TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Real-time system control and X, from specification to implementation

van Rosmalen, J.

Award date:
1998

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

Real-Time System Control and χ, from Specification to Implementation

J. van Rosmalen

Systems Engineering, 420164

Master's thesis

Coaches: ing. H.W.A.M. van Rooij
dr.ir. J.M. van de Mortel - Fronczak
Supervisor: prof.dr.ir. J.E. Rooda

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF MECHANICAL ENGINEERING
SECTION SYSTEMS ENGINEERING

Eindhoven, January 1998

EINDSTUDIE-OPDRACHT

TECHNISCHE UNIVERSITEIT EINDHOVEN
Faculteit Werktuigbouwkunde
Vakgroep WPA
Sectie Systems Engineering

juli 1996

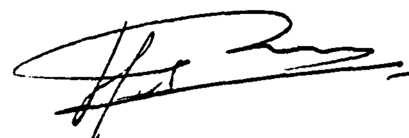
Student:	J. van Rosmalen
Hoogleraar:	Prof.dr.ir. J.E. Rooda
Begeleider(s):	Dr.ir. J.M. van de Mortel-Fronczak Ing. H.W.A.M. van Rooij
Start:	juli 1996
Einde:	maart 1997
Titel:	Koppeling van χ met een real-time kernel en een I/O systeem.

Onderwerp:
Machinebesturingen kunnen met χ worden gespecificeerd. Vervolgens is simulatie van de besturing bruikbaar om de functionaliteit ervan te toetsen.
De overgang van specificatie naar implementatie kan op diverse manieren worden uitgevoerd. Hierbij worden meestal andere software gereedschappen gebruikt dan bij de specificatie. Onderzocht wordt of deze 'gereedschapswissel' voorkomen kan worden door de specificatie te gebruiken voor de directe afleiding van een executeerbaar besturingsprogramma.

Opdracht:
Bestudeer de werking van digitale besturingen. Analyseer de interactie tussen machinebesturing en de machine en beschrijf de basis operaties die nodig zijn om het adequaat besturen van een machine mogelijk te maken. Neem een van de Festo werkstations en het Interbus-S I/O-systeem als uitgangspunt.
Probeer een set basisfuncties af te leiden, waarmee besturingen met de specificatietaal χ goed te beschrijven zijn.
Ontwikkel een executeerbare versie van de bovengenoemde specificatie van het werkstation. Maak (gebruik van) de drivers voor het Interbus-S systeem.
Onderzoek de werking van RT-Kernels. Geef aan of de huidige χ-kernel gebruik kan maken van een RT-Kernel voor het besturen van machines. Voer een aantal experimenten uit.

Prof.dr.ir. J.E. Rooda

Dr.ir. J.M. v.d. Mortel-Fronczak


Ing. H.W.A.M. van Rooij

Systems
Engineering

Faculteit Werktuigbouwkunde

"The boundries of my language are the borders of my world."

L.J.J. Wittgenstein (1889-1951)

I would like to thank Lilian, for patiently putting up with me for the long years I spent at the university. Her support means a lot to me.

Also, I would like to thank Albert and Wilbert. They kindly let me share in their expertise in the χ-kernel and C^{++}.

Samenvatting (in Dutch)

In de Sectie Systems Engineering aan de Technische Universiteit te Eindhoven wordt een specificatie taal ontwikkeld genaamd χ. Met de specificatie taal χ is men in staat om modellen te maken van industriele systemen. (Deze industriele systemen kunnen zowel fabrieken als machines zijn.) Een model van een machine bevat een deel waarin de besturing van de machine gemodelleerd wordt. Wanneer er een model wordt gemaakt van de besturing van een machine is het wenselijk om de functionaliteit van dit model direct te kunnen overdragen op een besturingsprogramma voor de fysieke machine. Het probleem hierbij is dat het model implementatie-onafhankelijk is en aangepast dient te worden om te kunnen functioneren in een real-time omgeving. Er moeten een aantal toevoegingen gedaan worden aan het model om, met het model als uitgangspunt, een besturingsprogramma te kunnen genereren voor de werkelijke machine. Dit project onderzoekt welke deze toevoegingen zijn, wanneer ze moeten worden toegevoegd, en door wie dit gedaan moet worden. Aan de ene kant is er de modelleur, die geen of nauwelijks kennis heeft van het fysieke systeem, aan de andere kant de implementator, die niet of minder op de hoogte is van de functionaliteit die in het besturingsmodel beschreven is.

In dit rapport worden eerst de besturing en computer interfaces in zijn algemeenheid bekeken. Daarna komt het gedrag van real-time programma's aan bod en welke eisen er gesteld worden aan de communicatie met actuatoren en sensoren. Er worden verschillende manieren aangedragen om deze eisen te beschrijven in het χ-formalisme. De verschillende mogelijkheden worden hierna onderzocht in een kleine case waarna op basis van de resultaten één van de mogelijkheden wordt uitgekozen om verder mee te experimenteren. Hoe deze mogelijkheid is geïmplementeerd wordt hierna uitgelegd. Vervolgens wordt beschreven hoe een gebruiker om dient te gaan met de nieuwe χ-kernel. Aan bod komen nog enkele onderwerpen waarmee tijdens toekomstig onderzoek rekening dient te worden gehouden. Tenslotte volgen de conclusies en aanbevelingen.

De appendices bevatten een introductie in het interface-systeem InterBus-S, dat tijdens dit onderzoek gebruikt is, en de code van de programma's die voor dit onderzoek zijn geschreven.

Summary

At the Section Systems Engineering of the Eindhoven University of Technology, the Netherlands, a specification language, called χ, is being developed. With the specification language χ it is possible to model industrial systems. (These industrial systems can be either machines or complete plants.) A model of a machine control system contains a part in which the machine control is modelled. It is desirable to be able to transfer the functionality described in the model directly to an executable program that can be used as control program for the physical machine. The problem that arises is that the model is completely implementation independent. The model must therefore be adjusted in such a way that it is able to function in a real-time environment. To generate a control program from the model, a number of additions to the model is required. In this project the necessary additions are determined, when they should be added, and by whom these additions should be made. On the one hand there is the modeler, with his limited knowledge of the physical system, and on the other hand there is the implementor, with his limited knowledge of the functionality of the control system as described by the model.

This report first examines the control program and its interfaces in general. Then, the behavior of real-time systems is examined, as well as the demands that are made on the communications with actuators and sensors. Different ways to incorporate these demands into the χ-formalism are presented. Hereafter, these different methods are presented in a small test case. The results of this test case are then used to choose one method that is incorporated into the χ-kernel. The way in which the kernel is adjusted to create a χ-kernel using the chosen method is explained. How a user should operate the new kernel is explained. Several subjects that are to be considered in further research are mentioned after which the conclusions and recommendations are presented.
The appendices contain an introduction to the interface system, InterBus-S, that is used during the experiments, and the source code of the software that is written for the experiments.

Contents

Samenvatting (in Dutch) **v**

Summary **vii**

1 Introduction **1**

1.1 Purpose of the Report 1

1.2 Extent of the Report 2

2 Physical Implementation of Control Systems **3**

2.1 Control System 3

2.1.1 Control Programs 5

2.1.2 Computer Interfaces 6

2.1.3 Standard Interfaces 9

2.1.4 The Test Interface for χ 10

3 Interfacing and χ **11**

3.1 Real-Time Program Structure 12

3.1.1 System Initialization, Reset and Shut-down Procedure 13

3.1.2 The User's Program 14

3.2 IO-Communication Behavior 15

3.3 IO-Communication Requirements in χ 15

3.3.1 Continuous IO-Communications 16

3.3.2 Discrete IO-Communications 17
3.4 Testing of IO-Signals in χ 17
3.5 Possible Means for IO-Communications in χ 19
3.5.1 IO-process in χ 20
3.5.2 IO-variables in χ 23
3.5.3 IO-channels in χ 25

4 Case Study: A Piston Control 29

4.1 A χ model 29
4.1.1 Types 31
4.1.2 System Definition 31
4.1.3 Process *Op* 32
4.1.4 Process *C* 32
4.1.5 Process *Ph* 33
4.2 IO-Communication in the χ Model 35
4.2.1 IO-process 36
4.2.2 IO-variables 37
4.2.3 IO-channels 39
4.3 A Choice for an IO-Mechanism in χ 40

5 Interfacing χ with Hardware 43

5.1 The χ Formalism in Hardware Control 43
5.2 The Experiment, IO-communication in χ 45
5.2.1 Combining the χ-Kernel with DOS Driver Software 45
5.2.2 The Real-Time χ-Kernel 45
5.2.3 Initialization and Shut-Down Procedure of the Interface System 53

6 Using Real-Time χ **55**

6.1 Restrictions to Real-Time χ 55

6.2 Creating a Control System with Real-Time χ 56

6.3 Test Results with Real-Time χ 57

7 IO-Communications Incorporated in the χ Kernel **59**

7.1 Commonly Used IO-Mechanisms 59

7.1.1 Polling 60

7.1.2 Interrupt-driven IO 60

7.1.3 Mailbox 60

7.2 Suggested IO-Mechanism for the χ Kernel 61

8 Real-Time Aspects of χ **63**

8.1 Time and Timing in Real-Time χ 63

8.2 Error Handling in χ 64

8.3 Concurrency in χ 65

9 Conclusions and Recommendations **67**

9.1 Conclusions 67

9.2 Recommendations 68

Bibliography **69**

A Introduction to the InterBus-S System **71**

A.1 The InterBus-S System 71

A.1.1 General Structure and Method of Operation of InterBus . . . 72

A.1.2 InterBus Protocol Sequence 72

A.2 Process and Parameter Data 73

A.2.1 IBS Data Format: Motorola versus Intel 73

A.3 InterBus Topology and Data Addressing 74

A.3.1 InterBus Topology 74
A.3.2 InterBus Data Addressing 74
A.4 Working with InterBus-S 78
A.4.1 Utilities for InterBus-S 79
A.4.2 Start-up of the InterBus-S System 79
A.4.3 Shut-Down of the InterBus-S System 80
A.5 InterBus-S Documentation 81

B Library Text **83**
B.1 Library Source Code 83
B.1.1 IBSCHI.H 83
B.1.2 IBSCHI.CPP 84
B.2 Additional Utilities 90
B.2.1 Ibsutil.h 90
B.2.2 Ibsutil.cpp 91

Chapter 1

Introduction

At present, machines become more and more complex. As a result, the demands placed upon the machine control systems also increase. The design and development of these control systems, therefore, is a task that can no longer be considered to be separate from the design process of the machine itself. When dealing with complicated control systems it would be preferable to be able to validate their performance before they are implemented. This can be achieved by modelling the control systems and their controlled systems. These models can then be used in simulation runs.

Various languages have been developed to design and validate models of machines and control systems. Most often, these languages were developed with a specific field of implementation in mind (for instance, *gProms* for continuous systems), and they were able to specify either continuous systems or discrete systems. Hybrid system design, for those systems in which both continuous and discrete parts are present, is supported poorly at best.

At the Eindhoven University of Technology, the Netherlands, in the section Systems Engineering, a specification language χ is being developed which is able to specify continuous systems, discrete systems, as well as hybrid systems. The language χ is already being used to specify and validate industrial plant control systems. For machine control systems, many questions are yet unanswered.

1.1 Purpose of the Report

The purpose of this report is to examine the ability of the language χ to be used for machine control systems. Not only does it examine whether χ is capable to specify

machine control systems, it also presents the means to communicate with a real-time system.

Examined must be if it machine control systems can be specified in χ. If so, is it then possible to generate a control program for a physical system? Problems that arise are how the control system is to communicate with its environment and if this communications can be executed real-time.

1.2 Extent of the Report

By no means is this work meant to be a manual on real-time control systems and implementation. As the area of interest of this work is the functionality of control system specifications in χ, most technical characteristics of processors, sensors and converters are not discussed. Most often the components that are combined to form a control system are considered to be 'black boxes'. If information about a component is vital to the understanding of its functionality, this information is, of course, presented.

Chapter 2

Physical Implementation of Control Systems

Before being able to implement a control system, it is necessary that one understands the various components of an industrial system. In Figure 2.1 the structure of a physical system and its control system is presented as defined by [Lei92].

In this system 3 different areas can be distinguished.

- At the bottom, there is the physical system. This system is the actual machine or combination of machines that is to be controlled.
- Above this, the control system operates. It is this part of the system that is examined in detail in this report. The formalism χ is used to specify the functionality of this part.
- A Man-Machine Interface (MMI) is linked to the control system. If one is not able to give orders to the control system as well as to monitor its progress, a control system is useless.

Of the three areas mentioned above, only the control system is considered. The other two are only mentioned when their relation with the control system is examined.

2.1 Control System

The central part of Figure 2.1 is the control system, which usually consists of a PC or an embedded system. The interest of this paper lies in the software description

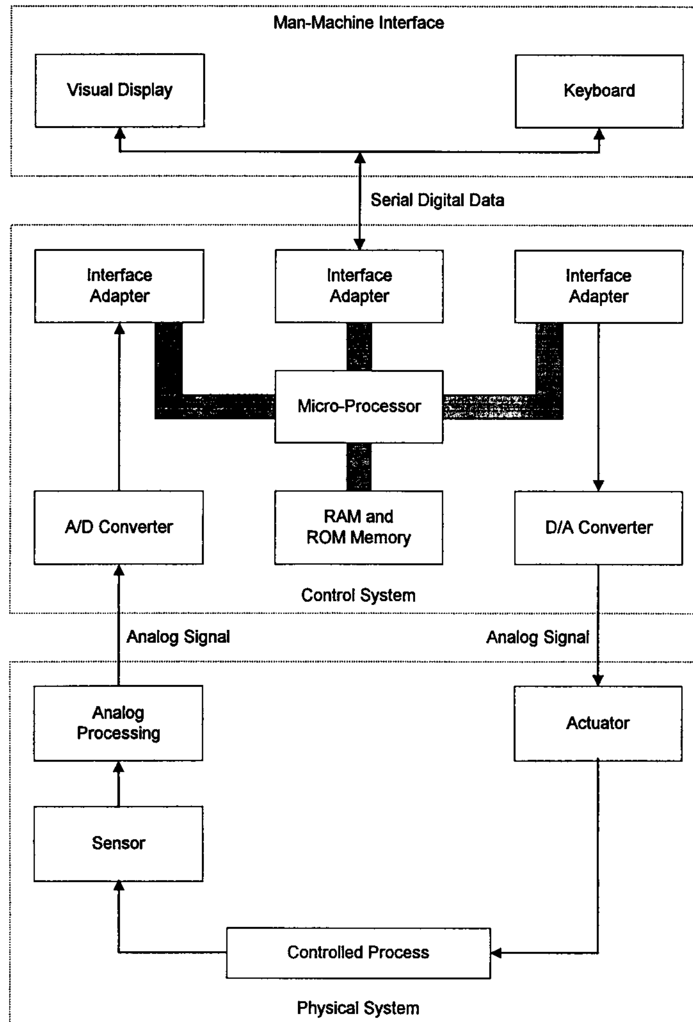


Figure 2.1: A Controlled System [Lei.1992]

of the control program. Within the software, the functionality of the control system has been defined using a programming language. The hardware that the processor is made of and that allows it to communicate with its environment is not examined in this work (for further information on processors, see [Hor85]).

This paper examines the possibility of using the formalism χ in a control system operating in a 'real-world' environment. The functionality of the control system is defined in the formalism χ. The specification in χ is then used as a basis from which the control program is generated.

2.1.1 Control Programs

Demands on control systems can vary greatly. Nonetheless, it is possible to distinguish different sorts of control systems. Three different sorts of control programs are presented in [Ben94]. These are *sequential programs*, *multi-tasking programs* and *real-time programs.*

- *Sequential programs* are strings of actions ordered in time. The performance of a sequential program depends on the effects of the actions and the order in which these actions are made. The time needed to perform an action is of no importance. The logical correctness of a sequential program is therefore ensured when the actions are defined correctly, and the order in which the actions are taken is the right one.

- *Multi-tasking programs* are programs in which the actions are not necessarily disjoint in time, though the order in which the actions are done is still important. A multi-tasking program is built from separate parts, often called processes, which are executed concurrently. These processes can themselves be partly sequential. The processes communicate with each other through shared variables or synchronization.

- *Real-time programs* differ from the previous two in that the sequence in which actions are performed is not completely predetermined, in addition to these actions being not necessarily disjoint in time. The sequence in a real-time program is not determined by the programmer but by the environment in which the program is embedded. Extensive communication is necessary between the program and its environment to decide on the course of action. Actual time is of great importance in real-time programming, as certain requirements specify timing aspects. The most common timing requirement in real-time programs is a guaranteed maximum response time.

This report's main interest lies in real-time applications. These applications operate under certain timing requirements. These requirements can be divided into two categories: *hard* real-time timing requirements and *soft* real-time timing requirements.

- *Hard real-time requirements* have to be met for an application to be able to function correctly. An example of a hard real-time requirement is an emergency shutdown procedure of a machine. If the activation of an emergency stop is detected, the machine must be shutdown within a certain time period. If this takes too much time, this could lead to catastrophic malfunction.

- *Soft real-time requirements* are more like a preferable time in which an action should be completed. If a cash machine is not able to deliver its money within 20 seconds because of heavy telephone traffic, this does not result in a catastrophic malfunction.

To be able to meet hard real-time requirements demands a very precise knowledge of the hardware and software being used both in the control system as well as in the physical system. In this report only soft real-time requirements will be considered.

2.1.2 Computer Interfaces

As is already mentioned in the previous chapter, a real-time application depends heavily on the ability to communicate with its environment. This communication is established through a multitude of sensors and actuators; they may be used to measure temperature using thermocouples with voltage signals, or they can be used to measure flow, generating pulse signals. There are about as many different signals as there are sensors and actuators. To design a different interface for every application would be costly and a lot of work. Most interfaces are part of one of the following categories:

- *Digital interfaces* can be 1 bit interfaces, like a switch being open or closed, or they can be several bits wide, like the digital quantities from the signal of a digital voltmeter in BCD (Binary Coded Decimal).
- *Analog Interfaces.* Some sensors, like thermocouples and strain gauges e.g., have voltage output signals. These outputs are then amplified to the range of -10 to 10 V; conventional industrial instruments often use current outputs in the range of 4 to 20mA. (A current signal is less sensitive to interference than a voltage signal.) All these signals have in common that they are continuous signals. All these signals must therefore be sampled and converted to a digital value.
- *Pulse* and *pulse train interfaces.* Many sensors, particularly flow meters, produce inputs in the form of pulses or pulse trains. The increasing usage of stepping motors also increases the need for step output.

To divide interfaces into the above groups can help to limit the work that has to be done to design an interface for a particular process. Common practise for hardware producers is to provide a variety of interface modules each for a specific type of interface. These modules are then combined to provide the right interface for a certain

process. For example, the interface for a process containing a large amount of flow measurements would be made up largely from modules made for pulse interfacing.

In this report, only single bit digital interfaces are considered. This interface is the least complex of the interface types mentioned. Also, only the discrete modelling in χ has been completely tested and is fully operational. The other interface types are considered to be beyond the scope of this paper.

Digital Interfaces

If requirements for interfaces in the formalism χ are to be defined, one must first examine the hardware operations of such an interface. As mentioned before, in this report only digital interfaces are considered. In this chapter, an output and an input interface are presented.

Digital Input Interface A simple digital interface is shown in Figure 2.2, as defined in [Ben94]. The outputs of the physical system are considered to be logical signals. It is usual to transfer one word at a time to the digital input register. The input register will normally have the same number of lines as is needed to transfer one word. The logic levels of the signals that are transmitted are 0 and +5 Volts. If this is not the case, some sort of signal conversion will be required.

The physical system provides data that is put in the digital input register. The CPU signals which register may put its contents on the data bus by selecting the address of this register. To do so, the CPU uses an address decoder. After a particular register is selected, it must first receive confirmation before it can transmit its contents to the data bus. This is done to prevent the possibility that data from the register might corrupt information already present on the data bus. This conformation is given by an 'enable' signal from the control bus, after which the register copies itself on the data bus. Information in the register is continuously updated by the output from the machine.

Note that this input interface only presents data upon request of the CPU. It does not signal if new data is presented to its input register. To achieve this, interrupt abilities should be added to the interface and the control program. As these interrupts are not yet available in the formalism χ, this option is not considered.

Digital Output Interface A digital output interface is presented in Figure 2.3, as defined in [Ben94]. Again, the outputs of the output register are logical signals, with levels of 0 and +5 Volts. If these are not able to control the physical system,

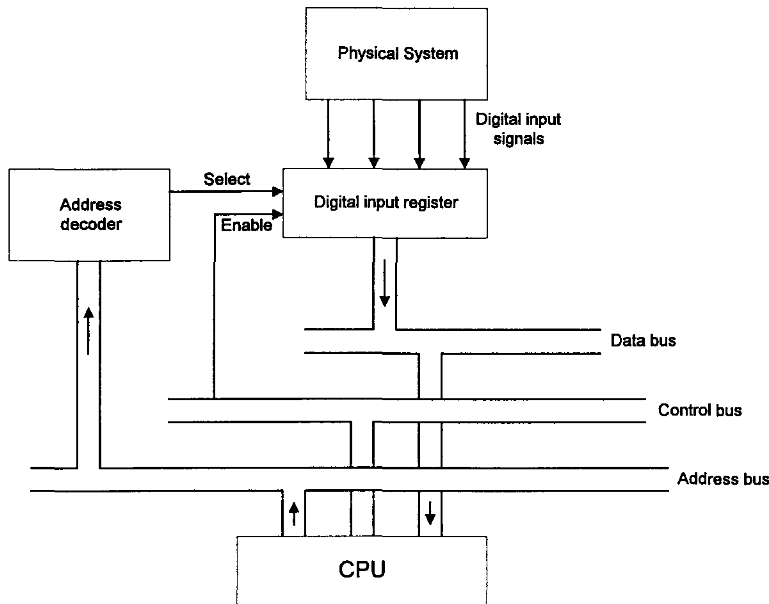


Figure 2.2: A Simple Digital Input Interface

signal conversion is needed. This conversion is achieved by sending the low-voltage signals to relays that switch high-voltage signals; an additional gain of this procedure is that a separation is made between the electrical components of the control system and the physical system.

All that is needed for an output interface is a register in which the computer can store information before this data is sent to the physical system, or the physical systems demands this data. The output register reads the data presented on the data bus. This only occurs if a particular register is selected to read the data bus. A register is signaled by an address encoder to select it. After the selection, the register must wait for confirmation by an enable signal from the control bus. This conformation is given only when the control bus detects that the data on the data bus is stable.

Interface Requirements

If the operation of the input and output interfaces presented in the previous chapter is considered, some interface requirements can be distilled. First, of course, there must be a notion of what is to be transmitted or received. This information is provided by the model that is made of the system.

Secondly, the computer must be able to transmit this information at the correct moment. At the level of the programming language, this decision is made by the

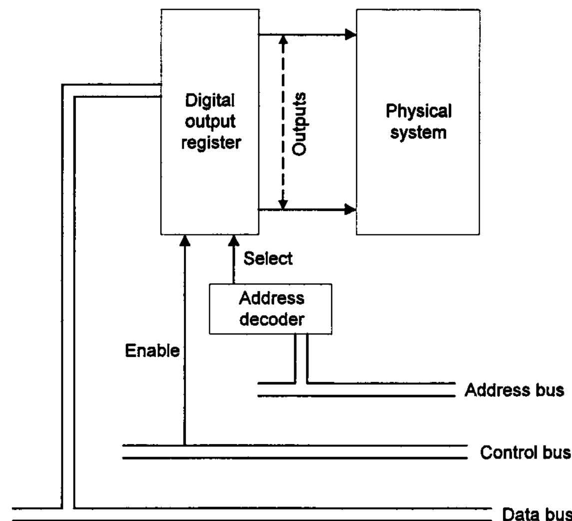


Figure 2.3: A Simple Digital Output Interface

control program, in this case written in χ. At the level of computer operations, the timing is achieved by the internal timing mechanisms of the computer's CPU as explained in previous sections.

Thirdly, the CPU must have an idea of where this communication is to take place. It is this requirement that poses a problem. Before, it has not been necessary in the χ-formalism to incorporate means to define hardware architectural information in the model. The reason for this is that, until now, the only models that have been made in χ were models of plant control systems. What was used of these models was the data that the simulations from the models of the systems yielded. As the plant that was modelled is not controlled by some all-encompassing control program, no electronic equivalent of these models existed. This is not the case with models of machine control systems where a electronic equivalent of the model is present by definition. This equivalent is the program that is used to control the machine. It would be desirable to be able to translate the functionality defined and tested in a model directly into an operational control program. How to do this is not explored in this chapter, but solutions are presented in Chapters 3, 5 and 6.

2.1.3 Standard Interfaces

Most companies that supply computer hardware and software for real-time control systems have developed their own 'standard' interfaces. Although a supplier is able

to support these systems with a wide variety of interface cards, the problem with interfaces from different suppliers is that they are not compatible.

An attempt to define a standard interface was made by the British Standards Institution (BS 4421, 1969). Unfortunately, this standard only defined the concept of how the components of an interface should interconnect. It did not standardize the hardware of the system. This standard was quickly overtaken by more recent developments.

In the early 1970's, Hewlett Packett developed a General Purpose Interface Bus (GPIB) to connect laboratory equipment to a computer. This bus was taken by the IEEE as their standard and adopted as the IEEE 488 bus system.

The International Organization of Standardization (ISO) has defined a standard protocol system in the Open System Interconnection Model (OSI). This is a hierarchical, layered model containing seven layers from the physical connection to the highest computer protocol. The standard of the ISO is an architecture and by no means unambiguously defines the interfacing of system components.

2.1.4 The Test Interface for χ

Because standard interfaces do not exist, an interface has to be chosen to be used for testing of the control programs resulting from the χ models. This choice for an interface has been an arbitrary one.

The test interface used in this report uses InterBus-S interface system, produced by Phoenix Contact. The InterBus-S interface system is a bus system that is independent of the control program. The connection between the InterBus-S system and the control system is made by a special host controller board for every control system, or, in the case of a connection to a PC, by an interface board. The controller board used in this particular project is the IBS PC ISA SC controller board, also designed by Phoenix Contact. A detailed description of this system can be found in Appendix A.

Chapter 3

Interfacing and χ

If χ is to control a real-time system, some form of interfacing is needed between the control system designed in χ, and the controlled system. The χ language, at this moment, does not support any possibilities to communicate with its environment. One should see to it, when these possibilities are integrated into the χ language, that no implementation-dependent code is added to the language. The χ language should be kept implementation-independent because the χ-formalism is used to specify functionality of a system, not its implementation. This aspect of the χ formalism should be maintained. The main problem, therefore, is to define a set of IO-functions without a direct link to a specific implementation. These IO-functions must be compiled according to the needs of a specific interface system. The definition of these IO-functions will be done in an extra layer between the control system (specified in χ) and the controlled system, being any physical machine or construct. This extra layer, called *control layer*, must be designed for each individual interface system (see Figure 3.1).

Before an implementation of the means for IO-communications in χ can be presented, it is necessary to define what functions are needed. To do so it is necessary to look at IO-communications, where IO-communications occur, what its behavior is and what is required to implement IO-possibilities in the χ language. The experiments of this project use only discrete communications. In the discussion how to incorporate IO-communications in the χ formalism, both discrete and continuous communications are considered. This is both easier and more complete. When the results of this examination are to be incorporated into the compiler, only the discrete part is implemented. The continuous part lies beyond the scope of this paper.

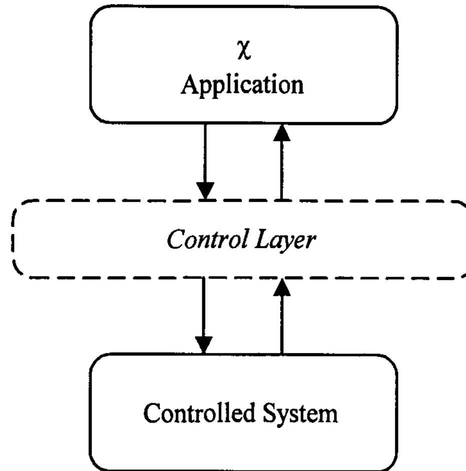


Figure 3.1: Application and Controlled System connected through a Control Layer

3.1 Real-Time Program Structure

A real-time control program can be divided into several parts. Some parts are almost exclusively defined by the functionality described in the χ model. Other parts are specified largely by the choice of the interface system that is used to communicate with the controlled system. To establish which parts of a control program are influenced by the χ model, it is necessary to have an idea of the structure of a real-time control program.

Three different parts can be distinguished in a real-time control program:

- A system initialization procedure determines the system configuration and recognizes communication partners and peripheral systems.
- A user program introduces the functionality that is to be enforced upon the real-time system. In this part of the program, the functionality of the χ model is integrated.
- A shut-down procedure ensures that a shut-down of the system is achieved in a controlled manner. A reset procedure for the system might also be incorporated in this part.

In each of these sections, IO-communications may occur. These parts are explained in more detail in the next sections.

3.1.1 System Initialization, Reset and Shut-down Procedure

First, it should be made clear what is meant by these terms in this context. System initialization is used with respect to the interface system and the software that is used by this interface system. In the system initialization one will only find initialization procedures of the interface system, and computer software and hardware. These procedures check hardware configuration, software requirements, etc. The initialization of the physical system has to be part of the user's program.

The same goes for the term shut-down and reset procedure. The shut-down procedure ensures a controlled shut-down of the interface system, its software and embedded processors at a machine-language level. The reset procedure takes the system into a state from which it is again operational after an error has occurred. Again, the shut-down and reset procedures for the physical system are a functional part of the user's program.

Both system initialization and shut-down are performed at machine-language level. The actual procedures, therefore, depend on the interface system and the software which are used. This means that these procedures are, by definition, implementation dependent. To implement the functionality of these procedures in a χ program would be incorrect. This undermines the implementation-independent character of the χ formalism. Also, most interface systems available have predefined functions that execute a complete IO-initialization or shut-down without the help of the user. In other words, as far as the user is concerned, in these two parts of the program no IO-communication is needed.

However, the notion that the IO-system is to be initialized before use, and shut-down in case of an error must be incorporated into the executable generated from the χ model that is made of the control system. This initialization can be an explicit function call in the χ text or it can be inserted automatically when the model is being compiled into an executable program. The precise contents of the procedures is left to the implementor.

The modeler must also consider the hardware requirements of his model. The number of necessary IO-positions must be stated as well as the timing requirements which the implementation must meet. This is an area in which the modeler, with his limited knowledge of computer and embedded hardware, must consult the implementor on a regular basis. Precisely which decisions are left to the modeler and which to the implementor is a problem that is dealt with later in the report.

3.1.2 The User's Program

The user's program is the main interest of this chapter. The functionality of this part of the control program is directly derived from the functionality of the χ model.

A model in χ of a machine control system usually consists of two parts. One part describes the model of the physical system. The other part is a description of the model of the control system operates. The two parts are linked together to form a closed model with which simulations and calculations can be performed.

If a real-time control program is needed, the model of the control system is the only part of the model that is of interest. In this report, this part will be called C. If the model is specified correctly it holds all the functional information needed to operate the physical system. The model of the physical system, Ph, is deleted from the model as it is replaced by the actual physical system (see Figure 3.2).

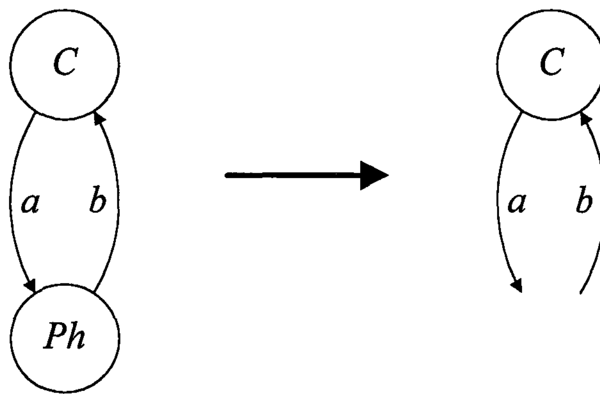


Figure 3.2: Model of Control System

When the model of the control system is uncoupled from the model of the physical system a clear problem arises. The separated model of the control system is an open model. The current χ-kernel cannot compile open models.

As it is obligatory for compilation that the model is closed, the compiler must be redesigned so that it is able to compile open models. Alternatively this open model must be adjusted in such a way that the compiler views it as closed. The means that might be used to do this are presented in Section 3.5.

3.2 IO-Communication Behavior

Before a specific solution is presented to incorporate IO-communications in the χ formalism, the behavior of IO-communications is examined to allow definition of the requirements of these IO-communications.

Communication is a transfer of information between tasks or processes. Usually, if a message is sent between two different tasks or processes, a confirmation is generated by the receiver that the signal has been received. Also, the sender of the signal waits for this confirmation before proceeding with the rest of his program. This form of message sending is called *handshaking*. A sender can only complete his communication when a receiver is ready, and capable to receive the message. In the χ formalism, transfer of discrete signals is organized in this way.

IO-communications behave differently. First, processes are not always in direct contact with the environment. Communication with the environment is established through various hardware and software tools. In this case, it is not always the initial sender that waits for the confirmation from the peripheral device. Instead, this handshake is made by a process or a tool closer to the device. The process initiating the communication does not receive a handshake for its signal.

Second, some physical devices do not generate a handshake. A sensor always creates new data and stores this in an address in the computer. Whether or not a process is using the data, is of no concern to the sensor. The stored data is always available but the sensor itself will not make an active attempt to present this data to other processes. Some actuators behave in a similar way, but the other way around. These actuators are always able to receive commands. No confirmation, or handshake, is generated. Instead, checking if the actuator executes the command correctly is done using sensors to detect a desired change in the state of the physical system.

3.3 IO-Communication Requirements in χ

What is said in the previous sections, is used to define the requirements of the functionality of IO-communications in the χ formalism. If the precise form of the signal and the address to which the signal is to be sent are not considered, it can be said that:

1. IO-communications must always be possible. This means that, if some output is sent to an actuator this actuator must always be able to receive this output. If a process requires input from a sensor, this input must always be available.

It must be possible to test the value of an input variable, especially in the case of sensory input.

2. The IO-communications must not rely on the hand-shake principle. This means that a sender of a signal does not have a direct means to test if the signal is received. Instead, the verification of the output is done by checking sensory input from the physical system. Also, a control process must be able to pull the input it needs. Another process is not explicitly presenting this input to the controller. The control process must receive this input without an apparent sender at the other end.

These two requirements must be met to properly deal with IO-communications in the χ formalism.

3.3.1 Continuous IO-Communications

There is in fact already an object, continuous channel, in the χ formalism that meets the requirements mentioned above. Literally spoken, this channel does not transfer data. This channel sends continuous variables. By doing so, it connects two equations containing the same unknown variables from different processes. A process is then able to solve an equation that contains unknown variables by enlarging the number of different equations in which these variables are present. The variable that is 'sent' over a continuous channel can better be seen as a variable that both processes connected to the continuous channel can read and influence. A process can, at any time, read and test the value of this variable. Also, a process can at any time change the value.

The problem with using the continuous channel as a means of IO-communication is the functionality of the continuous channel in χ. Computer communications are mostly discrete. A discrete signal could be faked by describing it as a continuous signal with discrete event changes. It would not be correct to use continuous channels in this way simply because they behave according to the requirements while, in fact, they are not related to the sort of signal that is specified. Also, if a continuous channel is used to specify a discrete signal, the compiler adds a large amount of unnecessary code to the executable program. This code has to do with solving equations, integration formulas, and so on. All this code is not needed for IO-communications and does not make it possible to communicate with the environment. Therefore, it is necessary to create a new kind of object in the χ language to make IO-communications in specifications of machine control systems possible.

In addition to the requirements mentioned, the process that wants to communicate must know where the other partner can be found. In other words, if a process wants to share information with peripheral equipment, it must know where the corresponding computer address is situated. This is implementation-dependent information and should not be visible in the model of the control system.

Again, the specific form of the signals sent to the pheripheral devices is not discussed in detail in this paper. In practice, sensors and actuators are bought as packages from a supplier. These packages can be made to support any signal that is desired. Thus, no standard can be assigned to the signals that are used by these IO-devices. The signals are adjusted according to the wishes of the implementor.

3.3.2 Discrete IO-Communications

The discrete communication in χ requires the most extensive adjustments. The discrete channel in the current χ formalism depends on the hand-shake principle. As has been explained in the previous section, this dependency must be removed if IO-communications are to occur using this channel. The only thing that remains, therefore, of the discrete channel that is currently used in χ is the discrete form of the signal that is sent over this channel.

A discrete output generated by the control system must be sent to the environment without waiting for a hand-shake from a receiver. This send action must always be possible. The signal that is sent is stored in a computer address and left there for other processes to use. Thus, the only difference between this channel and the discrete channel as it is currently being used, is the fact that it operates without a hand-shake.

A discrete input is more difficult to achieve. If the control system needs an input, it must be able to pull this input from a specific computer address. This communication is not initiated by an other process, as would be the case when the communication depended on hand-shaking. The control process itself must be able to initiate this communication.

3.4 Testing of IO-Signals in χ

A control system is acting on signals from an environment. Thus, the control system must be capable of monitoring the value of the input signals it receives from the environment. In most χ models, channels are used to transfer data and synchronize two processes, using the hand-shake principle that communication depends on in the

χ formalism. In a real-time environment one often encounters communications of which most instances are ignored. For instance, a certain sensor might measure the heat in a fermentor once every two minutes. Only when the temperature reaches critical levels, the controller will act. All other instances of the measurement are ignored by the controller. Also, a controller might want to wait for a specific change in the state of the physical system to occur. An other example is a large reactor reservoir that is heated to reactor temperature. As this takes much time, the controller checks the temperature occasionally.. When the temperature has reached the prescribed level, the program is continued.

In both these cases, the controller must have the ability to test the signal and wait for a specific state change to occur before continuing the program. There are two options. Either the designer of the model can code this testing in the model, creating some sort of polling in the χ code, or a construct must be designed that is able to test an IO-signal for a specific change in the system state. To illustrate both possibilities an example is presented.

Consider the discrete variable a. This variable is sent over the discrete channel b. The control process wants to wait until the variable reaches the value 10 or higher. Polling as a means to monitor an input signal can be achieved with the following χ text.

$$
\begin{array}{l}
\ ;\ b\,?\,a \\
*[\ \neg(a \geq 10)\,;\quad b\,?\,a \quad \longrightarrow \text{skip} \\
\ ]
\end{array}
$$

A major disadvantage of the polling option is that it increases the workload of the main program. In χ specifically, a polling statement causes even more problems because the program will lock in the polling statement as long as the boolean expression remains true. (Under the assumption that communication is timeless and non-blocking.) This causes all other tasks of the same process to be suspended and possibly resulting in control errors.

The other option the current χ formalism offers, is the use of the ∇ operator. The advantage of the ∇ operator is its simplicity, and the fact that it uses less program text. This operator is already available for continuous variables. To be able to use it with discrete input signals, the functionality of this ∇ operator must be expanded. A major difference exists between continuous and discrete communication in χ. A continuous channel always has a continuous variable linked to it. A discrete channel is not attached to a discrete variable. Thus, if a ∇ statement is used to test a discrete variable, it must either be specifically stated over which channel this variable is received, or the ∇ operator must be used directly on the discrete channel itself.

The functionality that is to be added to the ∇ operator would be to receive any

signal over the discrete channel it is connected to, to test it against its guard, and then either continue with the program if the guard evaluates to true or wait to receive the next signal. This would create a compound statement like the one below:

$\nabla(b\,?\,a, a \geq 10)$

In this compound statement, a variable a is received over channel b. If the guard $a \geq 10$ evaluates to true, the variable a holds the value received over channel b, that had caused the ∇-operator to respond. The variable a can then be used for further computations.

A disadvantage of the ∇ is the fact that the current χ engine used to test the functionality of the model of the control system through simulations does not support such use of the ∇ operator on discrete channels. The kernel must be adjusted to allow the ∇ operator on discrete channels if the coding out of polling is to be prevented in the χ code. Needless to say, it is impossible to do without polling. On some level polling will always take place. It is likely, however, that this polling can be made more efficient if implemented at a lower level than the χ formalism.

3.5 Possible Means for IO-Communications in χ

In this section, the means for IO-communications in χ mentioned in the previous sections are further examined. As has been explained, a control system must be able to communicate with its environment. To do so, the control system must be connected to the physical environment.

The functionality of communication methods in the current χ kernel differs from the functionality of the means for IO-communication. Different forms of communication must therefore be added to the χ formalism. This also means that one must be able to distinguish the methods for IO-communication from the communication between processes and systems within the control system. This is to avoid confusion over the specific form of communication that is used.

There are three alternatives to connect the control system with the environment.

- First, it is possible to define an additional process, which is called an IO-process, in which the connections to the 'real' world can be defined. This IO-process is added to the model of the control system.
- Second, the model of the control system can be considered as a separate model. Places in which the control system needs input from or writes output to the

environment are connected directly to this environment. No channels are used as connection.

- Third, the unconnected channels of the model of the control system can be connected directly to the various addresses of the PC or embedded system that is used to run the controller. These direct connections are then made in the connecting system of the model. There are no means to indicate this graphically. Graphically, the model remains open unless an addition is made to the graphical representation of the χ systems. This addition must define the IO-channels.

Each of these possibilities are examined in the following sections. The programs that are presented in these paragraphs will contain variables, especially those concerning addresses, that are not defined or initialized.. The reason for this is that, as yet, the specific addresses are unknown. If the model is compiled, specific addressing is not necessary.. If the executable generated by the compilation and linked is run, a separate data-file can be included specifying the addresses. The means to initialize, shut-down or reset the system are not integrated into the models at χ level. These actions are inserted while compiling.

3.5.1 IO-process in χ

As mentioned in the previous chapter, an additional process, called IO, can be defined to accomplish IO-communications. The function of this process is to connect the model of the control system with a real-time system. This real-time system might be a PC or an embedded system. The necessity of this IO-node arises when the model of the control system is uncoupled from the model of the physical system. This results in an open system. If an IO-process is added the system is closed (see Figure 3.3).

Consider the functionality of this process. For the control system, this IO-process is a black box representing the actual physical system. The channels that run from and to the control system are connected to this node. Over these channels signals are sent to actuators and signals are received from sensors. Within the IO-node the connection is made between these signals and the computer addresses that the signals must be sent to.

The above would result in the following χ code for the IO-process.

proc $IO(a : ?\,\mathrm{int},\ b : !\,\mathrm{int})=$
$[\![\ io_1, io_2 : @\mathrm{int}$

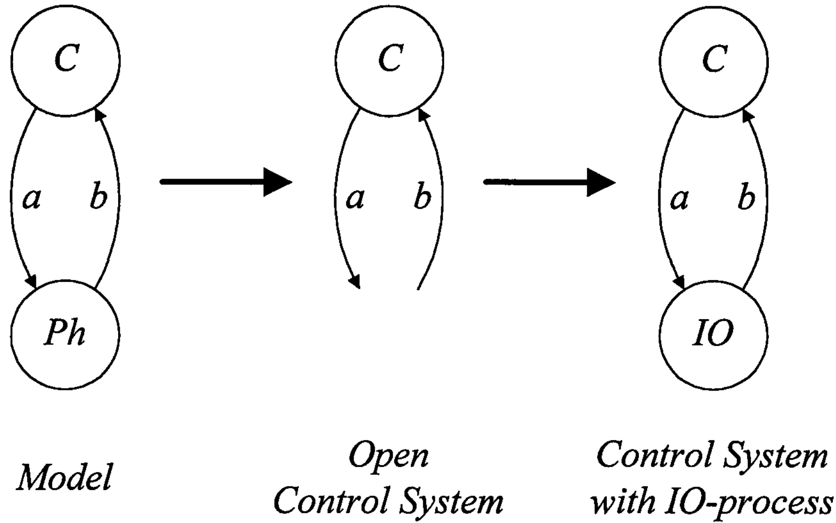


Figure 3.3: IO-Process added to Control System

```
| *[ a?io1 ⟶ skip
   [] b!io2 ⟶ skip
   ]
]|
```

In the IO-process the channels to and from the model of the control system are linked to specific computer addresses. The connection between the channels and the computer addresses is made by introducing a special kind of variable called the IO-variable. An IO-variable is distinguished from a normal variable by the addition of the symbol @ to its type definition. To explain the functionality of the IO-variables, the IO-variables are best viewed as pointers to a specific computer address. If an action is performed on the IO-variable, it is actually being performed on the value of the computer address the IO-variable points to. As is said before, the connection of these IO-variables with their corresponding computer addresses is done by the implementor when the executable is created.

The statements in the IO-process have the following meaning. The statement $a\,?\,io_1$ means if a signal is received over channel a, then write the value of the signal received to the address io_1. The statement $b\,!\,io_2$ tries to send the most up-to-date value found at address io_2 to the control process. The IO-process will never exit its repetition loop because the IO-process is always ready to send signals to the control process.

When the control system wants to send a command to an actuator using channel a, the IO-process executes $a\,?\,io_1$. The value at a computer address is adjusted

according to the value received over channel a by the IO-process. An actuator then uses this information and acts accordingly.

Considering the example given above, the process IO would behave as follows. In the body of the IO-process, the program has the option of executing one of two alternatives. If the control system requests the value of an input variable over channel b, process IO executes the statement $b\,!\,io_2$. The value that is sent over channel b is the latest update of the computer address that io_2 is pointing to. The value of the address is being generated by a sensor.

This concept works for both discrete and continuous channels between the control system and the IO-process. To give an example that uses a continuous channel, consider the example above, with this difference that the channel b is now a continuous channel of the type $[m/s]$. This results in the following code for the IO-process.

$$
\begin{array}{l}
\mathsf{proc}\; IO(a : ?\,\mathrm{int},\; b :\; \multimap [m/s]) = \\
[\![\; io_1 : @\mathrm{int},\; io_2 : @[m/s] \\
\mid b \multimap io_2 \\
\mid *[\; a\,?\,io_1 \longrightarrow \mathrm{skip} \\
\quad] \\
]\!]
\end{array}
$$

An IO-process has the following advantages. First, it demands minimal adjustments of the χ compiler. Only one new language item is introduced. This is the IO-variable used in the IO-process. This variable has a type definition preceded by a @ to identify it as an IO-variable. The declaration io_1 : @int is read as 'the variable io_1 of type *at* integer'. The adjustments that have to be made to the χ-kernel are rather straightforward. These IO-variables can be of any type. The functionality of the IO-variable linked to a channel is explained above. The IO-process also ensures that all IO-connections are made at one single place in the model. This simplifies the setup of the model. A second advantage is the fact that the processes used in the simulation of the model are the same as the processes used to define the control system.

A disadvantage is the fact that a process is used to connect the control system to the environment. In the χ formalism the system definition is used to define the various connections of the system, both between processes and between the system and its environment. (In the case of the current χ formalism, the environment of a specific system consists of other systems.) As all connections inside a model are made by the system definition it is logical that all the connections to the environment should also be initialized in this part of the model. Another disadvantage is the extra work that must be done. Of course, the system definition must be rewritten. This has to be

done anyway, as the model of the physical system is deleted from the system. But besides this, a new process, the IO-process, must be defined. This might seem easy in a small example. Remember, however, that industrial systems with hundreds of IO-connections are not uncommon.

3.5.2 IO-variables in χ

This option does not use any additional processes. Instead, the connections to the environment are made by directly inserting IO-variables in the χ-text.

When separated from the model of the physical system the model of the control system is considered without the channels that connected it to the model of the physical system (see Figure 3.4).

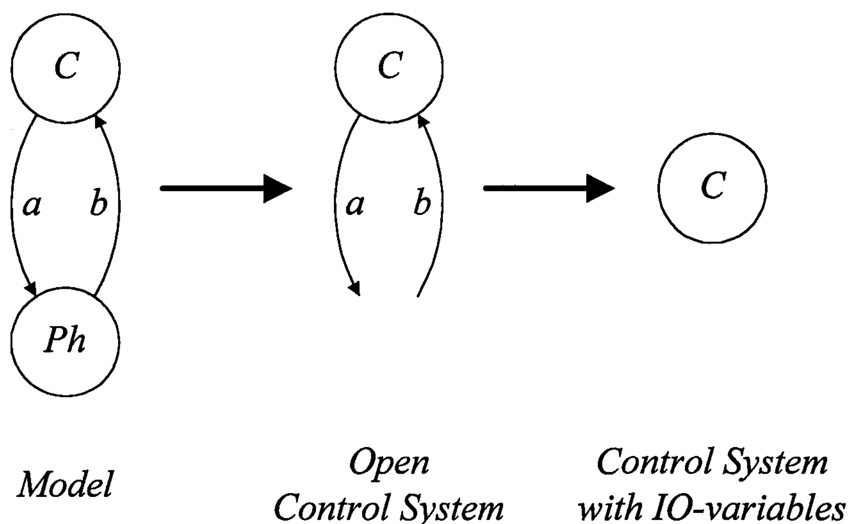


Figure 3.4: Control System using IO-Variables

The connection to the computer addresses is made separately for each variable that needs to communicate with the environment. This can be done by using the concept of IO-variables presented in the first option. IO-variables are defined by adding a @ for the normal type definition of the variable. Instead of transmitting the signal over a channel, the value is assigned to an IO-variable. The IO-variables are pointing directly to a computer address. Again, one new language item is introduced being the IO-variable.

Consider a process C as presented in Figure 3.4. To illustrate clearly the differences between the simulation model and the processes that are used to generate the

executable, the original χ-text of process C precedes the adjusted process C.

In the simulation model a binary variable y is used to store a value that is received from the physical system over channel b. The output is sent over channel a. The process C first receives a signal that is stored in variable y. If the value of y is 1, the process C sends the output 1 over channel a. This cycle is then repeated. In the simulation model process C is described as follows.

```
proc C(a : !int, b : ?int)=
|[ y : int
 | *[ b?y ⟶ [ y = 1 ⟶ a!1
              [] y = 0 ⟶ skip
              ]
    ]
]|
```

In the control system that is used as the basis for the real-time control program, this would result in IO-variables, io_1 and io_2, being input and output signals respectively. If the concept of IO-variables is incorporated into the model this produces the text presented below.

```
proc C=
|[ io1, io2 : @int, y : int
 | *[ y := io1 ⟶ [ y = 1 ⟶ io2 := 1
                  [] y = 0 ⟶ skip
                  ]
    ]
]|
```

The IO-variable does not suffer under the rules for synchronous communication in the χ formalism. Strictly speaking there is no communication because the IO-variables are directly linked to the computer addresses. An assignment will never block the system as an assignment will, per definition, succeed. Thus, timing requirements depend not on the communication procedures but on the remainder of the program, as well as on the performance of the hardware the program is executed on.

A disadvantage of this solution is that in the graphical representation of the system the various IO-connections are not apparent. This is no problem when using small systems. If a large system is specified, however, it is hard to understand the control system if it is not clear which of the processes communicate with the environment and which do not. The processes in which communication with the environment

occurs might be indicated in some way, for instance with an asterix added to its name. Even then it is not visible how and how much the process communicates with its environment. Also, the connections with an environment are not made in the system definition where they belong.

Another disadvantage is the fact that, in this option, one must change the processes that were used to simulate the behavior of the control system. First, this might result in tedious work when searching through the system trying to find the various places were IO-communications must take place. Second, if one tests a system only to change it when one is to implement it, one cannot be completely sure that the conversion from the model to an implementation is done without errors.

3.5.3 IO-channels in χ

An alternative to implement the means to IO-communications is the use of IO-channels. In this solution, the unconnected channels of the control system become IO-channels. These IO-channels are coupled directly to a certain computer address. Through these channels the IO-communications are established.

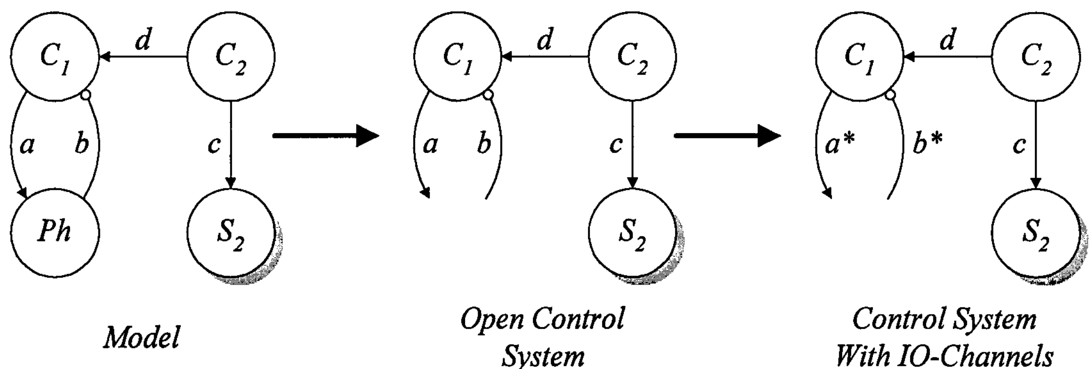


Figure 3.5: IO-Channels used for interfacing

See Figure 3.5. In this figure, a model of a machine control is presented. The physical part of the model is described in process Ph. The processes C_1 and C_2 are two controllers. The system S_2 is another system with which the process C_2 communicates.

Consider the IO-communications. At any time IO-communications are able to communicate. An actuator is always able to receive commands. A sensor always produces

data. In this respect this data is considered to be valid at any time. Thus, communication between the control system and the physical system is successful whenever the control system initiates a send or receive action.

Consider the functionality of the IO-channels. As in theory communication is always possible, the control system is always able to communicate using the IO-channels. This implies that the system will never lock in an IO-communication as these always succeed. Also, the IO-communications can be of any type.

It must be noted that the continuous IO-channel should be a stripped form the continuous channel that is used in the current version of the χ formalism. After all, there is no need for the various mechanisms to solve equations that are now part of the code of an executable program after compilation of a model that uses continuous IO-channels. At computer language level continuous data transfer of course does not exist. Computer communications are discrete per definition as the computer operates in a discrete domain. On the level of the χ model some communications can be viewed as continuous signals (sometimes with discrete changes). Some forms of sensory input can be seen as a source that produces data continuously.

A discrete version of the IO-channel must also be designed. As already has been mentioned, this discrete IO-channel cannot depend upon the hand-shake principle due to the fact that some equipment does not generate a hand-shake. Also, this discrete IO-channel must be able to pull its own input from a computer address.

The compiler must be able to recognize IO-channels and compile them accordingly. Using the IO-channels, the system must be able to send even when there is no apparent receiver to generate a hand-shake. It must also be able to pull data from a sensor when no apparent send signal is initiated by the sensor at the other side. In other words, the IO-channels would not operate conform the synchronous communication that is used in the χ formalism.

The adjustments that are demanded of the compiler are severe. Not only a new sort of channel is defined, the semantic expansion of the kernel for the various actions that can be performed over this channel are not to be underestimated. Send and receive actions over the IO-channels are made without hand-shaking as the hardware to which these signals are sent sometimes does not generate these sorts of confirmation. The compiler must recognize the IO-channels and their different way of communication and act accordingly.

This solution is more conform to the functionality of the χ formalism as it is used currently. The connections to the real-time system are made by the IO-channels. In the χ formalism channel connections are defined in the system definition and not in a process. The IO-channels are defined in the parameter definition of the system. In this option two sorts of channels are declared in the parameter declaration of the

system. The first already exists in the current χ formalism. These channels connect one system with another system. The second are the newly introduced IO-channels. The IO-channels are distinguishable from the conventional channels by a @ symbol preceding the type declaration of the channel.

Consider Figure 3.5. In this Figure a model is described with three processes, Ph, C_1 and C_2, and a system, S_2. The processes C_1 and C_2 are models of the control system. The process Ph is a model of the physical part of the system. The system S_2 describes another part of the model. The contents of each is not important. If a simulation is run using this model a system definition must be made, which is called S_1. The system S_1 is connected to system S_2 in the system definition of system S. This system definition defines in which way the three processes and the system S_2 interact. The system declarations are given below.

syst $S_1(c : !\,\mathrm{int}) =$
$[\![\ a : \mathrm{real},\ b : [m],\ d : \mathrm{int}$
$|\ C_1(a, b, d)$
$\|\ C_2(c, d)$
$\|\ Ph(a, b)$
$]\!]$

syst $S() =$
$[\![\ c : \mathrm{int}$
$|\ S_1(c)$
$\|\ S_2(c)$
$]\!]$

If the model of the physical system is deleted from the system, as shown in Figure 3.5, then a new system definition, called C, must be made. The process Ph is no longer part of this system. The process is replaced with the real physical system. The resulting system C contains the two processes C_1 and C_2. These processes are exactly the same as the ones that were used in the simulation run of the model. The system C is connected to the environment with channels a and b. The system C is still connected to another system, S_2. Using the concept of IO-channels, the system declaration of system C can be generated very easily. System C can then be connected to system S_2 in the usual way.

This results in system declarations as is shown below.

syst $C(a : !@\text{real},\ b :\ \multimap @[m],\ c : !\text{int}) =$
$[\![\ d : \text{int}$
$|\ C_1(a, b, d)$
$\|\ C_2(c, d)$
$]\!]$

syst $S(a : !@\text{real},\ b :\ \multimap @[m]) =$
$[\![\ c : \text{int}$
$|\ C(a, b, c)$
$|\ S_2(c)$
$]\!]$

In this example, the channels a and b are IO-channels whereas channel c is a discrete output channel to an other system or process. Notice that a is a discrete output channel, whereas b is a continuous channel. In the graphical representation of the systems IO-channels are defined by adding an asterix to the channel name. (Channel a is a conventional channel, channel a^* is an IO-channel.)

The extra work that has to be done to achieve a model that can be compiled into a real-time control system is minimal. No additional process has to be defined. The IO-process in the previous section seems to be easily made but remember that real-time systems with several hundred actuators and sensors are not uncommon. To make a process for these systems requires much more work. The open channels of the model of the control system are made into IO-channels. This is not done in the parameter definition of the processes themselves. Whether or not a channel is an IO-channel is only specified by the system definition. A new system definition is made. In this new system definition, the IO-channels are parameters of the system. Therefore the IO-channels are placed in the parameter declaration of the system definition and are not defined in the system itself. Another advantage of this approach is that the processes that are used in both the simulation model as well as the control system remain unchanged. Also, the IO-communications can be of any type.

To clarify these three possibilities presented here an example is given in the next chapter.

Chapter 4

Case Study: A Piston Control

In this chapter, a simple χ model is described. This model is used to illustrate three different methods presented in the previous chapter to incorporate the means for IO-communication in the χ formalism. First, the presented system uses the current χ-formalism and the ∇ operator on discrete channels as suggested in Chapter 3.4. The model is therefore not considered correct according to the current χ formalism.

Second, the model of the control system is adjusted according to one of the three methods explained in Chapter 3.5 to produce a possible real-time implementation of the control system. The presented model is used to illustrate the ways in which these methods work. The model is not used in tests.

4.1 A χ model

The system that is used in this example is an ordinary piston (see Figure 4.1).

This piston is controlled by two air valves, combined in a, one at each side of the piston. The position of the piston can be monitored by two sensors, combined in s, one at each end of the piston.

The χ model consists of three processes (see Figure 4.2).

- Process Op simulates the behavior of an imaginary operator. This operator waits for the process C to finish the initialization of the physical system. When the system is initiated correctly the operator receives a confirmation from the control system. The operator lets the piston extend and retract five times after which he shuts down the system.

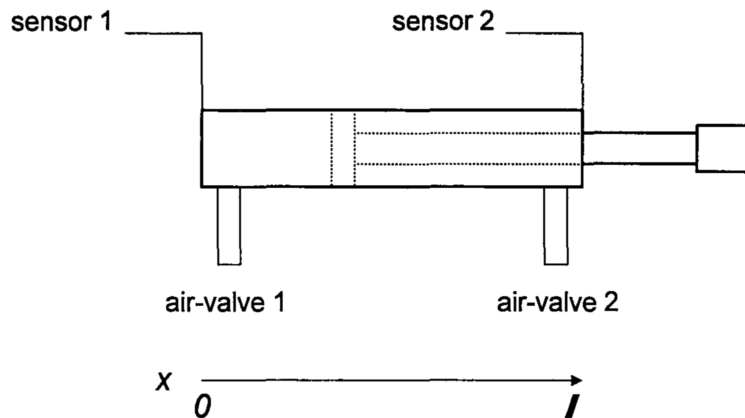


Figure 4.1: The Air-piston

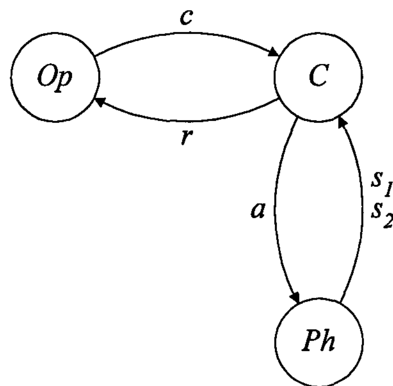


Figure 4.2: System Description

- Process C is the controller. The controller begins with an initialization of the physical system. The initial state the control system wants the physical system to be in is when the piston is completely retracted. When this has happened the control system sends a confirmation to the operator that the system is operational. The control system then waits for commands from the operator.

- System Ph is the representation of the physical system. The behavior of this process depends on the commands it receives from the control system. The physical system responds with sensory output.

4.1.1 Types

In this model a number of additional types is defined. These types are the following.

- A type is defined for the various commands the operator can give the control system. The instances of this type, called COM, are 1 (extend), -1 (retract), 0 (off) and 10 (idle).
- A type is defined to describe the status of the system. The values of this type, STATUS, are true (ok) and false (not-ok).
- A type ACT is defined to hold the two actuator commands. These commands are either 0, 1 or 10, representing OFF, ON or SHUT-DOWN respectively. These commands are combined in a tuple to be able to send them simultaneously. This prevents the system from entering an inconsistent state.

With these additions the type definition becomes the following.

$$\begin{array}{lll} \textsf{type} & \text{COM} & = \text{int} \\ , & \text{STATUS} & = \text{bool} \\ , & \text{ACT} & = < \text{int}^2 > \end{array}$$

4.1.2 System Definition

In order to function together, the various processes of the model must be connected. These connections are made in a system definition. In this system definition, the output channel of one process is coupled to the input channel of an other process. The type of each channel is also defined. This results in the following system definition.

$$\begin{array}{l} \textsf{syst}\ Piston() = \\ [\![\ c : \text{COM},\ r : \text{STATUS},\ a : \text{ACT},\ s_1, s_2 : \text{bool} \\ \mid Op(c, r) \\ \parallel C(a, s_1, s_2, c, r) \\ \parallel Ph(a, s_1, s_2, 10, 2) \\ ]\!] \end{array}$$

The two values in the definition of the process Ph are the values of the variables l and v.

4.1.3 Process *Op*

The process *Op* represents the operator of the system. Before the operator starts sending commands to the control system the operator waits for confirmation over channel r that the initialization of the physical system has succeeded. This confirmation is received by variable s.

As soon as confirmation has been received the operator starts sending commands over channel c to the control system. The commands can be to retract the piston, using -1, or to extend the piston, using 1. After each command the operator waits until the command is successfully executed. The operator extends and retracts the piston five times after which the operator shuts down the system with the command $c!0$.

```
proc Op(c : !COM, r : ?STATUS) =
|[ x : nat, s : bool
| x := 0
; r?s
; *[ x < 5 ⟶ c!1; r?s
              ; c!−1; r?s
              ; x := x + 1
  ]
; c!0
]|
```

4.1.4 Process *C*

The control system is described in process C. This process forms the connection between the operator and the physical system. The control system C receives commands from the operator over channel c and sends confirmation to the operator over channel r. The control system is connected to the physical system using the actuator channel, a, and two sensor channels, s_1 and s_2. Commands from the control system to the physical system are sent over the actuator channels. Response from the physical system is received from the sensor channels. The actuators are modelled using the tuple as described earlier. The sensor channels are discrete. In the χ text, one will notice that the sensor channels are only used to receive a confirmation once. The experienced χ programmer will argue that this is easier to model using a synchronization channel. The reason for not choosing a synchronization is that in real-life applications there are virtually no sensors that generate a synchronization signal as specified in the χ environment.

When started the control process first initializes the physical system. During this initialization the control system makes sure that the piston is completely retracted before it sends confirmation to the operator that the system is operational. The control system now enters a loop that executes the commands received from the operator.

At a command from the operator, the control system lets the piston extend with $a\,!<0,1>$. This command opens the correct air-valves. The control system waits for the sensor signal over s_2 to signal that the piston is extended. A ∇ operator is used for this purpose. In the simulation model it is not necessary to use this command. The process *Ph* only sends a signal when this signal becomes true. However, if the control process is converted to a real-time control program it is likely that the process also receives sensory input that is of no use to the process, making the ∇ operator necessary (see Section 3.4). The control system then signals the operator that it is ready to receive the next command. The retraction of the piston works in a similar manner.

If the operator signals the system to shut-down at the end of the cycle the variable *on* becomes false. The program leaves the control loop and terminates.

```
proc C(a : !ACT, s1, s2 : ?bool, c : ?COM, r : !STATUS) =
|[ on, sen1, sen2 : bool, cmd : COM
; cmd := 10; on := true
| a! < 0,1 > ; ∇(s1 ? sen1, sen1 = true)
; a! < 0,0 > ; r!true
; *[ on; c?cmd ⟶ [ (cmd = 1)    ⟶ a! < 1,0 >
                                   ; ∇(s2 ? sen2, sen2 = true); r!true
                  ▯ (cmd = −1)  ⟶ a! < 0,1 >
                                   ; ∇(s1 ? sen1, sen1 = true); r!true
                  ▯ (cmd = 0)   ⟶ a! < 0,0 > ; on := false
                  ▯ (cmd = 10 () ⟶ skip
                  ]
   ]
; a! < 10,10 >
]|
```

4.1.5 Process *Ph*

The process *Ph* models the behavior of the physical system, in this case an air-piston. The continuous part of this process contains an equation for the change in position of the air-piston. This equation defines the change in the position of the

air-piston to be equal to the velocity of the air-piston multiplied by a constant *st*. This constant *st* is used to fix direction of the velocity. The discrete part describes the various combinations of commands that might occur, the physical restrictions that the air-piston enforces upon the system and the possibilities of the physical system to communicate with the control system.

The following states are possible. When the air-valves are both closed, $com.0 = 0 \land com.1 = 0$, the air-piston will not move. This results in $st := 0$ which evaluates in the continuous equation to $x' = 0 * v = 0$. If both the air-valves are under pressure, $com.0 = 1 \land com.1 = 1$, the air-piston will slowly extend. This is caused by the fact that the surface on one side on which the pressure works is reduced by the surface of the piston-rod, causing the air-piston to extend slowly. The variable $st := 0.5$. When $com.0 = 1$ and $com.1 = 0$ the variable $st := 1$ and the piston extends. When $com.1 = 1$ and $com.0 = 0$ the variable $st := -1$ and the piston retracts.

When the piston is moving the process changes the value of the sensors according to the position of the piston rod. If the air-piston reaches the physical constraints of the system, either $x \leq 0$ or $x \geq l$ (where l is the length of the piston), the movement of the piston is stopped by $st := 0$ and the corresponding sensor is activated, $sen_1 :=$ true respectively $sen_2 :=$ true. When the piston is at a position between the end positions both sensors are false. As the operator is not able to send other signals during execution of his commands, no means are incorporated to allow communication possible during the movement of the piston rod.

```
proc Ph(a : ?ACT, s1, s2 : !bool, l, v : real) =
|[ on : bool, st : real, com : ACT, x : [m]
 ; com := < 0,0 > ; st := 0; x : := .5 × l; on := true
 | x' = st × v
 | *[ on    ⟶ a?com
              ; [ com.0 = 1 ∧ com.1 = 1    ⟶ st := 0.5
                [] com.0 = 0 ∧ com.1 = 0    ⟶ st := 0
                [] com.0 = 1 ∧ com.1 = 0    ⟶ st := 1
                [] com.0 = 0 ∧ com.1 = 1    ⟶ st := −1
                [] com.0 = 10 ∧ com.1 = 10 ⟶ st := 0; on := false
                ]
```

$$
\begin{array}{ll}
; [\ st = 1 & \longrightarrow \nabla x > 0;\ s_1\,!\,\text{false} \\
 & \quad ;\ \nabla x \geq l;\ st := 0;\ s_2\,!\,\text{true} \\
[]\ st = 0.5 & \longrightarrow \nabla x > 0;\ s_1\,!\,\text{false} \\
 & \quad ;\ \nabla x \geq l;\ st := 0;\ s_2\,!\,\text{true} \\
[]\ st = -1 & \longrightarrow \nabla x < l;\ s_2\,!\,\text{false} \\
 & \quad ;\ \nabla x \leq 0;\ st := 0;\ s_1\,!\,\text{true} \\
[]\ st = 0 & \longrightarrow \text{skip} \\
]
\end{array}
$$

$]|$

Note that in the initialization of the variables the position of the air-piston is said to be somewhere between the end positions. This can be any random position. Chosen here is $x := .5 \times l$, a position in the middle of the air-piston stroke. The process *Ph* presented above does not generate an executable when compiled. The process as described is a desired form. It differs significantly form the current χ syntax and semantics due to the usage of the ∇-operator. Also, the kernel used in this project is a discrete kernel. The process *Ph* in the presented model is hybrid.

4.2 IO-Communication in the χ Model

The model of the air-piston presented in the previous chapter is adapted in three different ways for IO-communication in this chapter. Each of these ways is described in Section 3.5 and represents a different approach to the integration of IO-communication in the χ formalism. The specific mechanisms behind each of the options have already been explained in Section 3.5. In these examples the introductions with the options are therefore brief. All the options discussed are using only the model of the control system from the model presented above. The physical system and the operator process have been deleted. The signals from and to the operator process are being seen as IO as far as the controller is concerned. The open control system remains.

4.2.1 IO-process

This option adds an additional process to the model of the control system. This additional process is called *IO*. In this process, the channels that run to and come from the model of the control system are linked to addresses. The additional process closes the system as shown in Figure 4.3.

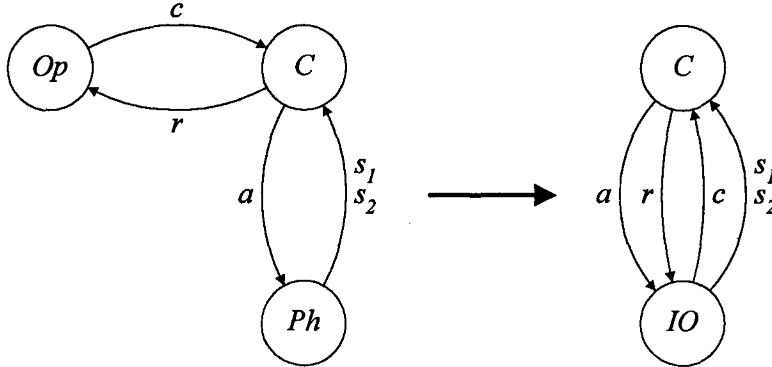


Figure 4.3: The conversion of a model to a control system with an IO-process

The process C is not changed and the text of this process is only repeated here for convenience.

```
proc C(a : ! ACT, s1, s2 : ? bool, c : ? COM, r : ! STATUS) =
|[ on, sen1, sen2 : bool, cmd : COM
; cmd := 10; on := true
| a! < 0,1 > ; ∇(s1 ? sen1, sen1 = true)
; a! < 0,0 > ; r ! true
; *[ on; c? cmd ⟶ [ (cmd = 1)  ⟶ a! < 1,0 >
                                    ; ∇(s2 ? sen2, sen2 = true); r ! true
                    [] (cmd = −1) ⟶ a! < 0,1 >
                                    ; ∇(s1 ? sen1, sen1 = true); r ! true
                    [] (cmd = 0)  ⟶ a! < 0,0 > ; on := false
                    ]
   ]
; a! < 10,10 >
]|
```

The process IO contains the following.

$$
\begin{array}{l}
\textsf{proc}\ IO(a : ?\,\text{ACT},\ s_1, s_2 : !\,\text{bool},\ c : !\,\text{com},\ r : ?\,\text{status} \\
\qquad)= \\
[\![\ io_1 : @\text{ACT},\ io_2, io_3 : @\text{bool},\ io_4 : @\text{com},\ io_5 : @\text{status} \\
\mid *[\ a\,?\,io_1 \longrightarrow \text{skip} \\
\quad [\!]\ s_1\,!\,io_2 \longrightarrow \text{skip} \\
\quad [\!]\ s_2\,!\,io_3 \longrightarrow \text{skip} \\
\quad [\!]\ c\,!\,io_4 \longrightarrow \text{skip} \\
\quad [\!]\ r\,?\,io_5 \longrightarrow \text{skip} \\
\quad] \\
]\!]
\end{array}
$$

Inside process IO all the channels coming from, and going to the control system are linked to specific computer addresses. Only one channel can be linked to an address and only one address can be linked to a channel. Notice that these computer addresses can be linked to channels of any type. The specific implementation of the computer address will always be an integer because computer addresses can always be denoted as integers. The type declaration only serves as a means to define the type of variable that is transferred to or from this specific address. The system definition is given below.

$$
\begin{array}{l}
\textsf{syst}\ Piston() = \\
[\![\ a : \text{ACT},\ s_1, s_2 : \text{bool},\ c : \text{com},\ r : \text{status} \\
\mid C(a, s_1, s_2, c, r) \\
\parallel IO(a, s_1, s_2, c, r) \\
]\!]
\end{array}
$$

This option seems very easy but might become a lot of work when the system that is described contains a large number of IO-communications. An advantage of this solution is that the same processes that are used in the simulation model can be used unchanged to describe the real-time control system.

4.2.2 IO-variables

This option creates a new sort variable which can be used in assignments. The model of the control system is decoupled from the rest of the model. The in- and outgoing channels of this process are also deleted (see Figure 4.4).

This causes major changes in the model text. Channels are deleted and replaced by IO-connections. The send and receive actions over these channels become IO-assignments. The text of the model of the control system changes to the following.

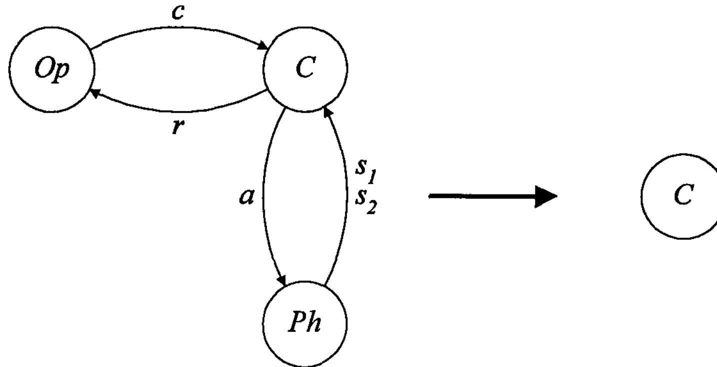


Figure 4.4: The conversion of a model to a control system using IO-variables

```
proc C =
[[ on : bool, a : @ACT, s1, s2 : @int, c : @COM, r : @STATUS, vc : COM
; on := true
| a := < 0,1 > ; ∇(s1 = true)
; a := < 0,0 > ; r := true
; *[ on   ⟶ vc := c; [ (vc = 1)   ⟶ a := < 1,0 >
                                      ; ∇(s2 = true); r := true
                      [] (vc = −1) ⟶ a := < 0,1 >
                                      ; ∇(s1 = true); r := true
                      [] (vc = 0)  ⟶ a := < 0,0 > ; on := false
                      ]
   ]
; a := < 10,10 >
]]
```

All the channels disappear. They are replaced by IO-variables. To these IO-variables other internal variables can be linked, with statements like $s_1 := sen_1$. Or one can write to these IO-variables directly, like $a := <1,0>$. With the disappearance of all the channels, all the instances of send and receive actions also disappear. They are replaced by assignments, instead of send actions, or they are replaced by a 'read', in case of receive actions.

The major disadvantage of this option is the enormous amount of work and the fact that it is prone to errors. After the processes are simulated and tested the processes that are tested are changed before they are compiled into a control system. This means that the complete system should once again be debugged and tested to make sure the processes are adjusted correctly. This is an unnecessary and tedious job.

4.2.3 IO-channels

This example uses IO-channels instead of an additional IO-process. These IO-channels are connected to the model of the control system on one side. At the other side they are linked directly to computer addresses. This results in a control system as shown in Figure 4.5.

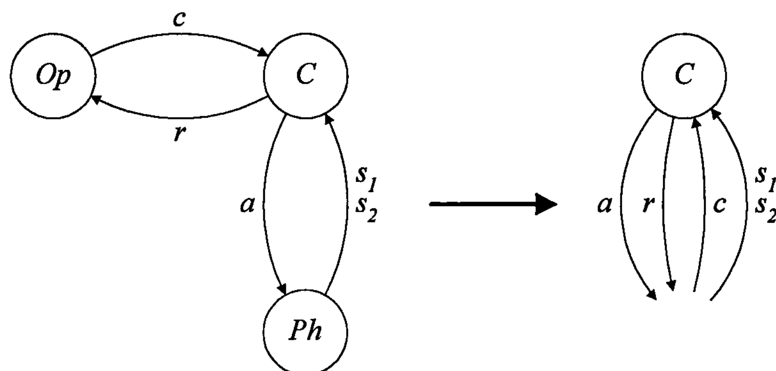


Figure 4.5: The conversion of a model to a control system with IO-channels

The process C is not changed. Instead, the system definition is used to link the channels to addresses. The channels that are to be linked to an address are recognized by the addition of a @ to the type declaration of a channel in the parameter declaration of the system. This is done to be able to discern them from channels that are connected to other model components like higher-level system definitions or other processes. The model of the control system is now described by the following text.

```
proc C(a : !ACT, s1, s2 : ?bool, c : ?COM, r : !STATUS) =
|[ on, sen1, sen2 : bool, cmd : COM
; cmd := 10; on := true
| a! < 0,1 > ; ∇(s1 ? sen1, sen1 = true)
; a! < 0,0 > ; r!true
; *[ on; c?cmd ⟶ [ (cmd = 1)  ⟶ a! < 1,0 >
                                  ; ∇(s2 ? sen2, sen2 = true); r!true
                 [] (cmd = −1) ⟶ a! < 0,1 >
                                  ; ∇(s1 ? sen1, sen1 = true); r!true
                 [] (cmd = 0)  ⟶ a! < 0,0 > ; on := false
                 ]
  ]
; a! < 10,10 >
]|

syst Piston(a : @!ACT, s1, s2 : @?bool, c : @?COM, r : @!STATUS) =
|[ C(a, s1, s2, c, r)
]|
```

Every IO-channel is assigned an IO-address by the implementor. Each channel can only be connected to one address and one particular address can only be connected to a single channel. Some channels transport tuple typed variables. In this case it might be necessary that each position of the tuple is connected to a different address.

The major advantage of this approach is that nothing of the model that does not change functionally, does not change in text. The processes of the model of the control system are still the same. A new system definition is made with IO-channels. The processes that were used to build the simulation model can be used unchanged in the control system.

This option is easy to implement and efficient. To implement this option only the concept of IO-channels must be defined in the χ formalism.

4.3 A Choice for an IO-Mechanism in χ

In the previous chapters, three different mechanisms to incorporate IO-communications in χ have been presented. Each of these mechanisms can be used to implement IO-communications in χ. The time available for this project is not sufficient to examine all these three mechanisms in detail. A choice must be made for one specific possibility.

The second option, the one using the IO-variables, is discarded. The reason for this is the amount of work the option takes. Also, the connections to the environment are scattered throughout the various processes. The connections are not collected in one place in the system. A third disadvantage is the fact that the option of IO-variables changes the syntax of the processes that are used in the simulation of the model. No certainty can be given on the correct conversion of these changed processes.

The other two options remain. On the one hand there is the IO-process, on the other hand there is the IO-channels. Both options have three similar advantages.

- The processes used in the simulation of the model remain unchanged when the model is converted to a real-time control system. No unnecessary and error-prone replacing of text has to be done.
- In both options, the signal used for IO-communication can be of any type.
- In both options, the places where IO-communication must occur are clearly visible. In the first option, all channels that are connected to the IO-process are used to transfer IO-communications. In the second option, all channels that are connected only to the control system on one side and not connected to another system or process on the other side, are IO-connections.

The first option using the IO-process has an additional advantage that the adjustments to the kernel to implement this option in the χ formalism seem rather straightforward. With the third option, that of the IO-channels, the kernel needs larger adjustments than the first one.

However, the first option also has two major drawbacks. First, it is not conform the philosophy of the current χ language. In the χ formalism, the connection to parts outside the system are made in the system definition. It is logical to define the IO-connections in this system definition. In the first option, the IO-addresses are defined in a process. This means that a modeler or implementor must first access the IO-process to examine the various IO-connections. In the third option, the connections to the real world are defined in the system definition together with all the other connections in the system. The values of the various IO-addresses are parameters of the system. Programs with the same functionality can have different address values when run on two different machines. As the IO-connections are parameters as far as the system is concerned, they should be found in the parameter declaration of a system, not in a process within that system.

A second disadvantage of the first option concerns the connection that is made between the IO-process and the rest of the system. The channels that run between

the model of the control system and the IO-process are conventional χ channels. This causes no problems when continuous data transfer is considered. When discrete channels are concerned however, some major adjustments must be made on the IO-process. First, the connections that are made between the channels and the IO-addresses in the IO-process must be made to generate a hand-shake to satisfy the synchronous character of the communication protocol in χ. This must be done for both input and output signals. Second, in case of a discrete input (from the control systems point of view), an external process must be defined that at regular intervals provides the control system with data. As the discrete channel between the control system and the IO-process is not able to pull its own data, the data must be presented to the control system. A mechanism must be designed to ensure correct data transfer.

When the option using an IO-process and the option using IO-channels are compared, the disadvantages of the first option outweigh the advantages. The third option, using IO-channels, seems to be the most promising. Hence, the third option is implemented in the course of the project. With this implementation, tests are conducted. Implementation of the first option is left for future research.

Chapter 5

Interfacing χ with Hardware

In this project, the χ formalism is used to design specifications of control systems. If these specifications are to be used to generate real-time control systems, it is necessary to interface the χ code with the IO-routines and drivers used to control the real-time system. To do so, two conditions must be met. First, it has to be possible to access driver routines or it has to be possible to make system calls, using the kernel of the real-time system. Second, a connection must be made between the χ code and the driver routines that control the real-time system. Currently, a χ-model is compiled into C^{++}. It is therefore required, that system calls and driver routines of the real-time system are also written in C or C^{++}.

5.1 The χ Formalism in Hardware Control

In the previous chapters, the means to incorporate IO-communications into the χ-formalism have been presented. Through the IO-channels, the χ specification is able to access the driver routines or the system calls of the real-time system.

It is possible to link function calls in χ to driver routines. This is done using an interface file as described by [Nau96]. If a real-time system is controlled in this way, some overhead will be present because the scheduling of the processes that control the application is done using the χ engine. It might also be possible to change the χ engine in such a way that it would generate code that is linkable with a real-time kernel. This is a better method because overhead in the machine code is reduced. It does mean that the code generation part of the χ compiler will have to be changed for each specific type of real-time kernel. Also, for each new hardware, the real-time kernel operating on that hardware, will have to be accessed.

The position of the χ application in respect to the hardware is illustrated in Figure 5.1.

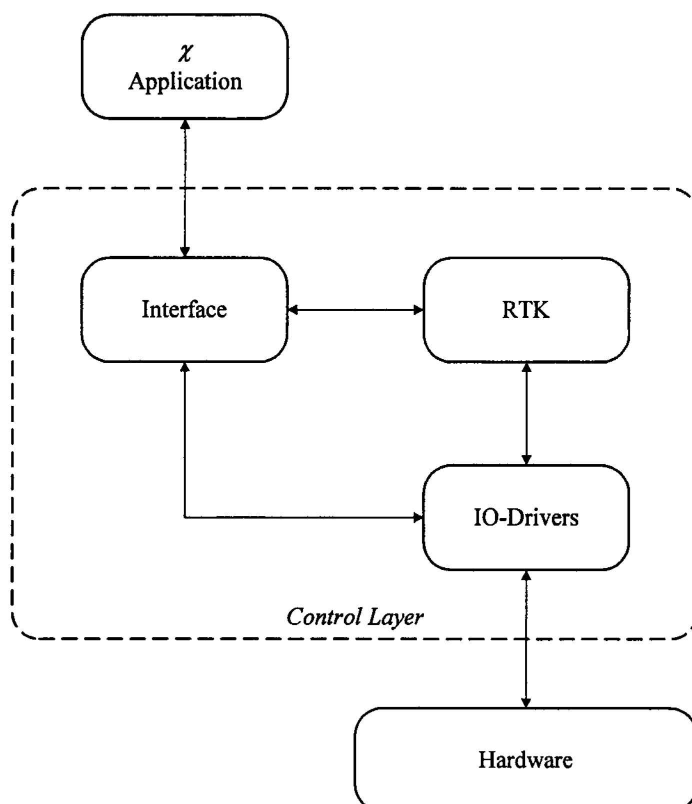


Figure 5.1: An interface between a χ application and hardware [Nau96]

Consider Figure 5.1. The χ application communicates with the interface. The interface communicates with the hardware, using a real-time kernel (RTK), or directly, accessing the IO-driver routines.

Because it is possible to access the driver routines directly using the interface file, it is better to bypass the real-time kernel, if possible. This will reduce overhead, and leaves only the overhead of the χ kernel.

The layer between the χ application and the hardware is called the *Control Layer*. In this layer, the χ application is coupled to the hardware.

5.2 The Experiment, IO-communication in χ

In this section, it is explained how the means for IO-communications are implemented into χ. The major problem in this respect is the fact that the χ-kernel is written using C^{++} in a Linux environment. The driver software that is used by the InterBus-S system is usable only on a DOS-based platform. Unfortunately, these two are not completely compatible. Some adjustments have therefore been made to the source code of the χ-kernel. These adjustments are presented first.

Second, the various actions executed to generate a control program are explained in detail. This explanation goes beyond what is visible for a normal user of the real-time χ-kernel.

5.2.1 Combining the χ-Kernel with DOS Driver Software

The interface system that is used with the experiments during this research is the InterBus-S system. The driver software for the InterBus-S interface card is supported under either C for DOS, Pascal for DOS, WIN 3.1x, WIN95 or WindowsNT. As the χ-kernel is written in GNU, this means that either the driver software must be rewritten in GNU, or the χ-kernel must be adjusted to fit the driver software.

For this research, the choice is made to adjust the χ-kernel in such a way that it is possible to compile χ-code on a DOS platform. The application that is used to create all the necessary code is Visual C^{++} 4.2 for WIN95.

The adjustments to the χ-kernel that have been made are the following:

1. The class `String` has been altered to `string`.
2. In the original χ-kernel, the class `CChannelInt` had a nested structure. This structure has been rewritten to remove the nested statements.

The rest of the kernel has been left unaltered. Most of the work in this stage of the research has been done by experts that have access to the source code of the χ-kernel.

5.2.2 The Real-Time χ-Kernel

The work done during this research yields a real-time χ-kernel. As is defined in Section 2.1.1, a control program functions real-time if the sequence in which its

actions are made is determined by its environment. The new χ-kernel corresponds with this property. Unfortunately, it is not possible to define timing requirements in the new real-time χ-kernel. In this respect, the new kernel is very limited. A coupling between the simulation time and real-time has not yet been achieved. The new kernel can be used to create control programs. This section describes the underlying functionality of the real-time χ-kernel.

The Batchfile of the Real-Time χ-Kernel

When a control program is generated, a batchfile, `chirt.bat`, is called by the user. This batchfile is used as a means to explain the working of the real-time χ kernel. First, the batchfile is presented. Then, the commands executed in this batchfile are explained one at the time to clarify the real-time χ kernel. To illustrate the contents of this batchfile a flowchart is given in Figure 5.2.

The text of the batchfile `chirt.bat` is given below.

```
@echo off

set CHIHOME=c:/rtchi
set PATH=s:\appl\chi\chi03;%PATH%
c:\rtchi\util\sed -f %chihome%/util/sedf %1.chi > tmp.chi
chi.exe -o -l%CHIHOME%/stdlb.l -l%chihome%/ibschi.l tmp.chi

c:\rtchi\util\sed -f %chihome%/util/sedf2 tmp.cc >tmp.cc1

set INCLUDE=c:\msdev\include;c:\rtchi\kernel;c:\ibsdriver\inc;c:\rtchi\ibschi
set INCLUDE=%INCLUDE%;c:\rtchi\ibsutil
set LIB=c:\rtchi\kernel;c:\rtchi\ibschi;;c:\rtchi\ibsutil;c:\msdev\lib
copy tmp.cc1 %1.cpp

cl /c /GX /W3 /D "WIN32" /D "_CONSOLE" /YX /nologo /I%CHIHOME%/kernel %1.cpp

link /NODEFAULTLIB:libc %1.obj kernel.lib ibschi.lib ibsutil.lib
```

First, the variable `chihome` is adjusted so that it is pointing to the real-time χ-kernel. Then, the path indication is altered to allow the program to use the correct version of the kernel.

After this, a line is added in the batchfile that calls a program, called `sed.exe`, that substitutes functions for a part of the χ code in the control program text. How this

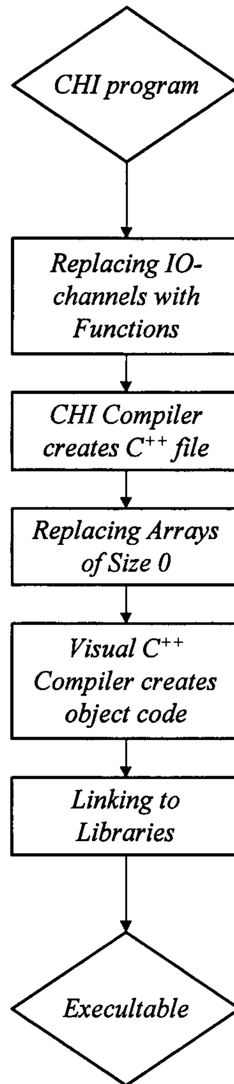


Figure 5.2: From CHI Application to Executable

is done, and what is substituted for what is explained in the next subsection. The adjusted program text is saved in the file `tmp.chi`. This file is then compiled using the χ-compiler. The file `ibschi.l` is used by the compiler to define the relation between the functions used in the χ-program `tmp.chi` with the functions defined in the various libraries for the hardware that is to be used, as defined by [Nau96].

The result of the compilation is a temporary file, called `tmp.cc`. To explain the addition of the next call to `sed.exe`, an explanation of the handling of a selective waiting statement is needed. A selective waiting statement contains guards with

either a communication or a delta statement. When a selective waiting is used in the χ-code of the model, the χ-compiler will keep track of how many communications and how many delta statements are used in the selective waiting. These numbers are then stored in arrays. For example, the following selective waiting contains 2 communication alternatives and 1 delta alternative.

$$
\begin{array}{l}
[\ x = 1 \quad ; b?a \longrightarrow \ldots\ldots \\
[\!]\ x = 0 \quad ; c!2 \ \longrightarrow \ldots\ldots \\
[\!]\ x = -1 ; \Delta\, 5 \ \longrightarrow \ldots\ldots \\
]
\end{array}
$$

The selective waiting statement below contains only 2 communications and no delta alternative.

$$
\begin{array}{l}
[\ y = 0;\ a1!1 \ \longrightarrow \ldots\ldots \\
[\!]\ y = 5;\ s1?x \longrightarrow \ldots\ldots \\
]
\end{array}
$$

As the Δ-statement cannot be used yet in the real-time χ-kernel, a selective waiting will always contain zero Δ-statements. The corresponding arrays in the `tmp.cc` are therefore at length zero. This poses no problem for the gcc-compiler but the Visual C^{++} 4.2 compiler will not compile arrays of length zero. To solve this, the corresponding arrays of length zero are replaced by empty arrays of length one. As these arrays are only declared and not used by any other part of the program, this is not a problem. This solution is certainly not an elegant one, as it does not cure the disease but merely hides the symptoms. It is only implemented for the sake of the experiment.

The next lines define the location of include files and libraries necessary for compilation. Then, the temporary file that is generated by the χ-compiler is renamed to allow it to be used by the compiler of Visual C^{++} 4.2. The file is then compiled by Visual C^{++} 4.2.

In the last line of the batchfile the generated object file is linked with the kernel libraries and the libraries `ibschi.lib` and `ibsutil.lib` that is created in this research. In the first the driver software of the InterBus-S system can be found. The latter holds utility procedures for the program.

Substituting Functions for χ-Code using `Sed.exe` and `Sedf`

Because the resources and time available for this research are limited, it has not been possible to generate a full-automatic conversion of the new components in χ for IO-communications to function calls during the compilation of a description of a

control system in χ. This results in the necessity to insert these function calls into the χ-text before compilation.

To do so, a small utility program, called `sed.exe`, is executed. This program is used in conjunction with another file, called `sedf`, in which is defined what should be substituted for what. The file `sedf` itself is given below. This file holds the various statements in χ and its replacements. After this a small example program is given together with the resulting text when it has been processed.

The program text of `sedf`.

```
s/\(IO[A-Za-z0-9]*\)!\([01]\)/\[PutBit("\1",\2) -> skip \]/g
s/\(IO[A-Za-z0-9]*\)?\([A-Za-z][A-Za-z0-9]*\)/\2:=GetBit("\1")/g
s/IO[A-Za-z0-9]*:\@!int[  ]*,/ /g
s/IO[A-Za-z0-9]*:\@!int[  ]*)/ )/g
s/IO[A-Za-z0-9]*:\@?int[  ]*,/ /g
s/IO[A-Za-z0-9]*:\@?int[  ]*)/ )/g
s/IO[A-Za-z0-9]*:!int[  ]*,/ /g
s/IO[A-Za-z0-9]*:!int[  ]*)/ )/g
s/IO[A-Za-z0-9]*:?int[  ]*,/ /g
s/IO[A-Za-z0-9]*:?int[  ]*)/ )/g
s/IO[A-Za-z0-9]*[  ]*,/ /g
s/IO[A-Za-z0-9]*[  ]*)/ )/g
/proc/ s/( *)//
/syst/ s/( *)//
s/( *)//
/syst/,/\]|/ s/,[  ]*)/)/g
/syst/,/\]|/ s/([  ]*,/(/g
/syst/,/\]|/ s/,[  ]*,/,/g
/proc/ s/,[  ]*)/)/g
/proc/ s/([  ]*,/(/g
/proc/ s/,[  ]*,/,/g
```

Example The execution of the `sed -f sedf` command is illustrated using a simple example program. The example program consists of one process that controls a piston. The piston is extended and retracted five times after which the program terminates (this is not the same example as used in Chapter 4). The topology of the various actuators and sensors on the piston is given in Figure 5.3.

In this example, the channels $IOa1$ and $IOa2$ sent to actuators 1 and 2 respectively. The channels $IOs1$ and $IOs2$ receive from sensors 1 and 2. The example is created using the concept of IO-channels. The resulting χ text is given below.

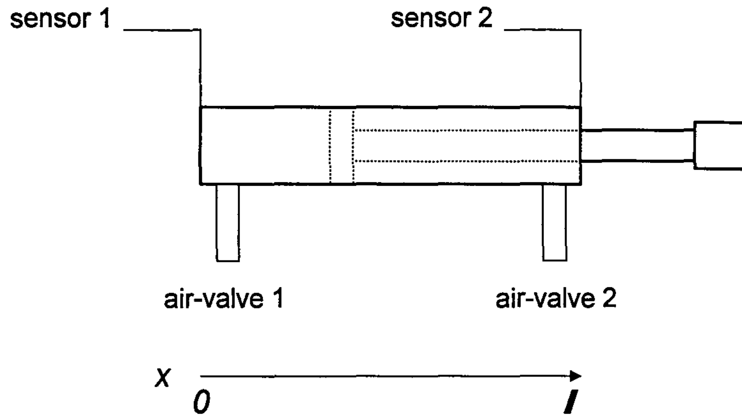


Figure 5.3: The Air Piston

```
proc pTest(IOa1,IOa2:!int, IOs1,IOs2:?int) =
|[on: bool, x,s1,s2: int
| on:=true; x:=5; s1:=0; s2:=0
; *[on -> [x>0 -> [ IOa1!1; IOs2?s2
                  ; *[s2=0 -> IOs2?s2]
                  ; IOa1!0
                  ; IOa2!1; IOs1?s1
                  ; *[s1=0 -> IOs1?s1]
                  ; IOa2!0
                  ; x:=x-1
                  ]
          |x=0 -> on:=false
          ]
   ]
]|

syst Test(IOa1,IOa2:@!int, IOs1,IOs2:@?int)=
|[
|| pTest(IOa1, IOa2, IOs1, IOs2)
]|
```

If the `sed -f sedf` command is used on this file, all the instances of the IO-channels are replaced by function calls. This results in a program that can be connected to function calls of the IO-drivers as explained by [Nau96]. The program is converted into the text below.

```
proc pTest =
|[on: bool, x,s1,s2: int
| on:=true; x:=5; s1:=0; s2:=0
; *[on -> [x>0 -> [ [PutBit("IOa1",1) -> skip ]; s2:=GetBit("IOs2")
                  ; *[s2=0 -> s2:=GetBit(''IOs2'')]
                  ; [PutBit(''IOa1'',0) -> skip ]
                  ; [PutBit("IOa2",1) -> skip ]; s1:=GetBit("IOs1")
                  ; *[s1=0 -> s1:=GetBit(''IOs1'')]
                  ; [PutBit(''IOa2'',0) -> skip ]
                  ; x:=x-1
                  ]
           |x=0 -> on:=false
           ]
   ]
]|

syst Test=
|[
|| pTest
]|
```

Notice the peculiar replacements of the send actions over the IO-channels. The reason why these are replaced by a selection statement is that the χ-formalism does not allow the use of separate functions outside a expression unless the function returns a boolean. If a function returns a boolean, the function can be used as a guard. In this case, the solution was to create a `PutBit` function with a boolean as return value. This return value now functions as a guard in the selection statement. The function can now be used separately without the need to assign the return value to a variable.

The program above can be compiled by the χ-compiler. It is compiled together with an interface file that connects the function calls above to the IO-drivers of a specific interface system, the InterBus-S System in this research. A description of the IO-drivers can be found in Appendix B, together with the source code of these drivers.

Substituting Arrays[1] for Arrays[0] using `Sed.exe` and `Sedf2`

As is mentioned, arrays of length zero cannot be compiled by the Visual C^{++} 4.2 compiler. To create a compilable file for the Visual C^{++} 4.2 compiler, the file created

by the χ-compiler is checked for empty arrays due to the usage of selective waiting statements. These are then replaced by arrays of length one. The program that is used is `Sed.exe` in combination with `Sedf2`. The file Sedf2 is printed below.

```
s/bDltGuard\[0\]/bDltGuard\[1\]/g
s/iDltCounter\[0\]/iDltCounter\[1\]/g
s/cDelta\[0\]/cDelta\[1\]/g
```

The arrays in question, being `bDltGuard[0]`, `iDltCounter[0]` and `cDelta[0]` are replaced by `bDltGuard[1]`, `iDltCounter[1]` and `cDelta[1]` respectively.

Linking Chi Functions with other Library Functions

It has been explained in the previous chapter, how the IO-channels in the specification of the model are replaced by functions. Now, these functions must be linked to functions described in the libraries with the driver routines. To do so, an interface file is used as described by [Nau96].

In this file, the functions in the χ-specification are translated to library functions. The interface file that is used is given below.

```
-- Program name: IBSCHI.1
-- Author      : Jasper van Rosmalen
-- Datum       : 7-10-1997
--
-- Description : This is the interface file for the IBS-system
--

func PutBit(string, int)        -> bool = BitWrite
func GetBit(string)             -> int  = BitRead
```

The interface file states the name of the function in the χ-specification together with its parameters. This function is then linked to another function defined in the libraries with the driver routines. From these functions, the return value is given. Important is that both functions have the same number and type of parameters. As is shown, these functions need not have the same name.

5.2.3 Initialization and Shut-Down Procedure of the Interface System

Before communication can take place, the interface system must be initialized. When a program is terminated, the interface system must be shut-down in a controlled manner. These two action depend on the interface system that is used. The initialization and the shut-down of the interface system are therefore implementation dependent. Because of this, it is not correct to include them into the χ-model of a control system. When this model is compiled into an executable for a specific interface system, the relevant procedures for initialization and shut-down must be inserted into the program text.

In the χ-kernel, the compiled χ-text of a model is inserted into a C^{++} `main`. This program `main` is then compiled by a C^{++} compiler into an executable. The initialization and shut-down of the system is inserted, invisibly for the modeler, in this program. The main for the real-time χ-kernel is given below.

```
// This is the main file of chi models
// It uses the include file xper.h which
// specifies the rest of the model.


#include ''xper.h''
#include ''sched.h''

void main(int argc, char** argv){

 char *pConfName;
 int   con;

 if(argc<2){
  cout << ''No configuration was set.'' << endl;
  cout << ''Set configuration and try again.'' << endl;
  return;
 }

 pConfName=argv[argc-1];
 con=ReadConfig(pConfName);
 if (con==0){
  cout << ''Unable to read configuration.'' << endl;
 }
```

```
 else{
  StartIBS();
  CScheduler cSched(argc-1, argv);
  CXPer      cX(&cSched);

  cSched.Run();
  StopIBS();
 }
}
```

The program checks if a configuration is provided for the executable. If no configuration is detected the executable terminates with a message that no configuration was set.

If a configuration is present, the program checks if the configuration that was set is valid. If so, the interface system is initialized with the command `StartIBS()`. When the executable terminates, the interface system is shut-down by the procedure `StopIBS()`.

Chapter 6

Using Real-Time χ

In this chapter, the real-time kernel is presented at user level. The underlying actions that are taken when a new control program is generated are not explained. A detailed description of the statements that are executed by the real-time χ kernel is given in Appendix B.

Considering the time and resources available for this research, it has not been possible to adjust the kernel in such a way that all the possibilities of the current χ formalism are supported by the real-time version of the χ-kernel. To be able to experiment with the real-time kernel, some restrictions are defined for the real-time kernel when making control systems in χ. These must be abidden by if one wants to use the real-time kernel.

6.1 Restrictions to Real-Time χ

As mentioned before, the current version of real-time χ does not support all the possibilities of the current χ formalism. Some restrictions are set to the use of real-time χ to create a control program.

1. This research does not consider continuous communications. The control system described with real-time χ is not allowed to contain communication other than discrete communications. The reason for this is that the real-time kernel is derived from the discrete Chi 0.3 kernel and only discrete interfaces are considered.
2. The discrete communication with the environment is bitwise. Thus, the value of the discrete signals is either `high` or `low`, or 0 or 1 respectively.

3. It has not been possible to implement a real-time variant of the Δ-statement. The use of this statement results in indeterministic program code. This is caused by the fact that the Δ-statement is connected to simulation time instead of a real-time clock. What happens, is that this simulation time ascends with the passing of statements, not time. Thus, this Δ-statement yields different timing intervals depending on the number of statements that are to be calculated in simulation time, the workload of the processor and the speed of the processor the program is running on.

4. Due to the provisional way in which the real-time χ-kernel operates, the IO-channels must be defined in specific way. The first two characters of the name of a IO-channel must always be the capitals IO. Also, the entire length of the name is not allowed to exceed four characters. For example, $IOs2$ and $IO34$ are correct names, but $ios2$ and $ioa34$ are not.

6.2 Creating a Control System with Real-Time χ

This section describes all the necessary steps to take if one wants to create an application using the real-time χ-kernel.

First, make sure that both Microsoft Visual C^{++} 4.2 and the driver software of the InterBus-S system are installed on the PC. The InterBus-S software should be installed in a directory called `IBSdriver`, instead of the default directory `IBS driver`. Both these programs are to be installed on the C drive of the PC. If components are installed elsewhere, the corresponding paths in the batchfile `chirt.bat` should be altered.

Second, set the real-time χ environment by typing at the prompt the following command: `setchirt`.

Third, an application should be designed. This program can be compiled with the command: `chirt <filename>`. This command yields an executable. This executable needs a configuration to run correctly. The creation of a configuration file is achieved by a small program called `mkconfig`. This executable is automatically available when the real-time χ environment is set. This utility program creates a configuration file. The identifier the program asks for is the name of the IO-channel that is used in the χ-program. The word in data array corresponds with the word in the data arrays that InterBus-S generates for its input and output. The bit position in the word is the bit position at which the actuator or sensor is installed. When an executable from an application is to be run, the name of the executable must be called followed by a whitespace and the name of the configuration file.

6.3 Test Results with Real-Time χ

The real-time χ-kernel designed for this research works. It is possible to create an executable based on a χ-model with a few simple steps. The only interface that can currently be used is the InterBus-S system.

The realization of an executable is a cooperation of modeler and implementor. The modeler must design the χ-model. The implementor is not able to do this because he has no knowledge of the desired functionality of the χ-model. When simulation has proven the model to be correct, the χ-code must then be adjusted according to the concept of IO-channels presented in Section 3.5. The modeler is not able to compile this pre-made control system because he has no knowledge on the interface system that will be used. The implementor compiles the χ-code of the control system and links the correct libraries for the interface system. The implementor must create the configuration files. The modeler has no knowledge of the precise configuration of the hardware on which the program is run.

The fact that a separate configuration file is used at the execution of the program works well. It is possible to run the same program on machines with the same function but a different configuration, assuming the same interface system is used. One only needs to use a different configuration file.

Chapter 7

IO-Communications Incorporated in the χ Kernel

If the χ language is to incorporate the changes suggested in Section 3.5, the compiler must be adjusted. These adjustments must add new language objects, like the suggested IO-channels, to the kernel and how to interpret these new items. In other words, what functionality these new objects have. In the case of the IO-channels, it is necessary to define the specific mechanism that is used when the control system designed in χ wants to communicate with its environment. The most common mechanisms that are used to communicate with peripheral equipment are examined first before a choice is made for a specific mechanism.

7.1 Commonly Used IO-Mechanisms

There are several ways in which a control process can accumulate data. This section presents the communication mechanisms most commonly used. In these mechanisms two different approaches can be distinguished.

The first possibility is to present data directly to the control process. This option itself can be divided further in two other options. The decision to communicate can be made by polling or an interrupt-driven approach.

The second possibility is to create an additional process that collects all communications. This process would then function as a sort of buffer in which IO-data is kept. This buffer is often called a mailbox.

7.1.1 Polling

Polling is the most straightforward of communication techniques. The control process runs its program. During execution, the control process constantly polls the appropriate addresses for data. The main problem with polling is the loss of valuable CPU cycles. The control program spends a lot of time checking all signals. If the physical system generates a lot of signals, as is the case in physical machines with a multitude of sensors, the CPU time needed to poll each of these signals will also increase. Eventually, this will lead to an unexceptably high load on the CPU due to the polling.

7.1.2 Interrupt-driven IO

Another possibility is to use interrupts to inform the control process that communication is initiated by a peripheral device. The advantage of this approach is that a control process never needs to wait for communications. The control process only responds when communication is possible. The peripheral devices initiate the communication.

The disadvantage is the following. Every time a significant change occurs in the state of the physical system, an interrupt is generated. This is not a problem when these changes are relatively rare. However, most communication in real-time control systems is generated by sensors that detect every change in the system. If these sensors present their information with interrupts, the result is a extremely large amount of interrupts of the control system. This will slow the execution of the control system considerably and might even compromise the timing requirements set on the control system. Interrupt-driven IO-communication is suitable for important messages but assigning an interrupt with a high priority to each signal would create serious problems. The controlled system interrupts the main program for each message that is received and truly important messages are stalled because messages with the same priority but of less importance are handled first.

7.1.3 Mailbox

It is also possible to create a process outside the control process to act like a kind of buffer that accumulates all IO-communications. Such a process is called a mailbox. The control process can then access this mailbox at any time to check the data that is needed. An advantage of this approach is that the design of this mailbox can be made to support the types of communication that the control process wants to

receive. For instance, blocks of data can be transferred instead of separate items. Two methods exist for a mailbox to buffer its data. It can use *queues* or *pipes*.

Queues

The queue is the easiest way to implement a buffer. Every data received is put in a queue. This data waits in this queue before it is processed. This data can be accessed using either LIFO or FIFO. The problem that can arise with this type of buffer is when the control system is emptying the buffer less often than new data is put in the buffer. The result is a growing queue that will stop growing only when the processor runs out of resources.

Pipes

This buffer type is best defined by an analogy with a pipe (hence the name). Consider a pipe in which marbles are dropped, each marble representing a data object. When data is received a marble is dropped in the pipe. At the other end of the pipe this data is occasionally picked up and used by an other system. The problem of the growing buffer does not occur because the pipe has a specified length. After the pipe is filled with marbles, no marbles can be added to it. However, this also might lead to errors as the marbles that are added to a full pipe drop to the ground, destroying the data they carry. Alternatively, the oldest marble is pushed through the pipe, destroying the oldest data.

7.2 Suggested IO-Mechanism for the χ Kernel

Before a solution is presented, consider once more the functionality of the IO-communications in χ. Not considering the changes to the χ language as presented in the previous chapter, a mechanism is needed that presents or holds one or more, most current, values of a specific input or output signal. The IO-channels that are to be used for IO-communications only establish communication between the environment and the control system. The IO-channels do not buffer input nor output. The values of the IO-signals must, therefore, be contained by a process outside the system.

The fact that a process, external to the control system defined in χ, is used for the values of IO-communications suggests that a mailbox must be used. For every address monitored, a mailbox must be created. This mailbox is large enough to hold one instance of the data type it is used for to store. The data that is put in this

mailbox is derived from either sensors, in case of input, or from the control system, in case of output. When new data is presented to this mailbox, the old data is overwritten. The value of this mailbox is adjusted or tested by the control system.

Chapter 8

Real-Time Aspects of χ

The previous chapters have dealt with the problems of how to incorporate the means for IO-communications in χ. If the χ formalism is to produce real-time applications there are several other problems to take into consideration. In this chapter, the problems that need solving are presented. To most of the questions that are posed in this chapter no specific answers are given. The reason for this, is that for most questions to be answered, a very specific knowledge is needed that would surpass the extent of this paper. Nonetheless, the directions in which the answers should be searched are presented.

8.1 Time and Timing in Real-Time χ

Consider the concept of time. In a simulation environment, simulation time is used. In most cases, this simulation time is not related to real time. The simulation time is simply increased after a set of actions has been calculated by the computer. It is initialized to zero when the model is started and is increased during the execution of the model. The χ formalism is no different. Using this simulation time, the model is able to time the various actions it makes. Time-outs used in the model are also calculated using the simulation time.

If a program is run in an environment, it is essential that the program and its environment operate using the same time scale. If this is not the case, major errors can occur due to miscommunication generated by the different concepts of time used by the program and the environment. When a model, designed in χ, is implemented, the simulation time used in the model must be connected in some way to the 'real' time, in this respect the time of the environment in which the implementation is run.

The decisions that must be made for this problem, lie on a very low level. The answers this question yields are therefore not considered to be part of this particular research. The fact that a coupling must be made between the simulation time and the environment time is an important one however.

8.2 Error Handling in χ

A difficult problem in the control of real-time systems is the handling of errors in the controlled system. Errors can occur due to deterioration of the controlled system. This is caused by wear and aging. Also, components have their tolerances and moving parts suffer from imprecise positioning. The ability to handle these errors is vital to a real-time control program. Control programs operating under error-free conditions are usually many times smaller and less complex than those programs that have to deal with errors.

If real-time machine control systems are to be generated using χ, a method to handle errors must be incorporated into the χ-formalism. The method that is introduced to do this must be consistent with the current χ syntax and semantics and at the same time contain as few new elements as possible. Two things must be kept in mind. First, the detection of errors must be done in one single place as much as possible. This is done to ensure a clear program text by avoiding repetitive declarations of the testing of the constraint monitors throughout the program. Secondly, if an error is detected, this must be immediately propagated to other parts of the model. The χ-formalism has a structure of processes working concurrently. If one of these processes detects an error this must be reported to the other processes in the model. This cannot be achieved by declaring the error detection global, as χ does not know the concept of global variables.

An important concept when dealing with error handling in a structured way, is the concept of exception. Using this concept, [Bee96] presents a method that incorporates exception handling in the formalism χ. To this method, [Bee96] adds a new mechanism based on constraints and constraint monitors. This new mechanism is then used to facilitate the use of exceptions. If the χ-formalism is to be used to create real-time machine control systems, the suggested method of exception handling should be studied. If found useful, the changes suggested by [Bee96] can be incorporated into the χ-formalism in such a way that they are consistent with the suggested changes for IO-communications.

8.3 Concurrency in χ

In the χ formalism, all processes are theoretically operating concurrently. If a χ model is implemented to run as a real-time application, how much of this concurrency is preserved in the functionality of the control program?

Considering the execution requirements of the control system it is very unlikely that all the processes that were distinguished in the χ model, must run concurrently in the application. Most of the processes in χ are only theoretically running concurrent while actually these processes are always waiting for some confirmation from another process. In practice, therefore, they are running sequentially. The same goes for the various parts of a control system in most industrial applications. Some parts of this control system must run concurrently while others are allowed to function sequentially.

Therefore, the χ formalism should not lose its concurrent character completely when an implementation is made of a χ model. The most ideal situation is when an implementor would be able to allocate the processes of the system into subsets of processes. All the processes in one of these subsets must run concurrently to all the processes in one or more of the other subsets, while allowing the processes within each subset to run sequentially after each other.

When the above is implemented in the χ formalism, another problem arises. When executing processes concurrently some of these processes will be more vital to the correct execution of the program than others. For example, when resources are low, it is more important that the processes run that deal with the monitoring of the emergency break, than that the processes run that make up the screen output for the user. So, if this mechanism is implemented into the χ formalism it will also be necessary to assign to each subset of processes a priority. This priority is then used to define the thread level at which the subset operates.

The problem of the preservation of concurrency is a very complex problem. It is likely that the answers to this problem can be found only after an extensive research. This research is left for others to pursue.

Chapter 9

Conclusions and Recommendations

9.1 Conclusions

If a control system wants to communicate with its environment, three things are needed.

1. **Signal**. There must be a communication signal. Whether this is an input or an output signal is irrelevant. The value of this signal is determined by the control system (in case of an output signal) or the environment (in case of an input).
2. **Timing**. The communication signal is sent at a certain point in time. This point in time can depend on decisions made by the control system when an output signal is considered. If an input signal is considered, the timing of the signal is defined by the events in the environment.
3. **Location**. The control system needs a location at which it can communicate.

The current χ-kernel does not support IO-communications. The χ-kernel meets the requirements mentioned above in a simulation environment. In a real machine control environment, the χ-kernel does not have the means necessary to define a certain hardware location. It is possible to design a control layer in which the χ code generated by the kernel is coupled to IO-routines specific for a certain interface system. By using this control layer the control system is able to communicate with its environment. The control layer is created by an implementor, because the creation of this control layer requires knowledge of the interface system that is used in the control.

The current χ-kernel does not truely support real-time IO-communications. A real-time program can be generated using the adjusted χ-kernel as developped during this project. However, with this adjusted kernel it is not possible to incorporate timing requirements into a control system. If the χ-kernel is to be used as a means to create real-time control systems, the simulation time used in the models of the control system must be coupled to the real time.

During this research it has become apparent that the curriculum followed by a student at the faculty Mechanical Engineering does not prepare him for research in the field of machine control. Knowledge on programming languages, interfacing, sensors and actuators is little and non-specific.

9.2 Recommendations

If a real-time χ-kernel is designed, the following should be examined (as described in Chapter 8):

- The coupling of the simulation time with real time.
- Error detection and error handling in χ.
- Concurrency and priority in the resulting executable.

To be able to fully examine a real-time χ-kernel, the χ-kernel must be made to run on a real-time platform. The platforms used in this research, DOS and Windows95, do not qualify as such.

The only interface system that is, as yet, been used to create machine control system is the InterBus-S system. If the validity of the suggested changes in χ is to be checked, libraries must be created for other interface systems. These libraries can then be used to confirm the correctness of the current real-time χ-kernel.

The current χ-formalism has a very limited Man-Machine-Interface (MMI). In machine control, a MMI is very important for the interaction between the control system an the operator. It should be examined if the current possibilities of the χ-formalism suffice to create an MMI. If a MMI is created, should this be specified by the modeler in the χ-model, or should an implementor design the MMI according to specific needs.

The ∇-operator should be made to work on discrete channels, as suggested in Section 3.4.

Bibliography

[Bee96] D.A. van Beek and J.E. Rooda, *A New Mechanism for Exception Handling in Concurrent Control Systems*, European Journal of Control, nr. 2, pp. 88-100, 1996.

[Ben94] S. Bennet, *Real-Time Computer Control*, Prentice Hall International (UK) Ltd., Hertfordshire, United Kingdom, 1994.

[Hor85] M.F. Hordeski, *Design of Microprocessor, Sensor & Control Systems*, Reston Publishing Company Inc., Reston, Virginia, United States of America, 1985.

[IBS93] Phoenix Contact, *InterBus-S, User Manual*, Phoenix Contact GmbH & Co., Blomberg, Germany, 1993.

[Nau96] G. Naumoski, *Real-Time Systems Control with* χ (draft version), internal memo, University of Technology Eindhoven, Faculty Mechanical Engineering, Section Systems Engineering, the Netherlands, 1996.

[Lei92] J.R. Leigh, *Applied Digital Control*, Prentice Hall International (UK) Ltd., Hertfordshire, United Kingdom, 1992.

Further Reading

[Ban91] B.R. Bannister and D.G. Whitehead, *Instrumentation: Transducers and Interfacing*, Chapman and Hall, London, United Kingdom, 1991.

[Bra89] U. Brasche and Ph. Sonntag, *Intelligent Sensors*, VDI/VDE Technologiezentrum Informationstechnik GmbH., Berlin, Germany, 1989.

[Bri94] J. Brignell and N. White, *Intelligent Sensor Systems*, Institute of Physics Publishing, Bristol, United Kingdom, 1994.

[Roo96] J.E. Rooda, *The Modelling of Industrial Systems*, Lecture Notes Number 4746, University of Technology Eindhoven, Faculty Mechanical Engineering, Section Systems Engineering, 1996.

[Sch93] A.W. van Schadewijk et.al., *Sensoren en Actuatoren*, Stichting Centrum voor Micro-Elektronica, Delft, the Netherlands, 1993.

[Sol94] S. Soloman, *Sensors and Control Systems in Manufacturing*, McGraw-Hill Inc., United States of America, 1994.

Appendix A

Introduction to the InterBus-S System

This chapter briefly explains the InterBus-S system. The system itself is presented, as well as its data transmission protocol, its hardware topology and its data addressing. The last two items are important if one wants to use the software created for this project. This software allows a model made with the χ formalism to be compiled in such a way that a machine control system is created.

For a complete review of the InterBus-S system, see [IBS93].

A.1 The InterBus-S System

InterBus-S is a bus system that transmits data between different types of control systems (e.g. programmable logic controllers, personal computers, robot controls, etc.) and input/output units to which sensors and actuators are connected. This data is transmitted by means of a serial transmission method.

InterBus connects sensor/actuator signals to the control or computer system (in general: host). InterBus fulfills two important tasks:

- The cyclic transmission of process states that change rapidly.
- The tailored transmission of parameter data for complex I/O modules and specialized process devices that are to be parameterized during operation (e.g. frequency inverters, robots, etc.).

A.1.1 General Structure and Method of Operation of InterBus

The InterBus system is designed as a ring structure. The central controller for this data ring is the controller board. It exchanges data transmitted serially within the data ring with the higher-level control or computer system and the lower-level InterBus devices. The exchange of data is carried out simultaneously and cyclically in both directions (full-duplex). This means that the system reads input from and sends output to the InterBus devices at the same time.

The InterBus system differentiates between two cycle types:

- The **identification cycle** (ID cycle) that is run to initialize the InterBus system or on request. In the ID cycle, all the ID registers of all the InterBus devices connected to the ring structure are read. This information is used to define the configuration of the system. If this identification has succeeded, all the devices are switched to data registers and only data cycles are transmitted.
- The **data cycle** that is responsible for data transmission. During a data cycle, the controller board updates the input and output of all the connected InterBus devices at the same time.

The InterBus system has several methods to check the data transmission for correctness. These will not be discussed here. The input on the controller board and the output to the devices only become valid when the data cycle has proven to have been executed without error. If an error occurs, the faulty data cycle is ignored, as the correction of the corrupted data requires more time than the execution of a new data cycle.

A.1.2 InterBus Protocol Sequence

1. Starting with a loopback word, the controller board clocks the output data into the InterBus data ring. At the same time, the controller board receives input data transmitted through the data ring by the InterBus devices.
2. The loopback word runs between the input and the output data through the entire data ring. When the controller board receives the loopback word, all output data has arrived at the corresponding InterBus devices and all input data at the controller board.

3. The controller board checks if the data transmission has been executed without errors.

4. If the loopback word has been received properly and the transmission carried out without errors, the output data becomes valid and the devices put the data to their output registers. At the same time, the controller board passes the read input data it received from the devices to the controller or computer system to be processed.

5. Afterwards, the controller board causes the devices to store the new input data on the data ring before a new data cycle is started.

A.2 Process and Parameter Data

InterBus uses two different sorts of data. The first sort is called **process data**. This is the simple input and output data that is transmitted during normal data cycles. The second sort is called **parameter data**. This parameter data consists of complex data records that are transmitted at the same time as the process data.

Parameter data is information:

- required by special devices to receive or process data, or
- configures or initializes special devices (e.g. parameterization of a frequency inverter), or
- required by the controller board to acquire the state information of special devices.

Devices that are able to process parameter data are called PCP devices. The Peripheral Communication Protocol (PCP) allows communication between these devices and the controller board. PCP is part of the InterBus protocol.

A.2.1 IBS Data Format: Motorola versus Intel

The InterBus-S master processor works with data in Motorola format. The PC that is used to control the InterBus-S system needs data in Intel format. This gives conflicts if the communications between the IBS master and the PC are not translated.

The InterBus-S system contains several macros to ensure proper data translation.

A.3 InterBus Topology and Data Addressing

This section is important if one wants to use the software presented with this paper.

A.3.1 InterBus Topology

This introduction to the InterBus system does not describe all the different components of the system. For this, the manual and the various catalogues that are supplied with the InterBus system should be consulted. To be able to understand the topology of the InterBus system, it is however necessary to have a rudimentary understanding of the general layout of the system.

In general, InterBus consists of three system components:

- the InterBus controller board
- the InterBus devices
- the cabling that connects the devices with each other as well as with the controller board.

The main branch of cabling that runs to and from the controller board is called the remote bus. To this remote bus, InterBus devices are connected with a remote bus branch line. This branch line allows branching into a next remote bus level. This branch allows further I/O stations as well as deeper branching (see Figure A.1).

Owing to its structure, InterBus offers a segment/device-oriented functionality that is used for switching segments on or off as well as comprehensive diagnostics for the entire system. The device numbers define the exact position of a device in the system. Using these numbers the switching and diagnostic process is able to perform its function. The device number consists of its bus segment number and its position in this bus segment (see Figure A.2).

A.3.2 InterBus Data Addressing

To ensure proper operation of the InterBus system, the process data and the parameter data must be assigned to the correct positions in the memory of the connected computer or control system. The InterBus system has two different addressing types for this purpose.

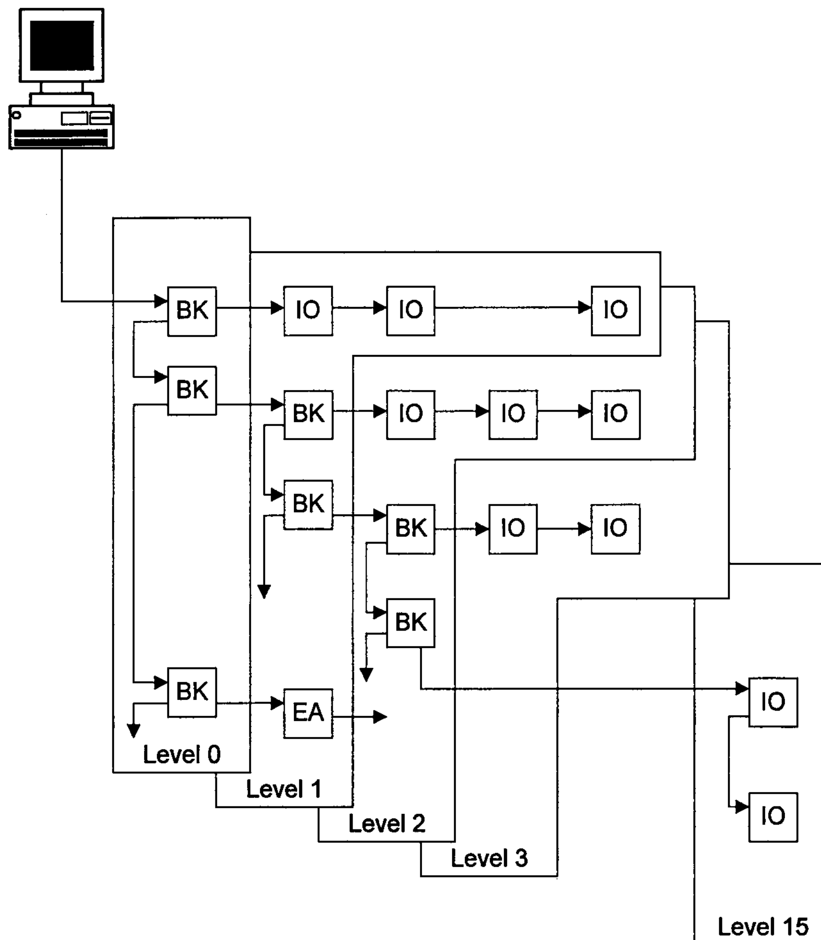


Figure A.1: Remote Bus Tree Topology with 16 Levels

Physical addressing is used for a quick and simple start-up of the InterBus system. It is a method that can be used providing that the system configuration does not change for the time being and/or that the address location of individual data must not be assigned freely. Physical addressing follows the settings automatically predefined by the controller board.

User-oriented addressing is used when one aims at high flexibility with regard to address assignment and changeability of the bus system. The address location of the InterBus system is adapted to the needs of the user or the presetting and requirements of a specific control system.

The decision which of these methods is used depends on individual requirements. The method used in the software used to support the theory presented in this research is

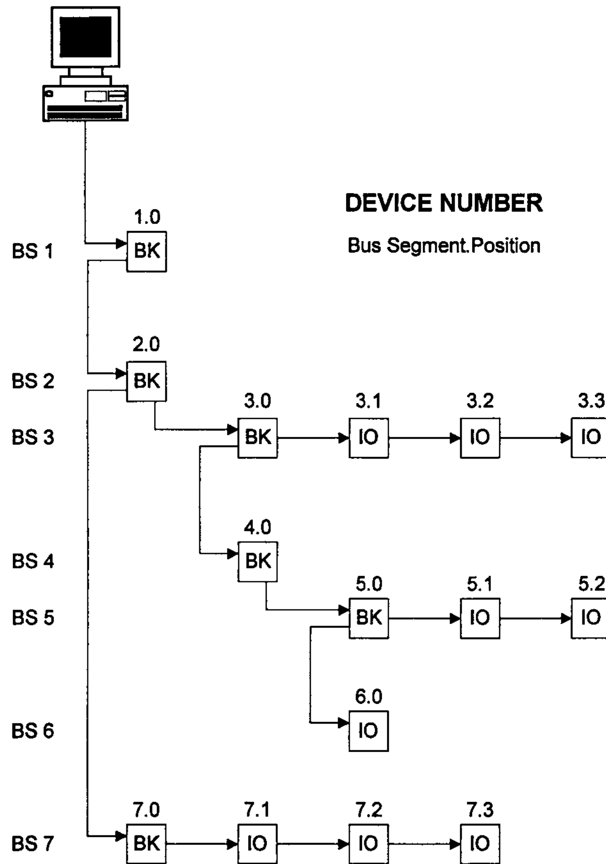


Figure A.2: Device Numbers of the Remote Bus

physical addressing. This method is the easiest to use and the system used for the experiments is not so difficult that it warrants the use of user-oriented addressing.

Physical Addressing

With physical addressing, the data of the various InterBus devices is stored in memory following an ascending order that depends on the physical location of the devices within the InterBus ring. The input received, is stored in the input section of the memory, and output is stored in the output section.

Starting with the controller board, there is a defined order in which the input and output is stored. The first device that the controller encounters is the first device to be stored in the memory, the second occupies the next unassigned location, etc. (see Figure A.3).

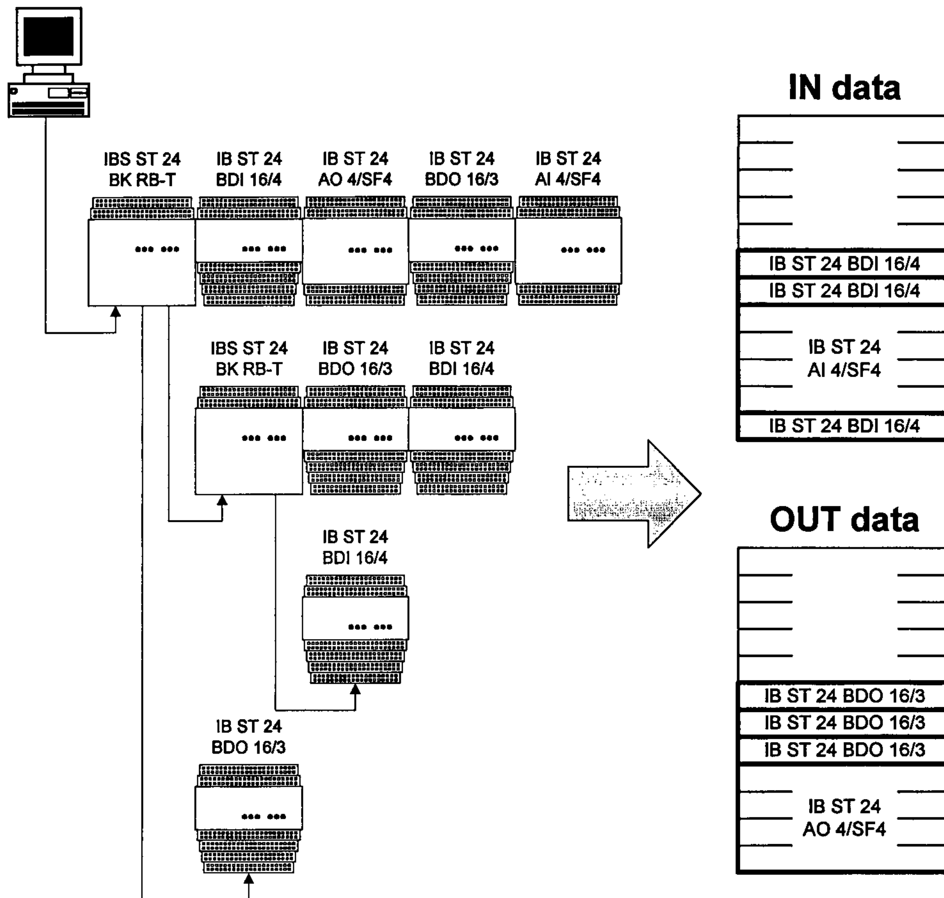


Figure A.3: Principle of Physical Addressing

Physical addressing can be generated automatically by the controller. Given the right commands, the controller board will identify the configuration, process data length, and type of all devices. According to this data, the controller board addresses the complete bus configuration. Devices that the controller board was not able to identify are ignored during addressing. The file that is created can be stored on the hard disk.

After every power up of the control system or every individual reset of the controller board, the controller board reads out the starting sequence and thus the configuration of the physical addressing from the parameterization memory and executes it automatically.

If the configuration of the system is changed (e.g. by adding or removing devices), it requires modifications in the program due to the fact that input and output addresses

will be shifted.

User-Oriented Addressing

User-oriented addressing is characterized by the fact that addressing of InterBus devices and special InterBus registers can be freely assigned to the memory of the controller board or the computer system independent of their physical order in the bus system.

User-oriented addressing is applied to:

- optimize memory division in the control or computer system,
- exclude address shifting when extending the system,
- allow a configuration to be changed without affecting addressing or
- to optimize comprehensibility of the address assignment.

The fact that addresses can be assigned freely allows switching system parts off or on within the bus configuration. In this way, when changing the system configuration, only one address list is changed instead of the all the addresses in the application program.

For devices with both input and output data, one can define the same address in the corresponding input and output process image of the controller/computer system.

Also, it is possible to assign an odd byte addresses to devices using only one byte address area. For devices to be addressed byte by byte during user-oriented addressing, 'byte gaps' produced during physical addressing can be filled without problems or restrictions. Devices that are addressed word by word must still be assigned to even byte addresses.

A.4 Working with InterBus-S

The InterBus-S manual is an extensive work on all the possibilities of the InterBus-S system. This amount of information makes it very difficult to discern the basics of the operation of the system. These basics are presented in this section.

A.4.1 Utilities for InterBus-S

With the InterBus-S manual, a disk is included called `SOFTWARE EXAMPLEPROGRAMS`. On this disk some very useful, pre-made utility programs can be found, under the path `\c\common`. The utility programs used to create the software for this research are given below. Only the names of the programs are given, without parameters or return types. For more specific information, see the source code on the disk mentioned above.

- `OpenHandles` opens the nodes for the DTI (data interface) and the MXI (mailbox interface). The nodes must be opened before communication can take place.
- `CloseHandles` closes the nodes for the DTI and the MXI. If the nodes are not closed properly problems can arise when the program is restarted. The reason for this is that the PCISA runs out of available node handles.
- `RequestResponse` sends a message to the MXI.
- `ConfirmationIndication` receives a message from the MXI. (Is used by WaitForMessage.)
- `WaitForMessage` polls for an expected message. (Is used by WaitForPosCnf.)
- `WaitForPosCnf` polls for a positive confirmation.
- `WriteData_I2M` writes process data to the DTI with Intel-Motorola conversion.
- `ReadData_M2I` reads process data from the DTI with Motorola-Intel conversion.

A.4.2 Start-up of the InterBus-S System

The controller board of the InterBus-S system has three different states. These are the states `ready`, `active` and `run`. Before the system is able to reach the state `active` it must first be `ready`. In the same way, the system must first be `active` before it can change its state to `run`. The different states are detailed below.

- `Ready` is the state in which the system starts. In this state, no data transfers are made.
- `Active` is the state that is achieved when a hardware configuration is detected and activated. In this state, ID cycles are run sporadically.
- `Run` In this state, the bus is ready for use. Cyclic data traffic on the bus is possible.

When one wants to work with the InterBus-S system, the system must first be put in the desired state. To do so, one must communicate with the system through the MXI (Mailbox Interface). Hexadecimal commands are sent to the MXI that are then executed by the system. (Before this is done, make sure that the handles to the MXI and the DTI are open by called the function `OpenHandles` mentioned in the previous paragraph.)

The commands used in the source code of the software written for this project are explained below in the order in which they are executed in the start-up sequence.

`0x1303`	The service `Alarm_Stop` causes a long reset on the bus. No data traffic is possible. Modules with process data set all their outputs to 0. If a data cycle is run, the service is executed directly after completion of the cycle. After execution of this service, the controller board is in the `Ready` state.
`0x0710`	This service, called `Create_Configuration`, causes the controller board to automatically generate a configuration from the currently connected hardware and to activate this configuration. After execution of this service, the controller board is in the `Active` state.
`0x0701`	The service `Start_Data_Transfer` activates the cyclic data traffic on the bus. On execution of this service, the controller board is in the `Run` state.

Chosen is for an automatically generated configuration because this is an easy way of dealing with different hardware configurations while still executing the same executable.

When the controller board is in the `Run` state, data traffic is possible using the utility programs `WriteData_I2M` and `ReadData_M2I` detailed in the previous paragraph.

For a complete overview of all available services, consult the user manual of the InterBus-S system.

A.4.3 Shut-Down of the InterBus-S System

To shut-down the InterBus-S system, the state of the controller board must be changed from `Run` to `Ready`. Although this is possible by calling the service `Alarm_Stop` immediately, it is more correct to use the command sequence explained below.

0x0702	The service Stop_Data_Transfer stops the cyclic data traffic on the bus. After execution of this service, the controller board is in the Active state.
0x1303	The service Alarm_Stop causes a long reset on the bus. No data traffic is possible. Modules with process data set all their outputs to 0. If a data cycle is run, the service is executed directly after completion of the cycle. After execution of this service, the controller board is in the Ready state.

Remember to close all node handles of the MXI and the DTI before you power down the system. This is done by calling the utility program CloseHandles, as detailed above. If this is not done, subsequent use of the program might result in errors because the PCISA runs out of available node handles.

A.5 InterBus-S Documentation

To ensure that the InterBus-S documentation is complete at all times, a list of contents is given. This lists gives every item that should be present, together with a small description. The InterBus-S documentation should contain the following:

- *Driver Software.* Driver software, tools and example programs for the IBS PC ISA SC board (3 disks).
- *General Introduction to the InterBus System.* Fundamentals of InterBus technology, method of operation and definitions. (Order No.: 27.45.21.1)
- *Quick Start Guide.* Quick start-up of the controller board. (Order No.: 27.47.87.9)
- *Description of the Driver Software.* Structure and functions of the driver software, method of operation and necessary settings. (Order No.: 27.45.17.2)
- *Services and Error Messages of Firmware 4.x.* Explanation of the G4 Firmware, library of all services and description of possible error messages. (Order No.: 27.45.18.5)
- *PCP.* Peripherals Communication Protocol Data transmission methods of InterBus. (Order No.: 27.45.16.9)

Appendix B

Library Text

When an executable is generated from a χ model to be used as a control program, several steps are taken as described in Chapter 6. During one of these steps, implementation-dependent information is added to the program. This implementation-dependent information describes the necessary knowledge of the interface system used to control the controlled system.

The information of the interface is presented in the form of a static library. In this paper, the InterBus-S interface system is used. For this interface system, a library is created. During tests executed during the research, functions described in this library are used to control the controlled system with the InterBus-S interface system.

In this Appendix, the library created for the InterBus-S system is presented. The library is written in C^{++} code. It is compiled using Visual C^{++} Version 4.2. Presented here, is the full C^{++} code that has been added to the code of the InterBus-S interface system. In the code remarks are incorporated on the use of the functions. No further comment is given.

B.1 Library Source Code

B.1.1 IBSCHI.H

```
/* Header file for IBS functions to be used with CHI */

/* Definition of WIN95 vesion */
#define IBS_WIN_95_VERSION
```

```
/* Standard include files */
#include <windows.h>
#include <conio.h>
#include <stdlib.h>
#include <string>

/* IBS utilities include */
#include <ibsutil.h>

/* InterBus-S include files */
extern "C"
{
#include "ibddiw95.h"
#include "ddi_macr.h"
#include "ibsg4uti.h"
}

/* Declaration of buffers
static USIGN16 OutData[254];
static USIGN16 InData[254];
static USIGN16 buffer[1024];*/

/* Function declarations */
IBDDIRET IBDDIFUNC StartIBS();
IBDDIRET IBDDIFUNC StopIBS();
bool BitWrite(string cName, INT16 BitValue);
INT16 BitRead(string cName);
```

B.1.2 IBSCHI.CPP

```
/*************************************************************/
/* This is the source-code for a Static Library created to      */
/* implement IO-communications into CHI using the InterBus-S    */
/* system.                                                      */
/*                                                              */
/* This library is intended for use with Windows95              */
/*                                                              */
/* File name:  ibschi.cpp                                       */
/* Editor:   Jasper van Rosmalen                                */
/* First release: 06-01-1998                                    */
```

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

#include <assert.h>
#include "ibschi.h"

/* Buffer definitions. All have been initialized at maximum size. */
extern USIGN16 InData[];
extern USIGN16 OutData[];
extern USIGN16 buffer[];

/* Definition of power of 2 */
static USIGN16 pow2[]={1,2,4,8,16,32,64,128,256,512,1024,\
                       2048,4096,8192,16384,32768};

/* Defintion of Handles */
static IBDDIHND DTI_Handle;
static IBDDIHND MXI_Handle;

/* Command definitions */
/* The commands are hexadecimal numbers that are sent to the MXI. */
static USIGN16 AlarmStop[]={0x1303, 0x0000};
static USIGN16 CreateConfig[]={0x0710, 0x0001, 0x0001};
static USIGN16 StartData[]={0x0701, 0x0000};
static USIGN16 StopData[]={0x0702, 0x0001, 0x0000};
static USIGN16 ConfirmDiag[]={0x0760, 0x0000};

/******************************************************************/
/* StartIBS is a function that creates a setup of the IBS-system. */
/* This is a generic startup. It does not foresee in specific     */
/* configurations.                                                */
/*                                                                */
/* In: Node Handles for the DTI and the MXI                       */
/* Out: ERR_OK or ERROR                                           */
/******************************************************************/

IBDDIRET IBDDIFUNC StartIBS()
{
IBDDIRET IBDDIFUNC ret;
IBDDIHND IBDDIFUNC ret1;

/* Initialization of OutData */
```

```
INT16 i;

for (i=0; i<254; i++){
OutData[i]=0;
}

/* Open Handles */
ret1=OpenHandles(1, 1, &DTI_Handle, &MXI_Handle);

if(ret1!=ERR_OK){
ret=ERROR;
printf("OpenHandles unsuccesfull.\n");
printf("Error code: 0x%x\n",ret1);
}

/* Reseting the IBS-system */
RequestResponse(MXI_Handle, AlarmStop);
ret1=WaitForPosCnf(MXI_Handle, 0x9303, 5, buffer);
if (ret1!=ERR_OK){
ret=ERROR;
printf("Unsuccesfull AlarmStop\n");
printf("Error code: 0x%x\n",ret1);
}

/* Creating system configuration */
RequestResponse(MXI_Handle, CreateConfig);
ret1=WaitForPosCnf(MXI_Handle, 0x8710, 5, buffer);
if (ret1!=ERR_OK){
ret=ERROR;
printf("Unsuccesfull CreateConfig\n");
printf("Error code: 0x%x\n",ret1);
}

/* Start IBS data transmission */
RequestResponse(MXI_Handle, StartData);
ret1=WaitForPosCnf(MXI_Handle, 0x8701, 5, buffer);
if (ret1!=ERR_OK){
ret=ERROR;
printf("Unsuccesfull StartData\n");
printf("Error code: 0x%x\n",ret1);
}
```

```
/* Reseting all OutData */
WriteData_I2M(DTI_Handle, 0, 254, OutData);

return(ret);
}


/****************************************************************/
/* StopIBS stops the IBS-application. It leaves the system in a  */
/* safe state. It also closes the node handles.                   */
/*                                                                */
/* In: Node handles for the DTI and the MXI                      */
/* Out: ERR_OK or ERROR                                           */
/****************************************************************/

IBDDIRET IBDDIFUNC StopIBS()
{
IBDDIRET IBDDIFUNC ret;
IBDDIRET IBDDIFUNC ret1;

/* Stop data transmission */
RequestResponse(MXI_Handle, StopData);
ret1=WaitForPosCnf(MXI_Handle, 0x8702, 5, buffer);
if (ret1!=ERR_OK){
ret=ERROR;
printf("Unsuccesfull StopData\n");
printf("Error code: 0x%x\n",ret1);
}

/* Reset the IBS system */
RequestResponse(MXI_Handle, AlarmStop);
ret1=WaitForPosCnf(MXI_Handle, 0x9303, 5, buffer);
if (ret1!=ERR_OK){
ret=ERROR;
printf("Unsuccesfull AlarmStop\n");
printf("Error code: 0x%x\n",ret1);
}

/* Close Node Handles */
ret1=CloseHandles(DTI_Handle, MXI_Handle);
```

```
 if (ret1!=ERR_OK){
  ret=ERROR;
printf("Unsuccesfull CloseHandles\n");
printf("Error code: 0x%x\n",ret1);
}

return(ret);
}


/*****************************************************************/
/* BitWrite writes to a bit nominated by the ID of the channel.  */
/*                                                               */
/* In: DTI Handle, ID and bit value                              */
/* Out: void                                                     */
/*****************************************************************/

void BitWrite(string cName, INT16 BitValue)
{
USIGN16 Data;
USIGN16 NewData;
LinkedList *pId;

pId=SearchId(cName.c_str());
assert(pId);

Data=OutData[pId->Word];
if (!BitValue){
if ((Data & pow2[pId->BitPos])==0){
}
else{
NewData=Data - pow2[pId->BitPos];
}
}

if (BitValue){
if ((Data & pow2[pId->BitPos])!=0){
}
else{
NewData=Data + pow2[pId->BitPos];
}
```

```
}

OutData[pId->Word] = NewData;
WriteData_I2M(DTI_Handle, 0, 254, OutData);
}


/******************************************************************/
/* BitRead evaluates the value of a bit at a designated position. */
/*                                                                */
/* In: DTI handle, ID                                             */
/* Out: Bit value                                                 */
/******************************************************************/

INT16 BitRead(string cName)
{
USIGN16 Data;
INT16 BitValue;
LinkedList *pId;

pId = SearchId(cName.c_str());
assert(pId);

ReadData_M2I(DTI_Handle, 0, 254, InData);
Data = InData[pId->Word];

if((Data & (pow2[pId->BitPos]))!=0){
BitValue = 1;
}
else{
BitValue = 0;
}

return(BitValue);
}
```

B.2 Additional Utilities

Some additional utilities are needed for the software written for the tests conducted during the project. A linked list is used to store the information the bit locations of different IO-positions. These IO-positions are defined by an identifier, a word in the data arrays of InterBus and a bit position in such a data array. How this linked list is created is explained.

The utilities also include functions for saving the linked list mentioned above to file or how to read a configuration file and create a linked list from this information.

B.2.1 Ibsutil.h

```
/* Ibsutil.h
*/
extern "C"
{
#include "stdtypes.h"
}

#ifndef H_LinkedList
#define H_LinkedList

struct LinkedListStr {
struct LinkedListStr *pNext;

/* Identifier for IO-communication */
    char   Id[6];        /* Identifier */
    INT16  Word;     /* Word in OutData */
    INT16  BitPos;   /* Bit position in Word */
};

typedef struct LinkedListStr LinkedList;

void Init_LinkedList();
void DeleteAll();

void NewIOid(char *pName, INT16 iWord, INT16 iBitPos);

int ReadConfig(char *Filename);
```

```
void CreateConfiguration(void);
void SaveConfig(void);

LinkedList *SearchId(const char *cName);

#endif
```

B.2.2 Ibsutil.cpp

```
/************************************************/
/* Ibsutil.cpp holds several utility functions  */
/* to be used with the real-time CHI kernel.    */
/* The functions are written for the IBS-system.*/
/************************************************/


/* Include directories */
#include "Ibsutil.h"
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <iostream>
#include <conio.h>

/* Definition of variables and types */
static LinkedList *pHead;
FILE *stream;
char cNameTemp[6], cWordTemp[6], cBitPosTemp[6];
int iWordTemp, iBitPosTemp, i, ch;


/************************************************/
/* Init_LinkedList initializes a linked list.   */
/*                                              */
/* Input: void                                  */
/* Output: void                                 */
/************************************************/
void Init_LinkedList() {
pHead = 0;
```

```
}


/*********************************************/
/* NewIOid creates a new member of the linked   */
/* list.                                        */
/*                                              */
/* Input: Identifier of IO-position             */
/*   Word in the memory array of IBS            */
/*   Bit position in memory word                */
/* Output: void                                 */
/*********************************************/
void NewIOid(char *pName, INT16 iWord, INT16 iBitPos) {
LinkedList *pLl;

pLl =(LinkedList*) malloc(sizeof(LinkedList));
assert(pLl);
strncpy(pLl->Id,pName,4); pLl->Id[4]='\0';
pLl->Word = iWord;
pLl->BitPos = iBitPos;

pLl->pNext = pHead;
pHead = pLl;
}


/*********************************************/
/* DeleteAll empties an existing linked list.   */
/*                                              */
/* Input: void                                  */
/* Output: void                                 */
/*********************************************/

void DeleteAll() {
LinkedList *pLl;

while(pHead) {
pLl = pHead->pNext;
free(pHead);
pHead = pLl;
}
```

```
}


/**********************************************/
/* SearchId searches the linked list for an ID. */
/* If this ID is found, a pointer to the       */
/* structure is returned.                       */
/*                                              */
/* Input: ID that is searched                   */
/* Ouput: pointer to ID                         */
/**********************************************/
LinkedList *SearchId(const char *cName) {
LinkedList *pLl;

for(pLl=pHead;pLl;pLl=pLl->pNext)
if(!strcmp(cName,pLl->Id))
    break;

return pLl;
}


/*************************************************/
/* ReadConfig reads a data file containing a       */
/* configuration declaration for an IBS-system.    */
/* The Configuration is put in a Linked List.      */
/*                                                 */
/* Input: FileName                                 */
/* Output: 0 - insuccesfull termination            */
/*   1 - succesfull termination                    */
/*************************************************/

int ReadConfig(char *FileName){
if ((stream=fopen(FileName,"r"))==NULL){
cout << "File does not exist" << endl;
exit(0);
}
else{
/* initializing Linked List */
Init_LinkedList();
DeleteAll();
```

```
/* reading and interpreting file */
ch=fgetc(stream);
while (ch!=EOF){
if (ch=='<'){
ch=fgetc(stream);
i=0;
while (ch!=','){
cNameTemp[i]=(char)ch;
i++;
ch=fgetc(stream);
}

ch=fgetc(stream);
i=0;
while (ch!=','){
cWordTemp[i]=(char)ch;
i++;
ch=fgetc(stream);
}

ch=fgetc(stream);
i=0;
while (ch!='>'){
cBitPosTemp[i]=(char)ch;
i++;
ch=fgetc(stream);
}

iWordTemp=atoi(cWordTemp);
iBitPosTemp=atoi(cBitPosTemp);

/* creating Linked List */
NewIOid(&cNameTemp[0], iWordTemp, iBitPosTemp);
}

ch=fgetc(stream);
}

fclose(stream);
}
```

```
return(1);
}

/************************************************/
/* CreateConfig creates a configuration which is   */
/* then stored in a LinkedList. This List is saved */
/* when CreateConfig is terminated.                */
/*                                                 */
/************************************************/

void CreateConfiguration(void){
char Id[9];
char cWord[8];
char cBitPos[7];
LinkedList *pLl;

INT16 Word;
INT16 BitPos;

char ch[6];
char con[6];

ch[2]='1';
con[2]='N';

Init_LinkedList();
DeleteAll();

while (ch[2]!='2'){
cout << endl << "SYSTEM CREATION (bitwise)" << endl;
cout << "1 - Define new IO-position" << endl;
cout << "2 - Save and Quit" << endl;
cout << "Make your choice: " << endl;
ch[0]=sizeof(ch)-1-2;
cgets(ch);

if (ch[2]=='1'){
cout << endl << "Type identifier of IO-position (max. 4 char.): ";
Id[0]=sizeof(Id)-1-2;
cgets(Id);
```

```
pLl=SearchId(Id+2);
if (pLl!=0){
cout << "This identifier already exists!" << endl;
cout << "Continue? (Y/N)" << endl;
con[0]=sizeof(con)-1-2;
cgets(con);
}

if (con[2]=='Y' || con[2]=='y' || pLl==0){
cout << endl <<"Type Word in Outdata array: ";
cWord[0]=sizeof(cWord)-1-2;
cgets(cWord);
Word=atoi(cWord+2);

cout << endl << "Type bit position in word: ";
cBitPos[0]=sizeof(cBitPos)-1-2;
cgets(cBitPos);
BitPos=atoi(cBitPos+2);
NewIOid(Id+2, Word, BitPos);
con[2]='N';
}
}
}

SaveConfig();
}

/**************************************************/
/* SaveConfig saves the current configuration to  */
/* file.                                           */
/**************************************************/

void SaveConfig(){
LinkedList *pLl;
FILE *stream;

char cName[11];

cout << endl << "Give name of configuration file (max 8 char.): ";
cName[0]=sizeof(cName)-1-2;
```

```
cgets(cName);
if ((stream=fopen(&cName[2],"w"))==NULL)
cout << "File \""<< &cName[2] << "\" could not be opened" << endl;
else{
for(pL1=pHead;pL1;pL1=pL1->pNext){
fputs("<", stream);
fputs(pL1->Id, stream);
fputs(",", stream);
fprintf(stream, "%i", pL1->Word);
fputs(",", stream);
fprintf(stream, "%i", pL1->BitPos);
fputs(">\n", stream);
}
fclose(stream);
}
}
```