Software design for the control system of a digital telephone exchange

Ligtenberg, M.F.J.

Award date:
1990

5663

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Electrical Engineering
Digital Systems Group (EB)

SOFTWARE DESIGN FOR THE CONTROL SYSTEM OF A DIGITAL TELEPHONE EXCHANGE

by M.F.J. Ligtenberg

Master Thesis
Eindhoven, August 1990

Supervisors: **Prof. Ir. M.P.J. Stevens**
Ir. M.J.M. van Weert

SUMMARY

At the Digital Systems Group (EB) of the Department of Electrical Engineering at the Eindhoven University of Technology, a digital telephone exchange is being developed. Quality constraints for this exchange are ensured by using a (4,2)-redundancy system. The exchange also uses non-blocking TST-networks for the central switch array.

This report describes the development and implementation of a control system for this exchange. The CASE-tool ProMod is used to model the control in three consecutive stages. Implementation is done in C using the iRMX II.4 multi-tasking operating system.

Development revealed some inconsistencies in ProMod regarding the step from requirements model to the models used in stages two and three. Still, the software for the control has been implemented and tested, and works according to the specifications. Only the functions concerned with interfacing still have to be developed.

CONTENTS

1. INTRODUCTION

At the Digital Systems Group of the faculty of Electrical Engineering at the Eindhoven University of Technology, a digital telephone exchange is being developed. Since 1986 a project group of five students on average (trainees or graduates) has been working on this project. First under supervision of Prof. Ir. A. Heetman and later taken over by Ir. M.J.M. van Weert and Prof. Ir. M.P.J. Stevens.

In May 1989 a first implementation of the hardware was expected shortly, giving rise to the need for a system that allowed the exchange to be tested and controlled. In the first place its task was to allow the prototype exchange to be tested. Later, it could be extended to a full-size control. [Verhoof, 1990] describes the so called requirements model for such a control. This report describes the revision of this requirements model, further development using the CASE-tool ProMod, and the implementation of the software.

A survey of the complete exchange is given in chapter 2. Chapter 3 discusses the CASE-tool ProMod and the methods used for modelling the control. How the control was developed using these methods is described in chapter 4. Chapter 5 shows how the model was implemented using the multi-tasking operating system iRMX. Finally chapter 6 gives some concluding remarks and recommendations.

2. THE DIGITAL TELEPHONE EXCHANGE

2.1 Pulse Code Modulation and Frame structure

This telephone exchange handles digital signals. An analog telephone signal has to be sampled before it can be transmitted over a digital communication link. At each sample moment, the signal has to be quantified by means of an analog-to-digital converter. The sample frequency, recommended by the CCITT is 8 kHz and the number of levels is 256. This way, the analog signal is converted to 8 bit time slots. An A-law or u-law companding Pulse Code Modulation (PCM) is used and the bitrate for one telephone signal is 8 (bits) * 8000 (samples per second) = 64 kbit/s.

By using time division multiplexing (TDM) it is possible to send more than one telephone call over the same line. The sampled data of each call is stored in a certain time slot. The compilation of these time slots is called a frame. Such a frame is repeated every 125 microsec because of the 8 kHz sample frequency. For this exchange, 64 channels are multiplexed on one line resulting in a bitrate of 64 * 64 kbit/s = 4 Mbit/s on one line. The original exchange has 64 of these lines resulting in a capacity of 4096 channels (i.e. 4096 half telephone calls). In figure 2.1 the structure of an incoming or outgoing PCM line has been given.

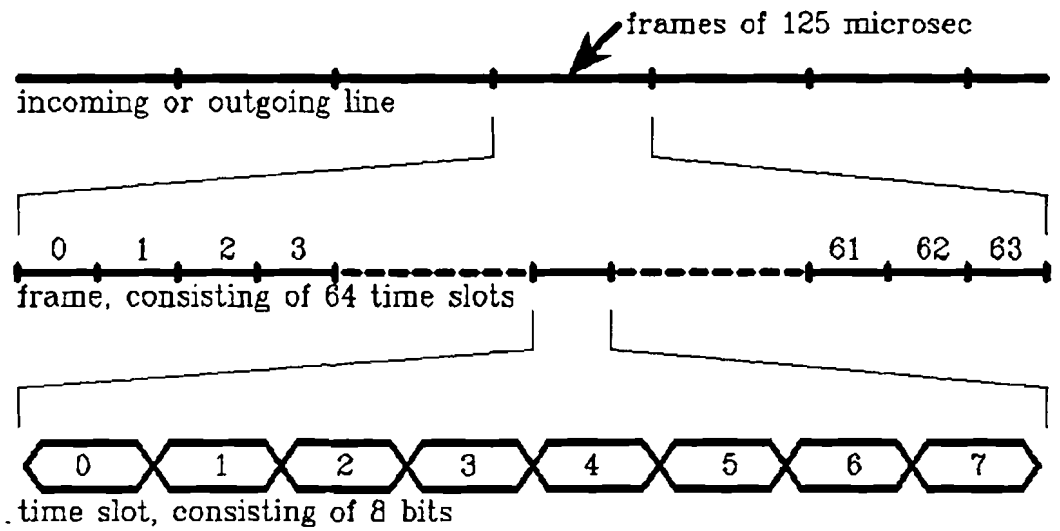


Figure 2.1: Frame structure of a PCM line

On average, a subscriber uses his telephone for about 6% of the time. This means that the traffic for one subscriber is 0.06 Erlang. The exchange has 64 time slots on every line so that it can handle 64 Erlang/line. However, considering that the blocking probability in the concentrators should be smaller than 0.1%, Erlang's equation indicates that there should be no more than 44 Erlang of traffic on every line. (Blocking can occur in the concentrators when a request for a call comes in, but there are no free time slots) This leads to a maximum of 44/0.06 = 750 subscribers for every line which means that the slots on one line are occupied for 44/64 = 70% of the time on average. Therefore, an exchange with 64 incoming (and outgoing) lines can handle about 64 * 750 = 48000 subscribers and has a capacity of 64 * 64 = 4096 Erlang.

2.2 Specifications and General structure

Typically, a telephone exchange is used to connect telephone lines coming from different places so a call can be routed to its destination. For this digital telephone exchange, this means that any time slot on any incoming line can be routed to any time slot on any outgoing line. However, there are some stringent requirements for this process. The most important ones are:

Reliability:
The exchange may be down for only a few minutes a year on average (During a period of 40 years it may be down for only 2 hours). Therefore it is necessary to install redundant systems because singular systems cannot meet this requirement.

Test facilities and maintenance:
The exchange should test parts in which errors occurred (possibly when there is little traffic, e.g. during the night) and disconnect them if they are found to be erroneous.

Non-blocking:
The exchange should be non-blocking, which means that if a free input and a free output are available, it should always be possible to connect the two.

Capacity:
The exchange should be able to handle 4096 half telephone calls which means 64 incoming (and outgoing) lines and 64 time slots/line. Although the prototype has got only 4 incoming (and outgoing) lines with 64 time slots/line resulting in an exchange that can handle 256 half telephone calls (see [Timmer, 1990]).

These requirements lead to the general structure of the exchange shown in figure 2.2.

The main part of the exchange is the switching network. This part performs the switching functions. It is a non-blocking Clos network (see [Clos, 1953]), to ensure that any incoming slot on any incoming line can be connected to any outgoing slot on any outgoing line. This is accomplished by using twice as many time slots on the internal lines of the exchange as externally. That is, external lines use 64 time slots/line, whereas internal lines use 128 time slots/line.

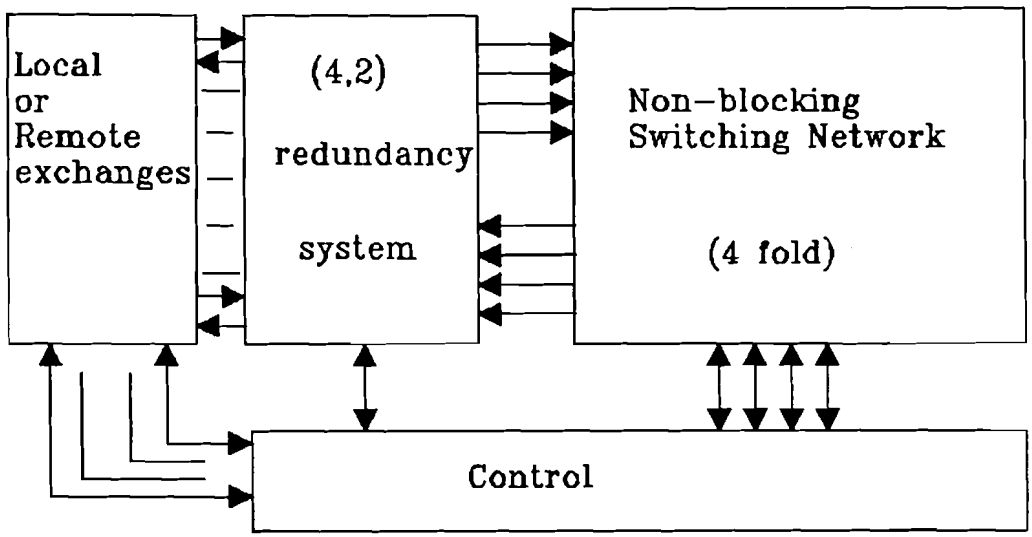


Figure 2.2: General structure of the exchange

Because the requirements for reliability are very high, the exchange has a (4,2)-structure. This means that the incoming 8-bit data word (one time slot) is coded into four 4-bit data words (see [Krol, 1983]). These four code words are handled separately by (four) fault isolation areas in the system. That is why the switching network is four fold. The idea is to place every fault isolation area in a separate chip. When the data words leave the exchange, the four code words that have travelled through isolated fault areas are decoded to form the original 8-bit data word (the incoming time slot). The (4,2) redundancy system takes care of the coding and decoding.

The advantage of using the (4,2)-concept is that several errors that may occur can be corrected because the four 4-bit words contain redundant information. Even when an entire switching network fails, the original 8-bit data word can still be regained because it takes only two of the four code words to restore the original data word. Another advantage is that the decoders of the redundancy system can indicate when an error has occurred in one of the fault isolation areas, thus providing opportunities for implementing diagnostic services in the control.
The disadvantages of the (4,2)-concept are the multiplication of the networks and the complexity of the decoders.

The exchange can be connected to various remote or local exchanges via 64 incoming lines and 64 outgoing lines. The control handles the incoming requests from outside (other exchanges or possibly the supervisor of this exchange) and controls the switching networks to satisfy these requests. It also checks if errors have occurred when a connection is broken down and runs diagnostic tests on the parts of the

exchange in which these errors have occurred. The control shuts these parts down if they appear to be broken and signals the supervisor that there is a defect in the exchange.

So far, the designed control was primarily intended to control the prototype exchange. That is, the supervisor should be able to set up and break down connections possibly with some diagnostic abilities. The protocol for communicating with other exchanges has not been given much thought yet, although [v. Engelen, 1988] discusses the design of a concentrator and gives some indication towards communicating with the exchange. But [Verhoof, 1990] shows that this communication is not sufficient.

2.3 The Central Switching network

2.3.1 The TST structure

The primary function of a telephone exchange is to set up a connection between two (or more) subscribers. This means that a particular slot on an incoming line must be connected to (probably) another slot on one of the outgoing lines. This requires two switching functions, time switching and space switching. These two functions are explained in figure 2.3.

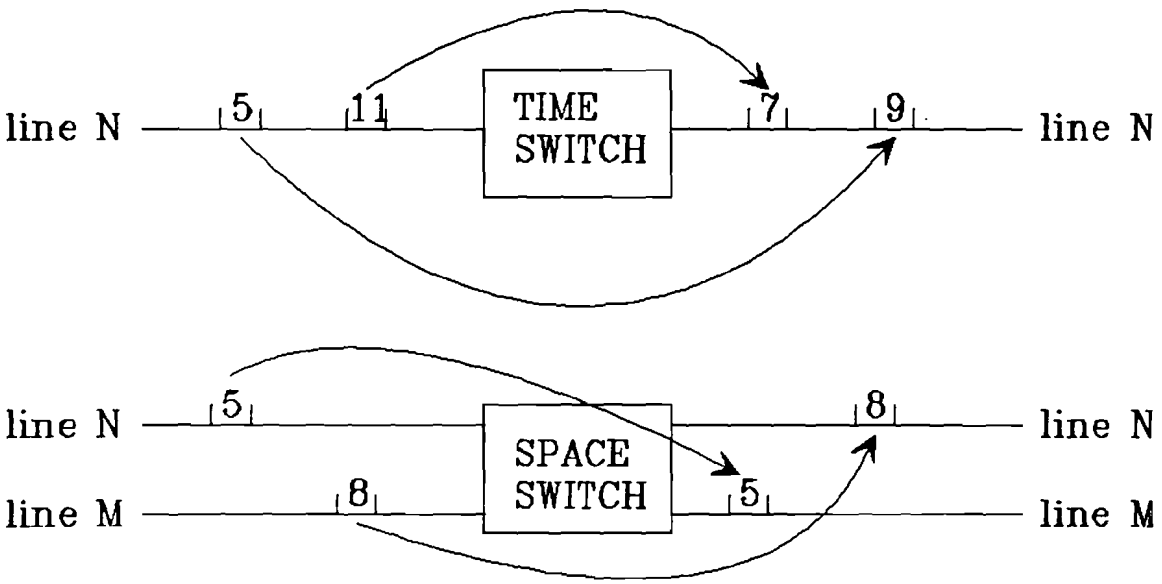


Figure 2.3: Basic switching functions

Time switching involves switching a slot to another slot on the same line. Space switching involves switching a slot to the same slot on another line. So to change slots you need a time switch and to change lines you need a space switch.

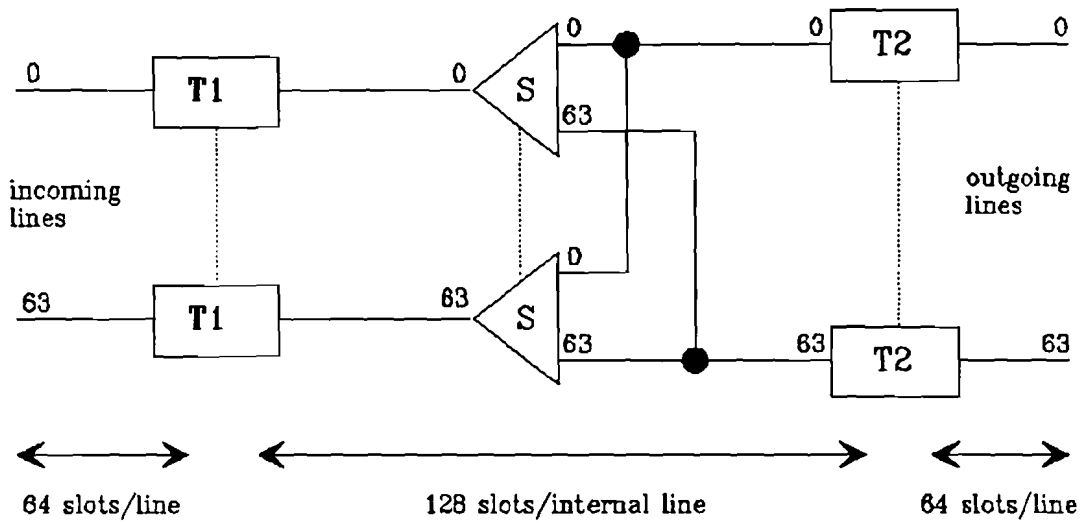


Figure 2.4: The TST-network

The central switching network has a TST structure. All lines are first time switched, then space switched and finally time switched again. This structure can be seen in figure 2.4. There are 64 incoming lines with 64 time slots/line. These enter the first stage of 64 time switches. Each time switch has one input line and one output line. On the input line there are 64 time slots of 4 bits (because the original 8-bit datawords were coded by the (4,2)-redundancy system into four 4-bit code words first). To achieve a non-blocking network, the number of internal time slots has to be twice the number of external time slots (see [Clos, 1953] or [Ronayne, 1986]). Therefore, the output line of the time switch has 128 time slots of 4 bits. The output of each first-stage time switch (T1-switch) is connected to a space switch in the second stage.

In the second stage, 64 space switches each have one input line and 64 output lines, with each line carrying 128 time slots of 4 bits. In a space switch, every slot of the input line is switched to one of the output lines (The output line is selected independently for each slot). The sequence is not changed, so they leave the space switch in the same order as they came in. Each of the output lines of a space switch (S-switch) is connected to a time switch in the third stage.

In the third stage, 64 time switches (T2-switches) each have one input line and one output line. The input line is connected to one of the output lines of all the space

switches. The input line has got 128 time slots of 4 bits. The output line has got 64 time slots of 4 bits. So after leaving the TST-network, the format is back to 64 time slots/line.

2.3.2 Structure of the switches

The time switches have to transfer incoming PCM data to other time slots. Therefore the incoming data is stored in a random access memory (the data memory RAMD). This is done sequentially, so every incoming slot is stored in the next memory cell. The order in which the data memory is read out, is stored in a connection memory (RAMC). This connection memory is read out sequentially. Each outgoing slot has got a corresponding memory cell. The data in this memory cell indicates which time slot in the data memory has to be read out.

The space switches receive the data from the first time switches and demultiplexers send the data to several outgoing lines. These switches don't need a data memory because the incoming slot is routed through right away. In the connection memory of a space switch every memory cell corresponds to an incoming slot. The data in this memory cell indicates to which of the outgoing lines that incoming slot has to be switched.

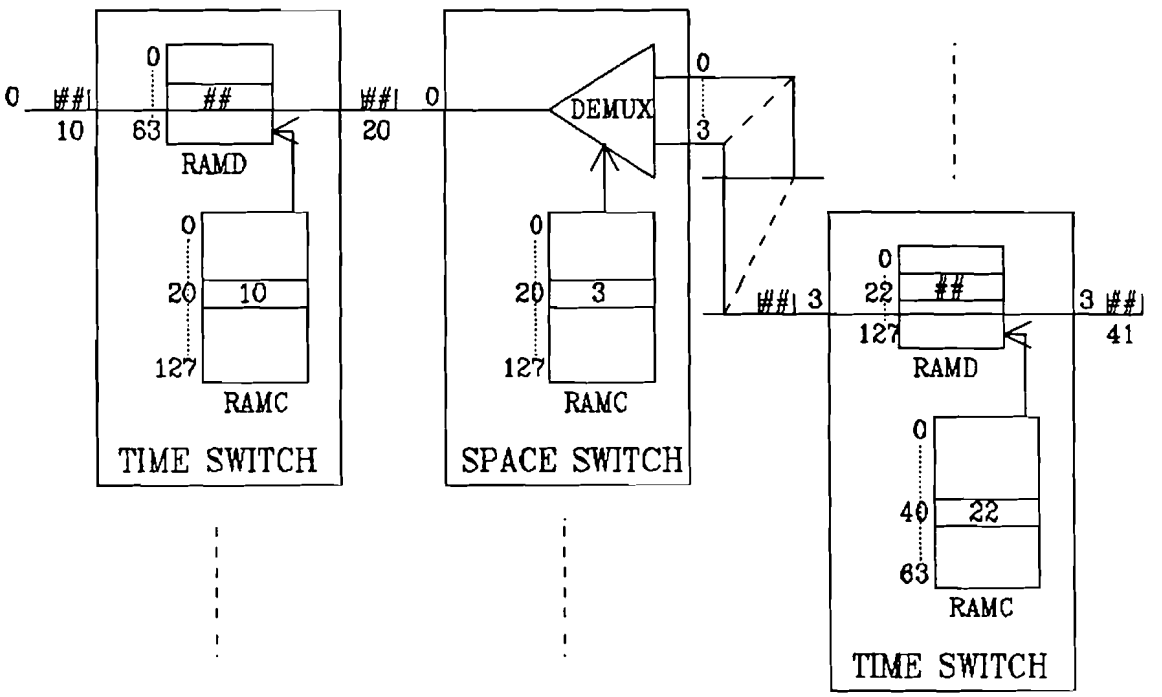


Figure 2.5: Routing through a TST-network

The third stage time switches are the same as those of the first stage. Only the data memory and connection memory have different sizes because a T1-switch has 64 incoming slots and a T2-switch has 128 incoming slots.

Figure 2.5 shows a route through a TST-network (of the prototype exchange with 4 incoming and outgoing lines) from a subscriber on the incoming line 0 and time slot 10 to a subscriber on the outgoing line 3 and time slot 41. The data memories are read in and the connection memories are read out, using a time slot counter for the addressing. Due to internal delays, a time slot sent out by the data memory of a T1-switch (during time slot N), arrives two time slots later in the data memory of a second time switch (i.e. it is stored in time slot N+2). Furthermore, a time slot read out in a second time switch (during time slot M), leaves the TST-network one time slot later (i.e. during time slot M+1).

The control system can read and write data to and from the connection memories of the switches. By writing it can make a connection, by reading it can check the contents of the connection memory.The protocol for the read and write operation is described in [Timmer, 1990].

2.4 The (4,2)-concept

2.4.1 The principle

The required reliability for the exchange cannot be reached with very reliable components alone. Some kind of redundancy should be used to ensure reliable operation over a long period of time. For this reason, the (4,2)-concept has been used. This concept was developed by Th. Krol for Philips (see [Krol, 1983]), who gives lectures on this subject at Eindhoven University of Technology.

According to this concept, an 8-bit data word (one time slot) is coded into four 4-bit data words. These four data words are handled separately by isolated identical systems so that there are four areas in which errors may occur. However, because these errors occur independently in one (or possibly more) of the four fault isolation areas, the redundancy in the information yields some error-correcting capabilities. The original 8-bit data word can be regained when two of four 4-bit data words are correct. Thanks to the redundancy in the information the following errors can be corrected:

- Complete failure of one of the four systems.
- Two random bit-errors in two different systems.
- If one of the four systems is known to be incorrect, a random bit error in one the other systems can be corrected.

The exchange with (4,2)-redundancy coding is depicted in figure 2.6. For every incoming line there is a coder and for every outgoing line there is a decoder. Figure 2.6 shows the coder/decoder configuration for incoming line n and outgoing line m. The coder codes the original data word into four 4-bit words and sends them to

different but identical TST-networks. The routing through these TST-networks is the same for the four 4-bit data words that were derived from one 8-bit data word. After leaving the TST-networks the four 4-bit data words are decoded by the decoder to the original 8-bit data word again. If possible, errors are corrected by the decoder.

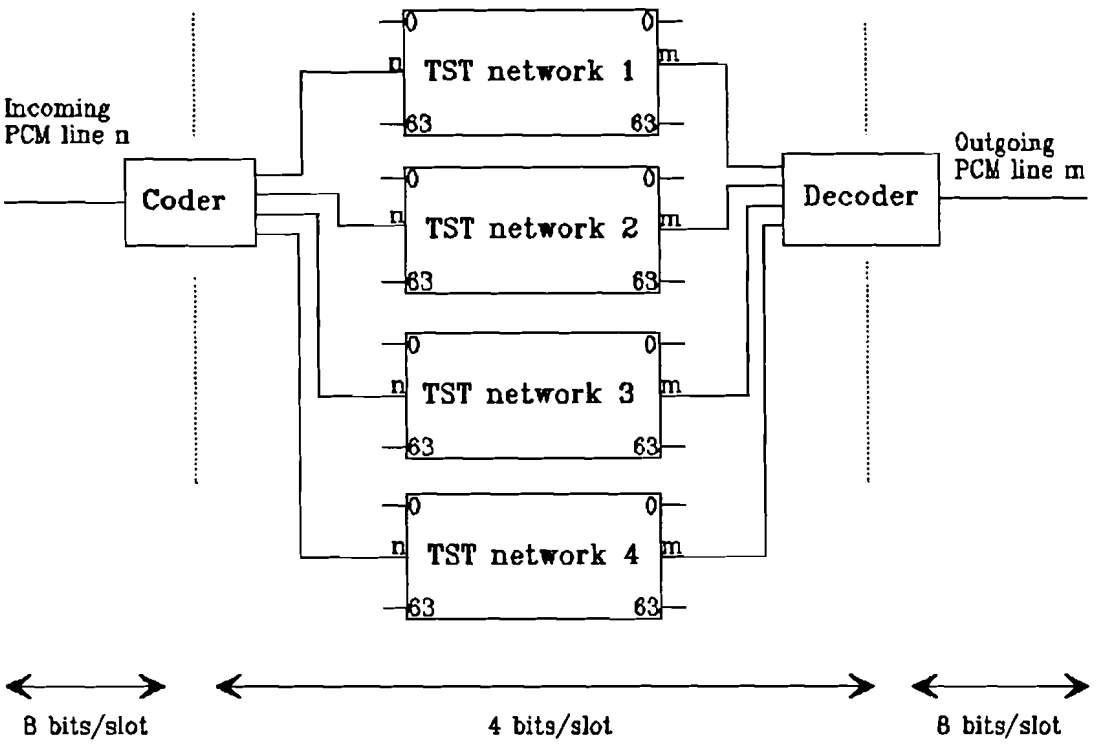


Figure 2.6: The exchange with (4,2)-system

2.4.2 The decoder

The decoders decode the four 4-bit data words and correct errors if possible. The decoders are also the only parts of the redundancy system that communicate with the control. For every outgoing line there is one decoder that takes care of all the 64 time slots on that line.

Each decoder has four incoming lines and one outgoing line. The incoming lines carry 64 slots of 4 bits per frame and the outgoing line carries 64 slots of 8-bits per frame. From the incoming slots, the decoder reconstructs for each slot the original 8-bit data word. For each slot, the decoder has a register in which it stores the so-called 'error/modus vector' for that slot. This vector indicates the errors that have been found and the mode of decoding for a slot. These registers can be read by the control

system so that it can locate defects in the TST-networks. The control system can also write to this register to set the mode of the decoder.

An error/modus register contains one out of 31 possible values. Only 12 of them can be written to the register by the control (the 12 modes in which a connection can start). The others occur when there has been an error and the decoder changes the vector and/or mode to indicate this. There are 3 kinds of modes for the decoding.

'Random mode' is normally used when a connection is set up. This means the connection uses all four fault isolation areas (TST-networks), using the maximum error-correcting capability of the decoder. There are 12 possible error/modus vectors in this mode to indicate various errors that have occurred. Only one of them can be written to the decoder by the control system to set up this mode.

'Erasure' mode is the mode in which the connection uses only three fault isolation areas, leaving a few error-correcting capabilities. This can be necessary when the control knows that there is an error in one of the areas or when one of the areas is being repaired. There are also 12 vectors in this mode and 4 of them can be written to the decoder by the control system (one for every area that should not be taken into account).

Finally there is the 'single mode' in which only two areas are used. In this mode there are no error-correcting/detecting capabilities. Although the original 8-bit data word can be regained from two correct 4-bit words, there is now no way of knowing if the decoded word is correct. There are 6 vectors in this mode that can all be written to the decoder by the control system. One for every combination of two areas that are not taken into account.

When no connection is set up through a particular slot, the 'idle' vector should be written to the error/modus register for that slot (the vector 0). In that case, the decoder will not decode for that slot.

During operation, the value of an error/modus vector is changed whenever the decoder detects an error. Before a connection is broken down the error/modus vector is read by the control system, which makes it in the long run possible for the control system to localise errors. If more than two non-correctable errors occur during a connection, the decoder will generate an interrupt (a non-correctable error is an error that does not belong to any of the four categories discussed on page 8). The control system must check the contents of the error/modus vectors of the decoder to determine which slot was responsible for the interrupt.

For more explicit information about the error/modus vectors and decoder hardware, see [Cornelisse, 1990], [Aggenbach, 1988] or [v. Duin, 1989].

2.5 The Prototype Exchange

The structure of the TST-networks as described in section 2.3.1 causes a problem

because of the high number of interconnections after the space switches. Every space switch has got 64 output lines leading to 64 * 64 = 4096 interconnections between the space switches and the final time switches. The surface needed for these interconnections would be 1.6 cm * 1.6 cm. This does not fit on one chip. Therefore, [Timmer, 1990] describes a prototype exchange with 4 incoming (and outgoing) lines thus reducing the number of interconnections after the space switches to 4 * 4 = 16. This prototype exchange is depicted in figure 2.7.

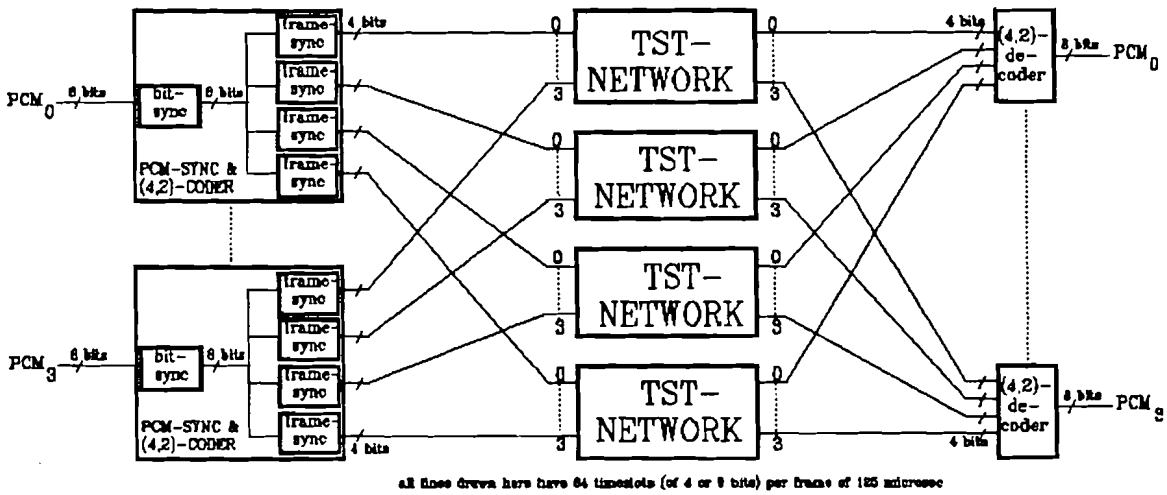


Figure 2.7: The prototype exchange

The exchange has 4 incoming PCM lines (PCM_0 to PCM_3) that carry 64 slots of 8 bits each, resulting in an exchange that can handle 256 half telephone calls. Because the incoming lines have different phases than the internal clock of the exchange (and in plesio-synchronous operation also different frequencies), bit synchronisation is needed. Because the time slots of the incoming PCM lines will also be out of phase with the time slot counter in the exchange, frame synchronisation is needed. Bit synchronisation and frame synchronisation together constitute the PCM synchronisation. After (4,2)-coding the data is sent to the TST-networks with 4 incoming (and outgoing) lines each. (4,2)-decoding is needed to restore the original data words.

[Timmer, 1990] also shows that with the set up of the prototype it is possible to realise a 4096 Erlang exchange in 1.5 micron CMOS technology. This is possible by using 16 incoming (and outgoing) lines with 256 time slots each. Also resulting in an exchange that can handle 16 * 256 = 4096 half telephone calls. Interconnection problems are avoided by using fewer lines and the prototype simulations suggest that the increased speed (of handling 256 slots/line instead of 64 slots/line) of 16 Mbit/s can be realised.

3. THE PROMOD MODEL

3.1 Introduction

Designing a control system for the digital exchange is a rather complicated job. Therefore, it is necessary to use structured methods in the development. These methods must both be standardized as well as formal. Standardized, because other people have to be able to co-operate and check or continue the work. Formal, because that is the only way to cope with vagueness.

For a long time, system development was treated as an arcane art. It was only slowly recognised by designers as a problem in engineering to be solved using the same methods and conceptual tools used in other engineering disciplines. Today a number of methods and tools have been created to aid in every phase of system development. An integrated set of such methods and tools is called computer-aided software engineering (CASE).

ProMod is such a CASE-tool. It supports each of the first three phases of a project life cycle using a single method for each phase. Those three phases are:

- Requirements Analysis and Definition
- System design
- Program design

Figure 3.1 shows how a system is developed using ProMod. The first phase begins with the user's ideas. A structured model of the system is created, containing all technical requirements expressed in a consistent way. A graphical method is used in which the analyst manipulates data flows and creates entries in a data dictionary. This way, the system is visualised using only a few simple symbols and conventions.

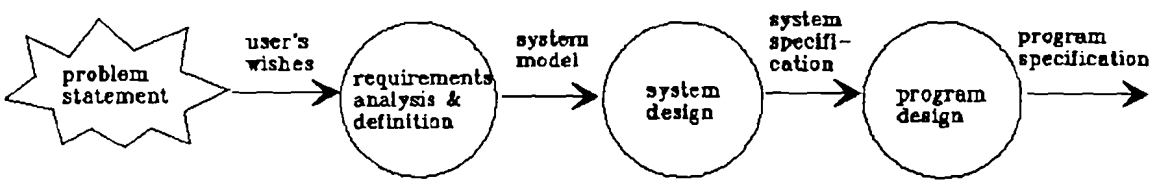


Figure 3.1: The ProMod project model

During system design, a general structure for the system is developed. This is done using the principles of the modular design method. That is, the designer builds a hierarchical modular structure made of modules, functions, subsystems and datatypes, encapsulating the requirements model developed in the first phase of the design.

The task of program design is to create the specifications which outline each of the program functions in pseudocode. That is, the functions are described in the user's natural language plus a small number of formal constructs (e.g. DO, ENDDO, IF, ELSE, THEN). The idea is to get a program specification that is very specific but not yet tailored to a specific programming language. The formal constructs used in this phase are general programming constructs that are available in nearly every high-level programming language.

The three phases as supported by ProMod version 1.7a are described in more detail in the next sections.

3.2 Requirements Analysis and Definition

3.2.1 Introduction to structured analysis

ProMod's first phase provides tools to perform structured analysis. The key to structured analysis is the manner of thinking about the problem. Conventional specifications describe functions, or tasks, which the system is to perform. In most cases each function, taken by itself, is easy to understand but difficulties become apparent when interrelations are considered. In structured analysis attention is given not only to functions, but to data flow as well. Because all data flows are explicitly identified, no part of the system can remain hidden because all information used and produced by a function has to be specified.

ProMod supports the method described in [Hatley, 1987] although it does not completely cover this method. This method by D.J. Hatley and I.A. Pirbhai is an adaptation of earlier methods for structured analysis, which can be found in the so-called Yourdon-literature. E. Yourdon is the writer of one of the first books on this subject (see [Yourdon, 1975]) and publisher of a series of books which presented extensions, adaptations and improvements on this method.

[Hatley, 1987] tries to capture a system design in 2 phases. First a requirements model is developed, and after that an architecture model is derived from the first model. ProMod's first phase largely corresponds to developing a requirements model. There are some minor differences (see [Verhoof, 1990]) but the main idea is the same. The architecture model differs quite a bit from the ProMod approach. In [Hatley, 1987] the architecture model is developed using an intermediate stage called the 'enhanced requirements model'. Purpose of the architecture model is to:

- show the physical entities that make up the system

- define the information flow between these physical entities
- specify the channels on which this information flows

As admitted by the authors of [Hatley, 1987] this model is not suited for mapping software requirements into software modules. It is mainly intended to model systems with a lot of hardware. ProMod's method for developing an 'architecture model' with the phases 'system design' and 'program design' is much more versed towards software design. Therefore, the methods described in the next sections are those that are supported by ProMod and which have been used to model the control software for the exchange.

So the first phase of ProMod largely corresponds to developing a requirements model in [Hatley, 1987] and this is described in the next section. The architecture model from this book has not been used because ProMod provides methods more suited for software development.

3.2.2 The Requirements Model

The requirements model is used to describe what the system must do and should leave as much space as possible to the designer so he can decide how to implement the system. A structured analysis model for real-time systems is built using 5 fundamental components:

- Data Flow Diagrams (DFDs)
- Control Flow Diagrams (CFDs)
- Data Dictionary (DD)
- Mini Specifications (MSPs)
- Control Specifications (CSPs)

Basically it is a layered model of processes and the data and control that flows between them. On the top level only one process exists, namely the entire system itself.In a DFD the processes are shown with the data flows, and in the corresponding CFD the same processes are shown with their control flows. Each process can again be decomposed into a number of other processes and data and control that flows between those processes. This leads to another DFD and CFD with those new processes. Both diagrams can also be represented in one diagram, the flow diagram (FD). An example of a FD can be seen in figure 3.2.

In these diagrams processes are represented by circles. They act upon the data (and control) flows that enter them and are part of the hierarchical structure of processes. Every process is described in a Mini Specification (MSPEC). It specifies in concise terms each detail of the process' functional requirements. The MSPEC shows how the incoming flows are handled and how the outgoing flows are produced.Each process can also be decomposed. How it is decomposed is indicated in the corresponding FD (or DFD and CFD). Data flows are represented by the directed arcs. They represent one or more data items that flow according to the direction of the arc. Control flows are represented by dashed directed arcs and represent control

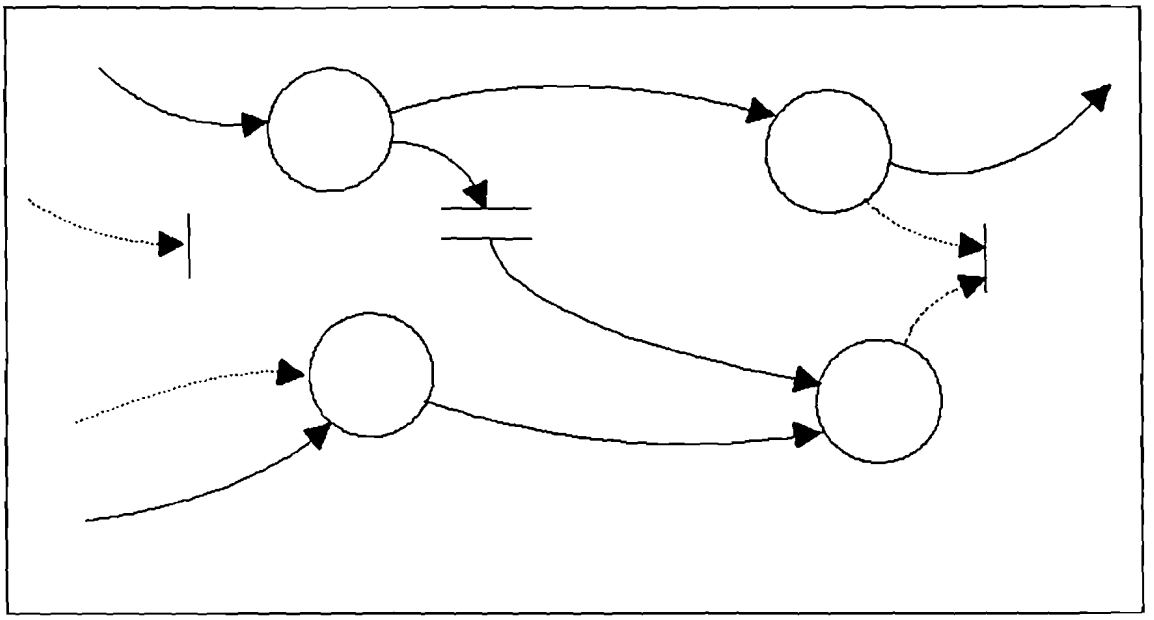


Figure 3.2: The elements in a flow diagram

signals flowing according to the direction of the arc. Lose flows that seem to come from or go nowhere are connected to the parent process. All flows are described in the Data Dictionary (DD).

The two parallel horizontal lines in figure 3.2 represent a store. Its function is to sustain the information on its incoming flows even after its source stopped generating it. The vertical line (bar) indicates an input or output of a control specification (CSPEC). A CSPEC indicates how to act upon the incoming control flows. It also tells how to produce control flows that leave a bar. Furthermore, CSPECs may activate or terminate certain processes depending on the incoming control flows.

There is a special FD called the context diagram. In this diagram there is only one process that represents the entire system. It shows the connections with the outside world (the so-called terminators). This is the highest level diagram from which the decomposition of the system begins.

In this stage of the development, processes are assumed to be data triggered and infinitely fast. That is, as soon as a process has got the necessary input (via flows) to perform its task, it will do so infinitely fast. This means that the model is highly idealised.This is not a problem in this stage because requirements and functionality are the main issues.

This section has given a very brief overview of the requirements model. It is beyond the scope of this report to describe it in more detail. The interested reader is referred to [Hatley, 1987].

3.2.3 ProMod tools for structured analysis

ProMod supports the development of a requirements model. It provides an integrated set of tools to visualise and check the work of the analyst. The three most important tools in this phase are the graphics editor, the text editor and the SA (Structured Analysis) analyzer.

With the graphics editor all flow diagrams can be designed as well as state diagrams (for in the CSPECs). The editor is menu driven and therefore very simple to use. Objects can be added, changed or deleted. Before a diagram is saved and incorporated in the project library, ProMod checks for errors. Diagrams with errors cannot be incorporated in the project library. When they need to be saved, this can be done with the OUT option. This option creates a separate file for the diagram but does not put it in the library.

With the text editor, all text in the CSPECs, MSPECs and DD can be typed in. This editor is not a very sophisticated editor but external editors can be used as well. Again, text with errors cannot be incorporated in the project library.

The SA analyzer performs checks on the consistency of the model and produces a number of reports. Scope of the analysis and extent of the reports can be selected through the menu options.

3.3 System Design

3.3.1 Introduction to the method

During ProMod's second phase a general structure for the system is developed based on the requirements model. The method used in this phase is modular design. In modular design, a hierarchical modular structure of modules, functions, subsystems and data types is built. A module is a unit containing all functions and data that are implemented together. The guiding principle in modular design is that if the module must be altered, its alteration must not affect other modules.

A module may be considered as self-enclosed with its internal functions and data hidden from other modules or functions outside that enclosure. The module is accessible to other modules through rigidly controlled 'doors' (interfaces). This way, the principle of 'information hiding' is used, so that design decisions made within a given module and details of implementation are hidden from other modules.

The next sections describe how modular design is used in ProMod's second phase.

3.3.2 Modular Design

As stated in the last section, modules use the principle of information hiding. Such modules (sometimes called abstract data types) usually specify only one data type with all the necessary access functions. Communication between modules takes place through interfaces which say nothing about the hidden information and are, consequently, independent of implementation within any module. Figure 3.3 shows an example of the structure of two modules. In this figure the larger rectangles represent modules, boxes represent functions and ovals depict data.
Important characteristics of modules are:

- Cohesiveness; the module should represent a single function, or multiple functions where interfaces are largely internal to the module.
- Independence; the module should be loosely coupled, having the simplest possible interface with other modules.

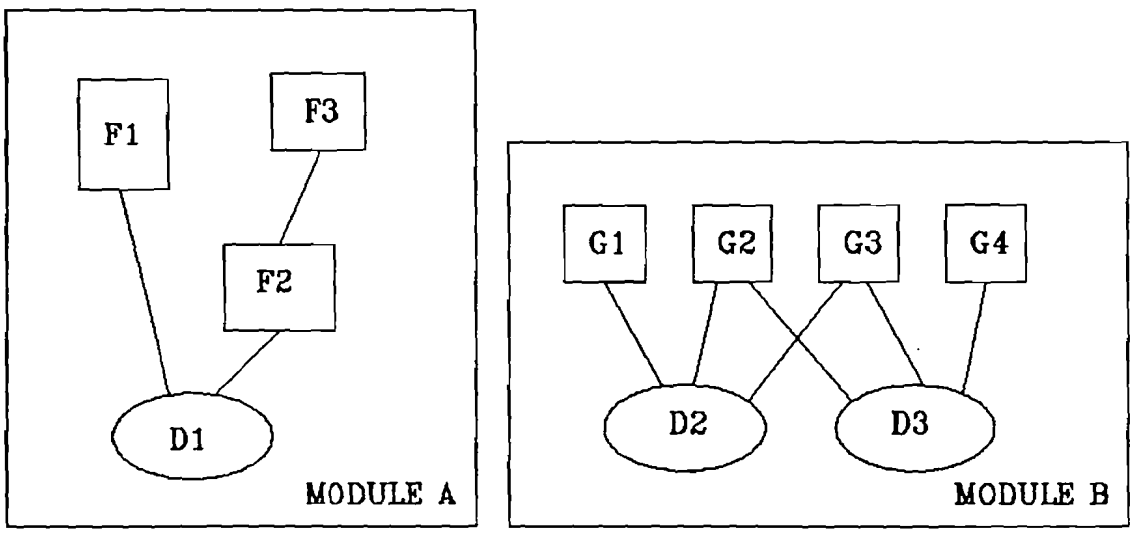


Figure 3.3: Structure of modules

The 'doors' through which other modules communicate with functions within any module are called 'export interfaces' and 'import interfaces'. The export interface of a module provides functions (export functions) exported from the module. Other modules (which may be considered as using the module) access the module through the export interface.

The import interface of any module is the door through which that module 'imports' the export functions of other modules. At the using module, these functions are called import functions. Conventions of modular design require that the using module declares which functions it intends to import from other modules. An example of

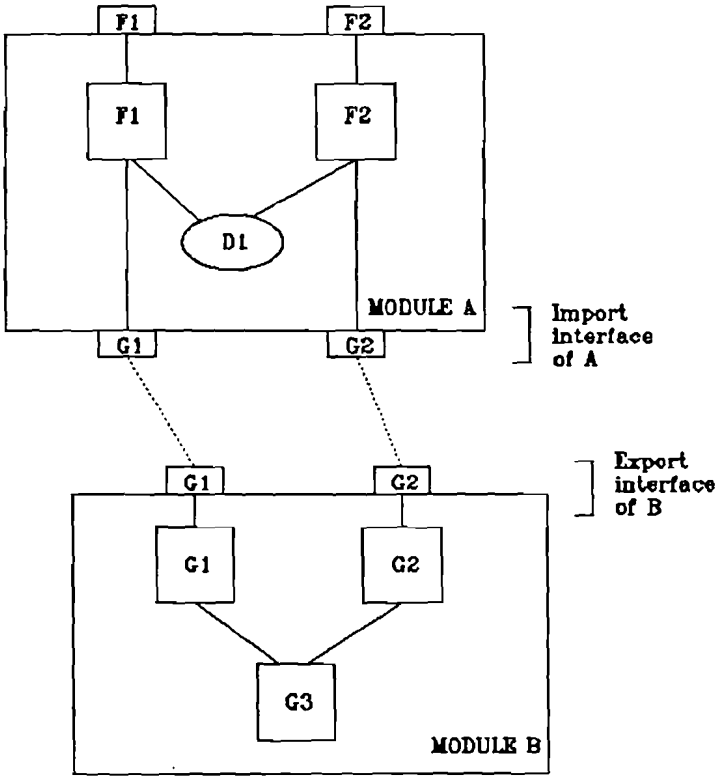


Figure 3.4: Modules with their interfaces

modules with their interfaces can be seen in figure 3.4. In this figure module B provides two functions (G1 and G2).Module A uses these functions through its import interfaces. Note that function G3 in module B has no interface and therefore cannot be used outside its module. Module A provides the functions F1 and F2 for higher-level modules.

A system is built of modules linked together in a hierarchical structure through their export and import interfaces. In this hierarchical structure, a module can use the export functions of lower level modules. In figure 3.5, module A uses module B. That is, functions in A call functions in B. A also uses modules C and E. Module B uses D and module C uses D and E. Modules on the same level are not allowed to use each other's functions. So B is not allowed to use C's export functions (and the other way around). These cyclic function calls are permitted only within a module.

Subsystems can also be declared. A subsystem may be sort of a 'supermodule', or a combination of all modules of a given worker or working group, or a combination of modules using certain larger data structures. Subsystems generally should be defined only when the module count becomes too large for easy understanding and management.

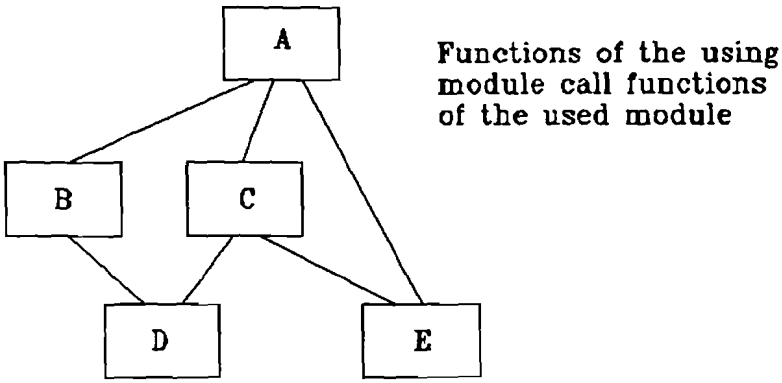


Figure 3.5: Relationships between modules

In structured analysis all data is declared in the data dictionary (DD). During modular design the equivalent is the data type dictionary which contains the data types (DTs). This makes it easier to generate different data having the same structure, attributes, or features. The different data types are stored in this dictionary. The user may use these data types to define local data in any function, module, or subsystem.

So modular design is based on the definition of modules that may contain data (used locally in that module) and export functions (functions the module provides to higher-level modules). The module should also declare import functions (the functions that this module uses from lower-level modules). Furthermore, it may also contain internal functions. These functions are local and cannot be used outside the module.
After finishing the modular design, the system usually has a structure as in figure 3.6. Nearly all systems, regardless of nature or size, share this structure consisting of four layers in the finished design.

Modules of the bottom layer define the basic data structures on which the system is based and the primitive access functions having to do with single elements of a data structure or type. The third layer contains managing modules whose functions manipulate several data structures or types. Modules of the second layer implement the problem-oriented functions that solve the user's problem. And modules of the upper layer contain control functions which combine and manage the problem-oriented functions of the lower layers in order to obtain the required system model. The relative number of modules in each layer is generally expressed by the onion-shaped form shown in figure 3.6. Of course, modules may be layered within any of the four layers.

3.3.3 ProMod tools for modular design

ProMod provides an integrated set of tools to visualise and check the work of the

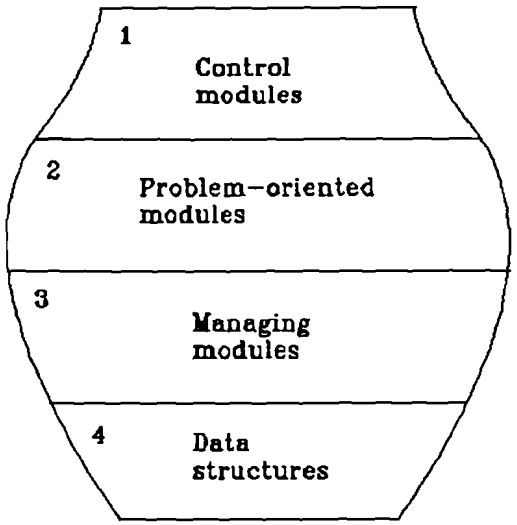


Figure 3.6: Layered Nature of Systems

designer. The text editor, the charts designer, the MD analyzer and the transformation option are the four most important tools in this phase.

All module definitions and data type declarations are entered with the text editor. This text editor is used in all three phases.

With the design charts option, visual representations of the module structure can be constructed. Although these figures can be edited, it is not suitable for actually designing a module structure because the changes are not incorporated in the project library (Changing modules has to be done in the textual declarations of the modules). It is more a way of visualising an intermediate or final design.

The MD analyzer performs global consistency checks on module interfaces, subsystem structures, and declaration and use of data. Scope of the analysis and extent of the produced documentation can be selected by the user.

The most valuable tool in this phase is the transformation option. When selected, it turns the system model developed during structured analysis (the requirements model) into a suggested system design that may be expanded and modified during modular design. All data items defined in the data dictionary of the requirements model are transformed into data types in the data type dictionary. Processes are transformed into functions. For every flow diagram (FD) a module is created which contains the functions used in that FD. This way, the hierarchical structure of the FD's is transformed into a hierarchical structure of modules. Furthermore, for every store a module is created with access functions to the data in the store (information hiding). This automatically performed transformation usually gives a system design that needs a lot of rearranging, but as a first basic structure it is quite useful. During modular design, functions can be added and/or grouped together in modules.

3.4 Program Design

3.4.1 Introduction to the method

ProMod's third and last phase is called 'Program Design'. The task of program design is to create the specifications which outline each of the program functions in pseudocode. That is, the functions are described in the user's natural language plus a small number of formal constructs. With these constructs, the program specification can be very specific but not yet tailored to a specific programming language.

Development of the higher-level programming languages that support structured programming created, as a byproduct, a technique of program design initially called 'stepwise refinement'. IBM and others call the same technique 'pseudocode', a term which has become widely adopted. The method of pseudocode has two principal purposes:

- Abstraction of implementation problems in a programming language.
- Application of principles of structured programming throughout the design project.

The use of the user's natural language simplifies communication between the analyst and other workers, and with the user. A forgotten semicolon or incorrectly spelled data name, or similar syntactical error, will not result in wasted time and effort. The few formal constructs, taken from conventions of structured programming, clarify some representations.

3.4.2 Pseudocode Design

Pseudocode allows the designer or programmer to implement the system design created during modular design using a combination of English phrases and keywords that represent the logic of the programming language in which the code will ultimately be written. The system design is abstracted using the worker's natural language to formulate programming solutions. The natural language of pseudocode permits creative freedom as well as easy communication with others.

However, to impose some form upon the natural language a few formal constructs must be used to insure a clear representation of programming logic, especially the decisions and loops. These few constructs are the keywords that are combined with the natural English phrases: IF, THEN, ELSE, ELSEIF, ENDIF, DO, ENDDO, CASE, BEGIN, END, ENDCASE, OTHERWISE, IN, OUT, PURPOSE and ENDPURPOSE. Function calls are indicated by preceding the function name with the dollar sign '$'. A typical function specification in pseudocode can be seen in figure 3.7.

It starts with the function name ('example') and the declaration of the input variables and the output variables. These are the so-called 'formal parameters'. After the header, the purpose of the function can be explained between the keywords

```
FUNCTION example ( IN inputvariable : TYPE-1 ;
                   OUT outputvariable : TYPE-2 );

PURPOSE
This is were the purpose of the function is explained
ENDPURPOSE

DATA localvariable : TYPE-3 ;

$getlocalvariable ( localvariable );

IF &inputvariable = &localvariable THEN
    &outputvariable = &localvariable
ENDIF
```

Figure 3.7: Pseudocode example

PURPOSE and ENDPURPOSE. In the next section, the local variables can be declared. Note that all variables are declared with a type that must be present in the data type dictionary. This is the same dictionary as in the system design phase and can be manipulated in this phase as well. After the local data has been declared, the pseudocode statements may begin. In this example the function 'getlocalvariable' is called followed by an IF, THEN, ENDIF construct. All data names are preceded by the '&'-sign in the pseudocode text. Except when it is clear that data is meant (e.g in function calls or data declarations).

Note that the phases of system design and program design are highly iterative. Very often the system design has to be changed because of problems arising in the pseudocode phase. This can even go back to changing the requirements model.

Eventually, the pseudocode gives a very precise specification of what the system must do and how it must do it. Although the pseudocode functions can often be compared to actual program text, they cannot be run. So it cannot be tested if the algorithm actually works. After this phase a high-level programming language can be selected and implementation can begin.

3.4.3 ProMod tools for program design

ProMod provides an integrated set of tools to visualise and check the work of the designer. The text editor, the charts designer and the PC (PseudoCode) analyzer are the three most important tools in this phase.

All function definitions and data type declarations are entered with the text editor. This is the same editor used in the first two phases. It is not very sophisticated and can be

replaced by an external editor. Text that is not correct cannot be saved into the project library but can be put in a separate file using the OUT option.

With the charts option, structure charts and function charts can be constructed. These charts show how the functions are related and are derived from the textual specifications of the functions. The difference between function charts and structure charts is that structure charts also show the variables that are passed between functions. These charts can be edited but are not suitable for actually developing a function structure because changes are not incorporated in the project library. The charts are more a way of visualising an intermediate or final design.

The PC analyzer performs global checks on the module interface, the module functions, and the declaration and use of data. Scope of the analysis and extent of the produced documentation can be selected by the user.

4. DEVELOPING THE CONTROL

4.1 Introduction

A control system must be developed to control the hardware of the exchange as described in chapter 2. The main functions that the control must perform are the following:

- Handle requests from the 'outside world' (other exchanges or a supervisor for this exchange). These can be requests to make a connection or break down a connection.
- Try to localise erroneous parts in the exchange and report these to the supervisor.
- If a decoder indicates that a certain connection is faulty and cannot be decoded correctly, try to find another path through the exchange.
- Allow the supervisor access to the exchange. (To make connections, influence the operation of the exchange, or change data relating to the subscribers)
- Performing measurements with regard to the traffic.

To design the software for this system, ProMod's three phases have been used (see chapter 3). The first part of the development is described extensively in [Verhoof, 1990]. This report includes feasibility studies and a description of the development of the requirements model. Because this model has been slightly altered, all three phases in the design are described in the next sections. In these sections flow names are referenced by preceding them with the '&'-sign just as in ProMod text.

4.2 Requirements Analysis and Definition

4.2.1 The context

Appendix 1 shows the DFD and the CFD for the context. In these diagrams there is one process called 'control_exchange' which is the control system. The control communicates with the terminators of which there are four. Note that not much attention has been given yet to the interface between the control and the terminators. [Verhoof, 1990] discusses parts of the interfaces but some of these are no longer correct due to changes in the hardware.

The terminator 'TST_switch' represents the actual switching network. This terminator gives the 'control_exchange' process access to the &connection_status, containing information about how the switches are set. This data is regarded as being always

present, though in reality a read command must be performed before the data can be offered. By sending a &new_connection to the TST_switch terminator a connection can be set up. Writing is always performed to all four fault isolation areas at the same time and involves writing in the connection memories of the T1, S and T2 switches.

The terminator 'decoder' is also part of the exchange and represents the (4,2)-decoders. By sending &dec_address the control can address a certain decoder and a certain error/modus register in that decoder. This results in the decoder sending the &status_word, which is the error/modus vector that was addressed. When two non-correctable errors occur in a connection, the decoder generates an interrupt and identifies the connection by sending &connection_to_be_changed to the control. The control system can then take appropriate steps (see 4.2.2). When a connection is set up, the control writes &new_dec_status to the decoder. This can be seen as writing in the error/modus register to set the right mode for that connection. This data flow includes the address of the decoder and the register.

The terminator 'concentrator' can give a request to 'control_exchange' to set up or break down a connection. The name concentrator is actually no longer correct for this terminator. The request to set up or break down a connection can come from a local exchange (with concentrator/deconcentrator, therefore the name concentrator) but also from other remote exchanges. To set up a connection, this terminator sends a &double_numbered_connection to the control. This data flow contains the ingoing and outgoing slots for the caller (so these have to be determined by the local or remote exchange) as well as the subscriber number for the receiver. So a connection from caller to receiver and from receiver to caller is set up. The control tries to find a free path through the exchange. If such a path cannot be found, the control signal &concentrator_connection_not_possible is sent to the terminator. Although the exchange is non-blocking it can be possible that a free path is not available because slots in the exchange may be broken or there may be no free slots on the outgoing line.

To break down a connection the 'concentrator' sends &input_address_of_connection as well as &output_address_of_connection to the control. These flows contain the number of the ingoing and outgoing slot for the caller. The control makes sure that the slots used in these connection are marked as free again. So the connection is broken down only when the caller hangs up. Later when the communication between this exchange and other exchanges has been given more attention this may be changed so that the receiver can also break down a connection.

The terminator 'supervisor' is a man-operated machine, which includes a system for man-machine communication. In a more advanced version of the software this terminator may be automated but for the prototype exchange it is the man who sets up and breaks down connections to check the exchange. When testing the prototype exchange it will probably not yet be connected to remote exchanges so the supervisor is the only one who makes the connections. The supervisor can request a connection by sending &addressed_connection. This flow contains an ingoing and outgoing slot and therefore constitutes a one way connection. If this connection cannot be set up (e.g. one of the slots to be connected may be occupied), the control sends

&supervisor_connection_not_possible. To break down a connection the supervisor sends the &output_address_of_connection which is the outgoing slot for that connection.

The supervisor can also change various parameters used by the exchange by sending &new_status_data. With the control flow &status_command the supervisor indicates which data is to be changed. With this command it can also indicate that a portion of the data is to be shown. The control replies by sending this data in the flow &status_info. With &check_method the supervisor can influence the error locating strategy. (Extensive checking may be undesirable during busy hours)

Compared to the model discussed in [Verhoof, 1990], this model has a Decoder terminator instead of a Codec terminator. The first design of the exchange contained four codecs before every decoder. These codecs could be seen as intermediate decoders that also had coded output. Due to impracticality the codecs were left out, leading to the exchange as we know it. All data names and processes associated with the codecs were replaced or left out in this model.

4.2.2 First level partitioning

Appendix 1 also shows the first level DFD and CFD. These diagrams indicate how the 'control_exchange' process in the context diagrams has been partitioned. So flows coming from and going nowhere correspond to flows entering and leaving the 'control_exchange' process in the context diagrams. In the next five sections each of the five first level processes is discussed.

In these first level diagrams there are also two stores. The &status contains all information about the switches. If they are occupied or free, in which areas they are not broken and the errors that occurred in connections using a particular slot. The store &parameters contains information needed for the error locating strategy. Compared to [Verhoof, 1990], the &parameters have been placed in a store because they should be constantly available and updateable by the supervisor.

4.2.2.1 Set_up_link

The process 'set_up_link' can set up a number of different connections. A &double_numbered_connection is a request from another exchange and contains the incoming and outgoing slot on the side of the caller as well as the subscriber number for the receiver. The control determines the outgoing line for the receiver and free slots on this line for the connections from and to the caller. If internal slots are found as well for these connections, they are set up. If one of the connections cannot be set up, the signal &concentrator_connection_not_possible is sent to the requesting exchange.

An &addressed_connection contains an incoming slot on an incoming line and an outgoing slot on an outgoing line. The control checks if they are free and tries to find internal slots for this one way connection. If these can be found the connection is set

up, else &supervisor_connection_not_possible is sent.

A &checked_addressed_connection is a request from another process ('change_link') and also contains the incoming and outgoing slots for the connection. But these do not have to be checked for availability. Internal slots are searched to connect the two slots and if they are found, the connection is set up and the signal &change_possible is sent to the requesting process. Finally, a routed connection can be requested. This connection has already been checked for availability and contains an entire connection (with internal slots). The only thing that has to be done is setting it up in the switches.

All connections are physically set up by writing &new_connection to the TST_switch and &new_dec_status to the decoder to set up the right mode for decoding.

4.2.2.2 Break_down_link

The process 'break_down_link' is used to break down a connection and keep track of errors that may have occurred. It either gets an &output_address_of_connection alone or in combination with an &input_address_of_connection. The control finds the connections for these slots and marks them as free. It also checks the contents of the error/modus registers for these connections by sending out &dec_address for every connection and thereby receiving &status_word.

Depending on the contents of &status_word and &check_method, the actual &connection_status is checked. The weight of any errors that are found is determined by checking &parameters (containing the weight for all errors). This weight is added to the total weight of errors for the erroneous slot. If no particular erroneous slot is found, the weight is added to all slots in the connection.

4.2.2.3 Change_link

The process 'change_link' changes the path of a connection when the decoders can no longer decode the data over that path correctly. When a decoder detects two non-correctable errors it sends an interrupt to the control. The process 'change_link' gets the &connection_to_be_changed and asks for a rerouted connection by sending &checked_addressed_connection to the process 'set_up_link'. This is a request for a connection with the same incoming and outgoing slot as the &connection_to_be_changed (these slots cannot be changed because they are used outside the exchange). The only thing that can be changed is the slot that is used inside the exchange, the internal slot. If the slots between the synchronisation and the TST-networks cause the problem (or perhaps a broken memory cell in the data memory of a T1-switch), the problem cannot be solved. The idea is that when a connection is so bad that it is no longer usable, the caller and receiver will hang up anyway.

If the &checked_addressed_connection can be set up, 'set_up_link' responds by sending &change_possible. In this case the internal slots of the old connection are

marked as free and possibly the &connection_status is checked (depending on the &check_method) to determine the source of the error.If the &checked_addressed_connection cannot be set up, it is left as it is. It may be advisable to try again later when other connections in the exchange have been broken down, but this has not yet been implemented. Again, when the connection is no longer usable, the receiver and caller will probably just hang up.

4.2.2.4 Access_status

This process is used to access the data structures used in the exchange, that is, the information contained in &parameters and &status. &Status contains information about errors (how many for every slot), statistics (although these haven't been defined very well yet), the switches (which connections are currently set up), the directory (which telephone numbers are connected to which lines) and set up times (so the length of a call can be determined). &Parameters contains information about how errors should be weighted, how long test links should last and when slots should be marked as defect.

The supervisor can access most of these data structures. He can ask for certain data by indicating which data in &status_command. The process 'access_status' looks for the requested data and sends it to the supervisor in &status_info. The supervisor can also change data by sending the changes in &new_status_data and indicating which data must be changed in &status_command.

4.2.2.5 Localise_errors

The process 'localise_errors' tries to determine regions (for every slot) in the exchange which are responsible for repeated errors, so that paths can be chosen through the TST-switch avoiding these bad regions and the decoders can reject data being passed through these regions. It is a process that is constantly active, but if and how it actually works is determined by &check_method.

Finding bad regions is restricted to finding combinations of lines and slots which are bad. For instance, it can be determined that the combination of line number 0 and intermediate slot number 20 is bad. Whether this error is caused by a T-switch or an S-switch or another problem is not certain.

When a connection is broken down, it is checked for errors and the weight of the error is recorded for one or more slots in that connection (see 4.2.2.2 Break_down_link). This leads to a record for every slot (in every fault isolation area) that holds the total weight of errors for that slot. The process 'localise_errors' constantly searches for slots that have a total weight of errors above two limits (which are held in the store &parameters). If the total weight of errors for a slot is higher than the highest limit, it is marked as broken. The &defect is reported to the supervisor and the slot (in a certain area) will no longer be used in connections until the supervisor has fixed the problem (if possible).

If the total weight of errors is below the highest but above the lowest limit, a test connection is set up through this slot to find out if the slot is really broken. The other slots for this connection are determined and a &routed_connection is sent to 'set_up_link'. After a certain time (indicated in &parameters) the connection is broken down again by sending &output_address_of_connection to the process 'break_down_link'. And that process checks if errors occurred in the connection.

The idea is that bad slots will eventually be found because their total weight of errors will grow faster than for slots that are not broken. However, eventually the total weight of errors for all slots will rise above the highest limit, because slots that were part of a bad connection may also receive a certain 'weight of error' whereas they may actually not be broken. To prevent this, the total weight of errors must be reset regularly for all slots. Furthermore, when a slot is marked as broken, there is no guarantee that it is indeed broken. Its total weight of errors may have just risen above the highest limit because it was often part of a connections in which errors occurred.

By means of &check_method, the supervisor can choose if the entire error localisation strategy must be performed, only parts of it, or not at all. This option is included to be able to make processing time available for call processing if required.

4.2.3 Further decomposition

All five processes are decomposed into lower level processes in two more levels. This decomposition is in most cases very straight-forward and need not be discussed. For more information about lower level partitioning and the development of the requirements model in general, the reader is referred to the ProMod SA report and [Verhoof, 1990].

4.3 System Design

In this second phase, the requirements model is transformed to a system specification. A hierarchical function structure is developed using the techniques of modular design (see 3.3). The first transformation is performed automatically by ProMod. In this transformation, the processes are transformed into functions and the flow diagrams are transformed into modules exporting the functions that were in the FD. This leads to a first basic structure that needs a lot of rearranging because the outset of the requirements model was not to combine functional equivalent processes. In the requirements model it is even desirable to represent the same process by more than one circle if it appears in different parts of the model. In the system design phase the object is to combine related functions in modules.

Appendix 2 gives an overview of the module hierarchy that was suggested by ProMod after automatic transformation. In this picture every module corresponds to a FD or a store in the requirements model. The basic data structures (stores in the requirements model) can be found in the bottom layer of the model, although the top layer contains

a module called 'status' which is also a store in the requirements model. However, this module can be removed because it is not used. This is because the status is never referenced in its entirety. The store 'status' is divided into a number of other stores in the requirements model, each having their own module in this phase. So the module for the store 'status' as a whole can be removed. Furthermore, layer 3 contains more modules that are not in this picture. This layer also contains the modules 'do_double_routing' and 'make_changes' that do not fit in the picture. References to these modules are represented by arcs going nowhere.

In the next sections the layers ,the modules in these layers, and the functions they provide are discussed.

4.3.1 The bottom layer; layer 4

Appendix 3 contains the latest module hierarchy. Compared to appendix 2, the bottom layer has got two more modules. The module 'interface' contains all functions that are used to control the interface with the TST-switches and the decoders. This includes reading from and writing to the TST-switches and the decoders. The module 'parameters' is new because the parameters were not in a store when the transformation was made and therefore no module was created for the parameters in the first version. The module 'parameters' exports the functions that are needed to access these parameters.

The module 'directory' contains the data structure that holds the information about the lines that subscribers are connected to. The module 'set_up_times' contains information about the set up time of a call and the appropriate access functions. This information can be used to perform traffic measurements. The modules 'error_statistics' and 'timing_statistics' provide functions to access statistical information about the occurrence of errors and the length of calls.

The module 'switch_status' contains the largest and most complicated data structure in the bottom layer. It contains all data about the switches and slots. The module provides nine functions to access this data structure in various ways. 'Errors' holds the information about the errors that occurred for each slot in every fault isolation area, and functions to access this information.

4.3.2 Layer 3

Layer 3 in appendix 3 (final structure) is quite different from layer 3 in appendix 2 (ProMod's structure). This is because the processes in the requirements model at this level are often quite similar in their function. During the requirements analysis phase processes are created with exact specifications of their function. If there is no other process with that function a new process is created with another name although it may be quite similar to an already existing process. This leads to a lot of similar functions in the system design phase (because processes are transformed into functions) that are spread all over the modules in layer 3. It is not very efficient to

implement the structure like this and therefore layer 3 needs a lot of rearranging.

The idea is to put related functions together in a module. This way they can call each other (because very often one function is part of another) whereas their names can stay the same as in the requirements model. When they are not grouped together in a module they cannot call each other because they are in modules that are at the same level. Another advantage of putting related functions into the same module is that a more comprehensive function structure can be created. The parts in the functions that are the same can be made into separate functions, declared as internal for the module. This way, double coding of certain parts can be avoided, but the 'world' outside the module still sees only the functions whose names correspond to the names of the processes in the requirements model. Using all these principles leads to a structure of layer 3 as in appendix 3.

The module 'make_changes' contains all functions necessary to change the data structures ,connections in the switches and the contents of the error/modus registers (of the decoders) when a connection is set up or broken down. The module 'find_connection' can be used to access functions that try to find a connection when it must be changed or broken down.

The module 'do_routing' provides all functions needed to find the slots for a connection that has to be set up. It is a good example of combining similar functions in one module. In the requirements model every type of connection that can be set up (see 4.2.2.1) has its own process. Needless to say that these processes differ, but parts of them are the same because the object is still to find free slots. So the functions derived from these processes can be combined in a module so that they can use each other, and similar parts can be separated and coded as internal functions.

The module 'make_test_link' contains all functions needed to set up a test link through a particular slot. This also concerns finding free slots, and therefore these functions may be placed in the module 'do_routing'. However, this is not wise because one look at the pseudocode for these functions confirms that they are substantially different (A good example of how the program design phase can lead to changes in the module structure). When a test connection is set up the object is to test a certain slot. This means that the other slots for the test connection have to be as 'good' as possible. That is, they should have a low total weight of errors, so that an error in the test connection is most likely due to the slot that was tested. In a normal connection this is not a requirement, all free slots can be chosen as long as they were not marked as broken. Otherwise the total weight of errors for all slots would be nivellated because slots with high weights would be used less. And furthermore, there is no reason to exclude a slot in a normal connection when it has not yet been marked as broken. This leads to substantial differences in the code structures and therefore these functions cannot be combined.

The module 'analyse_switches' provides the functions that analyse the errors that occurred in a connection when it must be changed or broken down.

4.3.3 Layer 2

The last section described the management layer which contained basic functions needed to solve specific parts of the control problem. This layer, layer 2, contains the modules that provide functions to perform the five larger tasks as described in section 4.2.2 and its subsections. Major changes in the module structure (compared to appendix 2) are not needed in this layer because there are just five modules corresponding to each of the five tasks. Only the import interfaces in these modules have to be changed because the layers 3 and 4 are different from the ProMod version in appendix 2.

The module 'set_up_link' provides the functions to find a connection and set up these connections in the data structures, the switches, and the decoders. The FD 'set_up_link' also contains the process 'find_intermediate_slot' which is not a function in the module 'set_up_link'. That is because the function 'find_intermediate_slot' is functionally more similar to the functions in the module 'do_routing' and also belongs more in layer 3. Therefore this function resides in the module 'do_routing'. So this is an example of a function being transported to another layer.

The module 'change_link' contains only one function, 'change_path'. The other function 'check_switches_of_interrupted_connection', which is in the FD 'change_link', can be found in the layer 3 module 'analyse_switches' which enables it to call the internal functions of that module. Module 'break_down_link' contains two functions. One to actually break down a connection in the data structure and the decoder (the connection in the switching network stays as it is, but the slots are marked as free). The other function is for checking the switches for errors.

The module 'localise_errors' provides two functions. One to find broken slots and one to set up a test link through a slot. The module 'access_status' provides one function to show the requested part of the data structure to the supervisor and one to make changes in the data.

4.3.4 The top layer; layer 1

This layer is used to control the functions provided by the lower layers. It only contains one module called 'control_exchange'. This module provides the function also called 'control_exchange' and its internal functions are the five major tasks (set_up_link, change_link, break_down_link, localise_errors, access_status). The idea is that the function 'control_exchange' gets all the requests from the outside world and distributes them over the five internal functions. These use the functions in layer 2 (and sometimes layer 3 due to rearranging of functions) that are contained in the module with the same name as the internal function.

4.4 Program Design

During this phase the functions are specified in pseudocode. Basically every process

in the requirements model becomes a function with the input parameters being the incoming flows and the output parameters being the outgoing flows. The function is described in plain English in the purpose part of the function. The pseudocode part is derived from the Mini Specifications for the process in the requirements model. Internal functions for which there are no MSPECs can be developed when needed. Although they have to be declared in a module during the system design phase.

Not much can be said about the actual writing of the pseudocode besides that it is very time consuming. It is just a rather tedious process of trying to specify the functions as precisely as possible, to speed up final implementation. It can be compared to actually writing the program, allowing informal specifications for parts that are not clear yet or parts which are language dependent. Although it looks like a program, the pseudocode cannot be run which is a major disadvantage. But more about this later.

A problem when writing pseudocode is the fact that is does not support parallel programming. This is rather surprising because in the requirements model all processes can run at the same time when the incoming flows are available. In pseudocode, each function calls the functions that were his subprocesses in the requirements model. So a problem arises in a situation as in figure 4.1. This figure shows the DFD for a process called F. This process has got an incoming and an outgoing flow named A and B. F is decomposed into the two processes F1 and F2 that communicate through the flows C and D.

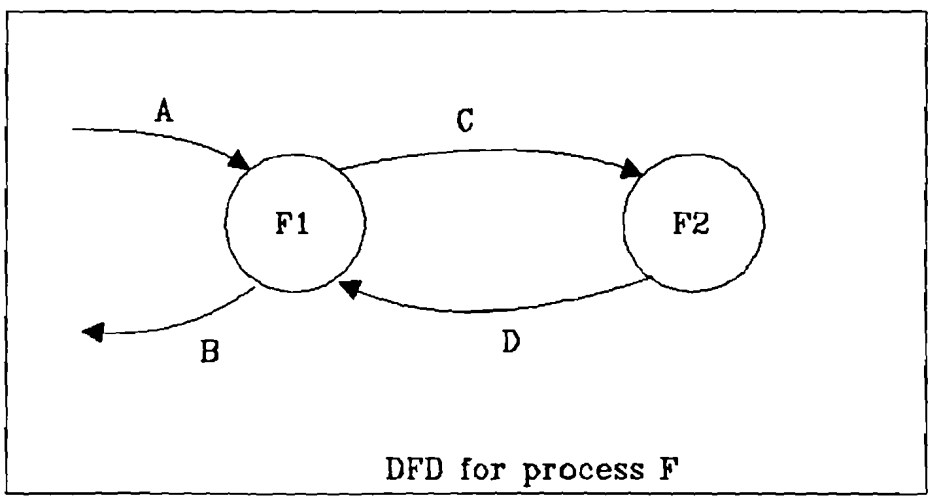


Figure 4.1: Problem with parallel processes

During the system and program design phases the processes are transformed to functions. So there is a function named F with input parameter A and output parameter B. In the pseudocode for this function, the functions F1 and F2 are called. The problem is that they have to be called sequentially, that is first F1 followed by F2 or first F2 followed by F1. However, neither is possible because F1 has to be called first to produce parameter C which is an input parameter for function F2. But F1 also has to end last to produce parameter B, which it can only do if it knows parameter D (so if F2 has ended). So F1 has to be called first but it cannot finish before F2 is called (to produce parameter D).

The solution seems very obvious, call F2 inside function F1. However, this may not always be possible. Because F1 and F2 are at the same level in the requirements model, they are most likely also at the same level in the system design (in modules that are at the same level). If they are in different modules, F1 cannot call F2 without violating the conventions of modular design (modules at the same level cannot use each other's functions). It also means that the functions are no longer structured as in the requirements model (that is, process F2 actually becomes a subprocess of process F1). In fact, F becomes the same as F1. Furthermore, another function G, at the same level as F may want to use function F1 without calling F2. If F2 is contained in F1, this is not possible. Also, the situation is highly simplified with F being decomposed into only two processes. When multiple processes are involved in this kind of communication, the problem becomes even more complicated.

Another solution is to split function (or process) F1 into two functions F1a and F1b as in figure 4.2. This way F1a can be called first with input parameter A and output

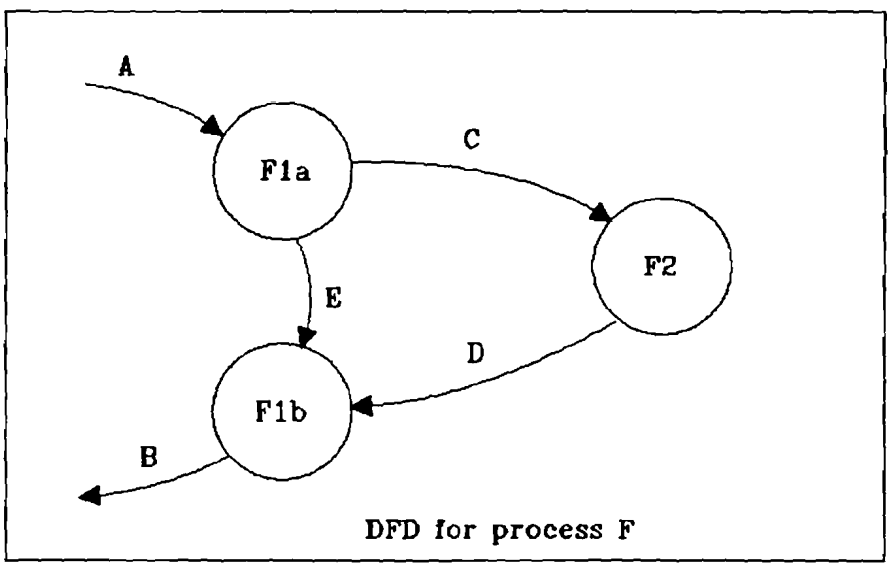


Figure 4.2: Splitting a process

parameters C and possibly E. After this, F2 can be called producing parameter D. Finally F1b can be called with input parameters D and E producing output parameter B. The splitting of F1 may in fact be very easy because it is usually already decomposed in a number of processes. But this method requires restructuring the requirements model which may not always be desirable. The splitting can also be done in the function structure only and not carried through in the requirements model. But again, this has the disadvantage that the functions are not structured as in the requirements model.

The third solution is to use concurrent programming. In this style of programming two processes can actually be active at the same time. So in case of the situation as in figure 4.1, F1 can be called and during its execution it sends the parameter C to the function F2 that is running concurrently. F1 can continue until it needs parameter D. It waits for F2 to produce this parameter (maybe it already has), produces B and returns control to the calling function. The problem is that C and D are no longer real function input and output parameters. They are passed between concurrent processes as they are running. Conventional input parameters need to be available at the start of a function and conventional output parameters are 'released' when a function ends.

ProMod's conventions during program design do not provide these kind of parameters formally. It is left up to the designer to describe the parameter passing in the informal part of the pseudocode. ProMod's formal pseudocode presumes a sequential program flow with parameters being passed only between sequentially called functions. Of course, program flow can be made sequential using the methods described above, but doing this for the entire design would obscure the functional blocks and depart heavily from the structure in the requirements model.

Therefore, the pseudocode for the control of the exchange uses all three methods described above. At the top level, concurrent programming is used because there are five (clearly different) functions that communicate and can be run more or less at the same time. Parameter passing between parallel processes is indicated in the pseudocode by issuing a parameter to a parallel process. In the lower levels everything is kept sequential as much as possible to keep things from getting too complicated during implementation. Because concurrent programming also means introduction of mutual exclusion for data that the concurrent processes share. For example, when slots are searched for a connection, these slots are marked as occupied only when an entire routing through the exchange has been found. So one process may choose a certain slot whereas it already may have been chosen by another process that just didn't mark it as occupied yet. Therefore the process of finding slots through the exchange and marking them as occupied has to be mutual exclusive.

It can be argued that the whole process of developing pseudocode is not very useful. In many cases the writing of pseudocode is nearly as detailed as the eventual program implementation, but it cannot be run. So why bother writing a program in pseudocode if it can just as well be written in the programming language directly. Indeed for sequential programs that are not too complicated developing pseudocode may very well prove to be superfluous. For this control system however, it was very useful for developing functions and discovering similarities in functions. The pseudocode is

definitely not the same as the eventual implemented program because concurrent programming is needed which makes the implementation much more complex than the pseudocode, because extra statements are needed to create parallel tasks and communication facilities between them. Also, mutual exclusion has to be implemented using the facilities of a multi-tasking operating system.

So in this case pseudocode was a useful intermediate step in the development of the software. Advertisers of pseudocode also claim that pseudocode can be used for program maintenance. This seems more doubtful. Every change that is made to the program has to be incorporated in the pseudocode (even in the system design and sometimes in the requirements model) leading to a fair amount of work for even small changes. It is doubtful that anyone else besides the designer of the pseudocode (who knows it well enough) will actually keep the pseudocode up to date. Therefore, the pseudocode seems a less likely candidate for maintenance purposes however desirable it may be to keep it up to date. However, the requirements model and system design can be used for maintenance if they are kept up to date. They are very suitable for getting a general idea of the structure of the program.

5. IMPLEMENTATION WITH CRMX

5.1 Introduction

Any computer that is controlling or monitoring another machine can be or should be a real-time system. Real-time computer systems are combinations of hardware and software that integrate computers with external processes. The control for an exchange is a typical example of such a real-time system. It must respond to external requests that occur unpredictably. Critical events that require immediate attention must be guaranteed a worst-case response time. This can be done by performing a task switch in which the execution of the normal program is stopped and a special interrupt task is started to handle the request.

Chapter 4 shows that the control software is structured so that there are several independent tasks that service a particular request or perform particular functions. These tasks require only spurious communication with other tasks. Otherwise they are completely independent.

These considerations lead to the conclusion that a multi-tasking operating system is needed to implement the control. iRMX is such an operating system and was used as the operating system for the control. The actual program was written in C.

5.2 The iRMX II.4 Operating System

5.2.1 Concept and overview

Intel's operating system iRMX combines the concepts of real-time response with object oriented programming. Object oriented programming focuses on data structures and the operations performed on them; top down programming focuses on control flow. These programming concepts tend to be complimentary rather than mutually exclusive. Both types of programming are used to break large difficult problems into smaller simpler problems.

An object oriented solution breaks the large complex problem into smaller more manageable tasks or objects. Each task can then be designed with minimum dependence on other tasks in the system. Taken all together, the effort of the many smaller tasks provides the solution for the original complex problem.

As a real-time system, iRMX can respond to external events called interrupts and immediately set tasks in motion to process the interrupts. Because interrupts can occur simultaneously and at random times, the operating system supports multi-tasking

operations to handle the multiple events concurrently. In addition, the operating system allows a multi-programming environment in which several unrelated applications can run independently.

The iRMX operating system is a collection of subsystems or layers, each of which provides one or more features that can be used in applications. The subsystems are called layers because each one builds on the capabilities of the previous ones. Figure 5.1 shows these layers. These layers can be included or excluded to form a tailored operating system.

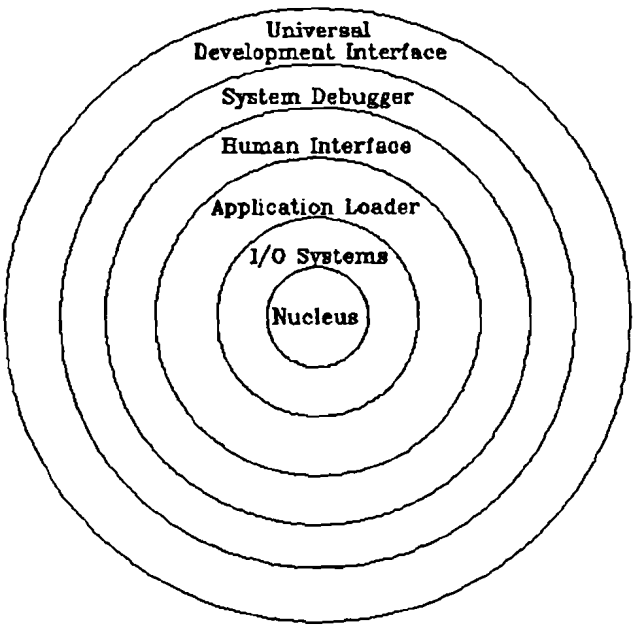


Figure 5.1: Layers of the operating system

The nucleus is the heart of the operating system and the only required layer. All other layers of the operating system are optional. Its activities include:

- Supplying scheduling functions
- Controlling access to system resources
- Providing for communication between processes and processors
- Enabling the system to respond to external events

The nucleus provides the building blocks from which the other subsystems and application systems are constructed. These building blocks are called objects. They are classified into the following categories:

- Tasks
- Jobs

- Segments
- Mailboxes
- Semaphores
- Regions
- Extension objects
- Composite objects

These objects can be created, manipulated, and deleted with the system calls provided by the nucleus.

The I/O systems (basic and extended) provide file management and the device-independent interface to input and output devices. The I/O systems are optional components of the iRMX operating system so they can be excluded if they are not needed. The extended I/O system is built on the basic I/O system, so when the extended system is included, the basic system must also be included. The difference between these I/O systems is that the basic I/O system emphasizes flexibility rather than ease of use. The system calls of the basic I/O system involve many parameters. Using these parameters, tasks can tailor the behaviour of each system call to match the specific requirements of each application. The system calls of the extended I/O system require fewer parameters, simplifying their use, and reducing the complexity of an application.

The application loader enables applications to load programs and overlays from disk into main memory under the control of iRMX tasks that are part of application programs. This allows programs to run in systems with insufficient memory to accommodate all programs at one time. It is also possible for programs that are seldom used to reside on secondary storage rather than in main memory. Large programs can be divided into overlays that are loaded from disk by the application loader whenever they are needed. The application loader is optional and can be excluded.

The human interface may control the application system with commands entered at a terminal. It allows the console operator to execute commands on two levels, the standard command line interpreter level, and the human interface level. So after booting the iRMX operating system without special configurations (the human interface layer must be included), human interface commands can be used to handle files or start user programs etc.. But human interface commands can also be executed by using system calls from this layer in user defined programs or tasks. The command line interpreter can be seen as the interface between human interface commands typed in at the terminal and human interface system calls. The human interface is an optional component and can be excluded when applications are bootloadable.

The system debugger is an extension of the standard system debug monitor. The monitor provides disassembly, execution break points, memory display, and program download capabilities. The system debugger extends the monitor's disassembly functions by interpreting iRMX calls, data structures, and stacks. It is a configurable layer that can be excluded when not needed.

The universal development interface is a set of system calls compatible with several of Intel's operating systems. If an application program makes only calls to this layer and no explicit calls to an individual Intel operating system, the application can be transported between operating systems. It is a software interface that allows language translators and other software development tools to access the facilities of the iRMX operating system. This layer is the outermost layer of any application system but may be excluded if it is not needed.

These six layers from the iRMX operating system provide a broad number of functions. The major system functions are:

- Monitor and control unrelated events occurring outside the single board computer.
- Communicate with a wide variety of input, output, and mass storage devices.
- Provide a base on which to run a number of languages and other software tools.

In the next two sections the features are described that are important for the control software.

5.2.2 Task management

Tasks are the active objects in an iRMX II system. A task can be seen as a program that handles part of the overall problem. Because iRMX is a multi-tasking operating system, it allows more than one task to be active at the same time, although only one task can have actual control of the processor at any one time. Another task can obtain control of the processor by a task switch. The context of the previously running task is saved by the nucleus and the context of the new running task is loaded. After this, the new task begins executing at the point where it was stopped the last time it had access to the processor. This allows several tasks to be active at the same time. Each task can actually be considered a virtual CPU. A task has two basic goals:

- Its primary goal is to do a specific piece of work.
- Its secondary goal is to obtain control of the processor so that it can progress towards its primary goal.

One of the main activities of the nucleus is to arbitrate when several tasks each want control over the processor. The nucleus does this by maintaining an execution state and a priority for each task. The execution state for each task is, at any give time, either running, ready, asleep, suspended, or asleep-suspended. The priority for each task is an integer value between 0 and 255 inclusive, with 0 being the highest priority. The arbitration algorithm that the nucleus uses is that the running task is the ready task with the highest (numerically lowest) priority. This is called preemptive, priority-based scheduling. When a task with a higher priority than that of the current running task becomes ready, the low priority task is preempted and the high priority task begins execution immediately.

To ensure that equal priority tasks all get a chance to run, the iRMX operating system features round-robin scheduling as well. With round-robin scheduling, each task is allocated a time quota. When the time quota expires, the task is preempted and the next task of the same priority is allowed to run. This technique allows equal priority tasks to take turns running. Without it, a task can gain control and continue running indefinitely until another higher priority task preempts it. This blocks out all other tasks of the same priority. Of course, higher priority tasks can still preempt any running task, regardless of the time left in its quota. The default value for the time quota is 50 ms.

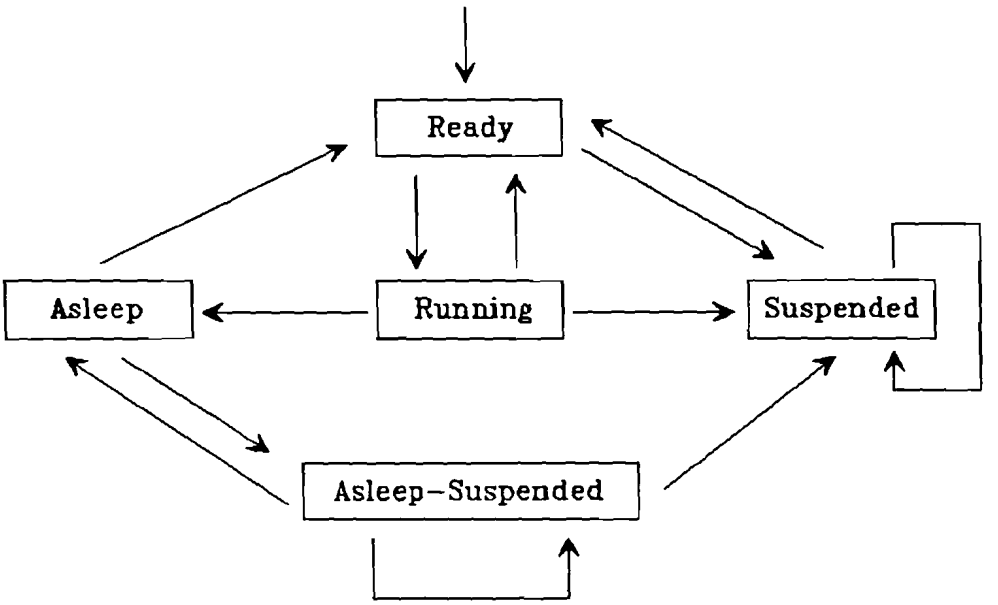


Figure 5.2: Task state transition diagram

Figure 5.2 shows the execution states of tasks and the transitions they can go through. When a task is created it starts in the ready state. Every task that is not asleep, suspended or asleep-suspended is in the ready state. For a task to become the running (executing) task, it must be the highest priority task in the ready state. A task can go from the running to the ready state if it is preempted by a higher priority task or if it is rescheduled due to round-robin. A task is in the asleep state when it is waiting for a request to be granted (e.g. the task is waiting at a mailbox for a message) or when it puts itself to sleep by issuing the SLEEP system call. The length of time that the task stays in the asleep state can be specified in both cases. A task goes from the asleep to the ready state if the request it was waiting for was granted or after a specified period of time.

A task enters the suspended state if is suspended by another task, waiting for an interrupt, or suspends itself. The task can go to the ready state again by a RESUME system call issued by another task. When a sleeping task is suspended, it enters the asleep-suspended state. In effect, it is then in both the asleep and suspended states.

While asleep-suspended, the task's sleeping time might expire putting it in the suspended state. Also, if another task resumes an asleep-suspended task, the latter task will enter the asleep state.

Resources that a task may need (e.g memory) are taken from the task's containing job. If a task is subsequently deleted, these resources are returned to the job.

5.2.3 Inter-task coordination and communication

Multi-tasking is a technique used to simplify the designing of real-time application systems that monitor multiple, concurrent, asynchronous events. It enables engineers to focus their attention on the processing of a single event rather than having to contend with numerous other events occurring in an unpredictable order. However, the processing of several events may be related. This kind of processing requires that tasks be able to coordinate with each other. Basically, tasks interact with each other in three different ways: exchanging information, mutually excluding each other, and synchronising. For these three kinds of interactions, iRMX provides three kinds of objects, the exchanges: mailboxes, semaphores, and regions.

Mailboxes are used for exchanging information between tasks. When a task wants to exchange information (send or receive) with another task, it cannot do so directly because tasks are asynchronous. They run in an unpredictable order. If tasks want to communicate, they need a place to store messages and to wait for messages. This is the mailbox. The iRMX operating system provides two kinds of mailboxes. One to store data (of up to 128 bytes per message), and one to store tokens. A token is a 16-bit value associated with an iRMX object. So the second type of mailboxes can actually be used to pass objects. Messages can be send to these mailboxes by the SEND MESSAGE/DATA calls. The message is held at the mailbox until it is collected by a task that calls RECEIVE MESSAGE/DATA. If no message is present in the mailbox the receiving task can wait at the mailbox in the asleep state.

Semaphores can be used for synchronisation and mutual exclusion. A semaphore is a counter which can take on only non-negative integer values. Tasks can modify a semaphore's value by using the SEND UNITS or RECEIVE UNITS system calls. When a task sends n units (n >= 0) to a semaphore, its value is increased by n. When a task uses the RECEIVE UNITS system call to request x units (x >= 0) from a semaphore, one of the following two things can happen:

- If the semaphore counter is greater than or equal to x, the operating system reduces the counter by x and continues to execute the task.
- Otherwise, assuming the task was programmed to wait, the operating system begins running the task in the ready state having the next highest priority. The requesting task waits at the semaphore (in the asleep state) until the counter reaches x or greater.

Mutual exclusion is sometimes necessary when data is shared between a number of processes. It may be necessary to prevent other tasks from accessing the data when

it is being used or updated by another task. Mutual exclusion can also be needed when a resource can only be used by one task at a time. A semaphore can achieve mutual exclusion by giving it an initial value of one. Before any task uses the shared resource (data or otherwise) it must receive one unit from the semaphore. As soon as a task finishes using the resource, it must send one unit to the semaphore. This technique ensures that at any given moment, no more than one task can use the resource, and any other tasks that want to use it await their turn at the semaphore.

In addition, semaphores can also be used to synchronise tasks. One task can be instructed to wait at a semaphore (RECEIVE UNITS on a semaphore with value zero), until another task has reached some point in its execution (SEND UNITS).

The third exchange is the region which can also be used for mutual exclusion. This object can be used to guard a specific collection of shared data. Each task desiring access to shared data awaits its turn at the region associated with that data. When the task currently using the shared data no longer needs access, it notifies the operating system, which then allows the next task to access the shared data. However, regions should be restricted to specific uses. The priority of a task may be raised when it has access to the shared data to prevent so called priority bottlenecks. One of the effects of this is that an application started at the human interface level (like the control) cannot be stopped without rebooting (the usual CONTROL-C does not work). Therefore, designers are advised to avoid the use of regions in human interface applications. So regions have not been used in the control software.

5.3 Structure of the control using iRMX objects

The object oriented way of programming of iRMX matches the way a system is developed using the requirements model and ProMod (if parallel processes are allowed in the pseudocode). An object oriented solution breaks the problem into smaller more manageable tasks or objects that can be designed with minimal dependencies. This is very similar to dividing a problem into processes in the requirements model. The only intermediate step that has to be made is which processes are to be the iRMX concurrent tasks and which processes are to be made sequential (combining object oriented programming and top down programming). This decision was already made in the ProMod phases 'system design' and 'program design'.

Figure 5.3 shows the structure of tasks and mailboxes necessary for the control. In this figure circles are tasks and rectangles are the mailboxes these tasks use for communication. All mailboxes are data mailboxes that can store messages of up to 128 bytes. A directed arc towards a mailbox means that the task sends to that mailbox. A directed arc away from a mailbox means that the task receives messages (if available) from that mailbox.

All incoming requests as well as interrupts from the decoders must be handled by one or more interrupt task(s). These have not yet been implemented because they involve interfacing with other exchanges, which has not yet been examined. All external events

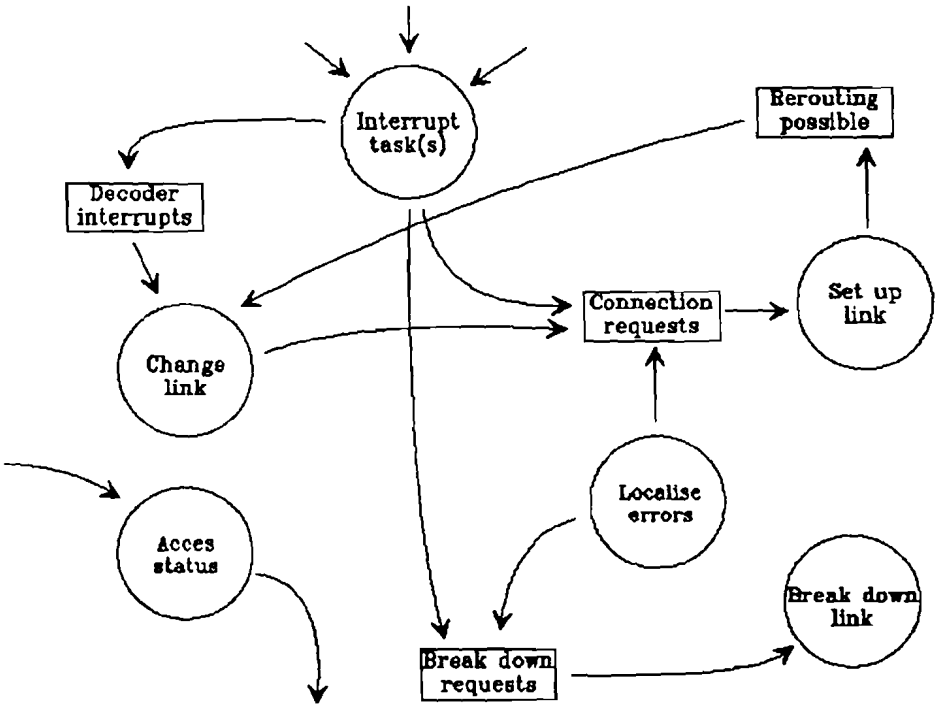


Figure 5.3: Task structure of the control

should be taken care of by interrupts. This directs control to an interrupt handler that starts an interrupt task. The interrupt task takes care of the request (or traces the decoder that caused the interrupt) and transforms it to a format that can be sent to the appropriate mailbox. The request is then taken care of by one of the other tasks (which have been implemented). A request for a connection is sent to the mailbox for connection requests. Break down requests are sent to the mailbox with that name. A decoder interrupt occurs when at least two non-correctable errors have been detected in a connection. The interrupt task finds out which decoder was responsible for the interrupt and for which outgoing slot it was intended (see [Cornelisse, 1990]). The address of that decoder (outgoing line and slot) is sent to the mailbox for decoder interrupts.

Connection requests are handled by the task 'set up link' and break down requests are handled by 'break down link'. The task 'change link' handles the decoder interrupts and asks for a rerouted connection by sending a connection request to the appropriate mailbox. The task 'set up link' indicates if the rerouting was possible by a message in the mailbox 'rerouting possible'.

The task 'localise errors' takes care of the error localisation strategy. It sets up test links by issuing a message to the mailbox 'connection requests' and after a specified period of time a message is sent the mailbox 'break down requests' to break down the test link. The task 'access status' is needed to allow the supervisor to influence the operation of the exchange in various ways and to give access to the data structures.

Also, this task contains some special features to test the software as long as there is no real exchange to control.

The control uses only one semaphore to guard the shared data structures. They need to be protected for two reasons. First, tasks that try to find a path through the exchange (like 'set up link' and 'localise errors') choose slots for a connection that are free. But they only mark them as occupied (preventing other connections from using them) when an entire connection has been found. Therefore it is imperative that the data is only used by one process at a time that is picking out slots for a connection. Because there is a delay between choosing a slot and marking it as occupied. Other tasks contain similar situations and require exclusive use of certain data. Furthermore, some functions supplied by the data structure modules corrupt the integrity of the data until they are finished. If another function tries to access the same data, it may produce results that are not valid or even cause a break down of the program. Hence the need for mutual exclusion which is provided by a binary semaphore.

Only one semaphore is used to protect all shared data. When a task starts to handle a request from one of the mailboxes it performs a RECEIVE UNITS operation on the semaphore and thereby gets exclusive rights to the shared data. Only when the request is finished or when communication with another task is necessary, the exclusive rights are released by the SEND UNITS system call. The reasons for using only one semaphore to protect all data (and thereby blocking nearly all processes that want to use this data as well) are twofold. The first reason is that it is the simplest solution to avoid deadlock. When a task claims the data, it is always released again because no other data has to be claimed anymore (which might block the task) and no RECEIVE DATA calls to mailboxes are made when the task is in possession of the shared data. So a task will not wait for a message tying up the data.

The second reason for using only one semaphore is that it reduces complexity. Every request is handled in one go without other tasks interfering. A task does not have to consider other tasks influencing its operation because they may be using the shared data as well. Nor does the programmer have to examine if exclusive rights to a particular piece of shared data is necessary, reducing possible programming errors. And in the end all tasks actually need nearly all shared data to do their job. Effectively it means that a message from one of the mailboxes is handled in its entirety or at least until communication with another task is necessary. Figure 5.4 shows the general structure for a task that does not need communication with other tasks when handling a request (e.g 'set up link', 'break down link' and 'access status'). Figure 5.5 shows the general structure of a task that does (e.g. 'change link' and 'localise errors').

In figure 5.4 the task first checks if there is a request in the mailbox that needs to be processed. If there is no such request the task waits at the mailbox in the asleep state. If there is a request, the task tries to get exclusive rights to the shared data by trying to get a unit from the semaphore. If the semaphore is one, the task starts processing the request. If the semaphore is zero, the task waits at the semaphore until it is increased by the task, now having access to the shared data. After the processing is completed, the semaphore is increased by one so that another task can use the data, and the mailbox is checked again for another request.

```
task()
do forever
{
  receive data (mailbox)
  receive unit (exclusive_semaphore)
  perform processing
  send unit (exclusive_semaphore)
}
```

Figure 5.4: Structure of a task that doesn't need communication

In figure 5.5 the task needs some processing by another task before it can finish a request. The task first gets the request from mailbox1 and the rights to the shared data. It processes the request as far as possible and sends a request to another process (by sending it to mailbox2). After that, this task releases the shared data to give the other task the opportunity to process the request in mailbox2. This task waits at mailbox3 until it gets the information from the other task necessary to finish this request. The shared data is claimed again and the request can be finished after which the shared data is released and the task waits at mailbox1 for the next request.

```
task()
do forever
{
  receive data (mailbox1)
  receive unit (exclusive_semaphore)
  perform first part of processing
  send data (mailbox2)
  send unit (exclusive_semaphore)

  receive data (mailbox3)
  receive unit (exclusive_semaphore)
  perform second part of processing
  send unit (exclusive_semaphore)
}
```

Figure 5.5: Structure of a task that communicates with other tasks

Only the task 'localise errors' deviates slightly from this structure. This task is not triggered by any request from a mailbox but runs continuously. It is constantly looking for bad or broken slots by reading from the data structure ERRORS. It does not have to claim the data first because it is not important if another process is changing it. No errors will occur when 'localise errors' acts upon a value in ERRORS even if it is just before or after another process changes that value. Furthermore, 'localise errors' only claims the shared data when it must find a test link through a bad slot or when it wants to mark a slot as broken.

5.4 Implementation of the tasks in C

The functions and tasks developed during the ProMod phases 'system design' and 'program design' were translated into C. For the most part this is very straight forward. However, most of the functions in pseudocode have multiple output parameters whereas C-functions only have multiple input parameters and one return (output) value. This was handled by supplying pointers to the output variables for a function in C. So when a function f() has to be called that has got two input parameters x1 and x2, and two output parameters y1 and y2, the function is called as:

```
f (x1, x2, &y1, &y2);
```

The function f() is then declared as having 4 input variables, x1 and x2, and two pointers that indicate where the results of the function must be placed. These output variables can be accessed inside f() by using the indirection operator '*'. The only time when this is not necessary, is when the results must be placed in an array. In that case the array can be given without the address operator '&' and can be referenced directly in the called function. This is because arrays are implemented in C as pointers to the first element in an array.

Type names used in the implementation are as much as possible identical to the ones used in the ProMod pseudocode and are declared in a separate file. Each ProMod module is also contained in a separate file.

On implementation, one more task was added that does not appear in figure 5.3. When the control is invoked, the function main() is started which is a task in its own right. It initialises the data and interface and creates all iRMX objects. The tasks are created with a priority one lower than this initialising task so that it can end the initialisation before the other tasks begin. After this task has finished, it suspends itself, thereby giving the other tasks opportunity to start processing requests.

The use of iRMX system calls in a C program requires that these calls and the required data structures are declared first. The iC-286 library contains a number of useful header files that can be used for this purpose. Also, the iRMX manuals recommend to use in-line exception processing. This means that possible errors due to system calls are handled by a user-supplied C-function and not by the system itself. The advantage is that an error in a system call can be traced because after each system call the function is called that checks for errors. This also requires the exception mode to be set at the beginning of each task so that the system does not trap any errors.

The C-compiler contains a demo program explaining much of the techniques for using iRMX system calls in C programs as well as some useful header files. My own experiences with the iRMX operating system can be found in appendix 4.

As already said, the functions that deal with hardware interfacing have not yet been implemented. This includes the interrupt task(s) as well as the functions that actually write to and read from the connection memories and the decoders. For now, because no real exchange is available yet, these functions write to and read from arrays that

symbolise the connection memories and decoder registers. Also, the modules that deal with timing and error statistics have not yet been implemented. This is due to the fact that it was not really defined what should be in them. But they are not vital to the operation of the control anyway, certainly not in its test stages. In the program it has been indicated where writing to these statistics should take place.

The actual program was compiled with the iC-286 R4.1 compiler for use with RMX II systems. Linking of the object files and iRMX interface libraries was performed with the iAPX Binder, Version 3.2. The sources for the program amount to a total of 340K. The object files from these sources amount to 69K and the executable file is around 160K. A description of how the program should be used and operated can be found in appendix 5.

6. CONCLUSIONS AND RECOMMENDATIONS

With the CASE-tool ProMod a model has been developed for the software of the control system. The model was built on the (slightly revised) requirements model described in [Verhoof, 1990]. All in all, the tool seems quite suited to develop complex systems in a very logical and 'one step at a time' manner, although some problems arise when parallel programming must be introduced. The end result of ProMod was a very specific program structure with the functions described in pseudocode.

Implementation of the model has been performed in C using the iRMX II.4 multi-tasking operating system. An operational version of the control is available and has been tested. Only the functions dealing with interfacing (such as writing to and reading from the connection memories as well as dealing with interrupts from the decoders or requests) still have to be written.

The next logical step in the development of the control seems to complete the software by writing functions that drive the interface between the control and the exchange, and to make a study of how the control is to communicate with other exchanges in a network. Finally, these communication protocols have to be implemented.

REFERENCES

Aggenbach, M.H.J.M.
ONTWERP VAN EEN (4,2)-DECODER/CODER VOOR EEN DIGITALE TELEFOONCENTRALE VOLGENS HET '(4,2)-CONCEPT'.
Eindhoven University of Technology, Department of Electrical Engineering, Digital Systems Group, June 1988.
Master thesis report. TUE-EB112

Axford, T.
CONCURRENT PROGRAMMING: Fundamental techniques for Real-Time and Parallel Software Design.
Chichester, John Wiley & Sons, 1989.

Ben-ari, M.
PRINCIPLES OF CONCURRENT PROGRAMMING.
Englewood Cliffs, Prentice-Hall, 1982.

Bink, S.
IMPLEMENTATIE EN SIMULATIE VAN DE CONTROL UNIT VAN EEN TST-SCHAKELNETWERK.
Eindhoven University of Technology, Department of Electrical Engineering, Digital Systems Group, June 1990.
Project report.

Boonen, P.E.J.
THE DEVELOPMENT OF BITBUS FIRMWARE AND THE ACCESSORY VME/ERM BITBUS DRIVER FOR THE dDCM804 BITBUS BOARD.
Eindhoven University of Technology, Department of Electrical Engineering, Digital Systems Group, August 1988.
Master thesis report.

Bustard, D. and J. Elder and J. Welsh
CONCURRENT PROGRAM STRUCTURES.
Englewood Cliffs, Prentice-Hall, 1988.

Clos, C.
A STUDY OF NON-BLOCKING SWITCHING NETWORKS.
Bell Systems Technical Journal, Vol.32. pp 406-424, March 1953.

Cornelisse, J.
ONTWERP EN IMPLEMENTATIE VAN EEN (4,2)-DECODER IN EEN DIGITALE TELEFONIECENTRALE.
Eindhoven University of Technology, Department of Electrical Engineering, Digital Systems Group, August 1990.
Master thesis report.

Duin, R.W.P. van
ONTWERP VAN EEN (4,2)-DECODER/CODER VOOR EEN DIGITALE TELEFOONCENTRALE TOT OP SEMI-POORT NIVEAU.
Eindhoven University of Technology, Department of Electrical Engineering, Digital Systems Group, August 1989.
Project report. TUE-EB228

Duijkers, L.J.M.H.H.
IMPLEMENTATIE VAN EEN MULTI-TASKING OPERATING SYSTEM (DE LEX).
Eindhoven University of Technology, Department of Electrical Engineering, Digital Systems Group, October 1986.
Master thesis report. TUE-EB023

Engelen, W. van
CONCENTRATOR EN DECONCENTRATOR VOOR EEN TELEFOONCENTRALE VOLGENS HET (4,2)-CONCEPT.
Eindhoven University of Technology, Department of Electrical Engineering, Digital Systems Group, August 1988.
Master thesis report. TUE-EB127

Gehani, N. and W.D. ROOME
THE CONCURRENT C PROGRAMMING LANGUAGE.
Summit, Silicon Press, 1989.

Harbison, S.P. and G.L. Steele Jr.
C: A REFERENCE MANUAL.
Englewoods Cliffs, Prentice-Hall, 1987.

Hatley, D.J. and I.A. Pirbhai
STRATEGIES FOR REAL-TIME SYSTEM SPECIFICATION.
New York, Dorset House, 1987.

Kelley, Al and Ira Pohl
AN INTRODUCTION TO PROGRAMMING IN C; A BOOK ON C.
Menlo Park, The Benjamin/Cummings Publishing Company Inc., 1984.

Kernighan, Brian W. and Dennis M. Ritchie
THE C PROGRAMMING LANGUAGE.
Englewood Cliffs, Prentice-Hall, 1978.

Kochan, Stephen G.
PROGRAMMING IN C.
Indianapolis, Hayden Books, 1983.

Krol, Th.
HET '(4,2)-CONCEPT' VOOR HET MAKEN VAN FOUTENTOLERERENDE COMPUTERS.
Philips Technisch Tijdschrift, Vol. 41, No. 1, pp 1-12, 1983.

Lier, A. van
DESIGN OF A TIME/SPACE SWITCH ELEMENT FOR A DIGITAL TELEPHONE EXCHANGE.
Eindhoven University of Technology, Department of Electrical Engineering,
Digital Systems Group, August 1989.
Master thesis report. TUE-EB202

Muijselaar, J.N.M.
INCREASING THE CAPACITY OF A FAULT TOLERANT DIGITAL TELEPHONE EXCHANGE.
Eindhoven University of Technology, Department of Electrical Engineering,
Digital Systems Group, August 1990.
Master thesis report.

Oudelaar, H.
IMPLEMENTATION OF A CCITT PROTOCOL ON AN ISDN TERMINAL BOARD.
Eindhoven University of Technology, Department of Electrical Engineering,
Digital Systems Group, December 1989.
Master thesis report.

Ronayne, J.P.
INTRODUCTION TO DIGITAL COMMUNICATIONS SWITCHING.
London, Pitman Publishing Ltd., 1986.

Timmer, A.H.
DESIGN OF THE CENTRAL SWITCH AND PCM SYNCHRONISATION OF AN EXCHANGE.
Eindhoven University of Technology, Department of Electrical Engineering,
Digital Systems Group, April 1990.
Master thesis report.

Verhoof, P.H.W.
REQUIREMENTS MODEL FOR THE CONTROL SYSTEM OF A DIGITAL TELEPHONE EXCHANGE.
Eindhoven University of Technology, Department of Electrical Engineering,
Digital Systems Group, February 1990.
Master thesis report. TUE-EB232

Ward, Paul T. and Stephen J. Mellor
STRUCTURED DEVELOPMENT FOR REAL-TIME SYSTEMS. (VOL. 1-3)
New York, Yourdon Press, 1985.

Yourdon, E. and L. Constantine
STRUCTURED DESIGN: Fundamentals of a discipline of computer program and systems design.
New York, Yourdon, 1978. (1st edition: 1975)
(Also, Englewoods Cliffs (NJ), Prentice Hall, 1979.)

Manuals:

AEDIT TEXT EDITOR USER'S GUIDE FOR iRMX 286 SYSTEMS.
Version 2.2,
Santa Clara, Intel Corporation, 1985.

iAPX 286 UTILITIES USER'S GUIDE FOR iRMX 286 SYSTEMS.
Version R3.2,
Santa Clara, Intel Corporation, 1985.

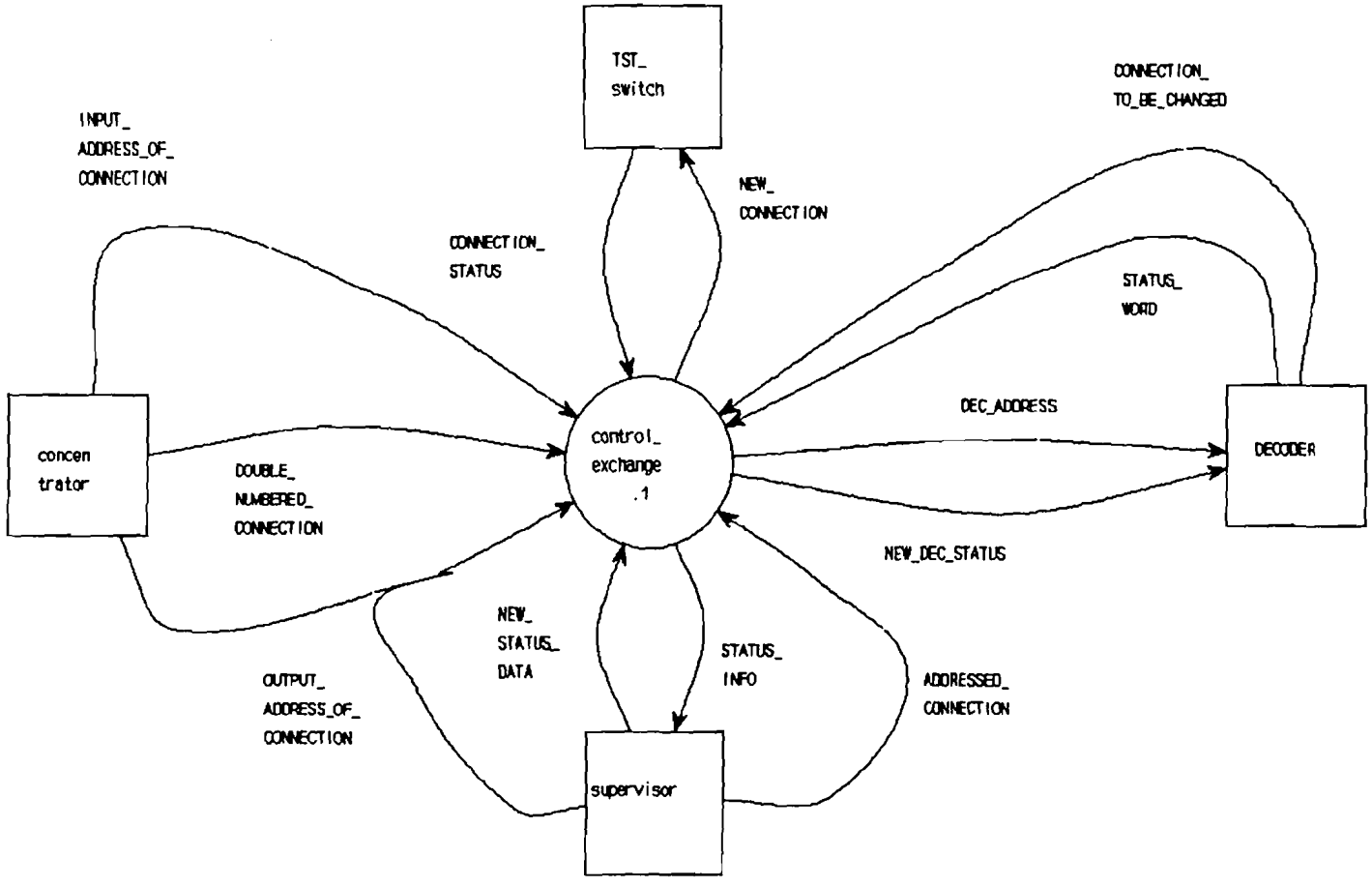
iC-86/286 COMPILER USER'S GUIDE FOR iRMX SYSTEMS.
Version R4.1,
Santa Clara, Intel Corporation, 1989.

iC-86/286 LIBRARIES SUPPLEMENT.
Version R4.1,
Santa Clara, Intel Corporation, 1989.

iRMX II.4 OPERATING SYSTEM.
Volumes 1 - 6
Santa Clara, Intel Corporation, 1989.

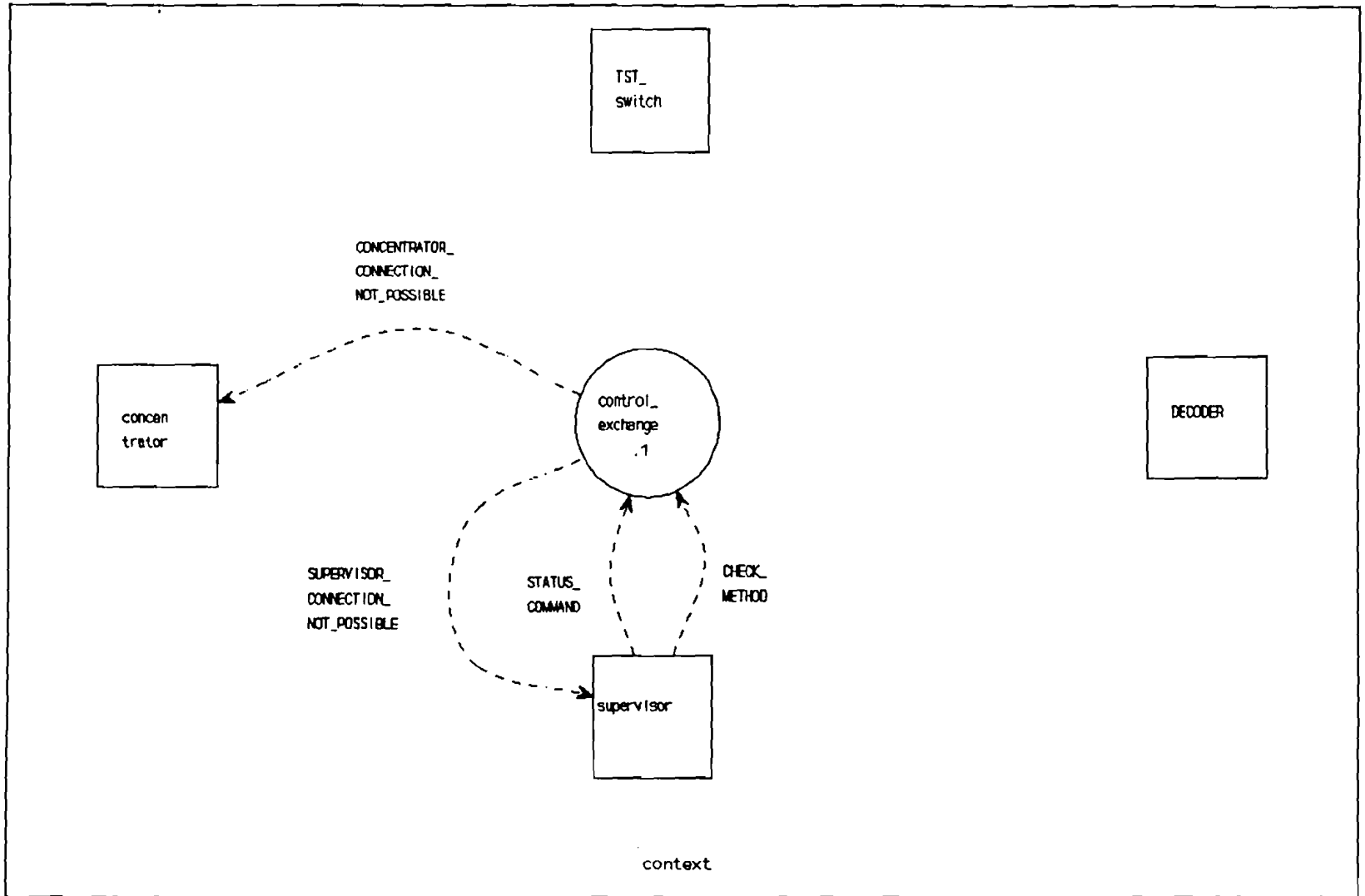
PROMOD COMPUTER AIDED SOFTWARE ENGINEERING USERS MANUAL.
Version V1.7a,
Aachen, GEI, July 1989.

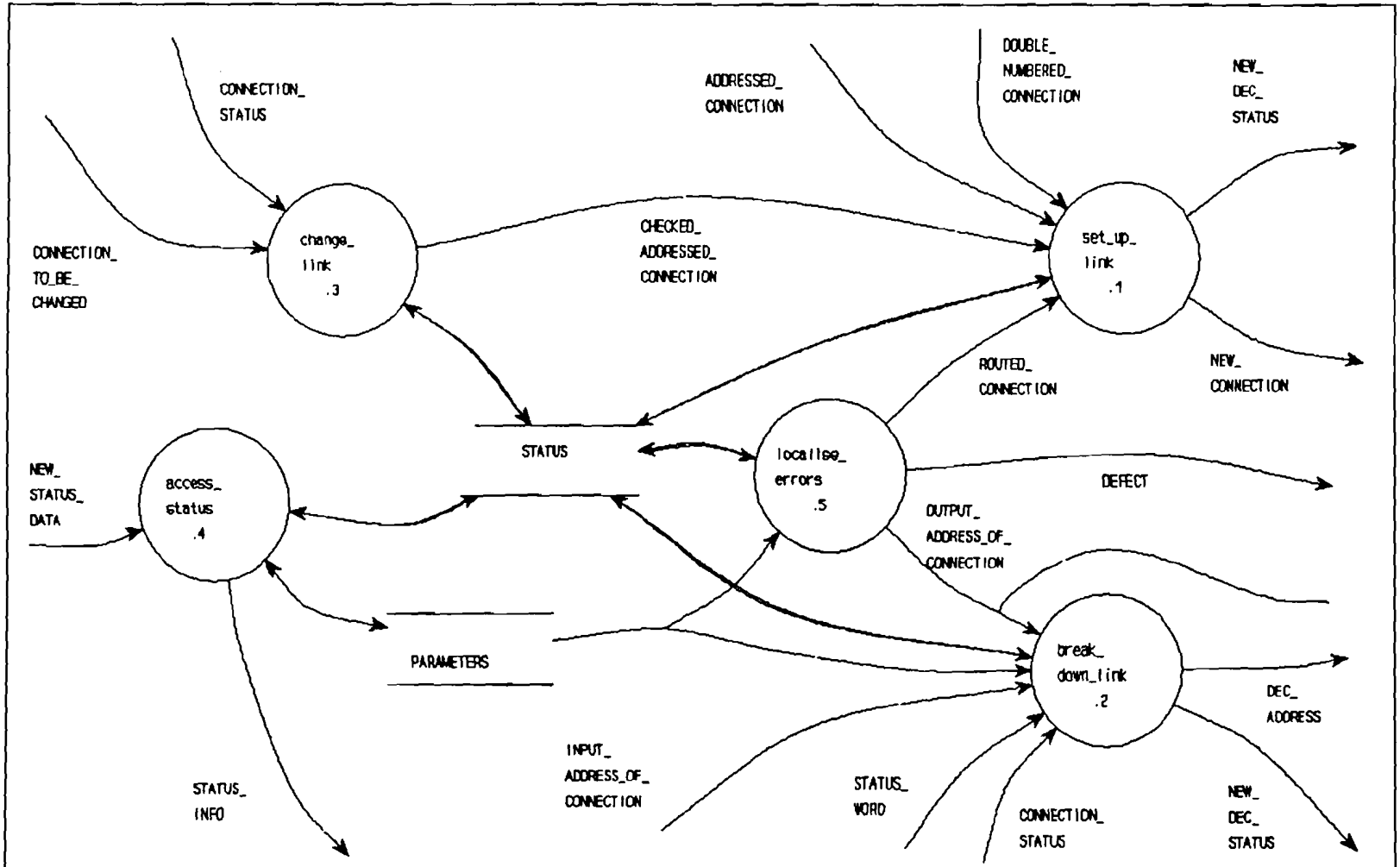
Appendix 1:
Flow Diagrams level 0 and 1



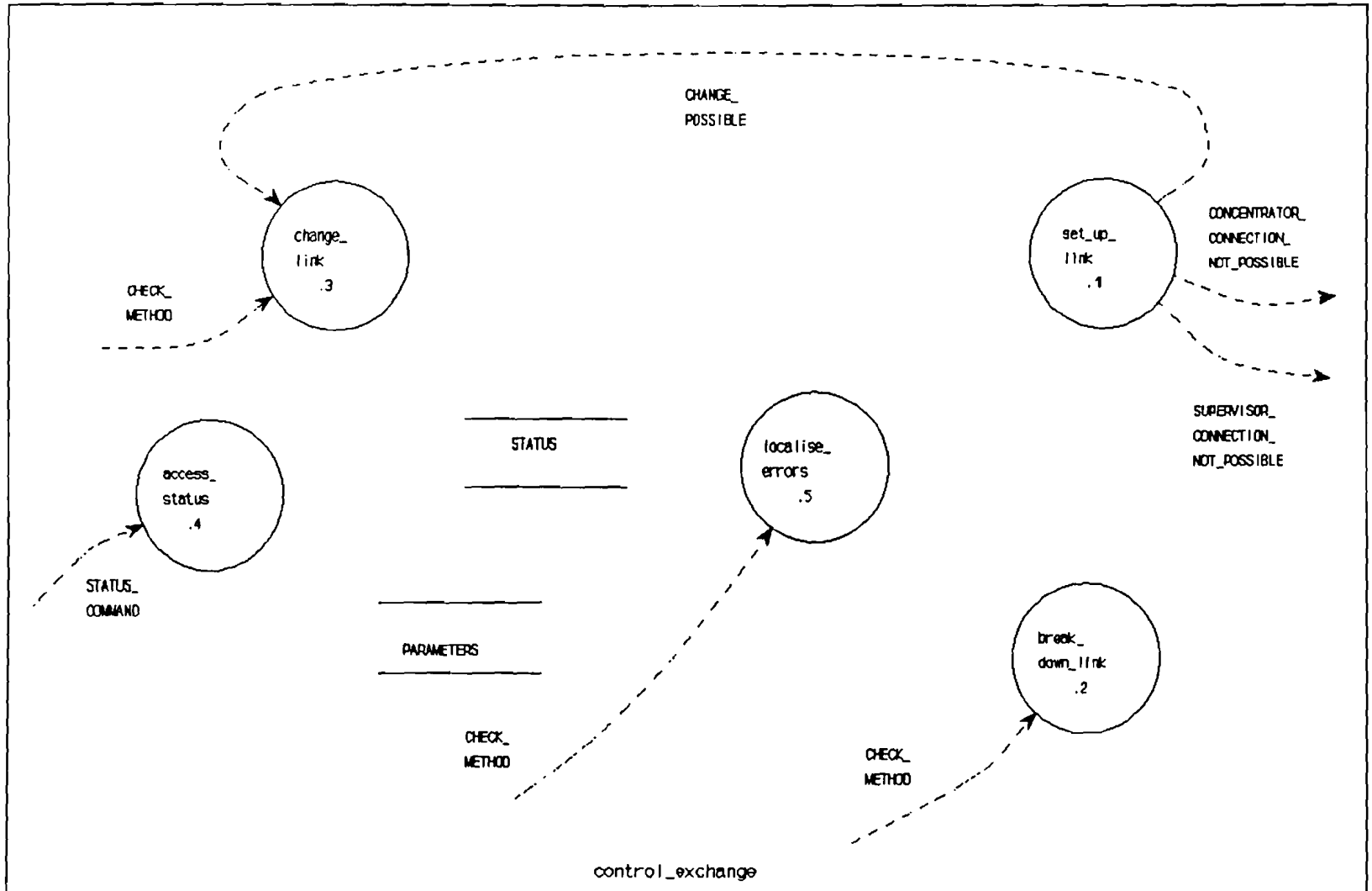
context

LEVEL: 0.0	LAST UPDATE: 08-FEB-1990 16:15
PROJECT: ctrl5	LAST ANALYSIS:


TST_
switch
CONCENTRATOR_
CONNECTION_
NOT_POSSIBLE
concen
trator
control_
exchange
.1
DECODER
SUPERVISOR_
CONNECTION_
NOT_POSSIBLE
STATUS_
COMMAND
CHECK_
METHOD
supervisor
context
LEVEL: 0.0
LAST UPDATE: 08-FEB-1990 16:15
PROJECT: ctrl5
LAST ANALYSIS:

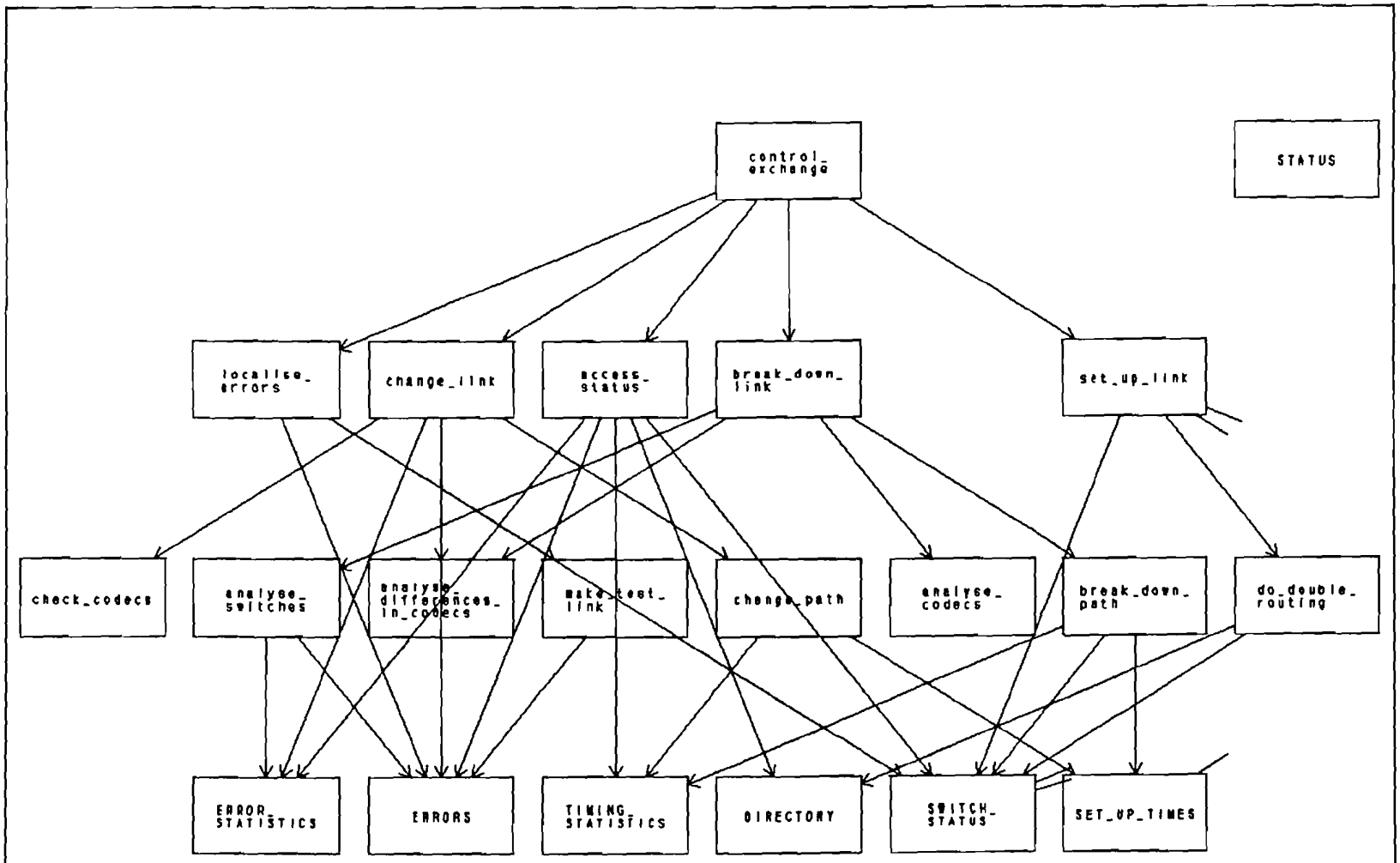


LEVEL: 0.0	LAST UPDATE: 23-FEB-1990 16:31
PROJECT: ctrl5	LAST ANALYSIS:



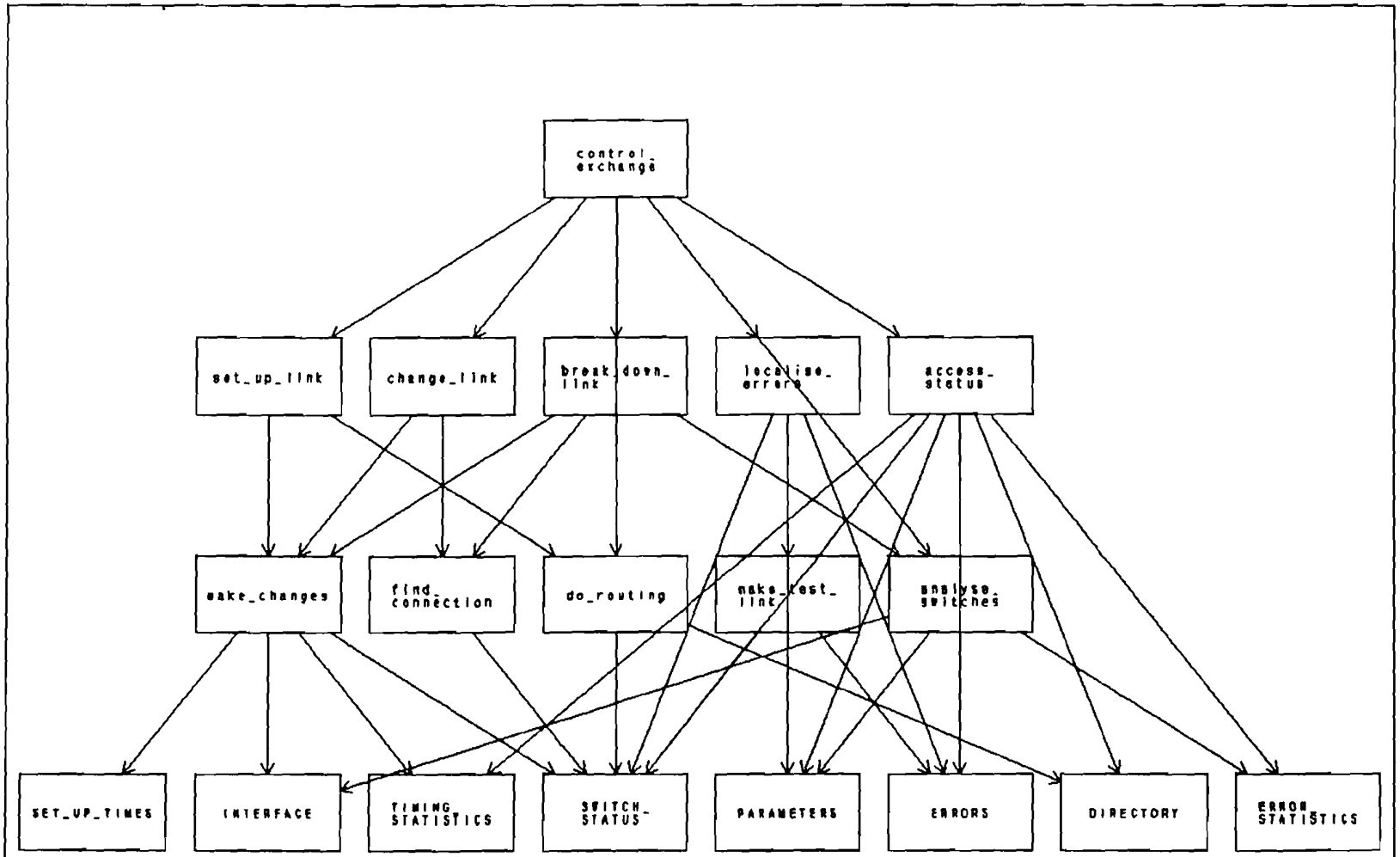
LEVEL: 0.0	LAST UPDATE: 22-MAY-1990 10:51
PROJECT: ctrl5	LAST ANALYSIS:

Appendix 2:
Module Hierarchy
by ProMod



Project: ctrl5	Date: 12-DEC-1989 13:18
Module Hierarchy	Last Analysis: 12-DEC-1989 12:31

Appendix 3:
Final
Module Hierarchy



Project: ctrl5	Date: 8-MAR-1990 10:44
Module Hierarchy	Last Analysis: 05-MAR-1990 14:57

Appendix 4: Using iRMX

This appendix describes some inconsistencies, bugs (??) and tricks I found, using iRMX and the compiler and binder for iRMX systems. It also contains a list of the manuals necessary to understand the features used in the control.

First of all, the iRMX partition on the disk can be reached by giving the command 'bootrmx' in the directory c:\rmx. This invokes the bootstrap loader which automatically boots iRMX. When it has been booted, the user can log in. You can always log in with the name 'world'. The password is empty for that username. After logging in, the user can invoke Human Interface (HI) or Command Line Interpreter (CLI) commands to handle files, change directories, etc.. The manual 'HI operators Guide' (iRMX II.4 Operating System, Volume 2) explains the file structure as well as the commands that can be entered by the user. By typing 'a', the user gets a list of aliases for these commands that have already been defined and can be used instead of the long commands. When needed, the user can also define aliases himself.
Logging of is accomplished by typing: logof.

RMXDISK

Another utility that can be invoked from the (DOS) c:\rmx directory is 'RMXDISK'. It is described in Volume 1 of the iRMX Operating System, System 120 User's guide, although not all manuals contain that section. With this utility it is possible to copy files from the RMX partition to the DOS partition on the harddisk. It is invoked by typing:

rmxdisk -ux

Where x is the user number (This can be left out, in which case the invocation is just 'rmxdisk'). All DOS commands can still be entered by preceding them with a '!'. The other commands used in this utility are very similar to HI commands. First the RMX partition is attached by typing:

a(ttach)d(evice) c: as :SD:

The letters between brackets are optional.
The directory in the RMX partition to which or from which files must be copied can be set by typing:

a(ttach)f(ile) :SD:*directory_name* as :$:

Where *directory_name* is the RMX directory. However, I found that once this directory was set, it could not be changed anymore (Well, you can, but when you ask for a listing of the current directory, it remains the same). If files must be copied from another directory as well, rmxdisk must be invoked again.

The copying from RMX to DOS is done by typing:

copy :$:*file_name* to c:*dos_directory*

Where *file_name* is the file to be copied and *dos_directory* is the destination on the DOS partition. From DOS to RMX is done by:

copy c:*dos_file* to :$:

:$: is always the current directory in the RMX partition.

After copying, the RMX partition must be detached, and this is where the problem arises. Detaching is done by typing:

d(etach)d(evice) :SD:

However, the utility will respond by giving the error message 'bad file number'.This error message is not contained in the list of possible messages for the utility. It is no problem when copying from RMX to DOS, but from DOS to RMX the files will not be copied. Even though they are contained in the directory listing for the RMX partition after copying. A solution which worked, is to attach the partition again and immediately detach it. So after the error message, type:

a(ttach)d(evice) c: as :SD:

d(etach)d(evice) :SD:

In this case the detaching will work and files are transferred correctly. The utility can be left by typing 'q'.

Copying files to/from disk

When logged in, files can be transferred to/from a floppy disk. However, every diskdrive is viewed by the Operating System as an external device and must be attached. However, the manuals didn't contain device information for the disk drive on the system 120 (at least, I couldn't find this information). Luckily the list of aliases contains commands that seem to work with high density disks. The disk must first be inserted after which the command

adah

must be typed in (This is an alias for attachdevice ah as :a:). After this, files on that disk can be accessed by using the logical name :a: for the device. After use, the device must be detached by typing:

dda

This is an alias for detachdevice :a:.

Even when another disk is inserted, the device must be detached, and then attached again with the new disk. Failing to do so may corrupt the information on the newly inserted disk !!

Formatting of high density disks is done by using the HI command 'format'. No attributes are necessary.

Supervisor mode

The supervisor mode can be entered by any user that knows the supervisor password, by typing:

supervisor

The mode can be left by typing:

exit

The supervisor has access to all files and can add new users.
Note that when the system was broken down incorrectly (e.g. by CTRL,ALT,DEL) this is reported before a new log on. The system must first be shut down correctly using the alias 'sh' in the supervisor mode. After the system has been shut down, it can be rebooted with CTRL,ALT,DEL. This can only be done in the supervisor mode. Users that do not know the password can not shut down the system, and therefore cannot go to the DOS partition correctly.

Using C with iRMX

iRMX system calls can be used in C. They must first be declared using the header files from the iC-86/286 library (in the directory /lib/ic286). These files also contain type and macro declarations useful for these system calls. To access these files during compilation you must first declare the logical name :include: by typing:

a(ttach)f(ile) /lib/ic286 as :include:

Also, during binding, the right interface libraries must be included ,explained in iRMX II.4 Operating System vol. 4, Programming Techniques. This manual also includes example programs on how to use system calls. However, these examples are not in C. Therefore, the C-compiler contains a demo program explaining certain iRMX concepts and the proper use of system calls as well as some useful header files (in the directory /rmx286/demo/c/intro).

The compiler generates the warning 'indirection to different types' for the returned status on some system calls. Usually, this means that a pointer that points to an element of a certain type is used as a pointer to an element of another type. But in this case, the cause could not be found. However, this warning is also generated for the demo program supplied with the compiler. So it is probably not serious and a bug of some kind in either the compiler or the supplied library header files. In any case, it does not interfere with the correct operation of the system calls or the program in general.

Also, the binder generates 'symbol types mismatch' errors when using system calls. This usually means that functions are declared with different types in the program. The reason for this could not be found either. In the demo program it is handled by just performing no type checking by using the 'notype' specifier. I adopted this solution. However, it is advisable to do type checking when developing an application to check your own functions.

MAILBOXES

When data mailboxes are used, the length of the message must be specified when it is sent to the mailbox. This can be accomplished using the sizeof() operator. When a message must be collected from a (data) mailbox, a pointer must be supplied to a buffer of at least 128 bytes even when the message is much smaller. This can simply be accomplished by using a union of the message to be received and a dummy array

of 128 bytes. Suppose you want to receive the variable mes (type is MES_TYPE) from a mailbox. This cannot be done by simply supplying &mes to the system call RECEIVE DATA because this does not create a buffer of 128 bytes. The solution is to declare the following union:

```
union
 {
      MES_TYPE mes;
      BYTE dummy[128];
 } buffer;
```

The pointer &buffer is supplied to the RECEIVE DATA call (actually rq$receive$data) because this buffer is 128 bytes. Because when a union is defined, enough space is reserved for its largest member (dummy[128], note that the type BYTE is declared in the header file rmxc.h which must be included anyway when using system calls). The received message can be accessed as buffer.mes.

Required reading

Because the number of manuals for the iRMX Operating system is very large, I've supplied a list of manuals used when developing the control. When just the volume is mentioned, it means that it is in one of the volumes from the iRMX II.4 Operating System.

- Introduction (Volume 1)
  Contains general information on the operating system's facilities.
- System 120 User's Guide (Volume 1, though not always present)
  Contains information about logging on and the DOS utilities.
- HI Operator's Guide (Volume 2)
  Explains the file structure and operator commands.
- Nucleus User's Guide (Volume 3)
  Describes the multi-tasking features, communication, synchronisation, interrupts, etc.
- Nucleus System calls reference manual (Volume 4)
  Explanation of each system call in the nucleus.
- Programming Techniques reference manual (Volume 5)
  Contains information about the use of system calls in applications.
- AEDIT text editor user's guide for iRMX 286 systems
  Explains the text editor on iRMX systems.
- iAPX 286 Utilities user's guide for iRMX 286 systems
  Explains the operation of the Binder.
- iC-86/286 Compiler User's Guide for iRMX systems
  Explains the operation of the Compiler.
- iC-86/286 Libraries Supplement
  Lists all library functions and header files.

Appendix 5:
User's Manual Control version 1.2

This appendix describes the control as it presents itself to the user.

Invocation

After the software has been compiled and linked, the control can be invoked by typing:

control

The control starts by initialising all data structures and the interface and exchange (when connected). The following default values are set for run-time adjustable parameters:

Length test link	Default is: 2 sec.
Error limit, test links	Default is: 10
Error limit, broken slots	Default is: 50
Check Method	Default is: Select all methods
Weights	Defaults contained in weights.def

The purpose of these parameters is explained in the following sections.

States

At the moment there are basically two states that the control can be in: 'active control', or 'supervisor control'.
In the 'active control' state, the control is fully operational and can process requests from outside. But because there is no real exchange attached to this version of the control, nothing will happen after initialisation. Requests can now only be generated using the 'supervisor control'. You get to the 'supervisor control' state by typing 's' followed by 'enter'. When entering this state, a list of options is displayed from which the supervisor can choose. After an option has been chosen, the control may ask for information necessary to perform that option. After the supervisor has entered this information, the control performs the option and returns to the 'active control' state.

Basic Operations

Basically, the control is used to set up a connection or break down a connection.

Requests for these operations can be generated using the options 14, 15 and 16 in the 'supervisor control' state. There are two types of connections that can be set up.

An addressed connection is a one way connection. The incoming and outgoing slot must be given and the control will set up a connection from that incoming slot to the outgoing slot. Note however that the incoming slot on the display reporting the set up is one higher than the slotnumber that was entered. This is, because the control always shows internal numbers. These are the numbers for the slots as they are used by the TST-network. The incoming slot that was entered is expected to be an external number, the slot number actually entering the exchange. However, the synchronisation introduces a delay of one time slot, so the data will enter the TST-network one time slot later, so one higher in number. This kind of delay is only relevant for incoming slots.

The second type of connection is the double numbered connection. This is a connection as used for a telephone call, two way. Two connections are actually set up. One from caller to receiver and one from receiver to caller. To generate a request for this kind of a connection, the control needs the subscriber number for the receiver (which should be present in the directory), the incoming slot for the connection from caller to receiver, and the outgoing slot for the connection from receiver to caller (so the slot where the caller expects the answers from). So two connections are set up with this information, with the incoming slot again adjusted for synchronisation delay.

The control can break down a connection given the incoming or outgoing slot for the connection. Note that the incoming slot must be the external slot number because usually these requests come from the 'outside world' that doesn't know about internal delays in this exchange.

Error Localisation Strategy

Because the exchange has got a (4,2) structure, there are some possibilities for error localisation. How and if these errors are localised is determined by the parameter Check Method which can be modified using option 18.

The default value for the Check method is: Select all methods. This means that the entire error localisation strategy is performed. For every slot in every area a total weight of errors is kept. Normally, a connection is checked when it is broken down. When the decoder for that connection indicates that no errors occurred during the connection, no error localisation is performed. If an error occurred, the connection memories for the slots in that connection are checked. If one of them contains a wrong value, the total weight of errors for that slot is increased by 10 (BC_ERROR defined in defs.h). If no error is found, the total weight of errors for all slots in the connection are increased. How much, is determined by the Weights. They contain for every status word a weight for every area. They can be modified using option 13.

The same happens when a decoder interrupt occurs (when at least two non-correctable errors have been detected by the decoder). Only then the error cannot be

found in the connection memories, a weight of 10 (interrupt error) is added for every slot.

The total weight of errors is constantly checked for all slots. If it is greater than the error limit for broken slots (option 11 to modify, default value 50) the slot is marked as broken and will no longer be used in connections. If it greater than the error limit for test links, a test link is set up through the slot. The test link lasts as long as indicated by Length test link (default is 2 sec.). After this period of time the link is broken down and checked for errors like normal connections. Only when no errors are found, the total weight of errors for all slots in the test link is reset (to 0).

Other values for Check Method are:

Select check connections
Everything is the same, except no test links are set up.

Select make test link
Everything the same as for Select all methods, but the connection memories are not checked when an error has been detected, or when a decoder interrupt occurs.

Select no localisation
Everything the same as for Select all methods except no test links are set up, connection memories are not checked, and the total weight of errors is not checked anymore to find broken slots. This explicitly cancels an entire task so computing time becomes available for actually processing requests (e.g. during busy hours). So the only part of the strategy that is still active is the updating of the total weight of errors when a connection is broken down (or in case of an interrupt error).

Select no redundancy
The same as Select no localisation except that a connection is not even checked for errors anymore and the total weight of errors is not updated.

Supervisor options

These are the options when in the state 'supervisor control' (type 's' followed by 'enter'):

1 Show Defects
Shows all slots that have been marked as broken.

2 Show directory
Shows the directory of subscriber numbers. It tells which number is connected to which incoming and outgoing line.

3 Show length test link
Shows how many clock ticks a test link lasts. (default 1 tick = 10ms)

4 Show error limits
For every slot in every area a total weight of errors is kept. These limits decide when a test link is set up through a slot and when it is marked as broken.

5 Show weights of errors
Shows which weights are added for every area when a connection is broken down and a particular error/mode vector is found in the decoder.

6 Reset error
Resets the total weight of errors for a particular slot.

7 Reset slot (mark as not broken)
Can be used to mark a slot as operational again after the problem has been tracked down that caused it to be marked as broken.

8 Add directory entry
Can be used to add another entry to the directory (number + incoming line + outgoing line).

9 Delete directory entry
To delete an entry (subscriber number with incoming and outgoing line from the directory.

10 Change length test link
Changes the length of a test link to a specified number of clock ticks.

11 Change error limits
Changes the limits above which a slot is marked as broken or above which a test link is set up through the slot.

12 Mark slot as defect
Can be used to mark a slot as broken. This means that it will not be used in connections anymore.

13 Change weights for a status word
Allows the supervisor to change the weights for a status word. These weights are added to the total weight of errors for the slots in a connection when the error could not be tracked down. (This happens when a connection is broken down)

14 Set up addressed connection
Allows the supervisor to make a request for a one way connection. If the specified slots are free and other internal slots for the connection can be found, it is set up.

15 Set up double numbered connection
Allows the supervisor to set up a two way connection. The number for the receiver must be specified as well as input address for the connection from caller to receiver as the output address for the connection from receiver to caller.

16 Break down connection
Allows the supervisor to break down a connections. This can be done using the input address (incoming line and slot) or the output address (outgoing line and slot) for the connections. Test links should not be broken down by this option, they take care of their own break down request. Only in extraordinary cases may test links be broken down by this option and only with the OUTPUT ADDRESS. Using the input address may lead to a break down of the control.

17 Show check method
Displays the error localisation strategy and explains what it means.

18 Change check method
Allows the supervisor to change the check method, for example during busy hours.

19 Write slot error (testing)
Allows the supervisor to write a particular weight of error to the total weight of errors for a certain slot in a certain area. Normally this feature is not needed, but it can be used to test the software. When a weight is written that is equal or greater than the second error limit, the slot should be reported as broken by the software (task localise errors). When a weight is written that is smaller than this limit but equal to or greater than the first limit, the software should start to set up a test links through that slot. (At least when the check method is equal to 'select all methods' or 'select make test link'.)

20 Read slot error (testing)
Can be used to check the total weight of error for a particular slot in a certain area.

21 Generate decoder interrupt (testing)
This is for test purposes. As there are no decoders attached to the control yet, this feature allows the supervisor to generate a decoder interrupt (for a certain outgoing line and slot) for a connection that is currently set up. Normally this interrupt would occur when two non-correctable errors have been detected in a connection. The control tries to change the connection by choosing another intermediate slot. The incoming and outgoing slot cannot be changed because they make the connection with the 'outside world'. When another intermediate slot can be found, the new connection is set up. Part of the old connection is checked by reading from the connection memories to see if they contain the right values. (The t2 switch cannot be checked because the rerouted connection has already written another value in that connection memory cell.) If no errors are found in the connection memories, as will be the case when no actual exchange is connected to the control, an INTERRUPT_ERROR (defined in defs.h) is added to the total weight of errors for all slots of the connection that caused the interrupt. The interrupt error is currently set to 10, as is the first error limit. This means that test links are set up through the slots as soon as they are free. The s_in and s_out slots are already free (because another intermediate slot was chosen) so test links are set up through them. Test links are also set up through the T1_in slot although it may still be occupied. Due to the structure of the switches and the control software, this is no problem. Test links through the T2_out slot are set up as soon as that slot is marked as free. (So after the connection has been broken down.) No errors will be found in this testing situation, so the total weight

of errors is reset when the test connection is broken down.

22 Display current connections
Allows the supervisor to display the connections that are currently set up in the exchange. (Actually in the control data structure switch_status.)

Remember that before another option can be selected, the user must first go to the 'supervisor control' state by typing 's' followed by 'enter'.

The program can be stopped by using the regular command CTRL-C (regular in iRMX systems at least).