FUNCTIONAL PARTITIONING OF MULTI-PROCESSOR ARCHITECTURES

by

SAMIR S. AL-KHAYATT, BSc, MSc

A Doctoral Thesis
submitted in partial fulfilment of the
requirements for the award of the
degree of
Doctor of Philosophy of the
Loughborough University of Technology

July 1990

Supervisor: J.E. Cooling, BSc, CEng, MIEE, MIEEE

Department of Electronic and Electrical Engineering
Loughborough University of Technology

To My Family

SYNOPSIS

FUNCTIONAL PARTITIONING OF MULTI-PROCESSOR ARCHITECTURES

Many real-time computations such as process control and robotic applications may be naturally distributed in a functional manner. One way of ensuring good performance, reliability and security of operation is to map or distribute such tasks onto a distributed, multi-processor system. The time critical task is thus functionally partitioned into a set of cooperating sub-tasks. These sub-tasks run concurrently and asynchronously on different nodes (stations) of the system. The software design and support of such a functional distribution of sub-tasks (processes) depends on the degree of interaction of these processes among the different nodes.

The research work carried out is concerned with the following points:

* The design and implementation of a loosely coupled multi-processor system that has been designed and implemented for use in fault-tolerant, real-time applications. Each processing unit (station) consists of a single board computer, where the communication and processing tasks are decoupled on each board. It uses a single shared parallel bus for communication between these stations, bus control being fully distributed.

* The development of software environment to support functional partitioning. This consists mainly of:

    i) A real-time kernel structure to support and manage partitioned sub-tasks on various processing sections of the system.

ii) A communication software protocol that supports communication between the different processing sections of the system. This is performed using message passing techniques based on token passing.

iii) A run-time support system for the operation of the communication protocol.

The communication and real-time kernel software have been written mainly in Modula-2. This required the use of two different compilers. A small amount of assembly language programming was also used. This software is hosted on a multi-processor demonstrator system which has been developed as part of the research programme.

ACKNOWLEDGEMENTS

I wish to express my deep sense of gratitude to my supervisor Mr J E Cooling for his guidance, inspiration, stimulating discussions, and most of all encouragement during the preparation of this work.

Thanks are also due to the following:

* The staff, and technicians for their valuable assistance.

* My wife for her endurance and support throughout the project.

TABLE OF CONTENTS

LIST OF FIGURES

Page No

CHAPTER 4

CHAPTER 5

CHAPTER 6

Page No

CHAPTER 7

CHAPTER 8

APPENDIX A

LIST OF CHARTS

Page No

LIST OF ABBREVIATIONS

GENERAL

ASCII - American Scientific Code for Information Interchange.

1100B - Binary number.

BIT - Binary Digit.

BYTE - 8 bits.

CAD - Computer Aided Design.

CPU - Central Processing Unit (80188/64180).

CS - Count of Stations.

DMA - Direct Memory Access.

DRAM - Dynamic Random Access Memory.

DUART - Dual Universal Asynchronous Receiver Transmitter.

EPLD - Erasable Programmable Logic Devices.

EPROM - Erasable Programmable Read Only Memory.

0FDH - Hexadecimal number.

FS - First Station.

I/O - Input/Output.

JSP - Jackson's Structured Program.

LS - Last Station.

NPE - Numerical Processor Extension (8087).

NS - Next Stations.

OBI - On-board Interfacing.

PCB - Printed Circuit Board.

PDF - Program Design Facility.

PS - Previous Station.

ROM - Read Only Memory.

RPC - Remote Procedure Call.

SRAM - Static Random Access Memory.

TMS - Temporary Memory Store.

TPBAM - Token Passing Bus Access Method.

TS - This Station.

VCB - Variable Control Block.

VDU - Visual Display Unit.

SIGNALS

(*) - Active low signal.

ALE - Address Latch Enable.

ACK - Acknowledge.

CLK - Clock.

DT/R - Data Transmit/Receive.

DRQ - DMA Request.

EDT - End of Data Transmission.

LMCS - Lower Memory Chip Select.

M/IO - Memory access/Input-Output access.

MMCS - Medium Memory Chip Select.

PCS - Peripheral Chip Select.

RD - Read signal.

RDT - Request Data Transmission.

RDY - Ready.

UMCS - Upper Memory Chip Select.

WR - Write signal.

CHAPTER 1

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

Recent advances in hardware computer technology, combined with component cost reductions, have spurred on the development of new distributed hardware systems. Eventually, future real-time applications will be targeted toward highly distributed, multi-processor environments because of their attractive cost-to-performance ratios compared to single processor systems.

As new high performance distributed architectures are explored and exploited, the nature of software developed for these new generations will tend to shift from being sequential in nature to being more parallel.

Developing software for such systems will be even more troublesome than it is for traditional computer systems due to synchronisation issues, new algorithms and languages. Yet, there is currently little software support for distributed environments.

Some vendors have succeeded in developing quality software for non-sequential structures using conventional technologies. However, lack of specialised support is already hindering long scale development of systems with this class of architectures.

Successful software development, however, will only stem from a better understanding of distributed systems. The design and synthesis of software for distributed systems requires the use of a design methodology and programming language which builds on the inherent parallel nature of such systems. Thus, an adequate software base (design tools, run time environments, dedicated operating systems, compilers, etc) and better software engineering techniques must be available before future needs, for high quality software, can be met.

Intense research activity in recent years has led to a more mature understanding of the problems of a distributed environment [1]. Still, however, the following points need to be resolved to facilitate advances in software development for distributed architectures [2]:

* Approaches to problem decomposition for mapping applications to the proper distributed architecture.
* Techniques for software design partitioning and allocation.
* Language issues for distributed architectures for future systems (e.g. language constructs to address parallel issues).
* Algorithm design and evaluation.
* Problem visualisation and animation techniques.
* Software testing to attain high reliability levels.

1.2 RESEARCH OBJECTIVES

Real-time, multi-processor, embedded systems are one application area where response times, throughput, reliability and fault-tolerance constitute the major design criteria [3]. Hence the distribution and management of the application software is a critical function.

A prototype loosely-coupled multi-processor system has been designed and implemented for use in fault-tolerant real-time applications (Figs. 1.1 and 1.2).

This thesis discusses the organisation and structure of the total system, concentrating in particular on the software environment that has been developed to support functional partitioning [4,5]; i.e. the communication and executive (kernel) functions. The communication system is based on a token passing bus protocol for use with single board computers connected via a fast parallel bus. The kernel is designed to support functional partitioning of application programs, and can be implemented using standard compilers. No special multi-processing features are required. Most of the software for this system has been written in a high level, structured language (Modula-2), though assembly language programming has been used in a few specialised areas.

1.3 THESIS ORGANISATION

Chapter 2 presents methods of task management in distributed environments. Partitioning schemes and allocation strategies are highlighted in particular.

Chapter 3 gives a general review of distributed, concurrent programming techniques. Different classifications and methods are presented together with the evaluation of each method.

Chapter 4 is devoted to the functional description of a multi-processor structure that is designed to support functional partitioning. Supporting issues such as inter-processor communication, operating system constructs, and choice of programming language are discussed within this chapter.

Chapter 5 describes the implementation of the multi-processor system developed in this research project at building block level. The function of each block and its role in the system is demonstrated. It introduces the idea of using two separate sub-systems; the communication sub-system for handling communication with the network and the processing sub-system for the execution of application tasks.

Chapter 6 and Chapter 7 concentrate on the design of the software environment which has been developed to support functional partitioning. Chapter 6 describes the implementation of the communication protocol and its run-time support system. Chapter 7, on the other hand, describes the structure and implementation of an operating system kernel for the support of functional partitioning. Software structure diagrams for Chapter 6 and Chapter 7 are given in Appendices C and E respectively.

Chapter 8 introduces the different approaches and techniques for the testing and validation of both the hardware system, and the implemented system software modules.

Finally, Chapter 9 reviews and assesses the different achievements of the research work. It also highlights areas for future research.

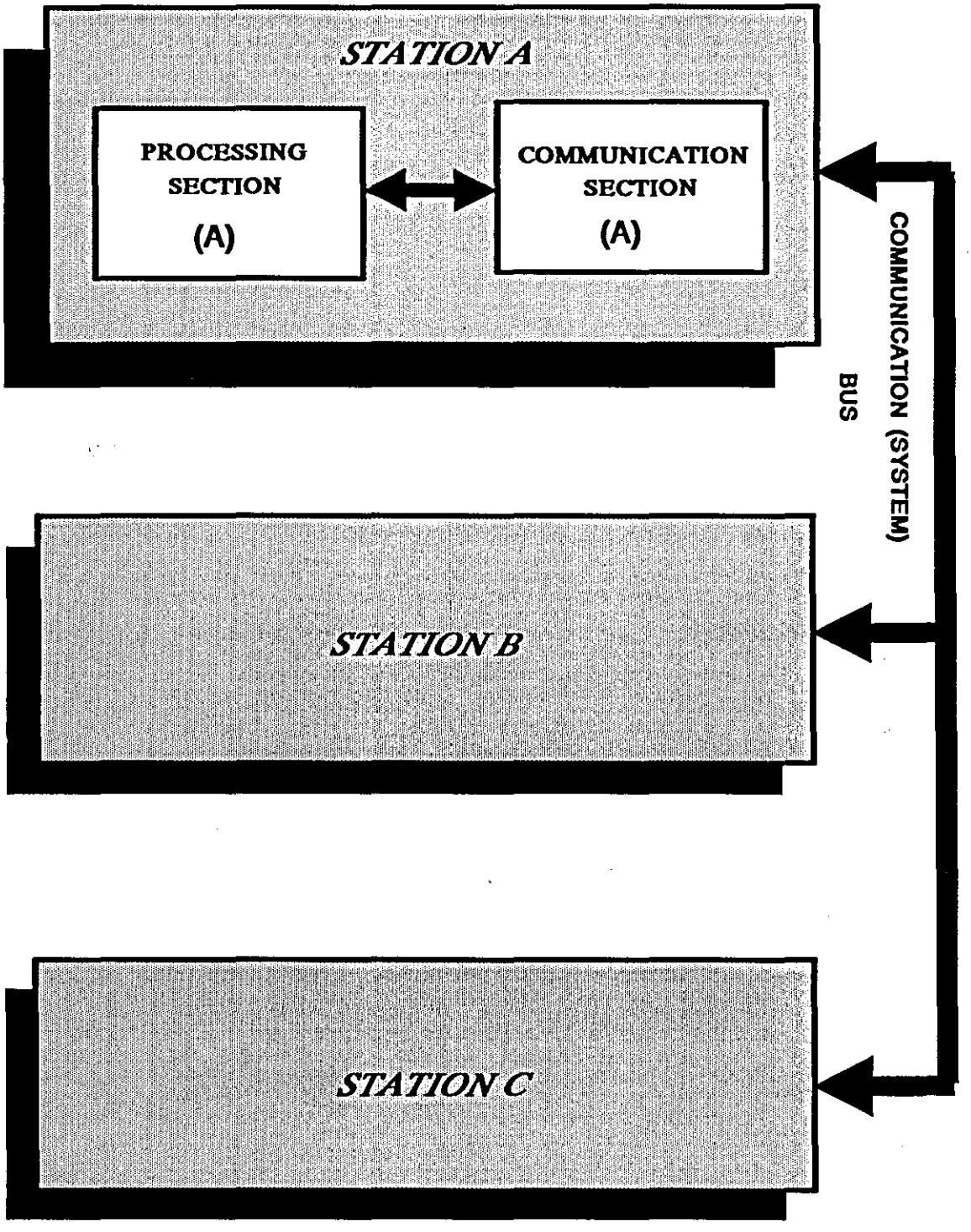


Fig. 1.1 SYSTEM CONFIGURATION

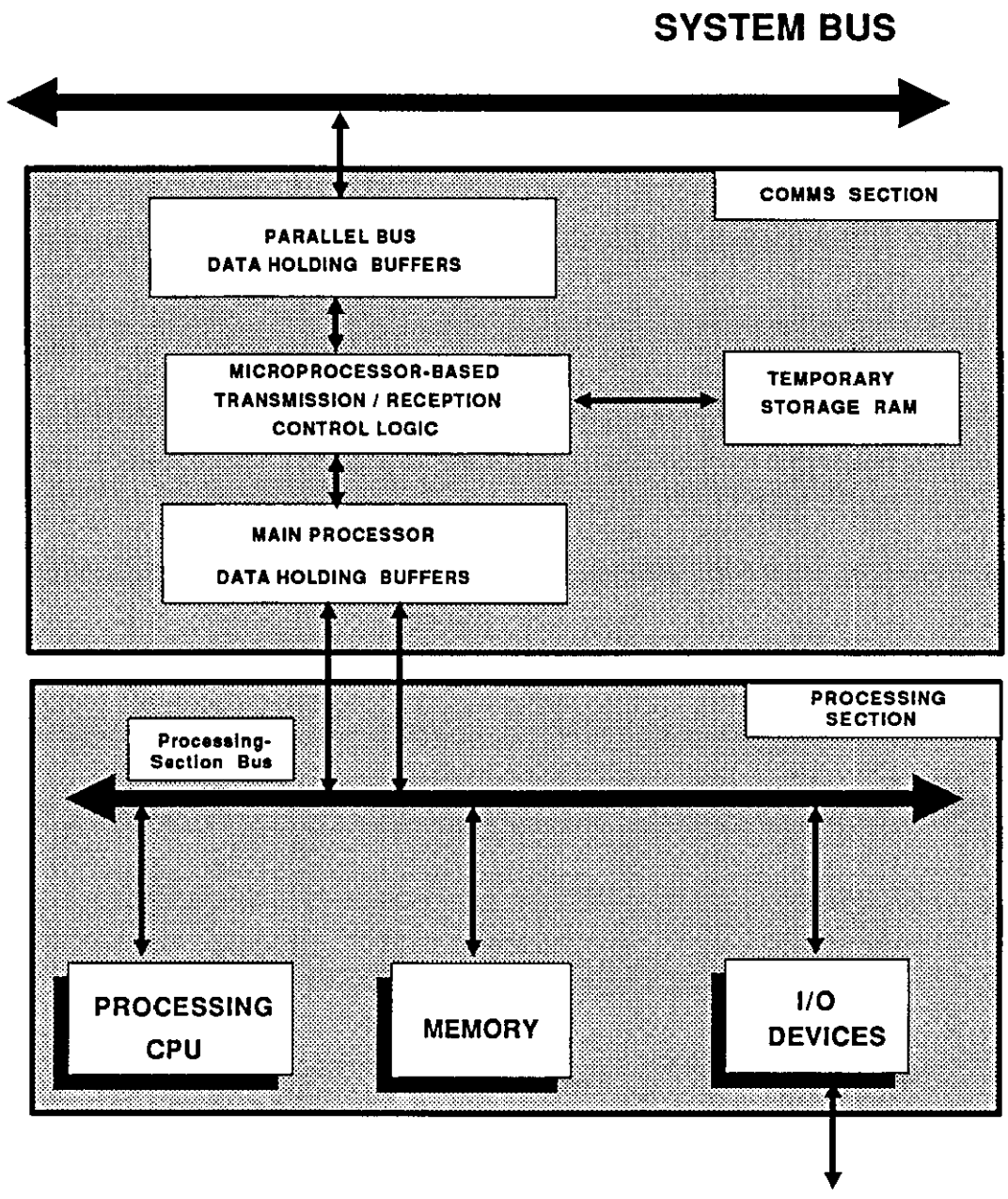


Fig . 1.2 MULTIPROCESSOR NODE - FUNCTIONAL STRUCTURE

CHAPTER 2

CHAPTER 2

METHODS OF TASK MANAGEMENT IN DISTRIBUTED SYSTEMS

2.1 GENERAL

Distributed systems have inherent problems which must be overcome by different concurrent programming methodologies. Certain demands and requirements have to be met in the design of distributed programs. Issues such as the complexity of the underlying hardware, partitioning and allocation schemes, the supporting constructs of the programming languages and the availability of the software environment tools play major roles in task management within a distributed environment. This chapter highlights the main issues relating to such environments i.e. those of partitioning, allocation, and communication aspects.

2.2 PARTITIONING SCHEMES FOR DISTRIBUTED ENVIRONMENTS

2.2.1 Overview

Partitioning is the process of breaking down a task into smaller tasks (sub-tasks), or a program into smaller programs called fragments or segments.

In many cases the partitions lead to an apparent reduction in the complexity of the system and reduces the problem at hand to manageable pieces. The partitioning unit or construct used (called granularity) should be carefully chosen as this will affect the type of system implementation. For instance, as the number of parallel processes into which a computational task is partitioned is increased, so the volume

of inter-process communications for control and data interchange also increases. This leads to a closely coupled system implementation.

Language constructs play an important part in simplifying partitioning schemes. For instance, the language StarMod [1] allows the programmer to partition a computation into a collection of processes and also to define the details of communication paths between the processors. Ada [2] has been criticised for not providing a suitable partitioning constructs; help is required from other tools in the support environment to provide such a scheme [3,4,5].

Two basic approaches may be identified for partitioning distributed software [6]:

i) Distribute fragments of a single program across processors and use a normal intra-program communication mechanisms for interaction;

ii) Write a separate program for each processor and devise a means for inter-program interaction. This method is not so applicable in distributed environments since the introduction of hardware specifications into a design at an early stage restricts program portability. It also leads to a change in the partitioned program structure whenever the configuration (say number of processors) changes.

2.2.2 Designing a Distributed System as a Single Program

The fundamental concept here is that the application software is viewed as a single program, distributed across the target system. Its main advantage is that all the interfaces between the distributed

fragments can be type checked for compatibility by the compiler. Within this approach two general strategies can be identified: post-partitioning and pre-partitioning [5,7,8]:

i) Post-Partitioning

In this strategy, partitioning of an application program is expressed after the design of the software is complete. The partitioning process does not attempt to force changes in the software design in order to achieve the required partitioning. Fig. 2.1 illustrates a typical ordering of system development steps. Partitioning is performed concurrently with and independently of coding.

The programmer produces an appropriate solution to the problem at hand. It is left to the partitioning specification software (Fig. 2.1) to:

* Describe the target configuration,
* Partition the program into components for distribution, and
* Distribute the components to individual nodes.

This method promotes portable software, i.e the same program can be mapped onto different hardware configurations. However, it needs a language that contains facilities for configuration management.

ii) Pre-Partitioning

This strategy is to select a particular construct as the sole unit of partitioning, to be used throughout the design and programming process (see Fig. 2.2). The notion underlying this strategy is that of a 'virtual node', which is an abstraction of a physical node in the distributed system [4,9]. A virtual node consists of one or more units

(which may share memory) communicating with other virtual nodes via some form of message passing over a communication sub-system. More than one virtual node, however, can be mapped onto a single physical node.

Note that the programmer must accept any constraints the choice of constructs entails (e.g. it might affect inter-process communication or system performance).

The notion of virtual nodes is found in most languages which have been designed specifically for supporting distributed programming (e.g. the 'guardian' of Argus [10] and the 'processor module' of StarMod [1]).

For a language construct to be effective as a virtual node it must be supported by [4]:

* Separate compilation.
* Library units or modules.
* Exception handling facilities to cope with process failures.

2.2.3 Functional Partitioning Schemes

Many real-time applications and tasks may be naturally distributed in a functional manner. Functionally distributed systems are often modelled and controlled as a set of communicating, distributed sub-tasks (processes) [11]. The software for such systems invariably reflects the distributed nature of the application.

The software design and support of such a functional distribution of sub-tasks (processes) depends on the degree of interaction of these processes among the different processors.

A simple implementation of functional partitioning may consist of functional or pipelining partitioning [12] (see Fig. 2.3). Here, the distributed processes interact occasionally, usually for transferring data results, using message-passing techniques.

Fig. 2.4 shows a more realistic approach to, and understanding of, functional partitioning within real-time environments. The total system task is partitioned into a number of functional sub-tasks which are then mapped onto the various nodes of a distributed system. In real-time systems such sub-tasks involve plant interfacing, network control, computation of digital control algorithms, etc. These run asynchronously and concurrently within the distributed system. Distributed processes, however, have to communicate and interact occasionally in order to achieve a common goal [13,14]. Management and interaction of distributed processes is usually achieved by supporting software embedded in each node of the system [15,16]. Some of the main advantages in using this method are:

* The software structures mirror the application structure, this being especially suitable for real-time application tasks.
* The individual software units (sub-tasks) can be implemented, type checked and compiled using uni-processor compilers.
* The granularity (unit of partitioning or sub-task), may be further divided and partitioned into other functional sub-tasks (see Fig. 2.4). These sub-tasks can be mapped, in turn, to one or more nodes of the distributed system.
* Finally, each sub-task can be considered as a unit sole of partitioning. This means, it can be separately processed, coded, and compiled using structured languages suited or even adapted for distributed environment.

2.3 TASK ALLOCATION STRATEGIES

Allocation assumes the existence of well partitioned or predefined units or modules, and discusses how to effectively map or allocate these units or modules to different nodes. The method of allocation chosen should allow for an efficient and reliable implementation of inter-process communication mechanism [17,18].

In distributed systems this effectively means 'how different program segments reside on different processors, and how they interact' [5,19].

The unit of allocation depends, among other things, on the constructs of the language use for the implementation. For instance, in Ada two main constructs have been considered as the basis of allocation; the 'task' and the 'package' [4,9]. The task is unable to encapsulate data in the same way as a package, and cannot be a library unit, hence its usefulness as a unit of distribution is limited. The package, however, is supported, by separate compilation and library units and thus favoured as a distribution unit.

Similarly, in Modula-2 [19] a 'co-routine' and a 'module' are two constructs that may be suggested as units for allocation [20]. Again, a co-routine fails to encapsulate data in the same way a module does, also it cannot be a library unit or even separately compiled. But most important, for a distributed application, the coroutine mechanism should be modified in order to allow for remote procedure invocations and resumptions. The semantics of remote coroutines appear to be applicable to Modula-2 [21]. A module, on the other hand, is inherently suitable for use as a distribution unit. Apart from

separate compilation, and use of library units, there are two main reasons for using a module as a distribution unit [9]:

* Procedures of a module need efficient access to the local shared data of the module. Hence, it is not possible to achieve efficiency if the module is split over several nodes or processors.
* Modules often form monitors [22], where mutual exclusion of processes is to be maintained. This is difficult to implement if a module itself is spread over several nodes.

2.4 INTER-PROCESS COMMUNICATION IN DISTRIBUTED SYSTEMS

Communication constructs fall into two groups; those designed to support processes which reside on the same node, and those used where processes reside on different nodes (processors). For processes on the same node, a typical and standard form of inter-process communication mechanism is the use of shared variables (using monitors for implementing mutual exclusion). Whereas for processes on different nodes, inter-processor communication is frequently implemented using the remote procedure call mechanism (RPC) [23] (this relationship can be viewed as a 'client-server' model).

However, a more constructive way of communication between processes in a distributed environment is through the use of message-passing techniques. Process communication may be implemented in both (or either) asynchronous and synchronous forms, using channels [24] or the rendezvous [2]. In a distributed environment, process communication must be transparent, i.e, the programmer is unaware as to whether processes reside on the same or different nodes. It is left to the

supporting software (operating system kernel) to decide whether processes need inter-process or inter-processor communication. To implement this structure, two types of messages can be executed in a distributed system, E-mode and T-mode messages [25] (Fig. 2.5):

* E-mode message refer to message transactions between various processes of a user program (inter-process communication).
* T-mode message refer to messages exchanged between the kernels or operating systems of two different nodes (inter-processor communication).

Usually all communication between the different processes are issued first as E-mode messages. These messages are subsequently interpreted by the underlying software (usually called a filter process) as to whether the source and destination processes reside on the same or different processors. If they reside on the same processor, then an E-mode message is adequate for communication. However, if they turn out to be on different processors, then a kernel process (usually called a communication process) issues a T-mode message to exchange data between the different nodes. These modes of message communication help constructing a 'naming' scheme in a distributed system.

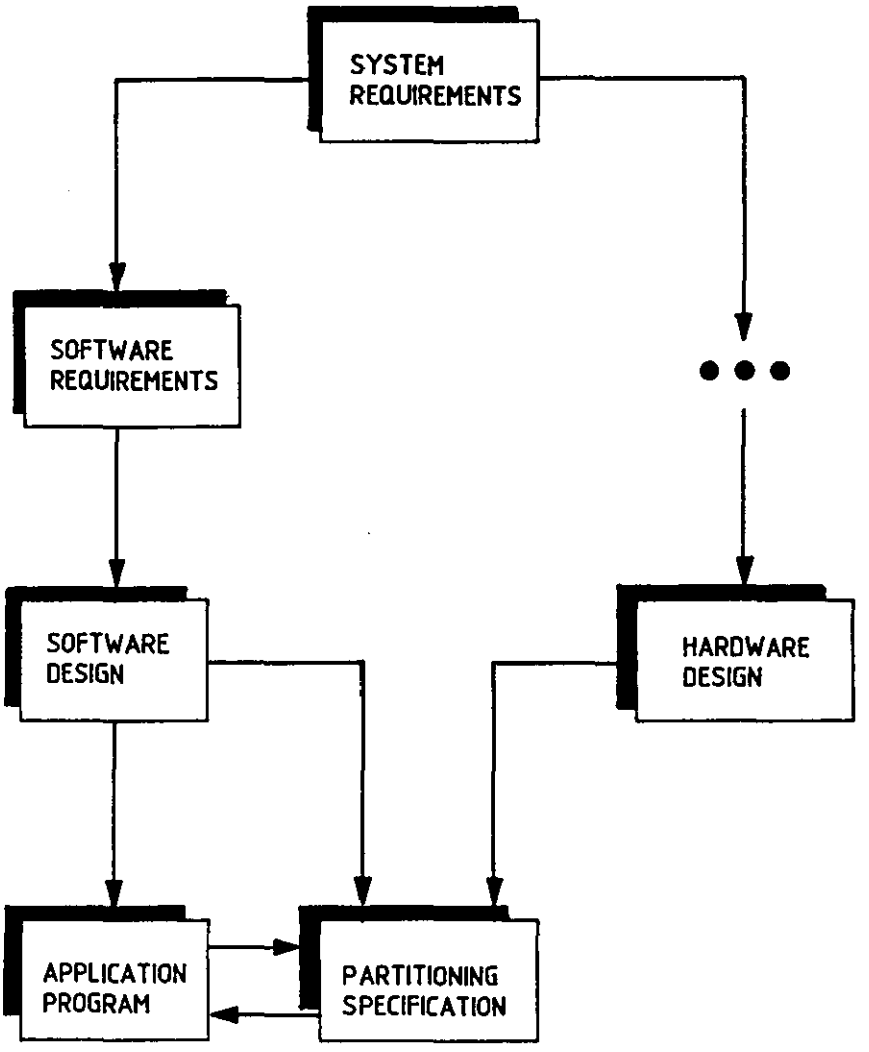


Fig. 2.1 SOFTWARE DEVELOPMENT INCORPORATING PROGRAM POST-PARTITIONING

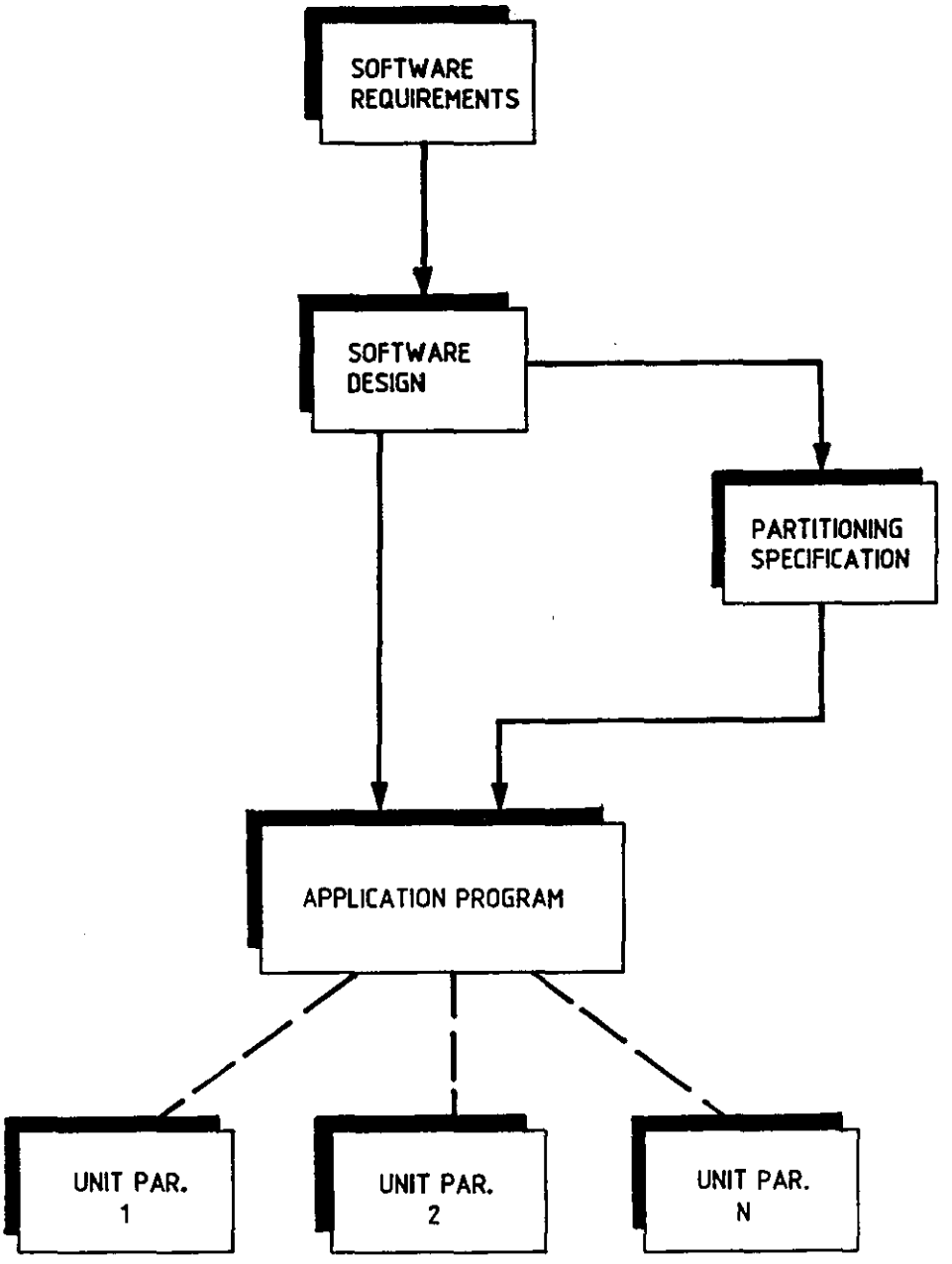


FIG. 2.2 STEPS IN SOFTWARE DEVELOPMENT IN DISTRIBUTED APPLICATION (PRE-PARTITIONING)

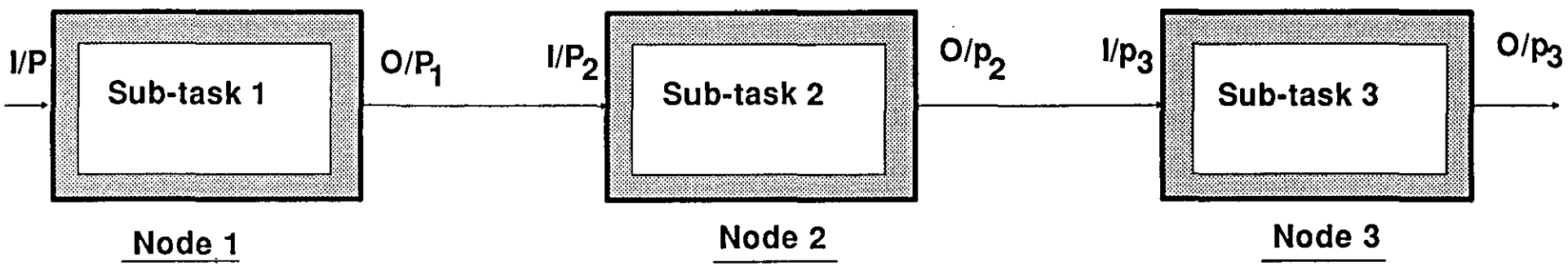


Fig. 2.3 FUNCTIONAL PARTITIONING (PIPELINE)

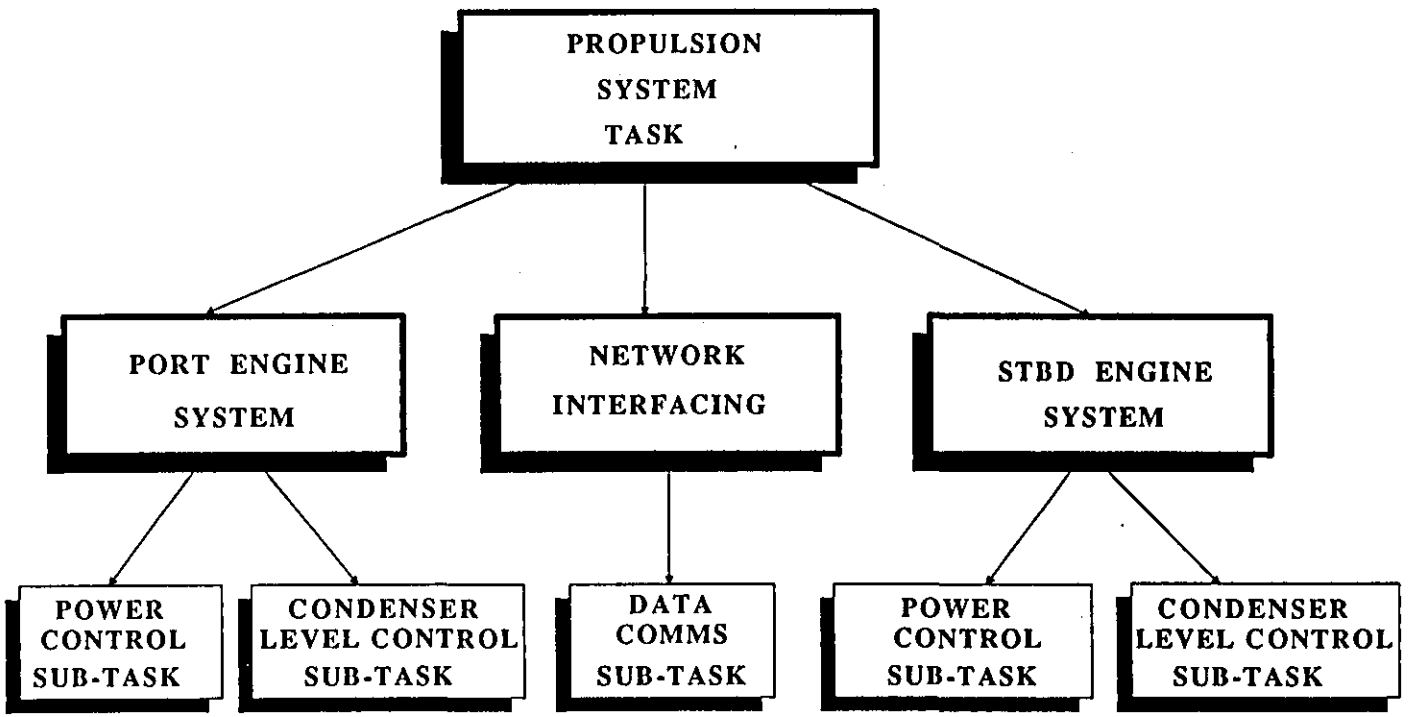


Fig. 2.4 FUNCTIONAL PARTITIONING

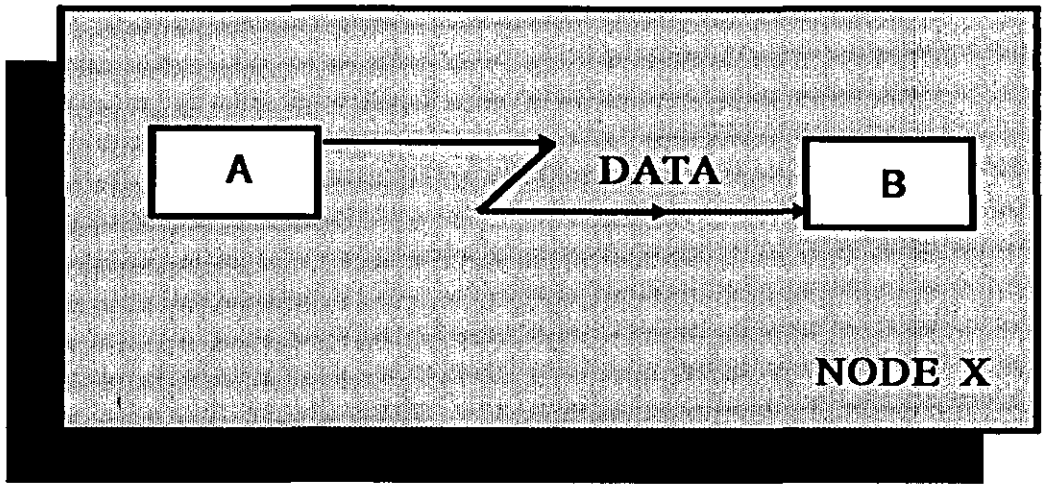


Fig. 2.5A E - MODE MESSAGE

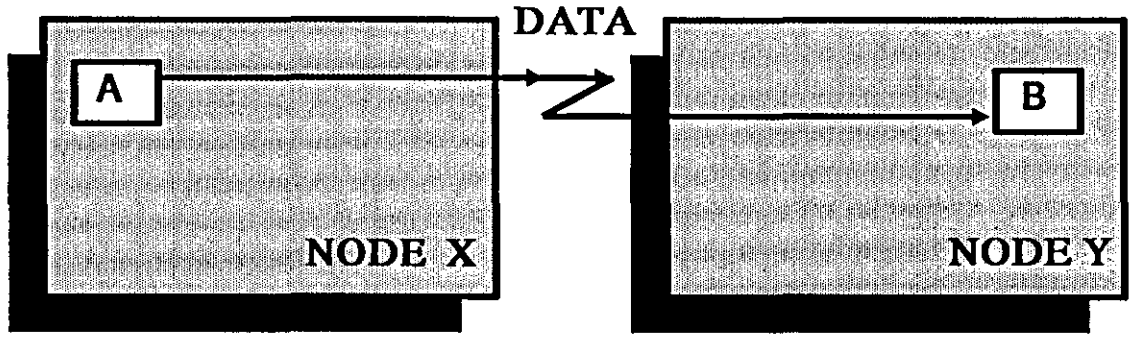


Fig. 2.5B T - MODE MESSAGE

CHAPTER 3

CHAPTER 3

FUNDAMENTAL ASPECTS OF DISTRIBUTED CONCURRENT PROGRAMS

3.1 CONCURRENT PROGRAMS (USE OF PROCESSES)

3.1.1 General

The nature of concurrent programming has changed substantially in the past ten years. First, theoretical research activities have prompted the definition of new programming notations that express concurrent computations simply and make synchronisation requirements explicit. Second, the advances in hardware technology, and hence the availability of inexpensive processors, have made possible the construction of distributed systems and multi-processors that were previously uneconomical.

Thus, implementations of concurrent programming are no longer limited to use in operating systems only. They are implemented in the design of database management systems, parallel scientific computations and real-time, embedded control systems.

3.1.2 Processes

A 'sequential program' specifies sequential execution of a list of statements; its execution is frequently called a 'process' [1].

A process may be in three main states (see Fig. 3.1):

i) Running: Instructions are being executed.
ii) Blocked: The process is waiting for some event to occur (such as input/output completion).
iii) Ready: The process is waiting to be assigned a processor.

A concurrent program, however, specifies two or more sequential programs which may be executed concurrently as 'parallel processes'. It can be executed by two methods:

i) Running more than one process on an individual processor. This is referred to as 'multi-tasking'. It has to be mentioned here, however, that 'quasi-concurrency' is the name referred to when processes share only one processor [2].

ii) Running each process on its own processor. This is referred to as 'multiprocessing' if processors share a common memory, or as 'distributed processing' if the processors are connected by a communications network [3]. A concurrent program that is executed in this latter way is often called 'a distributed program'.

3.1.3 Process Interaction

In order for concurrent processes to cooperate, they must communicate and possibly synchronise. 'Communication' is the transfer of data values from one process to another. Inter-process communication is based either on the use of 'shared variables' (variables referred by more than one process) or on 'message passing'.

'Synchronisation' is often necessary when processes communicate. Processes are executed with unpredictable speeds. Yet, to communicate, one process must perform some action that the other detects (an action such as setting the value of a variable or sending a message). This only works if the events 'perform an action' and 'detect an action' are constrained to happen in that order. Thus synchronisation can be viewed as a set of constraints on the ordering of events [3]. The

programmer employs a synchronisation mechanism to delay execution of a process in order to satisfy such constraints. Synchronisation can be more understood in an 'operational approach':

'Operational approach': Here the execution of a concurrent program can be viewed as a sequence of 'atomic actions', each resulting from the execution of an indivisible operation [4]. This sequence may comprise some interleaving of the sequences of atomic actions generated by the individual component processes. For example, suppose initially that x=0, that process P1 increments x by 1, and that process P2 increments x by 2:

P1: x:=x+1 P2: x:=x+2

It would seem reasonable to expect the final value of x, after P1 and P2 have executed concurrently, to be 3. Unfortunately, this will not always be the case, because assignment statements are not generally implemented as indivisible operations. So the above assignment may be implemented as a sequence of three indivisible operations:

* Load a register with the value of x,
* Add 1 or 2 to it,
* Store the result in x.

Thus in the program above the final value of x may be 1, 2, or 3.

This odd behaviour can be avoided by preventing interleaved execution of the two assignment statements, i.e. by controlling the ordering of the events corresponding to the atomic actions (if ordering were thus controlled, each assignment statement would be an indivisible

operation). In other words, execution of P1 and P2 must be synchronised by enforcing restrictions on possible interleavings.

3.2 SPECIFYING CONCURRENT EXECUTION

3.2.1 The Fork and Join Statements

The fork statement [5,6] specifies that once a designated routine starts executing, the invoking routine P1 and the invoked routine P2 proceed concurrently (Fig. 3.2). To synchronise with completion of the invoked routine, the invoking routine can execute a 'join' statement. Executing 'join' delays the invoking routine P1 until the designated invoked routine P2 has terminated (executing the 'end' statement). A use of 'fork' and 'join' is shown below:

```
Program P1;      Program P2;

  S1
  S2
  S3
fork P2;    SA
  S4        SB
  S5        SC
join P2;   end;
  S6
  S7
       ----------
```

P1 starts executing first statements S1, S2, and S3. Execution of P2 is initiated later when the 'fork' in P1 is executed; P1 and P2 then

execute concurrently (S4, and S5 in P1 and SA, SB, and SC in P2) until either P1 executes the 'join' statement or P2 terminates. After P1 reaches the 'join' and P2 terminates, P1 executes the statements following the 'join' i.e. S6, S7, etc.

The UNIX operating system makes extensive use of variants of 'fork' and 'join'.

3.2.2 The Cobegin Statement

The 'Cobegin' statement is a structured way of denoting concurrent execution of a set of statements (Fig. 3.3). This statement was first called 'Parbegin' [7]. Execution of:

Begin S1; Cobegin S2; S3; S4; S5; S6; S7 Coend; S8 end

Means that after the completion of S1, the statements S2, S3, S4, etc. (up to S7) will be executed concurrently, and only when are all executed will the execution of statement S8 be initiated [7]. Variants of 'Cobegin' have been included in Algol 68, Edison and Argus.

3.2.3 Coroutines

Coroutines are procedures that do not necessarily execute completely before returning control to their calling programs. A coroutine (Fig. 3.4) suspends itself and at some later point, via another call, resumes execution from the point at which it was suspended [8].

Each coroutine can be viewed as implementing a process, hence 'quasi-concurrent' programs may be implemented on a single processor using coroutines. In essence, coroutines are concurrent processes in which

process switching has been completely specified, rather than left to the implementation. Statements to implement coroutines have been included in Simula, Bliss and Modula-2.

3.3 INTRODUCTION TO SYNCHRONISATION TECHNIQUES

3.3.1 Critical Sections

Within a system, coordination of processes frequently involves access to shared data areas. Program segments that access shared data are the most hazardous to implement and are referred to as 'Critical Sections'. The safest general solution for sharing data is to adopt a policy for 'mutual exclusion' where access is restricted to one process at a time [7,9]. This policy is over-restrictive when a number of processes wish only to read data, but should be enforced if data is to be updated [10].

3.3.2 Semaphores

Early attempts to produce concurrent programs were based on semaphores [7], low level primitives from which mutual exclusion and synchronisation protocols could be constructed.

A semaphore(s) is a non-negative integer-valued variable on which two operations are defined: 'P' and 'V'. P(s) (also called wait(s)): IF s>0 THEN s:=s-1 ELSE the execution of the process that called P(s) is suspended.

V(s) (also called signal(s)): IF some process (Q) has been suspended by a previous P(s) on this semaphore (s) THEN wake-up (Q) ELSE s:=s+1

Test and decrement in P(s), increment in V(s) are done as indivisible (atomic) operations (Fig. 3.5).

A semaphore assuming values 0 and 1 only is called a binary semaphore. A semaphore which can take an arbitrary non-negative integer values is called a general semaphore [3,7,11].

Most semaphore implementations are assumed to exhibit 'fairness'. This is needed when a number of processes are delayed, all attempting to execute a 'P' operation on the same semaphore. A simple way to ensure fairness is to awaken processes in the order in which they are delayed.

A solution to the two process mutual exclusion problem in terms of semaphores is shown below:

```
PROGRAM Mutex-Example;
VAR mutex: semaphore initial(1);

 PROCESS P1;
  loop
   p(mutex); (*Entry Protocol*)
   Critical section;
   V(mutex); (*Exit Protocol*)
   Noncritical Section
  end
 end;
```

```
PROCESS P2;
 loop
  P(mutex); (*Entry Protocol*)
  Critical section;
  V(mutex); (*Exit Protocol*)
  Non-Critical Section
  end
 end;
end.
```

3.3.3 Synchronisation Techniques and Language Classes

A number of programming methodologies and languages have been developed to provide structured multiprocessed system (Fig. 3.6). These started with the definition of semaphores, then were extended in three ways:

* Constructs were defined that enforced their structured use, resulting in critical regions, and monitors.
* 'Data' were added to the synchronisation associated with semaphores, resulting in message-passing primitives.
* Finally, the procedural interface of monitors was combined with message-passing, resulting in 'remote procedure call'.

Although there are a variety of different synchronisation techniques, there are only three essentially different kinds: procedure oriented, message oriented, and operation oriented [3,12]. These three approaches are now considered in more detail.

3.4 PROCEDURE-ORIENTED SYNCHRONISATION METHOD

3.4.1 Monitors

The monitor concept, developed over a number of years [7,9,13], is one approach towards ensuring a reliable concurrent programming environment. Although the processes constituting a concurrent program may declare individual data areas, a frequent occurrence is the declaration of a common data area to be accessed by several processes. If the processes execute asynchronously, it is possible that more than one process will attempt to access this shared data area simultaneously, with unpredictable results. One such manager, known as a monitor, encapsulates the shared data area and the procedures that will act on this area (Fig. 3.7). Hence, the monitor will provide mutual exclusion of processes to a set of procedures that act on the shared data, and consequently ensure the integrity of that data.

The concept of the monitor was first implemented in the language Concurrent Pascal [14] and later in Modula [15]. These languages define two forms of components: processes and monitors. Processes are the active program elements which operate on monitors, which are the passive components containing shared data.

A monitor is a program module which encapsulates the definition of some data variables with procedures for their access. It is written as a set of global variables declarations followed by a set of procedures. The monitor has a body (begin----end) which is a sequence of statements executed immediately when the program is initiated. Henceforth, the monitor exists only as a module (data and procedures). Processes accessing the monitor only need to know which procedures are provided, the particular implementation details being confined to the monitor definition.

A typical monitor concept is shown in Fig. 3.8. A producer process in a program may insert items into the buffer by calling the monitor's procedure 'produce'. The items may later be extracted by another process calling the monitor's procedure 'consume'.

A process calling a monitor's procedure gains exclusive use of the monitor until it exist from the monitor. If a second process attempts to enter a monitor that is currently in use by another process, the second process is delayed until the first process releases the monitor.

To impose or synchronise the operation within a monitor most monitors define a type of variable called a 'condition variable'. This is used to delay processes executing in a monitor. For instance, it may be used to prevent a process from attempting to extract data from the buffer before any has been inserted. It may be declared only within a monitor.

If (c) is a condition variable then there are two operations that can be applied to (c):

i) Wait(c): The calling process is blocked and is entered on a queue of processes blocked on this condition, i.e have also executed Wait(c) operations. Unlike semaphores we assume that the queues are First In First Out (FIFO).

ii) Signal(c): If the queue for c is not empty then wake up the first process on the queue, otherwise continue (i.e invoker does not return from its monitor call).

Another approach to condition synchronisation has been implemented in concurrent Pascal [14], using a slightly simpler mechanism. Variables of type 'queue' can be defined and manipulated with the operations 'delay' (analogous to 'wait') and 'continue' (analogous to 'signal'). In contrast to condition variables, at most one process can be suspended on a given 'queue' at any time.

A number of other constructs for determining when a process should delay or continue have been proposed [16,17]. In general these require that a process accesses a monitor to evaluate an expression to determine when to continue its operation, rather than waiting for an explicit signal from another process.

Clearly the monitor is more complex than the semaphore. In practice it would normally be implemented using a number of procedures, as follows [18];

```
PROCEDURE InitialiseMonitor(VAR SharedResource :Monitor);
(* This allocates memory for the monitor *)

PROCEDURE InitialiseMonitorSignal(VAR Condition :MonitorSignal);
(* This initialises a monitor signal *)

PROCEDURE GainControl(VAR SharedResource :Monitor);
(* This allows a task to gain control of the monitor *)

PROCEDURE ReleaseControl (VAR SharedResource :Monitor);
(* This defines that a task has finished with the shared resource *)

PROCEDURE WaitInMonitor (VAR SharedResource :Monitor;
 VAR Condition :MonitorSignal);
(* This controls operation of the condition and priority queues *)
```

3.4.2 Nested Monitor Calls

Acquisition and release of exclusion leads to a problem when monitor calls are nested. For instance, suppose that a procedure Proc1 of a monitor Mon1 calls procedure Proc2 of monitor Mon2. If Proc2 contains a 'Wait' operation should mutual exclusion be released on both Mon1 and Mon2, or Mon2 alone ?

Such nested monitor calls have caused much discussion [19,20,21,22]. There are, though, a number of ways to handle this problem:

i) Prohibit nested monitor calls completely as implemented in SIMONE [23], or prohibit nested calls to monitors that are not lexically nested, as implemented in Modula [24].

ii) Release the mutual exclusion on all monitors along the call chain when a nested call is made and that process becomes blocked.

iii) Define a monitor-like construct that allows the programmer to specify that certain monitor procedures be executed concurrently and that mutual exclusion be released for certain calls [25].

3.5 MESSAGE-PASSING SYNCHRONISATION PRIMITIVES

3.5.1 General

Message passing may be viewed as extending semaphores to convey data as well as to implement synchronisation. When message passing is used for communication and synchronisation, processes send and receive messages instead of reading and writing shared variables. Communication is accomplished because a process, upon receiving a message, obtains values from some sender process. Synchronisation is accomplished because a message can be received only after it has been sent. Two main issues must be discussed: specifying channels for communication, and message synchronisation. Both are discussed in the following sections.

3.5.2 Specifying Channels For Communication

A message is sent by executing:

'SEND' expression_list
'TO' destination_designator.

The message contains the values of the expression in 'expression_list' at the time 'SEND' is executed. The 'destination_designator' gives the programmer control over where the message goes. A message is received by executing:

'RECEIVE' variable_list
'FROM' source_designator.

Where 'variable_list' is a list of variables. The 'source_designator' gives control over where the message came from. Receipt of a message

causes, first, assignment of the values in the message to the variables in the 'variable_list' and, second, subsequent destruction of the message.

Destination and source designators define together what is called a 'communication channel'. Various schemes have been proposed for naming channels. The simplest channel-naming scheme is for process names to serve as source and destination designator. We refer to this type as 'direct naming'. Thus:

'SEND' value 'TO' consumer

sends a message that can be received only by the 'consumer' process. Similarly,

'RECEIVE' value 'FROM' consumer

permits receipt only of a message sent by the 'consumer' process.

Direct naming uses a one-to-one communication scheme. It makes it possible for a process to control the times at which it receives messages from each other process.

Two processes communicating through message-passing could have the following form:

```
Process Producer;
   VAR: Declarations of variables;
   begin
    loop
```

```
      code to implement 'producer';
      'SEND' value 'TO' consumer
   end
end;

Process Consumer;
   VAR: Declarations of variables
   begin
    loop
      code to implement 'consumer';
      'RECEIVE' value 'FROM" producer
    end
   end;
```

An important pattern for process interaction is the 'client/server' relationship. 'Server' processes render services to 'client' processes. A client can request that a service be performed by sending a message to one of these servers.

Unfortunately, direct naming is not always suited for client/server interaction since more than one 'receive' has to be required for different clients, i.e the relation is MANY clients to ONE server (N-to-ONE).

A more sophisticated scheme for defining communication channels is based on the use of 'global names' sometimes called 'mailboxes'.

A mailbox can appear as the destination designator in any process 'send' statements and as the source designator in any process 'receive' statements. Thus messages sent to a given mailbox can be

received by any process that executes a 'receive' naming that mailbox. This method, therefore, uses an N-to-N communication scheme.

A mailbox is well suited for programming client/server interaction. Clients send their service request to a single mailbox; servers receive service requests from that mailbox. Unfortunately, implementing mailboxes can be quite difficult.

A special case of mailboxes occurs when a mailbox name appears as the source designator in 'receive' statements in one process only. This is called a 'Port' [26]. This is an N-to-one communication scheme. Ports are simple to implement, since all 'receives' that designate a port occur in the same process.

Finally source and destination designators can be fixed at compile time (called static channel naming), or they can be computed at run time (called dynamic channel naming). Static channels are widely implemented.

3.5.3 Synchronisation

General

Communication aspects may be divided mainly into 'Synchronous communication' and 'Asynchronous communication' [3,12,27].

a) Synchronous Communication

Synchronous communication is best understood from the perspective of the sending process. When the sending process, the synchronous sender, transmits a message to a receiving process, it waits until the receiving process responds with an acknowledgement that the message has been received (in the case of a synchronous receiver) or until the

receiving process explicitly returns from performing its task (in the case of a remote procedure call).

b) Asynchronous Communication

In simple terms, asynchronous communication is message exchange without acknowledgement, i.e no-wait send [28]. After sending a message, the sending process continues executing; it does not wait for the receiving process to respond. Furthermore the receiving process does not issue an acceptance. Because message exchange is not synchronised, communication requires buffering for messages that have been sent but not received. This buffering capability may be provided by the interconnection network or by specially designed receiver software.

Asynchronous communication provides a high degree of concurrency since the sender need not wait for the message to reach the receiver or for message acknowledgement. It also reduces message traffic. The drawback of the method is the need to provide message buffering facilities.

Distributed programming languages do not normally directly support both forms of communication; system designers prefer to minimise the required language features. An exception is the language SR [29], which provides mechanisms for both synchronous communication and asynchronous communication [27].

3.6 'OPERATION-ORIENTED' SYNCHRONISATION METHODS

3.6.1 General

To programme client/server processes that reside in different processors, higher level message constructs have to be used. Message passing primitives may be utilised to build such higher level message constructs in distributed programs [3,12]. Consider, for instance, where a client needs to 'call' a procedure for execution on a remote processor. This is done by interacting with the server processes using message communication techniques (SEND of message followed by RECEIVE of results). At the remote site the 'call' message is received by a server process (using RECEIVE), interpreted (procedure execution), and the results sent back (using SEND) to the calling client process [30].

3.6.2 The Remote Procedure Call (RPC)

When remote procedure calls are used, a client interacts with a server by means of a call statement. This statement has a form similar to that used for a procedure call in a sequential language:

CALL 'Service' (value-arguments; results-arguments) where 'Service' is the name of a channel.

A remote call is executed as follows: the value arguments are sent to the appropriate server (CALL message, Fig. 3.9), and the calling process delays until both the service has been performed and the results have been returned and assigned to the result arguments (Result message, Fig. 3.9). Thus such a 'call' could be interpreted or seen as a SEND immediately followed by a RECEIVE. The client waits for the results of the requested service.

The SERVER side of a remote procedure call could be specified as a declaration (like a procedure in a sequential language), this is shown as follows:

```
'Remote Procedure' Service
('IN' value-parameters;
 'OUT' results-parameters)
    ---------
    statements
    ---------
 END Service;
```

Note that the procedure arguments are optional. Variables can be declared as being for input (IN) or output (OUT). Such a procedure declaration is implemented as a process. This process, the server, awaits receipt of a message from some calling process, executes its body, and then returns a 'reply message' containing the values of the results parameters. A remote procedure declaration might be implemented as a single process in which case 'calls' to the same remote procedure would execute sequentially [29]. Alternatively, a new process can be created for each execution of 'call' [31,32,33]; these could execute concurrently and implement mutual exclusion where necessary.

3.6.3 Rendezvous

A rendezvous [34] is a technique for enforcing synchronisation and message communication between two tasks. Exactly two tasks may rendezvous at once; a client (here called a caller) and a server. The caller calls an entry (the name of the rendezvous) in the server. The

server, when it is ready to do so, issues an ACCEPT statement to receive the call (Fig. 3.10). If the caller calls an entry for which the server has not as yet issued an ACCEPT, then the caller waits until the ACCEPT is issued. If the server issues an ACCEPT for an entry that the caller has not as yet called, then the server waits (at the ACCEPT) for the caller to call the entry.

When a call has been accepted, the rendezvous occurs. The caller passes data to the server through parameters in the entry call. The data are processed by the statements within the ACCEPT statement body. Results, if any, are passed back to the caller through the entry parameters.

The caller waits while the server executes within the ACCEPT statement. When this processing is complete, parameters are passed back to the caller, the rendezvous ends, and the caller and server tasks resume independent operation.

One interesting aspect of the rendezvous is that the caller must know of the existence of the server and the various server entries. But the server accepts calls from any caller. Many callers may attempt to call one server. In this sense, the rendezvous is asymmetric (as in the case of Ada [34]).

Mutual exclusion is guaranteed by underlying system mechanisms; only one caller at a time may rendezvous with the server. Other callers attempting a simultaneous rendezvous are kept waiting. Synchronisation of the tasks is implicit during the rendezvous. After a rendezvous, any waiting callers are processed first-come-first-served. The ACCEPT construct in the server side takes the following form:

```
ACCEPT  Service ( 'IN' value-parameters;
  'OUT' result-parameters )
  -------
  statements
  -------
END  Service;
```

Note again that the ACCEPT arguments could be declared as input 'IN', or output 'OUT'. The statements within the ACCEPT...END are assumed to be a critical section and are executed in a mutually exclusive manner. They would normally be executed by the server process (here called task). The ACCEPT statement represents an entry point (the name of the rendezvous) and the calling task specifies the name of the entry point when it wishes to synchronise with the server task.

```
TASK A;
  VAR X:ADataItem;
     BEGIN
       --------
       B.Transfer(X);
       --------
     END;

TASK B;
  VAR Y:ADataItem;
     BEGIN
       --------
ACCEPT Transfer ('IN' item:ADataitem);
  Y:=item;
END;
     END;
```

In the above example task A wishes to pass information held in variable X to a variable Y in task B. The actual data transfer takes place using the normal parameter passing mechanisms: the actual parameters supplied in the call, in this case the variable X, are bound to the formal parameters of the ACCEPT statement, in this case 'item'. The synchronisation of the two tasks is obtained by the requirement that the procedure call entry, B.Transfer (X), cannot be completed until the corresponding 'ACCEPT Transfer' is executed. Conversely, the execution of the ACCEPT statement cannot be completed until the entry call is executed. The actual transfer is completed within the body of the ACCEPT statement; in this case the data supplied by the entry call is transferred to a variable which is local to task B.

Guarded command communication

It is possible to have a 'SELECTive-communication' form in the receiver side of a message, i.e the server side [27]. In a 'selective-communication' statement, a 'guarded command' has the form [35]:

```
SELECT
    WHEN condition 1 -----> ACCEPT entry 1 DO statements END;
 other statements
OR
    WHEN condition 2 ----> ACCEPT entry 2 Do statements END;
 other statements
     --------
ELSE statements
END SELECT;
```

The SELECT statement allows one to select between several alternatives separated by OR. The alternatives are prefixed by WHEN clauses called 'guards'. The guards are boolean expressions which establish what conditions must be true for an alternative to be a candidate for execution. If there are open alternatives (conditions true) then an ACCEPT statement is chosen for execution, possibly with a process currently waiting for a rendezvous. If, however, there are several open alternatives with processes waiting for rendezvous, the selection among them is done arbitrarily. The ELSE clause is executed in the case of no open alternatives or no waiting processes.

The difference between the SELECT statement and an IF statement is seen in the case that both guards are open (conditions true). Then if both tasks (e.g. a consumer and a producer) are waiting for a rendezvous, it is immaterial which rendezvous is executed. An IF statement, however, must specify which statement is to be executed in this case.

This is the essence of the 'guarded commands' style of programming. It avoids over-specification (as in an IF statement) by allowing the computer as much freedom of choice as possible consistent with the correctness requirements of the program.

3.6.4 Messages in Distributed Systems

Two types of messages can be implemented in a distributed system; E-mode messages and T-mode messages [36] (Fig. 3.11):

i) E-mode messages refer to message transactions between various modules (data and procedures) of a user program confined to the same processor (intra-processor communication). These normally take the usual message form, as described earlier:

'SEND' data 'TO' module B

'RECEIVE' data 'FROM' module A

where modules A and B reside in the same processor.

ii) T-mode messages refer to messages exchanged between the operating systems of two different nodes (inter-processor communication). These might take the form:

'SEND' (data, module B at node Y)

'RECEIVE' (data, module A at node X)

where module A at node X, is sending 'data' to module B at node Y.

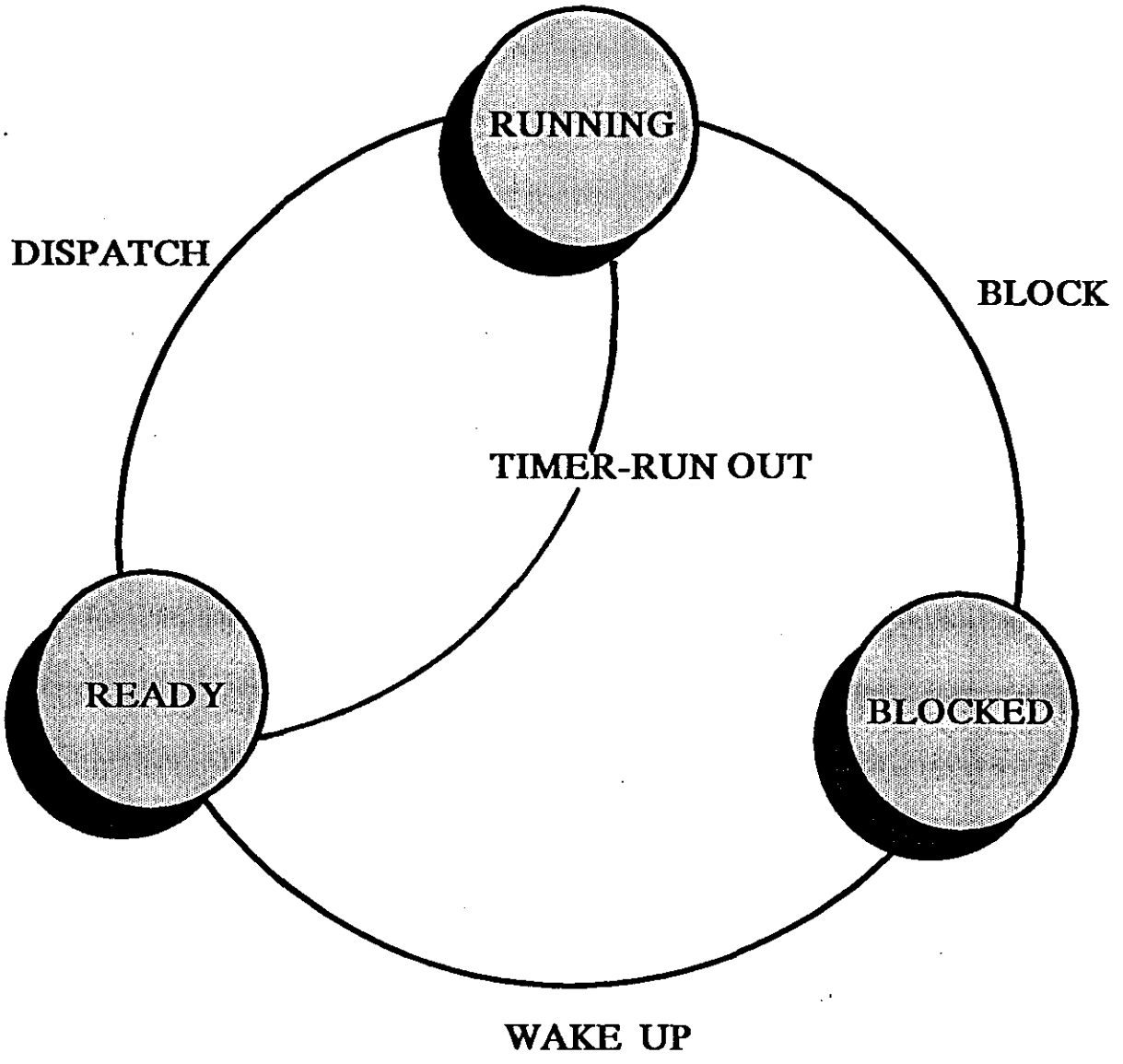


Fig. 3.1 PROCESS STATE TRANSITIONS

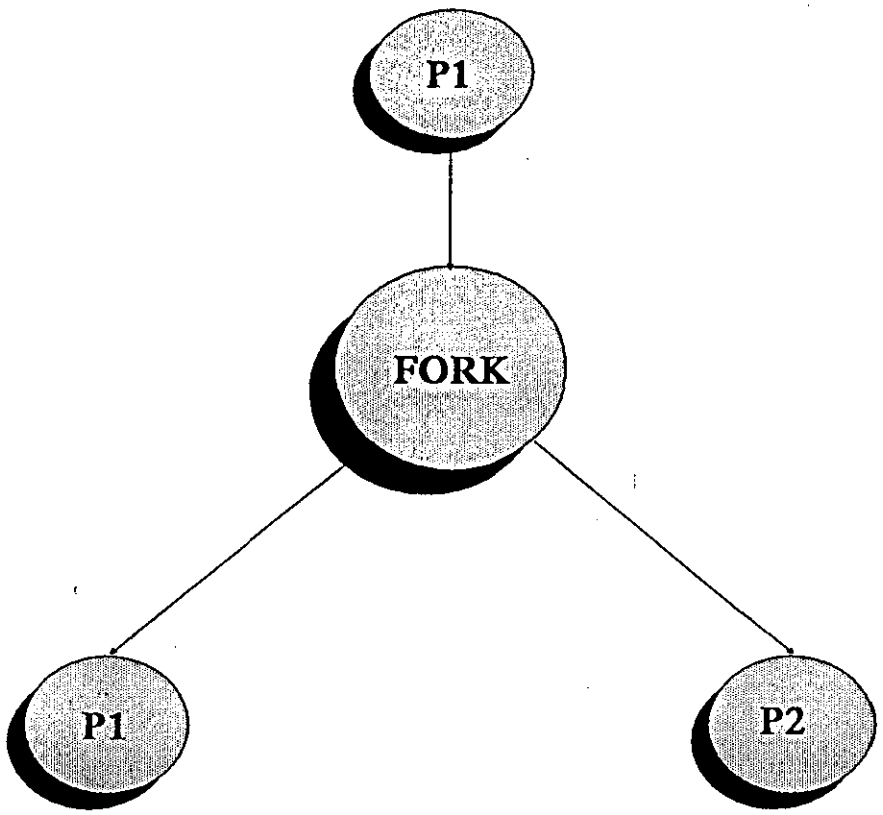
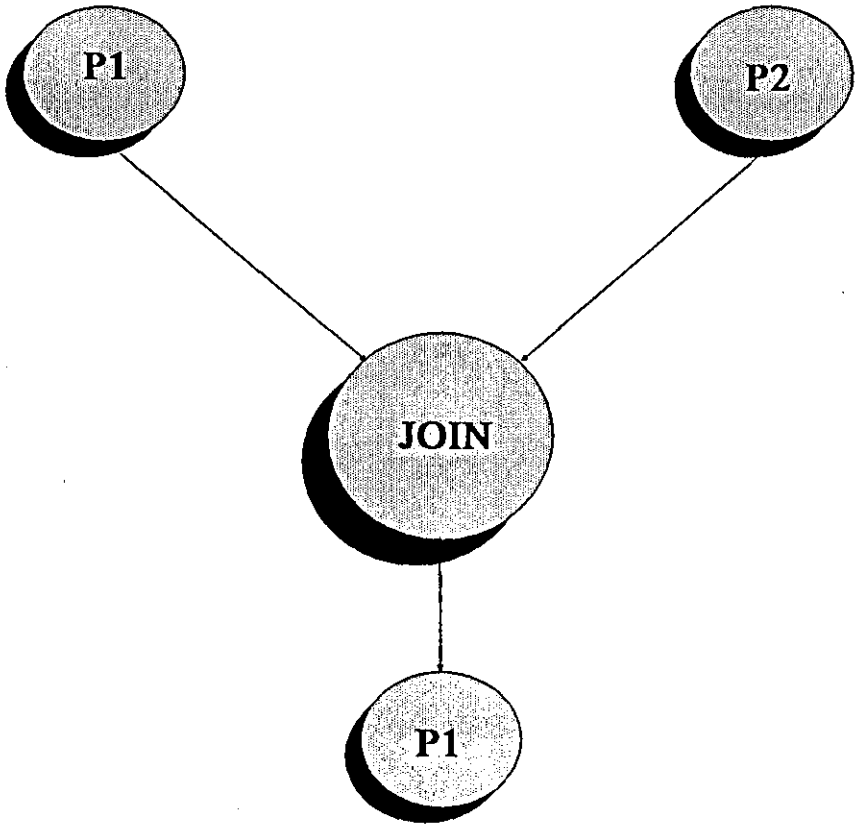


Fig. 3.2 THE 'FORK' and 'JOIN' STATEMENTS



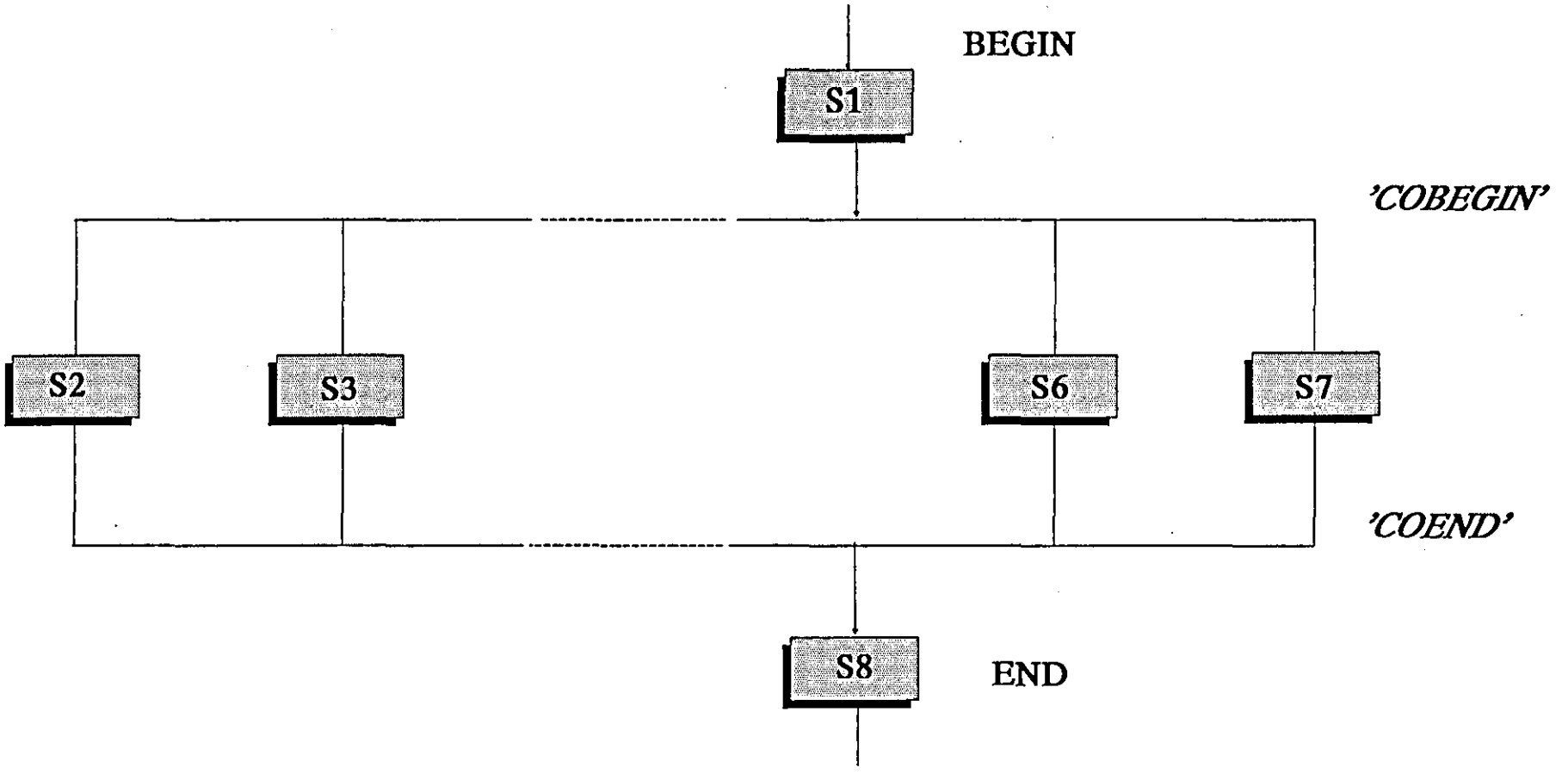


Fig. 3.3 THE 'COBEGIN' STATEMENT

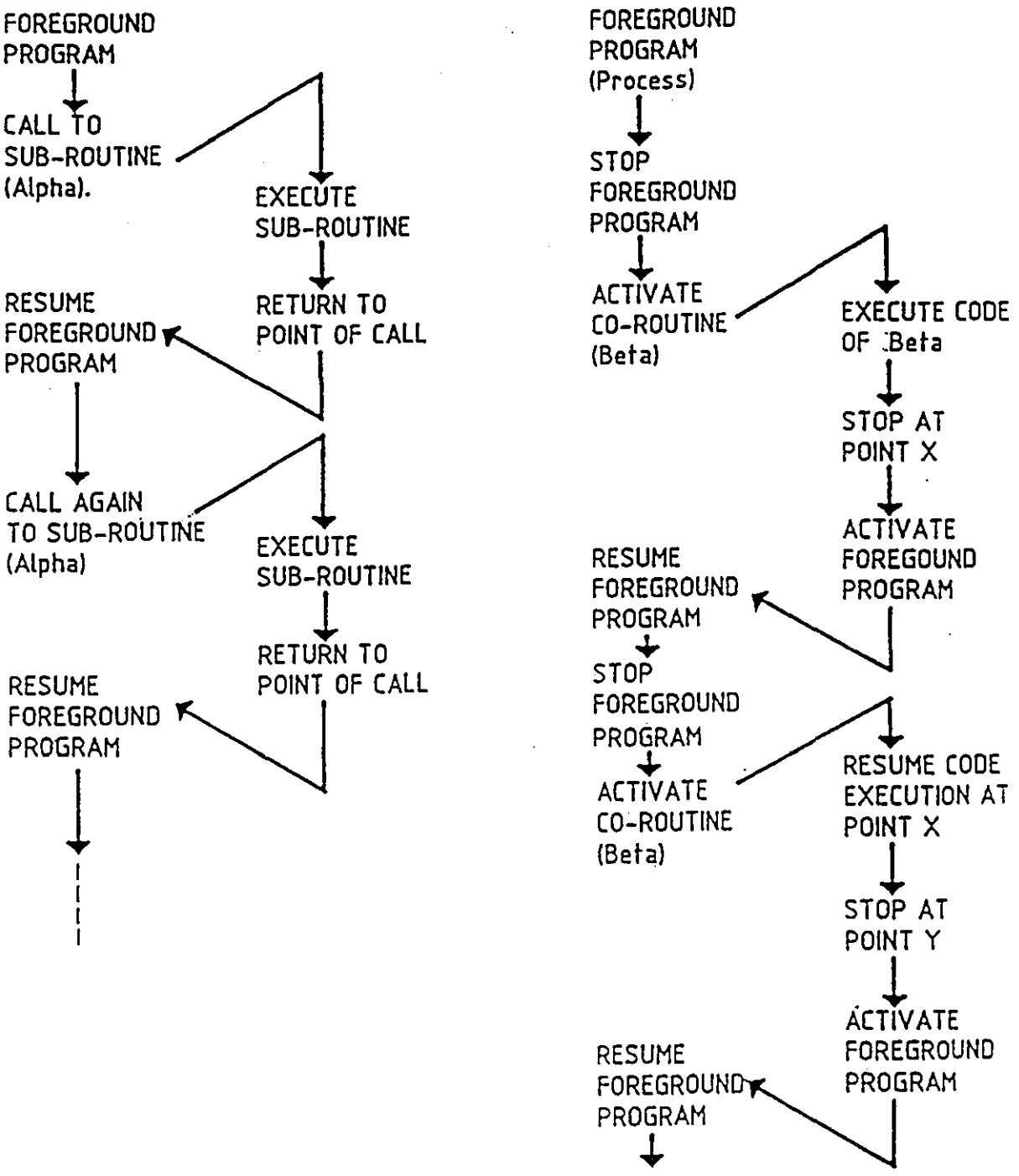


Fig. 3.4 SUBROUTINES v's COROUTINES - CONCEPTUAL DIFFERENCES

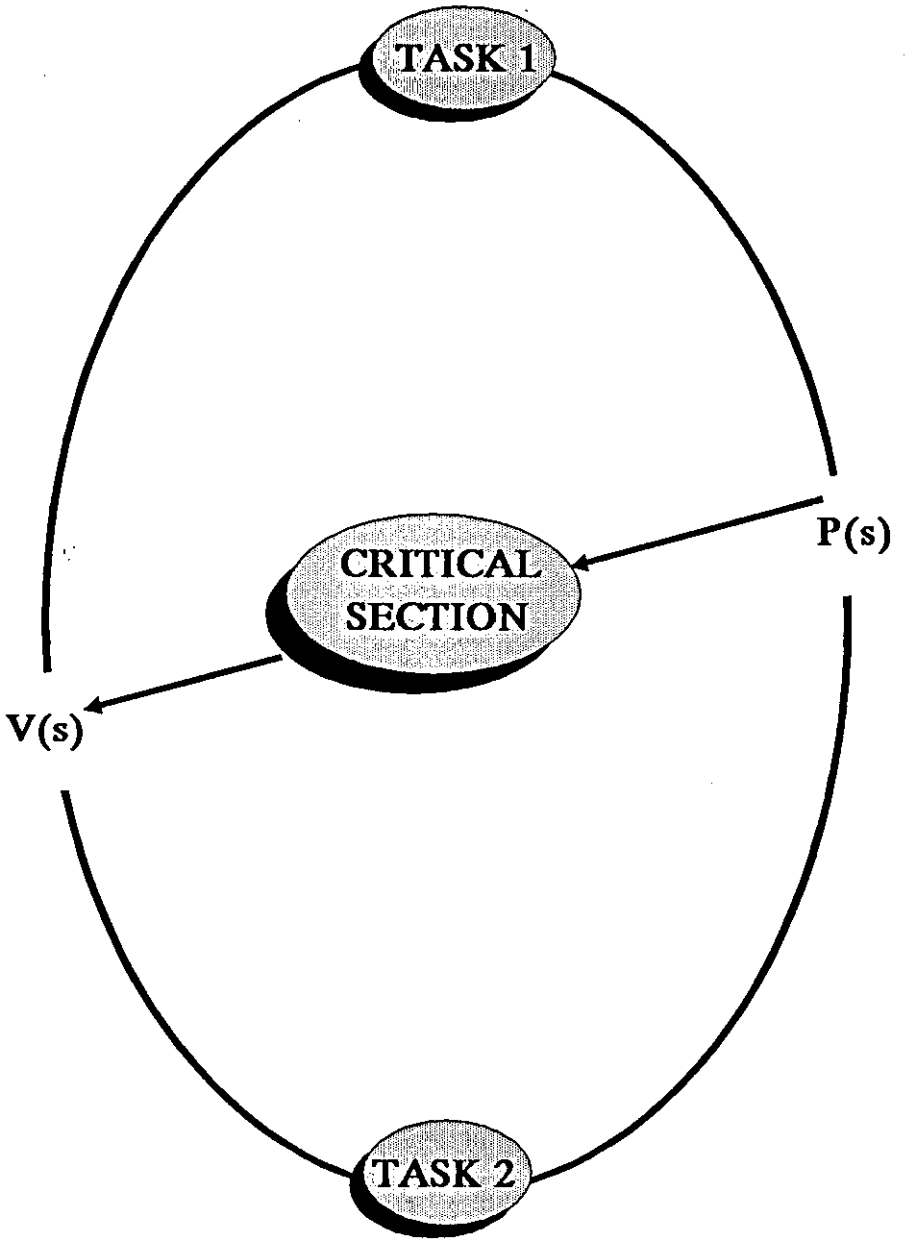


Fig. 3.5 TASK COMMUNICATION WITH 'SEMAPHORES'

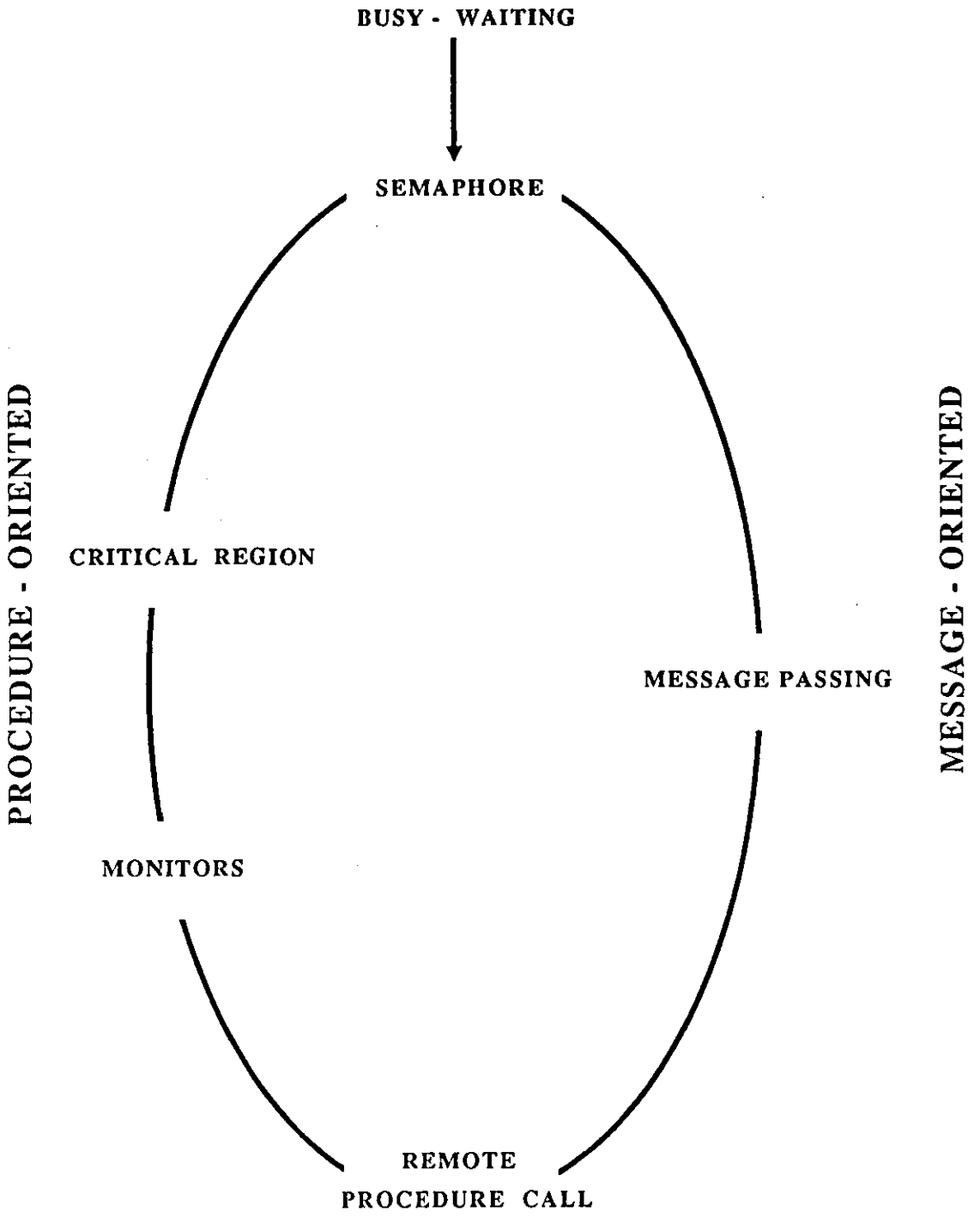


Fig. 3.6 SOFTWARE METHODOLOGIES

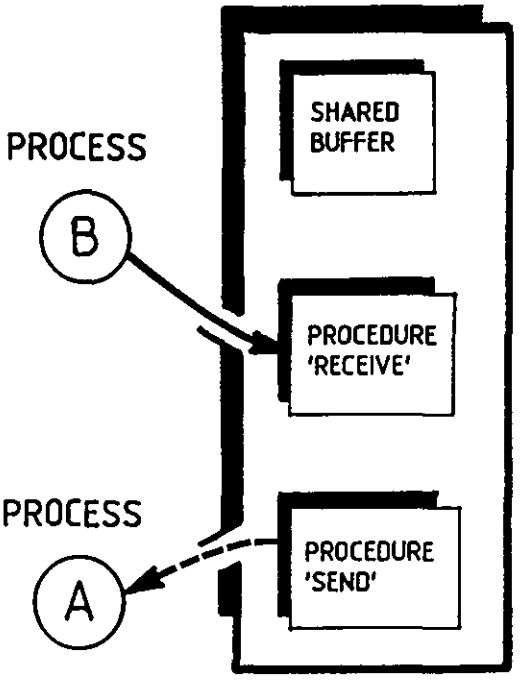


Fig. 3.7 MONITOR STRUCTURE

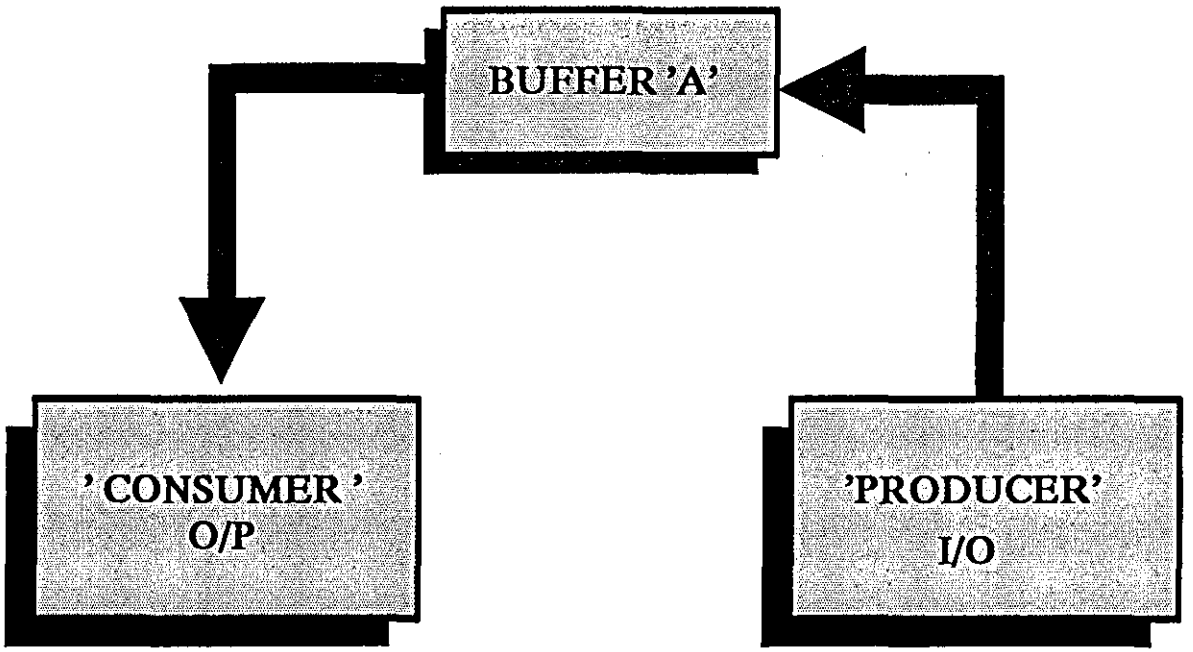


Fig. 3.8 THE 'MONITOR' CONCEPT

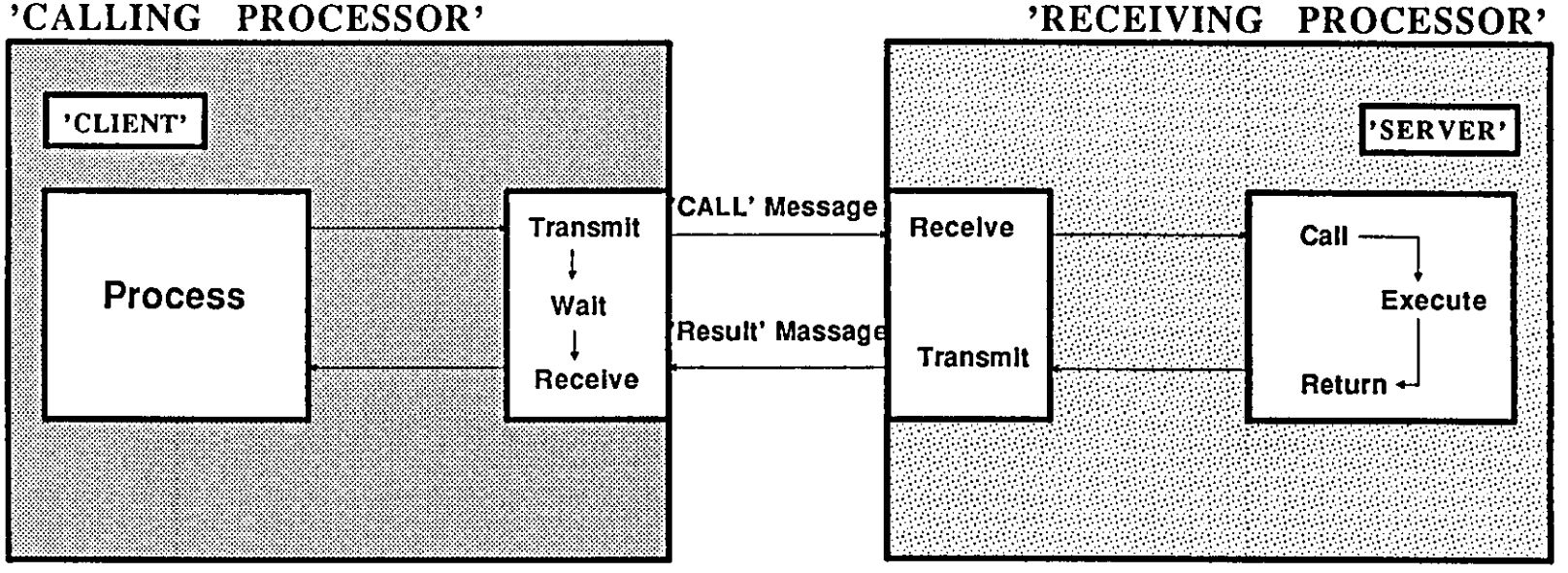


Fig. 3.9 REMOTE PROCEDURE CALL (RPC) - IMPLEMENTATION

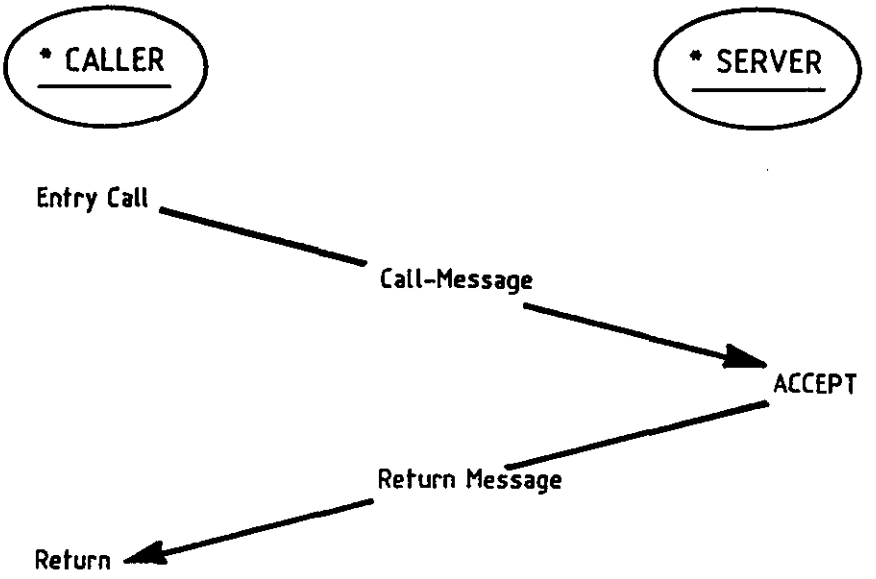


Fig. 3.10 RENDEZVOUS TRANSACTIONS

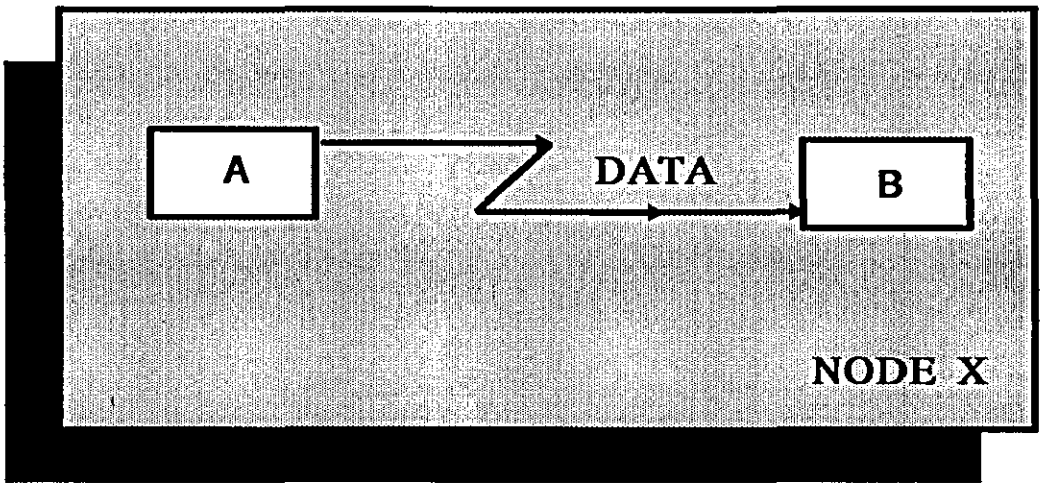


Fig. 3.11A E - MODE MESSAGE

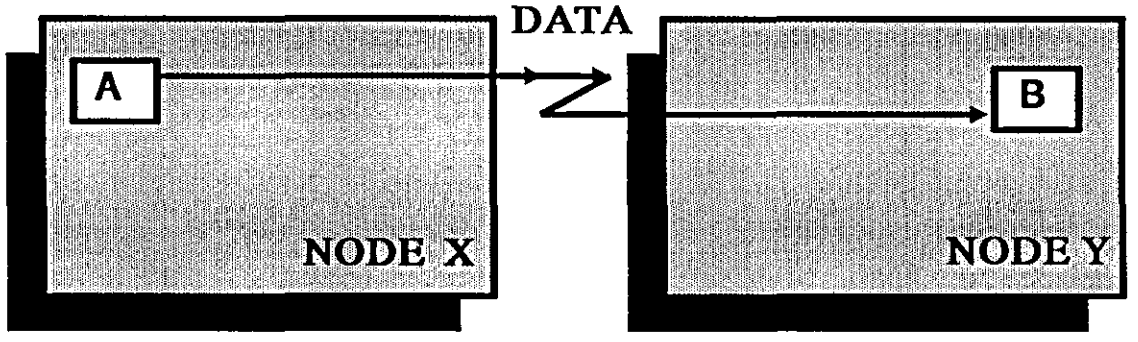


Fig. 3.11B T - MODE MESSAGE

CHAPTER 4

CHAPTER 4

A MULTI-PROCESSOR STRUCTURE TO SUPPORT FUNCTIONAL PARTITIONING

4.1 SYSTEM OVERVIEW

Real-time, multi-processor, embedded systems are one application area where response times, throughput, reliability and fault-tolerance constitute the major design criteria [1]. Hence the distribution and management of the application software is a critical function.

A prototype loosely-coupled multi-processor system has been designed and implemented for use in fault-tolerant real-time applications. This chapter discusses the organisation and structure of the total system, highlighting in particular the software requirements of the communication and executive (real-time kernel) functions. The communication system is based on a token passing bus protocol for use with single board computers connected via a fast parallel bus. The real-time kernel is designed to support functional partitioning of application programs, and can be implemented using standard compilers. No special multiprocessing features are required. Most of the software for this system has been written in the structured, high level, language Modula-2 though assembly language programming has been used in a few specialised areas.

4.2 FUNCTIONAL DESCRIPTION

A prototype loosely-coupled multi-processor system has been designed and implemented for use in fault-tolerant real-time applications [2,3]. It is composed of stations (nodes) linked together through a communication bus (Fig. 4.1). Each node consists of two sections:

* The main processor (or 'processing') section. This holds the application programs and the functional kernel.
* The communication section. This provides the interface between the processing section and the communication bus.

a) Processing Section

In the multi-processor concept described here, no assumptions are made about the structure of the main processing block. However, to support a distributed computing system, each processing section must have its own memory and I/O devices (Fig. 4.2). All application software and supporting programs run within this area. It is isolated from the system (backplane) bus, having nothing to do with communication control. Each processing section merely interchanges information with its own communication section, bus access being a transparent function.

b) Communication Section

The main function of the communication section is to handle all communications activities within each station. It isolates the processing section from the system bus, providing a transparent interface for message transaction within the system. Thus it removes a considerable burden, both in terms of software and time, from the main processor. The communication section has five modes of operation:

* Bus transmission: transfer data to the system bus.
* Bus reception: receive data from the system bus.
* Internal transmission: transfer data to the processing section.
* Internal reception: receive data from the processing section.
* Idle: No processing, ready to respond to data transfer requests.

Fig. 4.2 describes the communication section in functional block diagram form. It consists of a number of subsystems among them are; transmission/reception control logic, a temporary storage RAM, and a number of data holding buffers.

4.3 INTER-PROCESSOR COMMUNICATION

Communication between processing sections is performed using message passing techniques based on token passing. Basically the method allows a series of bus connected units ('stations') to communicate as a ring structure. Such a situation is shown in Fig. 4.3, where a number of stations, each one having a unique address, are coupled to a shared bus. The right to use the bus is transferred from station to station, thus forming a logical ring. When a station has this right it is said to hold the 'token'. At any given time one station, and only one station, holds the token, and is obligated to pass it on when finished with it. Each station can hold the token only for a limited period of time. This means that the maximum time taken by the token to traverse the network is defined, i.e., access to the system bus is deterministic.

For the system to function correctly each station must be in possession of three addresses: the preceding station (previous station - PS), the succeeding station (next station - NS) and its own (this station -TS). Station numbers do not need to be contiguous. This feature simplifies the tasks of adding and removing stations without re-arranging established addresses.

There is a substantial software complexity in the token bus system, particularly with regard to ring configuration and maintenance. To acquire station address information, a rigorous configuration process is required. Once the ring is formed it has to be maintained. Facilities are needed to allow new stations to enter the ring and to cater for station drop-out. Drop-out (station exit) can occur for two reasons: either as part of normal operations, or as a result of a failure. In either case, the ring must be reconfigured to accommodate the changes.

The token passing method has three major features [4]. It is:

* A fair access system. The method is fair; it offers each station an equal share of the bus.
* Reconfigurable: The method handles addition and deletion of stations easily, without any modification of the existing hardware or communication software (the protocol).
* Deterministic: The method provides computable, deterministic, worst case bounds on access delay for any given network. This feature is essential in real-time systems where system response time must be guaranteed.

4.4 OPERATING SYSTEM SUPPORT-THE DISTRIBUTED PROGRAM KERNEL

4.4.1 General

Generally speaking, operating systems for multi-processor networks can be classified as either network or distributed operating systems. In a network operating system each computer or station has its own private operating system. The different private operating systems are then augmented with communication facilities to permit interaction and communication with the other systems in the network [5]. Network operating systems are commonly used to connect spatially or geographically dispersed systems. The ARPANET [6] is an example of a network operating system.

A distributed operating system, however, is one that looks to its users like an ordinary centralised operating system but runs on a multiple processor system. The key concept here is transparency. In other words, the use of multiple processors for the implementation of the operating system should be transparent to the user [7]. These operating systems are suitable for loosely or tightly coupled multi-processor networks. Examples of distributed operating systems are MOS [5], and Medusa [8].

In this section, however, we propose another category of operating systems-kernels that support multi-processor, real-time systems, a 'Distributed-Program kernel' [2,3].

4.4.2 Distributed-Program Kernel

In a multi-processor environment an application task, such as process control or robotic application, is partitioned into a set of co-operating sub-tasks (processes). Each node of the system may be

allocated a single sub-task. In some cases a number of sub-tasks may be assigned to one specific node. Processes residing on different processors execute in a true concurrent fashion; processes allocated to the same processor, however, execute in a quasi-concurrent mode.

The software design and support of a such a functional distribution of sub-tasks (processes) depends on the degree of interaction of these processes among the different nodes. Distributed processes, however, have to communicate and interact occasionally in order to achieve a common goal [9,10]. In real-time applications this interaction has to take place within quite specific timescales otherwise unsatisfactory results might take place.

Management and interaction of distributed processes is achieved by a kernel; it is usually termed a 'Distributed-Program Kernel'.

Unlike many scientific and commercial applications, the kernel described here is not intended to support fragmented programs. Instead, the basis of the design is that of functional partitioning. Further, a major primary objective is to implement the kernel using standard compilers, i.e those designed for uni-processor systems. A second major objective is to build the kernel infrastructure using the standard constructs of Modula-2.

In the design of the kernel we are very much concerned with predictability of performance. Moreover, reliability of operation is paramount [11]. The kernel is structured as a set of primitives, replicated, if necessary, on various nodes. This provides a virtual machine in which processes allocated to different processors are executed concurrently. These processes cooperate and synchronise

themselves by means of message-passing. On the other hand, the system inside each node is viewed as a collection of cooperating sequential processes that share common data. Processes in each node synchronise and communicate through message-passing constructs.

4.5 PROGRAMMING LANGUAGE ISSUES - MODULA-2

4.5.1 General

The choice of a 'good programming' language plays a major role in the design requirements of real-time computations, operating systems, etc..[12]. In fact some of the primitives that are essential to the design of such systems are implicitly found as built-in constructs within high level, structured, languages such as Modula-2, Ada, and C. Other facilities, however, still have to be implemented when needed (e.g., generics , message-passing constructs, exception handling, etc.).

Mixed language techniques have been used before to implement these constructs efficiently. Nevertheless, currently popular real-time systems are implemented totally using a single structured, high level, language [5,8].

The following requirements have been identified as basic for providing a sound language-based programming environment for real-time systems:

* The language primitives (i.e. main language instructions or operations) must be small, simple, and well defined.
* Both procedural and data abstractions must be available.
* Separate compilation must be allowed.

* High level access to absolute addresses and interrupt handling facilities must be available.
* Concurrency features and constructs should be inherent in the language in order to allow multi-tasking and concurrent programming [13]. This effectively minimises task execution time, utilises more efficiently the computer hardware, and hence gives shorter response times.

Modula-2 satisfies all these requirements. In fact, an assessment of several concurrent programming languages shows that Modula-2 appears to be among the best languages for real-time programming [12].

Recent assessments consider Modula-2 as 'inherently more secure' than Ada, C and Pascal [14]. Still, however, Pascal, C and Ada constitute a great challenge that Modula-2 has to face, especially in the 1990's [15]. What follows is a brief assessment of its competitors.

4.5.2 Assessment of Competitors

a) C language: C has few intrinsic strengths other than its low level programming facilities. One particular strength is the ability to exploit target architectures, and for the low overhead imposed by C run-time systems for embedded applications [15]. C, however, was not designed as a software engineering language. Using C, we cannot 'hide' structures nor have modules, nor are there facilities to compile individual segments [16]. Moreover, there are no concurrent programming facilities (that was subsequently added by C++ [17], discussed later). Moreover, as every procedure in C is global to the whole program, there is no protection from mis-use for variables (side effects) [18]. Finally, analysis tool support

is poor and attempts are being made to legislate against its use in safety critical software [15].

b) Pascal: Pascal has the virtue of being small and well understood. It is the almost-universal pseudocode of computing. It has a rich range of supporting tools and high quality implementations. On the negative side, however, Standard Pascal is too small and restrictive for many. It does not have the type-secure separate compilation facility of Modula-2 and Ada. Strangely, Pascal is also let down by the lack of validated cross-compilers; hence, there are problems in fully exploiting target architectures.

c) Ada: Ada, on the other hand, was designed to include all the above requirements. In fact, it was designed with operating systems features as an integral part of it. Ada probably represents the biggest challenge to Modula-2 in the long term; especially as the number of compilers (even for personal computers) continues to grow. However, its size and complexity have been a cause of concern. The code size tends to be large, complex and, in some cases, very slow [11].

4.5.3 Possible Competitors of the Future

This is a difficult category to speculate about. However, there is a great interest shown up lately in languages like C++ and variants of Pascal with a corresponding interest in object-orientated programming. C++ is a special case, because of its relationship with C. We can observe C++ to C translators enabling the language to spread rather quickly. In fact, a number of software houses claim to be convinced of the benefits of developing applications using object-orientated

methods [15]. Despite the fact that C++ does not meet many of the requirements outlined above, nor is there a large supporting tools or libraries; Modula-2 is challenged by the object-orientated programming languages like C++ and variants of Pascal.

4.5.4 Why Modula-2

In this research programme, Modula-2 has been chosen for a number of reasons:

* The relative simplicity and flexibility of the language.
* Its wide availability on most micro-computers, at low cost [19].
* Its good execution speed and memory requirements (efficiency).
* Its good support for software development and building systems through the use of modules and process abstractions [20,21].
* Inbuilt device and interrupt handling facilities.
* Inbuilt low level (machine access) facilities.
* Inbuilt support for quasi-concurrency (this facility has been imitated precisely, in a recent project, in C [22]).

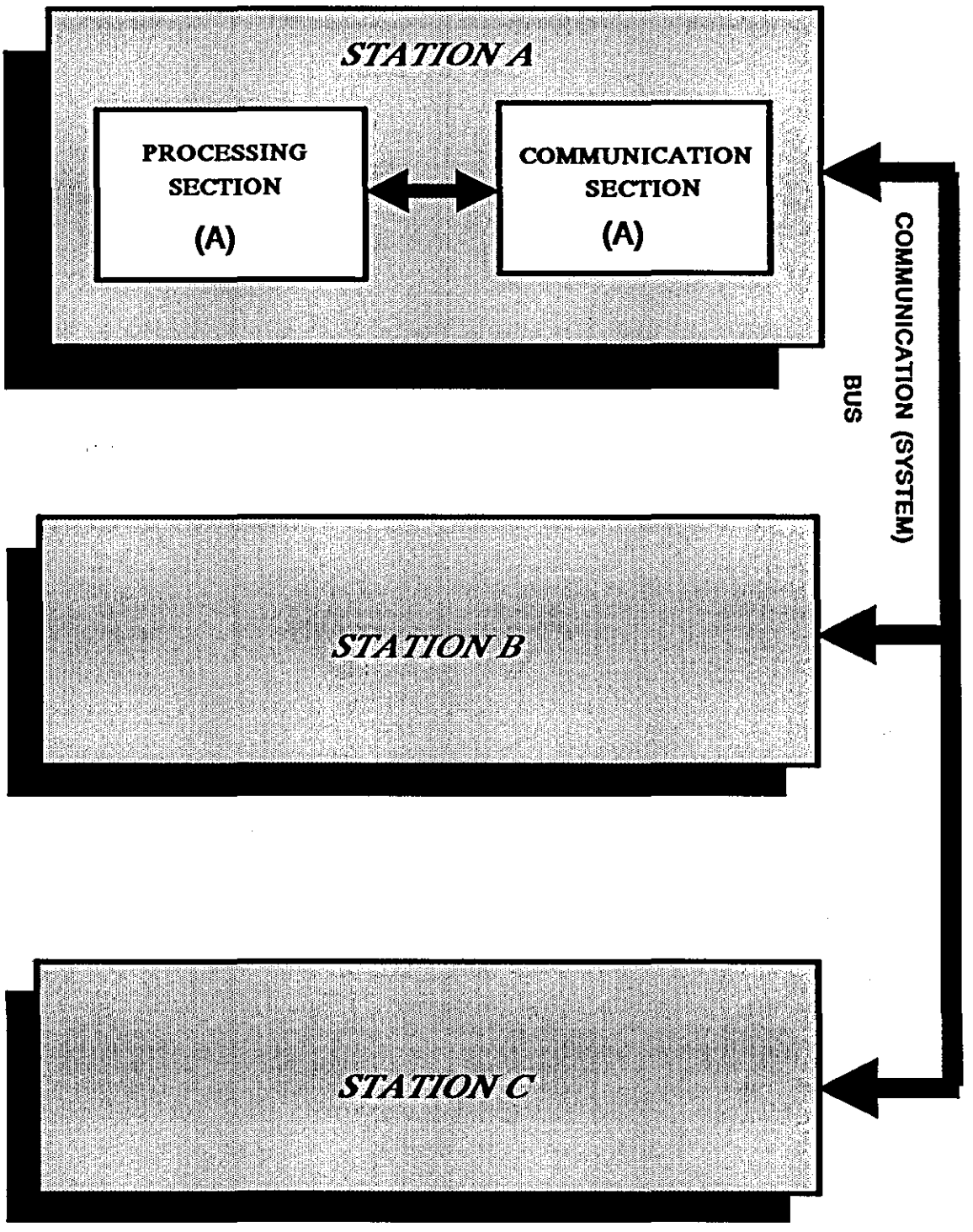


Fig. 4.1 SYSTEM CONFIGURATION

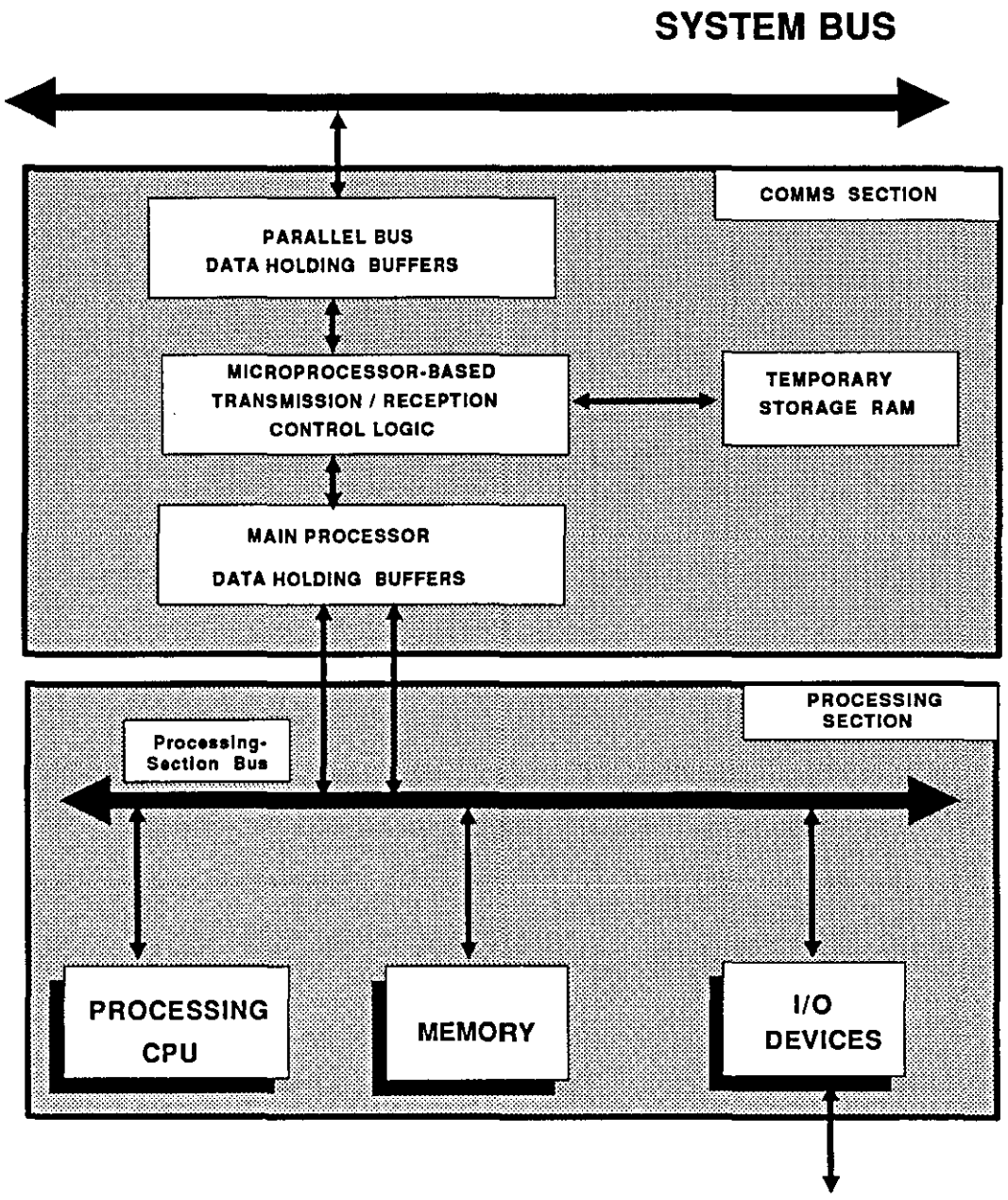


Fig . 4.2 MULTIPROCESSOR NODE - FUNCTIONAL STRUCTURE

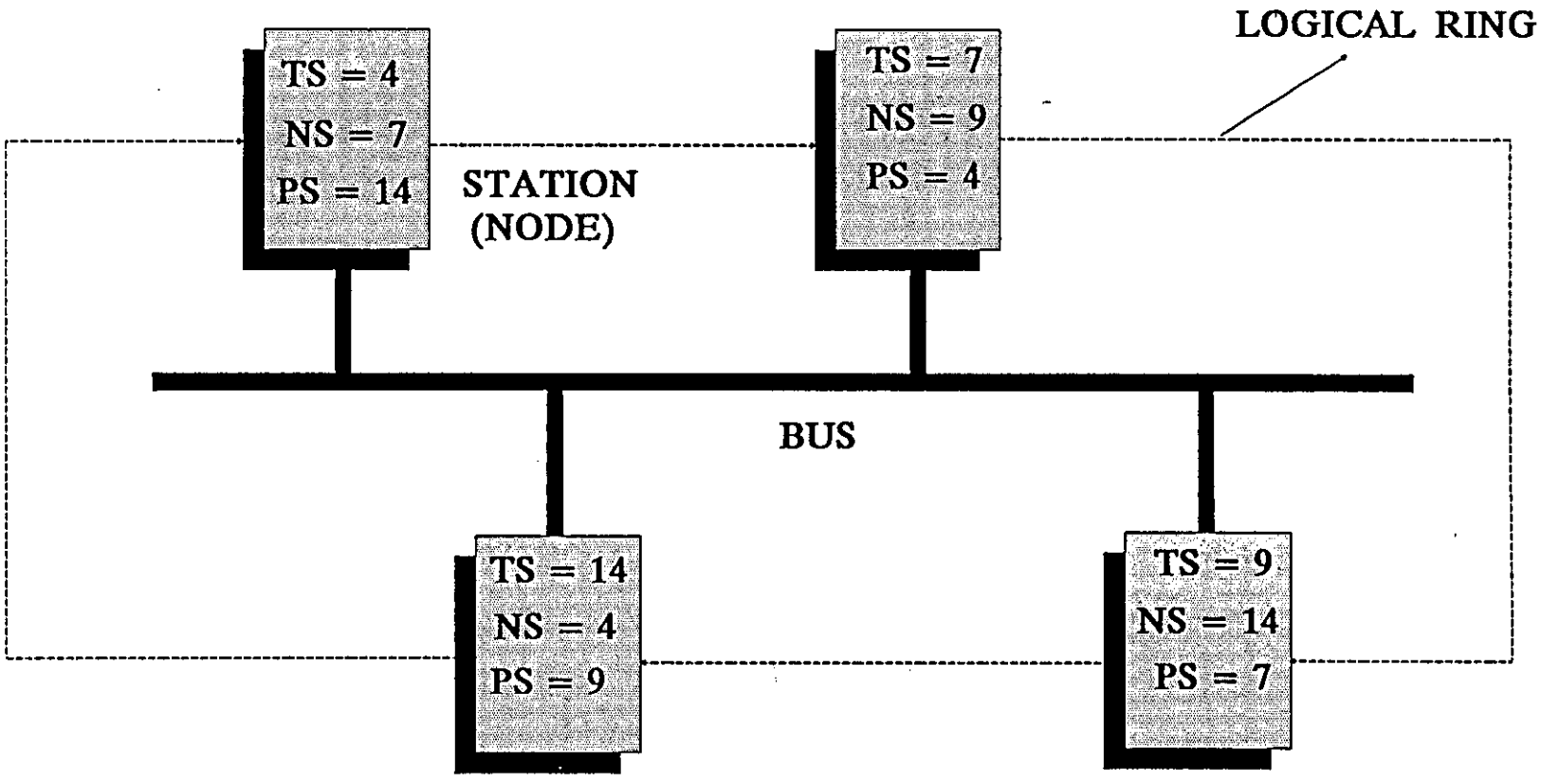


Fig. 4.3 TOKEN PASSING ON A LOGICAL RING

CHAPTER 5

CHAPTER 5

MULTI-PROCESSOR SYSTEM - HARDWARE STRUCTURE

5.1 OVERVIEW

A reconfigureable, loosely-coupled, multi-processor system has been implemented for use in fault-tolerant, real-time applications. Each processing unit (station) consists of a single board computer. The communication and processing tasks are decoupled on each board, a separate processor being dedicated for each task. It uses a fast, single shared, parallel bus for communication between these tasks, bus control being fully distributed. Each station has two main blocks (Fig. 5.1), a communication section and a processing section. The functions of each block are described fully in Appendix A.

5.2 SYSTEM INTERFACING

There are two interfacing stages within each station. First each station has to manage the flow of information sent over the system bus; secondly, within each station, data exchange between the processing section and the communication section must be supervised and controlled.

Interfacing and data transfer is designed to be fast and simple, being organised as follows (Fig. 5.2);

a) System bus interfacing: A total of 16 lines are used on the backplane bus (see Table 5.1). These consist of;

* Four address lines (SS0-SS3)
* Eight data lines (D0-D7)
* Four control lines (START, BUSY*, SWRT*, and SSS*)

Four address lines are required to address up to sixteen stations in the network. The four control lines (see Table 5.1) are needed to;

* Synchronise the start operation for token bus construction (START).
* Hold a station from transmitting data when the recipient station is still busy (BUSY*).
* Inform other stations that a particular station wants to transmit a data message over the system bus (SSS*).
* Control data transfer over the system bus between transmitting and receiving stations (SWRT*).

For critical systems where fault degradation must be gradual, single point failures need to be eliminated. In a bus-based processor system the bus itself (and its associated drivers/receivers) give rise to such a situation. Hence in such application the bus must be duplicated. By using a simple structure such as the one devised here the backplane bus may be replicated at a relatively low cost for use with standard Eurocard size backplanes.

b) On-Board Interfacing (OBI): This is the interface within each station between the processing and communication sections. A total of fourteen lines are used within this interface (see Table 5.2). These consist of;

* Eight data lines (D0-D7)
* Six control lines (MAINCS*, MAINWR*, MAINRD*, DMAREQ*, DMA0, and DMA1)

This interface gives the processing section the right to access the communication section's temporary storage RAM. It enables the processing section to:

* Access the communication section's temporary storage (MAINCS*) for a read operation (MAINRD*) or a write operation (MAINWR*),
* Signal the communication section (DMAREQ*) for a request of data transfer (RDT) and to indicate the end of data transfer (EDT).

All data is exchanged between the communication section and the main processing section using direct memory access (DMA) techniques. The DMA controller is located in the main processing section and generates the required control signals (read, write, and chip select). Control of all data transfers resides with the communication section(DMA0 and DMA1).

5.3 COMMUNICATION SECTION

The communication section (Fig. 5.1) consists mainly of a network control logic for transmission/reception of data, a temporary storage RAM, and a number of data holding buffers.

Fig. 5.3 shows a more detailed functional diagram of the communication section, the main sub-systems being:

* Communication CPU.
* A serial interface.
* Communication Support Module (CSM).
* Temporary Memory Store (TMS).
* A watchdog timer.
* System bus buffers.

5.3.1 Communication Processor

The sub-system is centred around a Hitachi 64180 processor [1,2] which controls the operation of all the main sub-systems (CSM, temporary storage, etc.). It consists of a 32K RAM and 32K EPROM memory space. The address decoding based simply on the state of one of the address lines (line A15). An RS232 compatible serial interface is provided for the connection of a terminal or VDU. A line driver/receiver pair of devices is used to boost the 64180's asynchronous serial port 1 to RS232 levels.

5.3.2 Communication Support Module (CSM)

The heart of the communications circuitry within this sub-system is the CSM module (Fig. 5.4), based on Eraseable Programmable Logic Device (EPLD) technology. This module, implemented using an Altera EP1800 device [3,4], has a number of advantages over designs based on discrete packages. The major one is a substantial improvement in PCB (Printed Circuit Board) component packing density.

The module provides a wide range of functions, as follows;

* Chip select lines for the memory devices.
* A series of registers for the control of the system bus by means of the communication processor.
* Address recognition logic.
* Timing and control circuitry.
* On-board interfacing.

The major features are discussed below.

a) Address recognition logic: The address recognition logic provides the ability for a station to read its own address, as set on selector switches on the card. It also provides an automatic recognition response when this station is addressed on the system bus.

b) Timing control circuitry: The timing and control circuitry operates in either one of two modes. In receive mode it takes the strobe signals from the system bus and latches data into the scratchpad RAM. When transmitting, it generates timing signals for both outputting the data from the scratchpad RAM and also strobing it across the system bus (this action is performed under the control of the communication processor).

c) On-board interfacing (OBI): The six handshaking lines controlling this interface (described earlier in section 5.2) make extensive use of the CSM module (refer to Appendix A). For communication between the processing section and the scratchpad RAM, the communication processor's data bus is released (under the control of the CSM module) and made available to the main processor. This action is initiated by the communication processor. The bus is

subsequently returned either by receipt of an interrupt at the end of the transfer, or by the reception of the station's address on the system bus.

5.3.3 Temporary Memory Store (TMS)

From Fig. 5.5 it can be seen that a major component of the communication section is a temporary storage RAM area (usually called a scratchpad RAM or TMS). This is used to store data messages from the system bus and to hold data which is ready for transmission onto the bus (Fig. 5.5). The device used, a TMS9650 dual port RAM [5], is shown as two sections to differentiate between its two ports. Port A is used with the processing section of the station (Fig. 5.6), while port B is used for communication with the system bus.

Port A interface is shared by the communication and main processor. Each processor accesses this port of the TMS via control and data signals (controlled by the CSM). Port B, on the other hand, is connected to the system backplane bus buffers. It is used to handle the transfer of data into and out of the TMS. Information is transmitted and received in byte serial form, the TMS being used as a temporary store for this information. Port B has three modes of operation; idle, transmit, and receive (refer to Appendix A for full details).

The communication processor controls the access to both sides of the RAM, these being mutually exclusive. Normally access to port A is given over to the communication processor but is transferred to the main processor when it wishes to read or write to the scratchpad RAM. The actual control of the transmission and reception of data is performed by the transmission control logic, which is controlled from

the communication processor (Fig. 5.5). Control of port B is given over to the local (own station) control logic during transmission. However, it is transferred to the remote control logic in reception mode.

5.3.4 A Watchdog Timer

The watchdog timer circuit provides a mechanism for program recovery in case of failure (program crash). The circuit (based on a monostable device) is designed to be constantly retriggered by the software before it times out. If the system fails to function properly then time-out occurs, and a non-maskable interrupt (NMI) is generated. The resulting exception response is user defined; in this implementation a program restart is initiated.

5.3.5 System Bus Buffers

This block includes the data and control buffers of the system (backplane) bus. Their function is to ensure that all system bus signals have the ability to drive the system bus and all devices connected onto it. These tri-state buffers are enabled only in reception or transmission modes, their direction being determined by the mode of operation.

5.3.6 Power-on Reset Circuitry

This circuitry provides a reset signal to the HD64180 microprocessor after power-on and in response to a manual reset command.

5.4 PROCESSING SECTION

In the multi-processor system described here no assumptions are made about the structure of the processing sub-system. For some applications a separate processor may not even be used (e.g. display sub-systems). However, where the design is used to support a distributed computing system each processing section will have its own memory and I/O devices. All application software runs in these sub-systems. They are completely isolated from the system bus, having nothing to do with the communications activities. The processing section merely interchanges information with the communication section; moreover the transfer protocol is kept simple by using a combination of interrupt and DMA interfacing between the two sections.

The processing section (Figs. 5.7 and 5.8) consists of the following main blocks:

* CPU section.
* Memory.
* Serial communication.
* On-Board Interface (OBI).

5.4.1 CPU Section

The CPU section is based on the use of an Intel 80188 processor together with an Intel 8087 numeric processor extension. An advanced bus controller (82188) is included to provide 80188/8087 interfacing. The complete CPU section is composed of:

* Microprocessor.
* Hardware maths unit.

* Bus controller.
* Address/Data buffers.
* Power-on reset circuitry.
* Single step control.
* Watchdog timer.

a) Microprocessor: Processing power is provided by an Intel 80188 high integration 8-bit microprocessor [6], which includes the following internal units:

   i) Clock generator.
   ii) Programmable interrupt controller.
   iii) Programmable DMA controller.
   iv) Programmable chip select unit.
   v) Programmable timers.

b) Hardware maths unit: Support for fast maths operation is provided by an 8087 numeric co-processor [7].

c) 82188 advanced bus controller: This controller is included to support 8087 interfacing with the 80188 [8].

d) Address/Data buffers: Buffers are included to increase the driving capability of the address and data signals.

e) Power-on reset circuitry: This circuitry provides a reset signal to the 80188 after power-on and in response to a manual reset command.

f) Single step control: A single step circuit is provided to allow for initial hardware testing and de-bugging.

g) Watchdog timer: A watchdog timer is included to provide a means for exception handling should program malfunction occurs.

5.4.2 Memory

The memory for the processing section consists of EPROM, static RAM (SRAM) and optional dynamic RAM (DRAM) (mounted on a piggy back board). Various sizes of EPROM (from 16K to 64K Byte) and SRAM (from 2K to 32K Byte) may be used in this design. The main board (processing section) currently uses the following configuration;

* One EPROM (size 8K byte) - used as a bootstrap.
* One SRAM (size 8K byte) - used as a memory for the application program's stack, data, and heap.
* One EPROM (size 32K byte) - used for the application software.

5.4.3 Serial Communication

Two RS-232 compatible serial communication channels are implemented using a Dual Universal Asynchronous Receiver/Transmitter (DUART) (Signetics 2681 [9]), together with appropriate line interface circuits.

5.5 HARDWARE-SYSTEM OPERATION

From the hardware point of view, the station's operation within the system can be divided into three phases; power-up, initialisation, and operational mode (i.e steady state). The initialisation sequence is performed by a station after power-up or reset procedure, a steady state or normal operational follows afterwards. The following sections highlight the sequence of hardware operations during such phases.

5.5.1 Power-up

When an individual station powers-up, the communication section starts up action first, holding the processing section in a reset state. This is an essential point in order to ensure that the system starts up in a safe mode. Only when the station has established itself in an operational network is this reset action released.

5.5.2 Initialisation

This phase consists of setting-up both the communication and the processing sections in each station.

a) Stage 1 - Communication section set-up (Fig 5.9): This consists of two main stages; hardware initialisation and token bus construction. Hardware initialisation consists of setting-up the:

   * Communication processor.
   * CSM module.
   * Temporary Memory Store (TMS).

   The communication processor set-up includes; the internal registers, wait state generator, serial line interface, watchdog timer, and the internal timers and interrupts required by the software. The CSM and TMS modules set-up consists of resetting and initialising the internal mode registers of each module.

   The second stage is the construction of the token bus. This is the process whereby a number of bus connected stations can communicate as a ring structure (refer to chapters 4 and 6 for more details).

b) Stage 2 - Processing section set-up (Fig. 5.10). This mode starts after the token bus has been constructed and the stations are set in an operational mode. It consists of setting-up the internal mode registers and units of the main processor (timers, interrupts, wait states, etc), the watchdog timer, the serial line interface, the DMA channels for transmission and reception. Finally it creates the program background process.

5.5.3 Operational Mode (Steady State)

When the network enters the normal operational mode, i.e the steady state condition, data messages may be exchanged between stations and task processing is performed by the system.

In an operational mode the communication section monitors the system bus for any message broadcast and the processing section for any data transfer request. The system bus is given priority over the processing section, in case of message reception.

5.5.3.1 Transmission of a Message

To transmit a data message a sequence of operations takes place. In the following discussion it is assumed that the token is currently held by another node, while this node is preparing for message transmission. The sequence of operations is given in a chronological order (refer to Fig. 5.11). For simplicity, the processing and communication sections are abbreviated as PS and CS respectively:

* PS and CS are in operational mode, running background processes (T0).
* PS requires to send a message; it sets channel for transmission (T1).
* PS requests for data transmission - RDT (T2).

* Request is received by CS (but no response).
* PS resumes background process (T3).
* A broadcast message is monitored and received by CS over the system bus, hence no immediate response to PS request (T4).
* CS responds to message request; sets scratchpad RAM area (T5).
* CS generates DMA signal to start transfer (T6).
* PS invokes data transfer to scratchpad RAM (T7).
* PS sends an end of data signal (EDT) to CS at the end of transfer (T8).
* EDT signal is received by CS.
* PS and CS resume background processes (T9).
* CS receives the token (T10).
* CS transmits the message across the network (hardware generated signals are used to output data from the scratchpad memory and to activate the bus buffers and bus control signals) (T11).
* CS sends the token to its successor station (T12).
* CS resumes background process (T13).

5.5.3.2 Reception of a Message

In case of message reception, channel of the processing section is already set for reception. Further, the communication section monitors the state of the system bus continuously. If it detects its own address, or a broadcast address (address for all stations) it prepares for a reception. The following steps are taken by both sections of the recipient station (see Fig. 5.12):

* PS and CS are in operational mode, running background processes (T0).
* CS monitors a system bus message (T1).
* CS initiates the receive data routines for message reception (T2).
* CS checks message and prepares for message transfer into the processing section, in case of a data message (T3).

* CS generates DMA signal to start transfer (T4).
* PS invokes data transfer to scratchpad RAM (T5).
* PS sends an end of data signal (EDT) to CS at the end of transfer (T6).
* CS resumes background process (T7).
* PS starts processing received data (T8).
* PS resumes background process (T9).

TABLE 5-1: SYSTEM BUS LINES

LINES	DESCRIPTION
D0-D7	These eight lines form a data bus over which all traffic between stations take place. The most significant bit is D7.
SS0-SS3	These four lines carry the address of the station onto which data is being transmitted. They are controlled by the transmitting station.
SSS*	This is one of the four lines used to control the action of different stations with respect to the data on the address bus. This line indicates that an address is being output by a station trying to transmit. When it is active all stations should compare address lines to see if they are being addressed.
SWRT*	This line acts as a write strobe. It is controlled by the station transmitting a message and is used by the receiving station to clock the data from the bus into the scratchpad RAM.
BUSY*	This line is used in the synchronisation process at the start of a transfer of a data frame. The line is controlled by the station to which the data is being sent. When a station wishing to transmit sends an address then the addressed station holds this line active until it is ready to receive the data. It then de-activates this line.
START	This line is only used during initialisation of the system. After power up the logical ring must be formed for token passing. This signal is used to synchronise this action.

TABLE 5-2: ON-BOARD INTERFACE (OBI)

LINES	DESCRIPTION
D0-D7	An eight bit data bus.
MAINCS*	A chip select line from the processing section. When this line goes active it indicates that the main processor is reading or writing across the interface. This signal should only be activated once the communication processor has indicated a start of transfer.
MAINWR*	This line is used to indicate a write operation by the main processor.
MAINRD*	This line is used to indicate a read operation by the main processor.
DMAREQ*	This line, pulsed by the main processor, is used both to request a transfer of data and also to signal its completion to the communication processor.
DMA0	This line is set by the communication section to indicate that the processing section should start a transfer of data. This can either be after a DMAREQ* signal from the main processor or after a data frame has been received from the system bus.
DMA1	This is an alternative line used to indicate that the main processor should start a transfer of data.

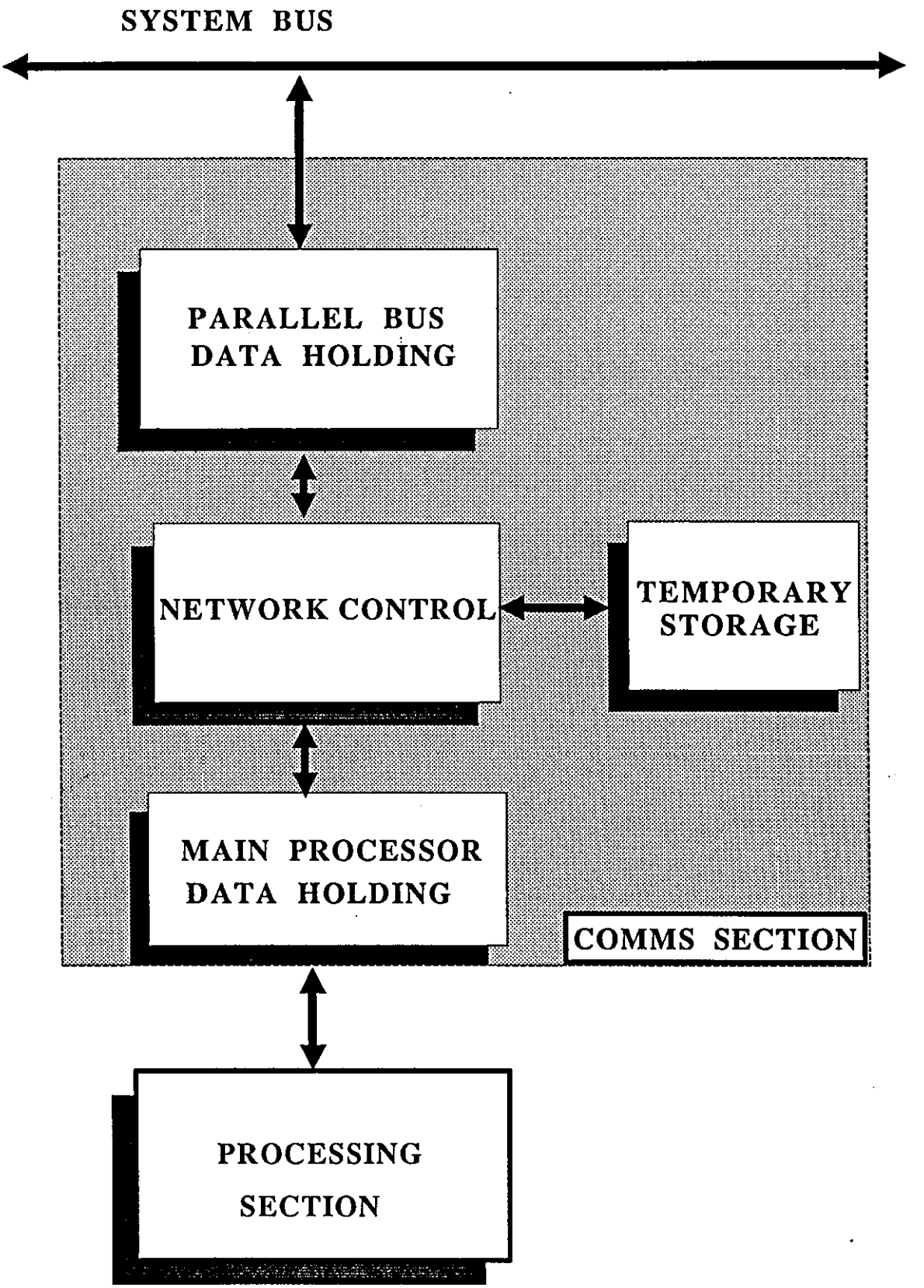


Fig. 5.1 FUNCTIONAL BLOCK DIAGRAM of a STATION

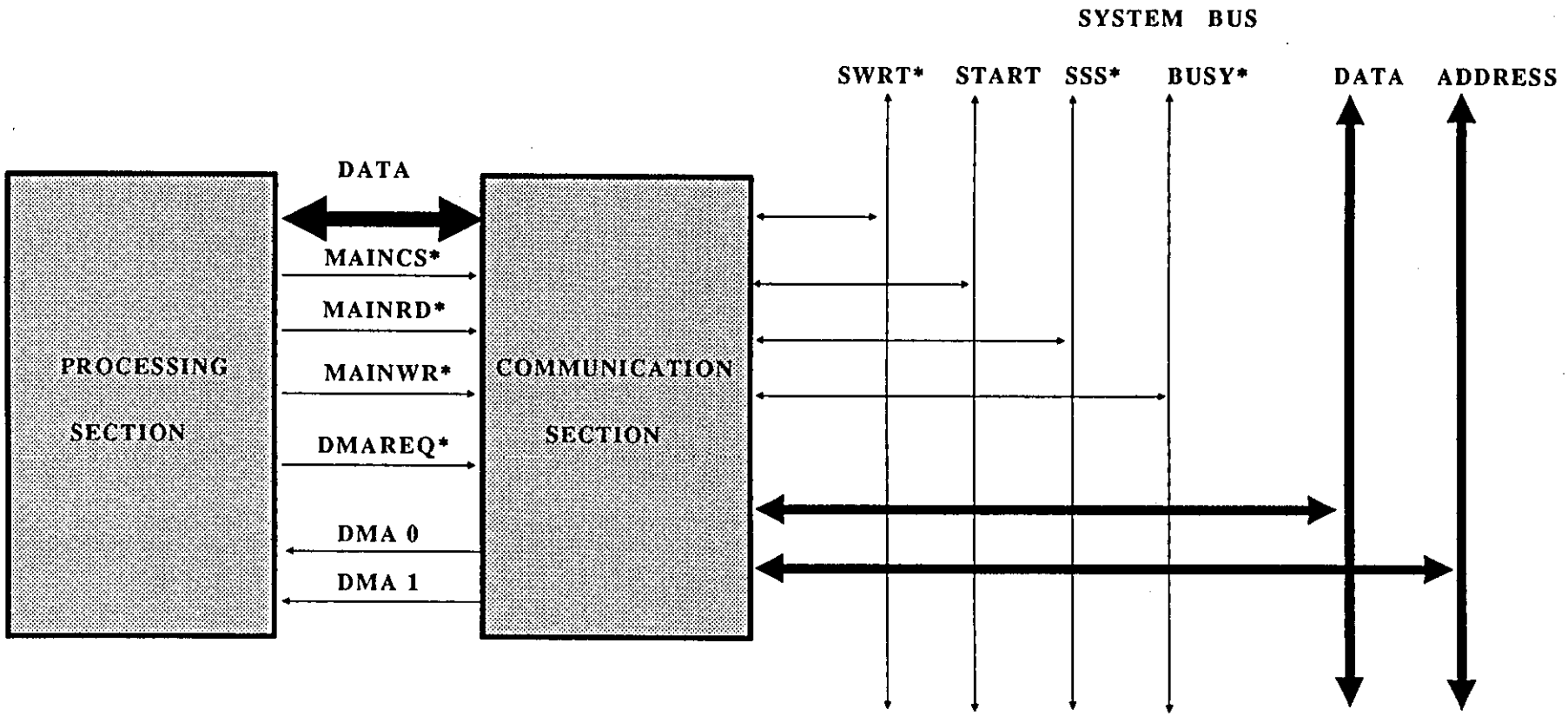


Fig. 5.2 SYSTEM INTERFACING

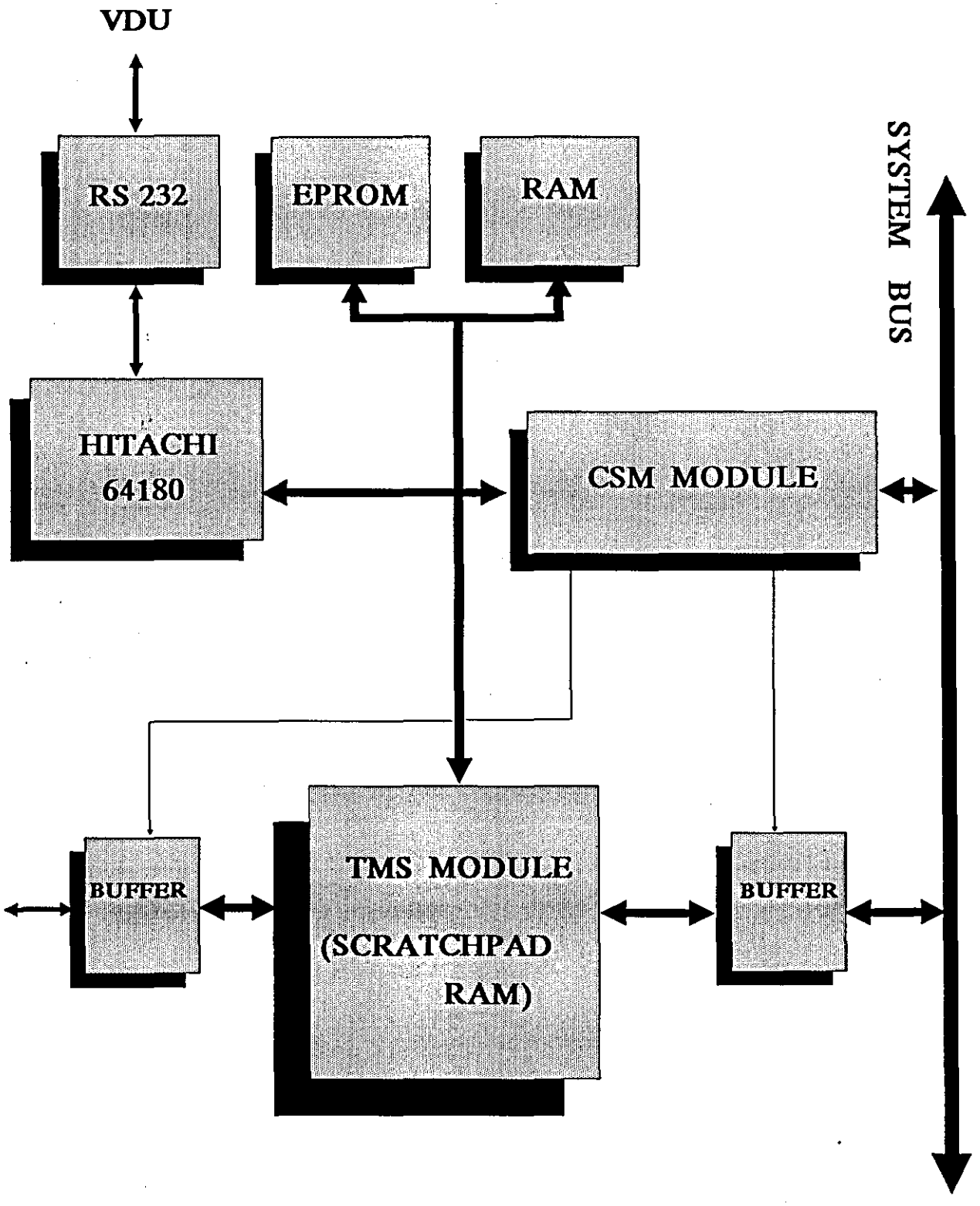


Fig. 5.3 THE COMMUNICATION SECTION - DETAILED STRUCTURE

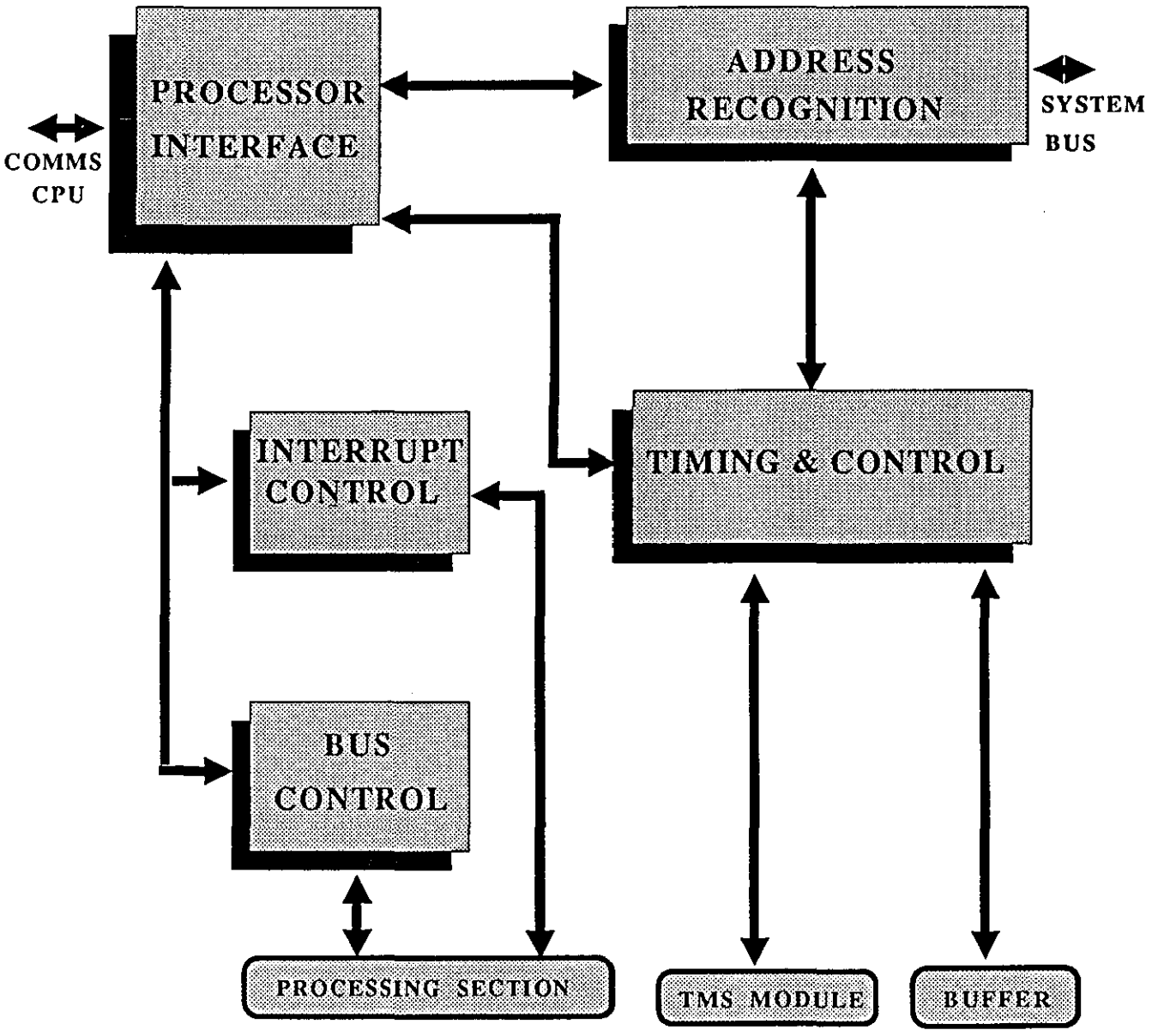


Fig. 5.4 COMMUNICATION SUPPORT MODULE (CSM)

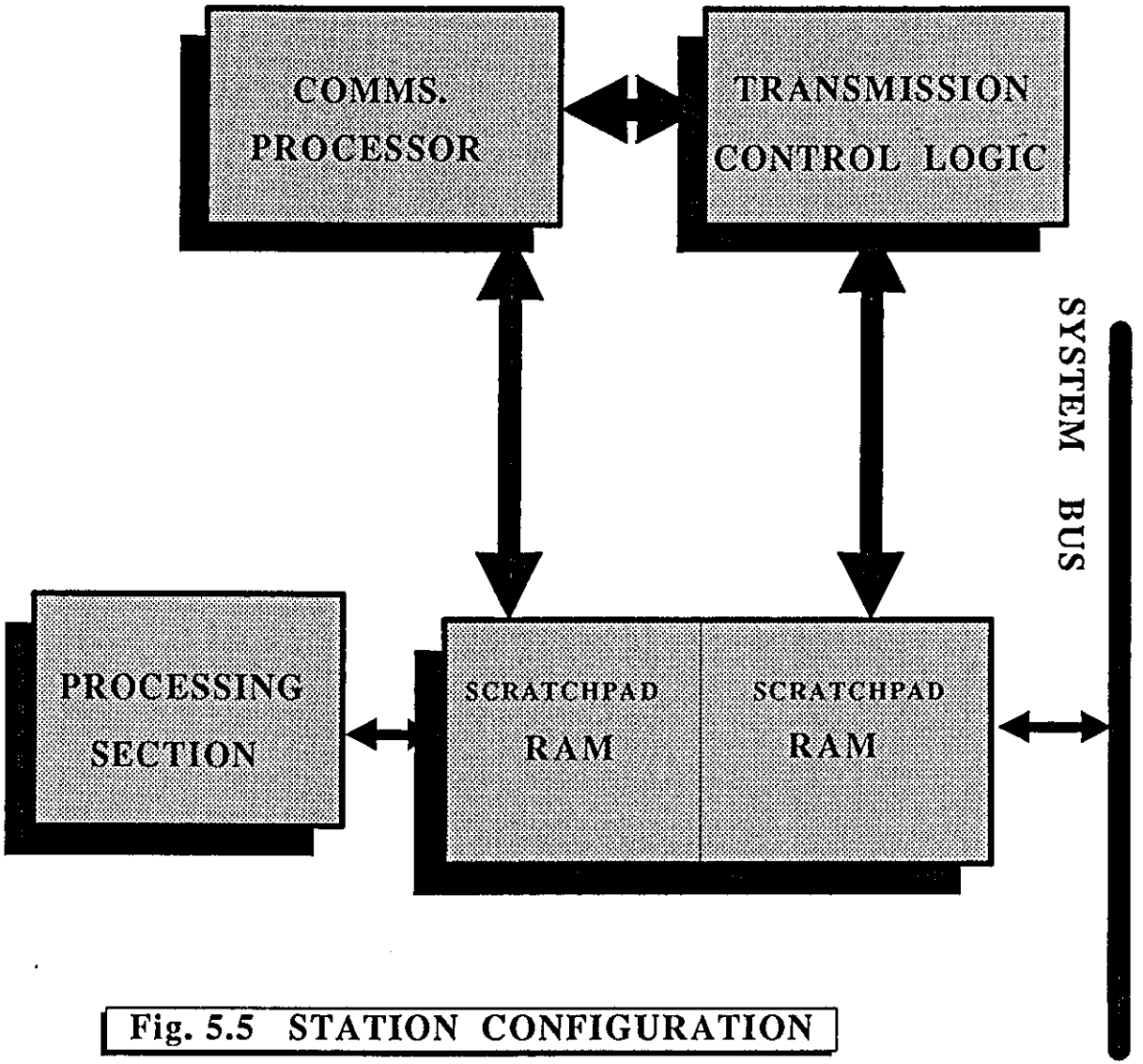


Fig. 5.5 STATION CONFIGURATION

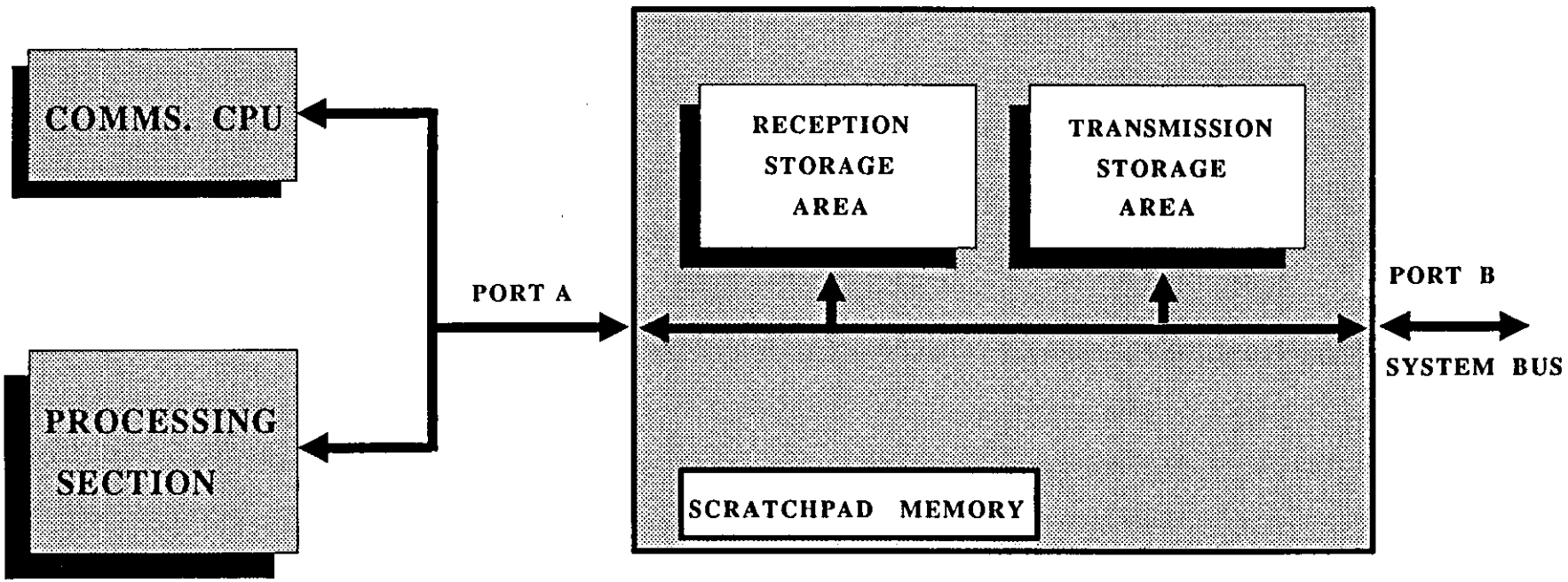


Fig. 5.6 BLOCK DIAGRAM OF THE SCRATCHPAD MEMORY (TMS MODULE)

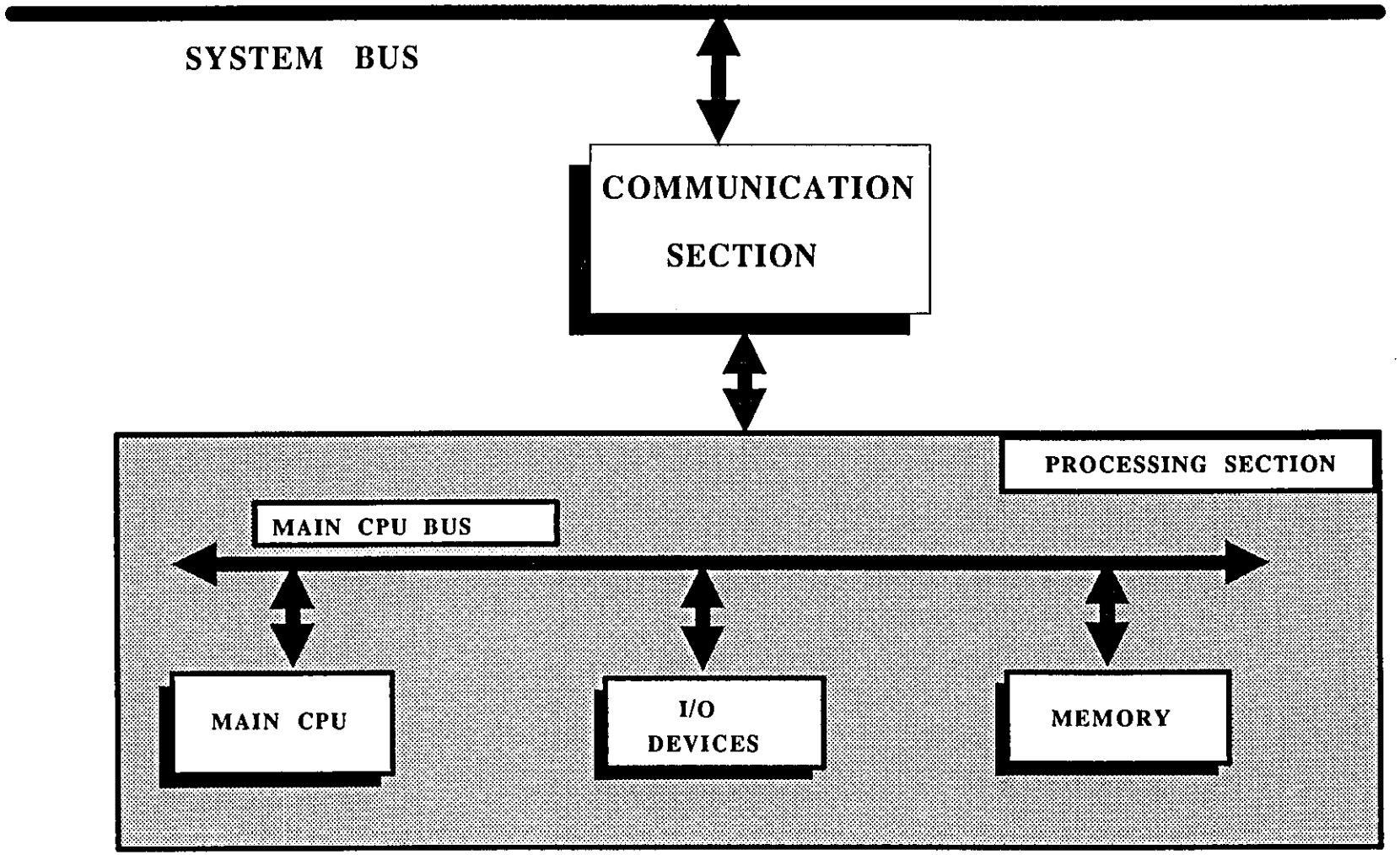


Fig. 5.7 PROCESSING SECTION - OVERALL STRUCTURE

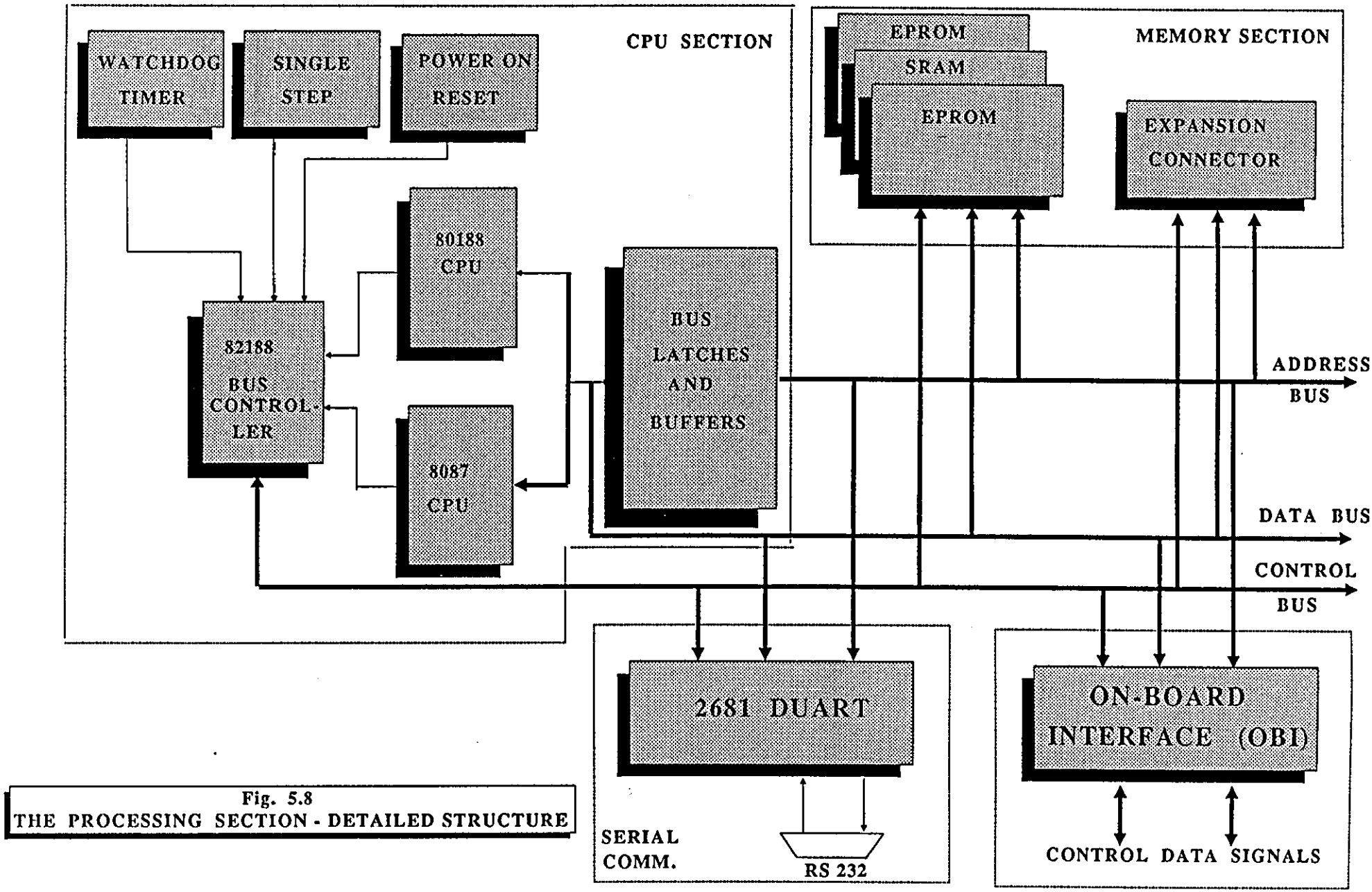


Fig. 5.8
THE PROCESSING SECTION - DETAILED STRUCTURE

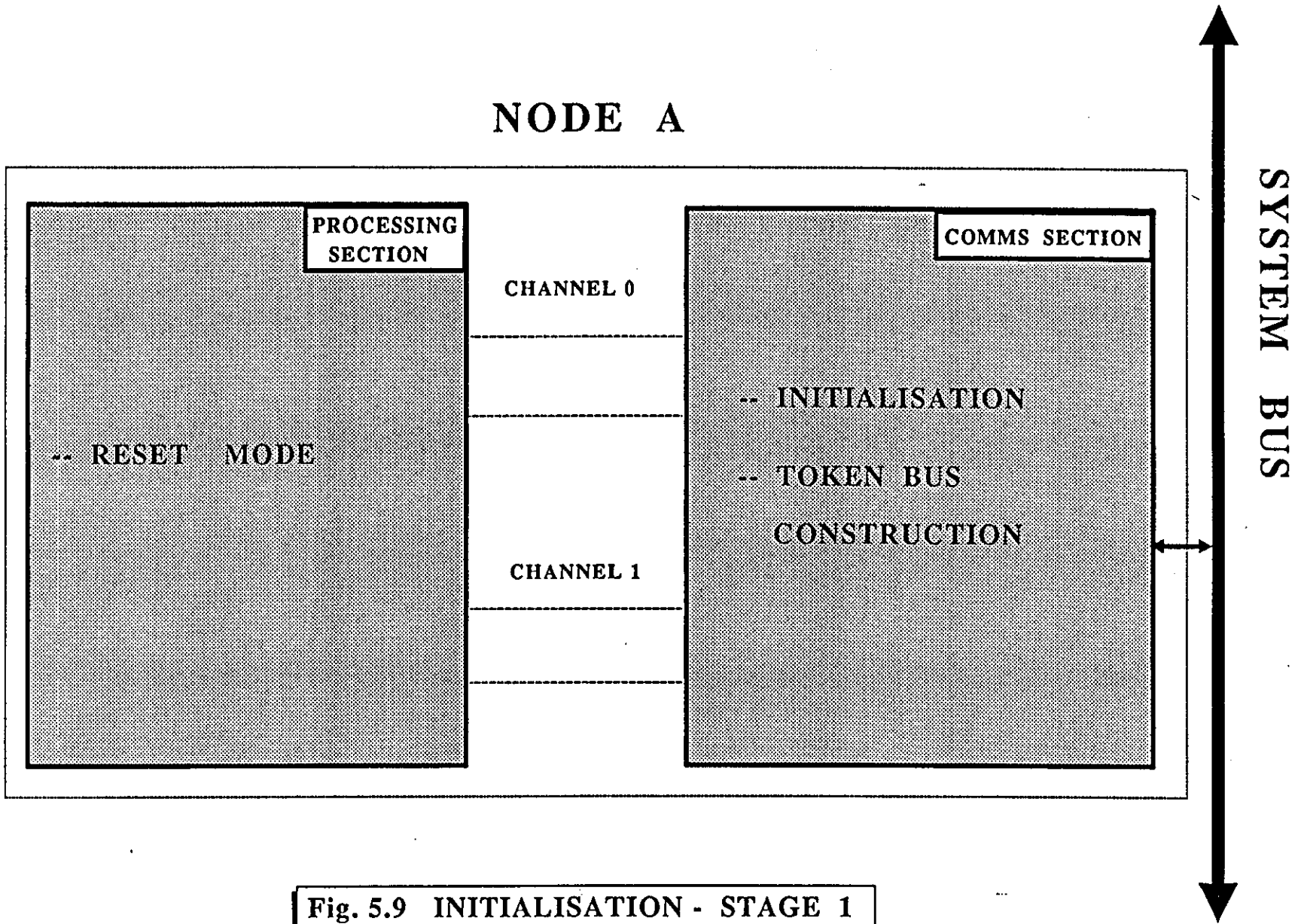


Fig. 5.9 INITIALISATION - STAGE 1

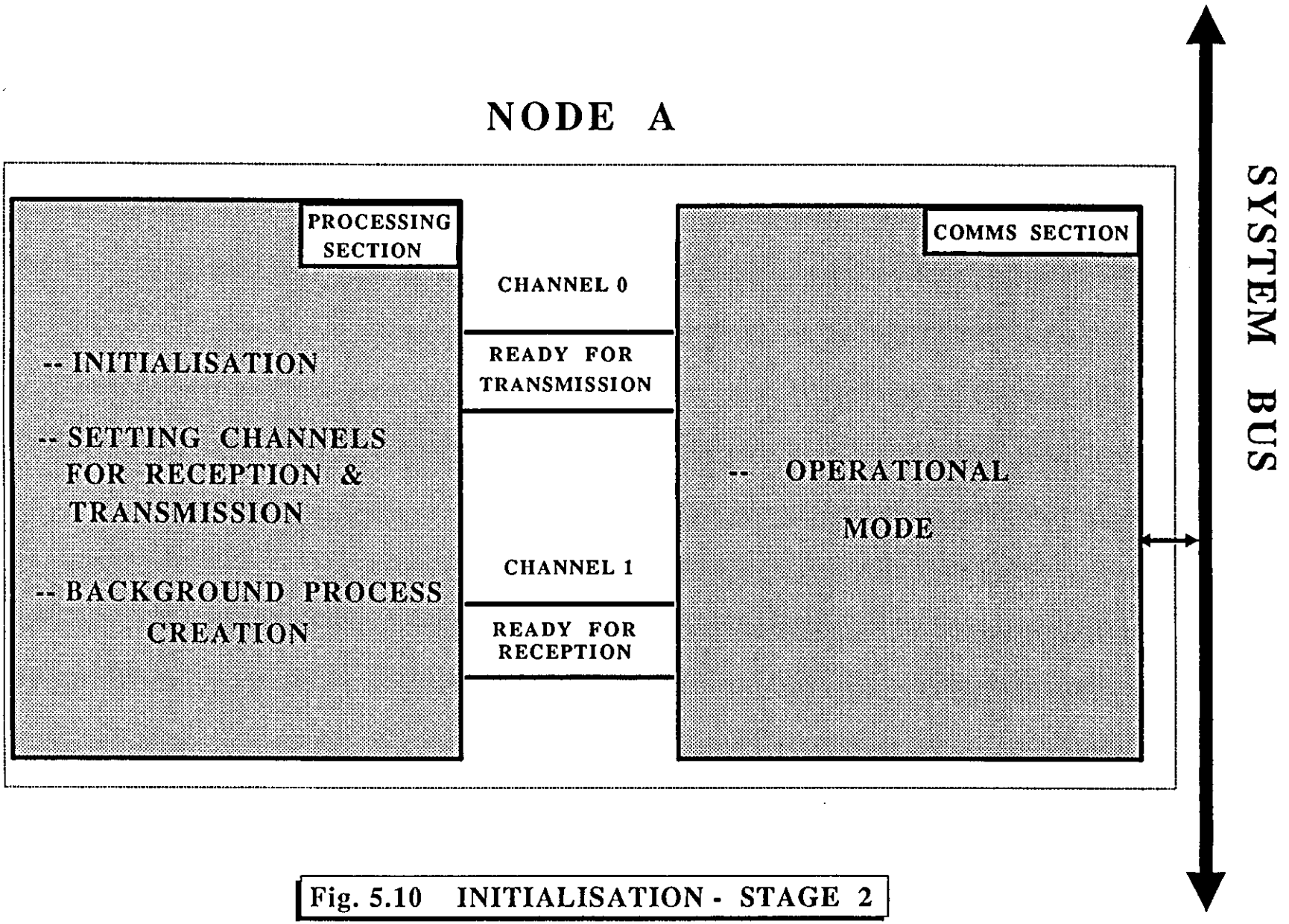


Fig. 5.10 INITIALISATION - STAGE 2

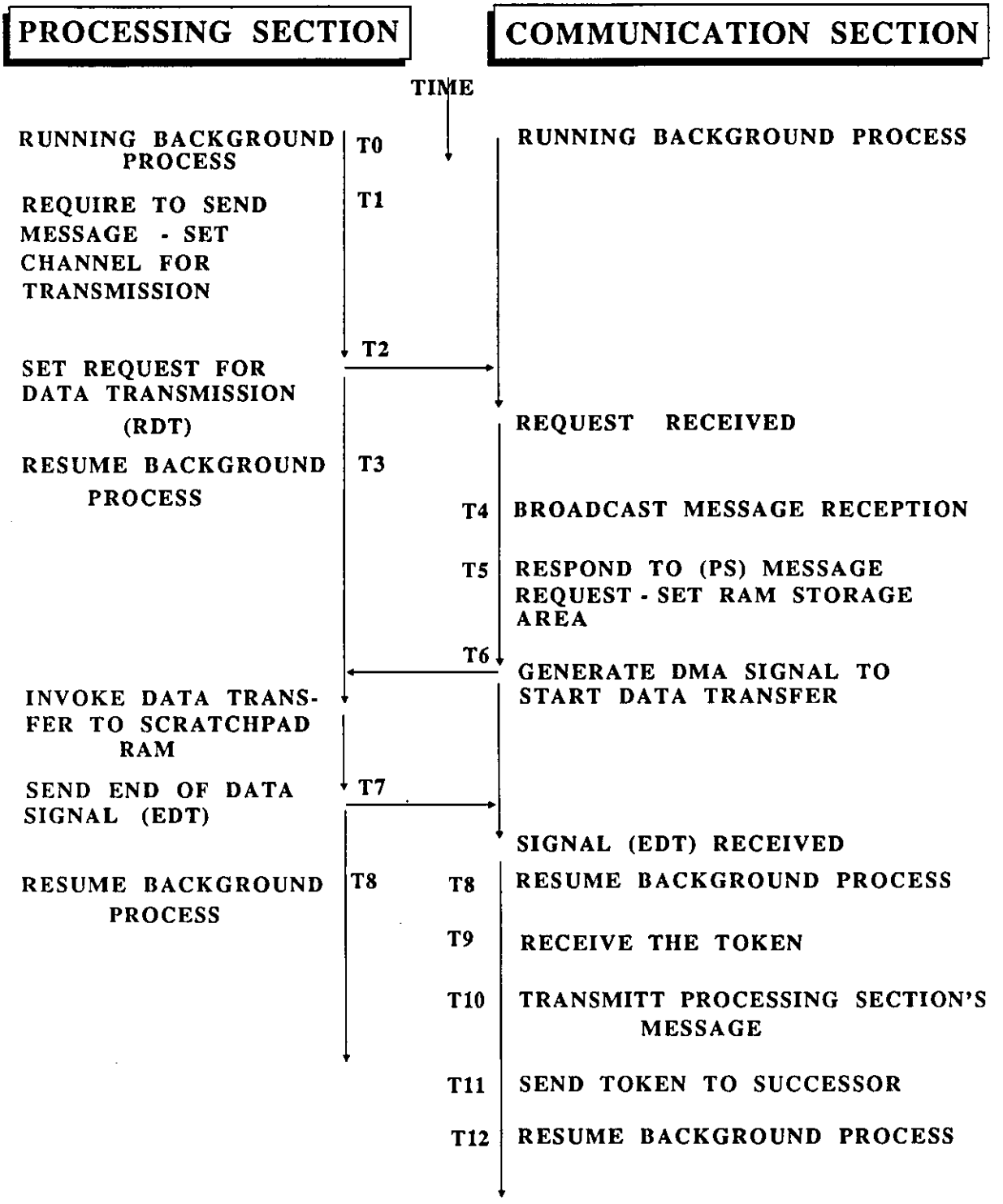


Fig. 5.11 TRANSMISSION OF A MESSAGE

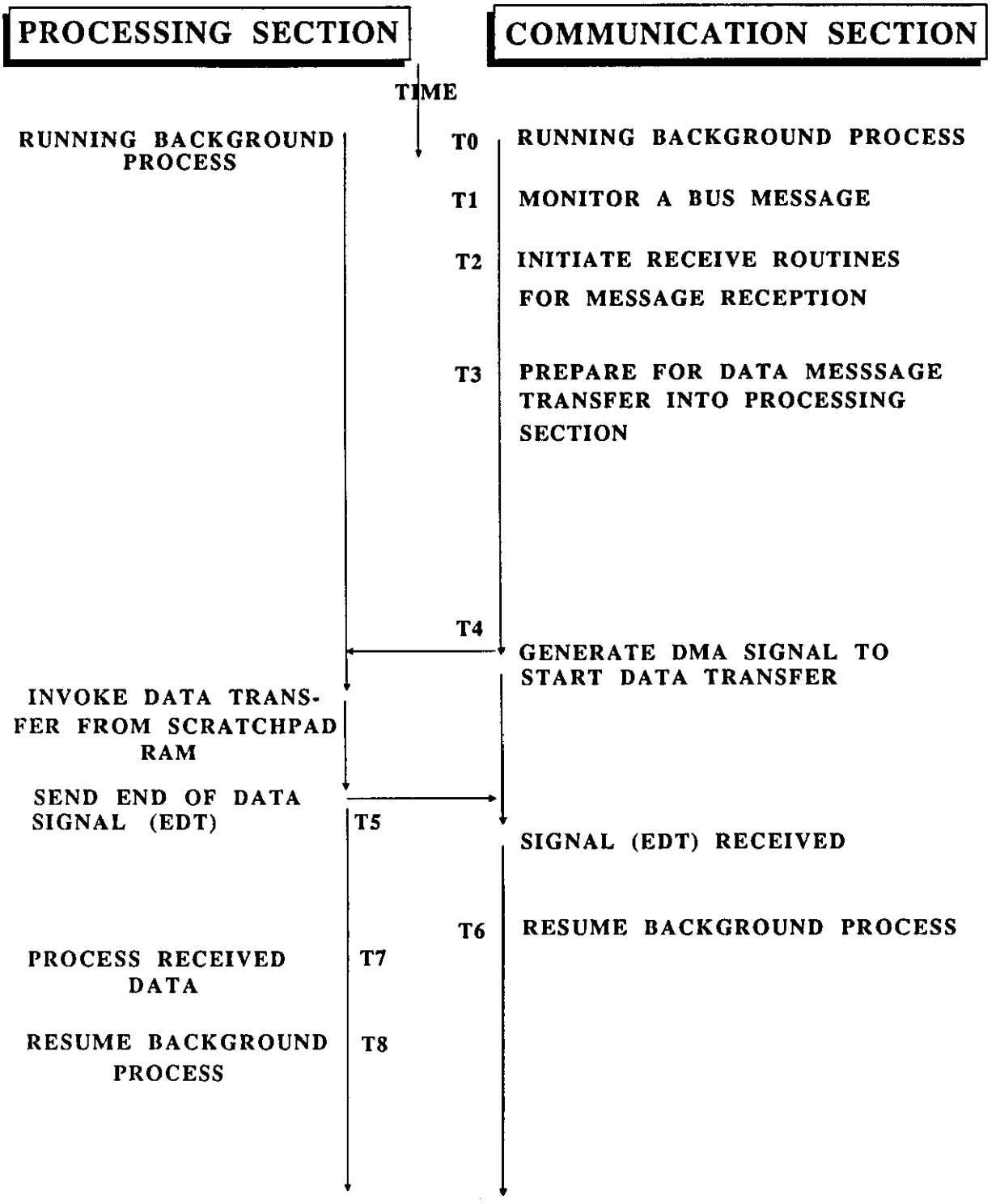


Fig. 5.12 RECEPTION OF A MESSAGE

CHAPTER 6

CHAPTER 6

MULTI-PROCESSOR SYSTEM - COMMUNICATION SOFTWARE

6.1 SOFTWARE REQUIREMENTS

As stated previously, the main function of the communication section is to support system communication activities. Its functions, at a detailed level, are to:

* Establish address information.
* Support all communication access functions of the network.
* Perform memory management of a fast message buffer (the scratchpad RAM).
* Control data exchange with the processing section.

These tasks are implemented mainly in software through the use of a communication protocol. The protocol controls and coordinates information flow between the processing section of a station and the system bus.

The communication software is designed in a modular, structured manner, being implemented using the Jackson Program Design Facility (PDF) package. The core element of the communication section is a Hitachi 64180 processor, which includes Z80 code as a sub-set of its instructions. Programs for this were developed using the FTL compiler, the application software being programmed into EPROM. A description of the program modules and their corresponding diagrams is fully shown in Appendix C.

6.2 DESIGN TECHNIQUES

There is a major difference between getting a program to work and getting it right. Thus, it is very important in the development of reliable software to have a complete and correct understanding of what the system is expected to carry out. The design method used to convert the specified requirements into software code also affects the reliability of the software [1].

The basic software design method used here is that of structured design [2], a technique which contains the merit of both Top-Down design and Modular programming [3].

Structured design is a technique that significantly increases the reliability and readability of program, while decreasing the required testing of such programs. It is a set of concepts and guidelines whose purpose is to reduce cost, time and effort in developing and maintaining computer programs.

Diagramming techniques are used throughout the software design described here, specifically that based on the Jackson Structured Program technique (JSP) [4]. Each program structure diagram (e.g. Fig. 6.3) is read from top to bottom to obtain more detail on program activities and from left to right to get the time sequences. Such techniques are excellent at describing what needs to be done rather than how it should be carried out. Furthermore, sections can be added or removed as the work proceeds without disturbing the rest of the diagram. This greatly assists programme development and modification activities.

The Jackson chart is constructed to describe the software to a specific level of detail. The lowest levels represent simple functions that can be translated into program format. Generally the recommended control structures of structured programming have been used in the writing of the program source code.

JSP can be automated using software packages such as the Jackson Program Design Facility (PDF) [5]. These can be used to construct program structure diagrams (PSD); from these code may also be automatically generated if suitable code generators are available (unfortunately not for Modula-2 at the present time). In the system implemented here the PDF package was used to construct PSDs and the program control structure. These diagrams were subsequently used to write the program code. It can be seen (Appendix C-software diagrams) that, at the lowest levels, there is a one-to-one correspondence between the diagrams and the code.

6.3 IMPLEMENTATION OF THE COMMUNICATION PROTOCOL

6.3.1 Software Module Structure - Overview

The communication software (protocol) code is implemented in a modular, structured way using Modula-2. The structure consists of:

* Main (program) module.
* Second level (functional) modules.
* Lower level (service modules).

a) Main module - 'Run Comms'

This is the highest level module. It holds the communication software executable code. It consists of a single program module (named 'Run Comms') that is functionally decomposed into a number of second level modules that are called within the main module.

b) Second level modules

The second level modules are:

* INITIAL module.
* STARTUP module.
* STEADY module.

These modules make use, in turn, of lower level service modules.

c) Service modules

A number of lower level modules (service modules) are available to provide specific software services to higher level modules (i.e main and second level modules). These modules are:

* Control-frame modules.
* Message-exchange modules.
* Hardware-related module.

i) Control-frame modules: These modules (SENDLIB, RECLIB, etc.) consist mainly of control frames necessary for the implementation of the Token Passing method. They are called (imported) by the main module and make use of other lower service modules; 'Messages' and 'Mains'.

ii) Message-exchange modules: Their main function is to implement communication functions between the system bus, the communication section and the processing section. These modules make use of a lower module, 'Signals', for the control of hardware. They consist of two modules:

* 'Messages' - For message-exchange with the system bus.
* 'Mains' - For message-exchange with the processing section.

iii) Hardware-related module: This module, 'Signals', is the lowest level of application. It controls the different hardware signals (i.e enabling and disabling).

Finally, there is one routine that supports the Modula-2 protocol code discussed above. This is the run-time support module 'CPM100', which is written totally in assembly language. This is discussed in a separate section.

6.3.2 Communication Software (Main Module - Run Comms)

This module holds the communication software code. Its function is to implement the token passing protocol described earlier in chapter 4. It makes use of second level (functional) modules, and a number of service modules. The different control message-frames (imported from Message-exchange module) are used extensively for initialisation, operation, and maintenance of the token passing protocol.

Generally speaking, from the network's point of view, operation of the main module can be divided into three main parts (Fig. 6.1):

* Initialisation.
* Steady state.
* Maintenance.

From the station's point of view, however, the system may be functionally decomposed into three top level functions (Fig. 6.2):

* Initialising the board (station).
* Entering the ring.
* Running in operational mode.

These, in turn, are divided into sub-functions (Fig. 6.3). This sub-division is further continued until a satisfactory lowest level of functional representation is obtained. These lowest level functions are usually simple and easily translated into program source code. Routines shown in Fig. 6.3 are listed below. For full module description and diagrams refer to Appendix C:

a) Initialise the board (station)

In this mode, the station initialises the communication section and synchronises its operation with respect to other stations (ready for constructing the token ring). This is done by setting various hardware control lines, timers and defining the station's address (TS).

b) Enter the ring

Once all the stations are initialised and synchronised, they activate their response timers (RT) in order to construct the token ring. Each station then monitors the bus for message reception, the result producing one of three possible courses of action:

* If the RT timer times out before any message is received, the station follows the routine for entering the ring as 'The First Station'.
* If the station receives a claim token frame message then the station follows the routine for entering the ring as 'Not The First Station'.

* If a message frame other than a claim token is received then the station follows the routine for entering the ring as ' a plugged-in station'.

In the various cases shown above, stations follow different paths to achieve the same task (i.e ring construction). Once a station knows its own address (TS), the previous station address (PS), and the next station address (NS), it passes the token to the next station in the ring. When the last station in the ring (LS) acquires the information and connects with the first station (FS), the token is said to be constructed. When this has been achieved, each station should be in possession of: number of stations in the ring, its own address (TS), and other stations' addresses (PS, NS, FS and LS)

c) Run in operation mode

Once the ring has been constructed the stations are said to be running in the operational mode. During this mode, the token is received periodically for message passing between the network stations. In normal operations the station has to respond also to other messages that may arrive as a result of adding a new station (insertion) or a station drop-out (deletion). In this case, as the ring construction has changed, token information-update has to take place.

6.3.3 Second Level Modules

These modules are imported by the main module (described earlier) and represent the functional decomposition of the main module. The functional structure shown in Figs. 6.2 and 6.3 are implemented fully through routines imported from these second level modules. They, in turn, make use of lower level service modules. A brief description of these modules is given below (see also Fig. 6.4):

* INITIAL module.
* STARTUP module.
* STEADY module.

a) INITIAL module: This module is the first one called in by the main program module. Its main purpose is to initialise the hardware of the communication section.

b) STARTUP module: This module implements the second mode of ring construction, i.e entering the ring. It has four routines (see Fig. 6.4).

c) STEADY module: This module implements the third mode of operation, the 'run in operation' mode.

6.3.4 Service Modules

6.3.4.1 Control-Frame Modules

These are the highest service modules, most of the their routines being imported by the second level modules. Their main function is to initialise, construct, and maintain the token ring through a set of control-frame service routines. They are grouped functionally in separate modules, hence some precedence occurs in their call or even in their processing (compilation process, refer to section 6.5). Control-frame modules rely for operation on the lower modules 'Messages' and 'Mains'. A list of these modules is given below (see Fig. 6.4):

* SENDLIB module.
* RECLIB module.
* TIMER module.
* ROUTINES module.
* GLOBALS module.

a) SENDLIB module: This module contains an extensive set of routines, whose main function is to 'send' control frames across the network.

b) RECLIB module: This module is built in a similar way to SENDLIB. It encapsulates all the 'receive' control frames received over the system bus.

c) TIMER module: This module consists of various 'timer' operations. It has a set of routines for setting, loading, and polling the different timers of a station (see Fig.6.4).

d) ROUTINES module: This module contains a group of repeatedly used control-frame routines, used for implementing the token ring.

e) GLOBALS module: This module contains all the global variables and constants needed for the operation of the token ring. It is called (imported) by many of the above modules.

6.3.4.2 Message-Exchange Modules

These consist of two sets of modules, 'Messages' and 'Mains'.

a) 'Messages'

This module is dedicated to network communication activities. It consists of two routines; one to send a data frame across the system bus (to another station) and the second to receive a data frame from the system bus. To achieve this, use is made of the 'Signals' module for the control of the hardware.

i) TransmitMessage: This routine transmits a data message in the communication processor's memory at location 'Start', of length 'Duration', to the system station 'SystemAddress'.

ii) ReceiveMessage: This routine functions similarly to the above except that it receives a data frame from another station. This is stored in a specified address in the Scratchpad RAM.

b) 'Mains'

This module software is dedicated to support communication functions with the processing section. Its construction is similar to that of 'Messages'. The module consists of two procedures, 'MessageFromMain' and 'MessageToMain'. Again, use is made of module 'Signals' to control the hardware.

i) MessageFromMain: This routine transfers data messages from the processing section of each station into its communication section, using DMA techniques. Each message is stored, for subsequent bus transmission, in a temporary buffer (in the scratchpad RAM) at a specific address ('ScratchPadArea').

ii) MessageToMain: This routine is used to send data messages received from the bus to the processing section of the station using DMA transfer methods. Again, it is buffered in a temporary storage (Scratchpad RAM), at an address 'Start' and of length 'Duration'.

6.3.4.3 Hardware Related Module - 'Signals'

This module contains the initialisation routines for the communication section hardware. It also contains routines which control the transfer of information across the system bus by activating backplane (bus) control lines. A brief discussion of these routines is given below:

a) Start: This procedure returns the state of the system initialisation line (START*) as a boolean value. TRUE is returned when the system start line is true.

b) Rxen: This routine returns the state of the receive enable line of a station (RXEN*). This line is set TRUE by the communications hardware when another station has placed its address on the system address bus and activated the valid address line (SSS*).

c) TmsInt: This routine returns the state of the temporary memory module (TMS) interrupt latch. This latch is set by the interrupt line from the TMS module, but can only be set during a transmission.

d) MainInt: This routine returns TRUE if the main interrupt latch has been set, indicating that the processing section is requesting or stopping the transmission of data to the TMS module.

e) TmsWrite: This routine writes a block of data of specified length ('Duration') into the scratchpad area. The address of the data is given ('Start'). The address of the data in the scratchpad is specified also ('TmsAddress'). This routine also sets the SELECT line.

f) TmsRead: This routine reads a block of data, of a specified length ('Duration') into the communication section at a given address ('Start'). The address of the data in the scratchpad is also specified ('TmsAddress').

g) TmsWriteRegister: This routine is used to write the data passed to the TMS register specified. This routine also sets the priority of access line (SELECT) before attempting the write.

h) TmsReadRegister: This has a similar effect with respect to the above routine. It reads, however, from the specified register. The same condition is true regarding the priority line (SELECT).

i) Ready: This is the first of a series of routines to set individual hardware lines to a defined state. This state is determined in the boolean parameter passed to the procedure. Positive logic is used for all routines i.e TRUE sets the line active while FALSE resets the line.

j) Stx: This routine sets the state of the transmission process line (STX).

k) Wait: This routine controls the initialisation state, indicating whether the station is ready to start ring initialisation. Setting this routine FALSE indicates that the communication section is ready to start initialisation.

l) Select: This routine sets the priority access line for the scratchpad RAM into a specified state. When access is TRUE the communication section has access to the RAM, when FALSE the processing section has priority.

m) Saen: This routine is used to send a station address across the system address bus. The address specified is sent if the boolean variable is TRUE. If it is FALSE, however, this removes the address data previously written from the system address lines. During this action the data specified for the system address is disregarded.

n) StationAddress: This routine returns the address of this station, as set on the station address selector switches on the board.

o) ClearMainInt: This routine is used to reset the processing section interrupt request latch. Such interrupts are usually used to request the start or end of transmission of a data block from the processing section to the TMS module.

p) ClearTmsInt: This routine clears the latch holding a TMS interrupt request set to indicate the end of a transmission cycle.

q) DmaZero: This routine initiates a transfer between the processing section and the temporary memory module (TMS). When this routine is called the communication section relinquishes control of its bus. It can only regain control of the bus if the station is addressed by the system address lines, or the processing section sets the main interrupt latch to indicate the end of the data transfer. For this reason the main interrupt latch should be cleared before this routine is called.

r) DmaOne: This is an alternate routine which initiates transfer between the processing section and the TMS module. It functions in a similar way to the above routine.

6.4 IMPLEMENTATION OF THE RUN-TIME SUPPORT SYSTEM

6.4.1 General

Since the FTL compiler [6,7] is designed for a CP/M environment, it makes certain assumptions concerning its run-time environment. These did not apply to the actual target system. Thus, to enable code to run successfully and reliably within the communication sub-system, a special run-time module had to be developed (in addition to the system and application software). It is called 'CPM100'. In particular, the console device software had to be modified to handle compiler generated exception handling messages (these are automatically routed to the console).

6.4.2 CPM100 Module

This program is a CPM environment emulator, and is written in assembly language. Its purpose is to provide a basic initialisation sequence for the communication section. It provides routines to drive the serial interface as a replacement for the CPM console device. A replacement for this device had to be provided as code generated by the compiler outputs exception handling error messages, such as divide by zero, to the console. The communication software would have misfunctioned, and crashed at some point, if the emulation hadn't been provided, as it would expect certain initialisation and exception handling routines, necessary for its execution, within the environment. The CPM functions emulated by the CPM100 program are described in Appendix D. All other calls to CPM produce the message 'CPM ERROR' on the device attached to the serial interface.

The functions specified provide all the support required by the system module within the Modula-2 code and also the Terminal module, and any users of it such as SmallIO. They do not support RealIO unless the module is modified to drive the Terminal device, as described in the RealIO definition file.

The CPM100 program uses a small portion of RAM which must not be overwritten by any application program. This is the top 20H bytes of RAM available in the system. If this is not done then the stack will be placed on top of the CPM100; variables would then corrupt the stack and calls to CPM100 would fail to return to the calling program.

CPM100 also supports the use of a watchdog timer. This causes an NMI interrupt if the watchdog timer times out. When an NMI interrupt occurs a jump is made to a specific location. This causes a jump to the CPM100 start and the entire system is re-initialised.

CPM100 provides a faithful emulation of the functions mentioned above. It will also produce an error message if an illegal function is called. The more severe problem is if an application program makes use of some other part of CPM. If this occurs the results will be unpredictable.

6.5 SYSTEM DEVELOPMENT AND OPERATION

The communication software has been written mainly in Modula-2, using the FTL compiler. The ROMable code generated (i.e the executable main module) occupies approximately 16 Kbytes, and the assembler code (CPM100) occupies 256 bytes. Fig. 6.5 shows the memory map of the communication system.

6.5.1 Compiling and Linking

In the compilation process, the definition modules must be compiled in a specific order; starting with the lower level (service modules) and ending up with the higher level (functional modules). The order of these is as follows:

* SIGNALS.DEF
* MAINS.DEF
* MESSAGES.DEF
* GLOBALS.DEF
* SENDLIB.DEF
* TIMERS.DEF
* RECLIB.DEF
* ROUTINES.DEF

Then follows the functional modules:

* STARTUP.DEF
* INITIAL.DEF
* STEADY.DEF

The implementation modules can be compiled in any order once the corresponding definition modules have been compiled. When linking, the top-most level modules need to be specified only. Linking options must be specified so that the code generated can run correctly on the target system (see Appendix C for details).

6.5.2 Downloading into EPROMS

The linking process produces an executable file (a COM file). Before down-loading into the target system, the code has to be converted into INTEL HEX format.

To develop a program in FTL Modula-2, the code is written on the development system and tested as far as possible. It is then linked to start at 0100H. The available memory space is partitioned as appropriate for code and data management. The application code is then down-loaded into EPROM together with the run-time program CPM100. These two programs require no linking as the entry points for each are pre-defined. The CPM100 program is loaded at 0000H and the (Modula-2) application code (communication software) at 0100H. Using the standard CPM entry point of 0100H for the Modula-2 code solves one other potential problem. The utility supplied with the FTL compiler, UNLOAD2, would normally be used to convert the COM file into HEX format for subsequent transfer into the EPROM programmer. This automatically assumes that the code has been linked for 0100H entry and hence writes this address into the hex file produced later on for down-loading.

6.5.3 System Start-Up Operation

On power-up or reset the communication processor starts executing code from address 00000H. This is the start address for the CPM100 program and contains a jump into the application program. Thus the CPM100 starts executing before the main application program. The serial interface and the wait state generator are set up (by CPM100) before control is handed over to the Modula-2 application program (at address 0100H). This is the usual start for a CPM application program (refer to Fig. 6.5).

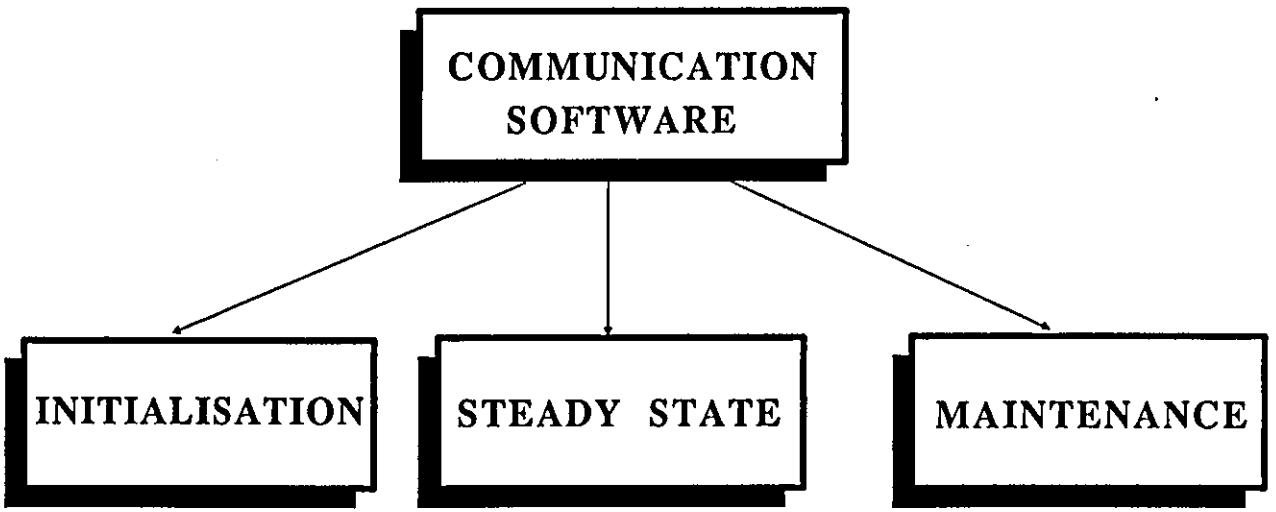


Fig. 6.1 COMMUNICATION SOFTWARE (NETWORK'S VIEW)

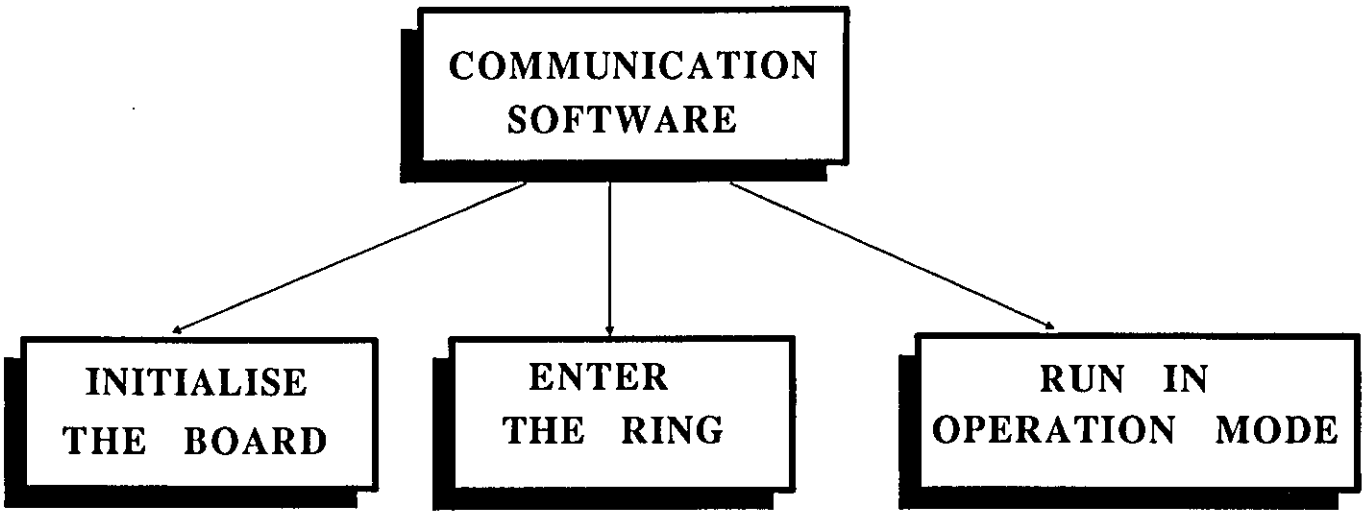


Fig. 6.2 COMMUNICATION SOFTWARE (STATION'S VIEW)

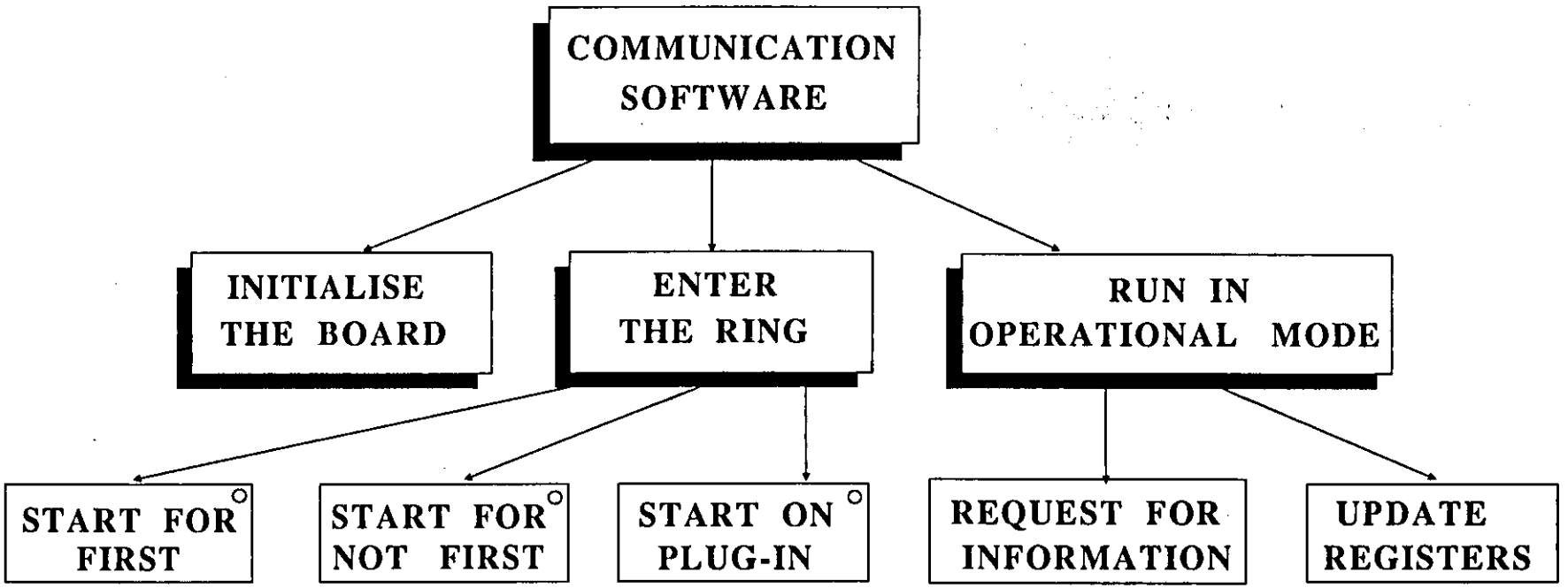


Fig. 6.3 COMMUNICATION SOFTWARE (STATION'S VIEW)

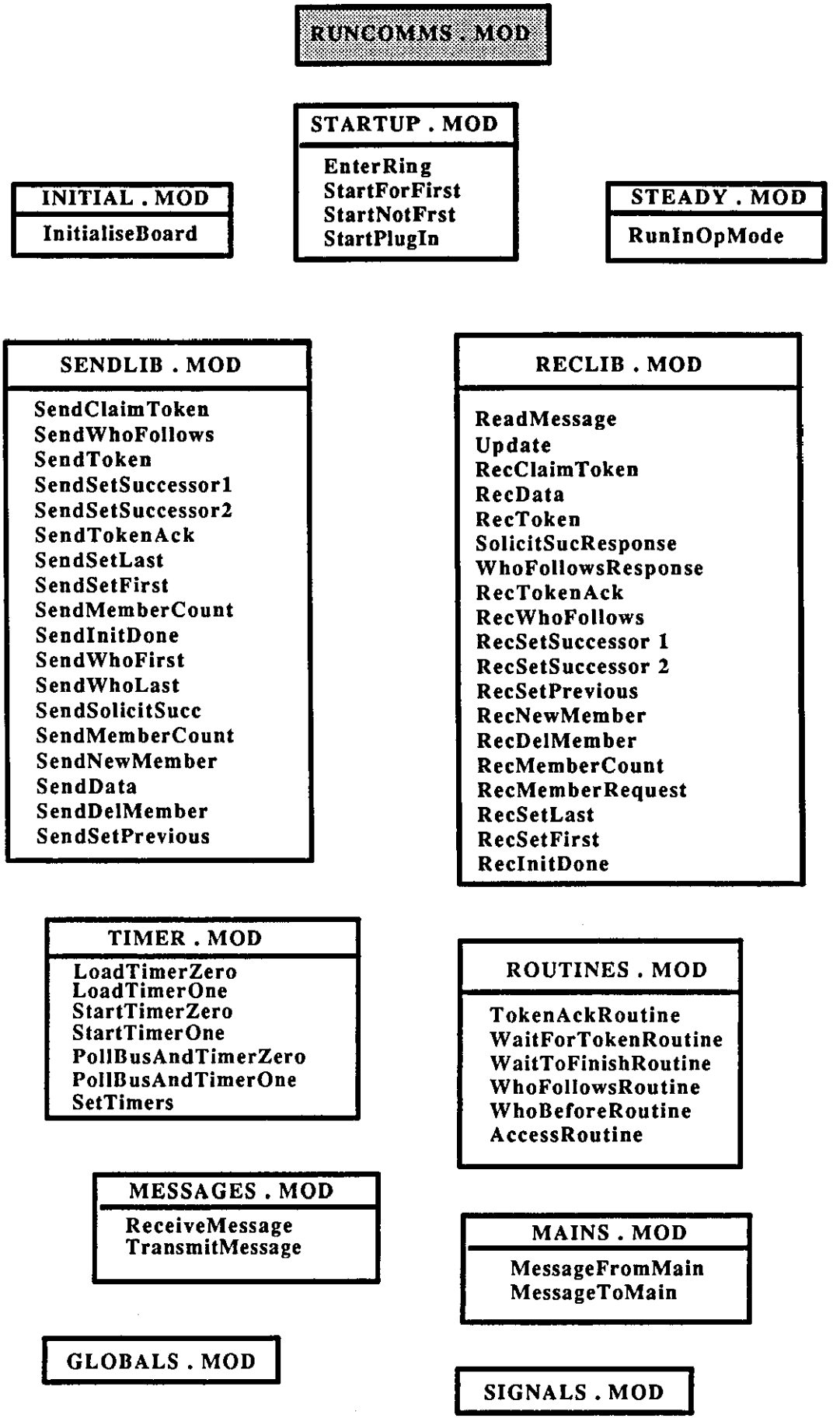


Fig. 6.4 IMPLEMENTED SYSTEM MODULES

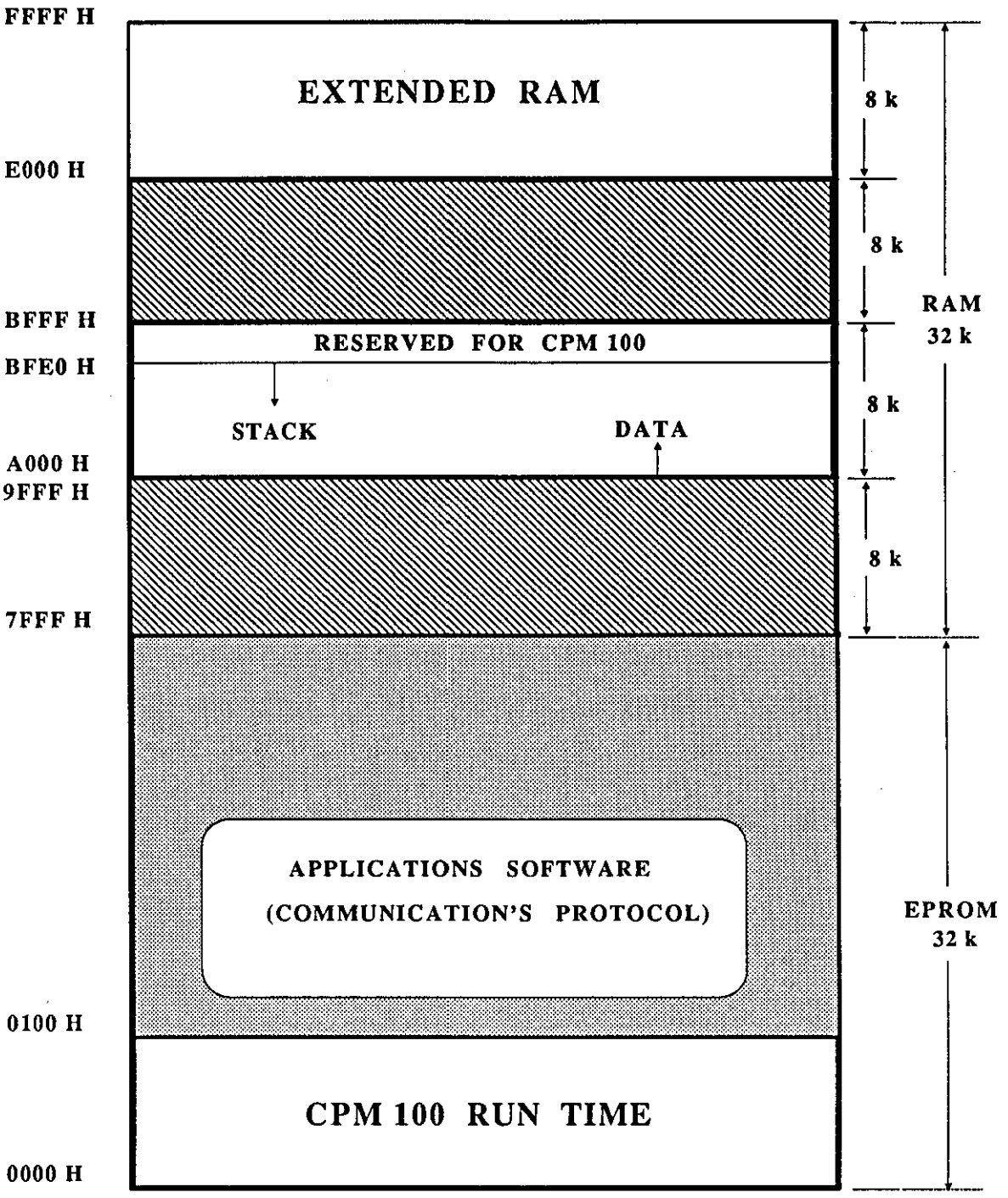


Fig. 6.5 SYSTEM MEMORY MAP

CHAPTER 7

CHAPTER 7

MULTI-PROCESSOR SYSTEM - KERNEL SOFTWARE STRUCTURE

7.1 INTRODUCTION

As stated earlier, the main function of the multi-processor system is to process and manage real-time application tasks that are functionally partitioned and distributed onto the system nodes as sub-tasks. The main functions of the various processing sections, at a detailed level, are to:

* Process and manage the time-critical sub-tasks in each node.
* Control data exchange with other processing sections through the use of message-passing techniques.
* Perform memory management of RAM (on buffered data, messages, and distributed variables).
* Provide management of timed/event interrupts within each node.

These tasks are implemented mainly in software through the use of a real-time kernel structure that supports and manages partitioned sub-tasks on various processing sections of the system.

The real-time kernel software is designed in a modular, structured manner, being implemented using the Jackson Program Design Facility (PDF) package. The core element of the processing section is based on an Intel 80188 processor together with an Intel 8087 numeric processor extension for mathematical operations. Programs for this are developed using the Logitech compiler, the application software being programmed into EPROM. A description of the program structure and the corresponding diagrams is fully shown in Appendix E.

7.2 THE REAL-TIME KERNEL STRUCTURE

In the design of the real-time kernel we are very much concerned with predictability of performance. Moreover, reliability of operation is paramount [1]. The kernel provides a virtual machine in which processes allocated to different processors are executed concurrently.

Process cooperation and synchronisation are achieved by means of message passing. On the other hand, the system inside each node is viewed as a collection of co-operating sequential processes that share common data. Unlike many scientific and commercial applications, the kernel described here is not intended to support fragmented programs. Instead, the basis of the design is that of functional partitioning [2]. Further, a major primary objective is to implement the kernel using standard compilers, i.e. those designed for uni-processor systems [3]. A second major objective is to build the kernel infrastructure using the standard constructs of Modula-2. It consists of the following structure:

* Program partitioning.
* Communication and synchronisation.
* Management of distributed variables.
* Process scheduling.
* Time-Server routines.

These have been designed to be independent of processor hardware.

a) Program Partitioning

The total system task is partitioned into a number of functional sub-tasks (processes); these run asynchronously and concurrently within the multi-processor system (Fig. 7.1). Functional partitioning is favoured over other schemes used for partitioning simply because:

* The software structures mirror the application structure, this being especially suitable for real-time application tasks.
* The individual software units (sub-tasks) can be implemented, type checked and compiled using uni-processor compilers.
* The granularity (unit of partitioning or sub-task), may be further divided and partitioned into other functional sub-tasks (see Fig. 7.1). These sub-tasks can be mapped, in turn, to one or more nodes of the distributed system.
* Finally, each sub-task can be considered as a unit sole of partitioning. This means it can be separately processed, coded, and compiled using structured languages suited or even adapted for distributed environments.

In real-time systems such sub-tasks involve plant interfacing, network control, computation of digital control algorithms, etc. Each sub-task forms the main process within a specific node.

b) Communication and Synchronisation

Inter-processor communication is implemented using message-passing primitives. In line with the overall strategy outlined above, inter-processor communication can be viewed in a 'client-server' model [4,5]. The message actually passes through several intermediary processors or subsystems (i.e. communication section, system bus, communication section, and finally, the receiving processing section).

Delays are experienced within the system at various points. Thus communication between the distributed sub-tasks is seen fundamentally as an asynchronous operation. Furthermore, it is non-blocking, i.e. the application software will continue after the message has been put out for transmission. The lower level communication aspects (physical, medium access control, logical link control) are considered to provide a highly reliable service. For improved levels of security, detection and correction of message errors must be implemented in the main process. In the same way, task synchronisation is the responsibility of the application programmer.

Data transfer between the processing and communication sections is done through two high speed DMA channels.

c) Management of Distributed Variables

Tasks communicate by passing control signals and data over the system bus. For simplicity each variable is defined to have a specific owner (sub-task). The owner is responsible for maintenance and updating of its own variables, and may export these - as required - to other processes (Fig. 7.2). The receiving process treats these as it would a value parameter; it can modify the copy but not the original. It can, of course, request updating of the original by the owner.

d) Process Scheduling

Because the overall system has already been partitioned, sub-tasks are likely to consist of only a few operational processes. In these circumstances a very simple and traditional approach to 'scheduling' is adopted. Each sub-task consists as a single background process (Fig. 7.3) - which runs continuously - and a set of timed and/or event driven processes. These are activated by hardware interrupt signals.

Setting priorities, disabling interrupts, etc., is the responsibility of the application programmer. By using this approach the time and complexity overheads of a real-time executive are avoided; moreover context switch times are minimised.

The communication handler is itself interrupt driven, activated only on receipt of incoming messages. It functions as a single thread sequential program, consisting of a set of mutually exclusive server procedures.

e) 'Time-Server' Routines

Apart from using timed interrupts for purposes such as setting control loops as mentioned earlier, it can be used to establish a sense of 'program-time'.

A certain activity may need to be activated after a certain time within a node or even after some elapsed time with respect to an activity or process in another node. Hence, to maintain this sense of 'program-time' across the system, the kernel provides a set of 'timed-interrupt service routines' in each node. These routines provide functions which can be used to establish a real-time clock. Synchronisation of local clocks is essential for future development. One way of achieving this task is to choose one clock as a master and update the others periodically with respect to this.

7.3 IMPLEMENTATION OF THE REAL-TIME KERNEL

7.3.1 Software Module Structure - Overview

The real-time kernel code is structured as a set of primitives, replicated, if necessary, on each node. It is implemented in a main module called 'MAIN-DISTKERNEL'. What follows is a list of primitives provided by the kernel:

* Init-MasterGlob
* Init-CopyGlob
* Request-Global
* Submit-Global
* Check-RecvData
* WaitFor-Data
* Validate
* Init-Send
* Send-Data
* Init-Receive
* Send-Receive
* SenMess-Setup
* RecvMes-Decode
* Set-Timer
* Set-ClockTimer
* Setup-Send
* Setup-Receive
* DMA-Stopped
* Send-Handshake
* Setup-DMAInterrupt
* Setup-TimerInterrupt

The module 'MAIN-DISTKERNEL' relies in its operation on a number of functionally grouped modules. These are:

* MAIN-CODE module.
* MAIN-DIST module.
* MAIN-BUILD module.
* MAIN-TIME module.
* MAIN-HARD module.
* T88InOut module.

Some of the kernel primitives are called within other higher level structured primitives to provide specific services. The various kernel primitives are functionally distributed to provide the following functions:

* Manage distributed variables within the network.
* Manage communication of one station (i.e. the processing section) with another.
* Manage message encoding and decoding within each station.
* Manage time services in each station.
* Manage hardware-access and set-up services in each station (i.e. hardware-related routines).

The function of each category and the related primitives are discussed in the following sections. For full details refer to Appendix E.

Finally, there is a 'Bootstrap' routine. This routine is not part of the real-time kernel. It is used, however, to set-up and initialise the system before transferring control into the application program. This module is written partially in assembly code and partially in Modula-2 code. It is discussed in a separate section.

7.3.2 Distributed Variables Management

Management of distributed variables across the network is implemented using a number of functionally grouped procedures. Before going further in the discussion, the following important points have to be clarified:

* A distributed variable is referred to as either a master or 'original variable'.

* A request of an original variable by another station is referred to as either a distributed variable copy or simply a 'copy variable'.
* A distributed variable can be exported (as a copy) to another process or station according to a request. This is treated similar to a value parameter.
* An 'original variable' cannot be distributed unless it is initialised, then calculated (or updated) according to a particular process in the application program.
* Similarly, a 'copy variable' cannot be accepted or received unless a variable is created and initialised in the requesting station.

Management of distributed variables is implemented in the following group of procedures:

* Init-MasterGlob.
* Init-CopyGlob.
* Request-Global.
* Submit-Global.
* Validate.
* Check-RecvData.
* WaitFor-Data.

These set of routines have the responsibility of initialising, maintaining, updating and exporting copies of variables which are held locally in a process within a station to others on request. The name and size of each variable has to be declared, by a call to the appropriate procedure, if the variable is to be shared across the network (i.e. distributed). Distributed variables are referenced by names specified by the user when the main module, MAIN-DISTKERNEL, is being called. These names are used by other stations' processes when

updates of the variables are sent across the network. A distributed variable, being used either as a master (i.e. original) or a copy, must be static in memory as it is declared to the module by its address and size. The type of the variable, however, is ignored. Routines here keep a list of variables which may need initialisation or even updating by another station. This data structure mechanism is transparent to the user. A description of these routines is given below:

a) Variable Control Block (VCB): VCBs are records used within the kernel module to hold information about the status of each distributed variable (i.e. whether original or copy). This includes the name of the variable, size, status (original, or copy), etc.

b) Init-MasterGlob: This routine is used to set up a variable control block for locally held variables (i.e. original). This means a control block for a variable calculated at this station and distributed subsequently to other stations on request.

c) Init-CopyGlob: This routine is used to set up a variable control block for a copy variable, i.e. to hold a copy for a variable that is calculated on a remote station.

d) Request-Global: This routine is issued by a requesting station for a copy of an initialised, updated original variable from a particular station.

e) Submit-Global: This routine is used to send a copy of an original variable, calculated by this station, to a list of requesting stations.

f) Validate: This routine works on both original and copy variables. It checks whether a particular variable has been calculated (i.e. updated) before starting to distribute requested copies across the network. Alternatively, it checks whether a distributed copy variable has been received before being used in a specific operation.

g) Check-RecvData: This routine checks received data, among a list of requested copy variables. If a request exists then the variable is validated and stored afterwards.

h) WaitFor-Data: This routine is called to wait for a requested copy variable until that particular variable is being calculated and hence distributed across the network. It relies on 'Validate' routine in executing this task.

7.3.3 Communication Management

Communication management between the different processing sections is implemented using message-passing techniques. The communication section in each station, however, provides a transparent interface for message transaction with the system bus. There are two modes of operations within each station i.e. the transmission and reception modes. A number of routines is used to implement the sequence of operations in each case. Those routines discussed here present the high level interface with the application program. Other, hardware-related, procedures are called within these procedures to implement the hardware interface (i.e. access and set-up of hardware).

a) Transmission mode

In this mode, the operation is viewed as non-blocking, i.e. the application software will continue after the message has been put out for transmission. The transmission mode of operation can be implemented using the following pair of routines:

* Init-Send.
* Send-Data.

i) Init-Send: This routine is called first when an attempt is made to transfer data to the communication section. It first sets-up a block of data then calls a hardware-related procedure (Setup-Send) to initialise a channel for a transmission mode.

ii) Send-Data: This routine is used to transfer message or data frames into other stations. First, it checks whether any DMA transfer is in progress. It then requests the communication section for data transmission (through the use of a hardware-related procedure 'Send-handshake').

b) Reception mode

Reception of a data message is performed as part of the 'Multi-process communication handler' which is an event-driven interrupt handler. The communication handler receives, decodes and acts upon message. The application program resumes execution afterwards. Two routines are used in reception mode:

* Init-Receive.
* Receive-Data.

i) Init-Receive: This routine is similar to 'Init-Send' mentioned earlier. It is used to set-up the channel for reception mode.

ii) Receive-Data: This routine checks whether a successful data transfer has taken place. If so, it reads the data block into a buffer and starts decoding.

7.3.4 Message Management

The procedures discussed here implement the code to build a message frame for transmission out of its constituent data segments. They are also used to split a received message into its constituent parts. These procedures also contain the type definition for all messages sent over the system bus, 'MessageType'. Two routines are used for these operations:

* SendMes-Setup.
* RecvMes-Decode.

a) SendMes-Setup: This routine is used to build or assemble a data frame out of its constituent parts. Frame type (data or message), address pointers, destination address, etc. form parts of the frame structure.

b) RecvMes-Decode: This routine works on a received message frame. It splits the message into its constituent parts (i.e. message or data frame, request or reply of a distributed variable, etc.). A transfer is subsequently made to an appropriate server when the decoding process is over.

7.3.5 Time Management

Time routines serve for two purposes;

* To set timers for timed interrupts (e.g. level and actuator loops of Fig. 7.3).
* To maintain the sense of 'program-time' across the system, hence establishing the basis for a real-time clock in each station.

To implement these tasks it uses two program-interfaced routines that call other hardware-related routines in turn. These routines are 'Set-Timer' and 'Set-ClockTimer'.

a) Set-Timer: This routine is used to load and set a timer for a pre-defined time. On time out, an interrupt occurs and a service routine (Timer-Proc) is called to service the interrupt.

b) Set-ClockTimer: This routine is used to establish the basis for a real-time clock. Once called, the routine sets a timer to call another routine (Timer-Proc) on every interrupt. This routine requires a parameter giving the number of milliseconds that should elapse between each interrupt and also the processor clock speed in MHz. It also requires the address of a routine to be called as part of the interrupt handler. This routine performs any counting that is required using global variables.

c) Timer-Proc: This routine is called on every interrupt to set-up the timer control registers and update counters.

7.3.6 Hardware-Related Routines

This set of routines comprise of the lowest level of routines, i.e. they form the interface with the hardware system. All data is exchanged between the communication section and the processing section using direct memory access (DMA) techniques. The DMA controller is located in the processing section and generates the required control signals (read, write, and chip select). Control of all data transfers resides with the communication section(DMA0 and DMA1).

Two DMA channels are utilised for data transfer. Channel 0 is used for DMA transfer from the processing to the communication section. Channel 1 handles DMA transfers from the communication to the processing section. A variety of procedures are used to initialise, set, and control DMA transfers and interrupts in the main section (these are discussed below). Finally, it should be pointed out that the buffers dedicated for DMA transfer must be declared global variables as they are handled by the DMA unit asynchronously. The main objectives here are to:

* Set-up and control channels in each station for transmission and reception of data.
* Set-up an interrupt service routine to handle an interrupt from an event interrupt (Multi-process communication handler).
* Set-up the interrupt service routines to handle interrupts from timed interrupts i.e. timers.

The following set of routines are used to implement the above:

* Setup-Send.
* Setup-Receive.

* DMA-Stopped.
* Send-Handshake.
* Setup-DMAInterrupt.
* Setup-TimerInterrupt.

a) Setup-Send: This routine is used to initialise and set-up the control registers of a DMA channel for a transmission mode.

b) Setup-Receive: This routine is used to initialise and set-up the control registers of a DMA channel for a reception mode.

c) DMA-Stopped: This routine is used to check whether a DMA transfer has been successfully finished.

d) Send-Handshake: This routine forms a handshake with the communication section. It sends a request-of-data signal (RDT) and an end-of-data signal (EDT) to the communication section at the start and end of a DMA transfer respectively.

e) Setup-DMAInterrupt: This routine is used to plant an interrupt in the vector address area. This address points to a service routine that is to be executed later on when an interrupt takes place.

f) Setup-TimerInterrupt: This routine is similar to 'Setup-DMAInterrupt' mentioned above. The vector type and priority, however, are different.

Other initialisation functions are required, these being part of the bootstrap routine. This is discussed next.

7.4 THE BOOTSTRAP ROUTINE

The function of the bootstrap loader is to initialise and set-up the system before control is handed over to the application program. It consists of two sections, an assembler part and a Modula-2 part. The reason for this is to use Modula whenever is possible. Modula-2 code is clearer, easier to understand, and is likely to be more reliable. It does mean, however, that two separate bootstrap files have to be produced for EPROM programming. It is imperative that the link between the two, a jump location, is set correctly. One EPROM is used to hold both the assembler and the Modula-2 bootstrap object code.

7.4.1 Assembler Routine

This routine starts first with the initialisation of the hardware system. It consists of the following procedures:

* Set-up the different segment registers (i.e. code, data, extra, and stack pointer registers).
* Set-up the appropriate memory partitions (i.e. upper chip select, lower chip select, middle chip select, etc.). It is essential to set-up the register data before execution of the application programs and memory management takes place.

A jump is then made to the Modula-2 initialisation routine.

7.4.2 Modula-2 Routine

This routine is located at the bottom of the boot EPROM. Its main function is to minimise the use of assembler for system initialisation. It consists of two main functions:

* Initialise serial line interface.
* Plant an interrupt return vector.

When the Modula-2 initialisation is over, a jump is made to the start of the application software.

7.5 SYSTEM DEVELOPMENT AND OPERATION

The real-time kernel software has been written mainly in Modula-2, using the Logitech compiler [6]. The code size generated is approximately 5 Kbytes. The bootstrap code (assembler and Modula-2 initialisation code) occupies less than 1 Kbyte. The ROMable code size depends, eventually, on the size of the application program implemented and the imported kernel routines. A 32 Kbytes EPROM is dedicated for this task. Fig. 7.4 shows the memory map of the processing system. Kernel primitive interactions, and operations within an application program are fully described in Appendix E.

7.5.1 Compiling and Linking

The module 'MAIN-DISTKERNEL' relies in its operation on a number of imported, functionally grouped, modules. In their compilation process, the definition modules must be compiled in a specific order: starting with the lower level (hardware-related modules) and ending up with the higher level (functional modules). The order of these is as follows:

* MAIN-HARD.DEF
* MAIN-TIME.DEF
* MAIN-CODE.DEF
* MAIN-BUILD.DEF
* MAIN-DIST.DEF

The implementation modules can be compiled in any order once the corresponding definition modules have been compiled. When linking, the top-most level modules need to be specified only. Linking options such as code and data segments must be specified so that the code generated can run correctly on the target system (code and data segments used are 9800H and 83H respectively).

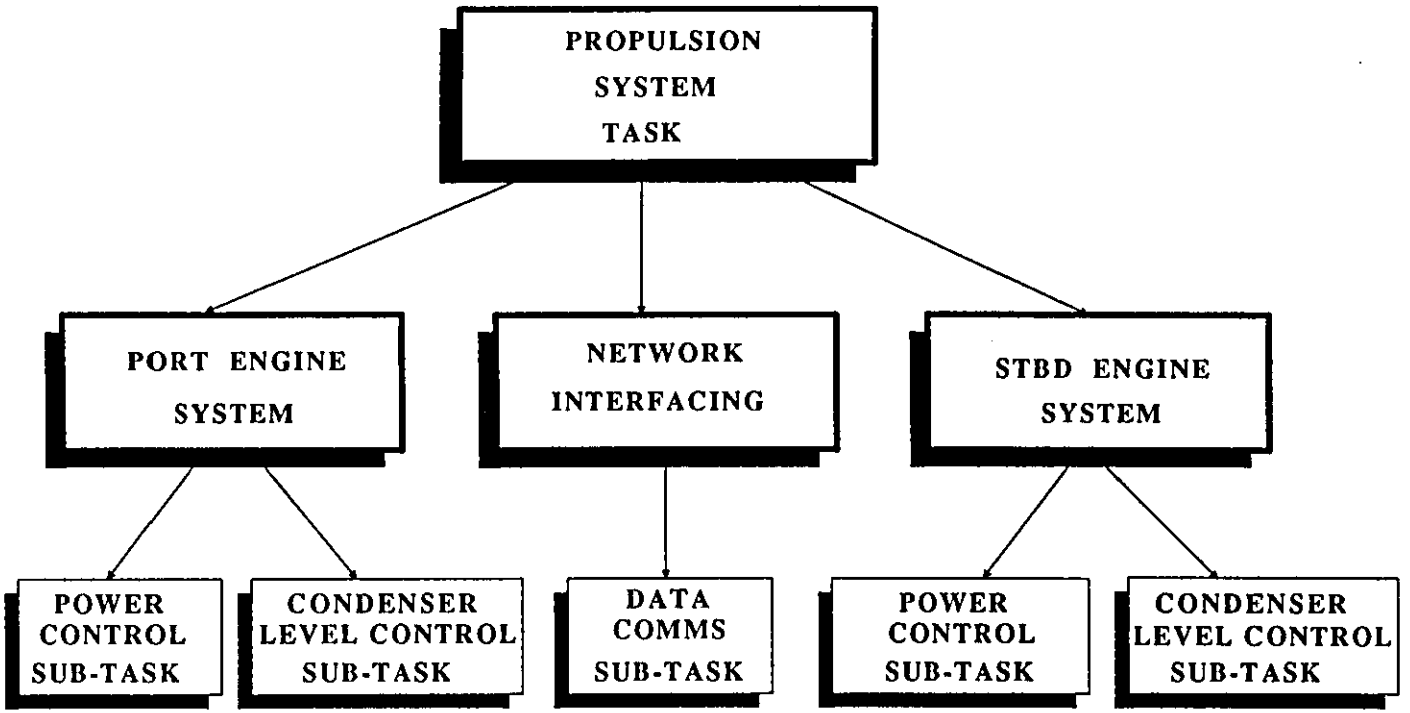


Fig. 7.1 FUNCTIONAL PARTITIONING

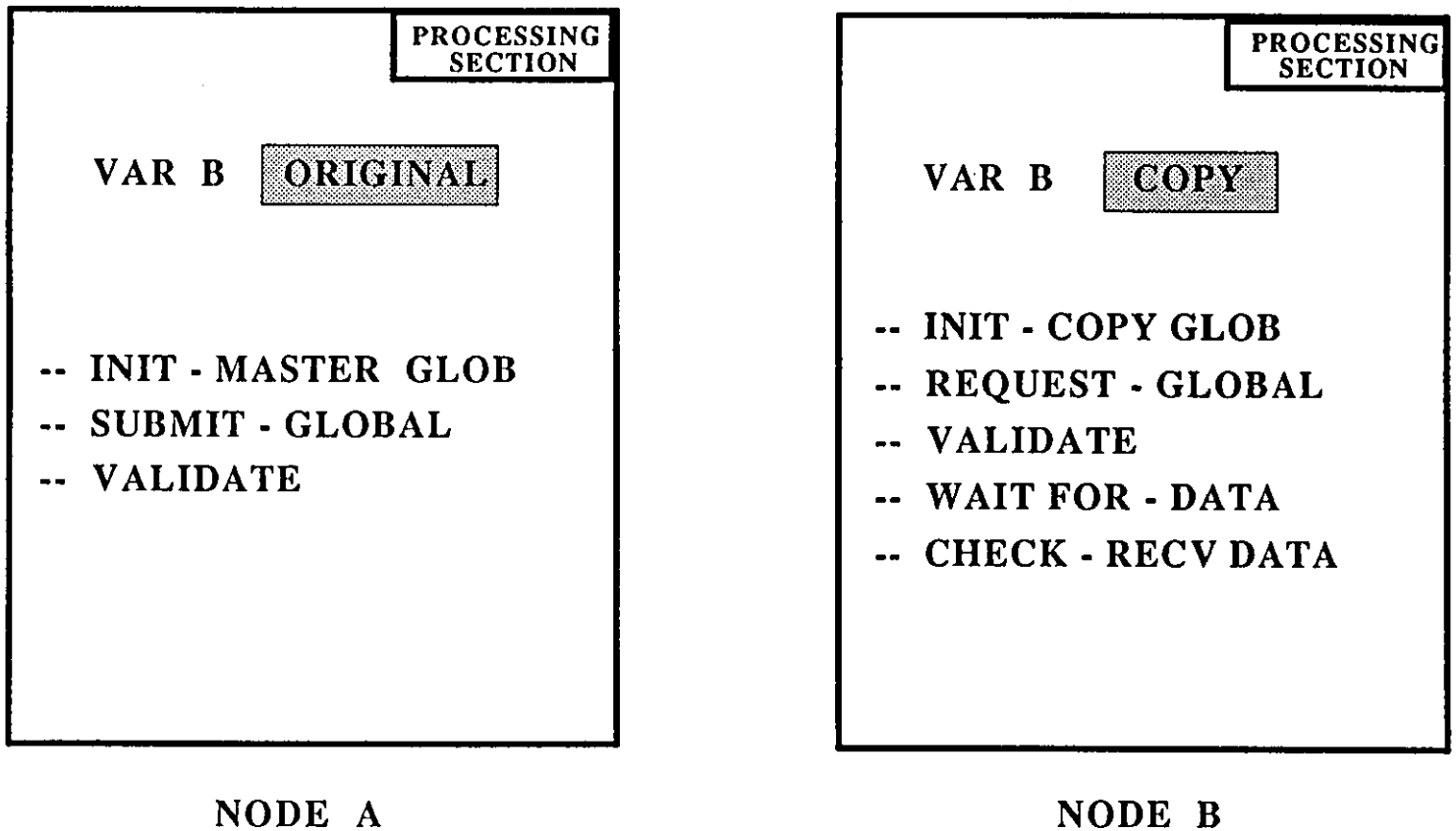


Fig. 7.2 DISTRIBUTED VARIABLE MANAGEMENT

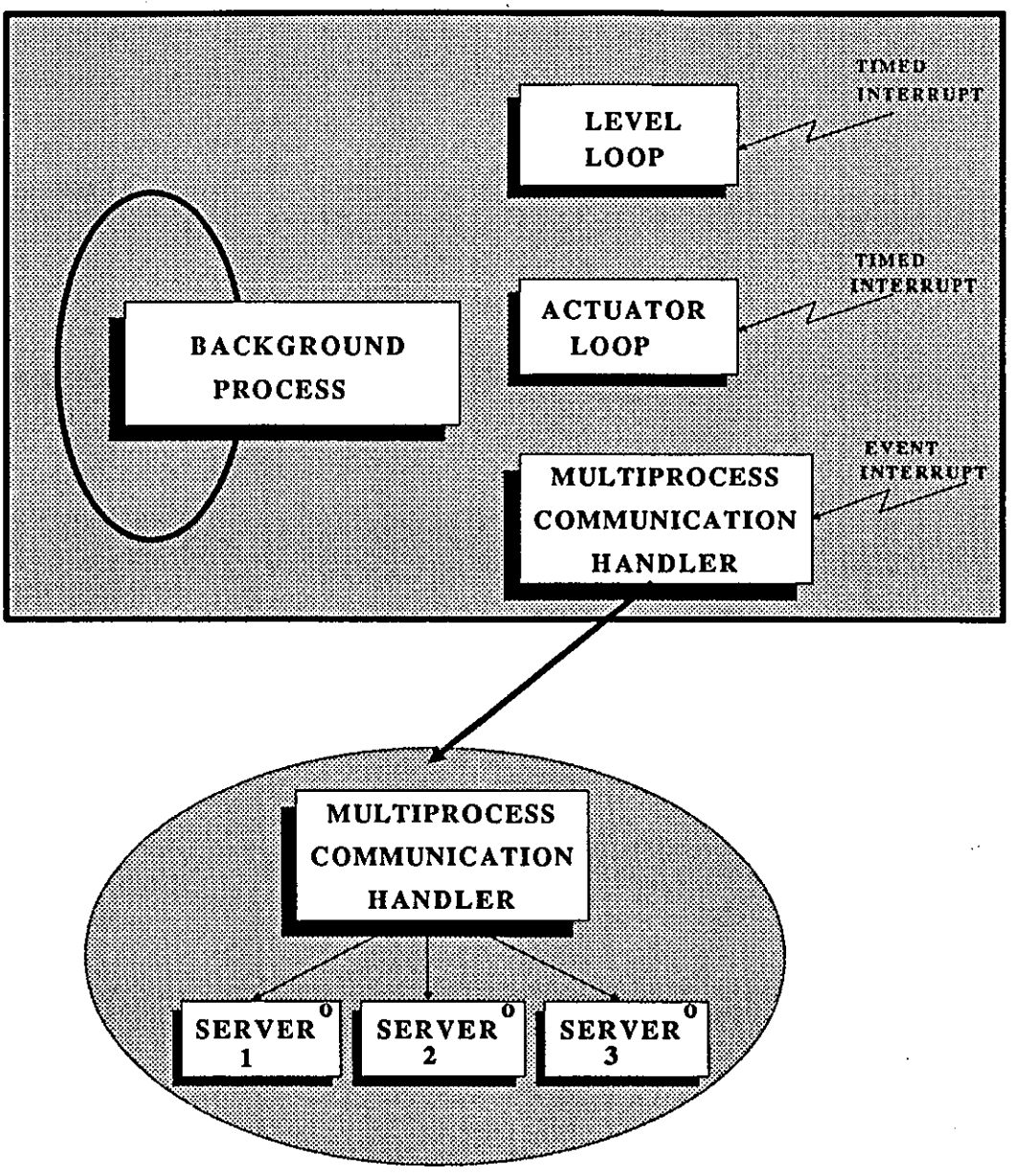


Fig. 7.3 FUNCTIONAL SCHEDULING

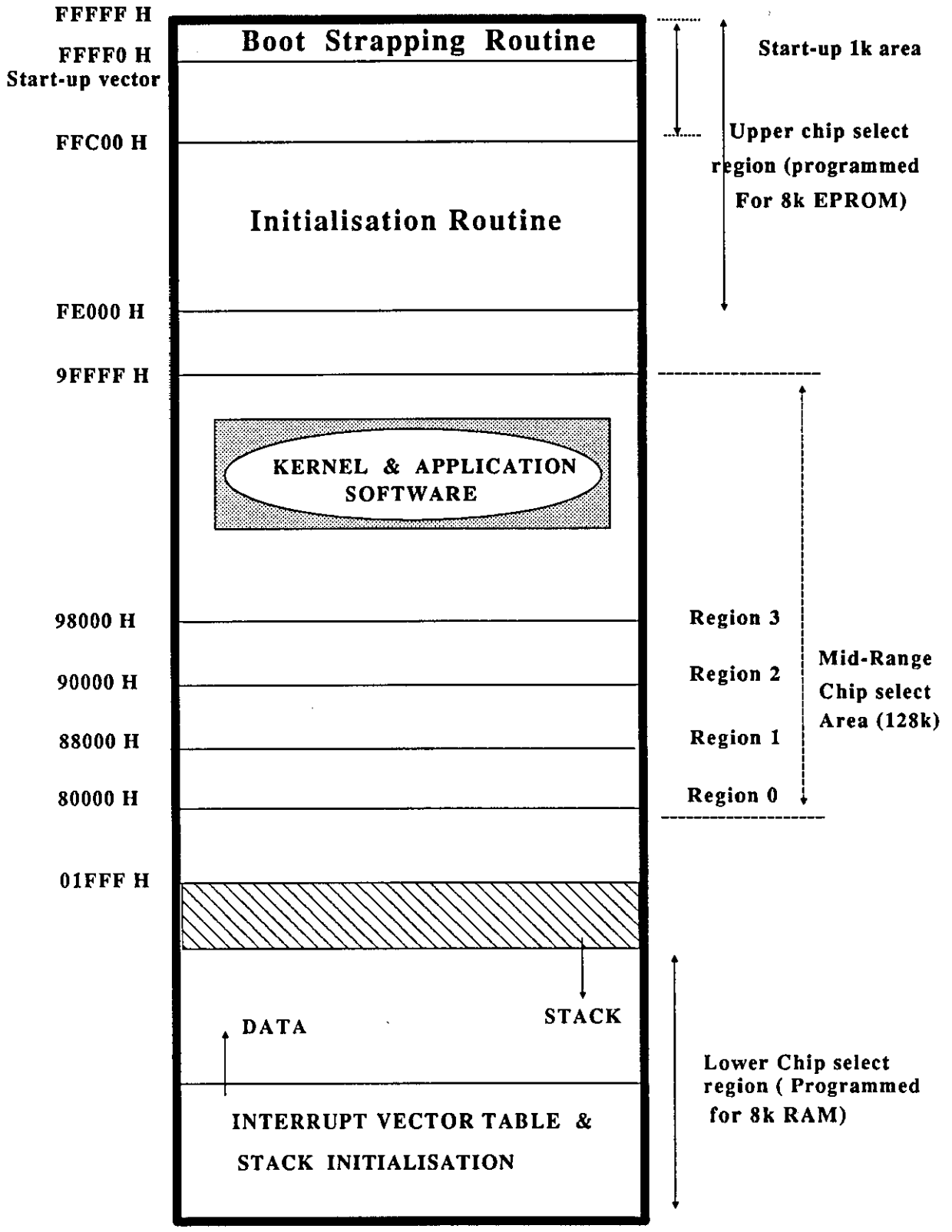


Fig. 7.4 SYSTEM MEMORY MAP

CHAPTER 8

CHAPTER 8

SYSTEM TEST AND VALIDATION

8.1 GENERAL

In this chapter a number of system test procedures relating to the process of system design and validation are discussed. Some of these routines validate the applicability of Modula-2 in such an environment.

The various test procedures are designed and implemented in a specific order, starting with the simplest procedures and ending up with the most sophisticated ones. This approach is important since the final test procedures depend upon the initial test results. These tests are organised in the following order:

* Processing Section - test procedures.
* Communication Section - test procedures.
* Overall System test.

8.2 PROCESSING SECTION - TEST PROCEDURES

In this section a variety of hardware test procedures are discussed. These were developed to support the maximum processor configuration, i.e. the combination of the 80188 cpu, 8087 math unit, and 82188 bus controller (see Fig. 8.1). Nevertheless, most of the programs (excluding the 8087 test programs) will run perfectly well on the minimum configuration, i.e. cpu 80188 only (see Fig. 8.2).

The main testing procedure consists of the following sections:

* Basic processor test.
* Chip select unit test.
* 80188 timer test.
* Serial line interface (DUART) test.
* SRAM test.
* DMA controller test.
* Numeric processor extension (NPE 8087) test.
* On-Board Interface (OBI) test.
* Initial Bootstrap test.

8.2.1 Basic Processor Test

The aim of this simple program is to check the operation of the following sections and signals (refer to Fig. 8.3):

* Address/data bus buffers.
* Upper memory block (EPROM).
* Single-step circuit.
* Signal buffers.

8.2.2 Chip Select Unit Test

The purpose of this test is to validate the operation of the chip-select unit and signals through accesses (i.e. memory and I/O read/write modes) to the different memory partition blocks.

8.2.3 Programmable Timer Test

In this test the different modes of the programmable timer are checked for proper functioning.

8.2.4 Serial Line Test (DUART)

This test is needed in the early stages of system testing as it is used in subsequent testing. This, when functional, allows test results to be displayed on a visual display unit (VDU). A list of the tests performed is shown below:

a) Auto echo-mode test.
b) Transmitter tests.
c) Receiver-Transmitter test (CPU registers access).
d) Receiver-Transmitter test (RAM locations access).

8.2.5 SRAM Test

In this test various memory search techniques are carried out to validate the access of memory blocks (SRAM). One of the routines is used to write a block of random data to a specified memory area. The data is then read back and compared to the data written. If the two data sets are different then an error message is displayed on the VDU terminal.

8.2.6 DMA Controller Test

This test checks the operation of the DMA controller. The DMA controller can be programmed to be activated internally (un-synchronised) or externally (synchronised). In this test both cases are evaluated.

a) Internally programmed DMA requests

DMA requests are programmed to be internally activated using the DMA control register. In this case, the output request lines (i.e. DRQ0 and DRQ1) are cleared. Internal triggering of the DMA transfer can originate with two sources, either from the DMA controller itself or from timer 2. Both cases are tested.

b) Externally programmed DMA requests

External DMA requests are activated by signals on output request lines (i.e. DRQ0 and DRQ1). The DMA registers are first loaded with the source and destination addresses and then with the appropriate control word. An external DMA request has to be simulated, however, for the DMA transfer to take place.

8.2.7 Numeric Processor Extension (NPE 8087) Test

The purpose of this test is to validate the operation of the 8087 processor and its interaction with the main 80188 CPU. The 8087 NPE is not a stand-alone processor, but functions as a co-processor with the 8086 family of microprocessors. It has a separate instruction set being inter-mixed with the host instructions as and where required. With such a configuration some mechanism is needed to synchronise the operation and interaction of the NPE with the main processor. When an Intel 80188 is used as the main processor (as in this case) an 82188 bus controller provides the synchronisation mechanism.

Two cases make it necessary to synchronise the execution of the main processor to the NPE:

a) An instruction that is to be executed by the NPE must not be started if the execution unit of the NPE is still busy executing a previous instruction.

b) The main processor should not execute an instruction that accesses a memory operand being referenced by the NPE until the NPE has actually accessed the location.

Test programs are implemented successfully to achieve both conditions above. Numerical processing is carried out by the NPE on data segments already stored in the main processor. Results are then stored in main memory.

8.2.8 On-Board Interface (OBI) Test

The purpose of this routine is to test the on-board interface (OBI) block. This is accomplished through message exchange using DMA transfer. These transfers are requested externally using lines DRQ0 and DRQ1. The Serial communications facility is used to provide a display of transmitted and received messages. Two main routines are implemented:

a) Transmit-mode routine

This routine is used to transmit messages from the processing section to the OBI block. Messages may be requested through VDU keyboard.

b) Receive-mode routine

The purpose of this program is to transmit messages from the OBI back to the processing section.

8.2.9 Initial Bootstrap Test

Prior to constructing the full bootstrap loader, mentioned earlier in Chapter 7, tests were carried out to validate the suitability and applicability of Modula-2 programs in such an environment (see Fig. 8.4). These routines are located in the upper 8 Kbytes EPROM, subsequently dedicated for the bootstrap loader. The test procedure is as follows (see Fig. 8.4):

* Jump to upper 1 Kbyte area.
* Set-up the various segment registers (i.e. code, data, stack, etc).
* Set-up the memory partition required.
* Initialise the relocation register for memory map of control block (once this is done, the control block can be accessed and programmed using memory-referenced instructions).
* Jump to the start of the upper 8 Kbytes EPROM area.
* Start executing the Modula-2 test routines.

8.3 COMMUNICATION SECTION - TEST PROCEDURES

This section deals with the testing of the communication section hardware and the verification of the design (Fig. 8.5). The various elements of the design are split into their small constituent parts. These are, then, tested individually. The main testing procedure consists of the following sections:

* Simulation of the hardware operation.
* PCB checking.
* Software Testing.

8.3.1 Simulation

Simulation processes in the EPLD design package, Altera, facilitate hardware design verification. This simulation facility enabled the function of the CSM module to be tested before any hardware was constructed. The simulator provides only functional simulation. These results are used during the implementation process to validate the design requirements. The timings for the CSM module are provided in Appendix A.

PHOTO 1: THE ALTERA EPLD DESIGN PACKAGE USED FOR DESIGNING THE 'CSM' MODULE OF THE COMMUNICATION SECTION

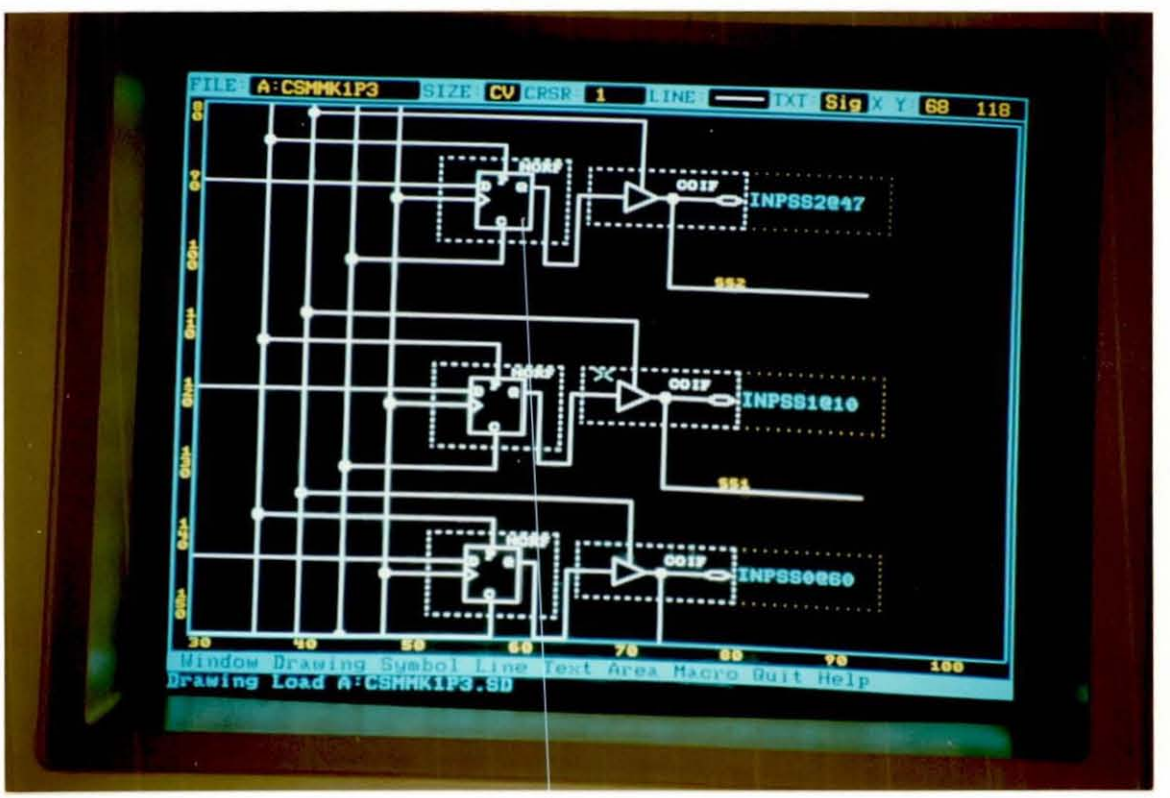
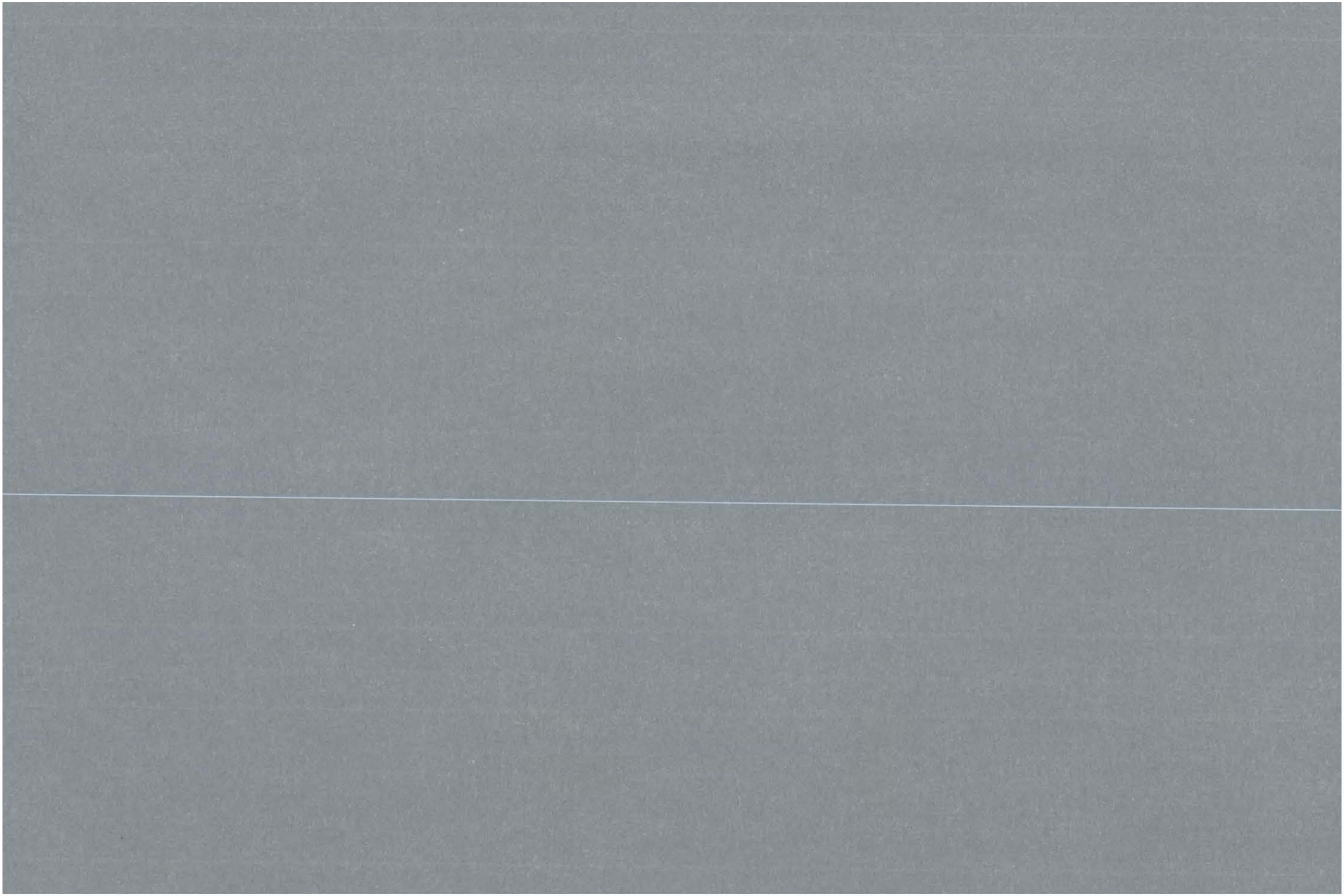


PHOTO 2: A HARDWARE SCHEMATIC ENTRY PROCESS USING LOGICAPS - A UTILITY WITHIN THE ALTERA PACKAGE USED FOR HARDWARE DESIGN



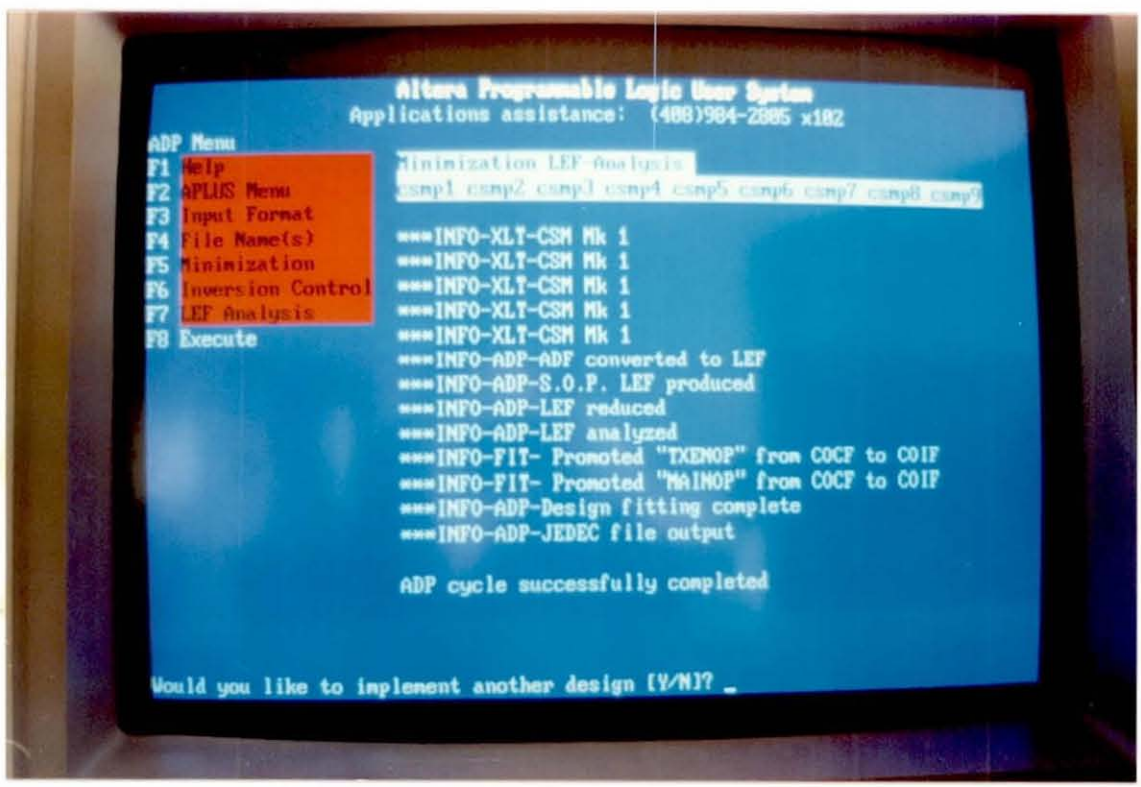


PHOTO 3: A DISPLAY SHOWING THE COMPLETION OF THE HARDWARE DESIGN PROCESS OF THE VARIOUS SECTIONS OF THE 'CSM' MODULE

To simplify the test process, tests are divided into several functional blocks. Although the module is simulated as a unit, this division simplifies the understanding of the test results.

The following functional simulation tests are carried out (for a full description refer to Appendix A):

* Reset test.
* Bus control test.
* DMA control transfer test.
* Receive mode test.
* Transmit mode test.
* Data read test.

a) Reset test

For every simulation run, the CSM module has to be reset before any active simulation takes place. This effectively simulates the action of the RESET* signal. The simulation consists of applying the RESET* signal for a number of cycles and checking all signals then settle at the correct defined state. Various other inputs have to be specified (e.g. the processor bus control lines).

b) Bus control test

This test is designed to check the operation of the CSM module interface, that is the data driving circuitry for the processor data bus. Operations of the CSM module like data latch and data read are checked here.

c) DMA control transfer test

This test checks the operation of the latches responsible for the transfer of control of the processor data bus during a processing section transfer. The triggering action is checked first. This involves checking the operation of the latch requesting bus control from the communication processor. Next, the two possible methods for re-gaining bus control are tested. These being either an interrupt from the processing section or a reception of the station's address by the system bus.

d) Receive mode test

This test simulates the action of a message reception. This starts with the station address being applied to the system address lines. An RXEN* is generated then. The module under test then generates a BUSY* signal until a write from the processor sets the READY line. At this stage a transfer of several bytes is simulated. Finally, station address is removed from the system bus.

e) Transmit mode test

This is the longest test which simulates the action of a message transmission across the system bus. The first step is a write operation by the processor to place the destination address on the system address lines. Then, a transmission is started when the simulator releases the BUSY* line. Transmission is ended by the simulation of an interrupt signal from the TMS module. This has the effect of halting the transmission of data and also causing an interrupt to the processor.

PHOTO 4: A DISPLAY OF INPUT/OUTPUT SIGNALS OF ONE OF THE FUNCTIONAL SIMULATION TESTS IMPLEMENTED USING THE 'ALTERA' PACKAGE

f) Data read test

The final test checks the rest of the module sections. This includes the address recognition, and reading of interrupt status registers.

8.3.2 PCB Checking

After thorough inspection and finalisation of the hardware design and simulation, a decision was made to build a PCB of the communication section. The PCB was laid out manually and then entered into the package (i.e. the 'Computamation system') for photographic quality artwork production. It would have been possible to introduce automatic checking of the design if this has been required. This is not attempted for two reasons:

* The highest level of checking would have involved the entry of a description of the design, either in schematic or net list form. The production of which may take a very long time due to the very specific requirements of the package.
* The lower level would have checked for physical violations, e.g. track spacing.

PCB package entry is performed in a logical fashion, based on entering functional groups of signals simultaneously. When the art design is entered, a multi-layer plot is produced for checking. This is done at a large scale (3:1) to aid the inspection of track clearances. At this stage both the physical and electrical routings of all tracks are checked.

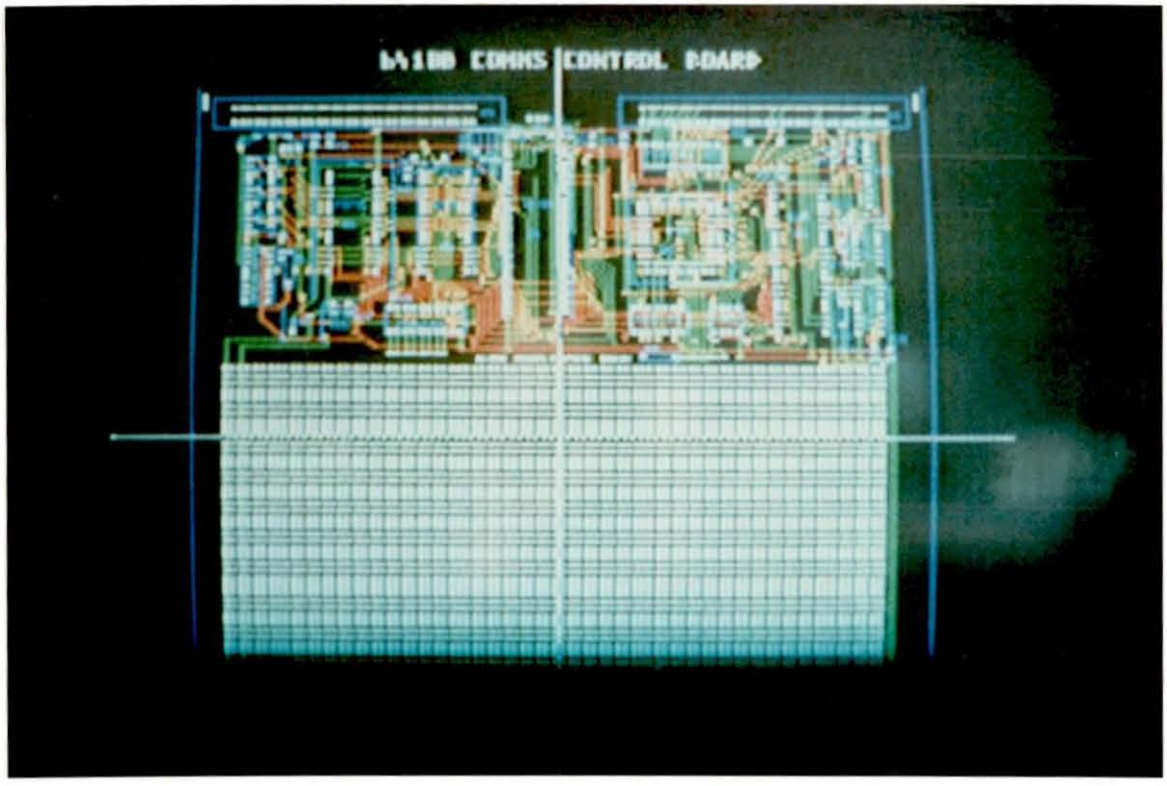


PHOTO 5: A COMPUTER AIDED DESIGN (CAD) PROCESS USED FOR GENERATING A 'PCB' LAYOUT FOR THE COMMUNICATION SECTION OF A STATION

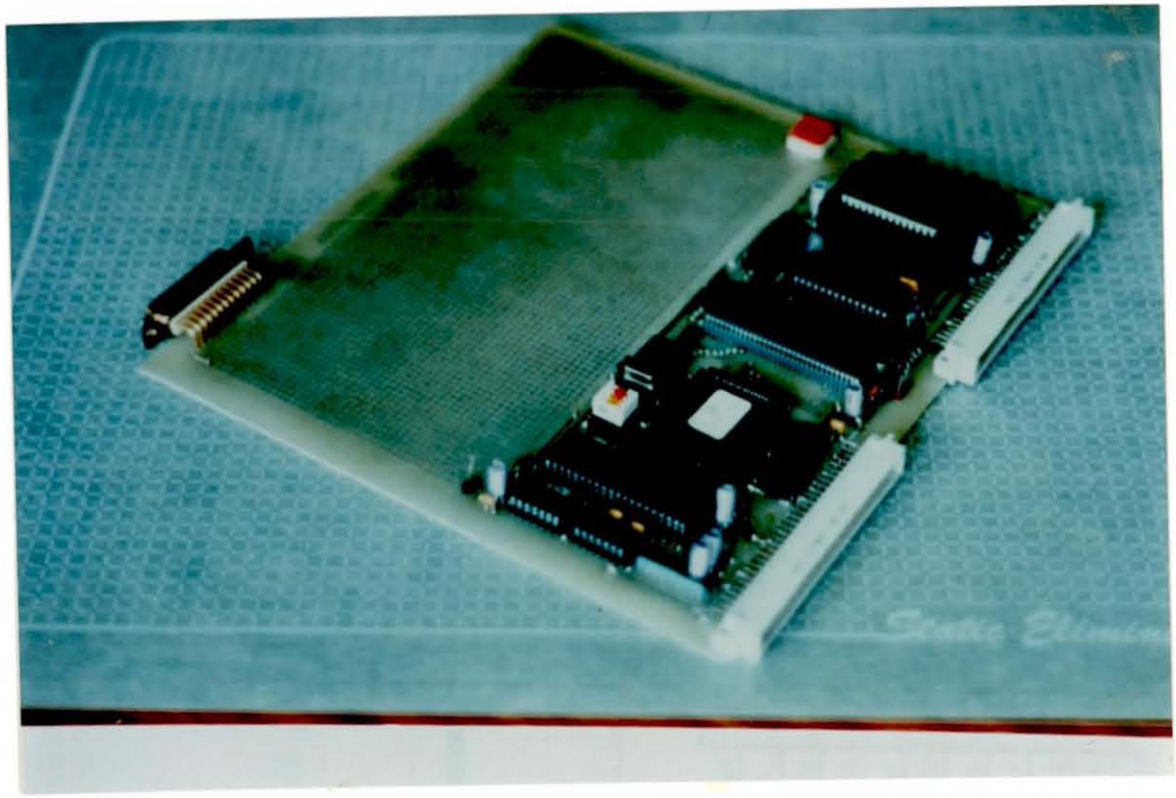


PHOTO 6: A PRINTED CIRCUIT BOARD (PCB) VERSION OF THE COMMUNICATION SECTION OF A STATION

8.3.3 Software Testing

This section consists of a set of routines which are used to check the function of the various blocks of the communication section, as follows:

* Initial test routines.
* CPM100 test routines.
* Linetest routines.

a) Initial test routines

These are similar to the test routines used in the processing section. They are established to test the processor and peripherals for proper functioning. These include:

* Basic processor test.
* Serial line interface.
* SRAM test.

b) CPM100 test routines

In this stage, it is decided to use Modula-2 in the testing process. To achieve this, the CPM100 routine had to be written and subsequently tested to support the Modula-2 code.

To check the CPM100 functioning, a program is written to test all the functions of the CPM100 emulation. This program is first tested on a CPM system and then on the target system. This is to ensure that the output is the same in both environments.

c) Linetest routines

Once the CPM emulation is successfully tested, the remaining test programs are written totally in Modula-2. The 'linetest' routine is a general purpose, menu driven, test routine that checks access modes (i.e. read or write operation) of various peripherals. The available functions are:

* Write to Memory.
* Read from memory.
* Test a block of memory.
* Output to I/O device.
* Input from I/O device.
* Start watchdog timer.

All the above tests are similar to those described earlier in the previous section (section 8.3.3-a), except for the watchdog timer test. The purpose of this test is to check the watchdog timer operation. This routine sets up a DMA transfer to start the watchdog timer. When an NMI interrupt occurs, the processor resets the section totally. This action is handled by code in the CPM100 routine.

8.4 OVERALL SYSTEM TEST

This section involves testing the various functions of the station as part of a system, i.e. its interaction with other stations. Various demonstration tests (called here 'Demo' tests) are established, all being written in Modula-2. What follows is a brief description of these demos:

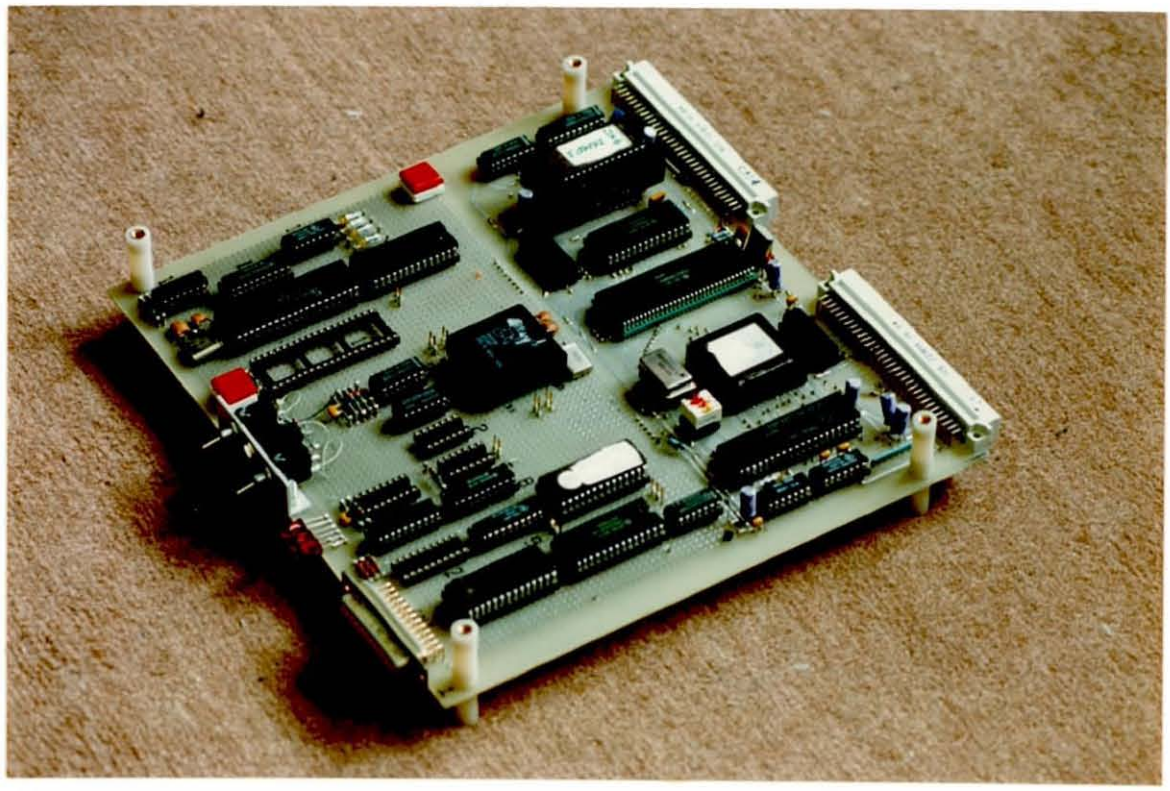


PHOTO 7: A COMPLETE WORKING STATION (NODE) OF THE NETWORK - CONSISTING OF PROCESSING AND COMMUNICATION SECTIONS

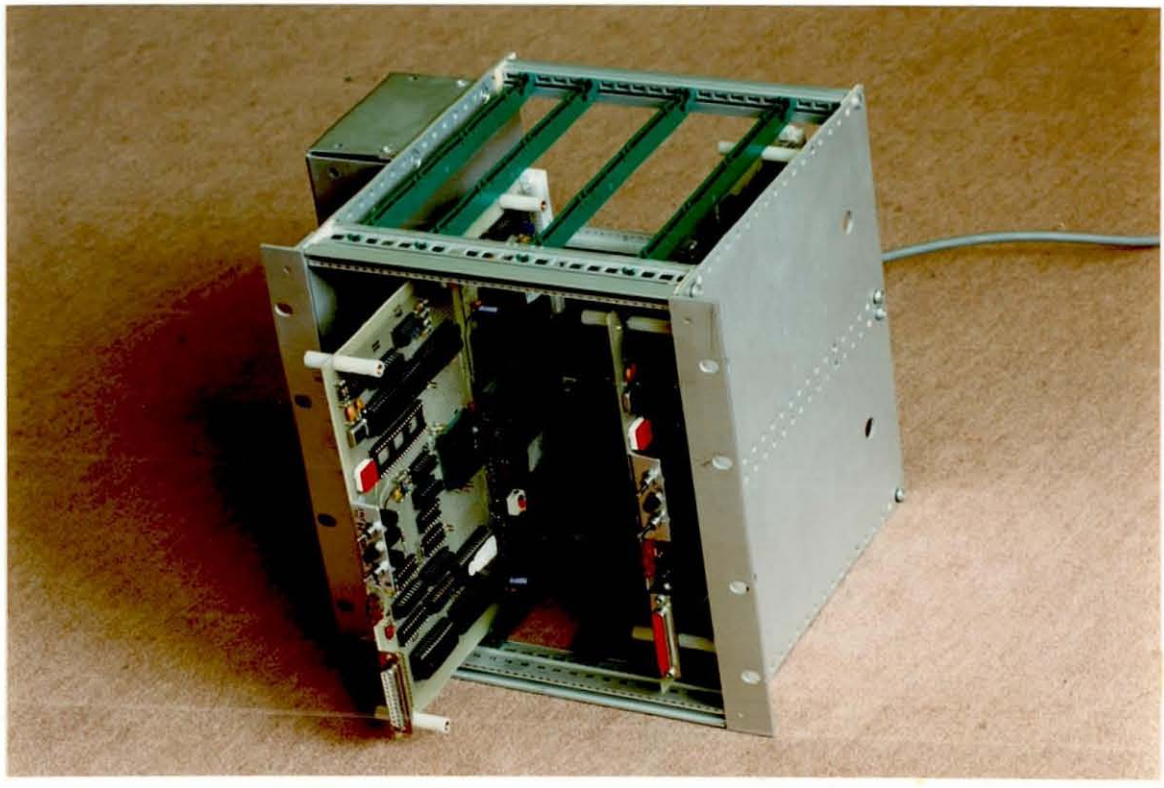


PHOTO 8: THE MULTI-PROCESSOR DEMONSTRATOR SYSTEM DEVELOPED AS A TEST RIG

a) Demo one:

This test validates the following functions:

* Message-exchange within the same station (i.e. between the communication and the processing section).
* Message-exchange within the network (i.e. with other stations).
* Test and set-up the various hardware control signals in the communication section of a station.

This demo is initiated and controlled from a keyboard/display interface, using menu driven facilities. This gives access to all routines supplied by the demo module, being implemented at two levels:

* Upper level.
* Lower level.

i) The upper level is used to test for the correctness of message transmission. When a menu is selected for transmission, the destination and message form are asked for by the program. The program continually monitors the system bus when waiting for an input. If it detects a message addressed to the station, it receives the message then display on the VDU terminal subsequently. It then sends the message back to the calling station, as an acknowledgement. This acknowledgement is, again, displayed as an incoming message by the transmitting station. An excessive use is made of the kernel and protocol routines in this level.

ii) The lower level, selected by the upper menu, provides a direct access to the routines in the 'Signals' module. These procedures enable any of the hardware lines to be set, reset or tested. It also enables blocks of data to be transferred into RAM locations.

b) Demo two:

This test demonstrates the communication and message-passing between the various stations of the network using the token passing bus access method (TPBAM). The demo is constructed using three network stations. The various stages of the token bus construction, described earlier in Chapter 6, are shown clearly in this demo. This includes:

initialisation process (i.e. initialising the different boards), entering the ring (i.e. start for first, start for not first, and start on plug-in), and finally running in operation mode. Message-frame exchange is shown clearly in this demo. All messages are displayed on various VDU terminals.

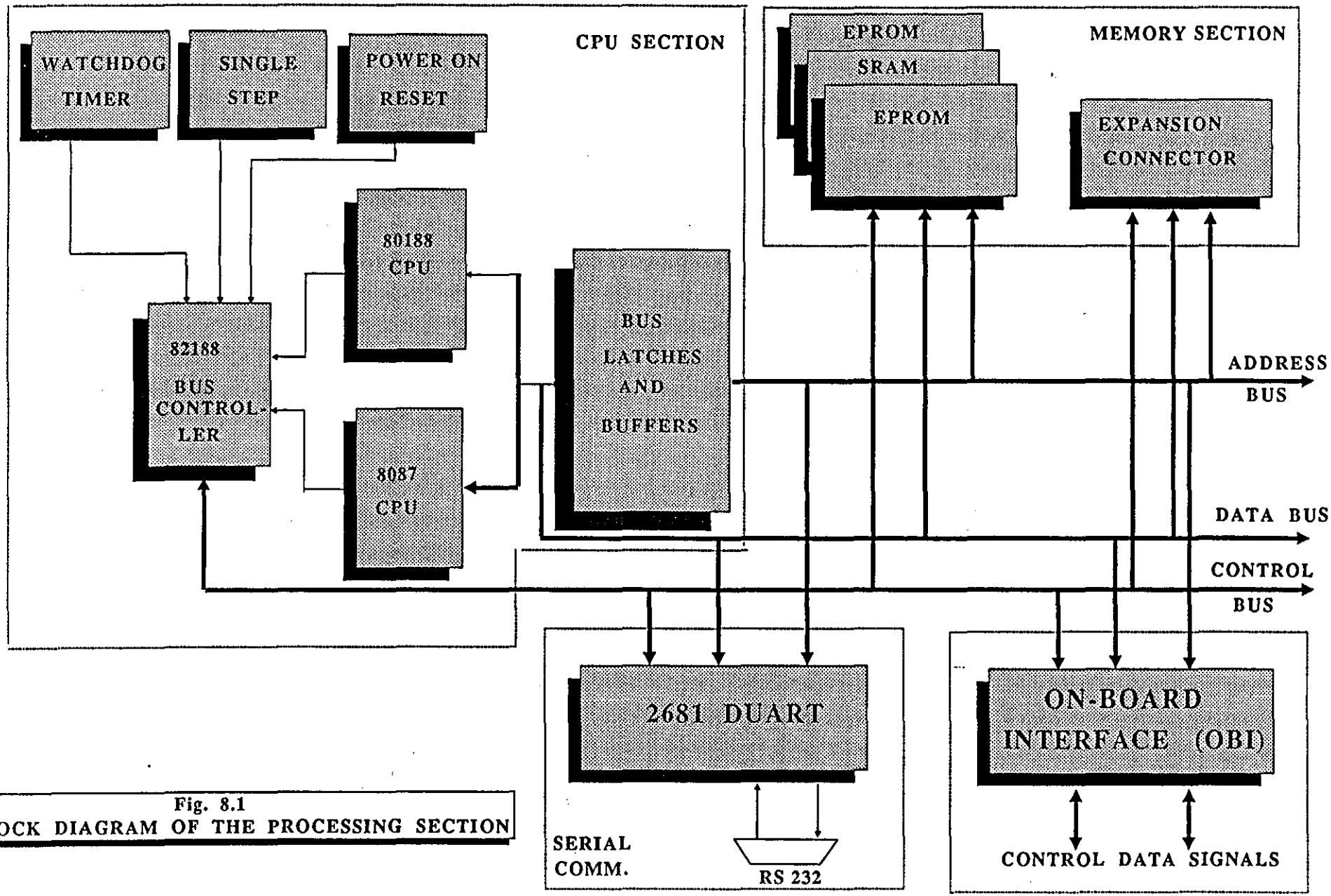


Fig. 8.1
BLOCK DIAGRAM OF THE PROCESSING SECTION

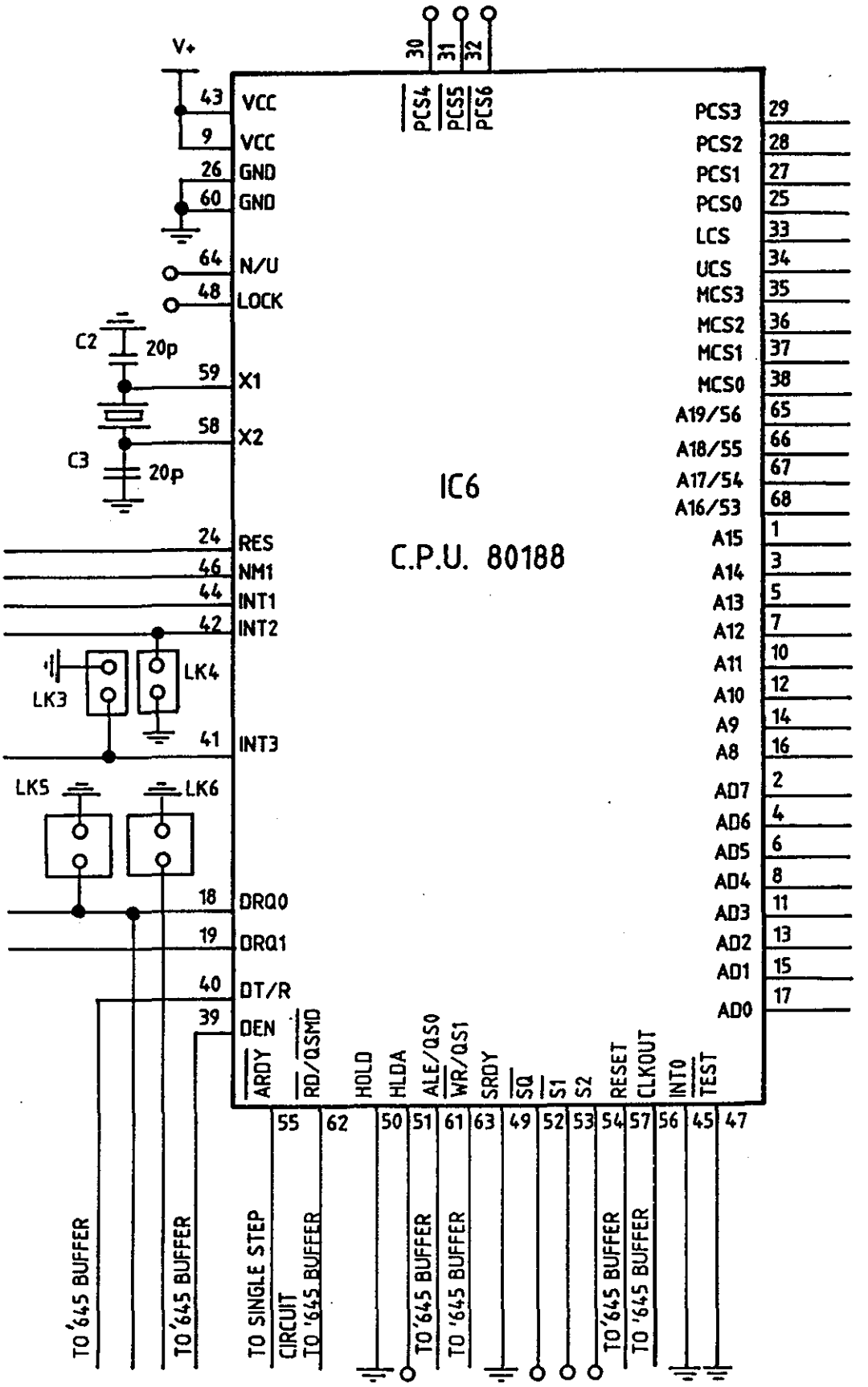


Fig. 8.2 MINIMUM CPU CONFIGURATION

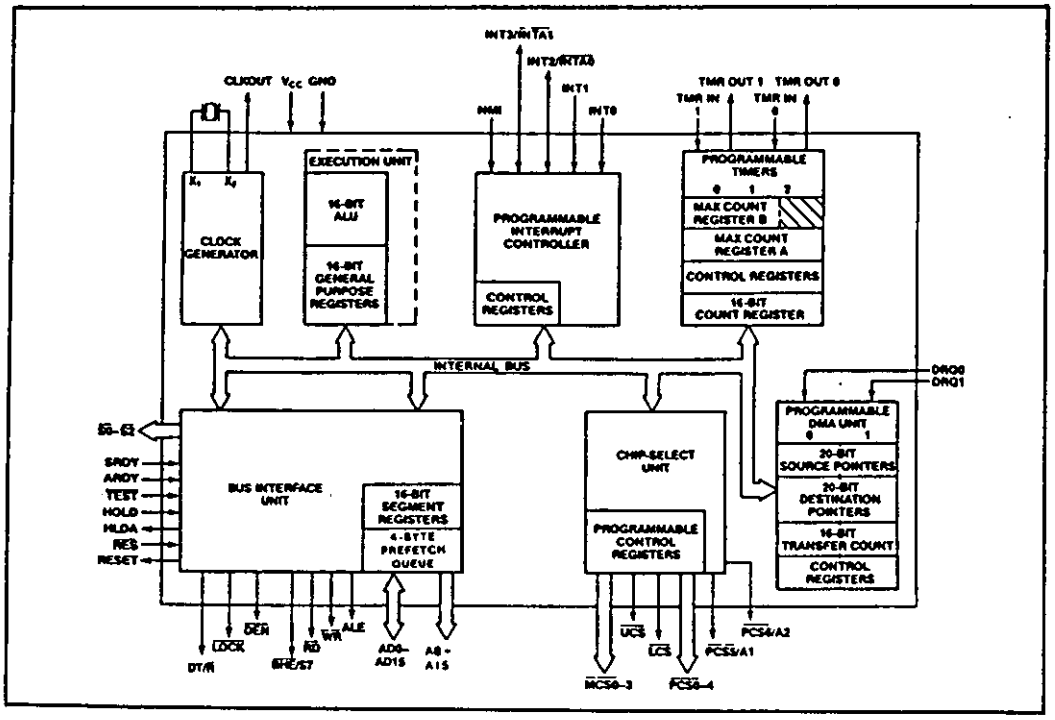


Fig. 8.3 80188 CPU BLOCK DIAGRAM

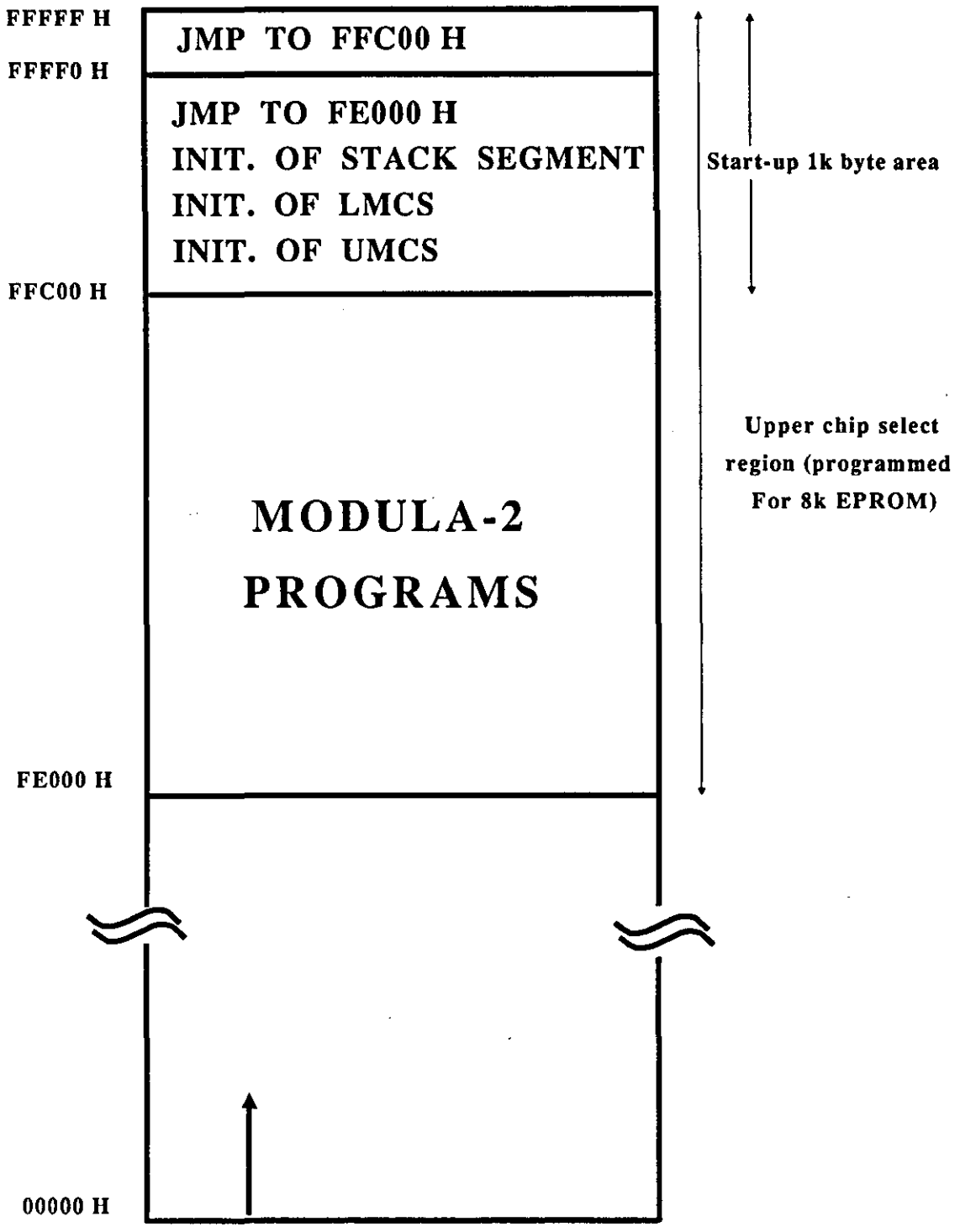


Fig. 8.4 INITIAL SYSTEM MEMORY MAP

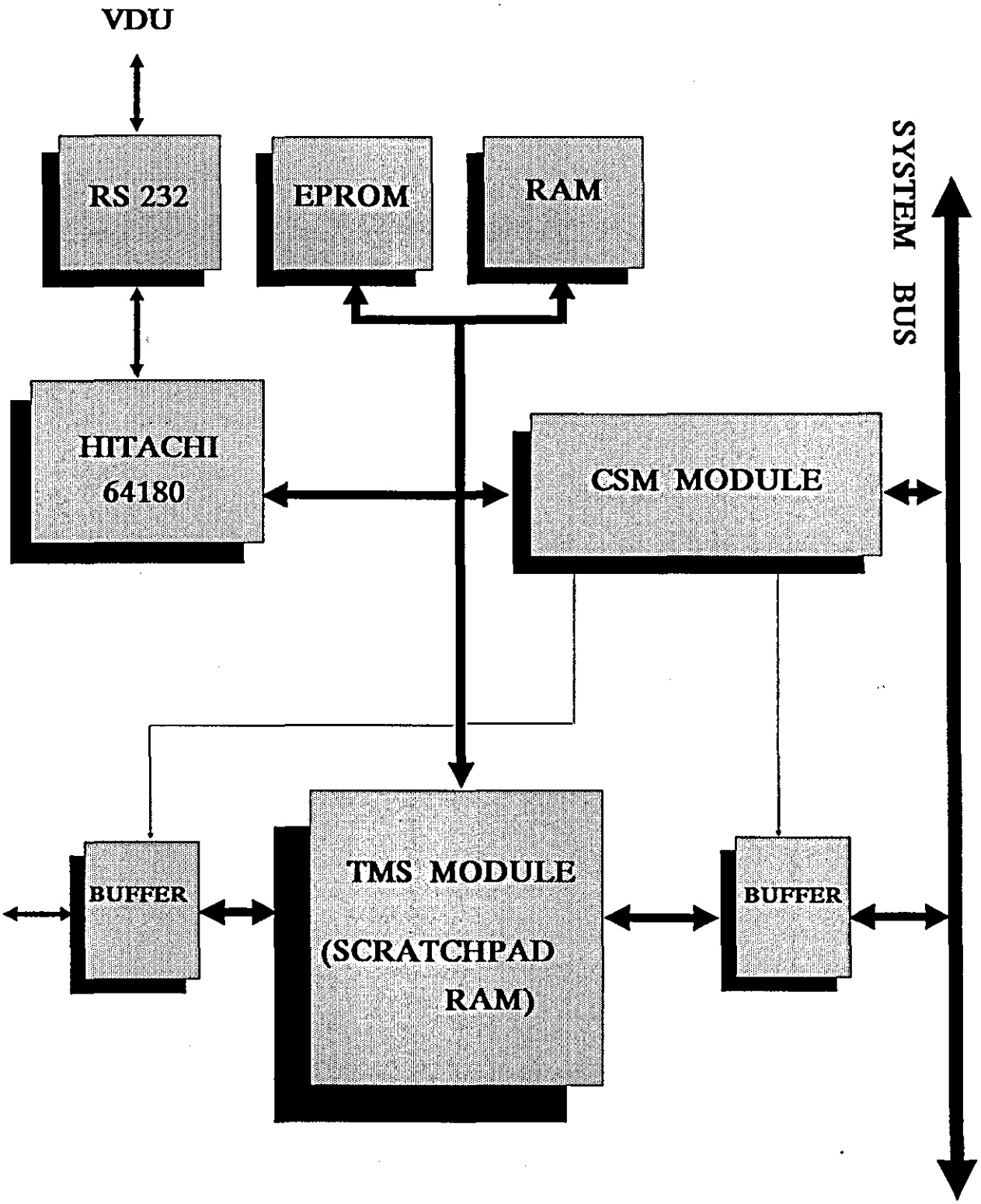


Fig. 8.5 THE COMMUNICATION SECTION - DETAILED STRUCTURE

CHAPTER 9

CHAPTER 9

COMMENTS AND CONCLUSIONS

9.1 ARCHITECTURE

9.1.1 Loosely-Coupled Systems

Both loosely-coupled and tightly-coupled multi-processor structures are applicable to the area of real-time, multi-processing systems. The decision to use a loosely-coupled structure was based on the need to host a functionally partitioned environment. This stems from the following points:

* Loosely-coupled systems generally perform quite well as the number of processors is increased. In contrast, most tightly-coupled systems experience severe performance rolloff fairly quickly with the addition of extra processors. One of the sources of this performance degradation is that the mechanisms commonly used for concurrency control work by specifically restricting parallelism, thereby limiting the value of additional processors [1].

* Loosely-coupled systems communicate only through the use of message-passing primitives. A spectrum of constructs are widely implemented. In tightly-coupled systems, however, message-passing and shared variable constructs may both be implemented. In practice the message-passing approach is used only infrequently, as in [1]; most tightly-coupled systems actually implement the shared variable model [2]. This approach has a number of weaknesses due to the interaction of processes. First, bus

contention can result from process scheduling; for instance, tasks may be engaged in a monitor queue. Second, context or process switch mechanisms occupy the common bus. This may also cause bus contention for a considerable period of time, thus degrading the response times of other processes [2,3].

9.1.2 Functional Partitioning

The importance of adopting a functional partitioning scheme for real-time embedded systems was laid down in Chapters 4 and 7. The primary reasons for developing a real-time, distributed-program kernel for such an environment are as follows:

a) For real-time systems, fast and deterministic responses are essential. In this scheme this is achieved by implementing a simple scheduling policy that relies on allocating each single functional sub-task, together with a collection of user/server interrupts, to a specific processor node of the multi-processor structure. Communication between the nodes takes place in a fast, secure and deterministic manner. This also eliminates the overhead and complexity associated with an intra-node scheduling scheme.

b) In normal distributed systems user programs communicate through the use of remote-procedure-calls (RPC). This mechanism is used because access to shared resources is frequently controlled by specific procedures. Furthermore, some node functions are implemented not on the user node, but as procedures on remote nodes. Thus, various procedures are distributed across the multi-processor system, where access to and execution of such procedures is carried on demand by the user programs remotely.

This policy is in direct contradiction with the nature of functional partitioning. Here the functional tasks are mapped onto the various nodes of the system, which necessitates a similar distribution of system variables. Hence the kernel designed for this project must manage inter-task communication and associated distributed variables efficiently and safely. This is quite different from the classical RPC method normally used with ordinary distributed systems.

c) The kernel structure used here eliminates the need for the use or development of special multi-processing compilers (usually required by closely-coupled distributed processing schemes). Individual software units (sub-tasks) may be implemented, type checked and compiled using uni-processor compilers. This was done successfully in developing the real-time kernel.

d) The implementation of 'Time-Server' routines within the kernel to provide synchronisation of processes and local clocks across the multi-processor system are simple and effective. This idea is not necessarily implemented in real-time kernels.

9.1.3 Communication Features

Inter-processor communication is implemented using an asynchronous message-passing mechanism. Simple and efficient constructs were implemented, allowing both blocking (i.e. wait) and non-blocking (i.e. no wait) schemes within the application task. Synchronous constructs such as rendezvous and channels were not implemented. These are inappropriate to such a loosely-coupled system (where messages pass through several intermediary sub-systems) as they impose a heavy demand on the real-time kernel software for their implementation [4].

The use of a non-contention token passing bus access method (TPBAM) has been shown to be an effective communication mechanism for real-time systems. The TPBAM is clearly well suited for use in hard real-time environments where deterministic operation and system reliability are of the utmost importance.

9.2 HARDWARE STRUCTURE

During the course of this research programme a loosely-coupled multi-processor system was designed and implemented. The system has been developed for use in distributed, real-time applications, three processor nodes (stations) being built to prove the concept. Each station consists of two processors, a Hitachi 64180 for handling communications and an Intel 80188 for executing application programs. The following are some comments on the hardware:

a) Processor node - hardware architecture

The hardware of each station is implemented as a set of functional blocks. This design philosophy was adopted to facilitate future developments. Moreover, the design is processor-independent to allow for replacement by enhanced of the same type or by new, various types. This applies for both the main and the communication processors.

b) Use of advanced programmable logic

The use of advanced, high density programmable logic devices in this project made a significant impact on the hardware aspects of the design. It minimised the chip count and significantly reduced the circuit, hardware complexity. This speeded up the development of the system and simplified the production of a PCB version of the system.

9.3 SOFTWARE STRUCTURE

A software environment to support functional partitioning has been developed and implemented successfully on the multi-processor demonstrator system.

The software has been developed and written mainly using the high level language Modula-2. This required the use of two standard compilers; FTL and Logitech. No special multi-processing features were required. The following comments apply to this software environment:

a) Use of compilers

Two compilers were used in this environment, FTL and Logitech. These are standard compilers, their library functions lacking features required by the software of embedded systems. However, the development of both system and application software for use in embedded system was successfully achieved in this project. Prior to this an investigation was made into the suitability of these compilers for use in embedded systems. For the FTL compiler a run-time environment (CPM100) emulating features of the CP/M operating system had to be specially developed for the project.

Similarly, the Logitech compiler package required adaption; particular system modules had to be modified for use in this embedded application (e.g. the I/O module and the Storage module). A bootstrap loader was also developed, partly in Modula-2, for use with application tasks.

b) Software environment

The communication protocol module and its run-time support have been developed and run successfully. Moreover, the real-time kernel module is fully designed and implemented, ready for use with an appropriate application task. The flexible software methods of handling data within the system eliminate certain synchronising operations which are essential parts of some other distributed-kernels [5]. Overall system functional demonstrations have been developed to prove the applicability of this environment.

9.4 APPLICABILITY OF MODULA-2

The reasons for adopting Modula-2 in this project were already given in Chapter 4 (refer to section 4.5.4). Modula-2 has been found to provide a suitable environment for the design of the software for a real-time, embedded multi-processor system. It provided a sound basis for constructing a software design based on functional partitioning and message-passing primitives.

The adoption of 'Modules' as the main unit of partitioning of software components was found to be most helpful in processing and allocating software components on different target processors. Further, the ability to separately compile such modules considerably speeded up the development process.

The following enhancements to the language should be made to further support work in the area of real-time, distributed applications:

* An exception handling mechanism.
* Remote procedure invocation and resumption (using a modified coroutine mechanism).

9.5 OVERALL COMMENTS

The following points are based on the experience gained in designing and developing the multi-processor system:

a) Architecture

* Loosely-coupled, multi-processor systems readily and simply support real-time, functional partitioning schemes.

* The system can be used for geographically distributed processing; this is facilitated through the use of its in-built serial communications feature.

* A real-time, distributed-program kernel is an essential feature of functional partitioning schemes implemented within loosely-coupled systems.

* Asynchronous message-passing is a suitable means for distributed programs to communicate in a distributed environment. Synchronous constructs such as rendezvous and channels are not appropriate for use by loosely-coupled systems.

* For distributed hard real-time systems, deterministic use of the communication medium is considered to be an essential requirement. The token passing method, being a non-contention deterministic scheme, is clearly well suited for use in such environments.

b) Hardware structure

* Significant flexibility is achieved by allowing the hardware design to be specified and implemented in functional blocks. This enables future modifications to take place easily and more efficiently.

* EPLD devices minimise hardware complexity, and reduce chip count immensely.

c) Software structure

* Standard compilers can be modified and used efficiently in real-time, embedded applications.

* Modula-2 is a highly suitable language for use in the programming of real-time systems.

* Modula-2 can be adapted for use in distributed processor environments, despite its lack of full concurrent constructs.

* Management of inter-task communication and associated variables is implemented efficiently and safely through the use of 'Distributed-Variables' within the real-time kernel.

* The communication modes of operation (transmission and reception) are effectively implemented in the real-time kernel. Transmission mode is serviced as part of the background process, whereas reception mode is serviced through an interrupt handler.

9.6 FUTURE WORK

The following hardware modifications should be made to improve system performance:

* Increase the data transfer rate between the communication and the processing sections (DMA rate increase).
* Use transparent dual port RAM.
* Increase the communication processor (64180 CPU) clock speed.

The data transfer rate can be increased by using a 16 MHZ clock for the 80188 CPU. This increases the DMA transfer rate to 1 M Byte/s.

The system performance can also be improved by replacing the current dual port RAM (TMS9650) by one which allows simultaneous access from the two ports. This will reduce the delay experienced when a station is transmitting a message to a station which is busy exchanging information with its processing section.

The 64180 CPU speed can be increased to 10 MHZ. This modification reduces the set-up time needed to prepare a message for transmission.

On the software side, the following enhancements are highly desirable:

a) Integration of a multi-tasking, real-time executive

In the model developed so far for functional partitioning the total system function is defined as a set of cooperating sub-tasks. Each sub-task is then mapped onto one node (or processing section) for further processing. This sub-task runs as one main process (refer to Fig. 7.3). If this main process was further structured as a

collection of cooperating processes a 'multi-tasking' kernel would, then, be needed to schedule and manage the processor resources, leading to increased software complexity and additional overheads. Nevertheless, in larger applications, where more than one sub-task may reside in each node, the introduction of such a multi-tasking kernel is highly desirable. Hence, this facility should be integrated with the distributed kernel already developed.

The multi-tasking executive has already been designed and developed for an embedded system using Modula-2 [6].

b) Improving the software development environment

Development and testing of the system software is a complex, time consuming task. Six processors (excluding 8087 math co-processors) have to be monitored simultaneously. Furthermore, six EPROMs have to be blown in each modification. Improvements to the development environment in general can be achieved through points mentioned earlier in Chapter 1 (refer to section 1.1). Specific improvements, however, can be achieved by:

* Downloading programs directly into the target system. This is achieved through the use of an EPROM emulator to speed up EPROM development process.

* The introduction of program debugging tools dedicated for use within distributed environments. These, at a minimum, should consist of a traditional debugger for sequential processes, together with a master debugger residing on a host system from where the user interacts with the system. The system should support symbolic level debugging on the host, and should have

knowledge about component and process relationship. More sophisticated techniques should be developed to derive performance analysis results from the target system.

c) Fault recovery methods

Currently, a watchdog timer mechanism is used to provide system restart in case of program failure. This is a powerful, defensive mechanism used in fault recovery. With less catastrophic situations, however, a fault recovery mechanism should be developed to handle errors as they arise during task execution. Thus, the need for exception handling mechanism in such cases is essential. One way of implementing this mechanism is to enhance Modula-2 with such a construct.

9.7 A FINAL SUMMARY

The outcome of this research project has been the development and implementation of a fully operational multi-processor system for use in hard real-time applications. The conceptual and practical aspects of a new technique for program structuring, that of functional partitioning, have been proven. A distributed-program kernel has been designed and implemented to support this technique. Considerable enhancements have been made to the software structure of the inter-processor communication mechanism. Extensive hardware design, development, build and test have been carried out in order to produce a 3-node processor system. Programming was performed in both assembly and high level languages. Two high level language compilers were used; both required extensions to fully cater for the needs of real-time embedded applications.

REFERENCES

REFERENCES

Chapter 1

1.1 **Whiddet, D.** 'Distributed programs: An Overview of Implementations', Microprocessors and Microsystems, Vol.10, No.9, pp.475-484, Nov. 1986.

1.2 **Cavano, J.P.** 'Software Issues Facing Parallel Architectures', COMPSAC 88, the 12th. Annual International Computer Software and Applications Conference, Chicago, I.E.E.E. Computer Society Press, pp.300-301, 5-7 Oct. 1988.

1.3 **Cooling, J.E. and Al-Hasawi,W.** 'Token Bus Communications Within a Multiprocessor System', Microprocessors and Microsystems, (11), 4, May 1987, pp.187-195.

1.4 **Cooling, J.E. and Al-Khayatt, S.S.** 'A Functionally Distributed-Program Kernel for Embedded Real-Time Multi-Processor Systems', COMP EURO 89 Conf., IEEE Proc., Hamburg, pp.170-173, May 1989.

1.5 **Cooling, J.E. and Al-Khayatt, S.S.** 'Software Management in a Modula-2 Environment for a Multi-Processor, Embedded, System', First International Modula-2 Conf., Bled, pp.145-149, Oct. 1989.

Chapter 2

2.1 **Cook, R.P.** '*MOD-a Language for Distributed Programmnig', IEEE Trans. Softw. Eng., SE-6 (6), pp.563-571, 1980.

2.2 **Department of Defense, U.S.** 'Programming Language Ada: Reference Manual', Vol.106, Lecture Notes in Computer Science, Springer-Verlag, N.Y, 1981.

2.3 **Downes, V.A. and Goldsack, S.J.** 'The Use of the Ada Language for Programming a Distributed System', in Hasse, V.H. 'Real Time Programming', Pergamon, Oxford, U.K., 1980.

2.4 **Jessop, W.H.** 'Ada Packages and Distributed Systems', Sigplan Not., Vol.17, No.2, pp.28-36, 1982.

2.5 **Hutcheon, A.D. and Wellings, A.J.** 'Ada for Distributed Systems', Computer Standards and Interfaces, Vol. 6, No.1, pp.71-81, 1987.

2.6 **Burns, A.** et al. 'A Review of Ada Tasking ', YCS.78, Dept. of Computer Science, Univ. of York, 1985.

2.7 **Hutcheon, A.D.** et al. 'Distributing Programs Written in Imperative Programming Languages', Dept. of Computer Science, Univ. of York, 1986.

2.8 **Snowden, D.S. and Wellings, A.J.** ' Debugging Distributed Real-Time Applications in Ada, Dept. of Computer Science, Univ. of York, 1987.

2.9 **Tedd, M.** et al. 'Ada for Multi-Microprocessors', The Ada Companion Series, Cambridge Univ. Press, 1984.

2.10 **Liskov, B. and Scheifler, R.** 'Guardians and Actions: Linguistic Support for Robust, Distributed Programs', ACM Trans. on Programming Languages and Systems, Vol.5, No.3, pp.381-404, July 1983.

2.11 **Carpenter, G.F.**, et al 'Guidelines for the Synthesis of Software for Distributed Processors', Proceedings of the 3rd. Programmable Electronics Systems Safety Symposium [PES 386], pp.164-175, 1986.

2.12 **Price, C.C.** 'The Assignment of Computational Tasks Among Processors in a Distributed System', Proceedings of the NCC, 1981.

2.13 **Ng, K.W.** 'Message-Passing Primitives for Multi-Microprocessor Systems', Microprocessors and Microsystems, Vol.10(3), pp.156-160, April 1986.

2.14 **Ng, K.W.** 'A Kernel for Distributed Programming Languages', Interfaces in Computing, (3), pp.199-216, 1985.

2.15 **Cooling, J.E., and Al-Khayatt, S.S.** 'A functionally Distributed-Program Kernel for Embedded Real-Time Multi-Processor Systems', COMP EURO 89 Conf., IEEE Proc., Hamburg, pp.170-173, May 1989.

2.16 **Cooling, J.E., and Al-Khayatt, S.S.** 'Software Management in a Modula-2 Environment for a Multi-Processor, Embedded, System', First International Modula-2 Conf., Bled, pp.145-149, Oct. 1989.

2.17 **Evans, D.J. and Rahma, A.M.** 'Notes on Parallel Processing', Dept. of Computer Studies, Loughborough Univ., 1984.

2.18 **Ma, P.R.** et al. 'A Task Allocation Model for Distributed Computing Systems', IEEE Trans. on Computers, Vol.c-31, pp.41-47, 1982.

2.19 **Wirth, N.** 'Programming in Modula-2', Springer-Verlag, Third Edition, 1985.

2.20 **Mellor, P.V.** et al. 'Adapting Modula-2 for Distributed Systems', Softw. Eng. Journal, pp.184-189, Sept. 1986.

2.21 **Gligor, V.D.** et al. 'An Assessment of the Real-Time Requirements for Programming Environments and Languages', Proc. of Real-Time Symposium, IEEE, Arlington, Virginia (USA), pp.3-16, Dec.1983.

2.22 **Hoare, C.A.R.**, 'Monitors: an Operating System Structuring Concept', CACM (17), No.10 , pp.549-557, 1974.

2.23 **Birrell, A.D. and Nelson, B.J.** 'Implementing Remote Procedure Calls', ACM Transactions on Computer Systems, Vol.2, No.1, pp.39-59, 1984.

2.24 **Hoare, C.A.R.** 'Communicating Sequential Processes', CACM (21), No.8, pp.667-677, 1978.

2.25 **Marshall, R.** 'The Creation, Dispersal and Execution of Concurrent Modules in a Distributed System: Methodological Considerations', IEEE Proc., pp.119-127, 1986.

Chapter 3

3.1 **Deitel, H.M.** 'An Introduction to Operating Systems', Addison-Wesley, 1984.

3.2 **Wirth, N.** 'Programming in Modula-2', Springer-Verlag, 1985.

3.3 **Andrews, G.R. and Schneider, F.B.** 'Concepts and Notations for Concurrent Programming', Computing Surveys, Vol.15, No.1, pp.3-43, 1983.

3.4 **Lamport, L.** 'The Mutual Exclusion Problem', Op.56, SRI International, Menlo Park, Calif., 1980.

3.5 **Dennis, J.B. and Van Horn, E.C.** 'Programming Semantics for Multiprogrammed Computations', CACM (9), No.3, pp.143-155, 1966.

3.6 **Conway, M.E.** 'A Multiprocessor System Design', In Proc. AFIPS Fall Jt. Computer Conf., Vol.24 Spartan Books, Maryland, pp.139-146, 1963.

3.7 **Dijkstra, E.W.** 'Cooperating Sequential Processes', In F.Genuys, Programming Languages, Academic Press., N.Y., 1968.

3.8 **Stankovic, J.A.** 'Software Communication Mechanisms: Procedure Calls Versus Messages', Computer (USA), 1982.

3.9 **Brinch Hansen, P.** 'Operating System Principles', Prentice-Hall, Englewood Cliffs, N.J., 1973.

3.10 **Courtois, P.J.** and et al. 'Concurrent Control with 'Readers' and 'Writers'', CACM (14), No.10, pp.667-668, 1971.

3.11 **Ben-Ari, M.** 'Principles of Concurrent Programming', Prentice-Hall, 1982.

3.12 **Whiddet, D.** 'Distributed Programs: an Overview of Implementations', Microprocessors and Microsystems, Vol.10, No.9, pp.475-484, Nov. 1986.

3.13 **Hoare, C.A.R.** 'Monitors: an Operating System Structuring Concept', CACM (17), No.10, pp.549-557, 1974.

3.14 **Brinch Hansen, P.** 'The Programming Language Concurrent Pascal', IEEE Trans. Soft. Eng., Vol.SE-1, No.2, pp.199-206, 1975.

3.1 **Wirth, N** 'Modula: a Language for Modular Multiprogramming', Soft. Prac. Exper.(7), pp.3-35, 1977.

3.16 **Lampson, B.W. and Redell, D.D.** 'Experience with Processes and Monitors in Mesa', CACM (23), No.2, pp.105-117, 1980.

3.17 **Kessels, J.L.W.** 'An Alternative to Event Queues for Synchronisation in Monitors', CACM (20), No.7, pp.500-503, 1977.

3.18 **Cooling, J.E.** 'Software Dsign for Real-Time Systems', Chapman and Hall, June 1990.

3.19 **Haddon, B.K.** 'Nested Monitor Calls', Oper. Syst. Rev., Vol.11, No.4, pp.18-23, 1977.

3.20 **Lister, A.** 'The problem of Nested Monitor Calls', Oper. Syst. Rev., Vol.11, No.3, pp.5-7, 1977.

3.21 **Parnas, D.L.** 'The Non Problem of Nested Monitor Calls', Oper. Syst. Rev., Vol.12, No.1, pp.12-14, 1978.

3.22 **Wettstein, H.** 'The Problem of Nested Monitor Calls Revisited', Oper. Syst. Rev., Vol.12, No.1, pp.19-23, 1978.

3.23 **Kaubisch, W.H.** and et al. 'Quasi-Parallel Programming', Soft. Prac. Exper.(6), pp.341-356, 1976.

3.24 **Wirth, N.** 'The Use of Modula', Soft. Prac. Exper.(7), pp.37-65, 1977.

3.25 **Andrews, G.R. and McGraw, J.R.** 'Language Features for Process Interaction', In proc. ACM Conf. Language Design for Reliable Software, Sigplan Not., Vol.12, No.3, pp.114-127, 1977.

3.26 **Balzer, R.M.** 'PORTS - a Method for Dynamic Interprogram Communication and job Control', In Proc. AFIPS Spring Jt. Computer Conf., Vol.38, AFIPS Press, Arlington, pp.485-489, 1971.

3.27 **Shatz, S.M.** 'Communication Mechanisms for Programming Distributed Systems', Computer (USA), pp.21-27, June 1984.

3.28 **Liskov, B.** 'Primitives for Distributed Programming', Proc. Seventh ACM Symp. on Operating Systems, pp.33-42, 1979.

3.29 **Andrews, G.R.** 'The Distributed Programming Language SR-Mechanisms, Design, and Implementation', Soft. Prac. Exper., Vol.12, No.8, pp.719-754, 1982.

3.30 **Carpenter, B.E. and Caillian, R.** 'Experience with Remote Procedure Calls in a Real Time System', Soft. Prac. Exper., Vol.14, No.9, pp.901-907, 1984.

3.31 **Brinch Hansen, P.** 'Distributed Processes: a Concurrent Programming Concept', CACM (21), No.11, pp.934-941, 1978.

3.32 **Cook, R.P.** '*MOD-a Language for Distributed Programming', IEEE Trans. Soft. Eng., SE-6, No.6, pp.563-571, 1980.

3.33 **Liskov, B. and Scheifler, R.** 'Guardians and Actions: Linguistic Support for Robust, Distributed Programs', In Proc. 9th. ACM Symp. Principles of Programming Languages, N.Y., pp.7-19, 1982.

3.34 **Department of Defense, U.S.** 'Programming Language Ada : Reference Manual', Vol.106, Lecture Notes in Computer Science, Springer-Verlag, N.Y., 1981.

3.35 **Dijkstra, E.W.** 'Guarded Commands, Nondeterminancy and Formal Derivation of Programs', CACM (18), No.8, pp.453-457, 1975.

3.36 **Marshall, R.** 'The Creation, Dispersal and Execution of Concurrent Modules in a Distributed System: Methodological Considerations', Dept. of Computer Science, U.S. Naval Academy, Annapolis, IEEE Proc., pp.119-127, 1986.

Chapter 4

4.1 **Cooling, J.E. and Al-Hasawi,W.** 'Token Bus Communications Within a Multiprocessor System', Microprocessors and Microsystems, (11), 4, pp.187-195, May 1987.

4.2 **Cooling, J.E., and Al-Khayatt, S.S.** 'A Functionally Distributed-Program Kernel for Embedded Real-Time Multi-Processor Systems', COMP EURO 89 conf., I.E.E.E. Proc., Hamburg, pp.170-173, May 1989.

4.3 **Cooling, J.E., and Al-Khayatt, S.S.** 'Software Management in a Modula-2 Environment for a Multi-Processor, Embedded, System', First International Modula-2 Conf., Bled, Yugoslavia, pp.145-149, Oct. 1989.

4.4 IEEE Standard 802.4-Token Passing Bus Access Method and Physical Layer Specifications, Draft D., IEEE Computer Society, Dec. 1982.

4.5 **Barak, A. and Litman, A.** 'MOS: A Multicomputer Distributed Operating System', Softw. Prac. Exper., Vol. 15(8), pp.725-737, Aug.1985.

4.6 **Mcquillan, J.M., and Walden, D.C.,** 'The ARPA Network Design Decisions', Comput.Network, vol.1, pp.243-289, Aug. 1977.

4.7 **Tanenbaum, A.S. and Renesse, R.V.** 'Distributed Operating Systems', Computing Surveys, vol.17(4), pp.419-470, Dec.1985.

4.8 **Ousterhout, J.K., Scelza, D.A. and Sindhu, P.S.** 'Medusa: An Experiment in Distributed Operating System Structure', CACM, pp.92-105, Feb.1980.

4.9 **Ng, K.W.** 'Message-Passing Primitives for Multi-Microprocessor Systems', Microprocessors and Microsystems, Vol.10(3), pp.156-160, April 1986.

4.10 **Ng, K.W.** 'A Kernel for Distributed Programming Languages', Interfaces in Computing, (3), pp.199-216, 1985.

4.11 **Cooling, J.E.** 'Software Design for Real-Time Systems', Chapman and Hall, June 1990.

4.12 **Gligor, V.D.** et al. 'An Assessment of the Real-Time Requirements for Programming Environments and Languages', Proc. of Real-Time symposium, IEEE, Arlington, Virginia (USA), pp.3-16, Dec. 1983.

4.13 **Davies, A.C.** 'Features of High Level Languages for Microprocessors', Microprocessor & Microsystems (11), 2, pp.77-87, March 1987.

4.14 **Cullyer** et al. 'The Choice of Computer Languages for Use in Safety Critical Systems', Softw. Prac. Exper., Under Publication 1990.

4.15 **Souter, J.** 'The position of Modula-2 Among Programming Languages' First International Modula-2 Conf., Bled, Yugoslavia, pp.89-94, Oct.1989.

4.16 **King, N.** 'Building Bridges to Better Software', Computing Tech., pp.37-43, April 1987.

4.17 **Stroustrup, B.** 'The C++ Programming Language', Reference Manual, AT & T Bell Laboratories, Jan.1984.

4.18 **Djavaheri, M., Osborne, S.** 'Modula-2: An Alternative to C for System Programming', Journal of Pascal, Ada & Modula-2 (5), No.3, pp.47-52, 1986.

4.19 **Feldman, M.B.** 'Modula-2 Projects for an Operating-Systems Course: Racing Sorts and Multiple Windows', IEEE Proceedings, pp.283-288, 1986.

4.20 **Ford, G.A. and Wiener, R.S.** 'Modula-2: A Software Development Approach', John Wiley and Sons, 1985.

4.21 **Wiener, R.S. and Sincovec, R.F.** 'Software Engineering with Modula-2 and Ada, John Wiley and Sons, 1984.

4.22 **Binding, C.** 'Cheap Concurrency in C', ACM SIGPLAN Notices, Vol.20, No.9, pp.21-26, Sept.1985.

Chapter 5

5.1 HD64180 family microprocessor software designers reference manual, Hitachi 1986.

5.2 CMOS 8 bit microprocessor, HD64180 user's manual, Hitachi 1985.

5.3 Altera data book, Altera corporation, 1987.

5.4 Altera design processor user's manual, Altera corporation, 1987.

5.5 TMS 9650 data manual, texas instruments, 1984, No.1602208-9701.

5.6 Intel iAPX 86/88, 186/188 user's manual-hardware reference, Intel 1985, No. 210912-001.

5.7 Intel iAPX 86/88,186/188 user's manual-programmer's reference, Intel 1985, No. 210911-002.

5.8 82188 Data Sheet, Intel 1985.

5.9 SCN2681 Dual Asynchronous Receiver / Transmitter (DUART), Mullard, Signetics 1985, No. 9397 093 66422.

Chapter 6

6.1 **Gilbert, F.** 'Software Design and Development', Science Research Associates Inc., Chicago, 1983.

6.2 **Stevens, W.P.** 'Using Structured Design', Wiley-InterScience, 1987.

6.3 **Jones, G.** 'Structured Programming Design', Hodder and Stoughton, 1985.

6.4 **Storer, R.** 'Practical Program Development Using JSP', Blackwell Scientific Publications, 1987.

6.5 **Jackson, M.** 'Jackson PDF User's Guide', Michael Jackson Systems Ltd., 1987.

6.6 **Moore, D.** 'FTL Modula-2 Language Reference', Workman and Associates, 1987.

6.7 **Moore, D.** 'FTL Modula-2 Z80 User's Manual', Workman and Associates, 1987.

Chapter 7

7.1 **Cooling, J.E.** 'Software Design for Real-Time Systems', Chapman and Hall, June 1990.

7.2 **Cooling, J.E. and Al-Khayatt, S.S.** 'A Functionally Distributed-Program Kernel for Embedded Real-Time Multi-Processor Systems', COMP EURO 89 Conf., IEEE Proc., Hamburg, pp.170-173, May 1989.

7.3 **Cooling, J.E. and Al-Khayatt, S.S.** 'Software Management in a Modula-2 Environment for a Multi-Processor, Embedded, System', First International Modula-2 Conf., Bled, Yugoslavia, pp.145-149, Oct. 1989.

7.4 **Tanenbaum, A.S. and Renesse, R.V.** 'Distributed Operating Systems', Computing Surveys, vol.17(4), pp.419-470, Dec. 1985.

7.5 **Andrews, G.R. and Schneider, F.B.** 'Concepts and Notations for Concurrent Programming', Computing Surveys (15), No.1, pp.3-43, 1983.

7.6 Logitech Modula-2/86, 'User's Manual LU-GU101-2', Logitech, Inc., 805 Veterans Blvd., Redwood City, CA 94063, USA.

Chapter 9

9.1 **Olson, R.A.** et al. 'Messages and Multiprocessing in the ELXSI System 6400', Proc. I.E.E.E, Parallel Processing Conf., pp.21-24, 1983.

9.2 **Lionel, M.N.** et al. 'Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems', IEEE Trans. on Software Eng., vol. 15, no.3, pp.327-334, March 1989.

9.3 **Campenhout, J.M.** et al. 'A stochastic Model for Closed Loop Preemptive Microprocessor I/O Organisations', IEEE Trans. Comput., vol.c-27, Dec. 1978.

9.4 **Shoja, G.C.** et al. 'A Control Kernel to Support Ada Intertask Communication on a Distributed Multiprocessor Computer System', Softw. and Microsystems, vol.1(5), pp.128-134, August 1982.

9.5 **Ng, K.W.** 'A Kernel for Distributed Programming Languages', Interfaces in Computing, (3), pp.199-216, 1985.

9.6 **Cooling, J.E. and Cooling, N.** 'Design and Implementation of an Embedded Real-Time Executive', First International Modula-2 Conf., Pre-Conference Workshop, Bled, Yugoslavia, Oct. 1989.

APPENDIX – A

APPENDIX A

SYSTEM HARDWARE DESIGN

A.1 COMMUNICATION SECTION DESIGN

Description of the communication section is shown in three successive sheets. Circuit diagrams are shown at the end of the section.

A.1.1 Sheet 1 (refer to Fig. A.1)

This consists of three main parts:

* CPU block.
* Memory block.
* Main-processor buffer.

a) CPU block

The design is centred around a 64180 Hitachi processor. The system runs at 6 MHz derived from a 12 MHz crystal. The processor requires a low reset signal. This is generated from a standard RC combination with a time constant of 100 mS. The diode is added to discharge the capacitor faster in the event of a short collapse of the power rails. The processor can be reset momentarily by a switch SW1. The reset signal generated is also used to reset the CSM module.

Two interrupt lines are used in this design; INT1* and NMI*. INT1* is driven by the CSM module on sheet 2. The nonmaskable interrupt, NMI*, is controlled by the circuitry on sheet 3.

The processor has two asynchronous serial communication channels. A use is made of one channel only. The channel is used for monitoring the system status. The TX and RX lines from the processor are buffered/transmitted via the RS232 driver/receiver modules, M6 and M7.

b) Memory block

Two main memory devices are used by the processor. The system is designed to utilise a 64 Kbytes out of the available 512 Kbytes memory address space. This is equally divided between a 32 Kbytes EPROM, M1, and a 32 Kbytes RAM, M2. It is possible to use a smaller RAM if desired (e.g. 6262) this has to be positioned, however, in the second and fourth 8 Kbytes segment of the 32 Kbytes space.

The 64 Kbytes of memory address space is repeated throughout the 512 Kbytes of available address space. Both memory devices are enabled by the processor signal MEM*. The action required by the appropriate device, i.e. read/write operation, is controlled by the CSM module on sheet 2. This activates the EPROM and RAM via the lines EPROMRD*, RAMRD* and RAMWR*.

c) Main-processor buffer

This buffer, 74HCT245, represents the interconnection between the two processors' data buses. The OBI interface control circuitry is described on sheet 2. This buffer is normally disabled by the BUSACK* signal. When a transfer of information is requested, however, the CSM enables the BUSREQ* signal low. The communication processor then responds by enabling the BUSACK* signal thus enabling the buffer. The direction of the buffer is controlled by the OEA* signal.

A.1.2 Sheet 2 (refer to Fig. A.2)

This sheet contains the CSM module, the TMS RAM, their support components, and the system bus drivers. Description of the CSM here is limited to its input and output lines. Full details, however, are given next section (A.2).

The CSM module interface with the communication processor is based on the following lines:-

TABLE A-1: CSM INTERFACE LINES

LINES	DESCRIPTION
IOE*	A line indicating a read/write to I/O address space.
EINP	A synchronous clock signal from the processor.
CWR*	Processor's write line.
CRD*	Processor's read line.
A13-A15	Processor's address lines used for decoding.
INT1*	Processor's interrupt line driven by the CSM module.
RESET*	A line used to reset the CSM module.
D0-D4	Part of the processor's data bus.

The CSM module also drives back three lines EPROMRD*, RAMRD* and RAMWR* used for address decoding of the memory block.

The CSM module controls the interface between the processing section and the TMS module via the lines MAINRD*, MAINWR* and MAINCS*. The processing section interrupts the communication processor via the line DMAREQ. The communication processor initiate transfers between the TMS module and the main processor via the lines DMA0 and DMA1 driven by the CSM module.

Eight system bus lines are directly connected to the CSM module as shown in Table A-2 below. These initiate actions within the CSM module and may be interrogated by the communication processor.

TABLE A-2: SYSTEM BUS LINES

LINES	DESCRIPTION
SS0-SS3	The system address lines.
SSS*	This is one of the four lines used to control the action of different stations with respect to the data on the address bus. This line indicates that an address is being output by a station trying to transmit. When it is active all stations should compare their address lines to see if they are being addressed.
SWRT*	This line acts as a write strobe. It is controlled by the station transmitting a message and is used by the receiving station to clock the data from the system bus into the scratchpad RAM.
BUSY*	This line is used in the synchronisation process at the start of a transfer of a data frame. The line is controlled by the station to which the data is being sent. When a station wishing to transmit sends an address then the addressed station holds this line active until it is ready to receive the data. It then de-activates this line.
START	This line is only used during the initialisation process of the system. After power up the logical ring must be formed for token passing. This signal is used to synchronise this action.

All the lines shown are either driven by tri-state buffers or by tri-state buffers connected to act as open collector drivers.

The station address is set by a set of select switches. An oscillator, either 16 or 8 MHz, is shown in this sheet. This is used within the CSM module to generate the timings for data transmission by this station.

The two ports of the TMS block are controlled by the CSM module. Port A is used for communication with the communication and main processors. It is controlled by the three lines CSA*, OEA* and WEA* generated by the CSM module for read/write control. The TMS module has eight registers that must be addressed using the AS0-2 lines. These are latched outputs from the CSM module. The TMS interrupt line TMSINT* is used to generate an interrupt at the end of a transfer.

Port B interface is connected exclusively to the system data bus. Its operation is again controlled by the CSM module through the lines CSB*, OEB* and WEB*. Generation of these lines is controlled by an oscillator during a transmission cycle, once operation is enabled by the communication processor. During message reception, lines are controlled by the system line SWRT* and the recognition of the station's address on the system address lines. Port B address lines are permanently grounded since the only action required is a read/write operation. All other control information is written into the TMS module via port A.

Port B data lines are connected to the system bus through a bi-directional buffer, M4. This buffer is controlled by the same signals used to control port B interface. It is enabled by CSB* line, the direction of transfer being controlled by the OEB*.

The TMS module has additional features not being required by this design. Also certain signals have to be pulled to certain levels to enable the operation in a desired mode. In particular the LOCKIN signals for both ports are pulled high and the signals M1 and M2 are connected to a CR combination to provide a reset signal for the TMS module. The time constant is small enough to allow block reset before initialisation starts on.

A.1.3 Sheet 3 (refer to Fig. A.3)

This sheet shows the watchdog timer which may not be required for all applications. If it is not required, however, then the NMI* interrupt line may be pulled high via the movable link on the card.

The watchdog timer is in fact a monostable formed around the 74HCT123, module M12. The trigger input of the monostable is connected to the comparator M11. This comparator generates an output when the correct action takes place on the bus. This has the effect of triggering the monostable. If the monostable is allowed to time out then an NMI interrupt occurs. The idea of the circuit here is designed to be constantly retriggered by the software before it times out. If the system fails to function properly then time-out occurs, and a non-maskable interrupt (NMI) is generated. The resulting exception response is user defined; in this implementation a program restart is initiated.

To trigger the monostable, the comparator needs to be enabled by reading the correct address in memory. The comparator is enabled by a read to the EPROM address space via the EPROMRD*. This read operation must also have the address lines A0-A6 and A18 set high to generate an output from the comparator. Lines A0 to A6 are relatively easy set high during this read operation. Line A18, however, never goes high during normal system operation. To enable this to occur the memory management unit within the communication processor would have to be programmed to locate 4K of the logical memory address space in the upper half of the physical address space. Alternatively, a DMA operation has to be initiated. This, in fact, may be the best way to reset the timer as all the DMA pointers may be left set up and a one byte transfer is sufficient to accomplish the task.

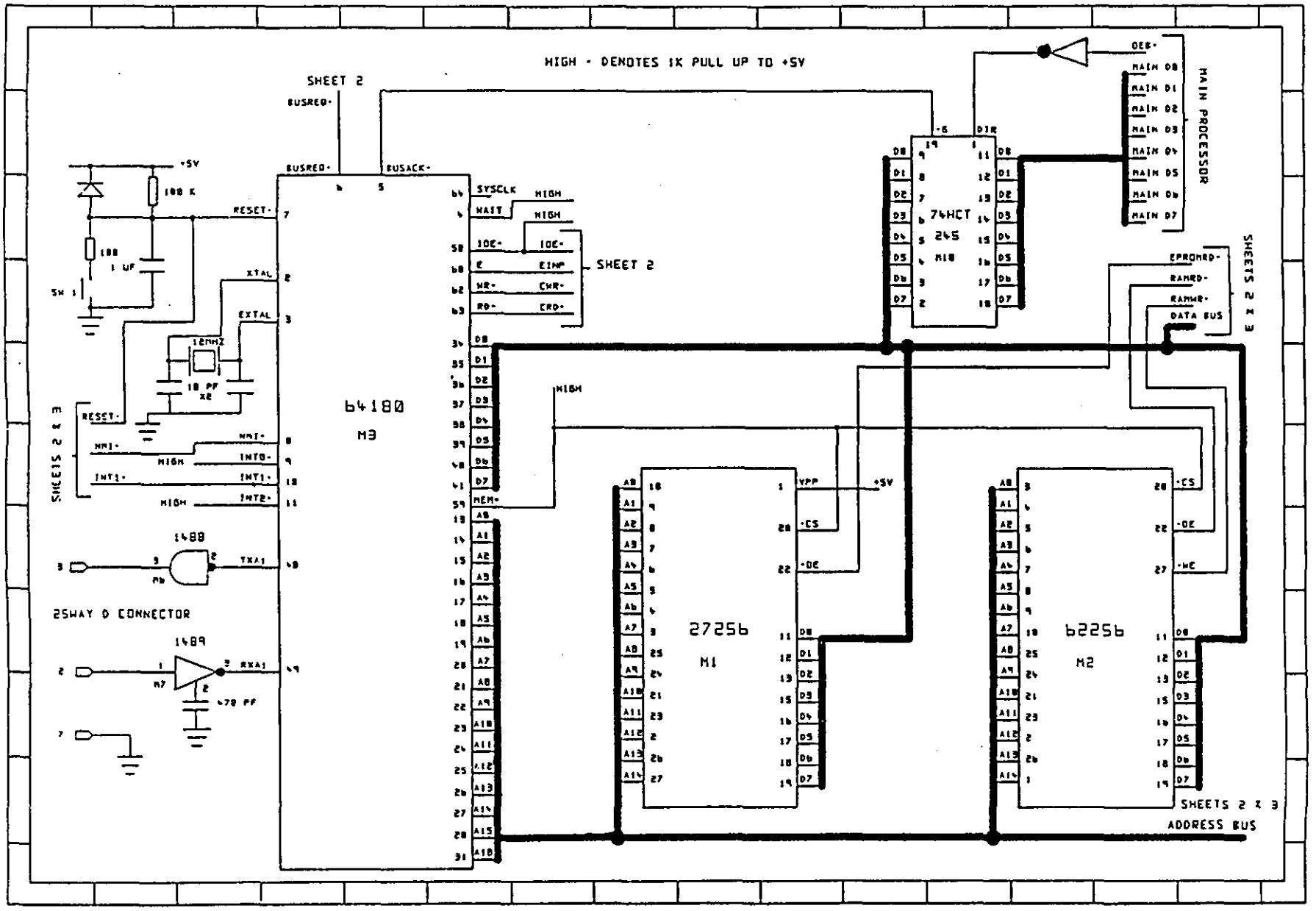


Fig. A.1 COMMUNICATION SECTION HARDWARE DESIGN - SHEET 1

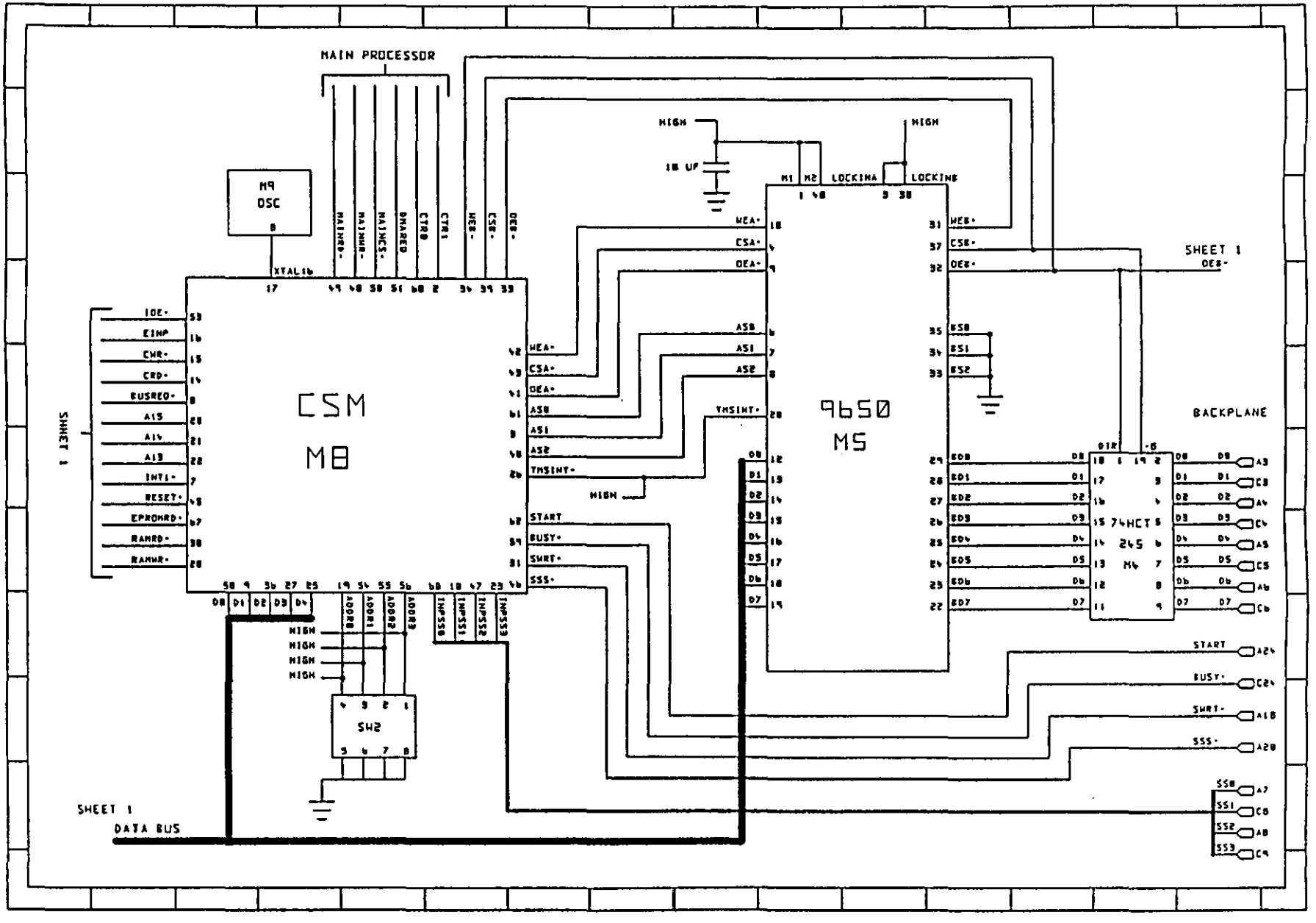


Fig. A.2 COMMUNICATION SECTION HARDWARE DESIGN - SHEET 2

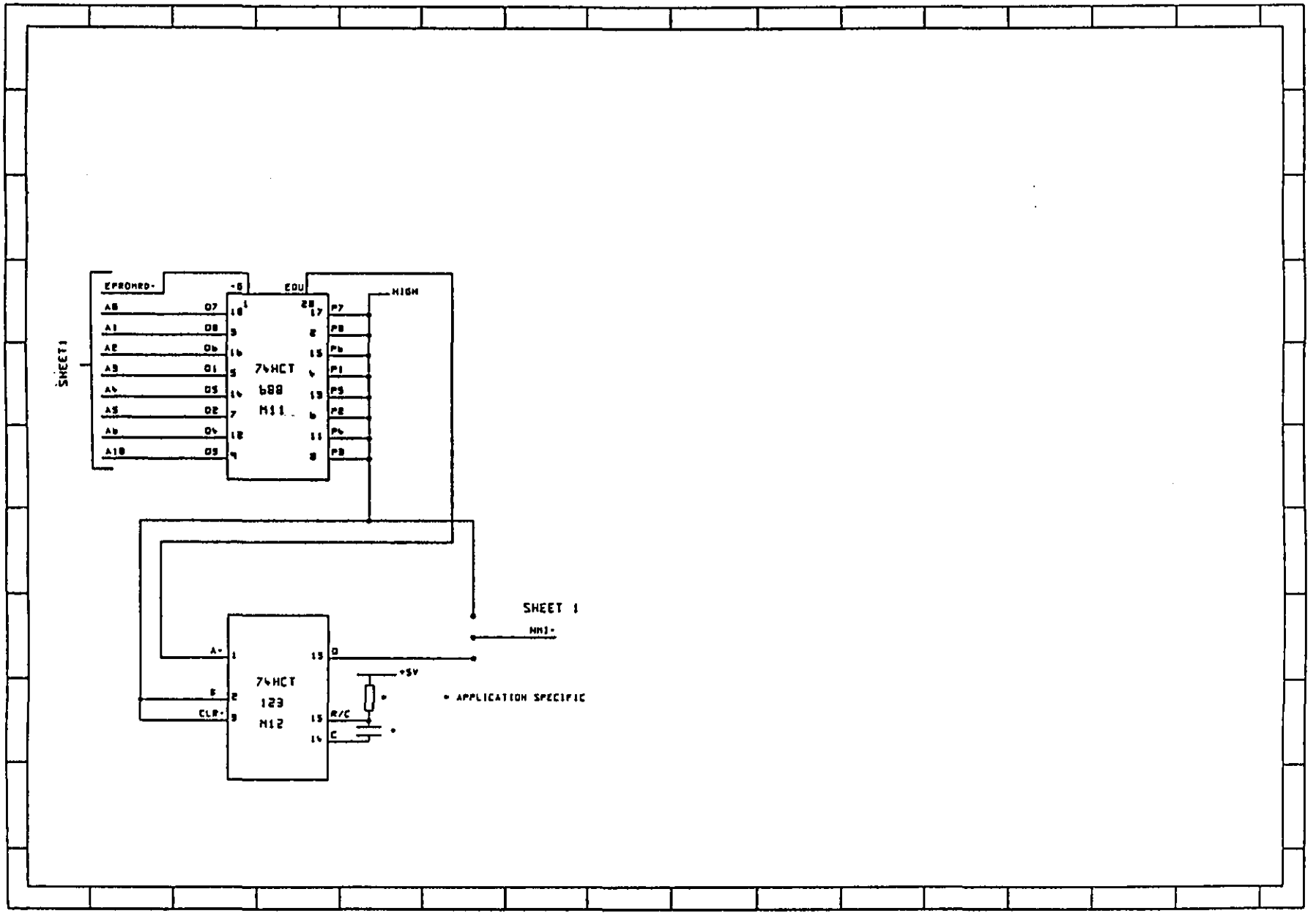


Fig. A.3 COMMUNICATION SECTION HARDWARE DESIGN - SHEET 3

A.2 CSM MODULE DESCRIPTION

The CSM EPLD module, an Altera EP1800, controls most of the processor's support functions and the system bus interface. What follows is a description of the module circuitry.

A.2.1 Module Structure

The EP1800 has 48 macro cells each containing a logic array, a register and an output buffer (refer to Figs. A.4 and A.5). There are also 16 dedicated inputs. The module is divided into four quadrants of twelve macro cells each. Feedback is possible between the different macro cells of each quadrant. Within the different quadrants, feedback is limited to four output feedback lines per quadrant. These must be from the I/O pins, as opposed to a function before the output buffer. This limits the flexibility of fitting a circuit into the module as there is a high degree of inter-connectivity required in the circuit.

A.2.2 Sheet 1 (refer to Fig. A.6)

This sheet contains all the address decoding circuitry apart from the watchdog timer. The Hitachi 64180 has two main address spaces, one for the memory and the other for the I/O space. The memory address space is 512 Kbytes while the I/O space is 64 Kbytes. In this design 64 Kbytes of the memory address space is used and is divided into two segments by the decoders on this sheet. Five registers are decoded in the I/O space to control the hardware. The sixth register is used internally by the processor to control timers, etc.

Three signals are generated to control the 64 Kbytes of address space used. These repeat within the available space (512 Kbytes). The lower memory space is used to hold the EPROM, while the upper is used .mb7

for the RAM. When the processor attempts to access the memory space it enables the MEM* signal which is taken direct to the memory block, i.e. both EPROM and RAM. Different memory operations are controlled by the signals EPROMRD*, RAMRD* and RAMWR*, generated by the CSM module. These are decoded from the address line A15, used to separate the two memory devices.

The 5 registers within the CSM module, each occupying 8 Kbytes block, are decoded in this sheet. They are addressed using the address lines A15, A14, and A13. These are gated with the IOE* line from the processor specifying that an operation is required in the I/O space. These five registers, activated for both read/write operations, are; TMS DATA REGISTER, TMS ADDRESS REGISTER, STATION ADDRESS REGISTER, COMMS CONTROL REGISTER and DMA INTERRUPT REGISTER.

Three other control lines lie within this sheet. The EINP line is the E line from the processor. This is a synchronous clocking line used in association with the other lines on the bus to latch-in data. The CRD and CWR lines are used for read/write operations.

A.2.3 Sheet 2 (refer to Fig. A.7)

This sheet contains the logic design for the communications control register (COMMS CONTROL REGISTER). This handles most of the signals to start or stop actions and also to enable the read operation of various status lines. The register select line from sheet 1 is shown at the top of the sheet and is gated with COMMSRD line to determine if a read operation is required. This generates the COMMS REG RD line which enables the appropriate status lines to pass data to the OR gates, controlling the output of the processor bus on sheet 7. When no read operation is taking place these remain at logic zero. The status lines are as follows:

TABLE A-3: COMMS CONTROL REGISTER (READ)
REGISTER 2 ADDRESS 4000H

BIT	DESCRIPTION
D0	START
D1	TMSINT
D2	RXEN
D3	MAININT

The line START is a system bus line used during initialisation to indicate the status of the overall system. TMSINT is the interrupt line from the TMS9650 scratchpad RAM. This is only enabled during data transmission and is latched elsewhere in the CSM module before being read via this register. The RXEN line indicates that a valid system address, corresponding to this station, is indicated on the system bus. MAININT line indicates an interrupt by the main processor normally requiring a DMA action. The last three lines also activate the processor's INT1* line.

Four latches are shown in this sheet, being used when the processor writes to the register. Shown also is the wait latch. The signal is inverted and then latched when a '1' is written to the wait latch. If the output of this latch is a '0' then this activates the buffer and pulls the START line low. This indicates that this station is not ready to start ring initialisation. Data is inverted before being latched to avoid problems at power up or reset. When the CSM module is reset, the RESET line, goes high for a short period of time. This has the effect of clearing this latch and so indicating that this station is not ready.

Latches connected to D1 and D2 are of a standard D-type used for the control signals READY and SELECT used elsewhere in the module. These latches are clocked by the same signal as the WAIT latch.

Data line D4 is used for the STX latch. This latch is formed from a cross coupled NAND gate latch as it requires special clear inputs. The latch effectively has one generated clock input, a data input and two clear inputs. The effect of the series of AND/NAND latches is to reset the STX output of the latch whenever the module RESET line or the TMSINT line are active. When neither of these lines is active, data is clocked into the NAND latch by a clock signal. The STX line requires two resets; one is needed to set to a reliable state after a module reset and the other is needed at the end of a system transmission cycle. This removes any constraints placed upon the processor as to the order in which the STX, READY and interrupt lines must be cleared.

Clocking the 4 data latches is generated by the COMMS CONTROL REGISTER, COMMS WR and EINP lines. The COMMS CONTROL REGISTER and COMMS WR lines go active at the start of a write operation to this register. Data at this time, however, is not guaranteed stable on the data lines. The EINP line, generated by the processor, is a delayed synchronous clock that delays clocking the data latches until data lines are guaranteed stable. The 4 written latches are:

TABLE A-4: COMMS CONTROL REGISTER (WRITE)
REGISTER 2 ADDRESS 4000H

BIT	DESCRIPTION
D0	WAIT
D1	READY
D2	SELECT
D3	
D4	STX

A.2.4 Sheet 3 (refer to Fig. A.8)

This sheet contains the logic design responsible for setting this station's address and also the driving and reception of the system address lines.

AND gates, on the right hand side of the sheet, are used for reading the station's address lines ADDR0-3. Signals from address select switches pass both to these AND gates and to sheet 8 for address recognition. AND gate outputs reflect the state of the ADDR lines when the register is correctly addressed and a RD operation is in action. This is determined by the state of lines STATION ADDRESS REGISTER and COMMS RD, which generate the line STAT REG RD. AND gate outputs pass to OR gates controlling the system bus output on sheet 7.

System address lines are controlled by a set of 4 latches shown in this sheet. Four address lines are driven by this station. Buffers are controlled by the line SAEN, generated by a latch. All five latches have a common clock line generated by a write action, being determined by COMMS WR and the register select signal STATION ADDRESS REGISTER. At the rising edge of this clock signal, data is latched-in by the five D-type latches from the data lines D0-4. If a '1' is written to the SAEN latch bit D4, then the output of this latch will place the output of the other four onto the system address lines INPSS0-3 by enabling the tri-state buffers. The value of these lines is also passed to sheet 8 for address recognition. This is the signal used for reading other station's address placed on the system bus.

The SAEN latch is cleared by the module RESET line, disabling the four address line buffers. This avoids any conflicts between stations during a reset or power on. Address latches are not reset as their

state is changed by the write operation enabling the SAEN line. Data read/written to this register is:

TABLE A-5: STATION ADDRESS REGISTER
REGISTER 3 ADDRESS 6000H

BIT	READ	WRITE
D0	ADDR0	INPSS0
D1	ADDR1	INPSS1
D2	ADDR2	INPSS2
D3	ADDR3	INPSS3
D4		SAEN

A.2.5 Sheet 4 (refer to Fig. A.9)

This sheet contains the logic design for the TMS block read/write access operations, both by the main and communication processors.

Processor control over the TMS module is determined by the SELECT line status; a '1' for the communication processor and a '0' for the main processor. In the case of a communication processor's access, the TMS CSA* line is enabled when the TMS DATA register is addressed. Then, determined by the state of the COMMS RD and COMMS WR lines, either the OEA* or WEA* lines to the TMS module are activated. In the case of a main processor's access, the CSA* line is now controlled by the MAINCS* line. The OEA* and WEA* lines are controlled by the MAIN RD* and MAIN WR* lines from main processor. Normally, the SELECT line is set high giving the communication processor the right to access the TMS module and set up the address line registers. The SELECT line resets low, however, enabling the main processor to access the TMS module after a reset operation.

A.2.6 Sheet 5 (refer to Fig. A.10)

This sheet contains the logic necessary to control the transfer of data over the system bus.

A clock signal, supplied to pin XTAL16, is used to control (i.e. clock) data across the system bus. The clock signal is divided by 2, 4 and 8 to give the three timing lines A, B and C. From these timing lines three further lines are generated. The TXCLOCK line is generated when A, B and C are all low, its rising edge indicates the start of a transmission cycle. This clock line is used to clock a latch initiating transmission on sheet 9.

Two further signals are generated at specific times during a transmission cycle. If the signal TXEN, from sheet 9, is active then these timing signals drive both the OEB* and SWRT* lines to the TMS and system bus respectively. This line is, then, used by the receiving station to latch-in data. If the system is in a reception mode, however, then the SWRT* line is routed to drive the WEB* line of the TMS module to latch-in data from the system bus.

A.2.7 Sheet 6 (refer to Fig. A.11)

This sheet contains two separate logic sections; management of processors control over the data bus, and the generation of the TMS address lines.

Data transfer between the TMS module and the processing section is requested either by the communication or the main processors. Prior to a TMS access, the communication processor releases the bus preparing for a data transfer. Following this, the communication processor monitors two states: an end of transmission signal (EDT) by the

processing section, and a possible network message received by the system bus. In case of a message reception by the system bus, the TMS transfer is suspended until the network message has been serviced.

For a DMA transfer to take place, the communication processor writes to the DMA interrupt register. This is indicated by the DMA INTERRUPT REGISTER and COMMS WR lines from sheet 1 going active. Data is then routed to lines D0 and D1 through to the CTR0 and CTR1 lines respectively. A pulse on these lines indicates that the main processor may start its transfer. The BUSREQ* latch is set if either D0 or D1 is high when this write operation occurs. The communication processor then releases the bus within 4 cycles (700nS). The processing section must, therefore, wait at least 700nS before it attempts to start the transfer. No physical damage occurs if it attempts to start too early but an external set of buffers, controlled by the BUSACK* line, will not be enabled and the transferred data is corrupted.

Data transfer is terminated when the processing section activates the DMAREQ line. This sets the MAINOP latch, resetting the BUSREQ* latch, and so releasing the data bus. Two other conditions may release the communication processor's bus; activating either the RXEN line or the module RESET line. These three conditions are ORed to form the reset input to the BUSREQ* latch. Although this is shown as a NAND latch, when implemented in the EPLD, it is in fact a combination of AND and OR gates. In this configuration if both signals are active then the latch is reset enabling the communication processor to investigate the state of the system.

The MAINOP latch is reset by two conditions. One is a write operation to the DMA interrupt register and the other is a module reset. The NAND latch making up this latch is configured so that the reset conditions override the set conditions.

There are two interrupt registers that can be reset by a write operation. When a write is detected and the DMA interrupt register is selected as shown by COMMS WR and DMA INTERRUPT REGISTER then the data on lines D2 and D4 is gated onto the reset lines for the MAINOP and TMSINT latches. The timing of this is controlled by the EINP line to ensure that data lines are valid before being applied to the latches. Data on these lines must be '1' to reset the appropriate latch. The TMSINT latch is shown on sheet 9. The DMA interrupt register is described below:

TABLE A-6: DMA INTERRUPT REGISTER (WRITE)
REGISTER 1 ADDRESS 2000H

BIT	DESCRIPTION
D0	DMA0
D1	DMA1
D2	Clear DMAREQ Latch
D3	
D4	Clear TMSINT Latch

A second register is shown in this sheet. This handles the address lines for the TMS module. The TMS module has eight registers selected by three address lines. These are driven by a latched register, the TMS address register. Latches used are clocked by the TMS ADDRESS REGISTER, COMMS WR and E lines. These latches are all cleared to zero on a module reset by the RESET line.

A.2.8 Sheet 7 (refer to Fig. A.12)

This sheet contains the driving circuitry for the communications' data bus. When CSM data is read four buffers are activated. Only two registers in the CSM module can output data, these being registers 2 and 3. The read condition for either one is given by a '1' on A14, a '0' on A15 and the signals COMMS RD and IOE. This is used as the gating signals for the output drivers. Signals on these pins are also routed back into the module when the communication processor is writing to the CSM module and its output drivers are disabled.

Data line D4 is never driven by any register within the CSM module and so is only connected as an input. Input to the RESET line, shown here, is inverted so that the processor reset line can be used to reset this module.

A.2.9 Sheet 8 (refer to Fig. A.13)

This sheet shows the logic circuitry used to compare a station's address with the system address lines.

Two addresses must be recognised by any station, its own address and the broadcast address, 0FH. The station's own address is recognised by the EXOR of the system address lines and the station's address lines. The broadcast address is recognised by the four input AND gate at the bottom of the sheet. These signals are then combined to show that an address has been recognised. These are then gated with the SAEN and the system line SSS*. If the SAEN line is asserted then it pulls the system SSS* line low showing that this station is outputting an address on the system bus. This also has the effect of preventing the RXEN line going active as the address recognition signal is gated off. If the SAEN line is not enabled, however, then the address recognition signal drives the RXEN line active, showing that another station has placed an address on the system address bus.

A.2.10 Sheet 9 (refer to Fig. A.14)

This sheet contains the logic to control interrupts from the TMS module.

The INT1* line, shown at the top, is formed by an OR combination of RXEN, TMS interrupt and MAININT signals. Below this, is the TMSINT latch. This is set by the interrupt signal from the TMS module, only when a transmission cycle is enabled. Latch is reset by a write operation to the DMA interrupt register setting the signal CLEAR COMMS INT. It is also reset by the module reset line.

The circuitry used to control the generation of the system BUSY* line and the TXEN signal is shown in the middle. The BUSY* line is pulled low by this module if the RXEN line is active. This occurs when the station recognises its address but is not ready yet to commence transmission. When this station wants to transmit data, the STX line is set and the system BUSY* line is monitored. When the BUSY* line goes inactive, the STX signal is applied to the TXEN latch. This latch is clocked by the TXCLOCK signal. Latch controlling this line may be reset for two conditions; a module reset indicated by the RESET line, or a TMS interrupt line going active, indicating the end of transmission.

The logic at the bottom of this sheet controls the enable line to the TMS port B, CSB*. This signal is enabled when either RXEN or TXEN is enabled and the READY line is set.

A.2.11 CSM Module Signal Description

The following signals are used internally within the CSM module:

SIGNAL	DESCRIPTION
RAMRD*	A signal used to access RAM for a read operation
RAMWR*	A signal used to access RAM for a write operation
EPROMRD*	A signal used to access EPROM for a read operation
EINP	A synchronous clock signal from the communication processor used for writing to registers in the CSM module
IOE	An inverted version of the I/O space access line from the communication processor.
COMMSRD	An inverted version of the communication processor's read line
COMMSWR	An inverted version of the communication processor's write line
A14	A communication processor's address line
A15NOT	An inverted version of the communication processor's A15 address line
TMSDATAREGISTER	A register select line for the TMS data register
TMSADDRESSREGISTER	A register select line for the TMS address register
STATIONADDRESSREGISTER	A register select line for the station address register
COMMSCONTROLREGISTER	A communication control register select line
DMAINTERRUPTREGISTER	A DMA interrupt control register select line
TMSUNLTCHINT	An unlatched inverted TMS module interrupt signal
STX	A line set by the communication processor to start a transmission operation
SELECT	A line set by the communication processor to control processor's access to the TMS module, a high level denoting the communications processor
READY	A line set by the communication processor when being ready for transmission or reception of data
START	A system bus line indicating when the station is ready to start system initialisation
MAININT	A latched version of the DMAREQ signal, generated by the main processor as an interrupt to the communication processor
RXEN	A signal indicating that the station's address has been placed on the system address lines
TMSINT	A latched version of the TMS module interrupt line
SS0-3	The system address lines

/continued

SIGNAL	DESCRIPTION
ADDR0-3	This station's address lines
SAEN	A line set by the processor enabling the station to place an address onto the system address lines
COMMSD0-3	Lines containing the data read from the comms control register
STATD0-3	Lines containing the data read from the station's address register
MAINCS*	A chip select line from the main processor used for communication between the processing section and the TMS module
MAINRD*	The main processor read line
MAINWR*	The main processor write line
COMMS	A line indicating the communication processor is accessing the TMS module
MAIN	A line indicating that the main processor is accessing the TMS module
A,B,C	Three clock lines used for the generation of transmission timing, generated by the XTAL16 clock signal
TXCLOCK	A clock signal indicating the start of a transmission cycle and used to clock the TXEN latch
BUSREQ*	A bus request signal from the CSM module to the communication processor
CLEARCOMMSINT	A line set by the processor to clear the interrupt latch set by the TMS module
AS0-2	Address lines for port A of the TMS module
SWRT*	The system write line
OEB*	Output enable signal of port B of the TMS module
WEB*	Write enable signal of port B of the TMS module
CTR0-1	The DMA lines from the processing section indicating the start of a data transfer with the TMS module, also known as DMA0 and DMA1
D0-D4	The communication processor's data bus lines
RESET	The module reset line
SSS*	A system line indicating the value on the system address lines is a valid address
TMSINT*	An interrupt signal from the TMS module
INT1*	An interrupt line to the communication processor
BUSY	A system line used to hold a transmitting station until the receiving station is ready
CSB*	Port B select line of the TMS module

TABLE A-7: REGISTER MAP

ADDRESS	REGION	FUNCTION		
2000H	1	DMA and Interrupt Control Register		
		Bit	Write	
		D0	DMA0 Initiate	
		D1	DMA1 Initiate	
		D2	Clear Main Interrupt Latch	
		D3		
		D4	Clear TMS Interrupt Latch	
4000H	2	Comms Control Register		
		Bit	Write	Read
		D0	WAIT	START
		D1	READY	TMSINT
		D2	SELECT	RXEN
		D3		MAININT
		D4	STX	
6000H	3	Station Address Register		
		Bit	Write	Read
		D0	INPSS0	ADDR0
		D1	INPSS1	ADDR1
		D2	INPSS2	ADDR2
		D3	INPSS3	ADDR3
		D4	SAEN	
8000H	4	TMS Address Register		

Table A-7 (continued)

		Bit	Write
		D0	AS0
		D1	AS1
		D2	AS2
A000H	5	TMS Data Register	
		Bit	Write
		D0-D7	To TMS Module

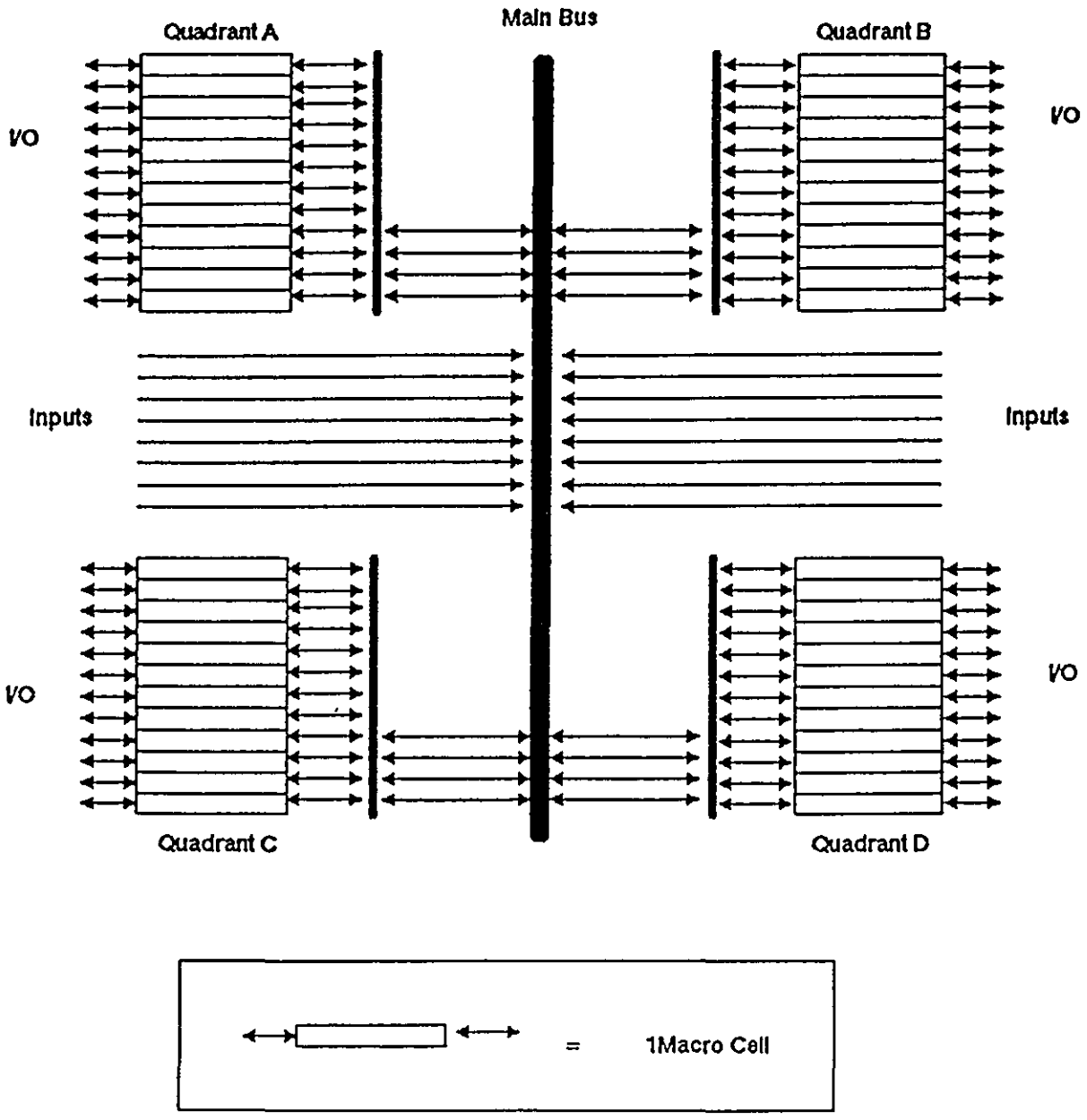


Fig. A.4 EP1800 MACRO CELL STRUCTURE

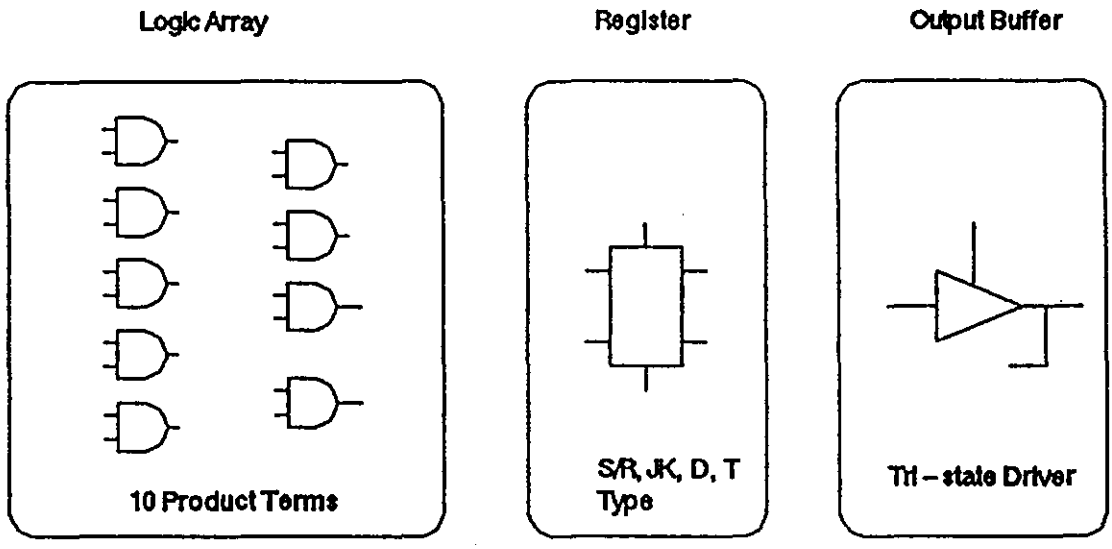


Fig. A.5 MACRO CELL COMPONENTS

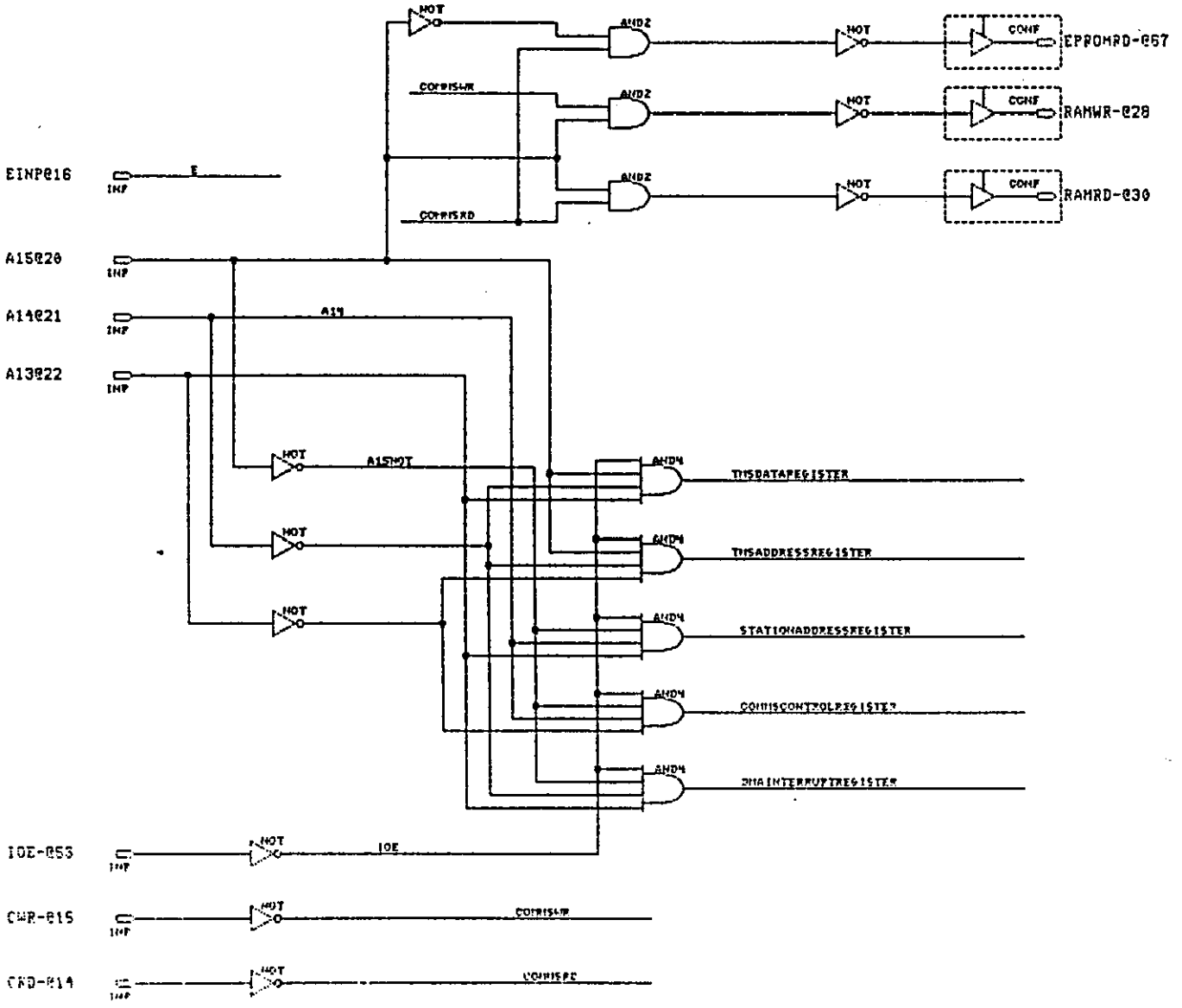


Fig. A.6 COMMUNICATION SUPPORT MODULE (CSM) DESIGN - SHEET 1

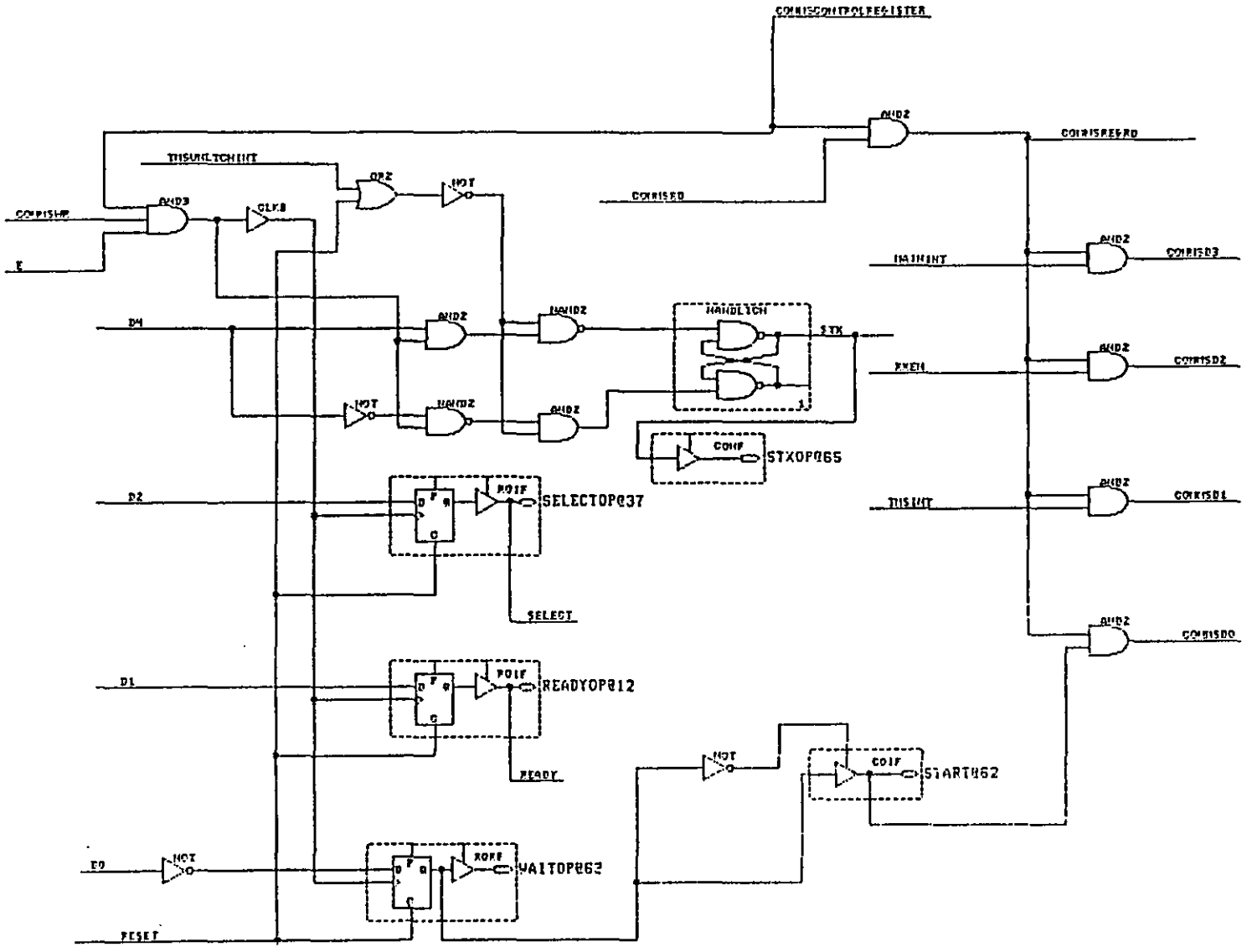


Fig. A.7 COMMUNICATION SUPPORT MODULE (CSM) DESIGN - SHEET 2

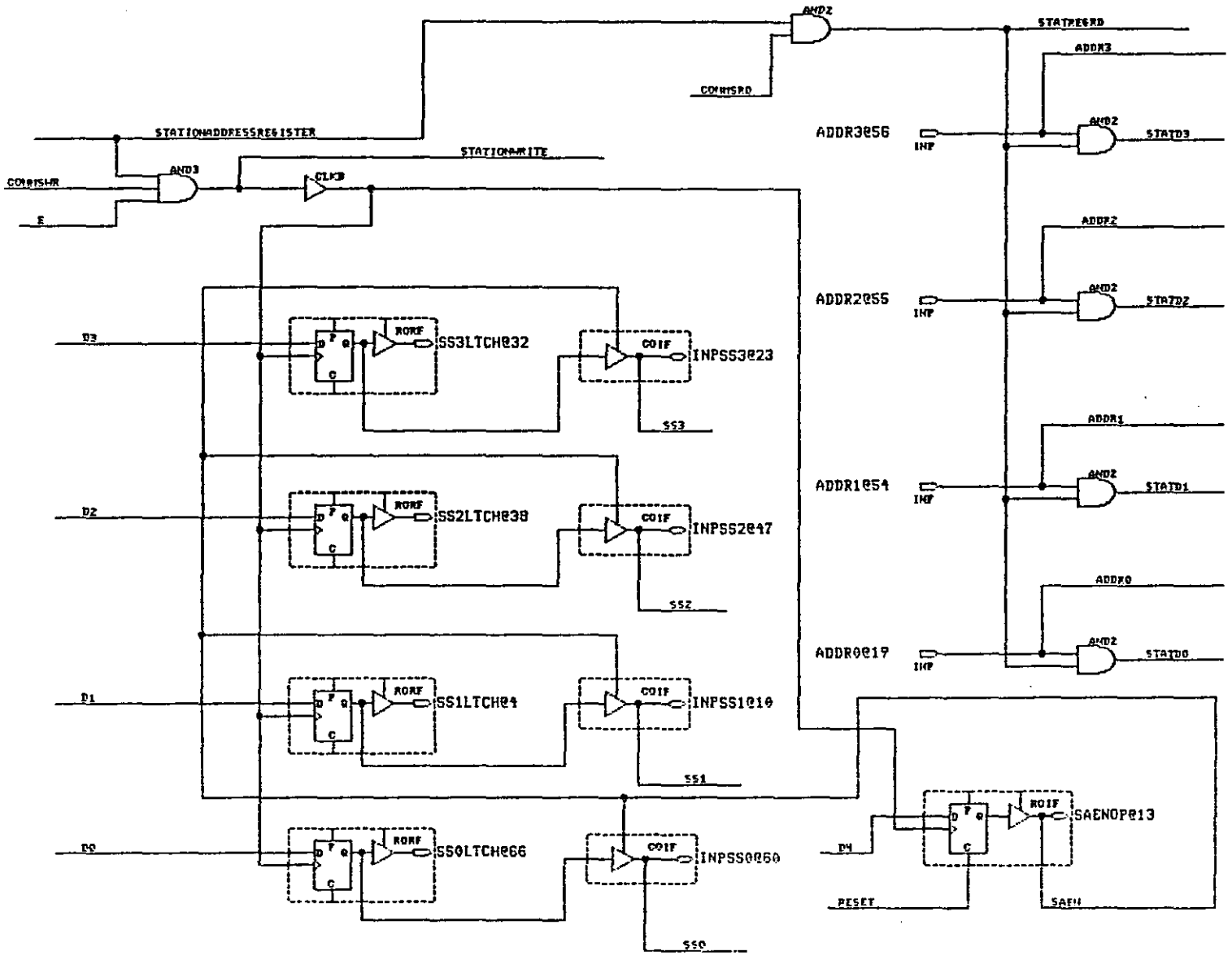


Fig. A.8 COMMUNICATION SUPPORT MODULE (CSM) DESIGN - SHEET 3

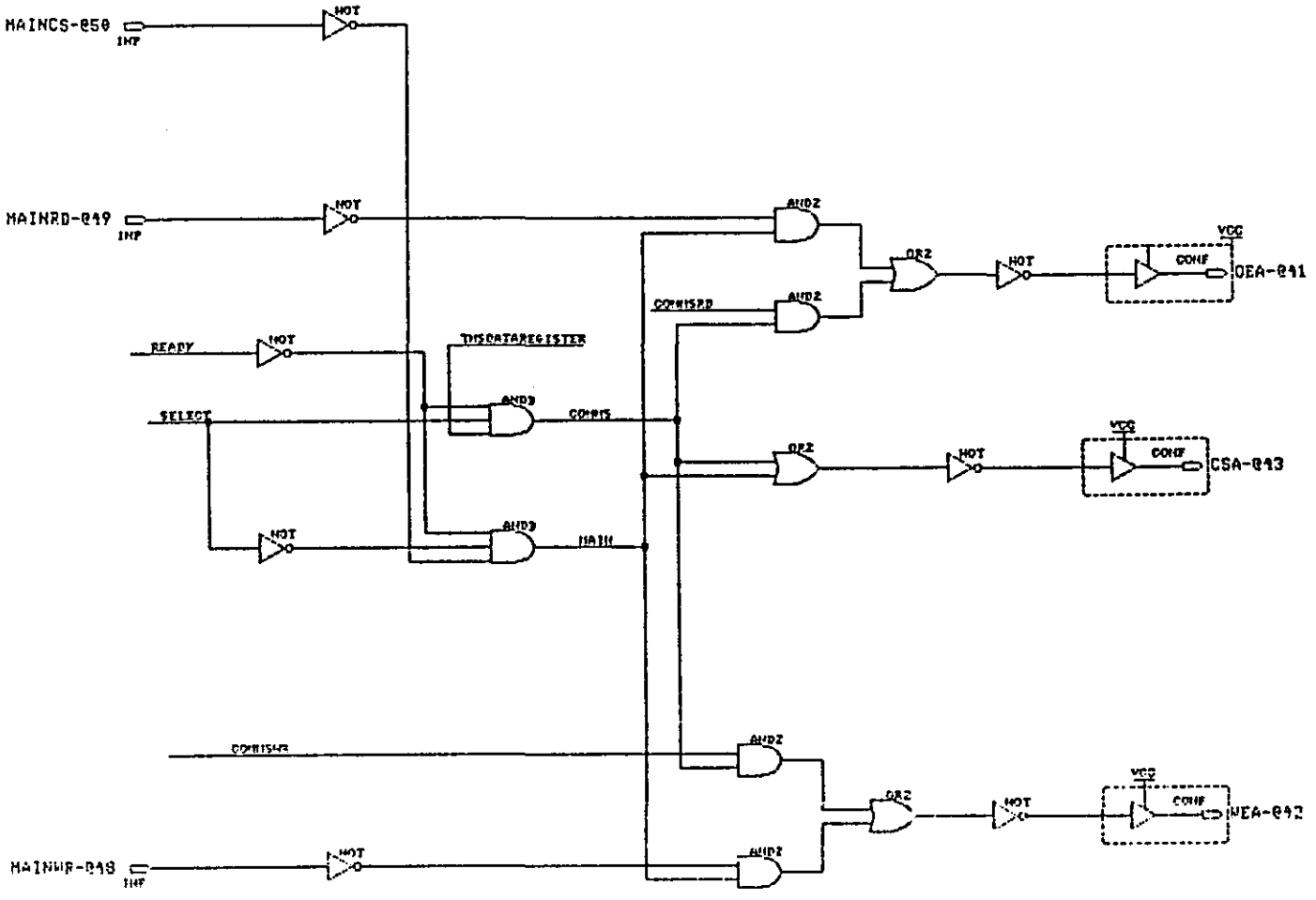


Fig. A.9 COMMUNICATION SUPPORT MODULE (CSM) DESIGN - SHEET 4

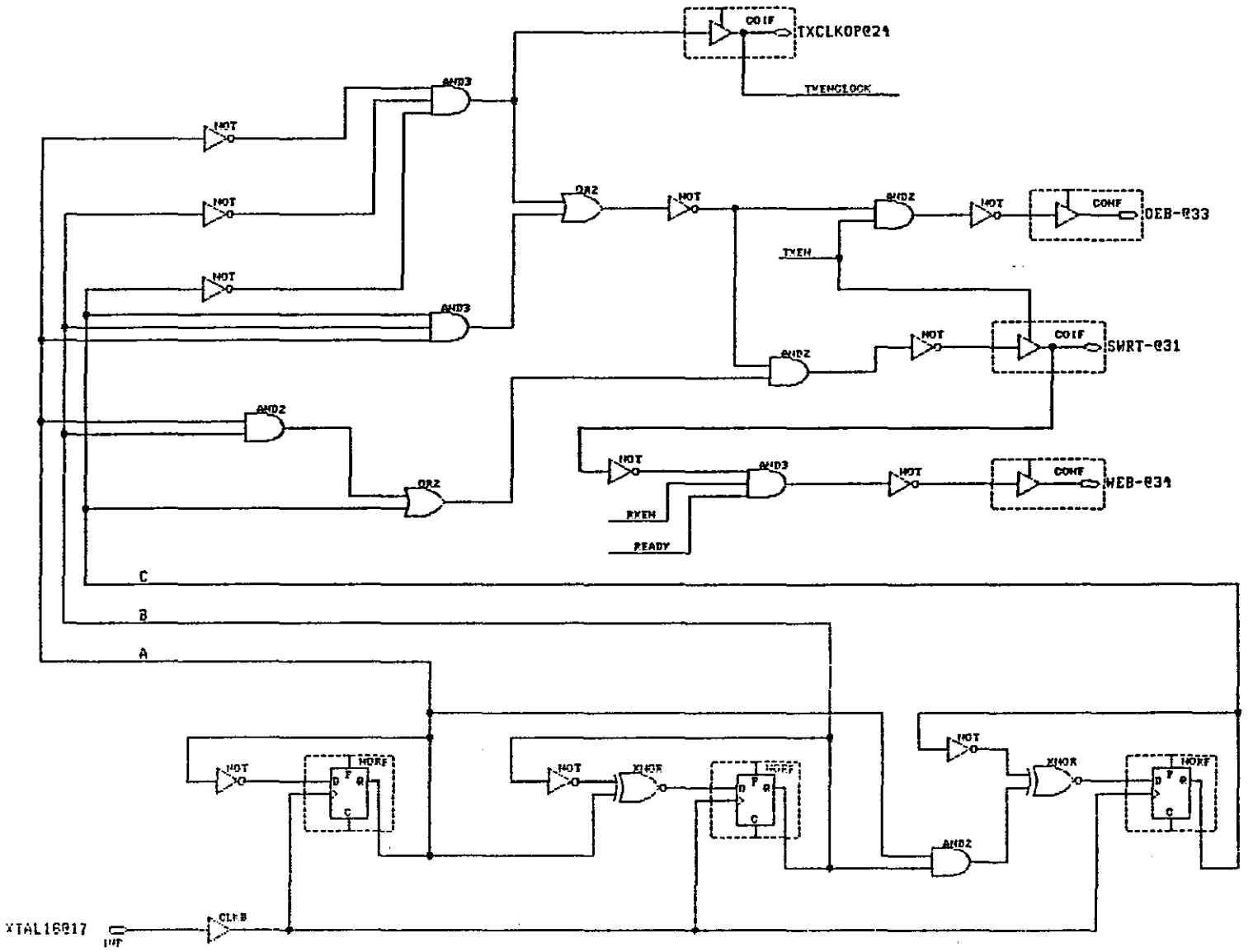


Fig. A.10 COMMUNICATION SUPPORT MODULE (CSM) DESIGN - SHEET 5

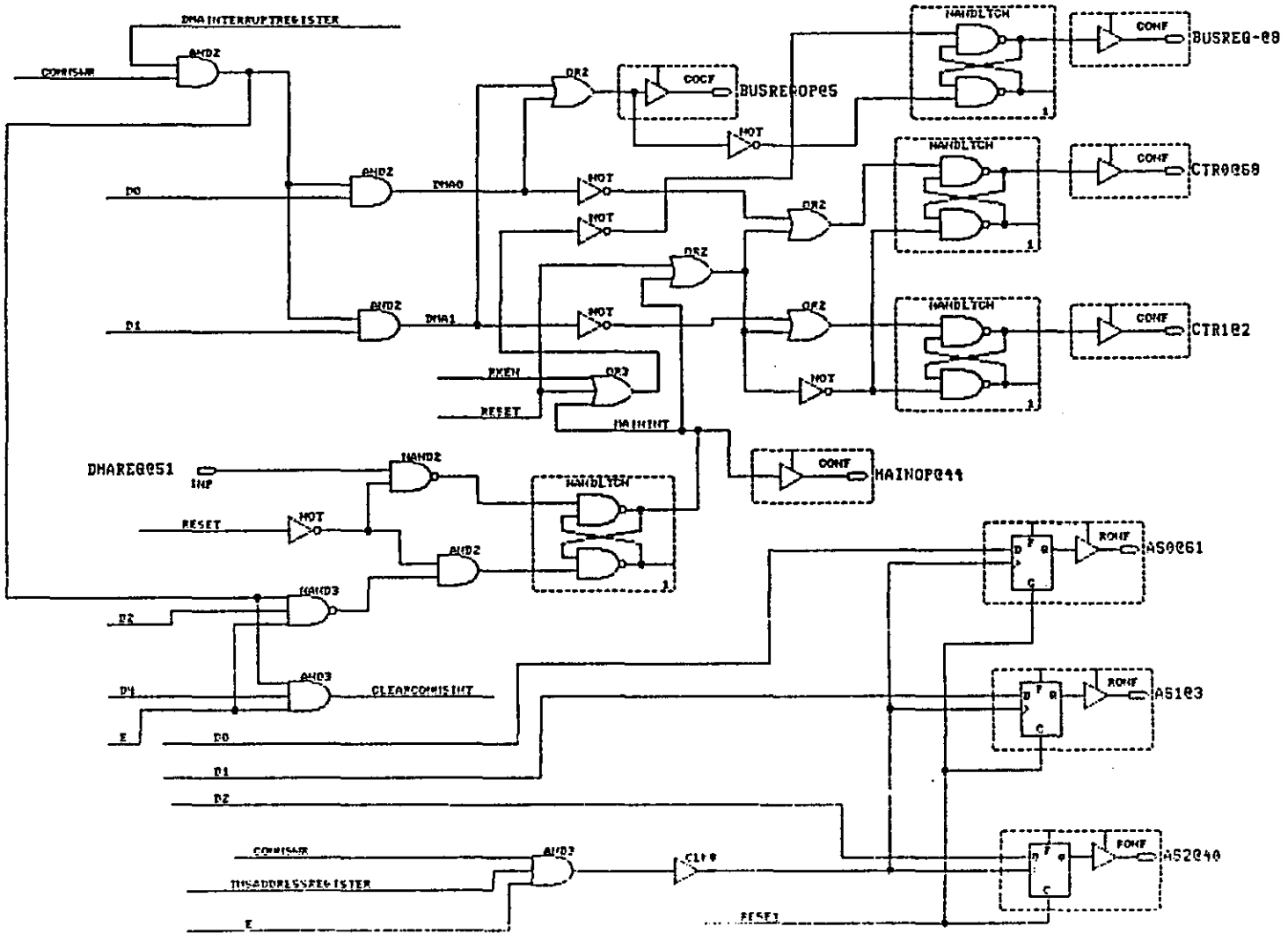


Fig. A.11 COMMUNICATION SUPPORT MODULE (CSM) DESIGN - SHEET 6

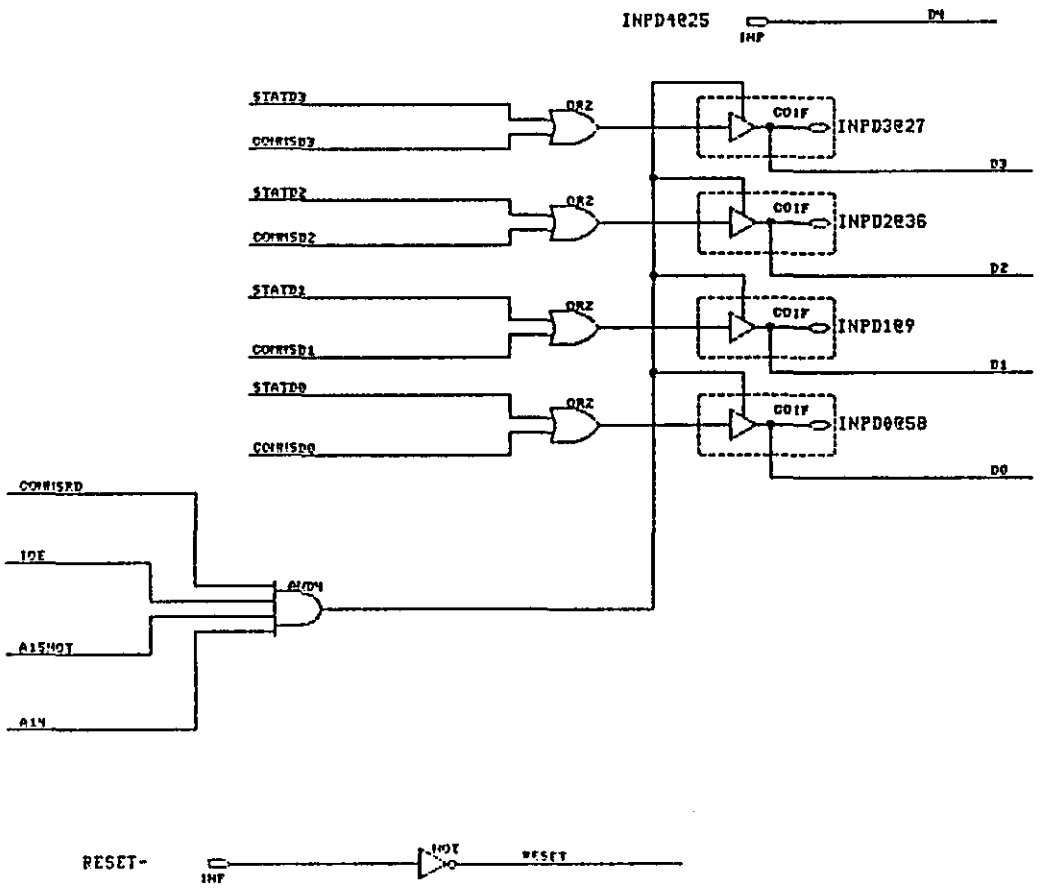


Fig. A.12 COMMUNICATION SUPPORT MODULE (CSM) DESIGN - SHEET 7

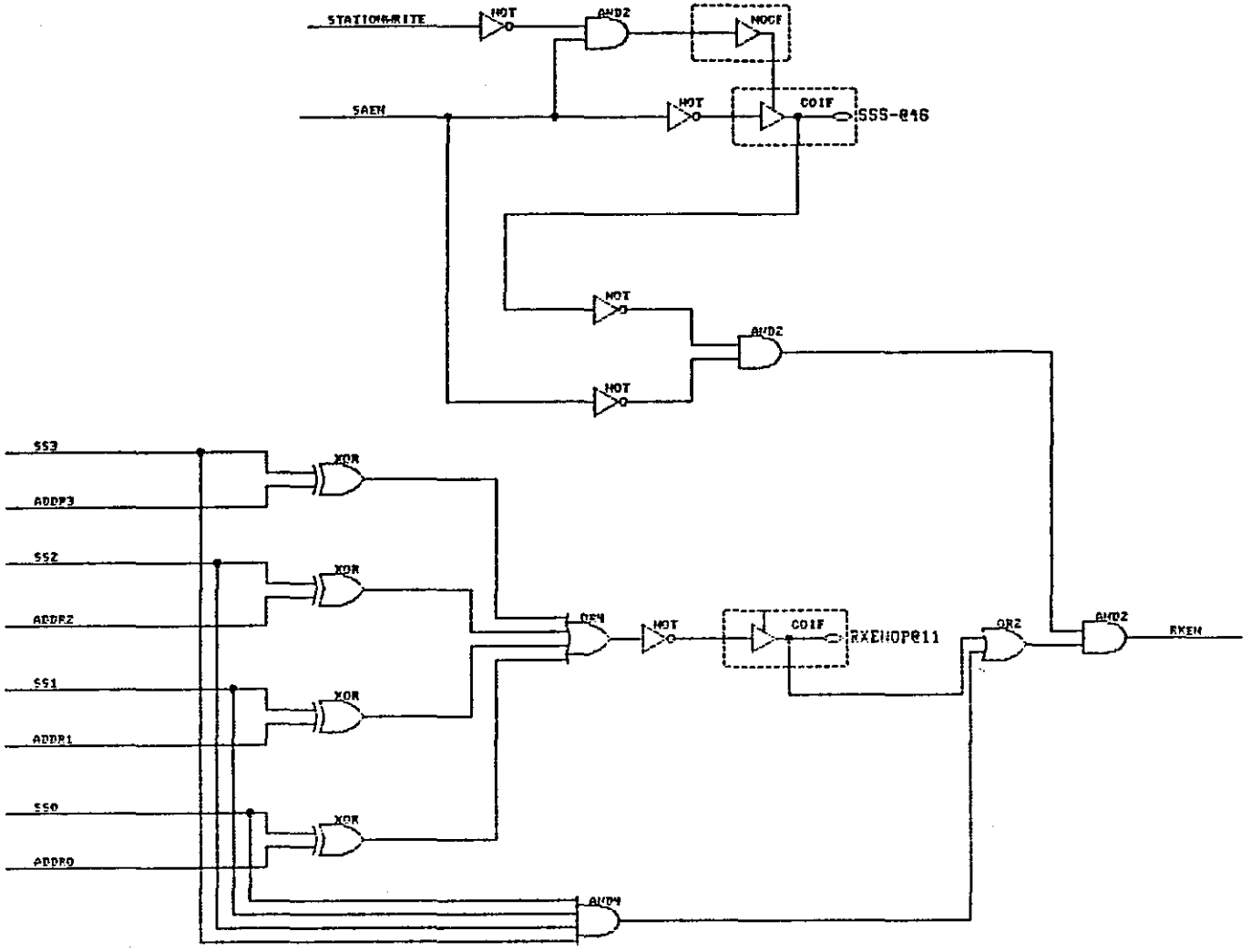


Fig. A.13 COMMUNICATION SUPPORT MODULE (CSM) DESIGN - SHEET 8

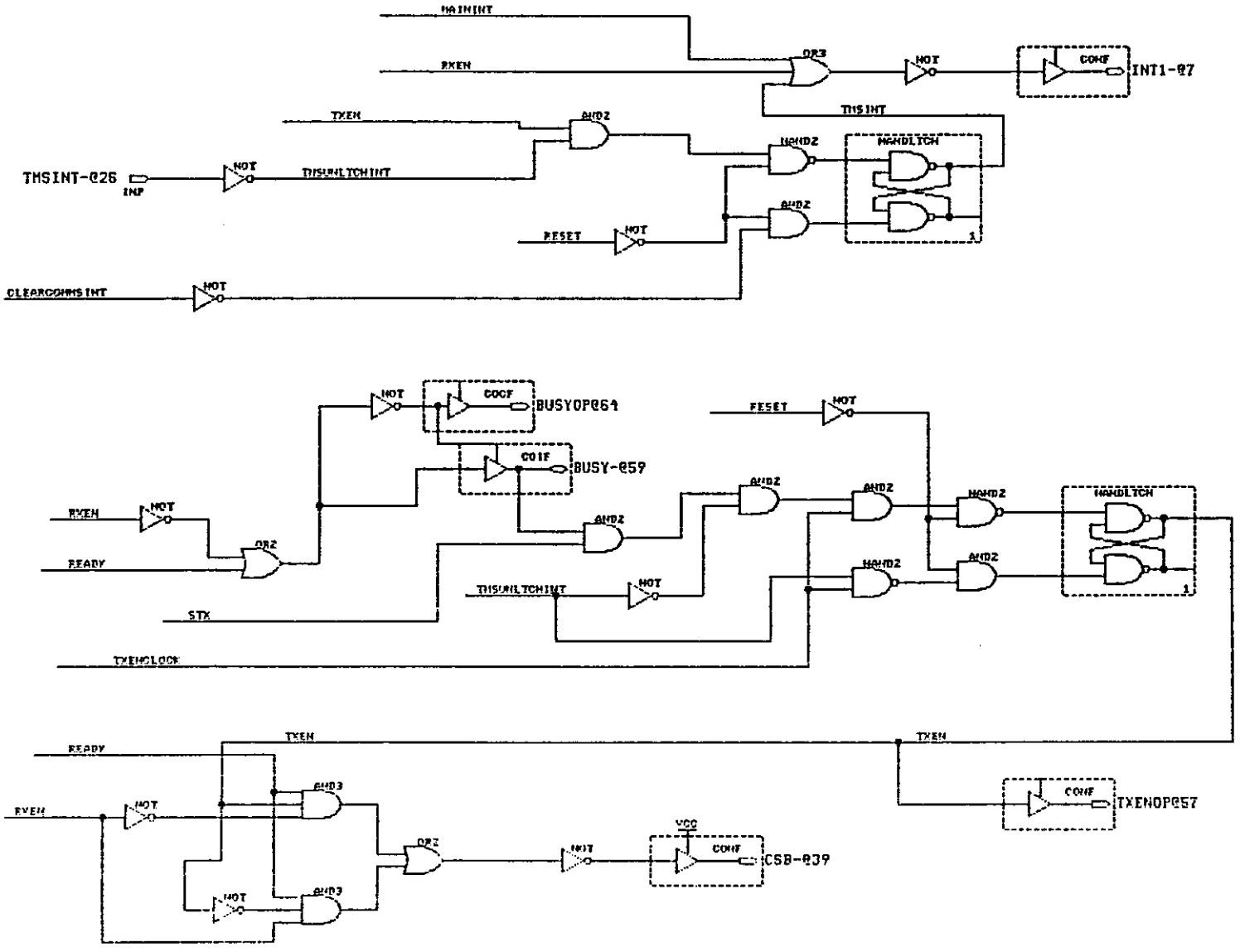


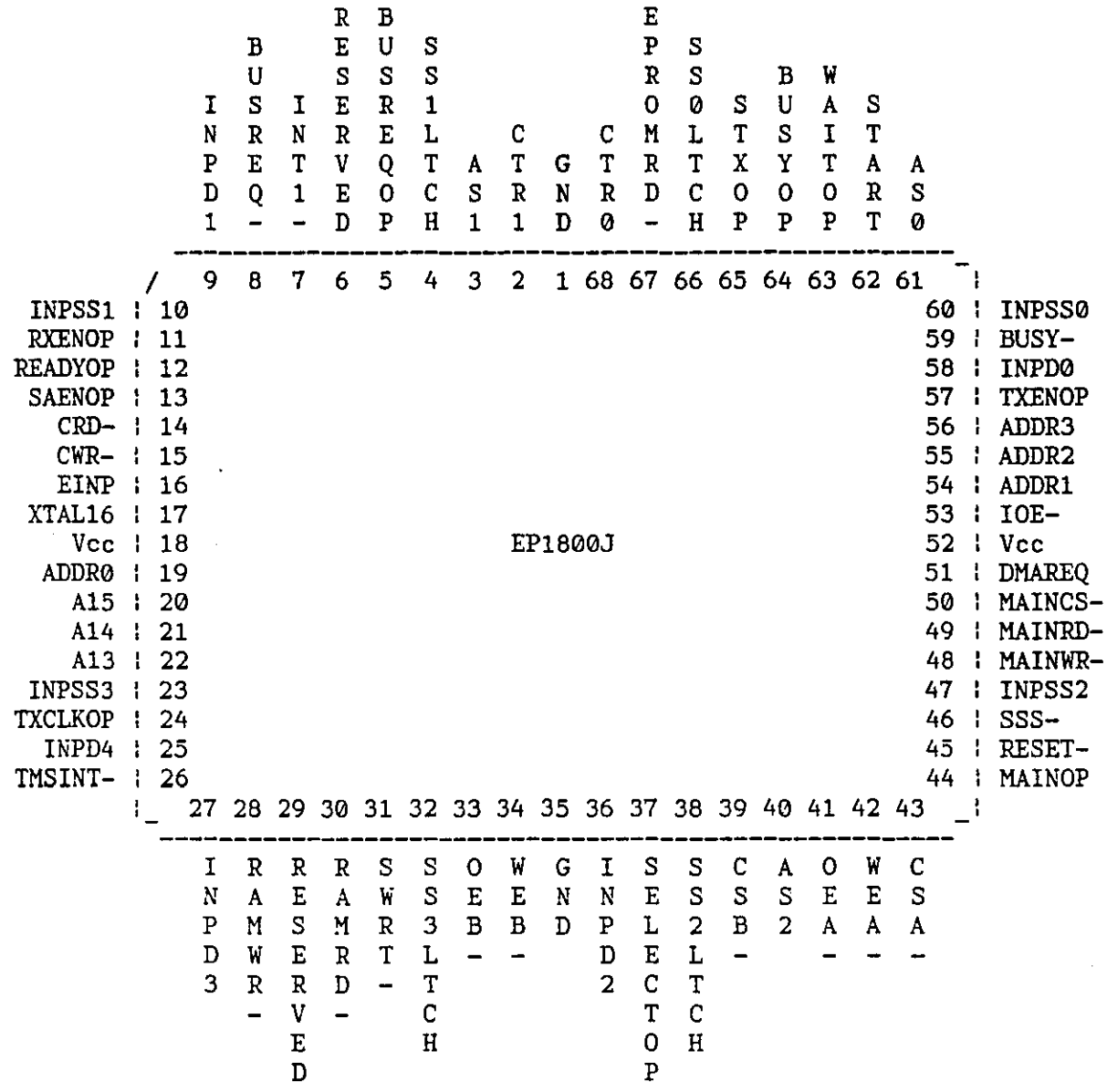
Fig. A.14 COMMUNICATION SUPPORT MODULE (CSM) DESIGN - SHEET 9

A.3 Altera design report

This section presents the report generated by the Altera design processor. Also included at the end of this section are the files used with the functional simulator to test the design. No actual results from this are presented as this was displayed using the VIEW program which produces no hardcopy.

1. Design processor report

```
ALTERA Design Processor Utilization Report
@(#) FIT Version 4.52    1/15/87 16:39:33 34.1.1.1
 ***** Design implemented successfully


Loughborough Univ.
26TH FEB, 1988.
1.00
C
EP1800J
CSM Mk 1
LogiCaps Schematic Capture Ver 1.5
OPTIONS: TURBO = ON
```


INPD1
BUSREQ-
INT1-
RESERVED
BUSREQOP
SSSLTCH
AS1
CTR1
GND
CTR0
EPROMRD-
SSOLTCH
STXOP
BUSYOP
WAITOP
STARTOP
AS0
INPSS1
RXENOP
READYOP
SAENOP
CRD-
CWR-
EINP
XTAL16
Vcc
ADDR0
A15
A14
A13
INPSS3
TXCLKOP
INPD4
TMSINT-
EP1800J
INPSS0
BUSY-
INPD0
TXENOP
ADDR3
ADDR2
ADDR1
IOE-
Vcc
DMAREQ
MAINCS-
MAINRD-
MAINWR-
INPSS2
SSS-
RESET-
MAINOP
INPD3
RAMWR-
RESERVED
RAMRD-
SWRT-
SSSLTCH
OEB-
WEB-
GND
INPD2
SELECTOP
SSSLTCH
CSB-
AS2
OEA-
WEA-
CSA-

OUTPUTS

Name	Pin	Resource	MCell	PTerms	FdBck Group	Sync Clock
RAMRD-	30	CONF	20	1/ 8	2	-
RAMWR-	28	CONF	18	1/ 8	2	-
EPROMRD-	67	CONF	47	1/ 8	4	-
START	62	COIF	42	1/ 8	4	-
READYOP	12	ROIF	11	1/ 8	1G	-
WAITOP	63	RORF	43	1/ 8	4	-
SELECTOP	37	ROIF	26	1/ 8	3	-
STXOP	65	COCF	45	8/ 8	4	-
SS0LTCH	66	RORF	46	1/ 8	4	-
INPSS3	23	COIF	13	1/ 8	2G	-
INPSS0	60	COIF	40	1/ 8	4G	-
INPSS1	10	COIF	9	1/ 8	1G	-
INPSS2	47	COIF	36	1/ 8	3G	-
SS1LTCH	4	RORF	3	1/ 8	1	-
SS2LTCH	38	RORF	27	1/ 8	3	-
SAENOP	13	ROIF	12	1/ 8	1G	-
SS3LTCH	32	RORF	22	1/ 8	2	-
WEA-	42	CONF	31	2/ 8	3	-
CSA-	43	CONF	32	2/ 8	3	-
OEA-	41	CONF	30	2/ 8	3	-
OEB-	33	CONF	23	3/ 8	2	-
TXCLKOP	24	COIF	14	1/ 8	2G	-
SWRT-	31	COIF	21	3/ 8	2	-
WEB-	34	CONF	24	2/ 8	2	-
CTR0	68	COCF	48	2/ 8	4	-
CTR1	2	COCF	1	2/ 8	1	-
AS0	61	RONF	41	1/ 8	4	-
AS1	3	RONF	2	1/ 8	1	-
AS2	40	RONF	29	1/ 8	3	-
BUSREQOP	5	COCF	4	2/ 8	1	-
BUSREQ-	8	COCF	7	5/ 8	1	-
MAINOP	44	COIF	33	3/ 8	3G	-
INPD3	27	COIF	17	2/ 8	2	-
INPD2	36	COIF	25	3/ 8	3	-
INPD1	9	COIF	8	2/ 8	1	-
INPD0	58	COIF	38	2/ 8	4G	-
RXENOP	11	COIF	10	8/ 8	1G	-
SSS-	46	COIF	35	1/ 8	3G	-
BUSYOP	64	COCF	44	2/ 8	4	-
INT1-	7	CONF	6	4/ 8	1	-
BUSY-	59	COIF	39	2/ 8	4G	-
CSB-	39	CONF	28	8/ 8	3	-
TXENOP	57	COIF	37	3/ 8	4G	-

BURIED REGISTERS

Name	Pin	Resource	MCell	PTerms	FdBck Group	Sync Clock
A	-	NORF	15	1/ 8	2	-
B	-	NORF	16	2/ 8	2	-
C	-	NORF	19	3/ 8	2	-
.7108029	-	NOCF	34	2/ 8	3	-
TMSINT	-	NOCF	5	8/ 8	1	-

INPUTS

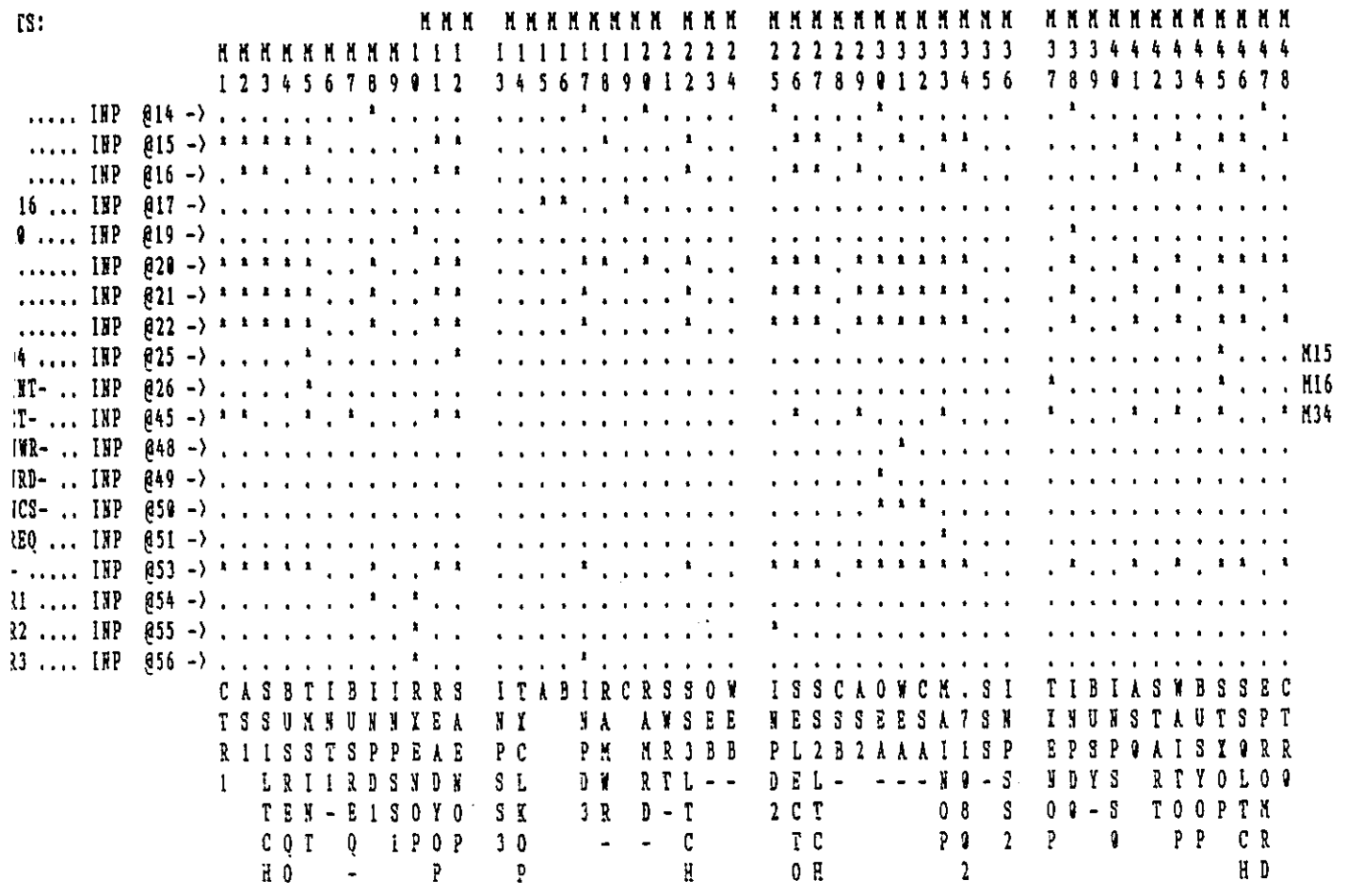
Name	Pin	Resource	MCell	PTerms	FdBck Group	Sync Clock
A15	20	INP	-	-	-	-
A14	21	INP	-	-	-	-
A13	22	INP	-	-	-	-
CRD-	14	INP	-	-	-	-
CWR-	15	INP	-	-	-	-
IOE-	53	INP	-	-	-	-
EINP	16	INP	-	-	-	-
ADDR3	56	INP	-	-	-	-
ADDR2	55	INP	-	-	-	-
ADDR1	54	INP	-	-	-	-
ADDR0	19	INP	-	-	-	-
MAINCS-	50	INP	-	-	-	-
MAINRD-	49	INP	-	-	-	-
MAINWR-	48	INP	-	-	-	-
XTAL16	17	INP	-	-	-	-
DMAREQ	51	INP	-	-	-	-
INPD4	25	INP	-	0/ 8	-	-
RESET-	45	INP	-	0/ 8	-	-
TMSINT-	26	INP	-	0/ 8	-	-

PART UTILIZATION

48/48 MacroCells (100%)
19/19 Input Pins (100%)
PTerms Used 28%

```
·ocell Interconnection Cross Reference

ACKS:                                 M M M   M M M M M M M M M M M M   M M M M M M M M M M M M   M M M M M M M M M M M M
                      M M M M M M M M M 1 1 1   1 1 1 1 1 1 1 2 2 2 2 2   2 2 2 2 2 3 3 3 3 3 3 3   3 3 3 4 4 4 4 4 4 4 4 4
                      1 2 3 4 5 6 7 8 9 0 1 2   3 4 5 6 7 8 9 0 1 2 3 4   5 6 7 8 9 0 1 2 3 4 5 6   7 8 9 0 1 2 3 4 5 6 7 8
 ..... COCF @M1 -> * . . . . . . . . . . .   X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X @2
 ..... RONF @M2 -> . . . . . . . . . . . .   X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X @3
CH .. RORF @M3 -> . . . . . . . . * . . .   X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X @4
QOP . COCF @M4 -> . . . . . . * . . . . .   X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X @5
IT ... NOCF @M5 -> . . . . * * . * . . . .   X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X (6)
- .... CONF @M6 -> . . . . . . . . . . . .   X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X @7
SQ- .. COCF @M7 -> . . . . . . * . . . . .   X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X @8
L .... COIF @M8 -> * * * * . . . . . . * .   X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X @9
S1 ... COIF @M9 -> . . . . . * * . . * . .   . . . . . . . . . . . *   * . . * . . . . . . . .   . . * . . . . * . . . . @10
DP ... COIF @M10-> . . . . . * * . . . . .   . . . . . . . . . . . *   * . . * . . . . . . . .   . . * . . . . * . . . . @11
YOP .. ROIF @M11-> . . . . . . . . . . . .   . . . . . . . . . . . *   . . . * . * * * . . . .   . . * . . . . * . . . . @12
DP ... ROIF @M12-> . . . . . * * . * . . .   * . . . . . . . . . . *   * . . * . . . . . * * *   . . * * . . . * . . . . @13

S3 ... COIF @M13-> . . . . . * * . . * . .   . . . . . . . . . . . *   * . . * . . . . . . . .   . . * . . . . * . . . . @23
KOP .. COIF @M14-> . . . . . . . . . . . .   . . . . . . . . . . . .   . . . . . . . . . . . .   * . . . . . . . . . . . @24
 ...... NORF @M15-> X X X X X X X X X X X X   . * * * . . * . * . * .   X X X X X X X X X X X X   X X X X X X X X X X X X (25)
 ...... NORF @M16-> X X X X X X X X X X X X   . * . * . . * . * . * .   X X X X X X X X X X X X   X X X X X X X X X X X X (26)
3 .... COIF @M17-> X X X X X X X X X X X X   . . . . . . . . . * . .   X X X X X X X X X X X X   X X X X X X X X X X X X @27
R- ... CONF @M18-> X X X X X X X X X X X X   . . . . . . . . . . . .   X X X X X X X X X X X X   X X X X X X X X X X X X @28
 ...... NORF @M19-> X X X X X X X X X X X X   . * . . . . * . * . * .   X X X X X X X X X X X X   X X X X X X X X X X X X (29)
D- ... CONF @M20-> X X X X X X X X X X X X   . . . . . . . . . . . .   X X X X X X X X X X X X   X X X X X X X X X X X X @30
- .... COIF @M21-> X X X X X X X X X X X X   . . . . . . . . . . . *   X X X X X X X X X X X X   X X X X X X X X X X X X @31
TCH .. RORF @M22-> X X X X X X X X X X X X   * . . . . . . . . . . .   X X X X X X X X X X X X   X X X X X X X X X X X X @32
 ..... CONF @M23-> X X X X X X X X X X X X   . . . . . . . . . . . .   X X X X X X X X X X X X   X X X X X X X X X X X X @33
 ..... CONF @M24-> X X X X X X X X X X X X   . . . . . . . . . . . .   X X X X X X X X X X X X   X X X X X X X X X X X X @34

2 .... COIF @M25-> X X X X X X X X X X X X   X X X X X X X X X X X X   . * * . * . . . * . . .   X X X X X X X X X X X X @36
CTOP . ROIF @M26-> X X X X X X X X X X X X   X X X X X X X X X X X X   . . . . . . * * * . . .   X X X X X X X X X X X X @37
TCH .. RORF @M27-> X X X X X X X X X X X X   X X X X X X X X X X X X   . . . . . . . . . . . *   X X X X X X X X X X X X @38
 ..... CONF @M28-> X X X X X X X X X X X X   X X X X X X X X X X X X   . . . . . . . . . . . .   X X X X X X X X X X X X @39
 ...... RONF @M29-> X X X X X X X X X X X X   X X X X X X X X X X X X   . . . . . . . . . . . .   X X X X X X X X X X X X @40
 ..... CONF @M30-> X X X X X X X X X X X X   X X X X X X X X X X X X   . . . . . . . . . . . .   X X X X X X X X X X X X @41
 ..... CONF @M31-> X X X X X X X X X X X X   X X X X X X X X X X X X   . . . . . . . . . . . .   X X X X X X X X X X X X @42
 ..... CONF @M32-> X X X X X X X X X X X X   X X X X X X X X X X X X   . . . . . . . . . . . .   X X X X X X X X X X X X @43
IOP ... COIF @M33-> * . . . . * * . . . . .   . . . . * . . . . . . .   . . . . . . . . * . . .   . . . . . . . . . . . * @44
18029 . NOCF @M34-> X X X X X X X X X X X X   X X X X X X X X X X X X   . . . . . . . . . . * .   X X X X X X X X X X X X (45)
- ..... COIF @M35-> . . . . . * * . . . . .   . . . . . . . . . . . *   * . . * . . . . . . . .   . . * . . . . * . . . . @46
3S2 ... COIF @M36-> . . . . . * * . . * . .   . . . . . . . . . . . *   * . . * . . . . . . . .   . . * . . . . * . . . . @47

IOP ... COIF @M37-> . . . . * . . . . . . .   . . . . . . . . * . * .   . . . * . . . . . . . .   * . . . . . . . . . . . @57
D0 .... COIF @M38-> . . . * . . . . . . . .   . . . . . . . . . . . .   . . . . . . . . . . . .   . . . . . * . * . . * * @58
Y- .... COIF @M39-> . . . . . . . . . . . .   . . . . . . . . . . . .   . . . . . . . . . . . .   * . . . . . . . . . . . @59
3S0 ... COIF @M40-> . . . . . * * . . * . .   . . . . . . . . . . . *   * . . * . . . . . . . .   . . * . . . . * . . . . @60
 ...... RONF @M41-> X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X   . . . . . . . . . . . . @61
RT .... COIF @M42-> X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X   . * . . . . . . . . . . @62
TOP ... RORF @M43-> X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X   . . . . . * . . . . . . @63
YOP ... COCF @M44-> X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X   . . * . . . . . . . . . @64
OP .... COCF @M45-> X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X   * . . . . . . . . * . . @65
LTCH .. RORF @M46-> X X X X X X X X X X X X   X X X X X X X X X X X X   X X X X X X X X X X X X   . . . * . . . . . . . . @66
```

```
MRD- . CONF @M47-> x x x x x x x x x x x x   x x x x x x x x x x x x   x x x x x x x x x x x x   . . . . . . . . . . . . @67
..... COCF @M48-> x x x x x x x x x x x x   x x x x x x x x x x x x   x x x x x x x x x x x x   . . . . . . . . . . . * @68
                   C A S B T I B I I R R S   I T A B I R C R S S O W   I S S C A O W C M . S I   T I B I A S W B S S E C
                   T S S U M N U N N X E A   N X     N A   A W S E E   N E S S S E E S A 7 3 N   X N U N S T A U T S P T
                   R 1 1 S S T S P P E A E   P C     P M   M R 3 B B   P L 2 B 2 A A A I 1 S P   E P S P 0 A I S X 0 R R
                   1   L R I 1 R D S N D N   S L     D W   R T L - -   D E L -   - - - N 0 - S   N D Y S   R T Y O L 0 0
                       T E N - E 1 S O Y O   S K     3 R   D - T       2 C T           0 8   S   0 0 - S   T O O P T M
                       C Q T   Q   1 P O P   3 0       -   -   C         T C           P 0   2   P     0     P P   C R
                       H O     -       P       P               H         O H             2                         H D
                         P
```


CS:
INP @14 ->
INP @15 ->
INP @16 ->
INP @17 ->
INP @19 ->
INP @20 ->
INP @21 ->
INP @22 ->
INP @25 ->
INP @26 ->
INP @45 ->
INP @48 ->
INP @49 ->
INP @50 ->
INP @51 ->
INP @53 ->
INP @54 ->
INP @55 ->
INP @56 ->
M15
M16
M34

2. Simulation test patterns

This section contains listings of the files used in the functional testing of the design. Each listing is in two sections. First the command file used to control the simulation and secondly the vector file.

2.1. CSMINIT command file

This file is used by all the simulations to initialise the module.

```
echo Starting CSM init test module;
group hex inputdata = INPD4 INPD3 INPD2 INPD1 INPD0;
group hex address = A15 A14 A13;
vec @csminit;
cycle 2;
init TMSINT- = 1;
sim 10;
```

2.2. CSMINIT vector file

```
PATTERN:

RESET-      = 0 0 0 0 0 1 1 1 1 1;

CWR-        = (1)*;

CRD-        = (1)*;

IOE-        = (1)*;

EINP        = (0)*;

XTAL16      = (1 0)*;

TMSINT-     = (1)*;

DMAREQ      = (0)*;

MAINCS-     = (1)*;

MAINRD-     = (X)*;

MAINWR-     = (X)*;
```

```
ADDR0       = (0)*;

ADDR1       = (1)*;

ADDR2       = (1)*;

ADDR3       = (1)*;

START       = (Z)*;

BUSY-       = (Z)*;

INPSS0      = Z (X)*;

INPSS1      = Z (X)*;

INPSS2      = Z (X)*;

INPSS3      = Z (X)*;

SSS-        = Z (X)*;
```

2.3. CSMTEST1 command file

This simulation tests the reading and writing of data by the microprocessor to the module.

```
echo Starting CSM module functional test 1;
echo TMS data read and write test;
exec @CSMINIT;
echo 40 step simulation.....;
log @csmtest1;
group hex inputdata = INPD4 INPD3 INPD2 INPD1 INPD0;
group hex address = A15 A14 A13;
vec @csmtest1;
cycle 2;
plot AS0 AS1 AS2 INPD4 INPD3.INP INPD2.INP INPD1.INP INPD0.INP
     A15 A14 A13 CWR- CRD- EINP IOE-  SELECTOP
     CSA- OEA- WEA- CSB- OEB- WEB- SWRT-
     TXCLKOP TXENOP;
sim 40;
view;
save @CSMTEST1;
```

2.4. CSMTEST1 vector file

```
PATTERN:
address   = 0 2 2 2 2 2 2 2 0 0 4 4 4 4 4 4 4 4 4 0 0 4 4 4 4 4 4 4 4 4 0
            0 1 1 1 1 1 1 1 0 0 5 5 5 5 5 5 5 5 5 0 0 5 5 5 5 5 5 5 5 5 0
            (0)*;

inputdata = Z 4 4 4 4 4 4 4 0 0 2 2 2 2 2 2 2 2 2 0 0 5 5 5 5 5 5 5 5 5 0
            0 4 4 4 4 4 4 4 0 0 X X X X X X X X X 0 0 X X X X X X X X X 0
            (0)*;

EINP      = 0 0 0 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0
            0 0 0 1 1 1 1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0
            (0)*;

IOE-      = 1 1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1
            1 1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1
            (1)*;

CWR-      = 1 1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1
            1 1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            (1)*;

CRD-      = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1
            (1)*;
```

```
ADDR0     = (0)*;
ADDR1     = (0)*;
ADDR2     = (0)*;
ADDR3     = (0)*;
MAINCS-   = (1)*;
MAINWR-   = (1)*;
MAINRD-   = (1)*;
XTAL16    = (1 0)*;
DMAREQ    = (0)*;
TMSINT-   = (1)*;
START     = Z Z (Z)*;
BUSY-     = Z Z (Z)*;
INPSS0    = Z (0)*;
INPSS1    = Z (1)*;
INPSS2    = Z (1)*;
INPSS3    = Z (1)*;
SSS-      = Z (1)*;
RESET-    = (1)*;
```

2.5. CSMTEST2 command file

This simulation checks the transfer of control of the data bus between the main and communications processors.

```
echo Starting CSM module functional test 2;
echo DMA transfer control test;
exec @CSMINIT;
echo 50 step simulation.....;
log @csmtest2;
group hex inputdata = INPD4 INPD3 INPD2 INPD1 INPD0;
group hex address = A15 A14 A13;
group hex addr = ADDR3 ADDR2 ADDR1 ADDR0;
group hex ssaddr = INPSS3 INPSS2 INPSS1 INPSS0;
vec @csmtest2;
cycle 2;
plot INPD4 INPD3.INP INPD2.INP INPD1.INP INPD0.INP
     A15 A14 A13 CWR- CRD- EINP IOE-
     BUSREQ- CTR0 CTR1 DMAREQ MAINOP INT1-
     SSS-.INP INPSS3.INP INPSS2.INP INPSS1.INP INPSS0.INP
     ADDR3 ADDR2 ADDR1 ADDR0
     MAINCS- MAINRD- MAINWR-
     CSA- OEA- WEA-
     RXENOP BUSYOP READYOP;
sim +50;
view;
save @CSMTEST2;
```

2.6. CSMTEST2 vector file

```
PATTERN:
address   = 0 1 1 1 1 1 1 1 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            (0)*;

inputdata = 0 1 1 1 1 1 1 1 0
            Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 5 5 5 5 5 5 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            (0)*;

EINP      = 0 0 0 1 1 1 1 1 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            (0)*;
```

```
IOE-      = 1 1 0 0 0 0 0 0 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            (1)*;

CWR-      = 1 1 0 0 0 0 0 0 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            (1)*;

CRD-      = 1 1 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            (1)*;

DMAREQ    = 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 1 1 1 1 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            (0)*;

addr      = (3)*;

SSS-      = 1 1 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1
            (1)*;

ssaddr    = 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 3 3 0 0 0
            (0)*;

MAINCS-   = 1 1 1 1 1 1 1 1 1
            1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
            (1)*;

MAINWR-   = 1 1 1 0 1 1 1 1 1
            1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1
            (1)*;

MAINRD-   = 1 1 1 1 1 1 0 1 1
            1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1
            (1)*;

XTAL16    = (1 0)*;

TMSINT-   = (1)*;

START     = (Z)*;
```

```
BUSY-      = (0)*;

RESET-     = (1)*;
```

2.7. CSMTEST3 command file

This simulation checks the receiving of data from the system data bus.

```
echo Starting CSM module functional test 3;
echo System bus data receive test;
exec @CSMINIT;
echo 50 step simulation;
log @csmtest3
group hex inputdata = INPD4 INPD3 INPD2 INPD1 INPD0;
group hex address = A15 A14 A13;
group hex adr = ADDR3 ADDR2 ADDR1 ADDR0;
group hex ssaddr = INPSS3 INPSS2 INPSS1 INPSS0;
vec @csmtest3;
cycle 2;
plot INPD4 INPD3.INP INPD2.INP INPD1.INP INPD0.INP
     A15 A14 A13 CWR- CRD- EINP IOE-
     BUSREQ- CTR0 CTR1 DMAREQ MAINOP INT1-
     SSS-.INP INPSS3.INP INPSS2.INP INPSS1.INP INPSS0.INP
     ADDR3 ADDR2 ADDR1 ADDR0
     RXENOP BUSY- BUSYOP SWRT-.INP READYOP RESET- TXENOP
     CSA- OEA- WEA- CSB- OEB- WEB-;
sim +50;
view;
save @CSMTEST3;
```

2.8. CSMTEST3 vector file

```
PATTERN:
address   = 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            (0)*;

inputdata = 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            (0)*;

EINP      = 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            (0)*;

IOE-      = 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            (1)*;

CWR-      = 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
                (1)*;

CRD-        = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                (1)*;

DMAREQ      = (0)*;

addr        = (3)*;

SSS-        = 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
                (1)*;

ssaddr      = 0 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
                3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 0
                (0)*;

MAINCS-     = (1)*;

MAINWR-     = (1)*;

MAINRD-     = (1)*;

XTAL16      = (1 0)*;

TMSINT-     = (1)*;

START       = (Z)*;

BUSY-       = 1 Z Z Z Z Z Z Z Z Z Z Z 1 1 1 1 1 1 1 1 1 1 1 1
                1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                (0)*;

SWRT-       = X X X X X X X X X X X X X X X X 1 1 0 0 1 1 1
                1 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 X
                (X)*;

RESET-      = (1)*;
```

2.9. CSMTEST4 command file

This simulation tests the transmission of data from the station across the system bus.

```
echo Starting CSM module functional test 4;
echo System data transmit test;
exec @CSMINIT;
echo 60 step simulation.....;
log @csmtest4;
group hex inputdata = INPD4 INPD3 INPD2 INPD1 INPD0;
group hex address = A15 A14 A13;
group hex addr = ADDR3 ADDR2 ADDR1 ADDR0;
group hex ssaddr = INPSS3 INPSS2 INPSS1 INPSS0;
vec @csmtest4;
cycle 2;
plot INPD4 INPD3.INP INPD2.INP INPD1.INP INPD0.INP
     A15 A14 A13 CWR- CRD- EINP IOE-
     INT1- TMSINT-
     SAENOP SSS- INPSS3 INPSS2 INPSS1 INPSS0
     ADDR3 ADDR2 ADDR1 ADDR0
     CSB- OEB- WEB- SWRT- BUSY-.INP
     XTAL16 A B C
     TXENOP TXCLKOP STX RXENOP BUSYOP READYOP;
sim +60;
view;
save @CSMTEST4;
```

2.10. CSMTEST4 vector file

```
PATTERN:
address   = 0 3 3 3 3 3 3 3 0 0 2 2 2 2 2 2 2 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            (0)*27
            0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 0 0
            (0)*;

inputdata = 0 17 17 17 17 17 17 17 0 0 12 12 12 12 12 12 12 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            (0)*27
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            (0)*;

EINP      = 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            (0)*27
            0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0
            (0)*;
```

```
IOE-      = 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            (1)*27
            1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1
            (1)*;

CWR-      = 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            (1)*27
            1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1
            (1)*;

CRD-      = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            (1)*;

DMAREQ    = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            (0)*;

addr      = (3)*;

SSS-      = Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z
            Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z
            (Z)*;

ssaddr    = Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z
            Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z
            (Z)*;

MAINCS-   = (1)*;

MAINWR-   = (1)*;

MAINRD-   = (1)*;

XTAL16    = (1 0)*;

TMSINT-   = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            (1)*29 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
            (1)*;

START     = (Z)*;

BUSY-     = Z Z Z Z 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            (1)*27
            1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            (1)*;

RESET-    = (1)*;
```

2.11. CSMTEST5 command file

This simulation test the reading of data by the main processor.

```
echo Starting CSM module functional test 5;
echo CSM module data read tests;
exec @CSMINIT;
echo 70 step simulation.....;
log @csmtest5;
group hex inputdata = INPD4 INPD3 INPD2 INPD1 INPD0;
group hex address = A15 A14 A13;
group hex addr = ADDR3 ADDR2 ADDR1 ADDR0;
group hex ssaddr = INPSS3 INPSS2 INPSS1 INPSS0;
vec @csmtest5;
cycle 2;
plot INPD4 INPD3 INPD2 INPD1 INPD0
     A15 A14 A13 CWR- CRD- EINP IOE-
     BUSREQ- CTR0 CTR1 DMAREQ MAINOP INT1-  START.INP START
     SSS-.INP INPSS3.INP INPSS2.INP INPSS1.INP INPSS0.INP
     ADDR3 ADDR2 ADDR1 ADDR0
     RXENOP BUSYOP READYOP;
sim +70;
view;
save @CSMTEST5;
```

2.12. CSMTEST5 vector file

```
PATTERN:
address   = 0 2 2 2 2 2 2 2 0
            0 3 3 3 3 3 3 3 0 0 3 3 3 3 3 3 3 0 0 0 0 0 0 0 0
            (0 2 2 2 2 2 2 2 0)*3
            0 1 1 1 1 1 1 1 0
            0 2 2 2 2 2 2 2 0
            (0)*;

inputdata = 0 0 0 0 0 0 0 0 0
            0 Z Z Z Z Z Z Z 0 0 Z Z Z Z Z Z Z 0 0 0 0 0 0 0 0
            (0 Z Z Z Z Z Z Z 0)*3
            0 4 4 4 4 4 4 4 0
            0 Z Z Z Z Z Z Z 0
            (0)*;

EINP      = 0 0 0 1 1 1 1 1 0
            0 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0
            (0 0 0 1 1 1 1 1 0)*3
            0 0 0 1 1 1 1 1 0
            0 0 0 1 1 1 1 1 0
```

```
            (0)*;

IOE-        = 1 1 0 0 0 0 0 0 1
              1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1
              (1 1 0 0 0 0 0 0 1)*3
              1 1 0 0 0 0 0 0 1
              1 1 0 0 0 0 0 0 1
              (1)*;

CWR-        = 1 1 0 0 0 0 0 0 1
              1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
              (1 1 1 1 1 1 1 1 1)*3
              1 1 0 0 0 0 0 0 1
              1 1 1 1 1 1 1 1 1
              (1)*;

CRD-        = 1 1 1 1 1 1 1 1 1
              1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1
              (1 1 0 0 0 0 0 0 1)*3
              1 1 1 1 1 1 1 1 1
              1 1 0 0 0 0 0 0 1
              (1)*;

DMAREQ      = 0 0 0 0 0 0 0 0 0
              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
              0 0 0 0 0 0 0 0 0
              0 0 0 0 0 0 0 0 0
              1 1 1 1 0 0 0 0 0
              0 0 0 0 0 0 0 0 0
              (0)*;

addr        = 2 2 2 2 2 2 2 2 2
              2 2 2 2 2 2 2 2 2 2 D D D D D D D D D D D D D D D
              (3)*;

SSS-        = Z Z Z Z Z Z Z Z Z
              Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z
              0 0 0 0 0 0 0 0 0
              0 0 0 0 0 0 0 0 0
              1 1 1 1 1 1 1 1 1
              1 1 1 1 1 1 1 1 1
              1 1 1 1 1 1 1 1 1
              (Z)*;

ssaddr      = Z Z Z Z Z Z Z Z Z
              Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z
              3 3 3 3 3 3 3 3 3
              3 3 3 3 3 3 3 3 3
              (Z);

MAINCS-     = (1)*;

MAINWR-     = (1)*;

MAINRD-     = (1)*;
```

```
XTAL16     = (1 0)*;

TMSINT-    = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
             (1)*;

START      = 0 0 0 0 0 0 0 0 0
             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
             0 0 0 0 0 0 0 0 0
             1 1 1 1 1 1 1 1 1
             (1)*;

BUSY-      = (Z)*;

RESET-     = (1)*;
```

2.13. CSMTEST6 command file

This simulation tests the reset function of the device. It does not require the initialisation routines.

```
echo Starting CSM module functional test 6;
echo Reset test;
echo 15 step simulation.....;
log @csmtest6;
group hex inputdata = INPD4 INPD3 INPD2 INPD1 INPD0;
group hex address = A15 A14 A13;
vec @csmtest6;
cycle 2;
plot AS0 AS1 AS2 STX SELECTOP READYOP WAITOP START SAENOP
     BUSREQ- MAINOP TXENOP TMSINT INT1- RESET-;
sim 15;
view;
save @CSMTEST6;
```

2.14. CSMTEST6 vector file

```
PATTERN:

RESET-      = 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 (1)*;

CWR-        = (1)*;

CRD-        = (1)*;

IOE-        = (1)*;

EINP        = (0)*;

XTAL16      = (1 0)*;

TMSINT-     = (1)*;

DMAREQ      = (0)*;

MAINCS-     = (1)*;

MAINRD-     = (X)*;

MAINWR-     = (X)*;

ADDR0       = (0)*;

ADDR1       = (1)*;
```

```
ADDR2       = (1)*;

ADDR3       = (1)*;

START       = (Z)*;

BUSY-       = (Z)*;

INPSS0      = Z (X)*;

INPSS1      = Z (X)*;

INPSS2      = Z (X)*;

INPSS3      = Z (X)*;

SSS-        = Z Z Z Z Z 1 1 1 1 1 1 1 1 1 1 (1)*;
```

A.4 PROCESSING SECTION DESIGN

A.4.1 The CPU Section

a) The 80188 Processor

The 80188 is a highly integrated microprocessor which combines a large number of the most common 8088 system components on a single chip. A block diagram of the 80188 is shown in Fig. A.15. As shown here it consists of the a DMA unit, timers, interrupt controller, clock generator, and a chip select unit. All are housed in a 64 pin package, external circuit connections being shown in Fig. A.16.

i) Clock Generator

The inputs X1 and X2 provide an external connection for a fundamental mode parallel resonant crystal for the oscillator. The crystal frequency selected is double the CPU clock frequency. Here an 8 MHz crystal is used to generate a 4 MHz clock signal for the 80188.

ii) Interrupt Controller

The 80188 programmable interrupt controller (PIC) can handle interrupts which are generated by either software or hardware. A table containing up to 256 pointers defines the proper interrupt service routine for each interrupt. Interrupts 0-31 are reserved for predefined interrupts which may be activated either by software or hardware. The software interrupts are generated by specific instructions (INT, ESC, unused OP, etc.) or the result of conditions specified by instructions (DIV, IDIV, etc.). The hardware interrupts are divided into two groups; internal and external. The internal interrupts are:

DMA 0: Used in channel 0 (transmission).

DMA 1: Used in channel 1 (reception).

TIMER 0: Used in application software.

TIMER 1: Used in application software.

TIMER 2: Used in application software.

The external interrupts are;

INT0: Connected to the 8087 numeric processor

INT1: Connected to the 2681 DUART

INT2: Not used

INT3: Not used

NMI: Watchdog timer

All these interrupts are maskable except the NMI interrupt which is connected the watchdog timer.

The internal interrupts DMA 0 and DMA 1 are used in this design to detect DMA transfer termination (in channel 0 and 1). The external interrupt INT0 is connected to the 8087 which uses it to indicate that unmasked exceptions have occurred during numeric instruction execution. INT1 is connected to the 2681 DUART to support interrupt handling of the communication process instead of polled operation.

iii) DMA Unit

The DMA unit provides two high speed DMA channels. Data transfer can be performed to or from any combination of memory and I/O space in byte form. A transfer count is also maintained in order to allow termination of DMA transfers after a pre-programmed number of transfers. Each data transfer consumes 2 bus cycles (a minimum of 8 clock periods), one cycle to fetch data and the other to store data.

The two external DMA request inputs, DRQ0 and DRQ1, are connected to the OBI interface. DRQ0 is activated when data is to be transferred from the processing to the communication section, while DRQ1 is activated when data is to be transferred from the communication to the processing section. The controller has the option of producing an internal interrupt when the transfer count reaches zero. This interrupt is used to inform the main processor that message transfer has been completed.

iv) Chip Select Unit

The integrated chip select unit provides programmable chip-select logic which can be used to select memory or peripherals (6 memory and 7 peripherals are provided) during processor controlled read or write operation. Note that these become inactive if the processor is forced into the "Hold" state.

The memory chip select lines are split into three groups for separately addressing the major memory areas in the system:

* 1 Upper memory (UCS*) - for reset EPROM (bootstrap).
* 1 Lower memory (LCS*) - for lower RAM area (stack, data, and heap).
* 4 Mid-range memory (MCS0* - MCS3*) - for the application software.

The size of each of these areas, and the starting location of the mid-range memory are user programmable, with some restrictions.

Each of the peripheral chip select lines (PCS0* to PCS6*) address one of seven adjacent 128 byte blocks whose base address is programmable. This block can be programmed to be part of the memory or in a separate I/O block.

The chip select lines are connected as follows;

UCS0* : EPROM

LCS0* : RAM

MCS0* : Reserved for application software.

MCS1* : Reserved for application software.

MCS2* : Reserved for application software.

MCS3* : Currently used for Modula-2 application software.

PCS0* : 2681 DUART

PCS1* : OBI interface - MAINCS*

PCS2* : OBI interface - DMAREQ*

PCS3* : watchdog timer

PCS4* : reserved

PCS5* : reserved

PCS6* : reserved

Each of the programmed chip select areas has a set of programmable ready bits associated with it. These ready bits control an integrated wait state generator which is programmable to provide 0 to 3 wait states for all accesses to the area of memory associated with a chip select signal.

v) Programmable Timers

The timer unit provides three independent 16-bit timers/counters. Two of these timers are available for use external to the CPU whilst the third timer is available only for internal use. All three timers operate independently of the CPU. In this design all external connections of the timer signals are unused.

b) Numeric Processor Extension (8087)

The 8087 is a numeric processor extension that provides arithmetic and logical instruction support for a variety of numeric data types. It executes numerous built-in functions such as tangent, log, exponential, etc.

The 8087 can execute numeric instructions approximately 100 times faster than a 80188 operating at the same speed.

As a coprocessor to the 80188, the 8087 is wired in parallel with the CPU (see Fig. A.17). The CPU's status (S0-S2) and queue status lines (QS0-QS1) enable the 8087 to monitor and decode instructions in synchronisation with the CPU and without any CPU overhead. All 8087 instructions appear as ESCAPE instructions to the 80188. Both the 80188 and 8087 decode and execute the ESC instruction together. The start of the numeric operation is accomplished when the CPU executes the ESC instruction. The 8087 can interrupt the CPU when it detects an error or exception, its interrupt being connected to INT0 of the 80188.

c) Advanced Bus Controller (82188)

The Intel 82188 generates system command and control timing signals as determined by the bus status lines signals (Fig. A.17). It also provides HOLD/HLDA -RQ/GT bus protocol exchanges; this allows it to be used where bus control mechanisms between devices differ, such as between the 80188 and the 8087. In this design some of the control signals are buffered to increase the drive capability.

d) Power-On Reset

The 80188 has a RES* input pin and a synchronised RESET output pin (Fig. A.16). The RES* input is provided with a Schmitt-trigger to allow power-on reset via an R-C network, the corresponding RESET output lasting an integer number of clock periods determined by the length of the RES* signal.

e) Address/data Bus Buffers

The 80188 has a time multiplexed address/data bus consisting of 8 lines (A/D0-A/D7) together with various control and status signals. The multiplexed lines are connected to latches (74LS573) which provide a demultiplexing function for the address bus signals (Fig. A.18). These are controlled by the advanced bus controller (82188) which generates the demultiplexing signal.

The high address bus (A8-A19) is also buffered (74LS645); this, together with the address latches, ensures that the address bus has a high drive capability on all its signal lines.

A.4.2 Memory

Three 28 pin memory sockets are provided to host EPROM or static RAM (SRAM) devices (Fig. A.19). Various sizes of EPROM (from 16K to 64K Byte) and SRAM (from 2K to 32K Byte) may be used in this design. The main board (processing section) currently uses the following configuration;

* One EPROM (size 8K byte)- used as a bootstrap.
* One SRAM (size 8K byte)- used as a memory (for stack, data and heap).
* One EPROM (size 32K byte)- used for the application software.

On reset the 80188 begins execution at address FFFF0H, thus a jump instruction must be inserted at this location to transfer execution to a bootstrap program. Consequently an EPROM chip must be mapped into the top of the memory (i.e. first EPROM). It contains the initialisation for the main program software, its chip select pin being connected to UCS*.

The 80188 uses locations 0H-3FFH (1K Byte) for its interrupt vector table. This vector table allows different interrupt types to be serviced. The 80188 also needs RAM for the storage of data variables, flags and stack. In this design an 8K SRAM is placed in location 0H to 1FFFH, its chip select pin being connected to LCS*.

The dynamic RAM store (DRAM) is located on a separate piggy back board as an option. It consists of sixteen 256K x 1bit DRAM I.C.s (1/2 mega-byte total), controlled by an Intel DRAM controller (8208), a set of data bus buffers (74LS245's) and associated control circuitry.

A.4.3 Serial Communications

The Signetics 2681 DUART provides two independent full-duplex channels in a single chip (channels A and B). The DUART has a software programmable transmission format (number of data bits, stop bits, parity, etc), programmable baud rate, error detection, multifunction counter/timer, 7-bit input port, 8-bit output port, interrupt system and on-chip oscillator. The circuit diagram of the serial communication system is shown in Fig. A.20.

To provide signals to meet RS 232C specifications a MC1488 line driver and a MC1489 receiver are used.

To ensure that the output slew rate of the line driver conforms to the RS232C specifications (30V/us) 390pF capacitors are connected between the outputs of the line drivers and 0V.

A.4.4 OBI Interface

The aim of this interface (Fig. A.21) is to transfer data between the main processor and the on-board interface block by using the DMA controller in the 80188.

The following table lists the interface signals with their functions;

TABLE A-8: OBI Interface Signal Description

Signal	Type	Function
PCS2*	O	Sets a DMA request flag to the OBI (DMAREQ*)
DRQ0	I	Channel 0 DMA request
DRQ1	I	Channel 1 DMA request
PCS1*	O	Chip select signal to the OBI (MAINCS*)
WR*	O	Data write enable (MAIN WR*)
RD*	O	Data output enable (MAIN RD*)
D0-D7	I/O	Data signals

The OBI interface gives the processing section the right to access the communication section's temporary storage RAM. It enables the processing section to;

* Access the communication section's temporary storage (MAINCS*) for a read operation (MAINRD*) or a write operation (MAINWR*),
* Signals the communication section (DMAREQ*) for a request of data transfer (RDT) and at the end of data transfer (EDT).

All data is exchanged between the communication section and the main processing section using direct memory access (DMA) techniques.

A.4.5 Ancillary Circuits

a) Single Step Control

The single step circuit (Fig. A.22) is included to aid de-bugging and testing of both hardware and software. Using this, a program can be executed one step at a time, making examination/testing of on-board devices more convenient. When the single step control is switched in the processor enters a continuous wait state. By pressing the "step" push button the wait (i.e. "not ready") signal is temporarily removed, allowing the processor to complete one bus cycle only. At this point it re-enters the continuous wait condition.

The single step circuit is switched into the ARDY line by the toggle switch. Two NAND gates are used to debounce the push-button switch, so that when it is pressed and then released a single positive pulse is produced. When flip-flop 1 receives a positive going edge from the de-bounce circuit its Q1 output is set high. This takes the D2 input of flip flop 2 high. On the next positive going edge of the CPU clock the Q2 output of 2 goes high, sending ARDY high, and the Q2* output of 2 goes low, clearing flip-flop 1. Since flip-flop 1 has now been cleared the D2 input to flip-flop 2 is now low. On the next positive going edge of the CPU clock the Q2 output of 2 (ARDY) goes back low. Thus the ARDY line has gone high for one CPU clock cycle. When the push-button is released flip-flop 1 receives a negative going edge from the debounce circuit, but this has no effect.

b) Watchdog Timer

The watchdog timer provides a mechanism of program recovery in case of failure (program crash). This is done using a non-maskable interrupt (NMI). The Watchdog timer (Fig. A.23) comprises a retriggerable monostable (74LS123) which is triggered by writing to a specific address. This is done repeatedly under program control so that, in normal circumstances, the monostable is always retriggered before its period expires. If the program crashes, the timer expires and so causes a non-maskable interrupt (NMI).

In normal operation, when the monostable receives a negative going edge on its A1 input (from decoded address and PCS3*) the output Q* goes low. This assumes that OP5 from the 2681 is low, otherwise Q* remains high. It stays low for a time determined by the resistor/ capacitor combination. Provided the timer is selected (addressed) before the end of its time-out period Q* stays high, the timer being re-triggered (normal operation). If the timer is not re-selected before the end of the time-out period Q* goes low, causing a NMI.

This forces the processor to go through a pre-programmed interrupt service routine to recover from the fault condition. The values chosen for the timing components (R3 and C2) are selected to give a one second time-out period. The primary purpose of the control signal from the 2681 DUART (OP5) is to allow power-on initialisation to be completed without having to cope with an instantaneous NMI request. It also ensures that the watchdog timer doesn't cause accidental interrupts when not in use.

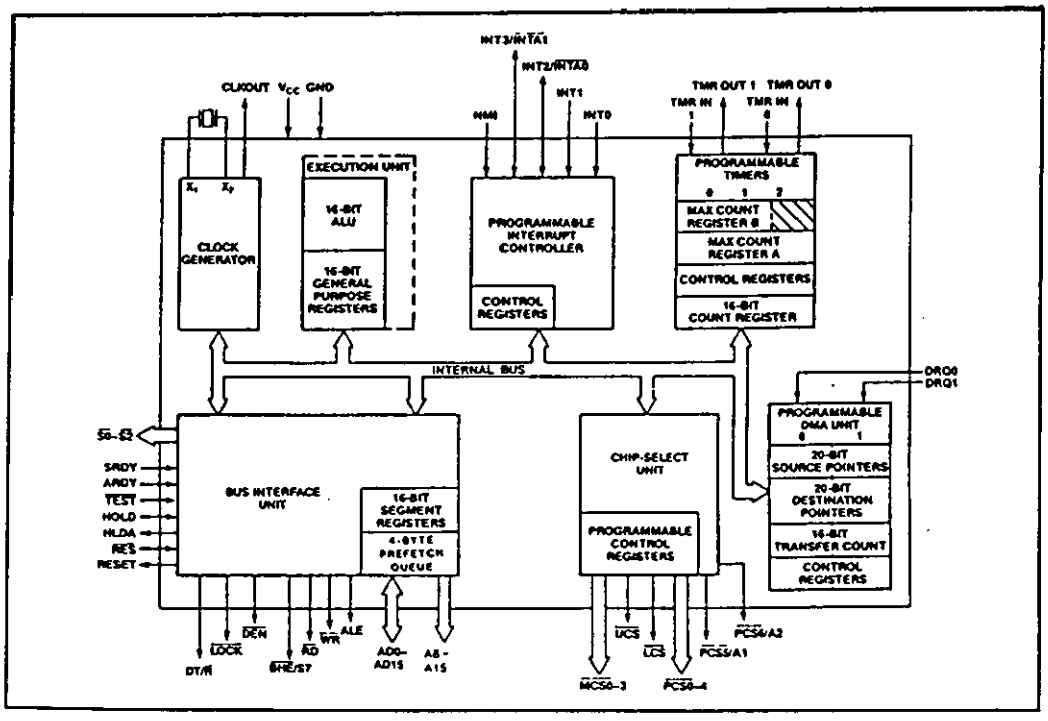


Fig. A.15 80188 CPU BLOCK DIAGRAM

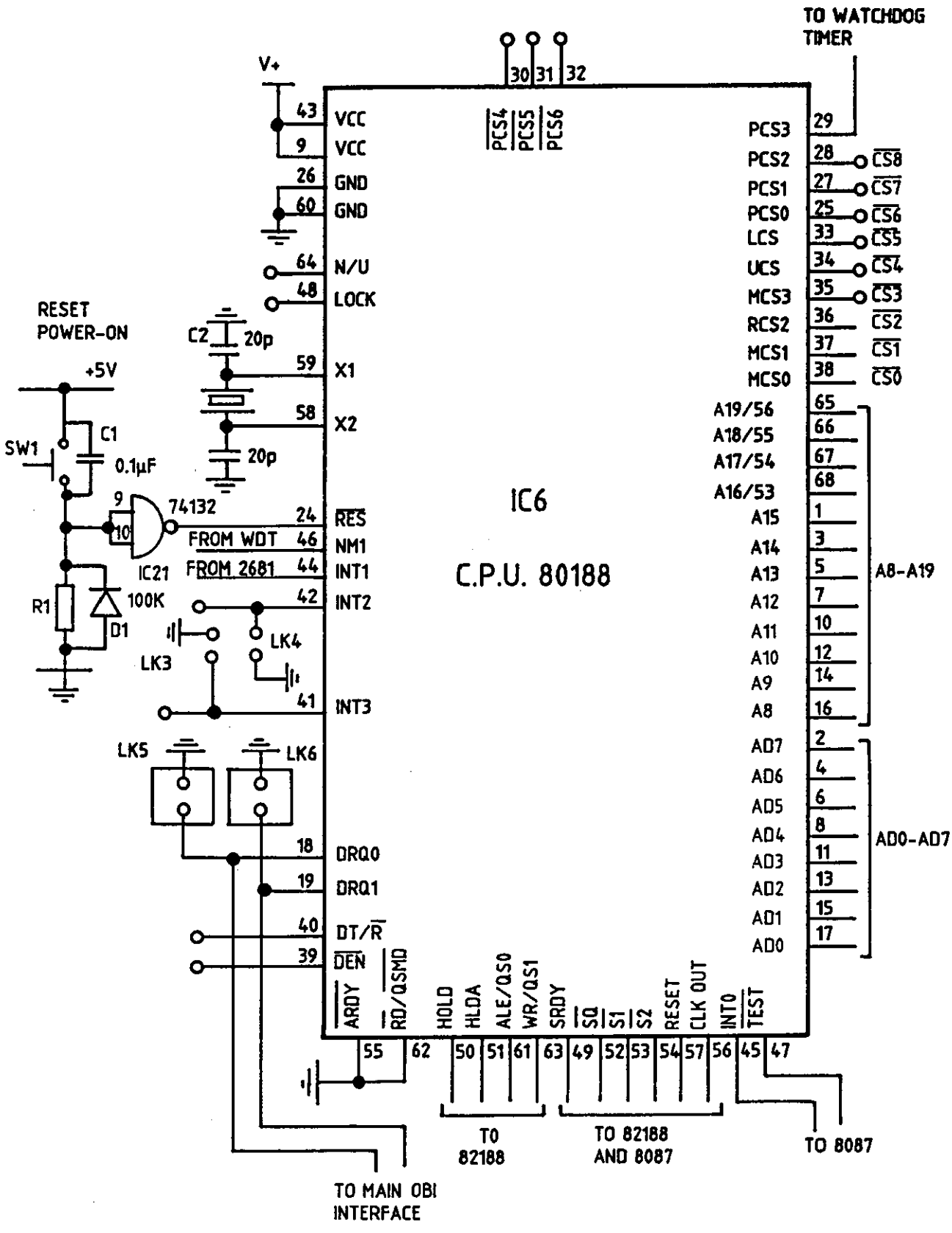


Fig. A.16 80188 CPU CONFIGURATION

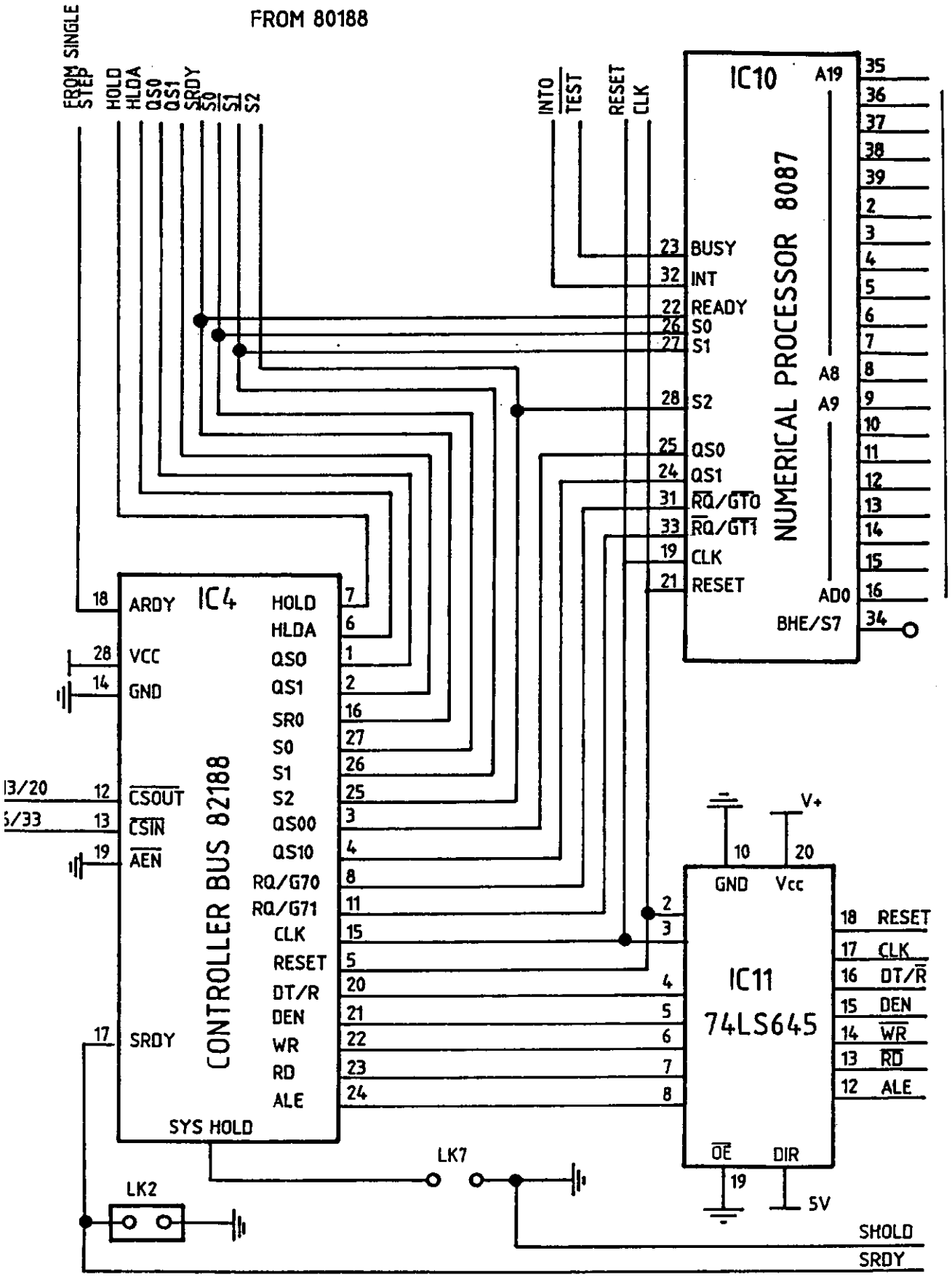


Fig. A.17 82188 CONTROLLER AND 8087 NUMERICAL PROCESSOR

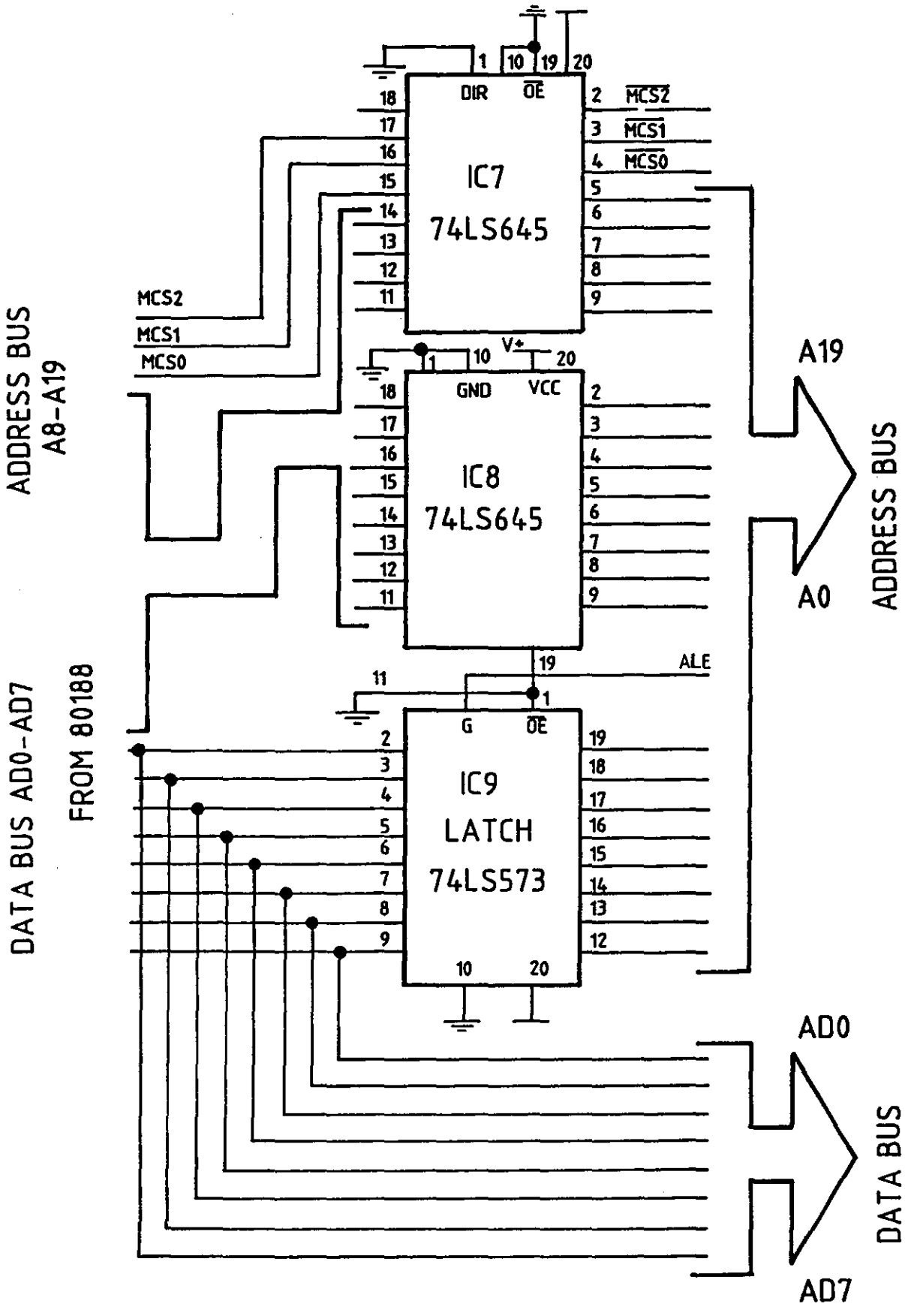


Fig. A.18 ADDRESS/DAT BUFFERS AND LATCHES

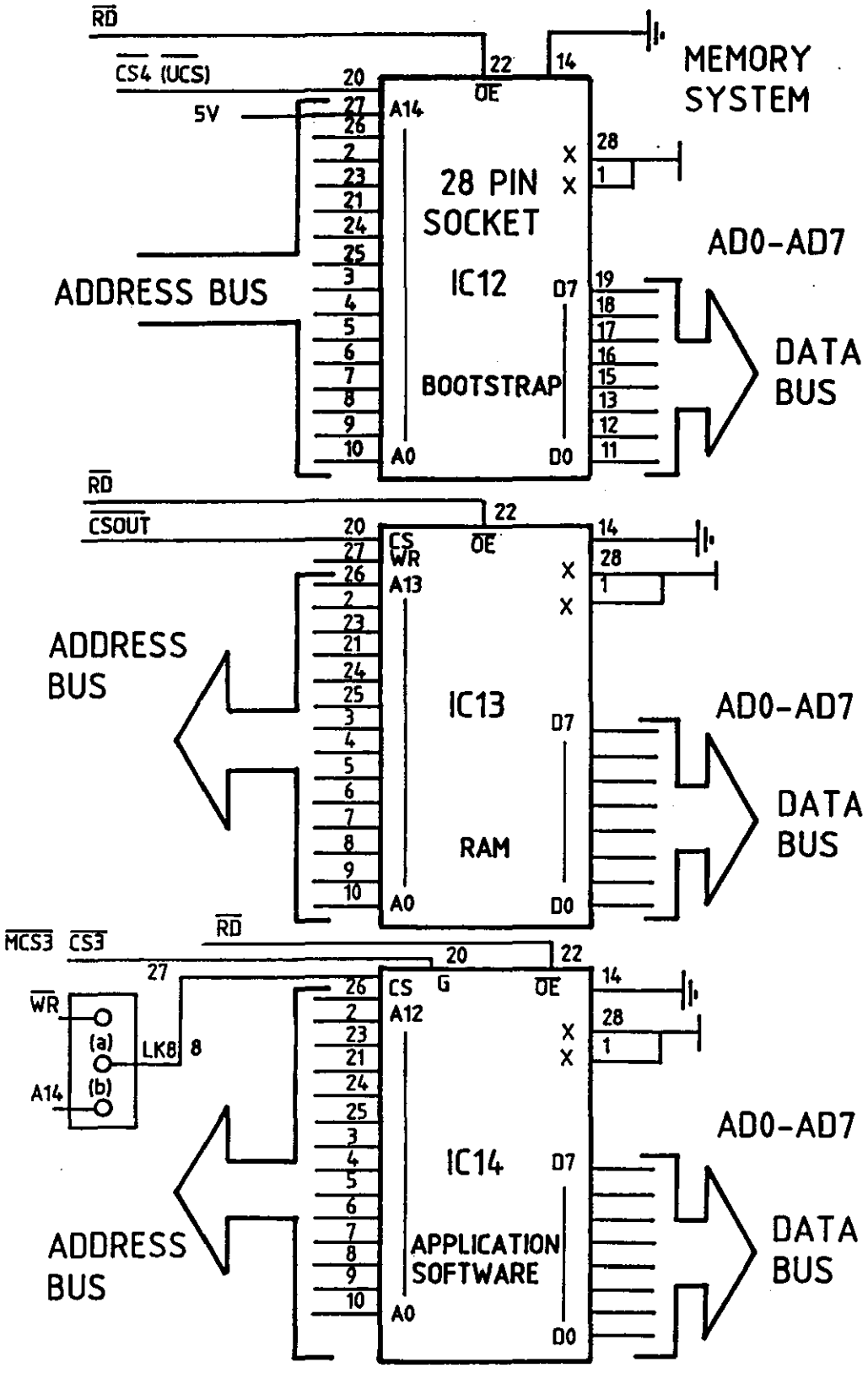


Fig. A.19 MEMORY SYSTEM

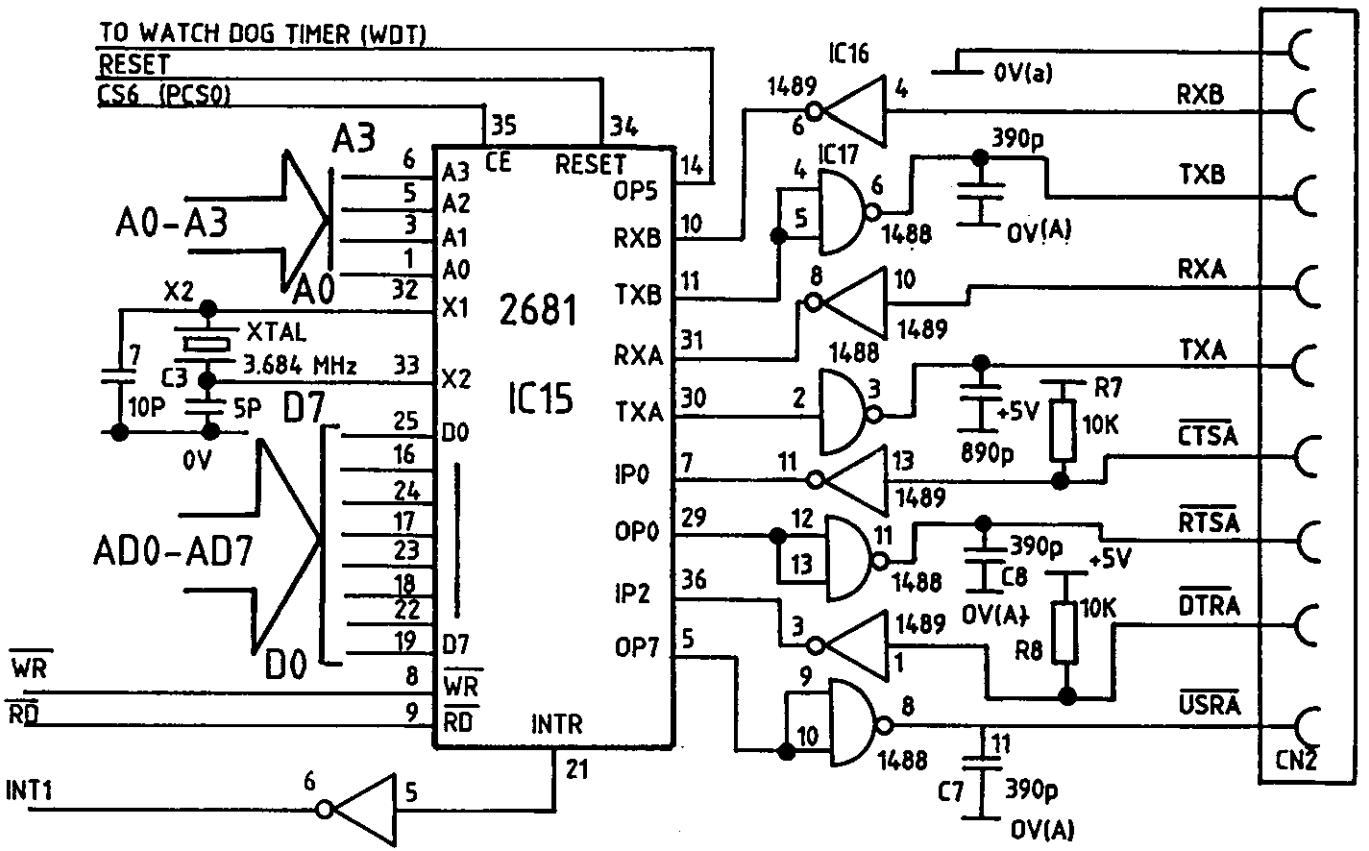


Fig. A.20 SERIAL COMMUNICATION SYSTEM

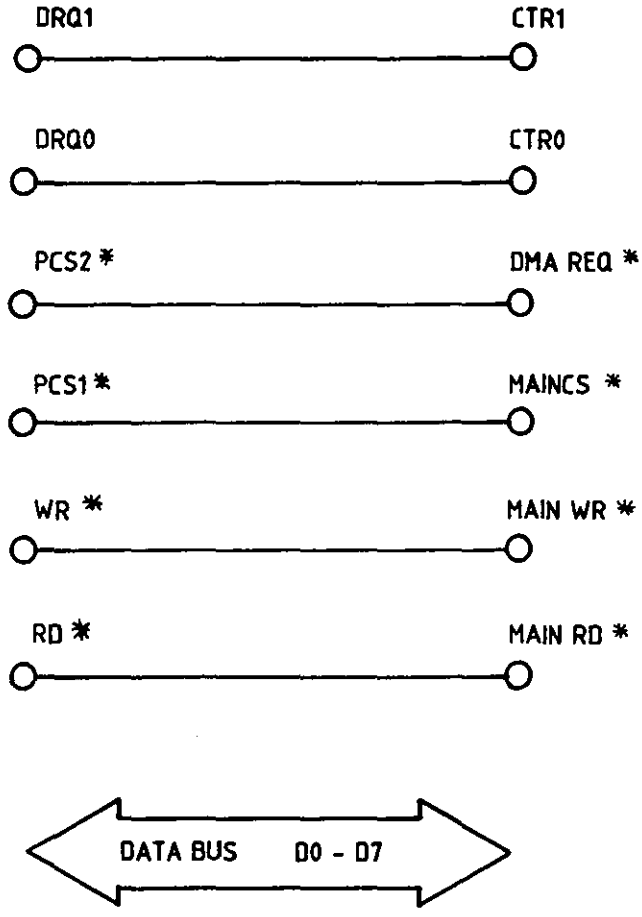


Fig. A.21 ON-BOARD INTERFACING BLOCK (OBI)

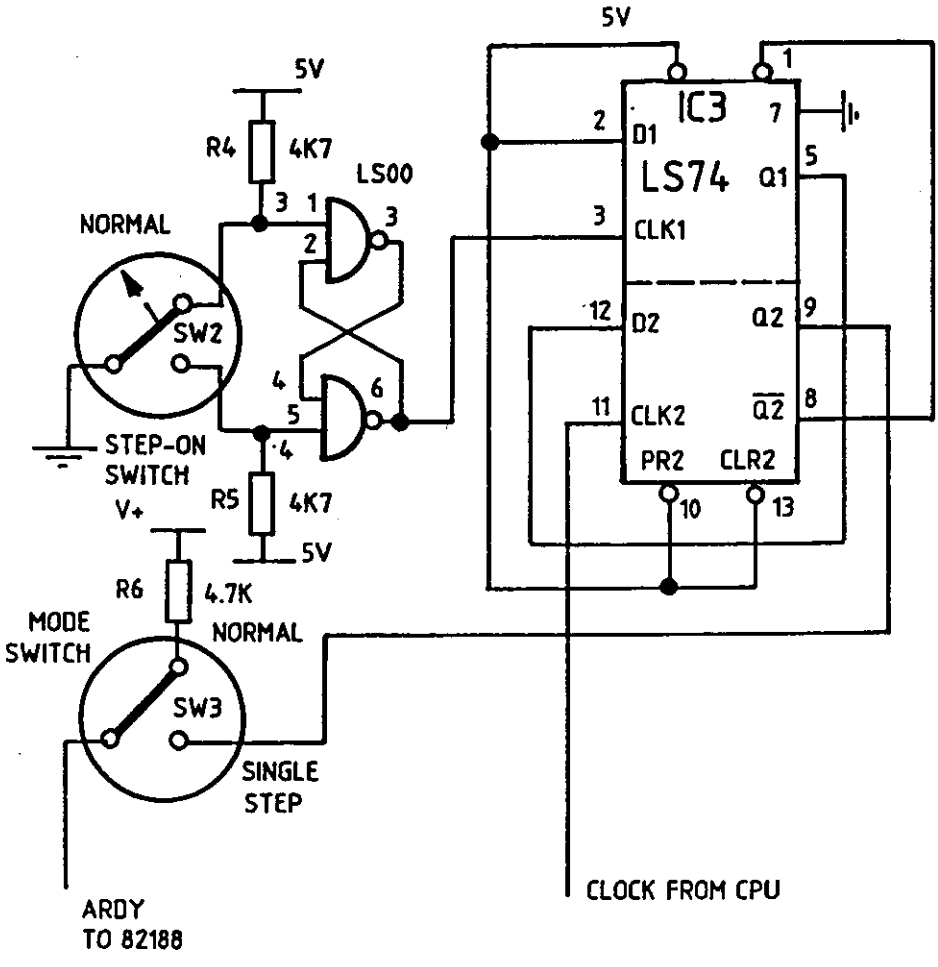


Fig. A.22 SINGLE STEP CIRCUIT

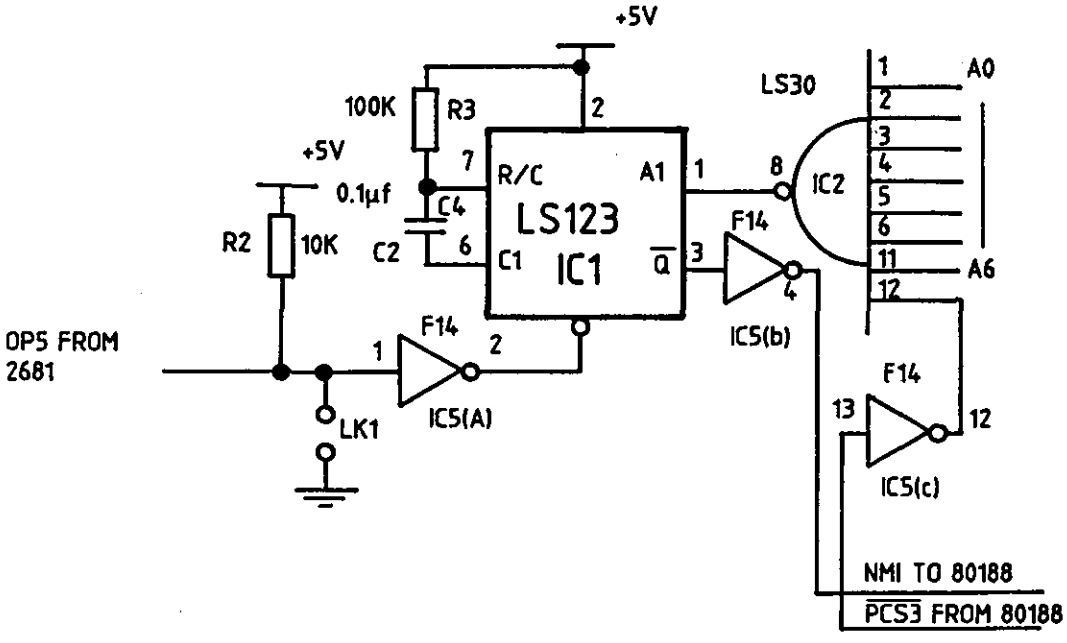


Fig. A.23 WATCHDOG TIMER

APPENDIX – B

APPENDIX B

COMMUNICATION SECTION'S MODES OF OPERATION

B.1 GENERAL

This section describes how the communication software is set up to manipulate the hardware for transmission and reception of data frames over the network. The communication section has five modes of operation; initialisation, no operation (idle), reception, transmission, and data exchange with the processing section's interface (OBI). The selected mode of operation is determined by the communication section which receives requests for each mode. Each mode is explained in the following sections.

B.2 INITIALISATION

On power-up or reset, the communication CPU and the CSM modules are cleared to a basic initial state, as explained in sections B.2.1 and B.2.2 below, with most actions disabled. This means that only a few actions need to be taken to set up these devices. The TMS module, however, is only cleared after a power-up. This requires setting up all its registers appropriately after a CPU reset.

B.2.1 Communication CPU

The main functions that must be set up in the communication CPU include: wait state generator, handling of interrupts, serial interface, watchdog timer, and the internal timers required by the software.

The wait state generator automatically resets to the maximum number of wait states for both memory and I/O space accesses. This means three wait states for memory and four wait states for I/O accesses. The number of wait states actually required by memory is largely determined by the speed of storage devices implemented. Devices with access times of 200 nS or less, require one wait state only. In I/O access, the number of wait states required depends greatly on the speed of the CSM and TMS modules. Theoretically, one wait state is sufficient. For safety purposes, however, an extra wait state is inserted with very little effect on system performance.

Interrupts are generated both internally and externally. External interrupts are generated by the CSM module and the watchdog timer. Watchdog timer generates a non maskable interrupt (NMI). The interrupt vector set-up is also fixed, so no further action is required after a reset. The only action required is to initially trigger the watchdog timer. The required registers for this action are set up during the initialisation sequence, transfer being initiated when required.

All other interrupts are masked after a reset. Signal operations that generate an INT1 signal from the CSM module do not have to be handled by interrupts. All lines can be examined by read operations to appropriate registers within the CSM module. For complex operations, however, it is suggested that signal operations are implemented through interrupts as this removes the need for polling the system bus continuously.

For most applications, the serial line interface is required for monitoring the system operations. This requires setting up the appropriate baud rate, and possibly an internal interrupt for data reception.

B.2.2 CSM module

Most lines of this module, apart from the WAIT signal, are set to an inactive state by the reset signal. This ensures that no system initialisation takes place before activating the WAIT signal. All the other lines connected to the system bus are set to a tri-state condition to avoid any contention. The only action required is to set the SELECT line, allowing the communication CPU to access the TMS module.

B.2.3 TMS module

This module requires a complete reset by software. Functions not required by the module are also cleared by software due to their undefined state as described earlier. Most of the registers within this module change state, depending on the operation being carried out, and hence there is no need for a reset. The only exception is the interrupt control, which can be left enabled at all times as it is gated off externally.

B.3 NO OPERATION - IDLE

In this state the communication section carries no operation (i.e. data transfer) with any destination. It simply keeps monitoring the state of the RXEN line, checking for either system bus activity, or a request by the processing section to access the TMS module. Both

activities are indicated by an interrupt to the communication CPU. If interrupts are not enabled, however, then the communication CPU has to poll the COMMS CONTROL REGISTER at regular intervals to detect either of these activities.

B.4 RECEPTION

Reception mode is requested when RXEN signal is active causing an interrupt to the 64180 CPU. When the communication section is in the idle mode then the message coming from a remote station can be handled immediately. If the communication section is in another mode, however, DMA transfer is then suspended and bus control is returned to the communication CPU. Data transfers to/from the processing section are checked. DMA transfers are re-issued, should data transfers are corrupted.

When RXEN signal is invoked, it generates a BUSY signal over the system data bus. This holds back the transmitting station until BUSY is removed from the system bus. This is accomplished by the receiving station setting the READY line in the CSM module. This must only be done when the communication CPU has set up the TMS module to receive data. This includes setting up the data pointers in RAM. To set up the TMS module, the SELECT line in the CSM module must be set, enabling the communication CPU to access the TMS module.

Once the READY line is set, the transmission cycle to this station commences, under the control of the transmitting station, as shown by time T2 in Fig. B.1. During this time no access is required to the TMS module.

The end of data reception is signalled by resetting the RXEN signal. This occurs when the transmitting station removes the address off the system bus and the SSS* line goes inactive. To detect end of transmission, the RXEN signal must be polled during the transmission cycle. During this time the interrupt line will be continually active and so must be masked off by the CPU until the end of transmission to this station, as signalled by clearing RXEN.

After a reception of data over the system bus, data should be cleared from the TMS module as soon as possible, so as to enable another reception to take place. If this is not done then bus traffic may be delayed.

B.5 TRANSMISSION

This mode is entered when there is a message to be sent to a remote station by the station which holds the 'Token'. In this case the communication CPU sets TMS9650 to point to the start of the message to be sent and programs the TMS9650 through its control register to activate an interrupt when last byte of the massage has been transmitted.

The first action of the transmitting station is to place the address of the receiving station on the system address lines and enable the SSS* line. This is achieved by writing the address to the station address register and enabling the SAEN signal which, in turns, enables the output buffers.

After the address lines have been set, the communication section must set up data pointers of the TMS module. To do this the SELECT line must be set to allow the communication section access the TMS module. The address pointers in the TMS module must be set so that port B points to the start of data and port A points to the end of data. The TMS module is programmed to activate an interrupt at the end of transmission.

The next operation is to set the STX and READY control lines through the communication control register. This action will then initiate the transmission sequence as soon as the receiving station releases the BUSY line. Once the BUSY line has gone inactive then transmission will start at the beginning of the next transmission cycle as determined by the signal TXCLK. This then sets the TXEN signal to start transmission.

The end of transmission operation is signalled by the TMS module when the two address pointers are equal. This generates an interrupt from the TMS module. When this is detected by the CPU, two actions must be taken. The CPU must reset the TMS interrupt line in the TMS module and also the TMS interrupt latch by writing to the appropriate interrupt control register. It must also reset the READY line as shown by T1 in Fig. B.2. If this is not cleared and another station attempts to transmit to this station, no busy signal will be generated until this station is ready.

B.6 DATA EXCHANGE WITH THE OBI INTERFACE

Data transfer between the processing section and the TMS module may result from two conditions. A request is initiated by the processing section when it has a message to transmit to a remote station. Alternatively, a request is initiated by the communication section when a network data frame has just been received in the TMS module.

The processing section signals its request for a transfer to the communication section by activating the DMAREQ line so setting up an interrupt to the communication section. The interrupt must be cleared by writing to the DMA interrupt register. This must be done before any other action takes place, as the same interrupt latch is used to signal the end of the transfer.

Once the communication section has determined that a transfer is required, it must set up the TMS data pointer, of port A, for the transfer. The next action is to clear the SELECT line to enable processing section's access. It must then signal to the processing section to start the transfer. Two lines are provided for this transfer; DMA0 and DMA1. One is used for transfers of data from the TMS module to the processing section and the other for transfers from the processing section to the TMS module. These signals are activated by write operations to the appropriate registers.

When a request for a transfer is sent by the communication CPU, the CSM module activates the BUSREQ* signal. The communication section will, then, release its data bus for the transfer, after a delay not exceeding 1 uS. This means that the processing section should wait at least 1 uS before starting the transfer. Otherwise, the buffer between

the processing and communication sections may not be enabled. During transfer, the communication section should poll the COMMS CONTROL REGISTER in the CSM module. It should check for a MAININT signal generated by a DMAREQ signal. This signal is activated by the processing section at the end of a transfer. During DMA transfer, the communication section is disabled as the processing section has control of its data bus. This operation is transparent to the communication software. It is suspended during the transfer and is re-initiated after the completion of the transfer when the MAININT line is set again.

Data transfer between the TMS module and the processing section may end prematurely by the RXEN signal activated by the CSM module. This generates an INT1 signal to the communication section, releasing its bus from the transfer process. Hence, the communication section must poll the COMMS CONTROL REGISTER in the CSM module to detect whether an end of transfer is caused by a valid end signal from the processing section or else by an RXEN signal indicating the start of a reception mode over the system bus.

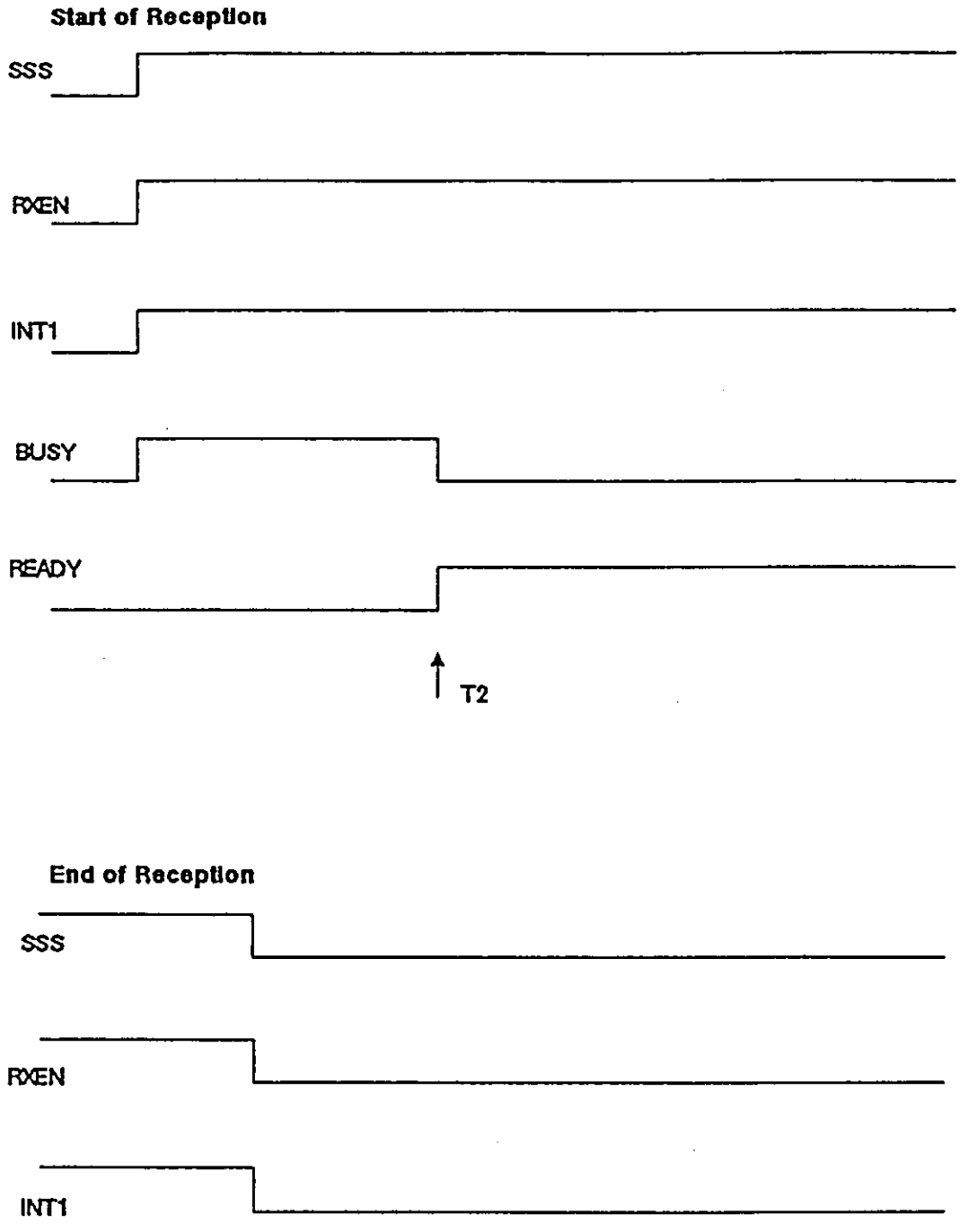


Fig. B.1 DATA RECEPTION MODE - SIGNAL TIMING

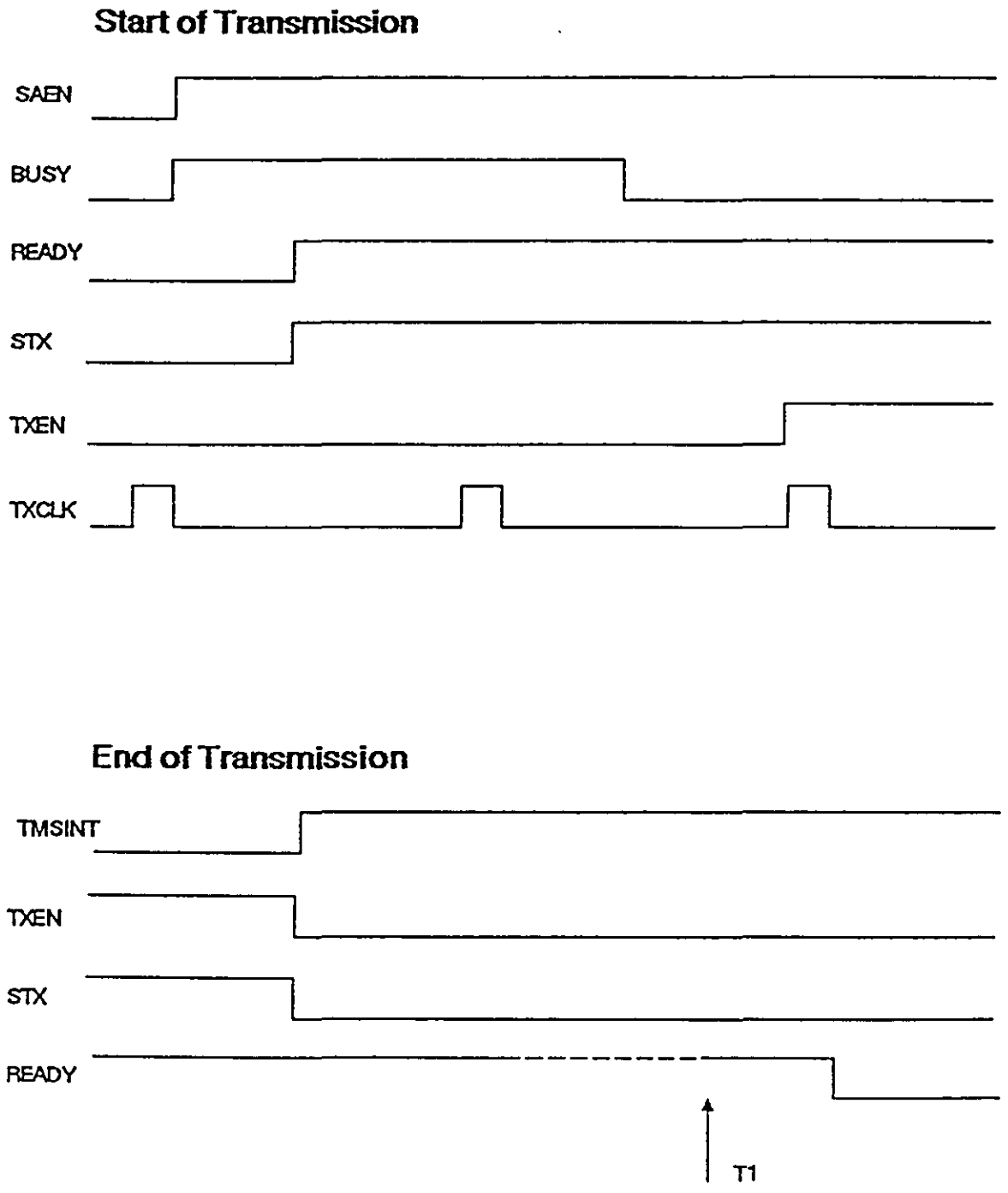


Fig. B.2 DATA TRANSMISSION MODE - SIGNAL TIMING

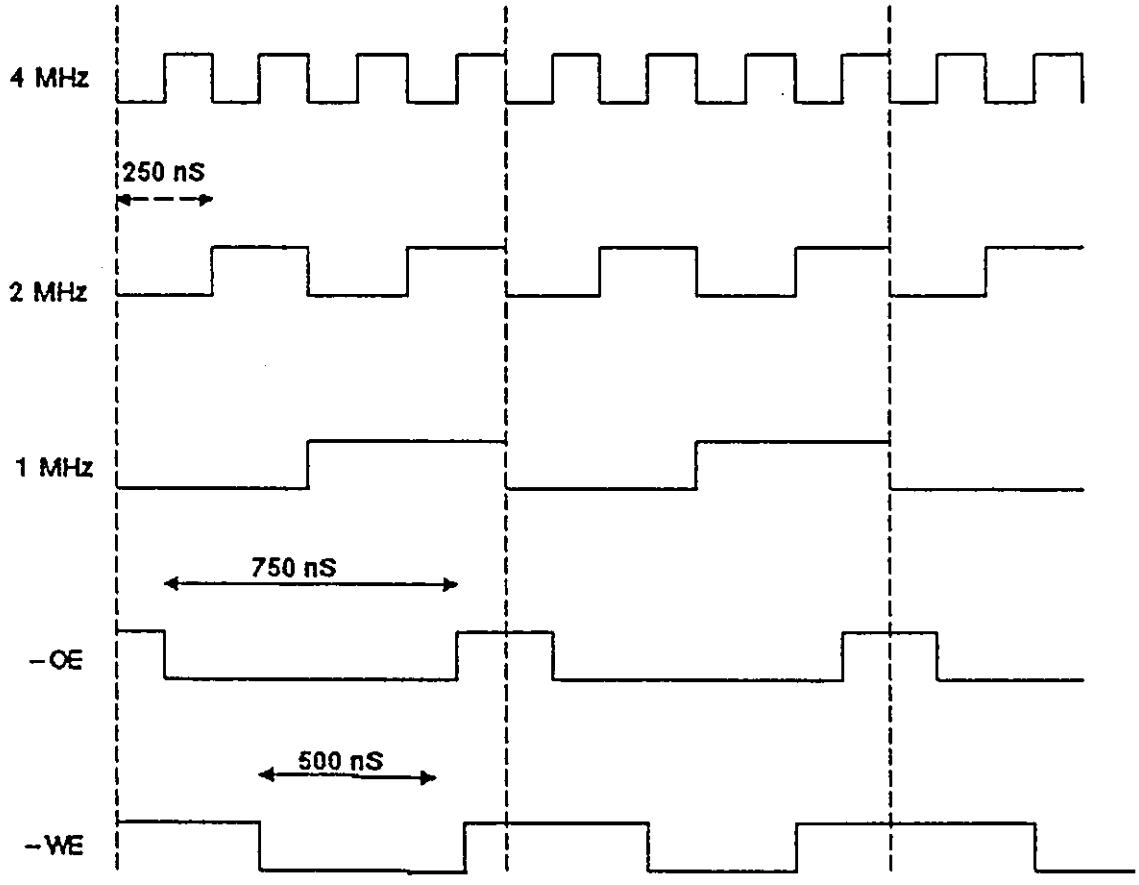


Fig. B.3 TRANSMISSION CLOCK SIGNALS

APPENDIX – C

APPENDIX C

MULTI-PROCESSOR SYSTEM - COMMUNICATION SOFTWARE STRUCTURE

C.1 RING CONFIGURATION AND MAINTENANCE

The basic concepts of token passing bus access method (TPBAM) were laid down in Chapter 4. In this section, token ring construction and maintenance are described. Before a station can start exchanging messages with other stations, it must know some information about other stations in the network (Fig. C.1). This can only be achieved once the logical ring has been established. Hence there must be an active configuration process (refer to Fig. C.2).

Once the ring is formed it has to be maintained. Stations may be allowed to enter or leave the network. Each station must periodically allow new stations to enter the ring. On the other hand, a station may exit the ring either as part of a normal operation or as a result of a failure. In either case, the ring must be reconfigured to accommodate such a change.

C.1.1 Ring configuration

On power-up, each station is supplied with its own address. To establish some information about other stations in the ring, each station uses a hardware timer called the response timer (RT). RT is directly proportional to the station's address. All stations activate their response timers (RT) simultaneously. The time-out period is directly proportional to the station's address. This means that the station with the lowest address has the shortest time-out period. In the example illustrated in Fig. C.2 this would be station 4. When

station 4 times-out, it sends a bus message to all stations called 'claim token' informing them that it is the first station in the ring. When other stations receive the message, they set their first station's address (FS) and wait for further bus messages. At this point station 4 has the token but still has no information concerning its successor (NS) or predecessor (PS).

The first station next sends a 'who follows' message. Included in this is the address of the sending station. On reception of this message, other stations activate their response timers once more. The first one to time-out is that with the next highest address in the network (station 9); it responds by sending a 'set successor' frame to the sender of the 'who follows' frame informing it of the next station in the ring. Station 4 now sets its NS address, and station 9 its PS address. Once this process has been done, the token can be passed on to station 9. Station 9, then, acknowledges reception with a 'token ack' frame. Hence, the link between station 4 and station 9, at this stage, has been patched.

Station 9 now goes through the same procedure to link with station 13. Note that, however, when station 9 issues a 'who follows' frame, only stations which have not already established a PS value may respond. Further, since station 4 has the lowest time-out period, it must reset its time-out to the maximum possible period once it has patched the link with its successor. If this has not been done, station 9 would patch a link to station 4 and the ring would appear to be formed correctly, even though only two stations are involved.

This process is repeated at each station. The logical ring is completely formed when the first station receives the token from the last one in the ring (station 13). Once the final link is patched the first station sends a 'set last' frame to all the stations defining the last station in the ring. This is followed by a message which specifies the total number of stations in the ring.

Configuration is completed when the first station finally sends an 'init done' message at which stage the system enters the operational mode.

C.1.2 Addition of a Station

During normal operations a station holding the token will periodically send a 'solicit successor' frame. This invites stations with an address between itself and the next station in logical sequence into the ring (see Fig. C.3). The transmitting station then waits for a time relative to the next station's address (the address of any station between TS and NS cannot exceed NS). Two events can occur:

* No response - there are no stations wishing to enter the ring. The token is passed-on as normal.
* A response - If there is a station that wishes to enter the ring it then sends a 'set successor' frame. The token holder sets its NS to the new station and passes the token to the new station. The station that was next to the station that has sent the 'solicit successor' frame sets its PS value to the address of the new station.

In the addition of a new station, the following points have to be taken into consideration (refer to Fig. C.3):

* If a number of stations between TS and NS are waiting to enter the ring, the one with the lowest address responds first and gains entry. The others, however, have to wait for another invitation.
* All stations increase their record of the number of stations by one.
* If the new station's address is less than the first station then all stations update their FS address to be the new station.
* If the new station's address is greater than the last station then all stations update their LS address to be the new station.

C.1.3 Deletion of a Station

A station may exit the ring if either a fault occurs or as part of its normal operation (i.e. drop-out). The two cases are described below:

a) Station failure

Failure may be detected in one of two ways. In the first case, if the token is transmitted to a defective station and no response has been detected (i.e. no acknowledgement has been received). The sender will, then, start reconfiguring the ring. It does this by sending a 'who follows+' frame (see Fig. C.4). The next operational station responds by sending a 'set successor' frame. The sending station then passes the token to what is currently its new NS.

Alternatively, a station may fail whilst holding the token. This is detected by the next station in the ring waiting for the token. Its token rotation timer will time-out if it does not receive the token within a preset time. When this occurs, it claims the token by sending a 'claim token' frame.

b) Station drop-out

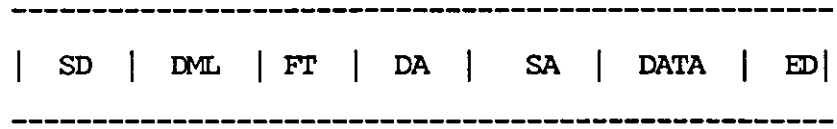
If a station wishes to drop-out as part of its normal operation, it waits until it receives the token. Then it issues a 'set successor' frame to its predecessor so that the link can be patched and the station dropping out can be by-passed.

A deletion of a station from the ring gives rise to the following change of information held by each station (refer to Fig. C.4):

* All stations decrease their record of the number of stations in the ring by one.
* The sender of the 'who follows' frame sets its NS to be the station next to the failed station.
* The station which was the next in succession to the failed station sets its PS to be the station that has sent the 'who follows' frame.
* If the failed station is the last station in the ring, then all stations update their LS to be the station that has sent the 'who follows' frame.
* Similarly, if the failed station is the first station in the ring then all the other stations update their FS to be the station next to the sender of the 'who follows' frame.

C.2 CONTROL FRAMES

This section describes briefly the frame formats used in sending messages between stations during the configuration and maintenance procedures. The general frame format is:



where:

SD	is the Start Delimiter	- one byte
DML	is the Data Message Length	- one byte
FT	is the Frame Type	- one byte
DA	is the Destination Address	- one byte
SA	is the Source Address	- one byte
DATA	is the Data Transmitted	- 1...120 bytes
ED	is the End Delimiter	- one byte

The data message length field contains the number of bytes in the data field.

The frame type field determines the response needed when a message has been received. The following is a complete list of the frame types:

* Claim Token
* Token
* Token Ack
* Who Follows
* Solicit Successor
* Set Successor1

* Set Successor2
* Set Previous
* New Member
* Del Member
* Member Request
* Member Count
* Who Last
* Who First
* Set Last
* Set First
* Init Done
* Data

Refer to Table C-1 for full description of the control frames.

C.3 TIMERS

There are a number of logical timers that are used in implementing the Token Passing Bus Access Method (TPBAM). These are:

* Token Hold Timer (TH)
* Token Lost Timer (TL)
* Response Timer (RT)
* Token Ack Timer (TA)
* Who Follows Timer (WF)
* Solicit Successor Timer (SS)

a) TH: This is the time that a station is allowed to hold the token for. It can only transmit data and control frames during this period.

b) TL: This is also known as the token rotation timer. It is the time that a station has to wait before it receives the token again. If the token is not received by this time, the station assumes it has been lost and, hence, claims the token with a 'claim token' frame.

$$TL = (TH * CS) + SM \qquad SM = \text{Safety Margin}$$

c) RT: This time is used to determine the position of a station in response to a 'who follows' and 'solicit successor' frames. RT is directly proportional to the address of the station.

$$RT = TS * \text{constant}$$

d) TA: This is the time taken by a station waiting to receive a 'token ack' frame when the token has been transmitted to its successor.

e) WF: This is the time taken by a station to receive a 'set successor' frame from its successor after issuing a 'who follows' frame. It is set with the response time of the last station in the ring, i.e. RT(LS). In this case it is the response time of the station with address 15.

$$WF = RT(LS) + SM$$

f) SS: This is the time taken by a station to receive a 'set successor' frame from a station wishing to enter the ring after a 'solicit successor' frame has been issued. It is the response time of the next station, i.e. RT(NS).

$$SS = RT(NS) + SM$$

C.4 SOFTWARE DEVELOPMENT AND STRUCTURE

The communication software is designed in a modular, structured manner, being implemented using the Jackson Program Design Facility (PDF) package. The Jackson chart is constructed to describe the software to a specific level of detail. The lowest levels represent simple functions that can be translated into program format. Generally the recommended control structures of structured programming have been used in the writing of the program source code.

The explanation that follows refers to a station's software and not to the ring as a whole.

The system consists of the three top level functions [Chart C.1]

* Initialise the board (hardware and software).
* Enter the ring.
* Run in operational mode.

C.4.1 Initialise the Board [Chart C.2]

In this mode each station reads its own address and set TS immediately when the power is applied, the appropriate variables being initialised. These include setting the member count to one and the TokenHeldBit to FALSE. The response time of the station is calculated from the station's address.

In order for the stations to be synchronised, each station sets its WAIT line to FALSE. When this is done, the START line goes high. This is shown on Chart C.2. Once this process is over, the station enters the ring.

C.4.2 Enter the Ring [Chart C.3]

When the START line goes high, each station starts its response timer (RT). It, then, monitors the system bus for any messages within its time-out period. There are three possible routes that a station can follow:

* If the timer times-out and there is no bus activity, then the station has to follow the routine for entering the ring as the 'first station'.
* If the station receives a 'Claim Token' frame, then the station has to follow the routine for entering the ring as 'not the first station'.
* If a frame other than 'Claim Token' is received then the station has to follow the routine for entering the ring as a 'plugged-in station'.

C.4.2.1 The First Station [Chart C.5]

When the response timer (RT) times-out, the station sends a 'Claim Token' frame. This informs other stations that the token has been claimed by the first station and to wait for further messages. This frame contains the address of the first station in the ring i.e this station's own address.

In order to establish the successor, a 'who follows' routine is executed, as described in section C.1. Once the NS address has been established, the station must reset its response time to the maximum possible value. Then it has to wait while other stations in the ring patch links together. The 'who before' does this waiting and at the same time it looks for a 'Who Follows' frame from its predecessor. It, then, waits for the token and acknowledges reception with a 'Token

Ack' frame. At this point the ring has been totally patched. Before the station can enter the operational mode it must inform other stations in the ring of the address of the last station in the ring. The station then sets the timer values. It then broadcasts the 'Init Done' frame starting to enter the operational mode. Table C-2 shows when the addresses are set by the first station.

C.4.2.2 Not the First Station [Chart C.6]

When the station receives a 'Claim Token' frame, it sets the first address (FS) and wait to patch the link with its predecessor. The 'who before' routine shows the activities it performs while it is waiting for such an event. Once the link with previous station has been formed, it links with the next station using the 'who follows' routine.

The station waits until the complete ring has been formed. Meanwhile, every time it monitors a 'Set Successor' it counts the number of stations in the ring. It, then, waits to receive the 'Set Last' frame followed by an 'Init Done' frame from the first station (FS). It can then set its timer values and enter the operational mode. Table C-3 shows the addresses being set for the station that is not the first in the ring.

C.4.2.3 A Plugged-In Station [Chart C.7]

When a station has just been plugged into an already established ring, it starts monitoring system bus messages. It is invited to the network when it receives a 'Solicit Successor' frame from a particular station. First, it must check whether its address lies between the inviting station's address and the next station's address (NS). If this is not true, it must then wait again for another invitation. If

this is true, however, it has to check whether it is the first in line, i.e there may be other stations wishing to enter as well. It achieves this by starting its response timer and looking for a response in the same way as if all the stations had just been powered up, starting to enter the ring.

The station with next lowest address gains entry first and all others must wait again for other invitations. Once the new station enters the ring and holds the token it sends a 'New Member' frame to the other stations, incrementing their record of the number of stations in the ring. When the token is passed on and the timer values have been set the station will enter the operational mode. Table C-4 defines the setting of addresses by a plugged-in station.

C.4.3 Running in Operational Mode [Chart C.4]

Once a station has entered operational mode it will periodically receive the token. The period is defined by the Token Rotation Time. If the token is not received by this time, it is assumed to be lost and the station will claim the token by issuing a 'Claim Token' frame. During normal operations the station also has to respond to any other messages that may arrive as a result of a station exiting or joining the ring. Table C-5 shows the response to a particular frame.

C.4.4 Repeated Routines

These are various routines that are called repeatedly throughout the main module (i.e program module). A list of them is shown below:

* Who Follows Routine.
* Who Before Routine.
* Wait For Token Routine.
* Token Ack Routine.
* Who Follows Response.
* Solicit Successor Response.
* Access Routine.
* Poll Bus and Timer Routine.

a) Who Follows Routine [Chart C.9]: This routine is used in the initialisation process. When a station is holding the 'token' it sends a control frame in order to identify its successor. It sends a 'Who Follows' frame and waits for a response within the who follows time. If there is no response within this time then something is wrong and the ring has to be reconfigured. A response, within the time period, will be the 'Set Successor' frame from the next station in sequence. This frame includes the address of the next station (NS). Now when a successor has been established, the token is passed on to it. Once the 'Token Ack' frame has been received the link has been made.

b) Who Before Routine [Chart C.10]: This routine is used in the initialisation process. It is used to identify the predecessor of a station. When this station receives a 'Who Follows' frame, it starts its response timer. If this expires before any of the other station's timers then it replies with a 'Set Successor' frame and the previous address (PS) address can be set. If, however, the timer does not expire and a 'Set Successor' frame is received, the station will increment its record of the number of stations in the ring and wait again for a 'Who Follows' frame. This is repeated until its timer expires before any of the other and is therefore next in the ring.

c) Wait For Token Routine [Chart C.14]: This routine simply loops, polling the bus until the station receives the token frame. Then it sets the 'TokenHeldBit' and exits the routine.

d) Token Ack Routine [Chart C.15]: This routine is used both in the initialisation and operational modes. It is used by a station after passing on the token frame onto its successor, waiting for an acknowledgement. The station waits for a 'Token Ack Time'. If the acknowledgement has not arrived within this time, it assumes that NS has failed and hence claims the token again. Otherwise it proceeds.

e) Who Follows Response [Chart C.12]: This routine is used by a station, in the operational mode, when it wishes to exit the token ring. This station has to check to see if the drop-out station is its predecessor. If so, this station (TS) sends a 'Set Successor2' to the predecessor of the failed station informing it of its new successor. If the failed station is not the previous station, however, then there is no action to be taken.

f) Solicit Successor Response [Chart C.13]: This routine is used by a station, in the operational mode, to check whether a new station has joined or entered the ring. If a 'Set Successor' frame is received then there is a new station in the ring and the TS will account for it by incrementing the record of the number of stations in the ring. If nothing happens within the maximum wait time then TS goes back into the operational mode.

g) Access Routine [Chart C.11]: This routine is executed by a station, once every 'N' token rotation cycles ('N' being set or defined by the programmer). It is used to invite any waiting station to enter the ring. This is done after holding the token N times. A 'Solicit Successor' frame is issued. If there is no reply then there are no stations wishing to enter. If a station does wish to enter, however, TS receives the 'Set Successor' frame, resets its NS address, clears the TokenHeldCount, transmits any data and then finally pass on the token.

h) Poll Bus and Timer Routine [Chart C.16]: This routine simply keeps polling the bus and the station timer simultaneously. An action is taken when either a message is received on the bus, or the timer expires.

TABLE C-1: DESCRIPTION OF CONTROL FRAMES

What follows is a description of the control frames used in implementing the token passing bus access method (TPBAM) during;

a) Configuration process

b) Operational mode.

FRAME	WHEN TRANSMITTED	ACTION ON RECEPTION
1. Claim Token	a) In the ring configuration by first station (FS). b) If the token is lost by TS.	a) Stop response timer and wait for 'Who Follows'. b) Reset token rotation timer.
2. Token	a) When the NS address has been set up. b) When TS has finished transmitting data.	a) Acknowledge reception with token Ack frame. b) Same as in (a).
3. Token Ack	a) On reception of token frame. b) Same as in (a).	a) Assume NS has received the token. b) Same as in (a).
4. Who Follows	a) When TS holds token and wishes to find NS. b) If NS does not send token ack frame, i.e. it has failed.	a) Start response timer and if it times out before the others, TS sends a set successor. b) If failed station is PS, TS sends a set successor. Otherwise TS behaves as in (a).
5. Solicit Successor	a) Not used.	a) If TS lies within the invited region, start response timer and if first to timeout send set successor. Otherwise wait for another solicit successor.

FRAME	WHEN TRANSMITTED	ACTION ON RECEPTION
	b) After holding token a preset number of times. To invite a station into the ring.	b) Look for messages on bus. If a new station joins the ring increment CS.
6. Set Successor1	a) If response timer is first to timeout either after a who follows or after a solicit successor.	a) Set NS address to the address of the sender.
	b) As in (a) but only after a who follows.	b) As in (a).
7. Set Successor2	a) Not used.	a) Not used.
	b) When TS wishes to exit the ring, send this to PS.	b) Set NS address to NS of exiting station.
8. Set Previous	a) Not used.	a) Not used.
	b) When TS wishes to exit ring, it sends this to NS.	b) Set PS address to PS of exiting station.
9. New Member	a) If TS is new station in ring.	a) Not used.
	b) Not used.	b) TS increments CS.
10. Del Member	a) Not used.	a) Not used.
	b) If TS wishes to exit ring, it sends this to all stations.	b) TS decrements CS.
11. Member Request	a) If TS is a new station, it sends this to NS to find CS.	a) Not used.
	b) Not used.	b) TS responds with a member count frame.
12. Member Count	a) When TS is FS and the ring is constructed.	a) If TS is new it sets its CS address.
	b) If TS receives the member request frame.	b) Not used.

FRAME	WHEN TRANSMITTED	ACTION ON RECEPTION
13. Who Last	a) If TS is new, it sends this to NS to get the LS address.	a) Not used.
	b) Not used.	b) Reply with the set last frame.
14. Set Last	a) If TS is new and greater than PS it sends this to all tations informing them to reset their LS address. If TS is the first station then before the system goes into operational mode, this frame is sent.	a) When TS receives this it sets the LS as its LS address.
	b) If the TS is the predecessor of the LS and LS fails to respond.	b) Same as (a).
15. Who First	a) If TS is new to the ring, it will send this to NS.	a) Not used.
	b) Not used.	b) When TS receives this it responds with the set first frame.
16. Set First	a) If TS is new to the ring and is the new FS, it sends this to all the stations.	a) When TS receives this it sets its FS value.
	b) If TS is successor to FS and FS fails, TS will send this.	b) Same as (a).
17. Init Done	a) If TS is the first station in the ring, once the ring has been configured and it has received the token it will inform the other stations.	a) TS will enter the operational mode.
	b) Not used.	b) Not used.

FRAME	WHEN TRANSMITTED	ACTION ON RECEPTION
18. Data	a) Not used. b) Data is transmitted only in the operational mode when the station holds the token.	a) Not used. b) TS transfers data to Processing section.

TABLE C-2: ADDRESS SETTING BY THE FIRST STATION

ADDRESS	WHEN ADDRESS IS ESTABLISHED
Own - TS	This is set when the station is powered-up.
First - FS	Since this is the first station in the ring FS is TS.
Next - NS	The next station will respond to a 'Who Follows' with a 'Set Successor' frame. The NS address will be included in the frame.
Previous - PS	When TS receives a 'Who Follows' frame and if its response timer times out before the other stations, it can set PS from the 'Who Follows' frame.
Last - LS	Since TS is the first station in the ring then LS is the same as PS.
Member Count - CS	Whenever TS receives a 'Set Successor' frame CS is incremented.

TABLE C-3: ADDRESS SETTING BY A NOT-FIRST STATION

ADDRESS	WHEN ADDRESS IS ESTABLISHED
Own - TS	This is set when the station is powered-up.
First - FS	When TS receives a 'Claim Token' frame FS is included within it.
Previous - PS	When TS receives a 'Who Follows' frame and if its response timer times out before the other stations, it can set PS from the 'Who follows' frame.
Next - NS	The next station will respond to a 'Who Follows' with a 'Set Successor'. The NS address will be included in the frame.
Last - LS	The first station will send the 'Set Last' frame specifying LS.
Member Count - CS	Whenever TS receives a 'Set Successor' frame CS is incremented.

TABLE C-4: ADDRESS SETTING BY A PLUGGED-IN STATION

ADDRESS	WHEN ADDRESS IS ESTABLISHED
Own - TS	This is set when the station is powered-up.
Previous - PS	When TS receives a 'Solicit Successor' frame and it is next in sequence in the ring, PS will be set by the frame.
Next - NS	The sender of a 'Solicit Successor' frame includes the address of its old NS, this becomes the NS of this station.
First - FS	If TS is less than PS then TS is the new FS and it will send a 'Set First' frame. If not it asks NS with a 'Who First' frame.
Last - LS	If TS is greater than NS then TS is the new LS and it will send a 'Set Last' frame. If not it asks NS with a 'Who Last' frame.
Member Count - CS	When TS has entered a ring it will ask NS for the station's member count by sending a 'Member Request' frame.

TABLE C-5: RESPONSE TO FRAMES RECEIVED IN OPERATIONAL MODE

FRAME RECEIVED	RESPONSE
CLaim Token	Clear TokenHeldBit and reset Token Rotation Timer.
Who Follows	Run Who Follows Response routine (described in section C.4.4).
Solicit Successor	Run Solicit Successor Response routine (described in section C.4.4).
Who First	Send the Set First frame.
Who Last	Send the Set Last frame.
Member Request	Send the Member Count frame.
Del Member	Decrement the station count record.
New Member	Increment the station count record.
Set Successor2	Reset next station address.
Set Successor1	Increment the station count record.
Set Last	Reset last station address.
Set First	Reset first station address.
Set Previous	Reset previous station address.
Token	Set WaitBit, send Token Ack and run Access routine [section C.4.4].

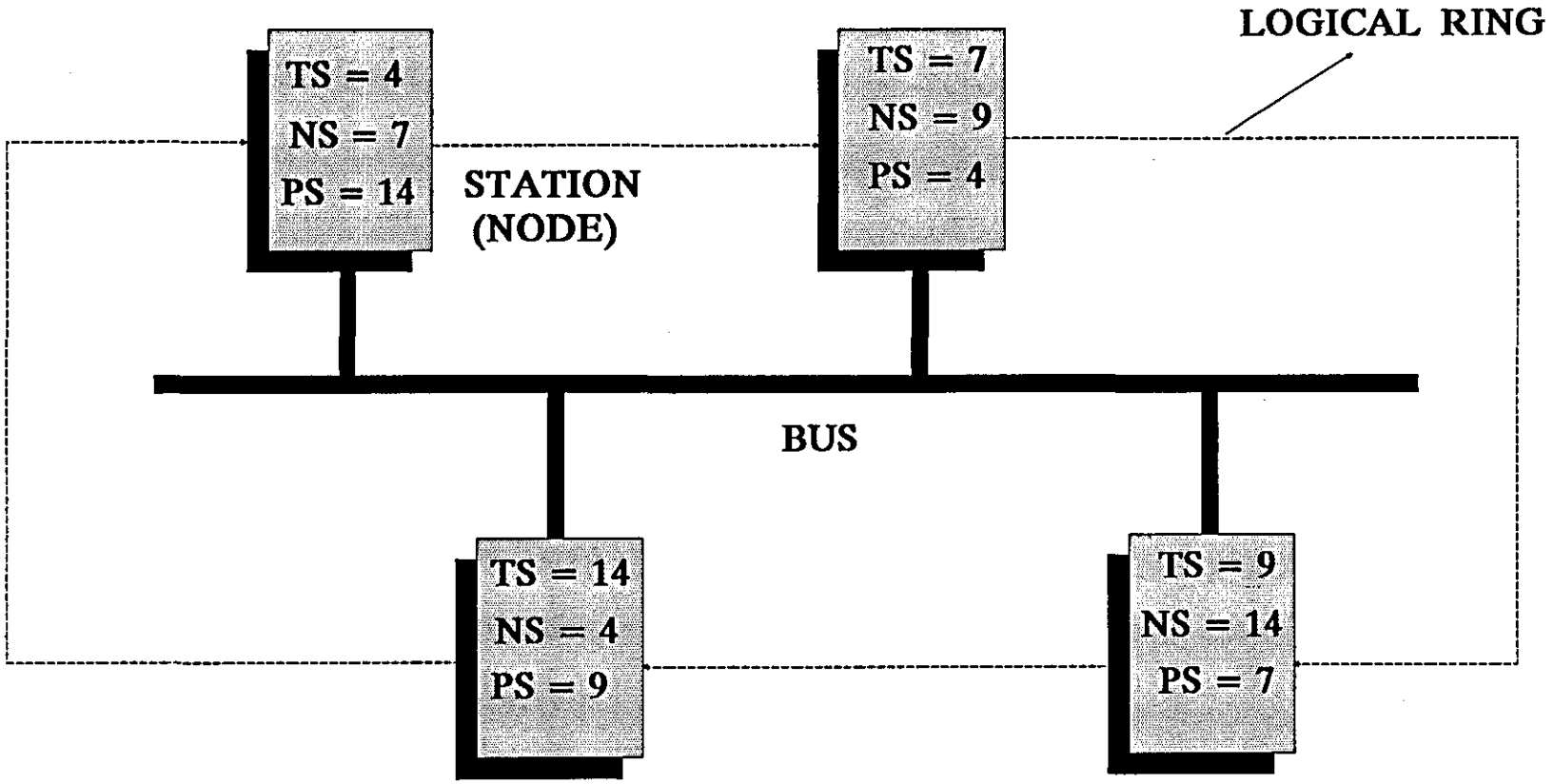


Fig. C.1 TOKEN PASSING ON A LOGICAL RING

Time	Station 4	Station 9	Station 13
Power Up T=0	Read Own Address	Read Own Address	Read Own Address
T = 1	Start Response Timer	Start Response Timer	Start Response Timer
T = 2	Time out	Still Running	Still Running
T = 3	Send Claim Token	Stop Timer	Stop Timer
T = 4	Send Who Follows	Start Response Timer	Start Response Timer
T = 5	Wait for Response	Time out	Still Running
T = 6	Set NS Address	Send Set Successor Set PS Address	Stop Timer
T = 7	Send Token to 9	Receive Token	Wait
T = 8	Receive Token Ack	Send Token Ack	Wait
T = 9	Start Response Timer	Send Who Follows	Start Response Timer
T = 10	Still Running	Wait for Response	Time out
T = 11	Stop Timer	Set NS Address	Send Set Successor Set PS Address
T = 12	Wait	Send Token to 13	Receive Token
T = 13	Wait	Receive Token Ack	Send Token Ack
T = 14	Start Response Timer	Wait	Send Who Follows
T = 15	Time out	Wait	Wait for Response
T = 16	Send Set Successor Set PS Address	Wait	Set NS Address
T = 17	Receive Token	Wait	Send Token to 4
T = 18	Send Token Ack	Wait	Receive Token Ack
T = 19	Send Init Done	Receive Init Done	Receive Init Done
T = 20	Go into Op Mode	Go into Op Mode	Go into Op Mode

Fig. C.2 RING CONFIGURATION PROCESS

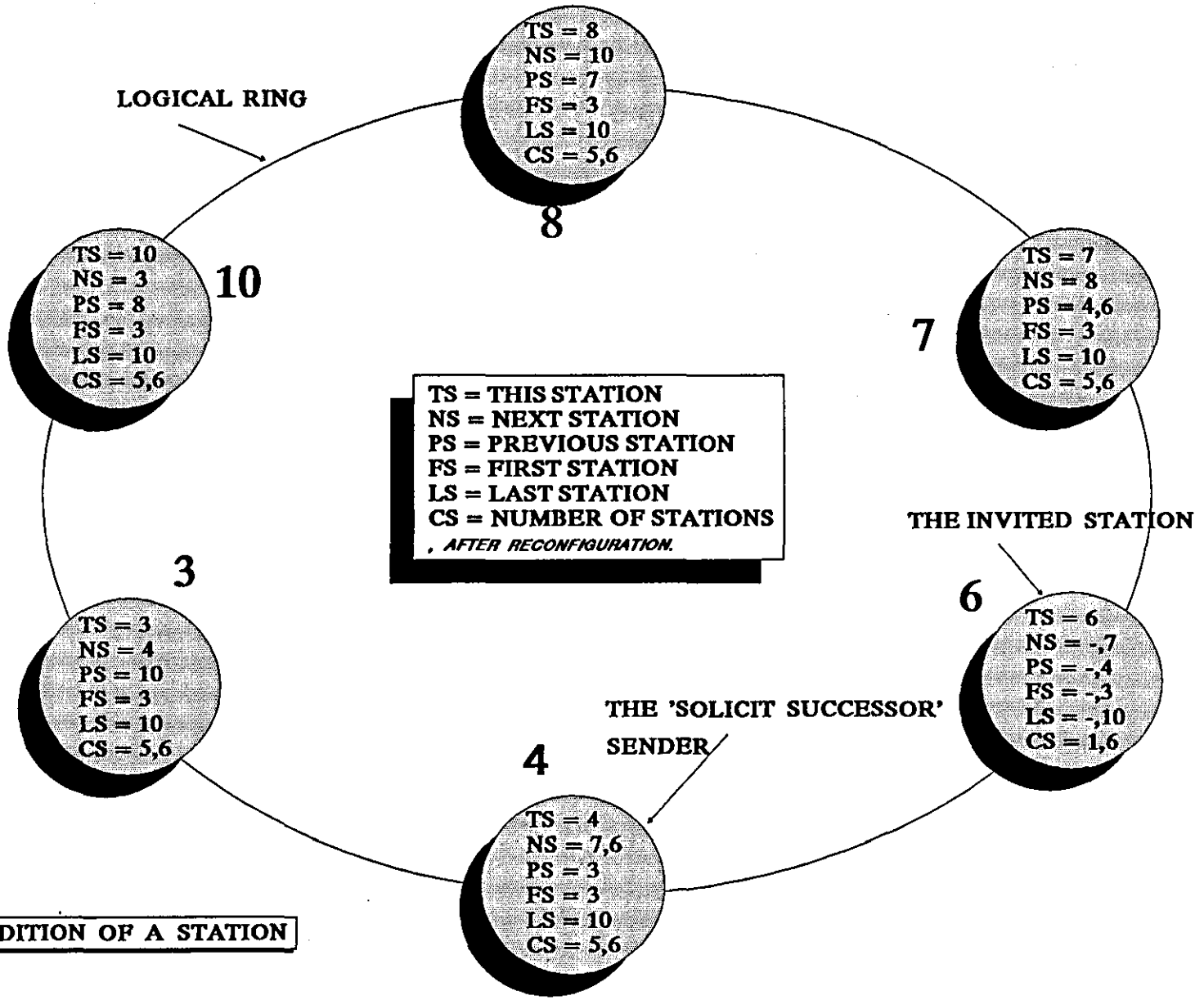


Fig. C.3 ADDITION OF A STATION

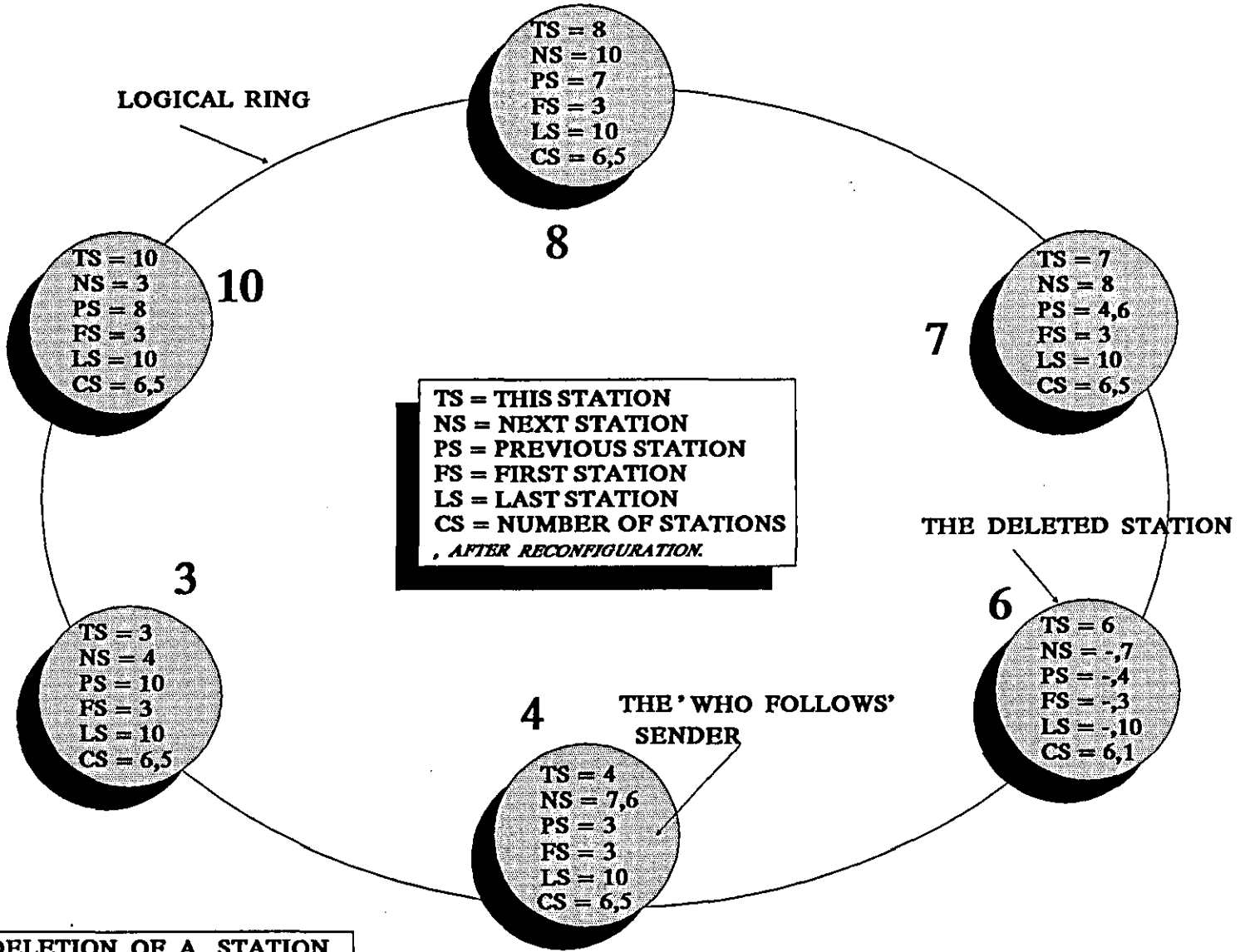


Fig. C.4 DELETION OF A STATION

CHARTS

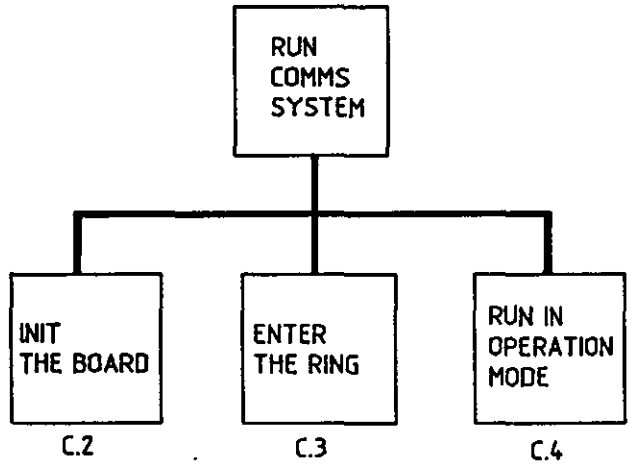


CHART C.1 RUN COMMS SYSTEM

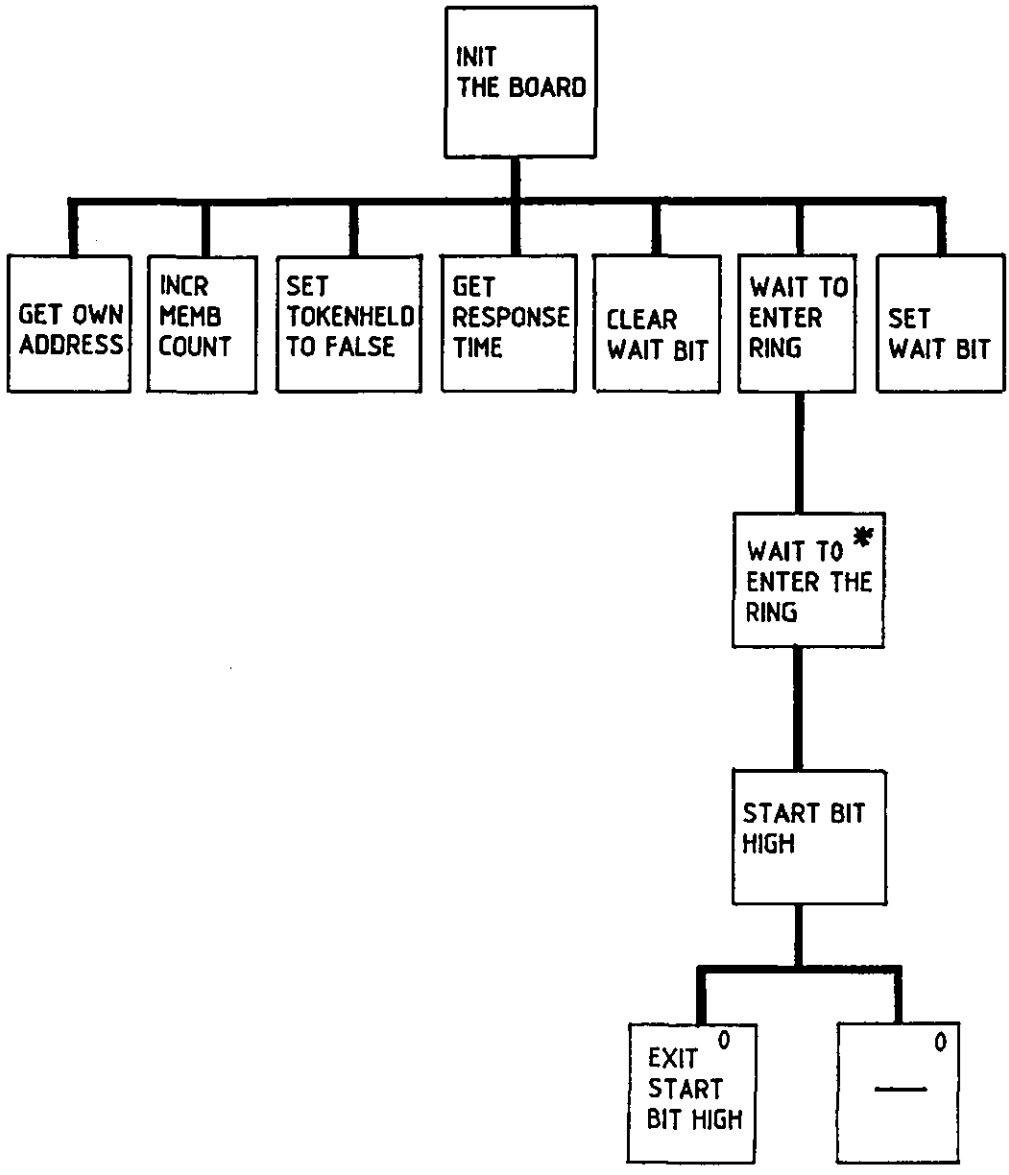


CHART C.2 INIT THE BOARD

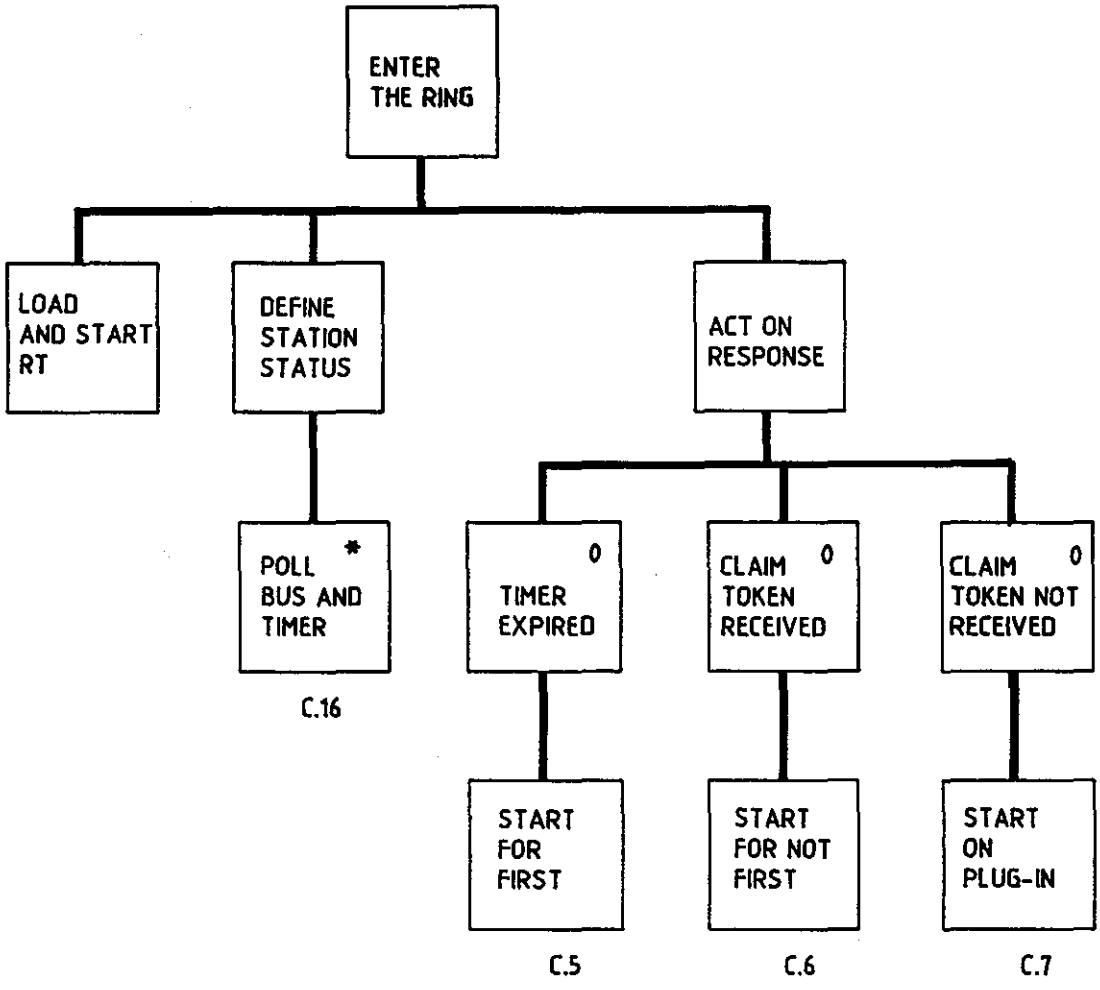


CHART C.3 ENTER THE RING

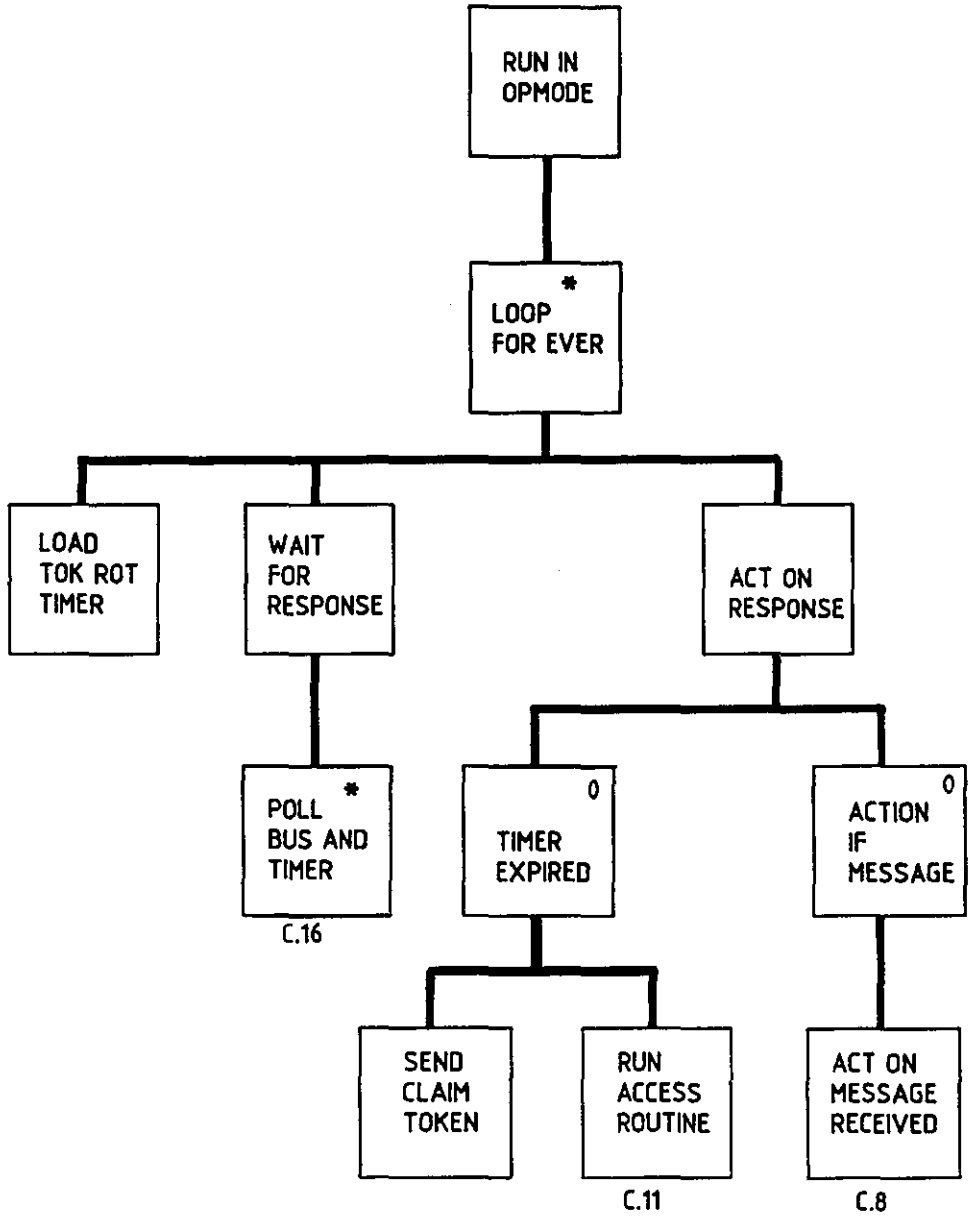
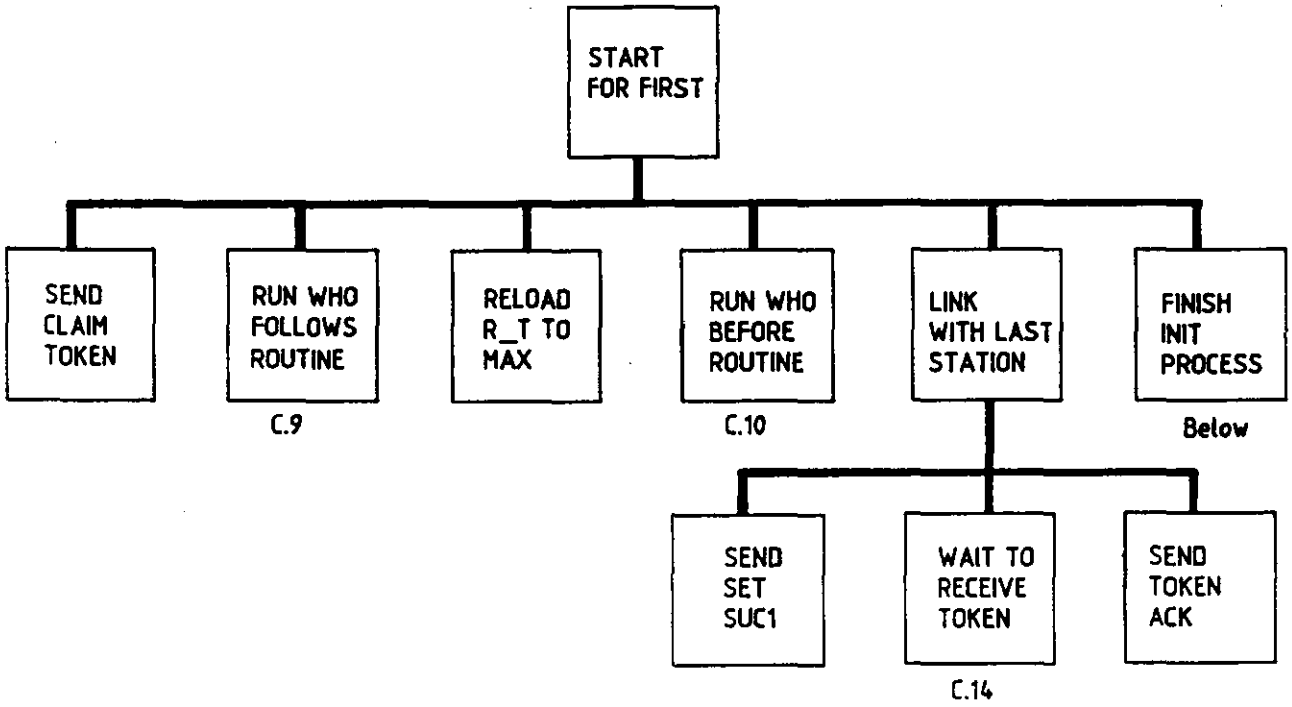


CHART C.4 RUN IN OPMODE



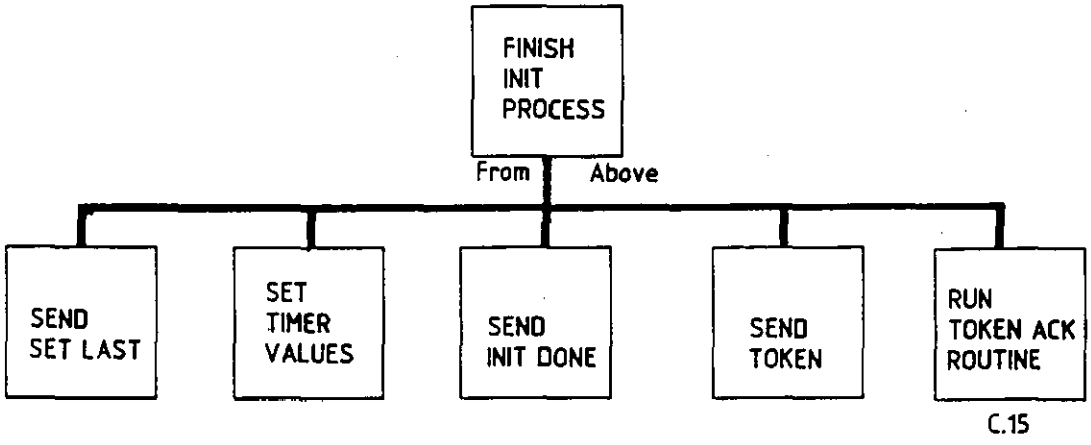


CHART C.5 START FOR FIRST

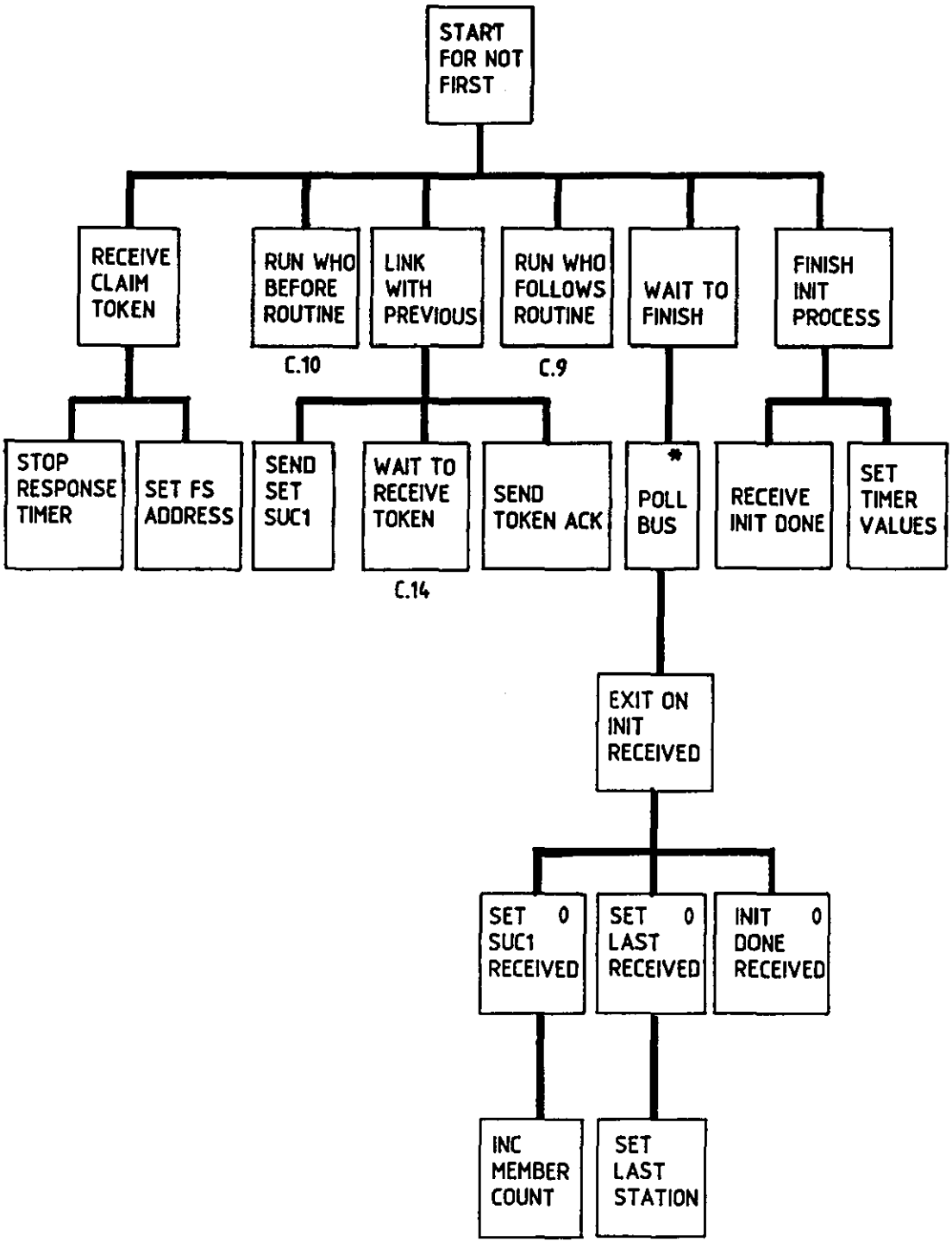


CHART C.6 START FOR NOT FIRST

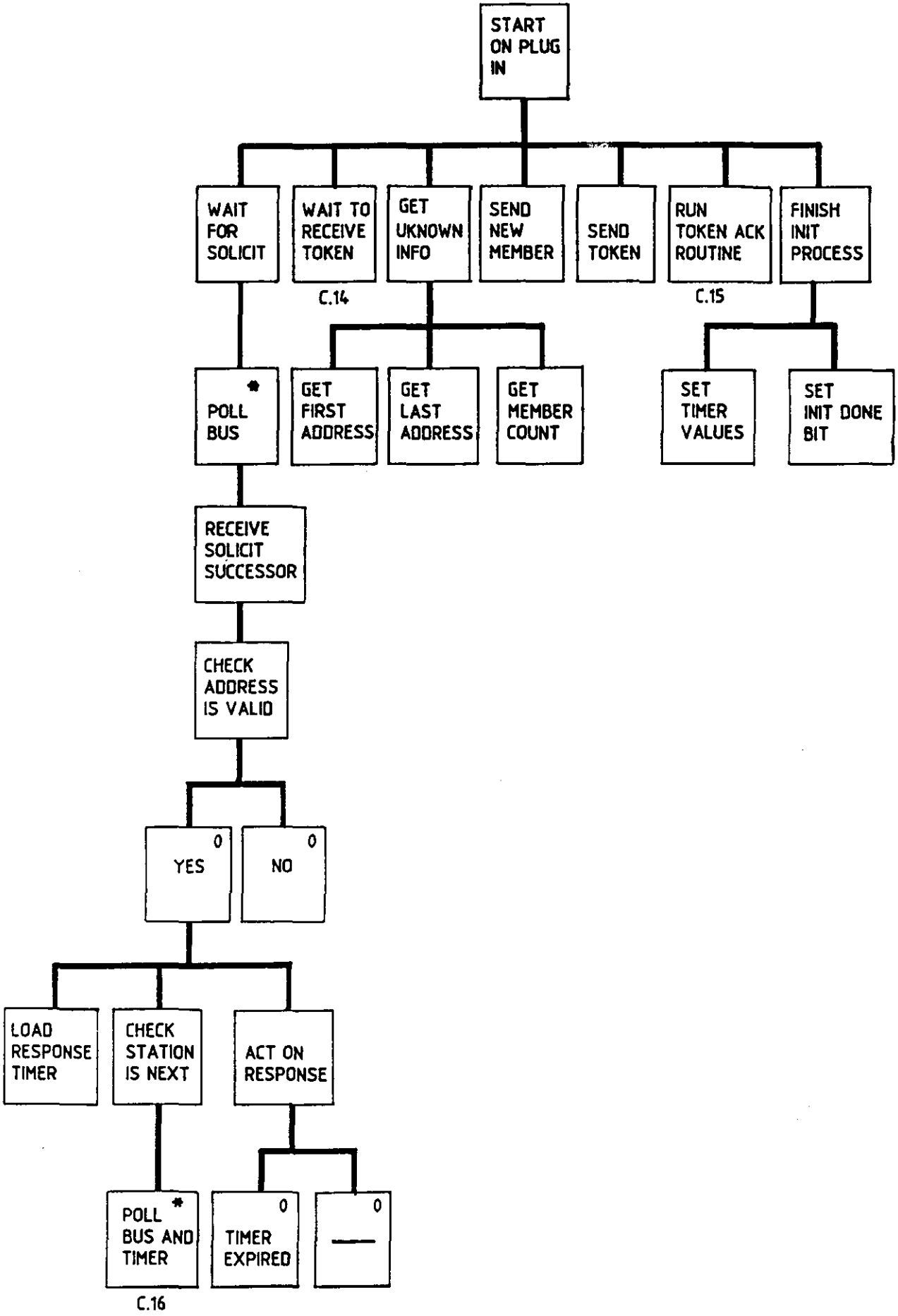
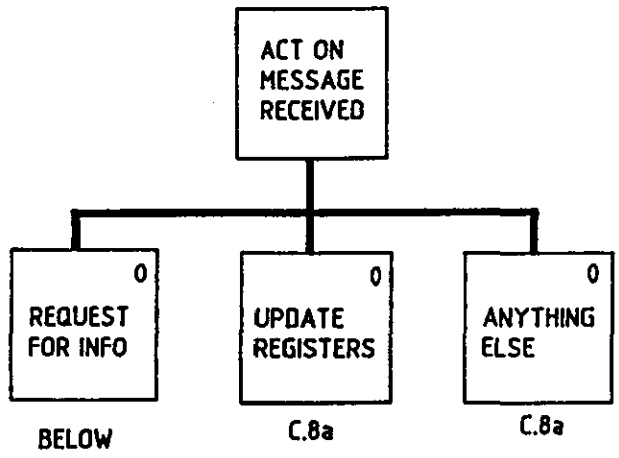


CHART C.7 START ON PLUG IN



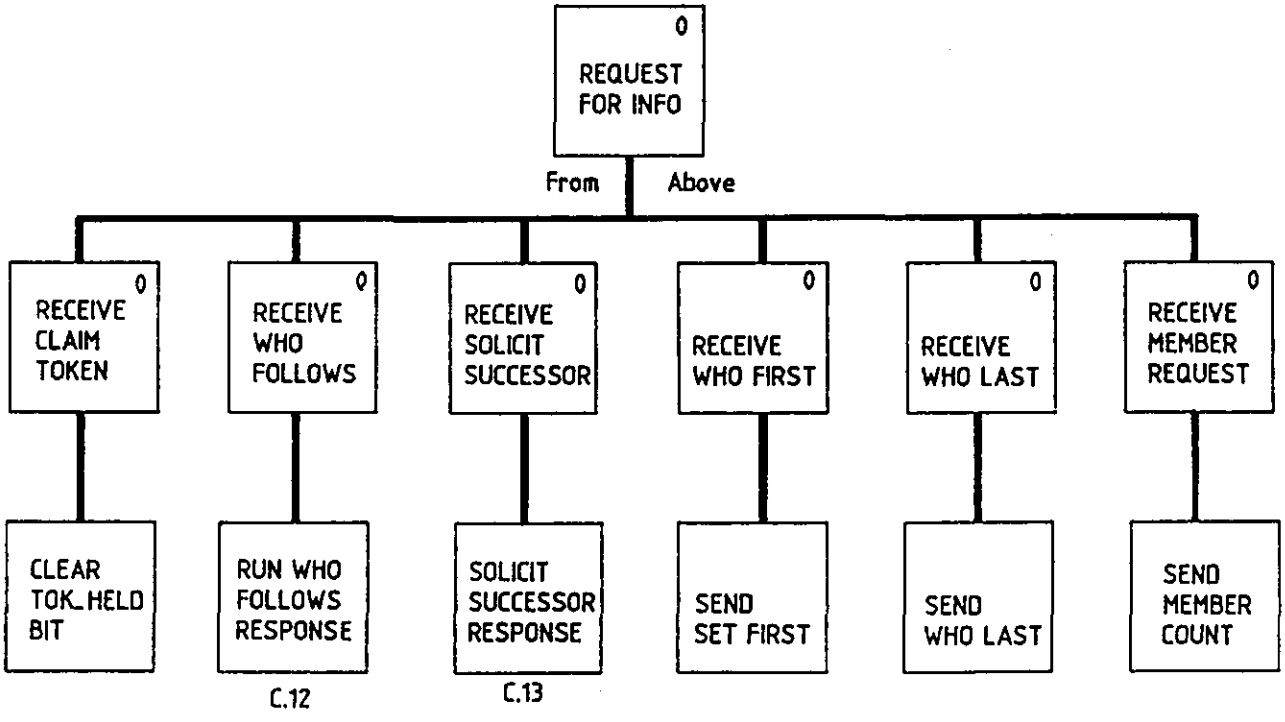
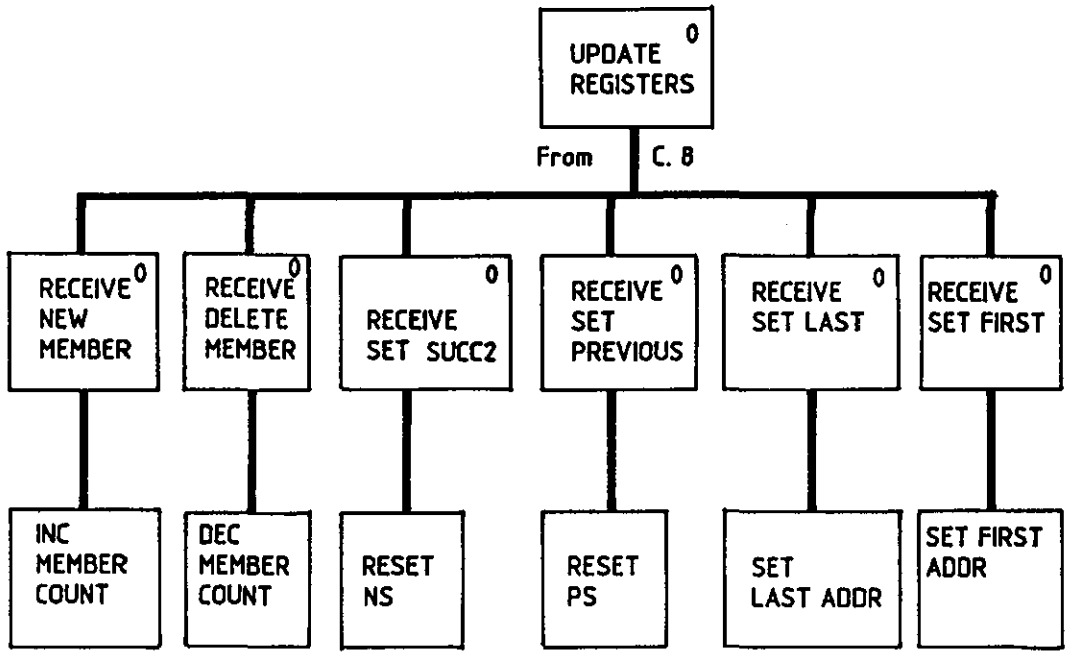


CHART C.8 ACT ON MESSAGE RECEIVED



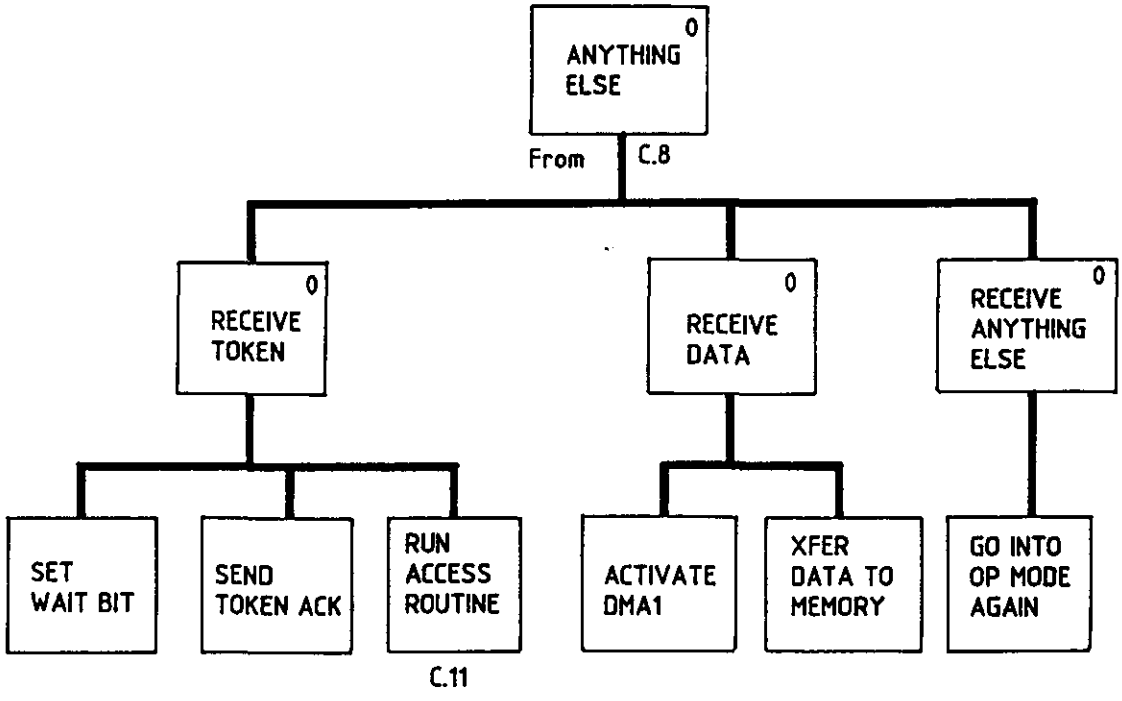


CHART C.8a ACT ON MESSAGE RECEIVED

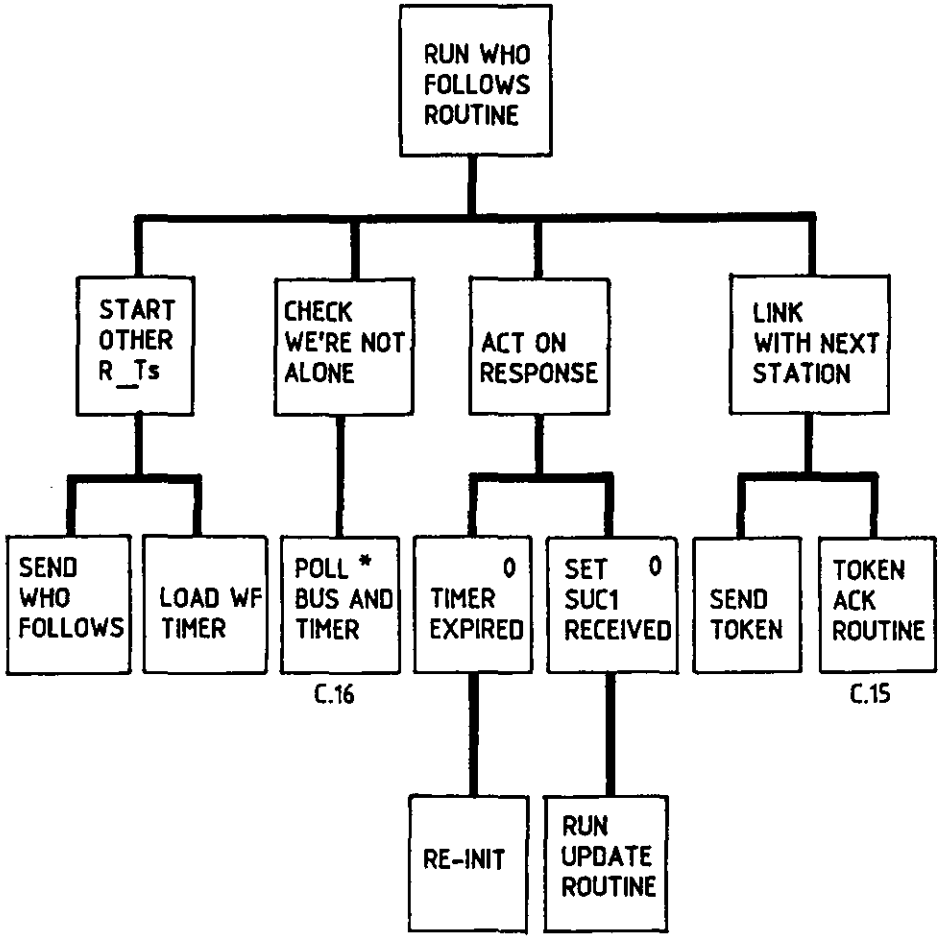


CHART C.9 RUN 'WHO FOLLOWS' ROUTINE

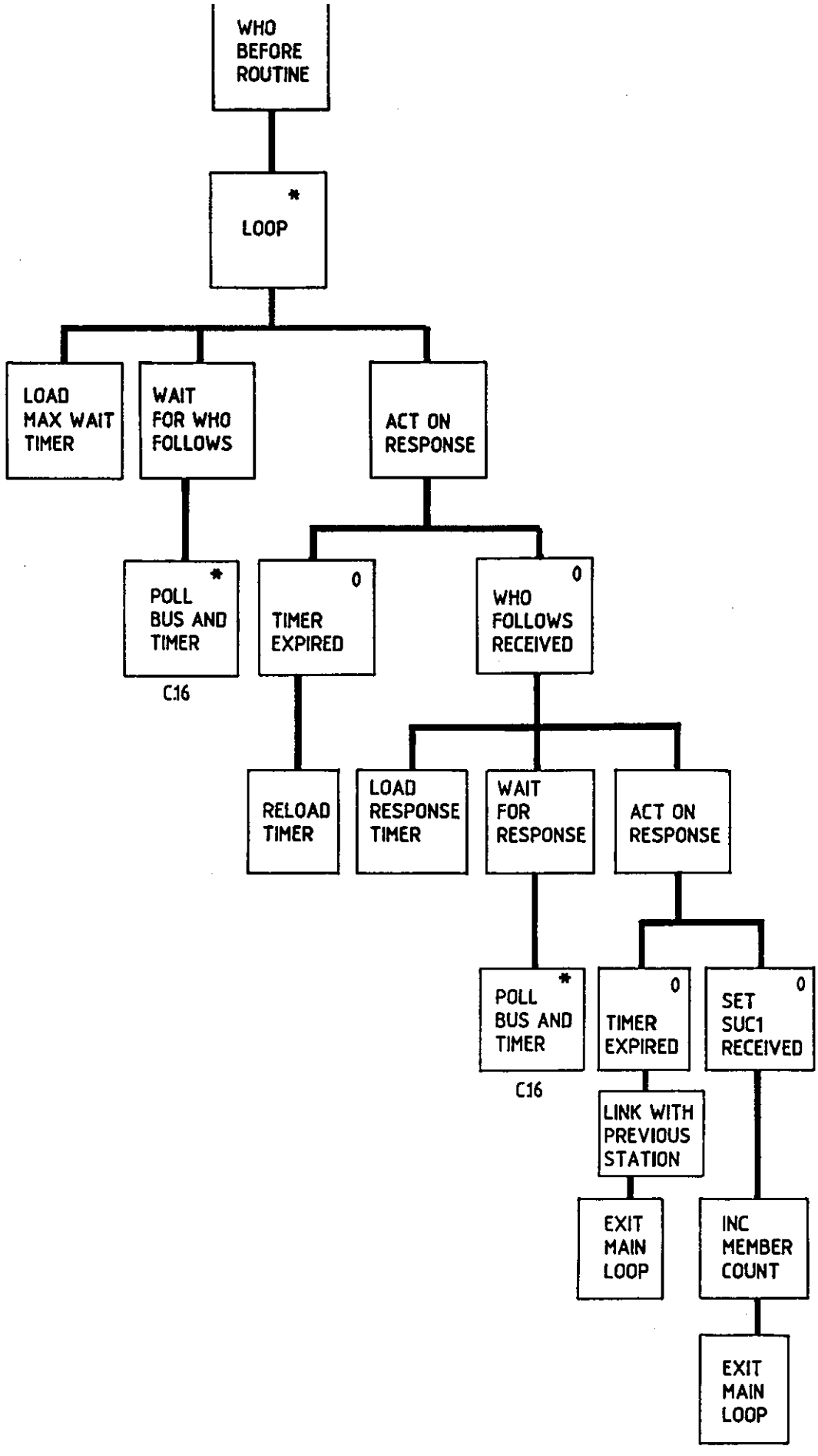


CHART C.10 'WHO BEFORE' ROUTINE

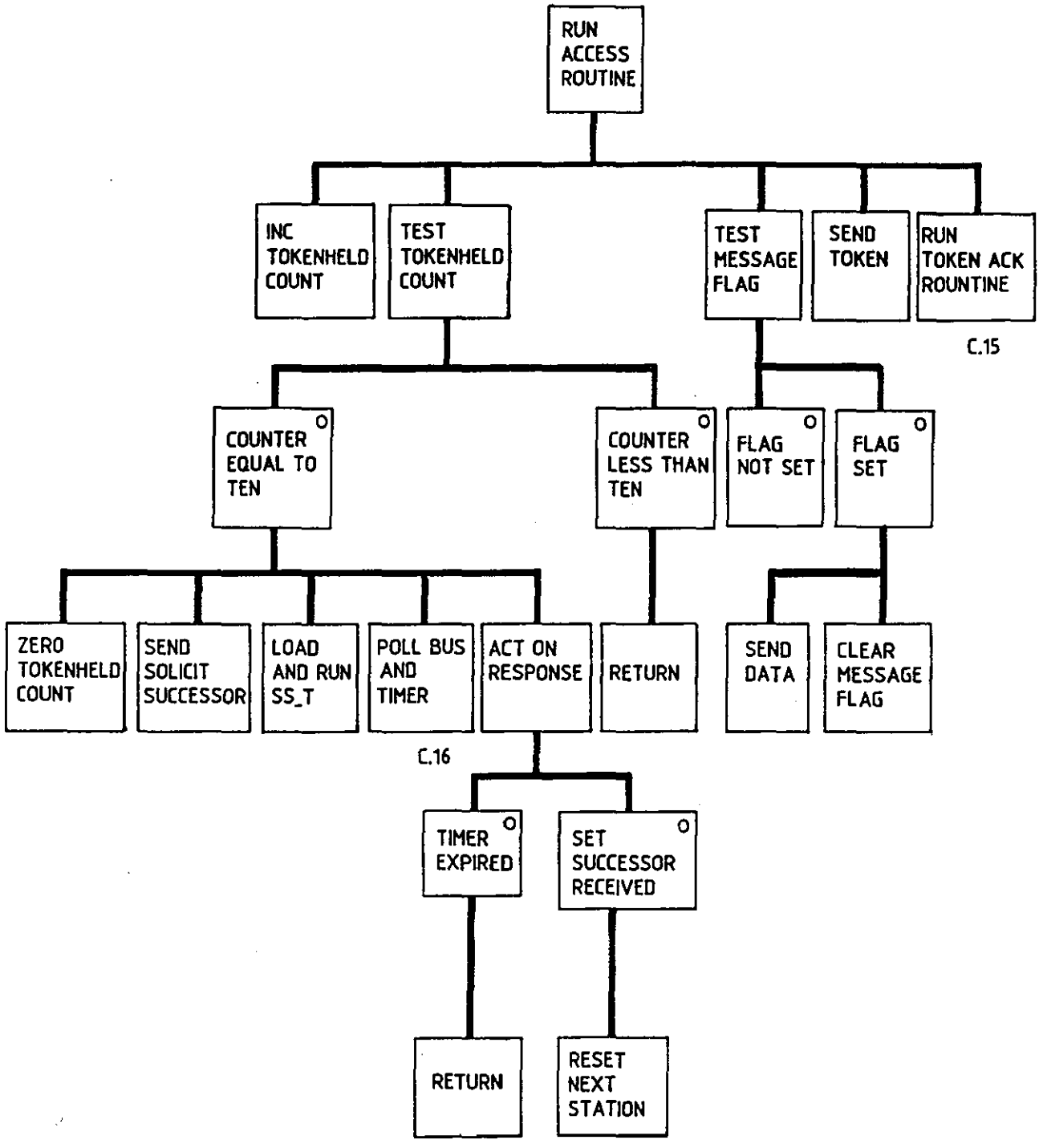


CHART C.11 `RUN ACCESS´ ROUTINE

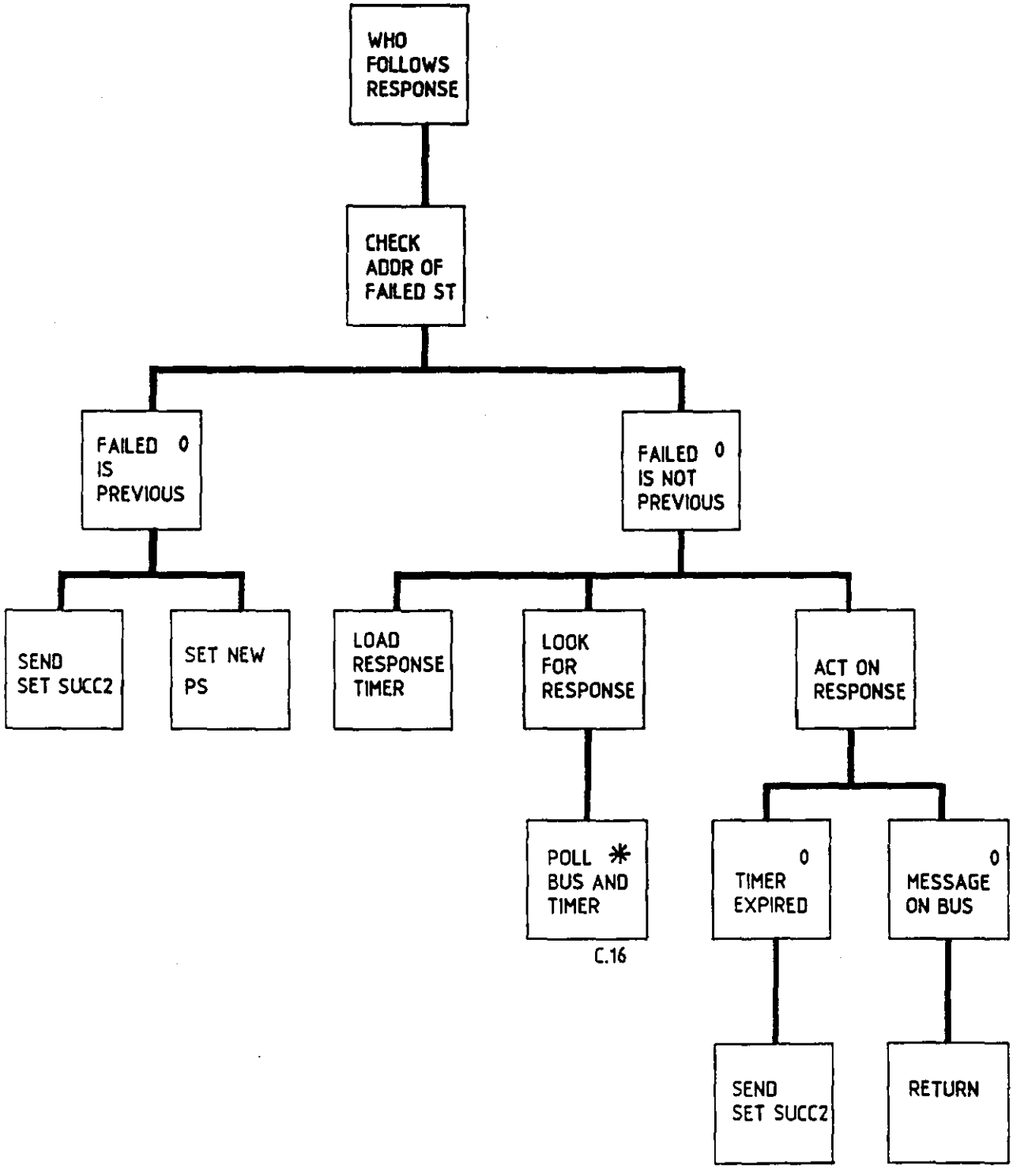


CHART C.12 'WHO FOLLOWS' RESPONSE

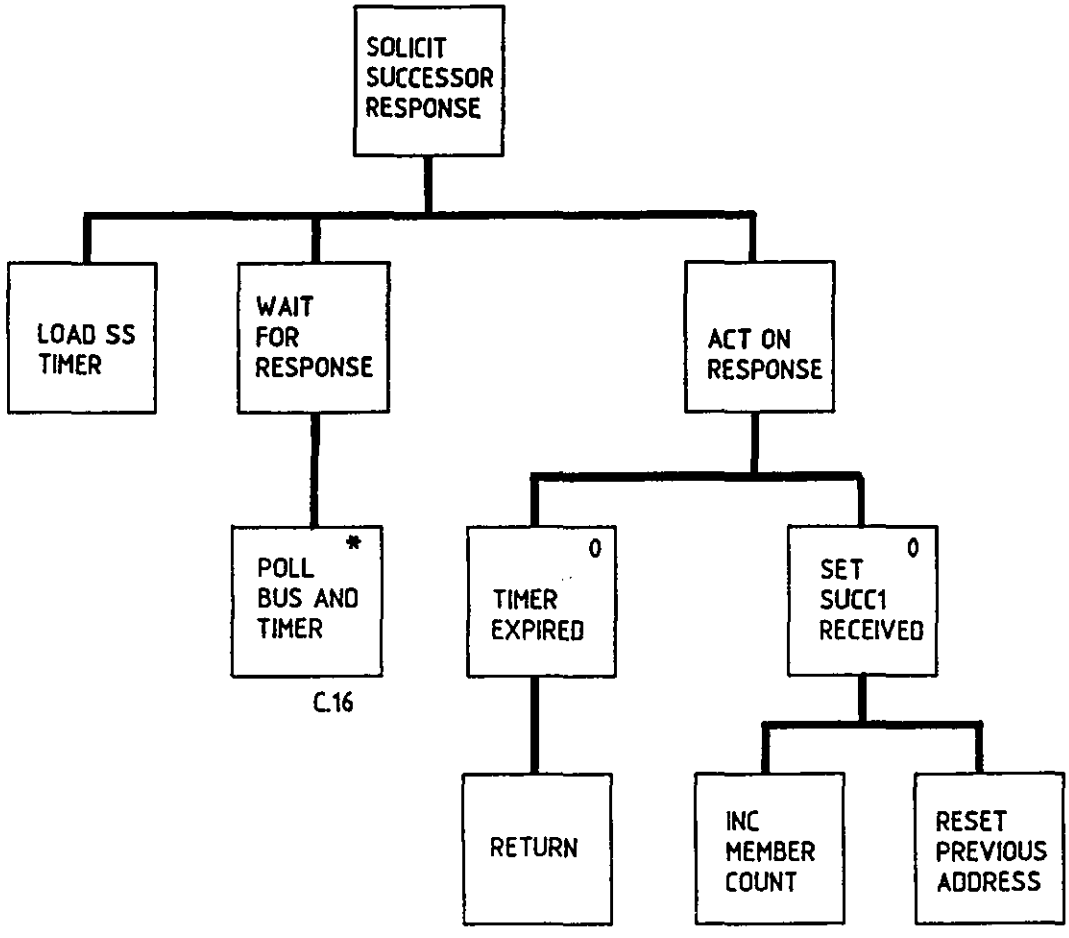


CHART C.13 'SOLICIT SUCCESSOR' RESPONSE

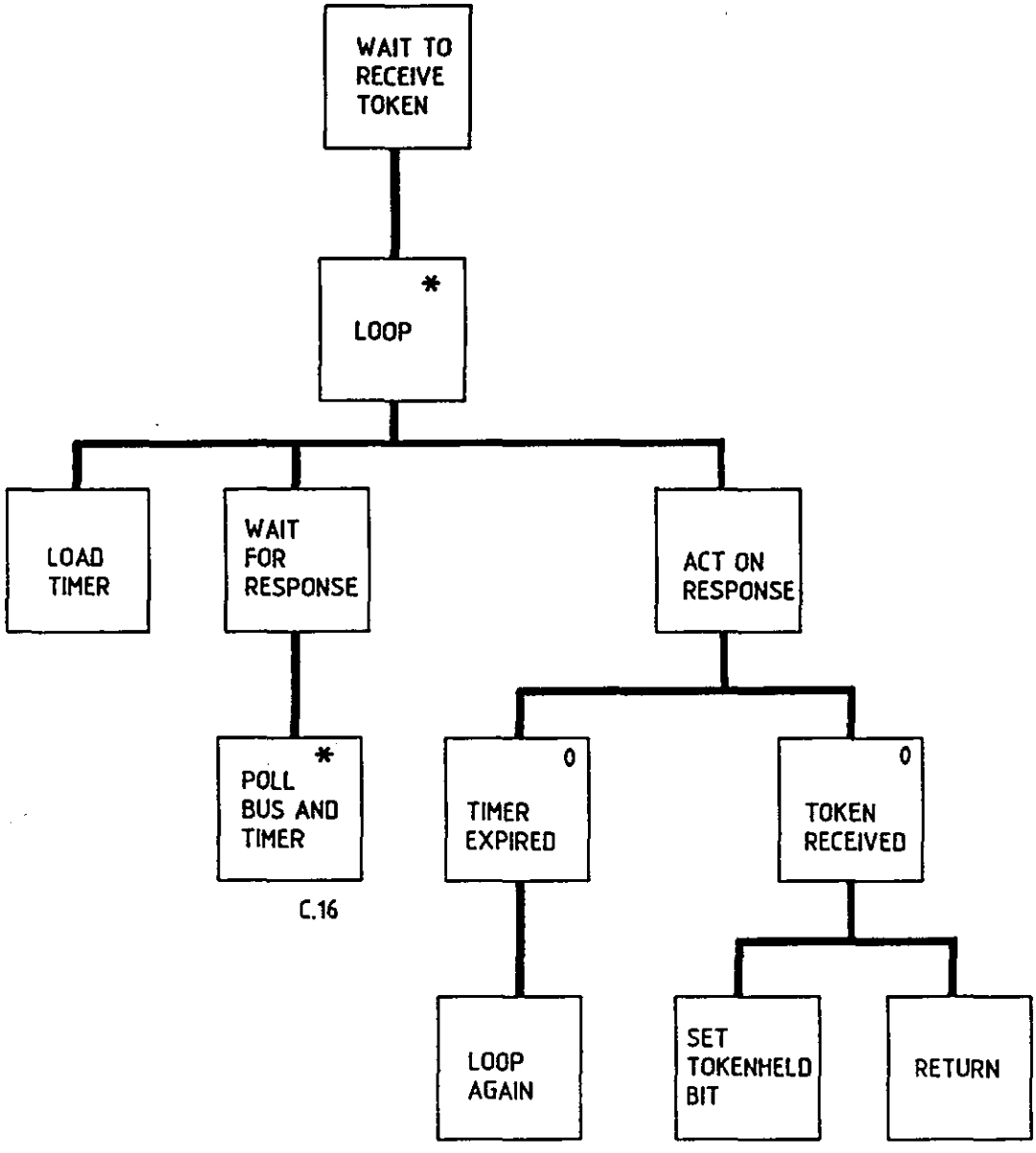


CHART C.14 WAIT TO RECEIVE TOKEN

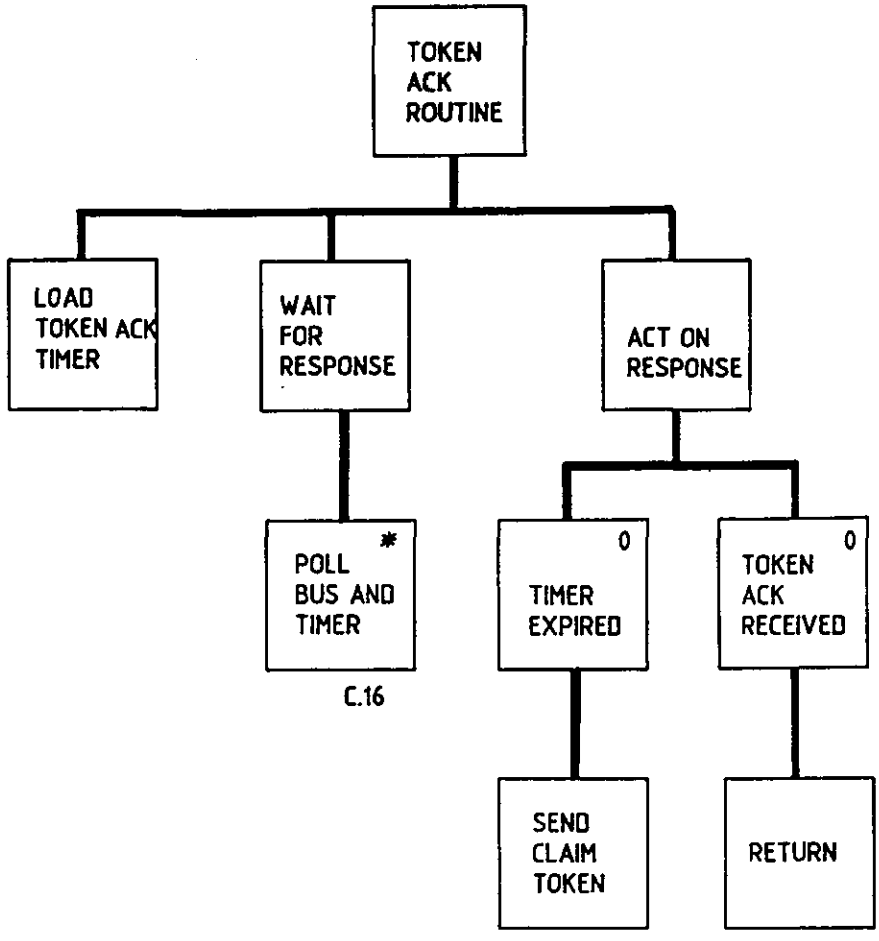


CHART C.15 'TOKEN ACK' ROUTINE

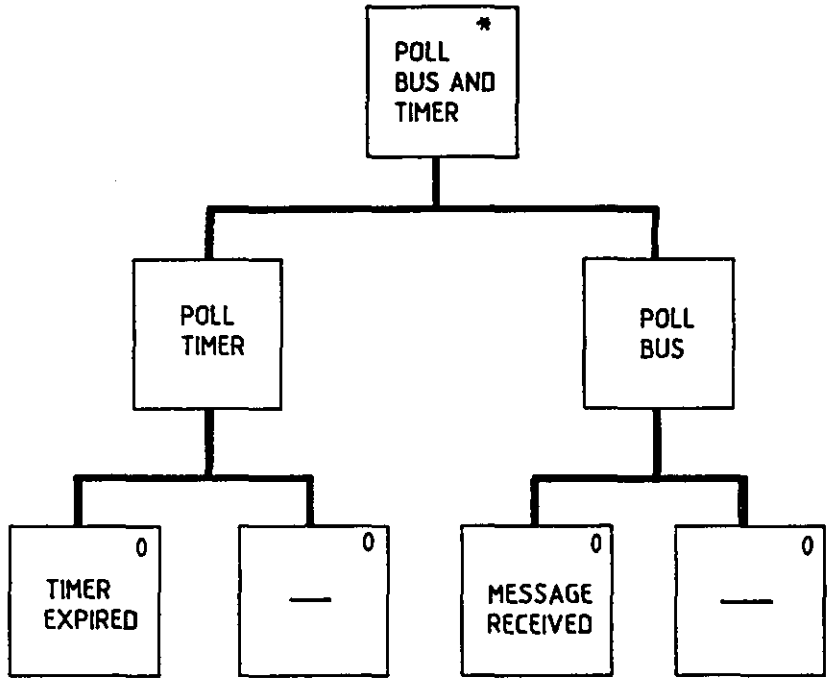


CHART C.16 POLL BUS AND TIMER

APPENDIX - D

APPENDIX D

CPM ENVIRONMENT EMULATOR - CPM100 MODULE

D.1 OVERVIEW

The CPM100 module is written to provide a basic initialisation process for the communication section. This module is designed for use in an embedded system running code generated by the FTL Modula-2 compiler. Code generated by the FTL compiler expects to run in a CPM environment. The appropriate CPM functions required by the code are being emulated by this module. The emulated CPM functions are listed below in Table D-1:

TABLE D-1: CPM FUNCTIONS EMULATED

FUNCTION NO	DESCRIPTION
0	Warm boot, resets system from 0000H
1	Console input, from serial interface
2	Console output, to the serial interface
6	Direct I/O, via the serial interface
9	Display message, to the serial interface
A	Line input, from serial interface
B	Console status, from serial interface

This module is designed for a 64180 processor with the code positioned in an EPROM at address 00000H and for a RAM positioned between 0A000H and 0BFFFH. The code generated by this routine must be positioned at address 00000H in the EPROM. This module also uses a small section at the top of the RAM. The code is written so that this address can be moved if desired. The lowest address used in RAM is 'TOP_OF_RAM' -

0FH. This address is actually used as the start of the stack . CPM requires the stack on an entry to an application program to be at least 8 levels deep so that the highest byte of RAM being used is 16 bytes below this location, i.e 'TOP_OF_RAM' - 01FH. Table D-2 gives the full description of the functions emulated.

TABLE D-2: CPM FUNCTIONS EMULATED

NO.	FUNCTION	DESCRIPTION
0	WBOOT	This has the same effect as a reset, jumping to location 00000H.
1	CONIN	Read a character from the console and then echo to the screen. Wait if no character is available.
2	CONOUT	Output a character to the screen.
6	DIRECTIO	If E is 0FEH return status; i.e. 000H if no character is ready and 0FFH if character is ready. If E is 0FFH, however, return character from console, do not wait if no character is available but return 000H. Do not echo character read to console. For any other value of E output that value to the console.
9	MESSAGE	Output a message pointed to by DE to the console. The message is terminated by 024H.
A	READBUF	Read a text buffer from the console. The buffer address is passed to DE. The first byte gives the maximum length of the buffer, while the second byte is set to the actual length of the buffer, i.e. characters actually read. The rest of the buffer contains the text read. The process of text reading into the buffer is terminated by a return, 0DH.
B	STATUS	Determines if a character is available from the console. The result being: 000H no character. 0FFH character ready.

Data required by these routines is passed via the E or DE registers, and returned subsequently to the accumulator. The only exception to this is the buffered input and message output functions where DE is used to point to the data. The required function is placed in the C register and a call is, then, made to 00005H.

If any call is made to a function which is not supported by the functions mentioned above then an error message is output to the console and control is returned to the application program.

These routines preserve all registers apart from the accumulator when called. The accumulator, however, is altered even if it is not used to pass data back from the function.

CPM100 also supports the use of a watchdog timer. An interrupt handler is incorporated into the program to handle an NMI interrupt from the watchdog timer. This has the same effect on the 64180 CPU as a jump to 0000H. The processor is subsequently re-initialised by the program before control is passed to the application program at address 0100H.

D.2 CPM COMPATIBILITY

What follows is a brief outline of how the environment set up by this set of routines, i.e. CPM module, varies from the standard CPM environment.

There is no support for BIOS calls. No vector table is produced and the value at location 00001H which CPM normally expects to point to the BIOS vector table points else where. Because of this situation,

any attempt to call a BIOS routine will crash the application program and the system may well lock up.

The memory space (0005CH - 000FFH) is usually used as buffers by CPM. Code produced by these routines is actually placed in this space. The application program will fail to write to this area as it is in EPROM address space. Some programs may try to read the default buffers on entry to determine if any command line parameters are being passed to them. If this occurs, results are unpredictable and could well crash the system.

CPM specifies that on entry to an application program, the stack pointer must be left pointing to an 8 level stack with the return address left on this stack. Implemented routines abide by this rule. The returned address, placed on the stack, is actually 00000H to cause the same action as a reset if an application program exits.

It is believed that FTL Modula-2 only calls CPM functions supported by these routines and makes no calls to the BIOS, unless directed to do so by the user. These routines are also believed to supply enough support to allow use of the Terminal and SmallIO modules as supplied with the compiler.

D.3 LIMITATIONS

CPM100 provides a faithful emulation of the functions mentioned above. It will also produce an error message if an illegal function is called. The more severe problem is if an application program makes use of some other parts of CPM. If this occurs the results will be unpredictable. Some of the more likely problems are detailed below:

The page zero area is normally used to hold certain buffers. Application programs should not directly address these buffers. The command line parameters and the default FCB's are held in this area. They are examined by some programs to see if any parameters have been set up on the command line. Any program which tries to examine this area is, in fact, accessing the code of CPM100 itself. Consequently, application programs must not be allowed to examine this area.

In addition to the standard entry point of CPM at location 0005H, there is a second batch of entry points. These give access to the BIOS routine, as opposed to the BDOS at location 0005H. The BIOS routines give a program access to a lower level of device driver. The BIOS actually has multiple entry points, one for each function. These entry points are normally near the top of memory, in a proper CPM system. No attempt is made to emulate this function, hence any program attempting to access the BIOS will crash. It should also be noted that the usual pointer to the address of these routines, normally at location 0001H, is not set up.

APPENDIX - E

APPENDIX E

MULTI-PROCESSOR SYSTEM - KERNEL SOFTWARE STRUCTURE

E.1 INTRODUCTION

The real-time kernel software is designed in a modular, structured manner, being implemented using the Jackson Program Design Facility (PDF) package. In the following discussion, a full description of the real-time kernel primitive operations is laid down. The Jackson chart is constructed to describe the software to a specific level of detail. The lowest levels represent simple functions that can be translated into program format. Generally the recommended control structures of structured programming have been used in the writing of the program source code.

The kernel module, MAIN-DISTKERNEL, differs from the communication's main program module, RUNCOMMS, in that it does not remain in control once called. Instead, however, it provides entry points (kernel primitives calls) where it may be called by the application software. Once called it performs the necessary communication or management routines before returning to the application program, in the shortest time possible for the required action.

E.2 REAL-TIME KERNEL STRUCTURE

The kernel structure consists mainly of three top level functions [Chart E.1]:

* Power-Up.
* Initialise System.
* Run Application Software.

E.3 POWER-UP [Chart E.1]

On power-up the system points to a specific location for code execution (FFFF0H). Initially the memory is mapped only for the top 1 Kbyte of memory (FFC00H). This area is not sufficient to accommodate the initialisation routines and the application program. Hence, the system has to be transferred to a larger working area. This is done by a jump instruction to the top 1 Kbyte of memory area, where the execution of the next stage of system initialisation starts on.

E.4 INITIALISE SYSTEM [Chart E.2]

Initialising the system consists of running the bootstrap loader. This consists of two sections, an assembler part and a Modula-2 part. The reason for this is to use Modula whenever is possible. Modula-2 code is clearer, easier to understand, and is likely to be more reliable. It does mean, however, that two separate bootstrap files have to be produced for EPROM programming. It is imperative that the link between the two, a jump location, is set correctly. One EPROM is used to hold both the assembler and the Modula-2 bootstrap programs.

a) Run Assembly Routine

This routine starts first with the initialisation of the hardware system. It consists of the following functions:

* Set-up the different segment registers (i.e. code, data, extra, and stack pointer registers).
* Set-up the appropriate memory partitions (i.e. upper chip select, lower chip select, middle chip select, etc.). These are important to be set at this point. The different memory ranges are used as follows:

    * Upper memory range for bootstrap loader.
    * Middle memory range for application programs.
    * Lower memory range for RAM management.

A jump is then made to the Modula-2 initialisation routine.

b) Run Modula-2 Routine

This routine is located at the bottom of the boot EPROM. Its main function is to minimise the use of assembler for system initialisation.

When the absolute linker is run the data setting should be '83H'. This ensures that the interrupt vector area (and the planted return for a system interrupt) is not affected by this module. It consists of two main functions:

* Initialise serial line interface.
* Plant interrupt return vector.

When the Modula-2 initialisation is over, a jump is made to the start of the application software. This is accomplished through the use of a software interrupt (SWI) planted at the end of the Modula-2 routine.

E.5 RUN APPLICATION SOFTWARE [Chart E.3]

Application software is the partitioned task among the different nodes of the system (i.e. a sub-task on each node). This is written totally in Modula-2 language. It consists of two parts:

* Initialise Sub-Task.
* Run in Operation Mode.

E.5.1 Initialise Sub-Task [Chart E.4]

This is concerned with the initialisation of:

* Distributed variables.
* Communication channels.
* Interrupts.

a) Initialise Distributed Variables [Chart E.5]

In this part initialisation of all distributed variables takes place. This consists of setting up pointers and variable control blocks (VCB) for all variables. VCBs are defined for all distributed variables whether defined in this station (i.e. originals) or being imported from other stations (i.e. copies). VCBs are records used within the module to hold information about the status of each variable i.e. the name of variable, its size, its status (original, or copy), etc.

b) Initialise Channel for Communication Reception

The communication channel has to be set first for reception mode. This is essential as any one of the system stations may expect data from others at unpredictable time. (The transmission mode, however, is set always when the application program issues a transmit mode - see below).

c) Initialise and Set-Up Interrupts [Chart E.6]

A variety of interrupts are initialised and set before starting with the main sub-task. These consist of:

* Timed interrupts for control loops (i.e. level, or actuator loops).
* Timed interrupt for program-time purposes (i.e. a real-time clock).
* Event interrupt for the multi-process communication handler.

E.5.2 Run in Operation Mode [Chart E.3]

This mode starts first with enabling the different interrupts, starting timers, and then finally running a background process. The background process keeps looping indefinitely, until process termination, where two main things are achieved:

* Process Data Available.
* Act on Results.

a) Process Data Available

Process, here, acts on the dedicated task which has been partitioned for, i.e. executing and processing whatever procedures and data are needed for.

b) Act on Results

In the course of action, a process may need, however, to execute a transmit routine. This happens in two cases:

* Send a Message for Display.
* Run a Variable Transmit Mode.

In both cases above a transmit routine is issued after a preparation of the message is carried out. Preparation for a variable transmit mode is more complex, however. This is discussed below.

E.6 TRANSMISSION MODE - RUN A VARIABLE TRANSMIT MODE [Chart E.7]

A variable transmit mode is needed in either of two cases:

* To request a distributed variable copy from another station (Request-Global).
* To submit a calculated (i.e. updated), original, distributed variable by this station (Submit-Global).

Request-Global

In this case, a variable value is requested from another node by issuing a transmit routine. Two modes of operation are possible. Either control is returned back to the application program or else it is retained by the transmit routine until data is available (WaitFor-Data). These two cases are designed to allow for different program implementations (i.e. Wait or no-Wait).

Submit-Global

This routine is issued whenever an original distributed variable has been evaluated by the station (refer to section E.8 for more details).

E.7 RECEPTION MODE - EVENT SERVICE ROUTINE [Chart E.9]

A reception mode is entered whenever an event interrupt is received, following a message transfer. A service routine called a 'multi-process communication handler' receives and decodes the message.

According to the decoded message, a transfer is made to the proper server to take the required action. The following servers currently exist:

* Receive a Message for Display - Server 1.
* Reply for a Distributed Variable (copy) - Server 2.
* Request for a Distributed Variable (original) - Server 3.
* Request to Synchronise All Local Clocks - Server 4.

Similar actions are taken at the start and at the end of each server; i.e. acknowledge message reception at the start and return from interrupt at the end.

a) Receive a Message for Display - Server 1

This server is used to receive and then display a message.

b) Reply for a Distributed Variable (copy) - Server 2

This server uses a 'Check-RecvData' routine to check a reply for a requested variable copy, needed by this station, then stores the variable copy.

c) Request for a Distributed Variable (original) - Server 3

This server is used to deal with a variable request issued by another station. A check is made first whether the particular variable has been updated. If so, a transmit routine is issued and a copy of the variable is sent. Alternatively, the request message is stored for subsequent processing i.e. whenever the variable is available.

d) Request to Synchronise All Local Clocks - Server 4

This server implements the synchronisation of all local clocks according to a pre-defined master clock (chosen in any one of the system stations). The information received is an update from the master clock. This is used to update a global register variable.

E.8 REPEATED ROUTINES

These are various routines that are called repeatedly throughout the main kernel module, MAIN-DISTKERNEL, and used by the application program. A list of them is given below:

* Return From Interrupt Routine.
* Transmit a Message-Frame Routine.
* Validate Routine.
* Submit-Global Routine.
* CheckRecv-Data Routine.
* WaitFor-Data Routine.

a) Return From Interrupt Routine [Chart E.12]

This routine is used excessively by the multi-process communication handler. It achieves three main things:

* Set-up the channel and buffers for a reception mode.
* Enable interrupts for further activation and hence servicing.
* Return to background process.

b) Transmit a Message-Frame Routine [Chart E.13]

This routine is used whenever a message frame is to be transmitted. It, first, assembles the message according to its constituent parts (i.e. frame type, number of data bytes, data segment, address of both source and destination, etc.). Then it sets-up a channel and buffers for transmission mode. Finally, it sends the message frame by issuing a 'Send-Data' routine, then returning to the background process.

c) Validate Routine [Chart E.14]

This routine acts on distributed variables, whether originals or copies. It updates flags in the variable control block (VCB). This routine is used excessively by the application program before accessing variables for further processing.

d) Submit-Global Routine [Chart E.15]

This routine is used to transmit a copy of an original variable, after being evaluated by this station. A transmit mode process is issued for an original variable in two cases:

* A request is received from other stations (i.e. Request-Global).
* An original variable has been evaluated in the current station.

In both cases, however, the following series of actions are taken before 'Transmit a Message-Frame Routine' is issued:

* Check whether the variable has been evaluated (i.e. Validate).
* Check if there is any request for that variable.
* Transfer variable name and data into an output buffer.
* Prepare and set-up for transmitting the variable.

e) Check-RecvData Routine [Chart E.16]

This routine is used to check and subsequently store requested copies of variables. A check is made first on the variable, compared with a list of variables, to ensure two points:

* Such a variable exists within the requested list of that particular station.
* The variable has not been updated prior to this time.

Having accomplished the above tasks, the variable is stored in its data buffer, and the variable control block (VCB) is set indicating the validity of the variable for subsequent use and access.

f) WaitFor-Data Routine [Chart E.17]

This routine is used whenever the application program waits while a copy of a variable (i.e. a variable copy) is being requested. Normally, the application program sends or requests for a variable early in the program, that is before intending to use the variable immediately. This, in fact, coincides with the communication strategy of the system, i.e. non-blocking transmission. In some cases, however, the application program has nothing to do, at later stages, than waiting for variables update (supplied by other stations) to proceed further in the program. Hence, this routine is used to control such an action. It actually relies on 'Validate' routine to achieve its task. It keeps looping and checking the flags in the variable control block (VCB) until the variable is valid for use.

CHARTS

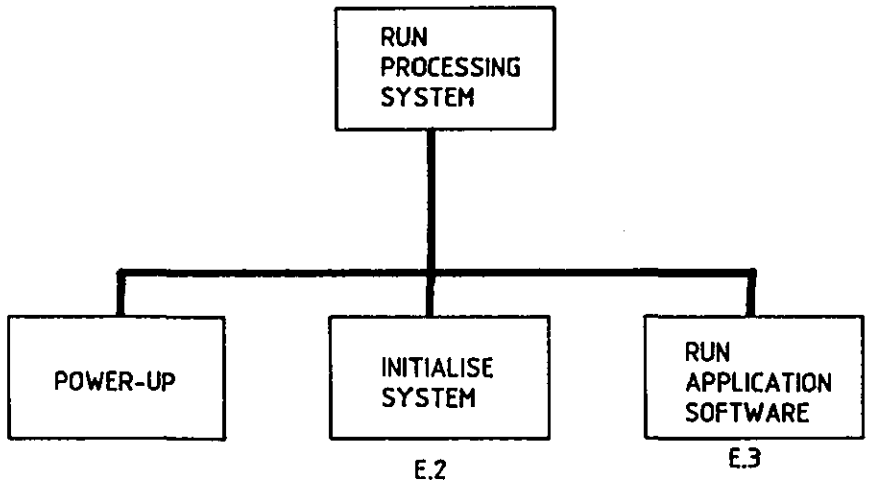


CHART E.1 RUN PROCESSING SYSTEM

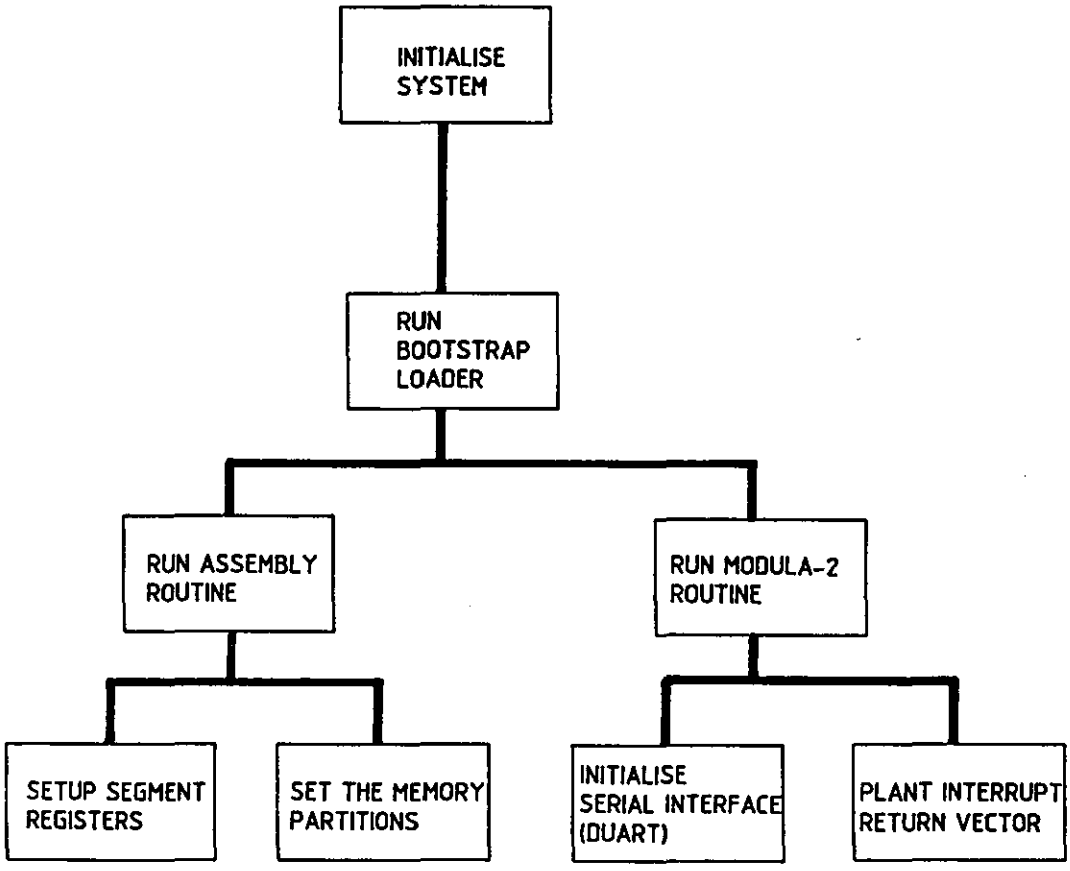


CHART E.2 INITIALISE SYSTEM

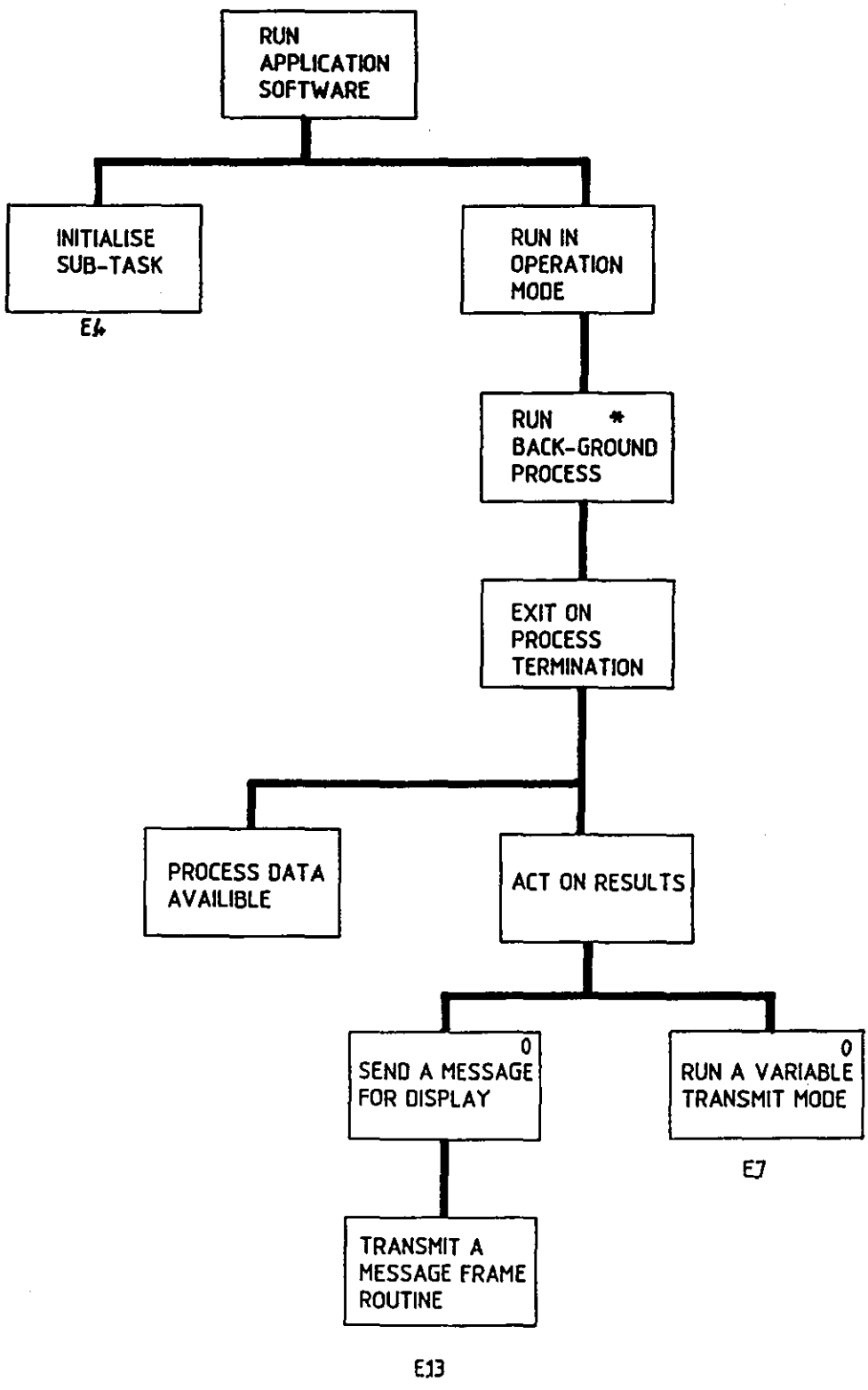


CHART E.3 RUN APPLICATION SOFTWARE

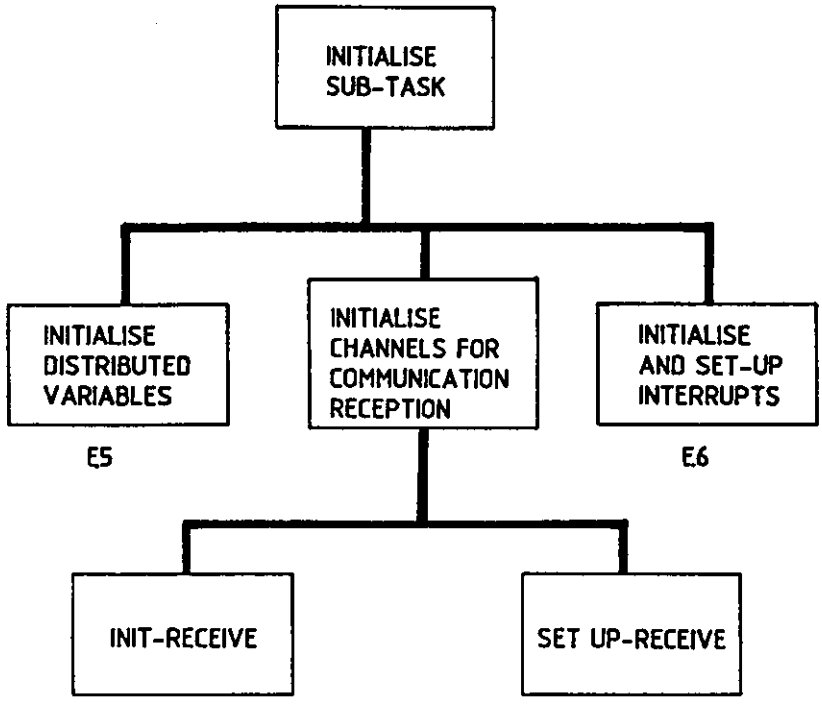


CHART E.4 INITIALISE SUB-TASK

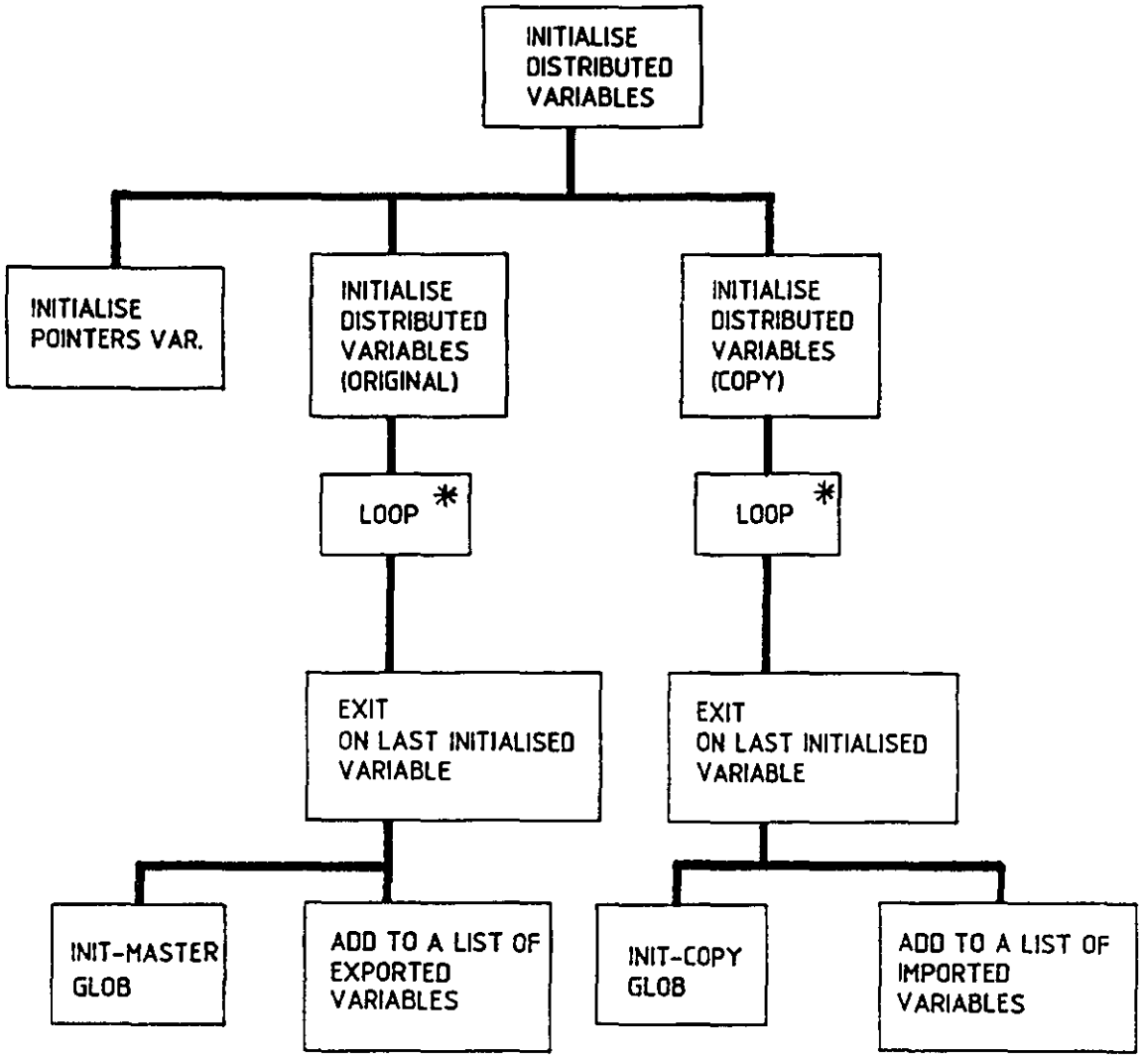


CHART E.5 INITIALISE DISTRIBUTED VARIABLES

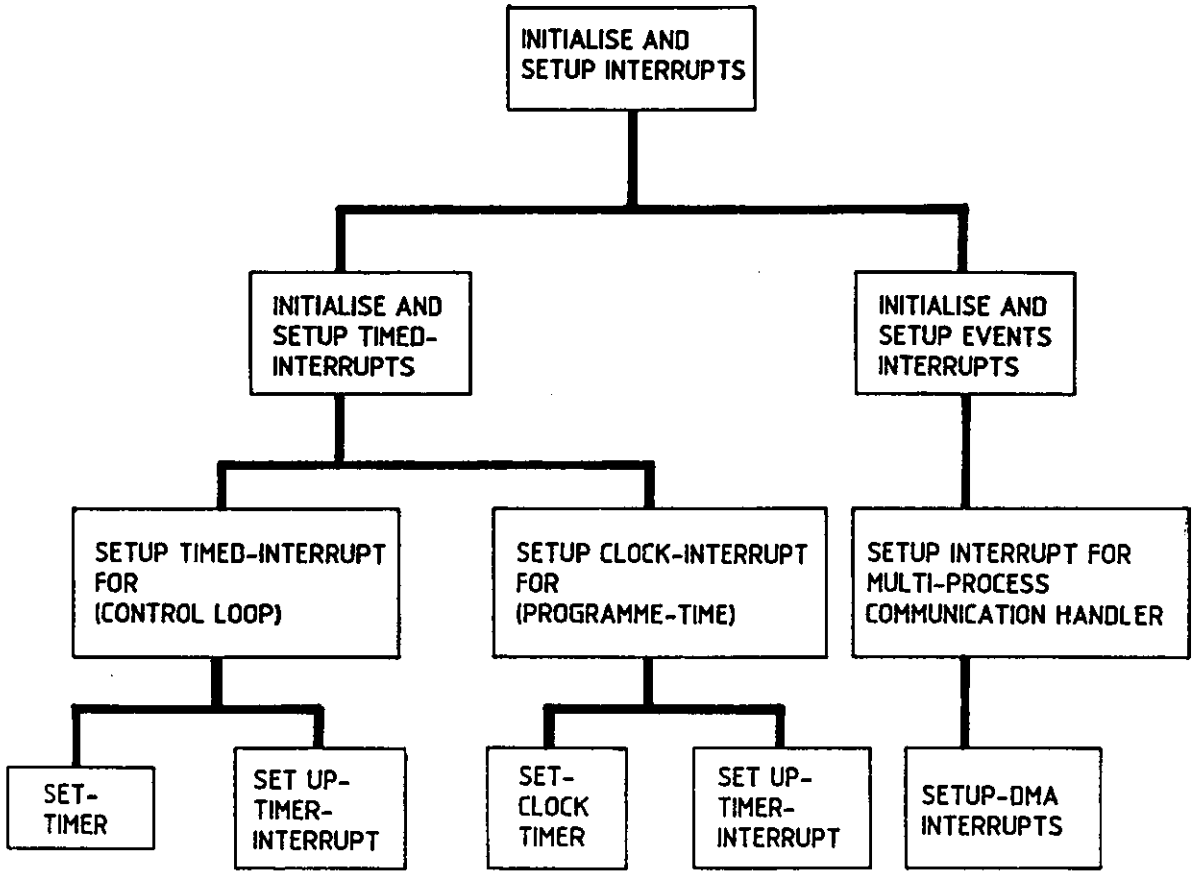


CHART E.6 INITIALISE AND SETUP INTERRUPTS

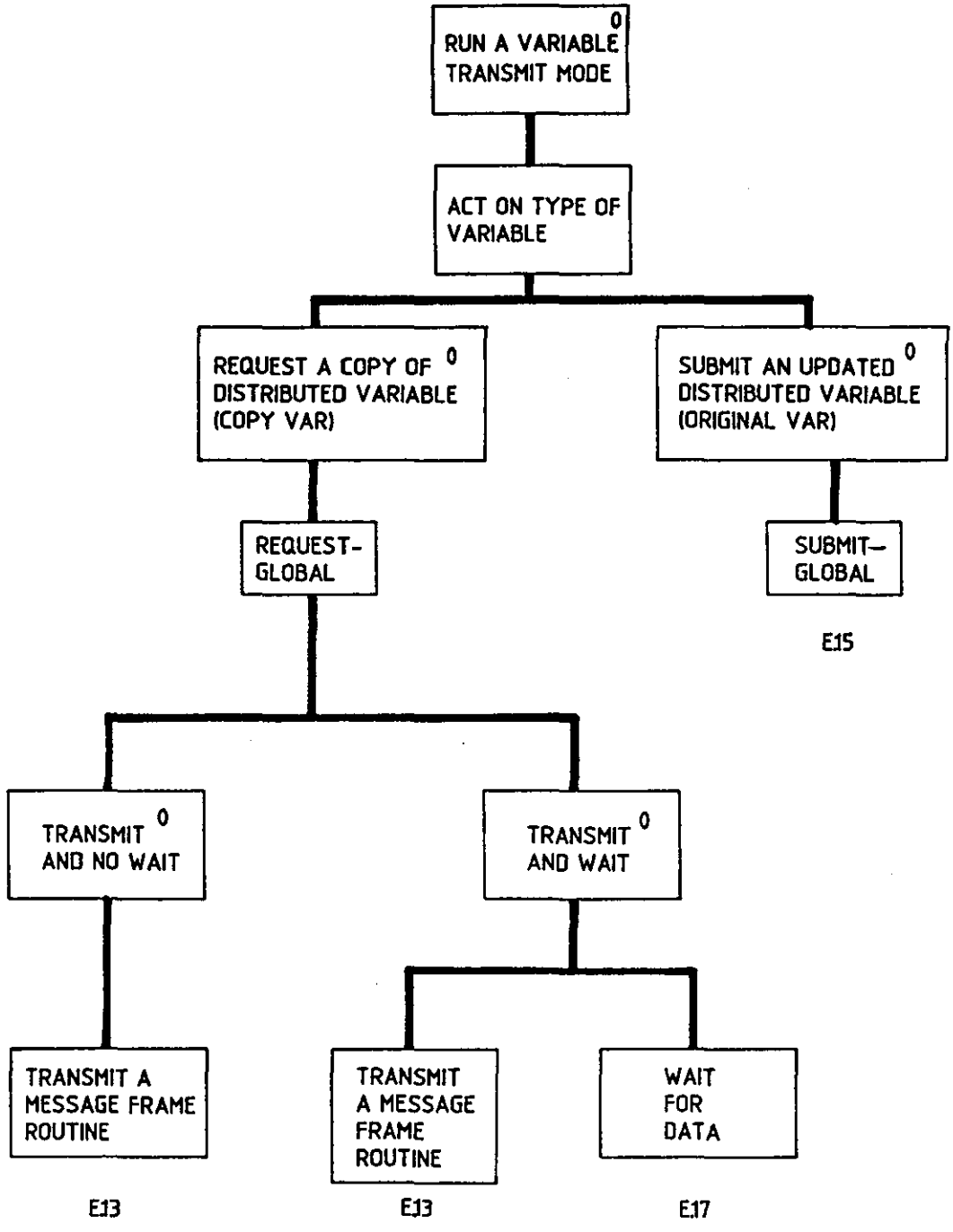


CHART E.7 RUN A VARIABLE TRANSMIT MODE

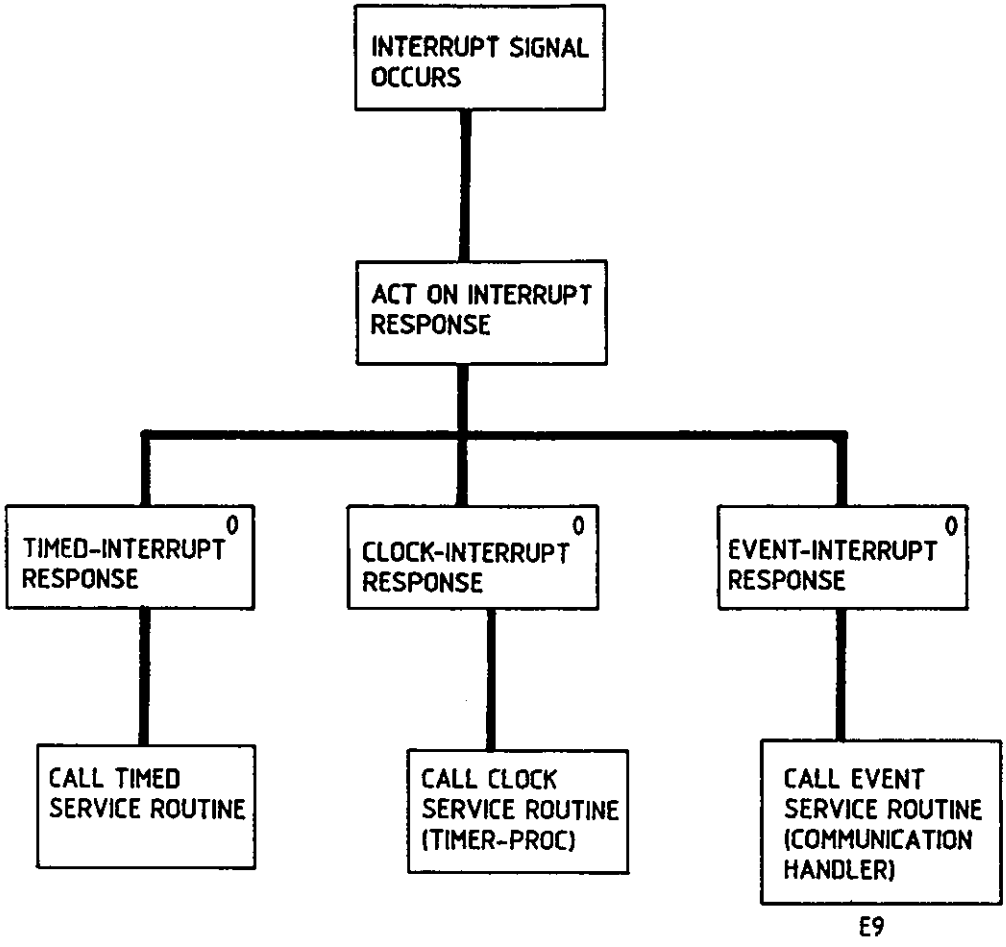


CHART E.8 INTERRUPTS

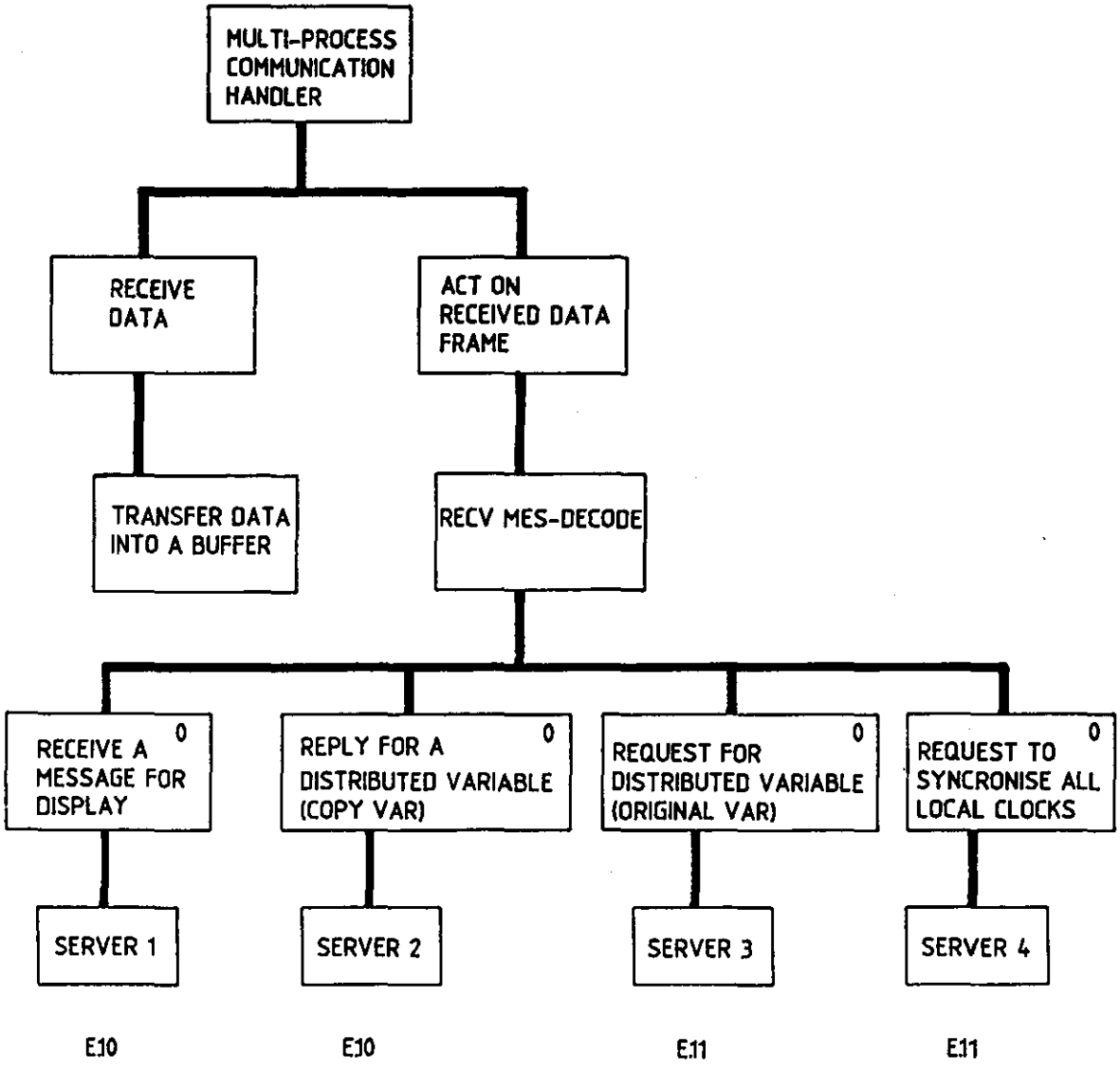
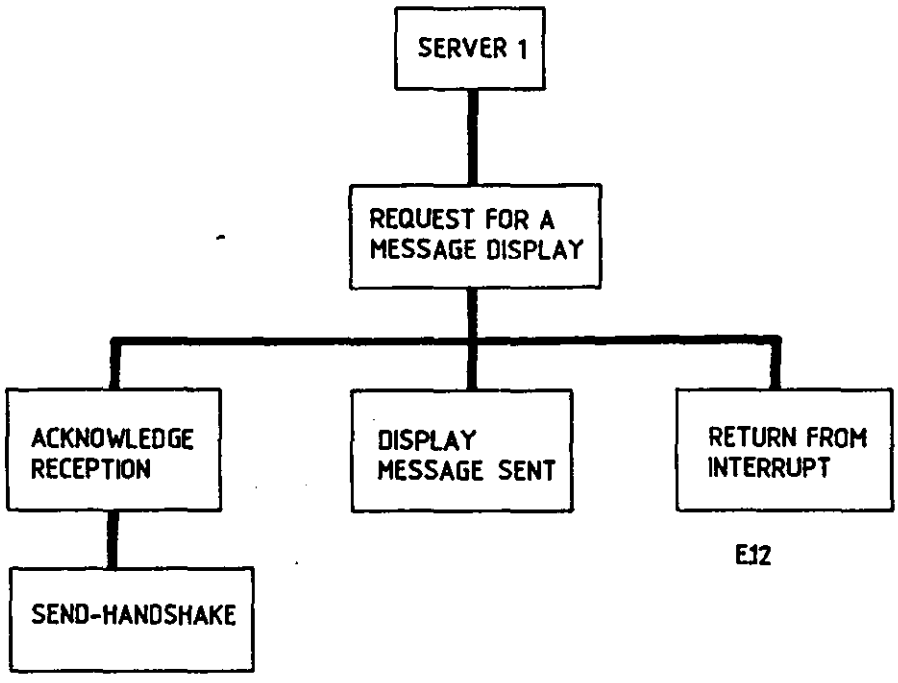


CHART E.9 EVENT SERVICE ROUTINE



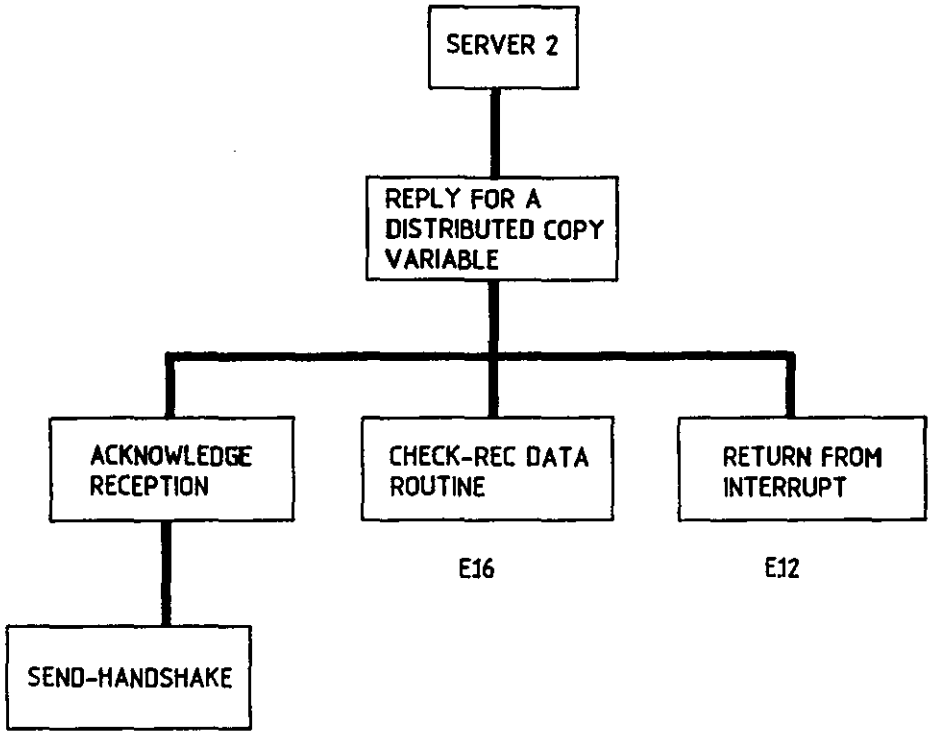
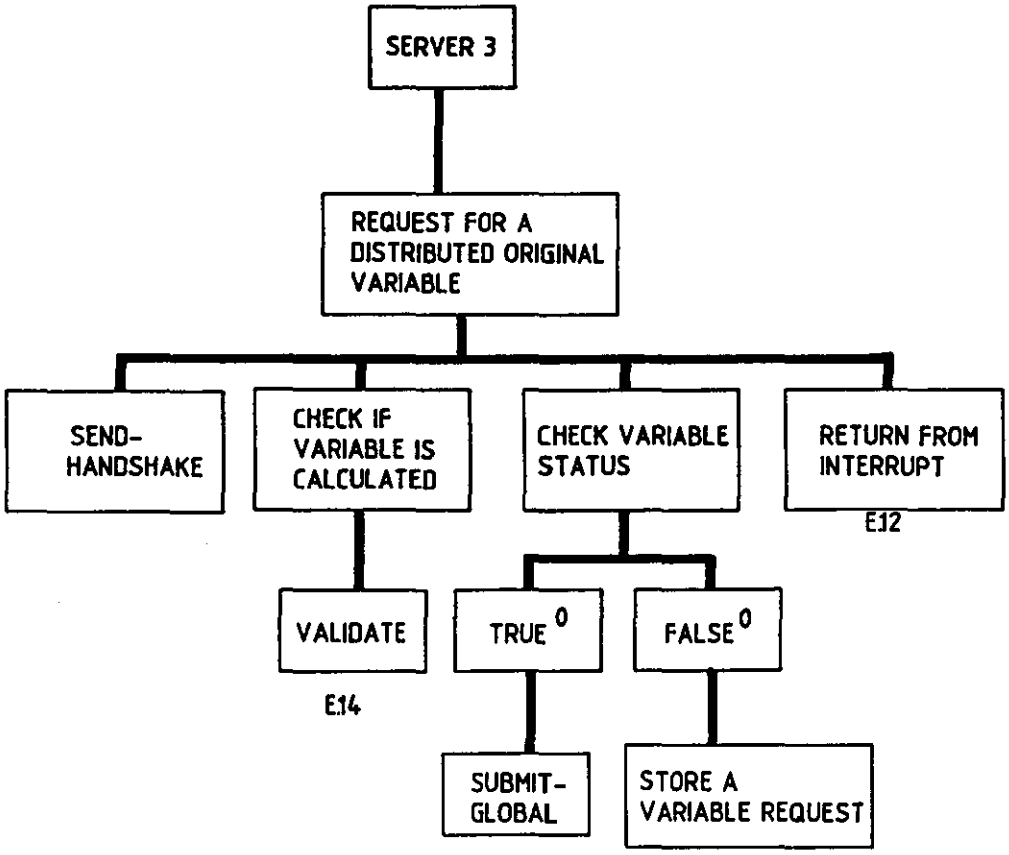


CHART E.10 SERVER 1 AND SERVER 2



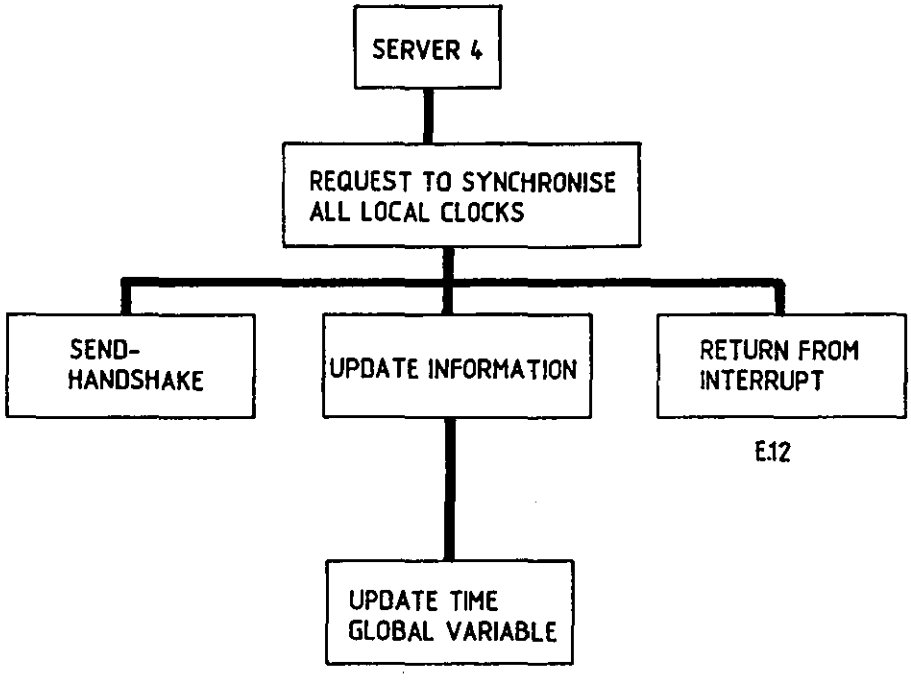


CHART E.11 SERVER 3 AND SERVER 4

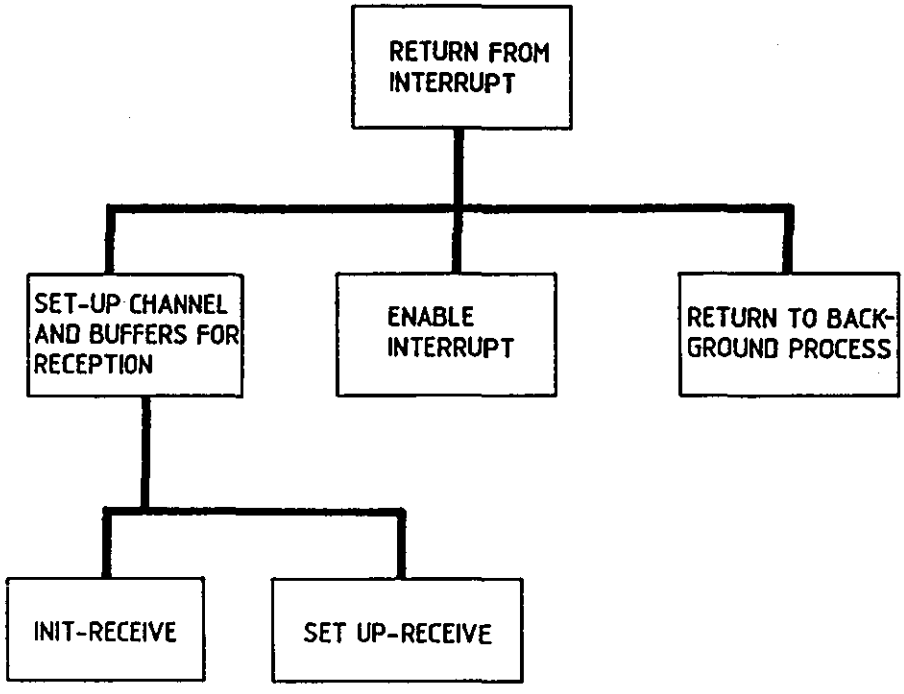


CHART E.12 'RETURN FROM INTERRUPT' ROUTINE

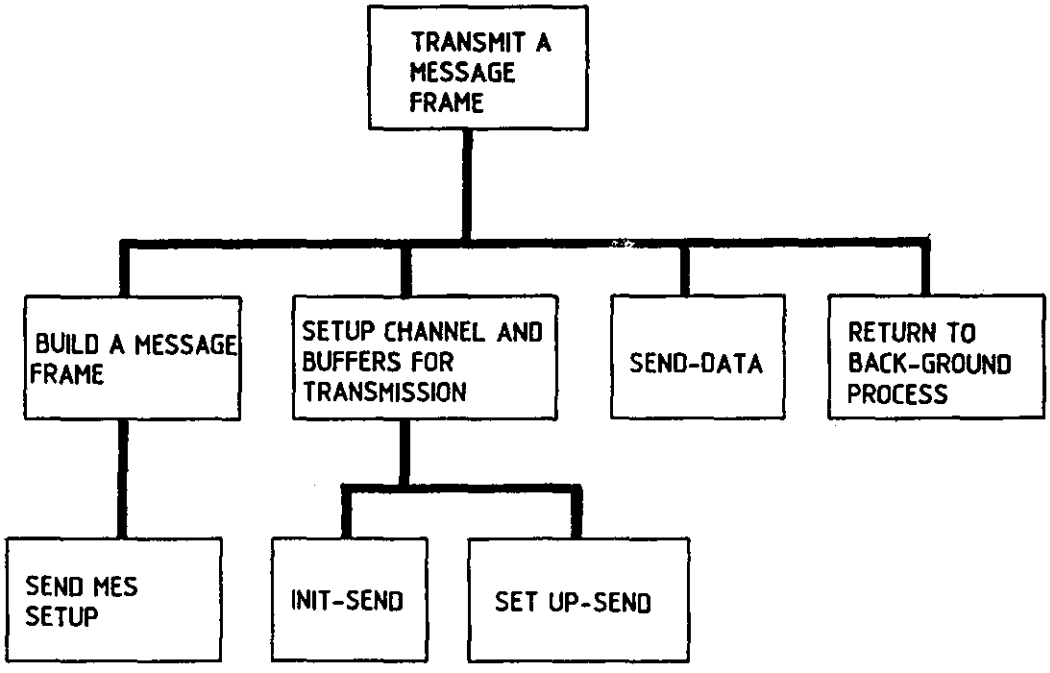


CHART E.13 'TRANSMIT A MESSAGE FRAME' ROUTINE

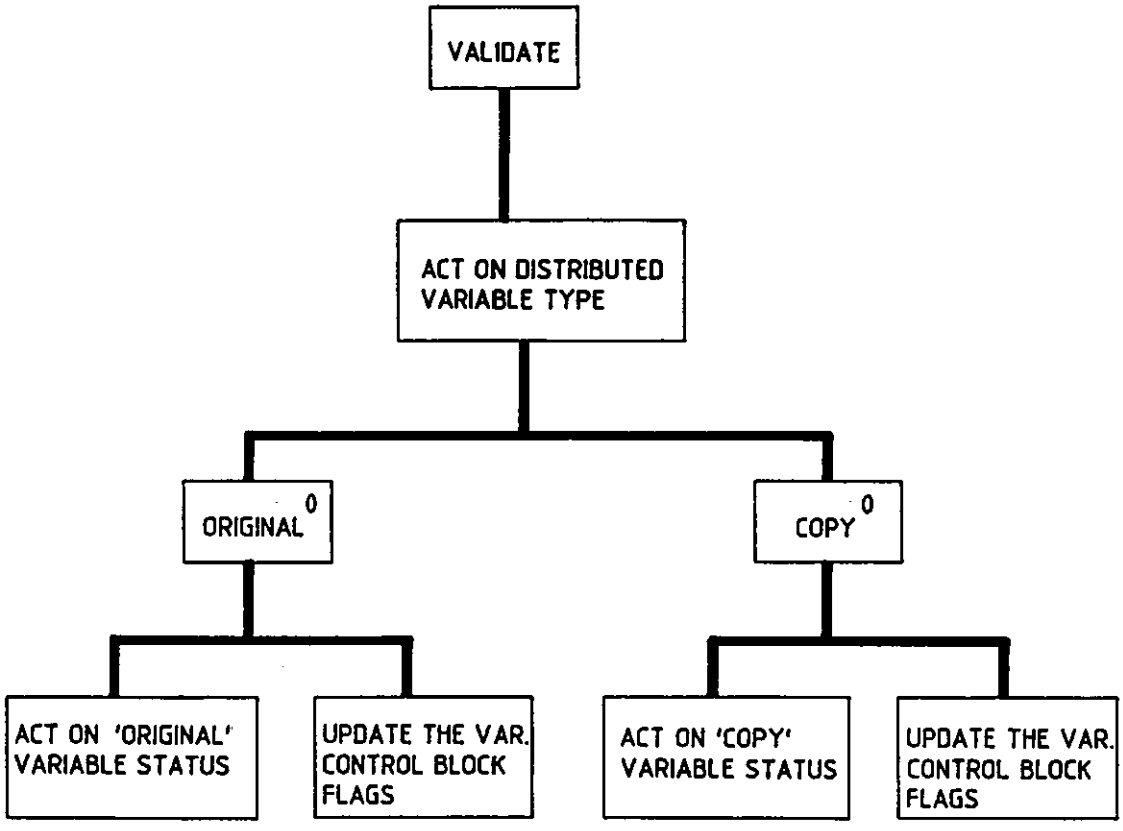


CHART E.14 'VALIDATE' ROUTINE

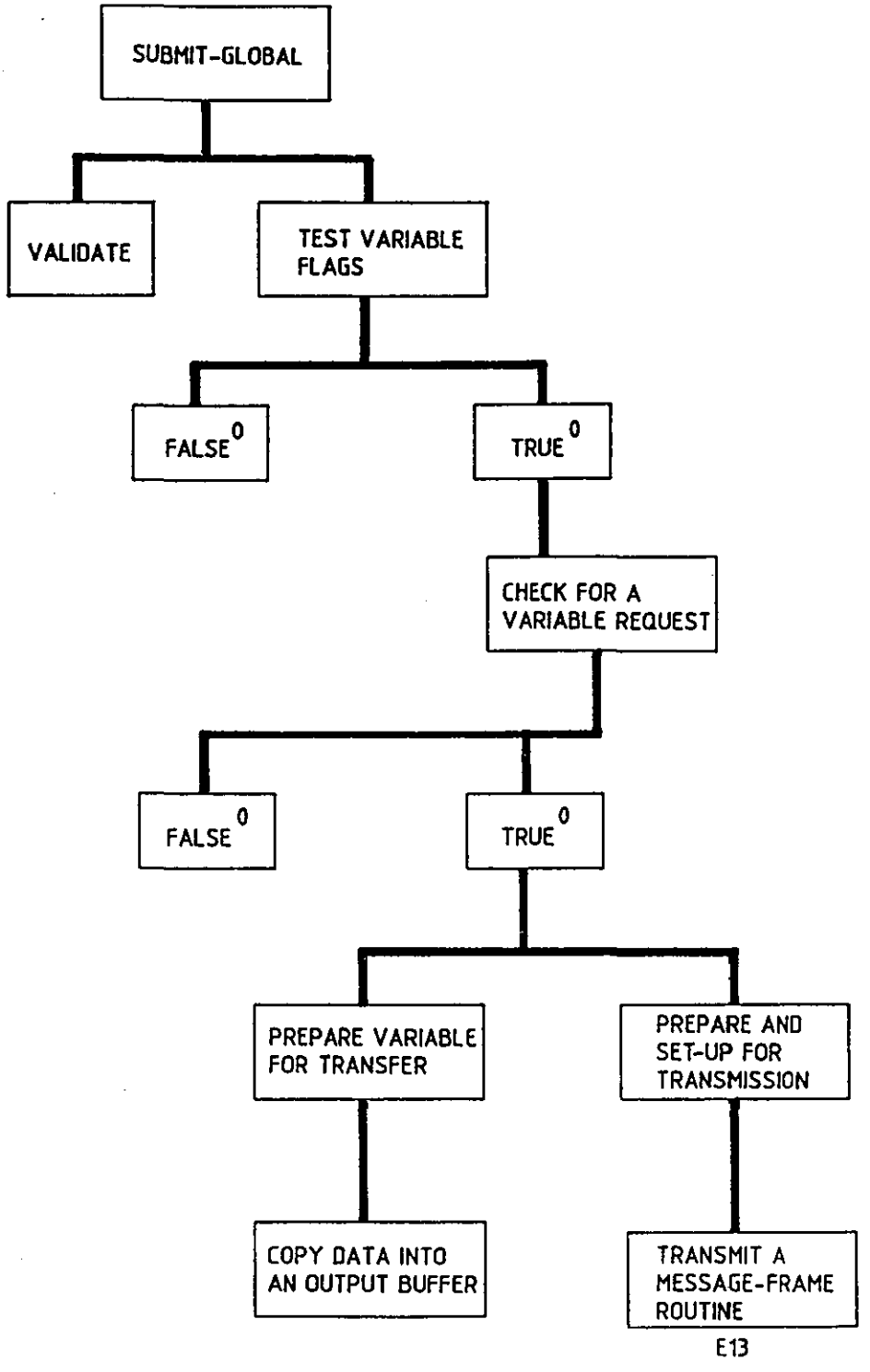


CHART E.15 'SUBMIT-GLOBAL' ROUTINE

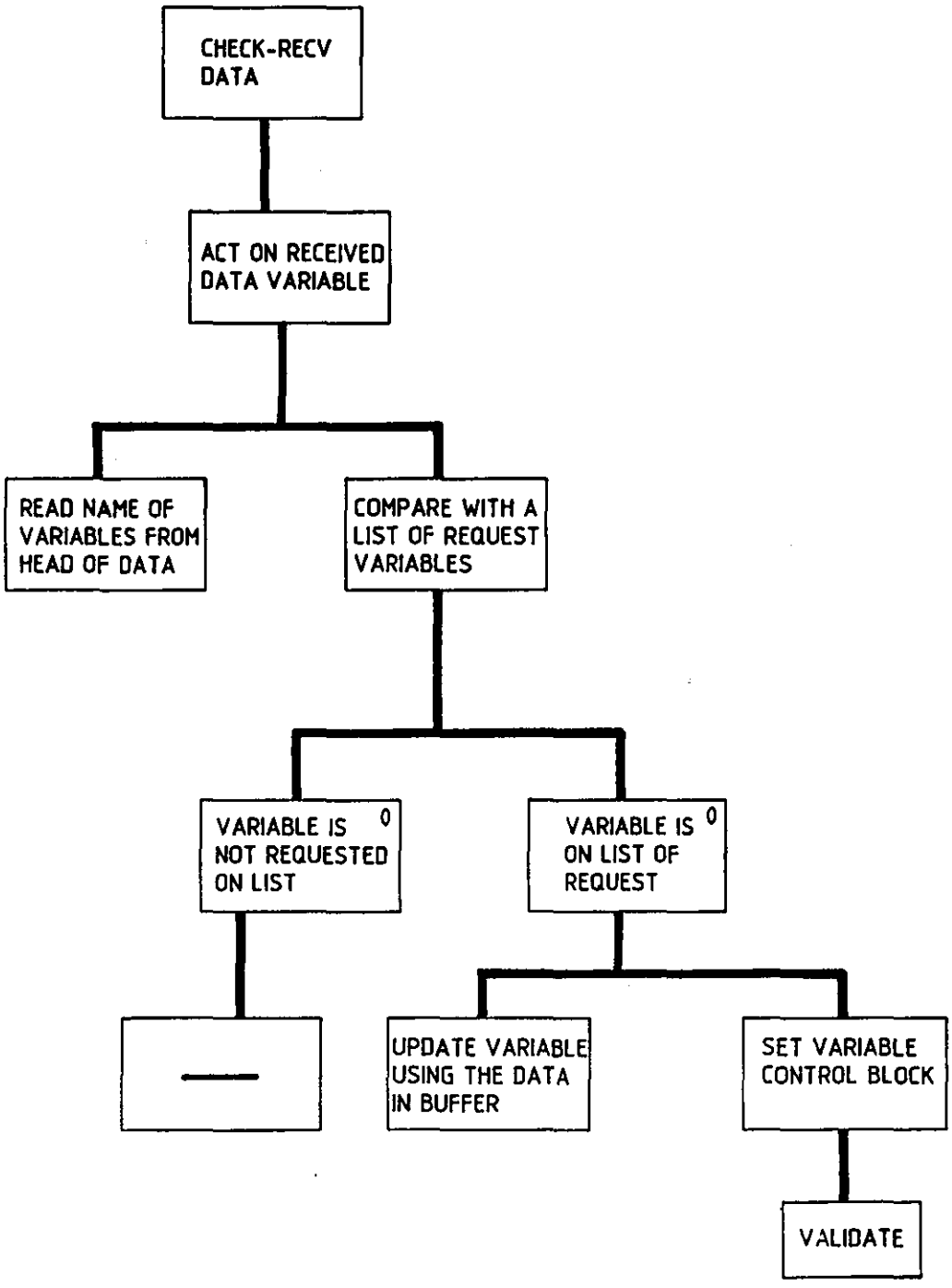


CHART E.16 'CHECK-RECVDATA' ROUTINE

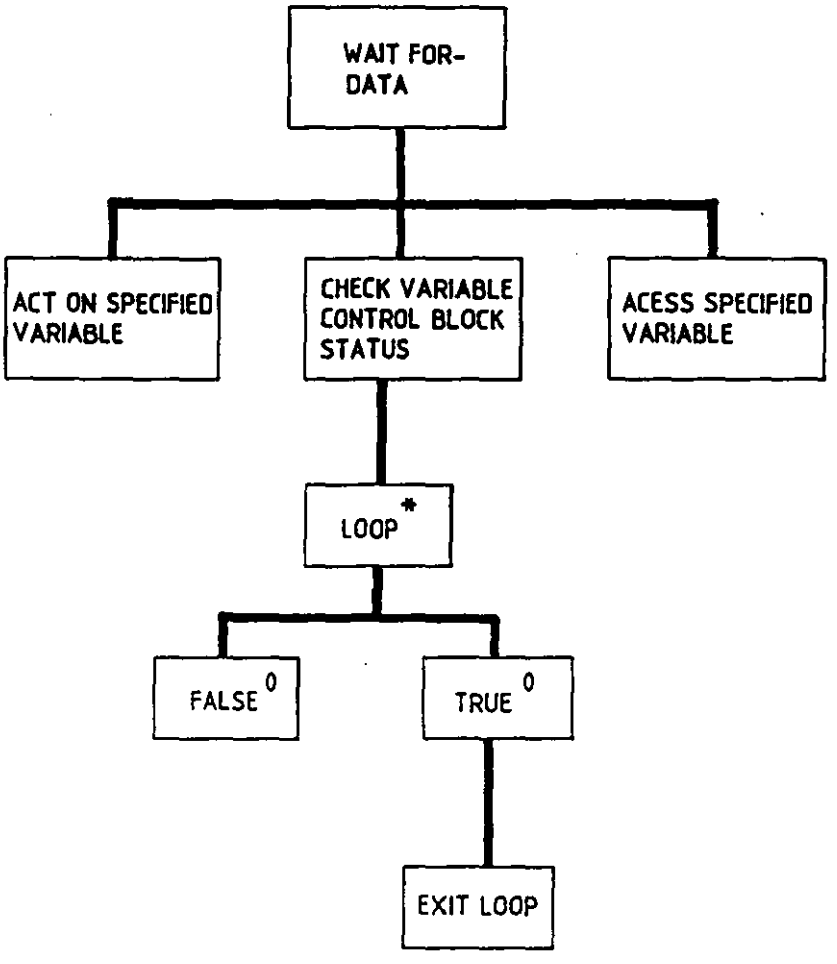


CHART E.17 'WAIT FOR-DATA' ROUTINE