memory wait-state selection. A inserted = bank 3 uses 1 wait-state A absent = bank 3 uses 0 wait-states B inserted = bank 1 uses 1 wait-state B absent = bank 1 uses 0 wait-states C inserted = bank 0 uses 1 wait-state C absent = bank 0 uses 0 wait-states	A	All banks use 0 wait-states	12-05-1995
LK 4: Used for factor testing.			
LK 5: Enable interrupts from DSP to PC (only one link can be inserted). A = IRQ3, B = IRQ4, C = IRQ5, D = IRQ6, E = IRQ7, F = IRQ9	No links inserted	C	12-05-1995
LK 6: Serial port 0 buffer directions (inserted = input, absent = output). A = CLKR0, B = PSR0, C = PSX0, D = CLKX0	All absent	A, B	12-05-1995

Table A.16: *Continuation of table A.15*

DESCRIPTION	DEFAULT	STATUS	DATE
LK 7: Serial port 1 buffer directions (inserted = input, absent = output). A = CLKR1, B = PSR1, C = PSX1, D = CLKX1	All absent	A, B	12-05-1995
LK8: INT1 selection. A = Prototyping area B = DSPLINK INT1	B	A	12-05-1995
LK9: XF1 selection (only one link can be inserted). A = DSPLINK flag-in B = PC interface control register C = memory expansion connector	C	B	12-05-1995
LK10: INT0 selection. A = memory expansion B = DSPLINK INT0	A	A	12-05-1995
User prototyping area	Not used	Not used	12-05-1995
Memory expansion	Not used	Not used	12-05-1995
Serial port 0	Not used	Not used	12-05-1995
Serial port 1	Not used	Not used	12-05-1995
DMA	Not used	Not used	12-05-1995
DSP mode	?	Microprocessor mode	12-05-1995

Table A.17: *Present status of the first 16 bit Stereo Interface Board*

DESCRIPTION	DEFAULT	STATUS	DATE
LK1, LK2: Control of the channel input filter by-pass. LK1 = channel A LK2 = channel B A = via filter, B = by-pass filter	A	B	12-05-1995
LK3: Links both channel A and B Sample and Hold inputs together. LK1 or LK2 should be removed to disable the unused channel.	open circuit	open circuit	12-05-1995
LK4: Determines the base address of the board. A = base address 0 (addresses 0, 1, 2 and 3 are used) B = base address 4 (addresses 4, 5, 6 and 7 are used) C = base address 8 (addresses 8, 9, 10 and 11 are used) D = base address 12 (addresses 12, 13, 14 and 15 are used)	B	A	12-05-1995
LK5, LK6: Determine the factor of the programmable gain amplifiers. A (LK5) and A (LK6): gain = 8 B (LK5) and A (LK6): gain = 4 A (LK5) and B (LK6): gain = 2 B (LK5) and B (LK6): gain = 1	B and B	B and B	16-08-1995

Table A.18: *Continuation of table A.17*

DESCRIPTION	DEFAULT	STATUS	DATE
LK7, LK8: Select the trigger source for sampling. LK7 = channel A LK8 = channel B A: trigger source = bit 0 in Control Register B: trigger source = timer C: trigger source = through pin 2 (channel A) or pin 3 (channel B) of the DB15 control connector	B	B, C	12-05-1995
LK9, LK10: Power supply source. Open circuit: supply from DB15 control connector	supply from PC	closed circuit.	12-05-1995
Timing of the sampling period.	Not configured	Timer on the first SIB is used to trigger all the I/O boards.	12-05-1995

Table A.19: *Present status of the second 16 bit Stereo Interface Board*

DESCRIPTION	DEFAULT	STATUS	DATE
LK1, LK2: Control of the channel input filter by-pass. LK1 = channel A LK2 = channel B A = via filter, B = by-pass filter	A	B	12-05-1995
LK3: Links both channel A and B Sample and Hold inputs together. LK1 or LK2 should be removed to disable the unused channel.	open circuit	open circuit	12-05-1995
LK4: Determines the base address of the board. A = base address 0 (addresses 0, 1, 2 and 3 are used) B = base address 4 (addresses 4, 5, 6 and 7 are used) C = base address 8 (addresses 8, 9, 10 and 11 are used) D = base address 12 (addresses 12, 13, 14 and 15 are used)	B	B	12-05-1995
LK5, LK6: Determine the factor of the programmable gain amplifiers. A (LK5) and A (LK6): gain = 8 B (LK5) and A (LK6): gain = 4 A (LK5) and B (LK6): gain = 2 B (LK5) and B (LK6): gain = 1	B and B	B and B	16-08-1995

Table A.20: *Continuation of table A.19*

DESCRIPTION	DEFAULT	STATUS	DATE
LK7, LK8: Select the trigger source for sampling. LK7 = channel A LK8 = channel B A: trigger source = bit 0 in Control Register B: trigger source = timer C: trigger source = through pin 2 (channel A) or pin 3 (channel B) of the DB15 control connector	B	C	12-05-1995
LK9, LK10: Power supply source. Open circuit: supply from DB15 control connector	supply from PC	closed circuit	12-05-1995

Table A.21: *Present status of the third 16 bit Stereo Interface Board*

DESCRIPTION	DEFAULT	STATUS	DATE
LK1, LK2: Control of the channel input filter by-pass. LK1 = channel A LK2 = channel B A = via filter, B = by-pass filter	A	B	12-05-1995
LK3: Links both channel A and B Sample and Hold inputs together. LK1 or LK2 should be removed to disable the unused channel.	open circuit	open circuit	12-05-1995
LK4: Determines the base address of the board. A = base address 0 (addresses 0, 1, 2 and 3 are used) B = base address 4 (addresses 4, 5, 6 and 7 are used) C = base address 8 (addresses 8, 9, 10 and 11 are used) D = base address 12 (addresses 12, 13, 14 and 15 are used)	B	C	12-05-1995
LK5, LK6: Determine the factor of the programmable gain amplifiers. A (LK5) and A (LK6): gain = 8 B (LK5) and A (LK6): gain = 4 A (LK5) and B (LK6): gain = 2 B (LK5) and B (LK6): gain = 1	B and B	B and B	16-08-1995

Table A.22: *Continuation of table 21*

DESCRIPTION	DEFAULT	STATUS	DATE
LK7, LK8: Select the trigger source for sampling. LK7 = channel A LK8 = channel B A: trigger source = bit 0 in Control Register B: trigger source = timer C: trigger source = through pin 2 (channel A) or pin 3 (channel B) of the DB15 control connector	B	C	12-05-1995
LK9, LK10: Power supply source. Open circuit: supply from DB15 control connector	supply from PC	closed circuit	12-05-1995

Appendix B

Directory structure and files

In this appendix some important directories and batch files of disk C are described. Furthermore, a "getting started" section is given to explain the user how to compile his source code.

B.1 Directory structure of disk C

Contents of directory C:\ORIGINAL:

- C:\ORIGINAL\APPLICAT: TMS320C30 Applications Support Software
- C:\ORIGINAL\CDEBUG: TI 320C30 "C" Source Debugger v1.10
- C:\ORIGINAL\DEBUG: TMS320C30 MON30 Debug Monitor v1.22/SDS
- C:\ORIGINAL\HLLIL: TMS320C30 HLL Interface Libraries v2.02
- C:\ORIGINAL\IOCARDEX: 16 Bit Stereo Card Example Programs v1.01
- C:\ORIGINAL\PLAINC: TMS320C30 Plain C Example Disk v1.00
- C:\ORIGINAL\SPOX_LSI: SPOX LSI Distribution Disk v1.40
- C:\ORIGINAL\DSPSYS: backup of the user work directory

Remark: the SPOX LSI Distribution Disk v1.32 and the TI 320C30 "C" Source Debugger v1.00 diskette have not been copied to this directory, because they have an older version number.

In directory C:\DSPSYS\DSP all files that must run on the DSP board are stored. These files have the extension *c30.* Contents:

- *dspmain.c30*: main file
- *dspmain.h*: header file for *dspmain.c30*
- *userctrl.c30*: control commands of user
- *defs.h*: definitions of offset address, on-chip memory space, debug mode, channel status, algorithm states, algorithm parameters and debug parameters
- *dsp.asm*: assembler routines
- *make.bat*: batch file which activates *cl30.exe* for compiling and linking C code on the DSP (see further)
- *temp.log*: the comments produced by *cl30* are stored in this file

When assembling and linking *dsp.asm* the output files are:

- *dsp.o30*: non-executable object code
- *dsp.lst*: list and cross-reference
- *dsp.out*: executable object code produced by the TMS320C30 Assembler/ Linker
- *dsp.map*: map of memory use
- *dsp.sym*: symbols sorted by name
- *dsp.adr*: symbols sorted by address
- *dsp.cmd*: memory allocation information

In directory C:\DSPSYS\PC all files that must run on the PC are stored. These files have the extension *c*. Contents:

- *pcmain.c*: main file
- *pcmain.h*: header file for *pcmain.c*
- *30librar.c*: library of functions to make the interface to the DP RAM easier
- *TMS30.h*: header file for *30librar.c*
- *grgui.c*: user interface in graphic mode
- *grgui.h*: header file for *grgui.c*
- *txtgui.c*: user interface in text mode

- *convert.c*: converts binary values into ascii values
- *interact.h*: header file for *txtgui.c*
- *keys.h*: header file for *txtgui.c*
- *pch.h*: include files for *pcmain.c*
- *data.frm*: format for logging data to hard disk
- *egavga.bgi*: screen driver
- *pc.prj*: consists of five files: *egavga.obj*, *30librar.c*, *pcmain.c*, *grgui.c* and *txtgui.c*.
- *read.bat*: batch file which reads file *data.log*
- *data.log*: ascii file which is produced by *convert.exe*
- *data.bin*: binary file which is produced if data is logged to the hard disk

Contents of directory C:\DSPSYS\DEMO:

- C:\DSPSYS\DEMO\DEMO1: simple algorithm without debugging and without logging to the hard disk
- C:\DSPSYS\DEMO\DEMO2: same algorithm with debugging
- C:\DSPSYS\DEMO\DEMO3: usage of digital input channels
- C:\DSPSYS\DEMO\DEMO4: PID controller without debugging and without logging to the hard disk
- C:\DSPSYS\DEMO\DEMO5: PID controller with logging to the hard disk, but without debugging
- C:\DSPSYS\DEMO\DEMO6: PID controller with debugging (defining debug inputs in file *defs.h*)
- C:\DSPSYS\DEMO\DEMO7: PID controller with debugging (reading debug inputs from the hard disk)

Directory C:\DSPSYS\ADC contains files that must be used when logging data to the hard disk in order to get an exact representation of an input signal. The data is used in debug mode.

Figure B.1 gives an overview of how include files on the DSP and the PC are organized. For example, file *dspmain.c30* has one include file *dspmain.h*. File*dspmain.h* has an include file *defs.h*. File *userctrl.c30* hasthe same include file as file *dspmain.c30*.

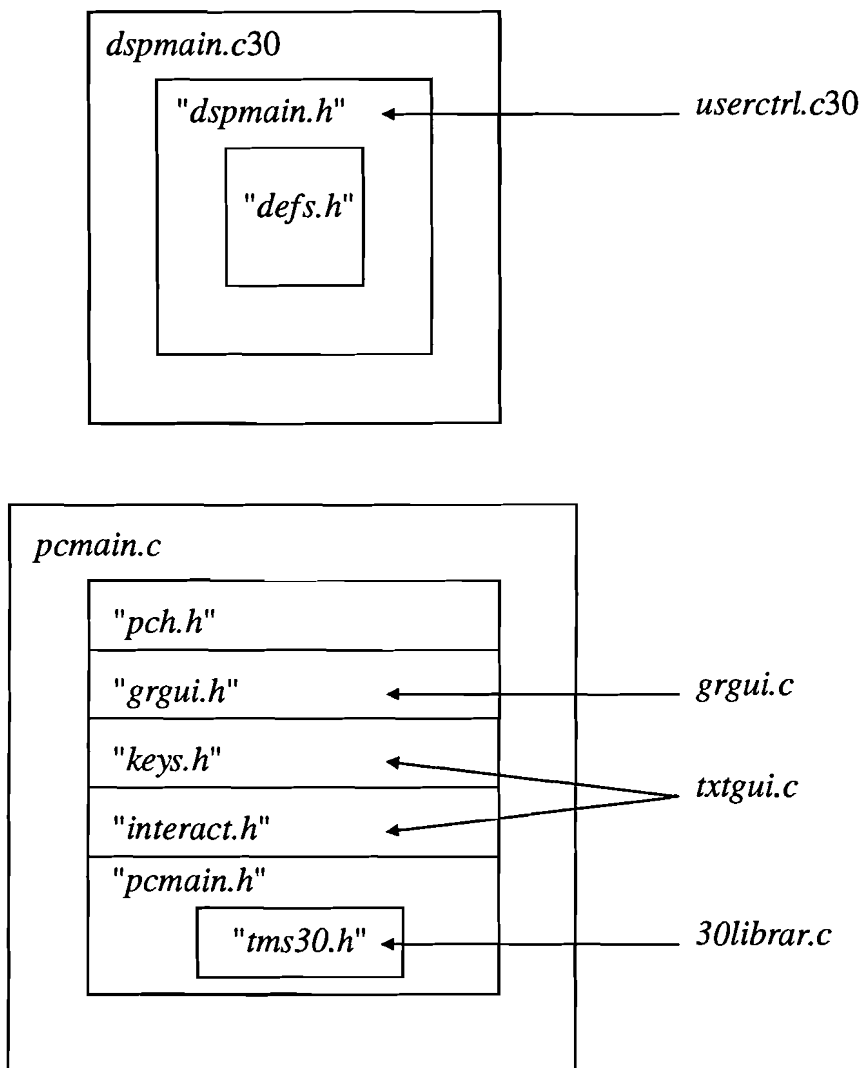


Figure B.1: *Organization of include files.*

B.2 Some important batch files

- *setdsp.bat*: When starting (i.e. compiling and running of files concerning the DSP) this batch file must be activated. The file is in the root of the C disk and by activating it the directory C:\BAT will be appended to the path. Then a call to C:\BAT*setspox.bat* is made.
- *setspox.bat*: This batch file appends three directories to the path: C:\ORIGINAL\DEBUG, C:\ORIGINAL\SPOX\BIN and C:\DSP\C30TOOLS. The latter directory contains *cl30.exe* (the TMS320C30 Assembler/Linker) which compiles and links ANSI C source code on the DSP. Furthermore, some environment settings are done for the DSP and the SPOX software. The explanation of these settings is given as remark statements in the batch file itself.
- *bc3.bat*: This batch file appends directory C:\BORLANDC\BIN to the path and activates Borland C++ version 3.1.
- *dsp.bat*: switch to directory C:\DSPSYS\DSP.
- *pc.bat*: switch to directory C:\DSPSYS\PC.
- *make.bat*: compile and link source code on the DSP. The explanation of the contents of this batch file is given as remark statements in the batch file itself.

B.3 Getting started

- Turn on power supply of the I/O cards.
- Turn on power supply of the PC.
- Activate *setdsp.bat* by typing *setdsp*. Now *bc3.bat* (a.o.) can be activated.
- Activate *bc3.bat* by typing *bc3* (still in the root).
- Open project *pc.prj*.
- If source code for the PC must be compiled, simply use the Borland C++ function keys.
- If source code for the DSP must be compiled, switch to DOS shell (use Borland C++ function keys), use *dsp.bat* to switch to directory C:\DSPSYS\DSP and type *make* to activate *make.bat*. When finished, type *pc* to return to directory C:\DSPSYS\PC and return to the Borland C++ shell by typing *exit*.

Index

30librar.c, 61

A

absolute time, 13
ack_adr, 60, 85
action, 41, 42
action table, 43, 48, 54
ad_0, 77
ad_adr, 60
ad_data.bin, 86, 87
ad_data.frm, 86, 87
ad_data.log, 86
ad_ptr, 62, 77, 80, 86
ad_status_adr, 60
adc.exe, 86, 87
aperiodic task, 13

B

background, 15, 17, 20
background module, 18, 19
background scheduling, 20
background task, 15, 17, 20, 46
buffering, 16, 19

C

c_int01(), 62, 73, 85
c_intxx(), 62
causal connection, 41
CISC, 37
clock-based system, 12, 46
close_fifo(), 62
code sharing, 22, 46
COFF, 29
command_adr, 60, 63
condition, 41–43
context scheme, 46
Control algorithm transformation, 47, 48
control command flow, 47
Control system transformation, 46, 47, 48
Control tasks transformation, 48, 63
Control timer transformation, 48
control transformation, 41, 42
Control user command transformation, 47, 48
controller task, 10
Controller tasks block, 10
convert ad_data.bin, 86, 87
convert data.bin, 79, 80
Convert selected parameters transformation, 48, 62
Convert signals transformation, 48, 62
correctness, 14, 15
cyclic process, 19
cyclic task, 12

D

da_0, 77
da_1, 77
da_adr, 60
da_ptr, 62, 77, 78, 80
da_status_adr, 60
data.bin, 79, 80
data.frm, 79, 80
data.log, 79, 80
deadlock, 21, 22, 46
debug_index_adr, 60, 86
debug_mode, 65, 73, 80, 85
debug_mode(), 65
debug_status_adr, 60, 74, 85
debugflt, 67
debugflt_adr, 65, 86
debugging, 10
debugint, 67
debugint_adr, 65, 85, 86
decode unit, 34
defs.h, 60, 61, 65, 73–77, 80, 85–87
dependability, 13, 15

display update module, 17
DMA channel, 34
DP RAM, 28, 29, 48, 57, 59–63, 65, 68, 85, 86
DSP interrupt counter, 59, 60
dsp.asm, 63
`dsp_int_time`, 78
`dspint_adr`, 60
DSPLINK, 25–29, 31
dspmain.c30, 60–62, 73–76, 78, 80, 85, 86
dspmain.h, 60, 61

E

`error_adr`, 60
event flow, 41
event-based, 12
event-based system, 12, 13
event-based task, 13
event-processing, 12
Execute display command transformation, 56
Execute set command transformation, 56
execute unit, 34

F

feasible, 19
fetch unit, 34
`flag_adr`, 60, 85
`float_par()`, 62
flow, 40, 41
`fltstates`, 61
`fltstates_ptr`, 60
`fltstatesarray`, 61
`fltstatesindex`, 61
`flush_fifo()`, 62
foreground, 15, 17, 20
foreground module, 15, 18, 19
foreground task, 15–17, 20, 22, 46

G

`Get32Bit()`, 68
`GetFloat()`, 68
`GetInt()`, 68
grgui.c, 63

H

`handler()`, 62
hard real-time system, 14, 19
Harvard architecture, 32, 33
`HDLOG`, 78, 79

I

I/O mapped I/O, 28
`init_data()`, 61, 75–77
`init_sett_par()`, 61, 77
`Init_vars()`, 75, 76
input image, 9
`int_par()`, 62
integrator windup, 73
interactive system, 12, 13
interface, 9, 12, 13
interrupt latency, 36

K

`kbhit()`, 63

L

`log_da0`, 78
`log_da1`, 78

M

`main()`, 60, 61, 63
malloc, 60, 61
management information module, 17
`max_debugflt`, 65, 86, 87
`max_debugint`, 65, 86, 87
memory-mapped I/O, 27, 28
monitor, 21
mutual exclusion, 20, 21, 46

N

net flow, 40
non-pipelined processor, 32

O

`open_fifo()`, 62
operator input module, 17, 19
output image, 9

P

`pariblock_ptr`, 75

Parse user command transformation, 55
pcint_adr, 60
pcmain.c, 61–63, 65, 75–78, 80, 86, 87
pcmain.h, 60–62, 65, 78, 79, 86, 87
Pentium, 37
Perform algorithm transformation, 48, 62
periodic task, 12
pipelined processor, 33
predictability, 13, 15
Process user commands transformation, 48, 63
process_cmd(), 63
Put hard disk data transformation, 48, 62
PutFloat(), 62
PutInt(), 62, 65
putpixel(), 69

R

RdBlkFlt(), 61
RdBlkInt(), 61
re-entrant code, 22
re-scheduling, 22
read data.log, 79, 80
read unit, 34
readiness, 15
real-time system, 8, 12–15
real-time system (definition), 8
relative time, 13
reset(), 77, 80
RISC, 37
robustness, 15

S

samp_freq_adr, 60, 63
Sample frequency store, 48
sample_freq(), 63
scheduler, 20
scheduling, 17, 19
Selection store, 48
semaphore, 21
serially re-usable code, 22
setbuf(), 62
setfltpar, 62
setfltpar_adr, 61, 62
setfltpar_ptr, 61
setfltpararray, 62
setfltparindex, 62
setparblock_ptr, 61, 76
Settable parameters store, 48
Settings store, 47, 48
setverify(), 62
show_algorithm_variables(), 61
signal_1, 78
signal_4, 78
simultaneousness, 13, 14
soft real-time system, 14
sporadic process, 19
start_int(), 63
starttimer(), 62, 63
state, 41–43
state graph, 21
state-processing, 12
state-transition diagram, 41, 48
state-transition table, 43, 48, 51, 54
statesblock_adr, 60, 61
statesblock_ptr, 60, 74
stop summation, 73
stop_int(), 63
stoptimer(), 62, 63
store, 40, 41
synchronization, 12, 19, 20

T

test.bin, 69
time-continuous behavior, 40
time-discrete behavior, 40
timeliness, 13, 14
Timer transformation, 48, 62, 63
transformation, 40, 41, 44
transition, 41, 42

U

userctrl.c30, 62, 63, 77, 80

V

Von Neumann architecture, 32, 37

W

WrBlkFlt(), 62, 65, 86
WrBlkInt(), 65, 86
write_fifo(), 62