FAULT TOLERANCE IN COMPUTER SYSTEMS

M.J.BURY

FAULT TOLERANCE IN COMPUTER SYSTEMS

Michael John Bury

A dissertation submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, for the degree of Master of Science in Engineering.

Johannesburg, 1986.

DECLARATION

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science in Engineering in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

M J Bury

M. J. BURY

19 th day of August 1986.

ABSTRACT

In many computer applications, it is necessary to ensure that the probability of failure is as low as possible. The degree of reliability required of the system is determined by its application. High reliability is particularly important in systems where computer failure could lead to loss of life, or to injury, or to financial loss.

Much research has endeavoured to develop techniques for reducing the probability of computer failure. In this dissertation, such techniques are described and discussed.

The dissertation proceeds to describe the development of an experimental fault-tolerant computer system which is sufficiently flexible to allow the examination of several techniques for achieving high reliability. Particular issues arising from the application of the techniques of triple-modular redundancy and software-implemented fault-tolerance to the system are discussed.

ACKNOWLEDGEMENTS

The author wishes to express his thanks to:

The National Institute for Aeronautics and Systems Technology of the Council for Scientific and Industrial Research, for sponsoring the project.

Professor Mike Rodd, who supervised the project, and provided extensive moral support in addition to considerable practical advice.

Mr. Hein Smit, who was instrumental in setting up the project to the satisfaction of the CSIR, and gave encouragement and assistance whenever it was needed.

Mr. Carel Combrinck and Mr. Johan Prinsloo, of NIAST, who also provided willing assistance at all times.

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 - INTRODUCTION

1.1 Introduction

Since the early use of computers in which they were installed primarily to produce answers to numerical problems, they have become integrated more and more into our everyday environment. We can now find them in a variety of forms, strapped to our wrists, installed in our cars, and suspended hundreds of kilometers above us.

With computers taking such an active part in our lives, their failure can often cause dangerous situations, where death, injury or financial loss can result. Consider, for example, a nuclear power plant, controlled by a process system using a series of digital computers. The failure of a computer can clearly be very serious, and it is naturally desirable to prevent such situations from arising. This has led to much work in designing computer systems which are as reliable as possible.

1.2 Minimization of Computer-related Dangerous Situations

Three main philosophies have emerged for the minimization of computer-related danger. These may be summarized as:

- Fault avoidance
- Fail-safety
- Fault tolerance

1.2.1 Fault Avoidance

An obvious way of preventing computers from causing harm to the plant, or to the environment which they control, is to make sure that they never fail! The philosophy which attempts to accomplish this goal is known as "fault avoidance".

Fault avoidance requires that the physical components of a computer system, and their assembly methods, are as perfect as pos-

sible. The cost of obtaining near-perfect components is often excessive, and maintenance staff must be continuously available because the system ceases to operate upon first failure. So, fault-avoidance techniques are clearly expensive and imperfect [1], and may consequently not result in adequate reliability.

1.2.2 Fail-Safety

In view of the problems relating to the design of perfect systems, the emphasis in research has focused on ensuring that computer failures do not lead to harm if and when they occur. This leads to another philosophy, namely that of "fail-safety".

To achieve fail-safety, it is necessary that when a computer system ceases to operate, it does so in such a manner that it can have no harmful effect on the environment over which it has influence. A particularly good example of the application of fail-safety techniques can be found in the area of railway signalling. If the system which manipulates the signals of a railway system fails, then all affected signals are set to STOP, so that trains in the area come to a halt, and hence avoid collision or derailment.

1.2.3 Fault Tolerance

There are situations, however, where the removal of the computer from a system cannot be accomplished without undesirable side-effects.

In many industrial processes, loss of control spells ruin of the product, with the additional possibility of permanent damage to equipment. In such a circumstance, a fail-safe end to control does little to prevent considerable financial loss.

An even worse case may be considered: the failure of a fly-by-wire aircraft control system (one in which control signals to the aircraft take the form of electrical signals rather than mechanical links) could lead to the loss of the aircraft and

crew. Mechanical back-up systems may be used in certain cases, but experimental aircraft are being developed which depend entirely upon the fast and accurate capabilities of a computer to maintain controllability [2].

In the case of remote equipment, such as weather monitoring stations and satellites, it is not possible to effect early repair should the computer system fail. Unless self-repair and/or graceful degradation facilities are built into the system, use of the system is totally lost when a fault occurs. Once again, therefore, fail-safety is inadequate.

There is consequently a need for computer systems which operate even when there are faults in the system. This leads to the concept of "fault-tolerance", which has been defined as "the ability of a system to operate correctly in the presence of faults" [3], and is the central topic of this dissertation.

1.3 Investigations into Fault-Tolerance

Over the past decade, much research effort has been dedicated to the development of fault-tolerant computer systems, and this has resulted in a large number of techniques being proposed as suitable for particular applications.

By definition, a fault-tolerant system must be designed assuming that some components will fail. The key ingredient in all fault-tolerance techniques is therefore redundancy - of information, resources and/or time. The type and extent of the redundancy employed in the system depends on the technique used, as well as on the intended application. The techniques of fault-tolerance fall into two loosely defined categories:

- replication

- back-up

1.3.1 Replication

From the literature, "replication" is evidently the most popular technique being used today. Many identical or similar units are used, and all fault-free units are active, that is to say, they contribute to the operation of the system as a whole. When a unit's failure is detected, the system attempts to reconfigure with one unit less. Hence, execution time might lengthen, but all essential services are maintained.

1.3.2 Back-up

"Back-up" is the second widely used technique (See fig. 1). In this approach, only one unit is operational, while one or more units are available as spares. If the spare units are powered in the idle state, the system is referred to as "hot" back-up. Sometimes, unpowered spares are used, in an attempt to lower the spare failure rate. The units are connected to the process through a switching mechanism that keeps only one active at a time. The active processor performs comprehensive self-checking, and is switched out when faulty.

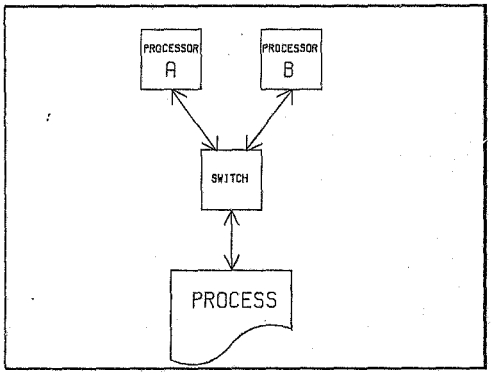


FIGURE 1 - The Hot Back-Up Configuration [4]

An extended version of hot back-up, known as "pair-and-a-spare" may also be implemented (See fig. 2). In this, the active and back-up units each consist of two modules, thus forming a tightly-coupled unit capable of reliable self-checking.

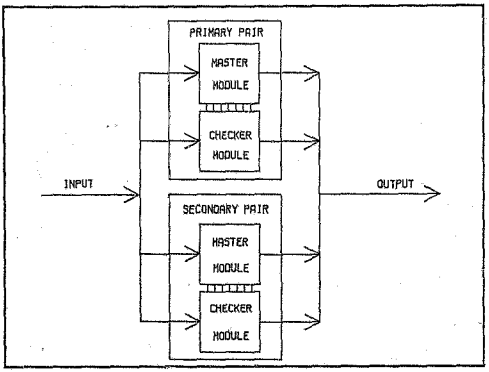


FIGURE 2 - The Pair-and-a-Spare Configuration [5]

1.4 Research Project Objectives

This project had two major objectives. Firstly, it aimed to generate insight into the field of fault-tolerance, and secondly to produce a flexible experimental system which could be used for studying various fault-tolerance techniques.

It was felt that at this stage in the development of fault-tolerant systems, little practical experience exists, creating a need for the project. However, the effort investigating fault-tolerance was not simply to facilitate the production of the experimental system; it was also desirable to gain expertise in the field of fault-tolerance, with a view to applying the knowledge in future projects.

It was therefore necessary to broaden the theoretical side of the project beyond that necessary for the production of the

experimental system. To this end, various aspects of the topic required further attention, such as:

- investigations into the theory of fault-tolerance

- study of important current systems, both commercially available, and undergoing development

It should be pointed out that the composition of much of the experimental system developed was defined by available equipment and tools. These constraints are discussed in the appropriate place in the dissertation.

Development of the experimental system consisted of:

- selection of a representative fault-tolerance technique for demonstration of the system

- investigations into software engineering techniques

- production of a suite of software modules for use in the various system configurations

- integration of the software and hardware components of the system

- application to a real-time, but simple servo-control system so as to provide a live demonstration

1.5 Overview of the Dissertation

The remainder of this dissertation covers the following areas:

Chapter 2 - Fault-Tolerance - a discussion of techniques for achieving fault-tolerance and for evaluating fault-tolerant systems

Chapter 3 - Software Development Techniques - Structured design, structured programming, data flow techniques, top-down design, verification, validation, and debugging

Chapter 4 - System Description - System requirements, functional specification of the system

Chapter 5 - System Design - Hardware characteristics, software requirements, functional specification of the software, and software characteristics

Chapter 6 - System Integration - Development of the software required for system control and testing, and implementation of the system

Chapter 7 - Conclusion - A brief summary of the results of the research followed by a discussion of the more important findings and conclusions as well as unsolved issues

1.6 Summary

Because of the wide use of computers in critical applications, it has become necessary for attention to be given to the problem of computer faults. The main computer-fault handling techniques are

- Fault avoidance,
- Fail-safety, and
- Fault tolerance.

Research effort into the technique of fault tolerance has led to many fault tolerance methods, which can be loosely divided into the categories of

- Replication, and
- Back-up.

Chapter 1 INTRODUCTION

The goals of the current project were to gain insight into the field of fault-tolerance, and to produce a flexible experimental system for use in the study of fault-tolerance techniques.

To provide a sound basis for the development of an experimental system, an extensive study into fault tolerance was undertaken, and this is covered in the next few chapters.

Chapter 2 - FAULT TOLERANCE - AN OVERVIEW

2.1 Introduction

The benefits of employing fault-tolerance to computer system design are many, but in essence, lead to reduced system lifetime costs. In this chapter, various aspects of fault-tolerance are explored, and the most critical areas are highlighted. Many current fault-tolerant systems are referred to, and are described in detail in appendix 1.

2.1.1 Motivation for the use of Fault-tolerance Techniques

A number of factors have led to the development of fault-tolerance techniques, the most important being:

Reliability. Since maintenance and general downtime costs have risen to become a large proportion of total system lifetime cost, it is obviously desirable that systems should be designed to fail as seldom as possible.

Data integrity. Because computers are used in highly critical areas, it is essential that data corruption is highly improbable.

Availability. From a users point of view, it is necessary that computer down-time is minimized, especially when the service provided by the system involves human interaction.

Graceful Degradation. Remote computer equipment must function for as long as possible without repair. In the extreme situation of, say, an unmanned spacecraft, no repairs at all are possible; any failure should not lead to a total system failure, but merely a drop in performance.

It should be noted that these are the most obvious points of improvement brought about through the use of fault-tolerance techniques; other facilities which may be provided by the application of the philosophy include:

On-line maintenance (the ability to perform repairs without switching the system off). A fault-tolerant system would regard the removal of a single unit as a unit failure, and continue in its normal fault-handling manner.

Fail-safe operation (the prevention of dangerous effects caused by failure of the computer). The failure of a unit can be automatically prevented from affecting the environment, by the fault-handling mechanism.

2.1.2 Criteria to be Satisfied by Fault-Tolerant Systems

The reliability requirements which must be imposed on a system naturally depend on the intended application [6]. For example, the primary function of information storage systems is the safe storage of data, so such systems can tolerate short losses of service, but not data loss or corruption. On the other hand, telephone exchanges require high availability, so that users do not have to wait for intolerably long periods before the required service is provided, but it doesn't necessarily matter if a few wrong connections are made. In the extreme case, life-critical systems can tolerate no failures at all.

The computer must be capable of a recovery time (the time it takes the system to function acceptably, after a fault) which is appropriate to the time-constant of the application (a measure of the speed of the system) (See fig. 3).

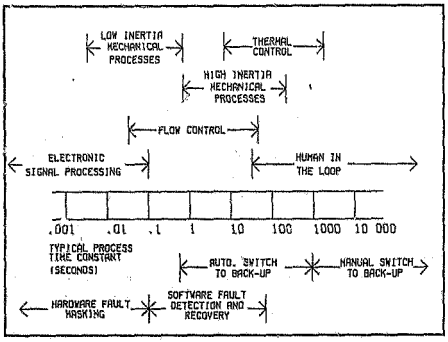


FIGURE 3 - Time Constants of Systems [4]

2.1.3 Basic Terms and Concepts

Many differing interpretations are placed on a number of terms and concepts used in the field of fault-tolerance. In order to avoid misinterpretation of terms used in this dissertation, the more important terms and concepts are defined below:

- Fault - any state of a computer's hardware or software which could cause the computer to operate incorrectly, or not at all

- Common-mode fault - a fault which affects all parts of the system simultaneously (for example, electromagnetic interference)

- Error - incorrect operation of the computer, leading to incorrect data or to invalid actions by the computer

- Reliability - the probability that the computer will operate correctly during a given time period

- Fault masking - the prevention of error propagation into other parts of the system

- Fault tolerance - the ability of a system to operate correctly in the presence of faults. The concept embodies:

  - fault detection - the discovery of a fault

  - fault recovery - removal of the effects of the fault and isolation of the faulty system component (i.e. ensuring that the component cannot exert any influence on the operation of the system as a whole)

2.1.4 Causes of Faults

In order to combat the occurrence of faults, it is necessary to know the way in which they arise (See fig. 4). It is often possible to perform "preventative design" (this entails the construction of the system in such a way that susceptibility to faults is minimized), which will cut down on the number of faults that must be catered for by the fault-tolerance mechanism.

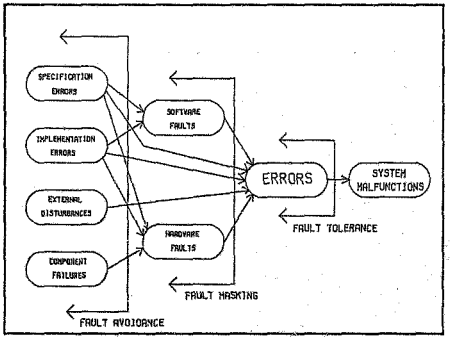


FIGURE 4 - Causes of Faults [7]

In essence, the origins of faults can be grouped into four categories:

- specification faults
- implementation faults
- component failures
- external disturbances

The way in which these fault origins relate to the conventional software development cycle is shown in figure 5.

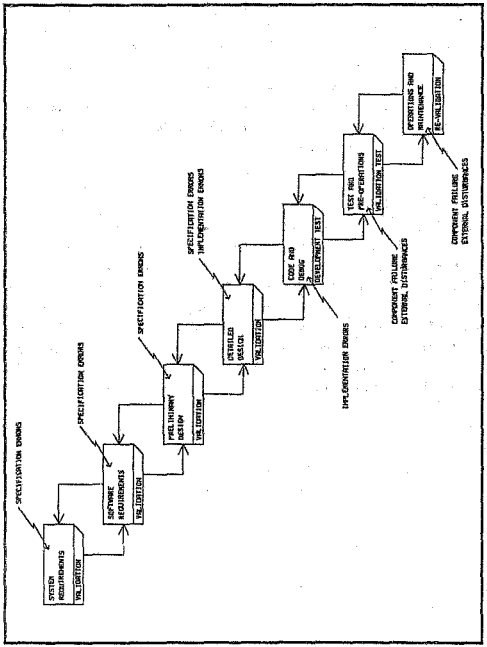


FIGURE 5 - Conventional Software Development and the Origins of Faults [8]

Specification faults.

The first possible cause of specification faults is that of hardware and software design specification mistakes, which occur when the hardware or software has been specified in a way that does not meet with the requirements of the system. Secondly, architectural mistakes mean that the system has been designed in such a way that it is not able to perform all operations required of it. Finally, algorithm mistakes arise when an algorithm implemented in the system is incorrect.

Implementation faults.

Following on from the design specifications, the system is susceptible to implementation faults. These can be the result of, firstly, poor design, which implies that the design of the hardware does not meet all the requirements of the specification. Otherwise, such faults can originate in poor component selection, where unsuitable components have been chosen. Furthermore, poor construction of the hardware can lead to weak points in the system, or software coding faults can lead software which does not always perform according to the software specifications.

Component failures.

Component failure can arise from manufacturing errors, where a component has been incorrectly constructed, or from component flaws or component aging.

External disturbances.

Radiation is the one of the most prevalent external disturbances. External electromagnetic fields can alter the operation of the system so that it fails. Physical damage to the system can also occur from an external source, or unexpectedly severe environmental conditions can cause the system to fail. Finally, inappropriate man/machine interaction via control or maintenance

panels, where the operator has made a mistake or is not qualified to control the system, can lead to system failure.

2.1.5 Classification of Faults

Faults are classified using one or more of the following parameters:

- Cause - one of the possibilities mentioned in section 2.1.4, which caused the fault

- Nature - whether the fault is in the hardware or the software

- Duration - whether the fault is permanent, transient or pseudo-transient (e.g. pattern dependent)

- Extent - the amount of the system which is affected by the fault

- Value - whether the errors are determinate or indeterminate (i.e. whether the errors are always the same, or random)

Knowledge of these details enables effective counter-measures to be taken.

2.2 Techniques of Fault-Tolerance

As mentioned previously, redundancy forms the basis of a fault-tolerant system, and this redundancy may take the form of information redundancy, hardware redundancy, software redundancy and/or time redundancy. Each of these topics will be covered in the following sections.

2.2.1 An Outline of Fault-Tolerance

The sequence of handling a fault begins with the detection of an error. The system then attempts to diagnose the fault which caused the error, and prevent the damage from spreading (confinement). Thereafter, it is necessary to reconfigure the system to a valid state,, bypassing the faulty components, and to continue operation - as fully as possible. Finally, if possible, repair to the faulty component(s) should be made, thereby ultimately restoring full capabilities to the system. In the sections which follow, the various stages involved are discussed in depth.

Error Detection.

Faults and subsequent errors typically manifest themselves as invalid data. To detect errors and faults, therefore, it is necessary to detect invalid data. To determine the validity of data, two types of test are possible:

Voting

Bounds of reason

Voting. Several answers to a calculation are obtained typically using one of two methods:

- repeated calculations - Each calculation (for which the result is to be validated), is performed two or more times. The answers obtained in each repetition (which may be performed by different processors), are compared, and any inconsistency reveals that an error has occurred. If executed on only one processor, then this technique detects the occurrence of faults and non-determinate faults only, since a permanent, determinate fault would manifest itself in the same way in each calculation, misleading the system into believing that the result is valid. Also, if the calculations are to be run in separate processors, care must be taken to ensure that the executions are staggered in

time so that common-mode faults do not produce the same errors in all processors.

- duplicated calculations - Each calculation (for which the result is to be validated), is performed in two or more different ways (possibly by different computers). Any discrepancy in the results indicates the occurrence of an error. All types of fault are covered by this method, but extra effort is required in development of the algorithms, and extra program storage space is needed for the different versions of the calculation.

There are two possible ways in which the answers to these calculations may be compared:

- hardware - Dedicated circuits are used to compare the results of computations. This method is fast, but requires the addition of components, increasing the cost and the risk of failure - because of the extra components! Furthermore, in order for hardware voting to be used, all values of data must be simultaneously available. This leads to the possibility of a common-mode fault affecting all versions of the data in the same way, causing the voter to pass the incorrect value

- software - Voting is accomplished using a software module

Considerable effort is also required in both hardware and software voting to make the voting mechanism itself fault-tolerant.

Bounds of reason. In this technique, the value of a data element is checked against pre-defined limits, beyond which it is determined to be invalid. The test may be applied to any data element for which bounds of reason can be defined. The limits are usually characteristics of the application, or possible constraints imposed by the data-typing provided by the programming language.

Fault diagnosis.

Faults that are to be tolerated by the system must be well defined at an early stage, so that they can be specifically catered for in the design. Such faults must be automatically detected and localized by the system, using the characteristics of errors which have occurred, or can be made to occur, using diagnostic programs.

Damage Confinement.

In order to limit the effects of a fault, it must be possible to reset the system to a valid (correct) state after the occurrence of a fault, so that the system does not continue producing more and more errors.

Reconfiguration.

The system must automatically bypass defective components and yet keep all system functions, which are not dependent on the lost component, available to the user, with a possible reduction in processing speed.

Recovery.

Data which was found to be erroneous must be corrected. Otherwise, recovery will be in the form of resetting to a previous valid state (when possible), or to a predicted future state. An important goal in the recovery process is that every restart must be accomplished with a minimum recovery time, in order to minimize down-time losses.

Fault Treatment and Continued service.

The system should remain in a degraded state for as short a time as possible. If possible, the fault should be repaired or the faulty unit replaced so that fault-tolerant operation is resumed.

The following sections discuss the techniques used to put these principles into practice.

2.2.2 Information Redundancy

Information redundancy is the use of more information about the data than is actually needed by the application.

A widely used technique for implementing information redundancy is by the use of "data encoding". Numerous information-redundant codes have been developed to provide for detection and, in certain cases, correction of errors. A code constructed in such a way that any single error transforms a valid code into an invalid code is called a single-error detecting code. A simple form of such error detection is the single-bit parity check. Another type is "M-out-of-N" coding, where code words are N bits long, and always contain M "1"s.

A number of "checksum" error-detection codes exist. The checksum is calculated by summing the binary data that is to be moved from one point to another. When the data reaches its destination, the checksum is recalculated, and if the new value and the one calculated previously do not agree, then an error is indicated. These codes are useful in the transfer of blocks of data.

Possibly the most common extension of parity checking is the Hamming error-correcting code. Hamming codes can detect double errors, and correct single errors. Once a single binary error has been detected, it is easily corrected by complementation of the data bit in the identified position.

Fault-tolerant systems often incorporate information redundancy into the fault-tolerance mechanism, especially in the memory sections of the system. In some cases, however, the primary fault-tolerance mechanisms of the system are so effective that they make the reliability improvement brought about by the use of information redundancy negligible.

2.2.3 Hardware Redundancy

Hardware redundancy is the use of more physical equipment than is required by the application.

Hardware redundancy methods may be grouped into two categories:

- replication

- back-up

The addition of spare resources to either category results in what is often called an "hybrid" system.

Replication.

In the technique known as "replication", more than one resource is available to perform tasks required of the system. Memory, processing, and/or input and output units may be replicated, depending on the requirements of the application. All units in the system contribute to the operation of the system as a whole, and may be run out of close synchronization (where every instruction is executed at the same time in all processors) to avoid the effects of common-mode faults. An example of such a system is the Fault-tolerant Array Signal Processor [9], which uses a form of replication to perform space-based signal processing (See appendix 1 for more details).

Three important forms of replication are:

- dual redundant systems

- triple-modular redundant systems

- gracefully-degrading systems

Dual Redundant Systems. One of the simplest redundant systems is the dual system, in which the same tasks are executed on two different units, and the outputs compared. If the outputs do not agree, then an error is signalled. The system is incapable of deciding which of the two units has produced the error unless further testing is undertaken. This means that the system must be shut down when an error occurs.

Triple-modular Redundant Systems. These systems use three units, all performing the same calculations, and are capable of masking all single errors, as well as indicating which unit was responsible for the error. Furthermore, the systems are capable of detecting simultaneous errors in all units, because the vote will fail. The Triplex 32 system [4] utilizes TMR (Triple Modular Redundancy) to accomplish fault-tolerant process control, while the Software Implemented Fault Tolerance [10] system applies TMR to aircraft control (See appendix 1 for detailed descriptions).

After a faulty unit has been pinpointed, its outputs are ignored, while the good units continue operation as a dual system. When the faulty unit has been repaired, it is set to a state consistent with the other units, and the system returns to its original degree of fault-tolerance.

Voting may be accomplished in either hardware (See fig. 6), or in software (See fig. 7).

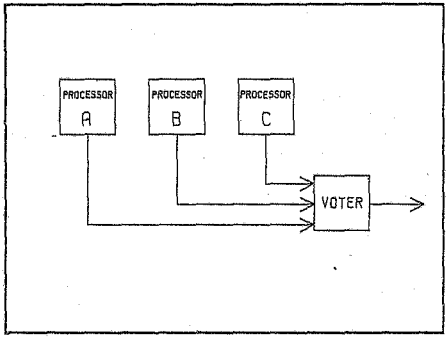


FIGURE 6 - Hardware Voting [4]

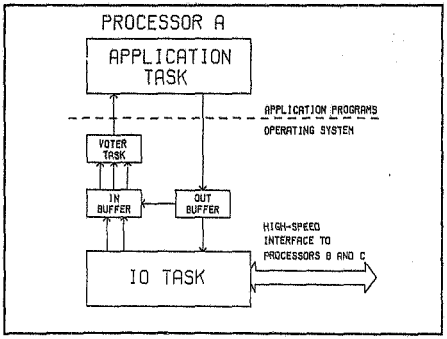


FIGURE 7 - Software Voting [4]

With voting, there is some loss of performance while data is passed through the voting mechanism. It is possible with hardware voting, however, to utilize parallel processing so that modules in the system are performing the next operation while the voter is finishing the previous one. A more serious drawback of hardware voting is that the voter components are unprotected, and expose the system to a single-point failure. One solution is to triplicate the voters (See fig. 8). The problem of common-mode faults is not overcome, however, because data must still be available to each voter at the same time. The major advantage of hardware voting is its speed, especially in control applications which often require large numbers of inputs and outputs.

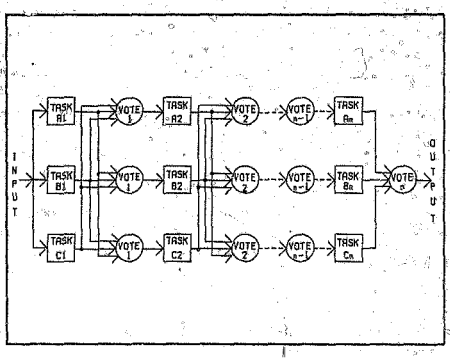


FIGURE 8 - Voter Triplication [5]

If more than three units are used in the system, then N-modular redundancy is being employed, where N is the number of units. Such replication may be used when inadequate reliability is provided by the triple-modular technique. There are usually an

odd number of units, so that a majority vote can always be obtained.

Gracefully-degrading Systems. When the multiple units in a system all perform different functions, then a gracefully-degrading system can be formed. Effective load sharing across system resources, and efficient communication between the units are necessary. When a unit fails, its load is shared among the other units. Hence, execution time increases, but all system functions are maintained. In addition to utilizing the technique of triple modular redundancy, the "Software Implemented Fault-tolerance" (SIFT) system also has the ability to degrade gracefully, because of replicated resources. Other systems which are capable of graceful degradation are the Basic Fault-tolerant System [11] and the Tandem transaction processing system [12] (See appendix 1).

Back-up.

The fundamental idea behind the principle of back-up systems is that one unit is operational, while one or more units wait in reserve. When the active unit has failed, a replacement takes over.

Two important back-up configurations are:

- dual-redundancy with switch-over

- pair-and-a-spare

Dual-redundancy with Switch-over. This technique is often applied in process control. A dual-redundant control system is constructed using two process controllers or commercial computers, with additional hardware and software to detect and recover from faults. The pair is connected to the process through a switching mechanism that keeps one active and the other in reserve (See fig. 9).

odd number of units, so that a majority vote can always be obtained.

Gracefully-degrading Systems. When the multiple units in a system all perform different functions, then a gracefully-degrading system can be formed. Effective load sharing across system resources, and efficient communication between the units are necessary. When a unit fails, its load is shared among the other units. Hence, execution time increases, but all system functions are maintained. In addition to utilizing the technique of triple modular redundancy, the "Software Implemented Fault-tolerance" (SIFT) system also has the ability to degrade gracefully, because of replicated resources. Other systems which are capable of graceful degradation are the Basic Fault-tolerant System [11] and the Tandem transaction processing system [12] (See appendix I).

Back-up.

The fundamental idea behind the principle of back-up systems is that one unit is operational, while one or more units wait in reserve. When the active unit has failed, a replacement takes over.

Two important back-up configurations are:

- dual-redundancy with switch-over

- pair-and-a-spare

Dual-redundancy with Switch-over. This technique is often applied in process control. A dual-redundant control system is constructed using two process controllers or commercial computers, with additional hardware and software to detect and recover from faults. The pair is connected to the process through a switching mechanism that keeps one active and the other in reserve (See fig. 9).

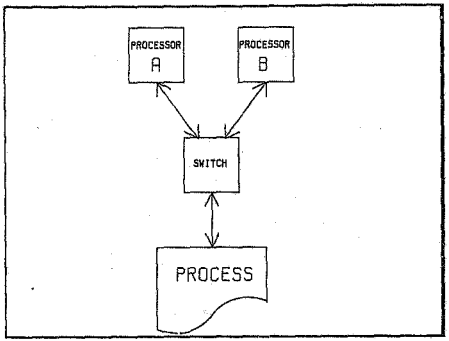


FIGURE 9 - The Dual-Redundant Configuration [4]

The active computer executes both a control program for the application, and a diagnostic program that continually checks for errors in the processing unit, memory and I/O circuits. When an error is detected, the switching mechanism transfers control to the reserve computer, which will have been passively monitoring the process. The Agusta 129 helicopter flight control system is an example of such back-up, as is the Resilient transaction processing system. These are covered in some detail in appendix 1.

For fast processes, several problems make this method unsuitable. The first problem is that errors may occur before the diagnostic program can detect that something is wrong. Secondly, the switching time at computer change-over may be too long, causing an unacceptable discontinuity in the control values. Finally, the switching mechanism could fail, causing complete loss of con-

trol. When such fast processes are to be controlled, techniques that provide fault-masking must be used.

Pair-and-a-spare. In the pair-and-a-spare configuration, four identical modules are organized as primary and shadow pairs of master and checker modules (See fig. 10).

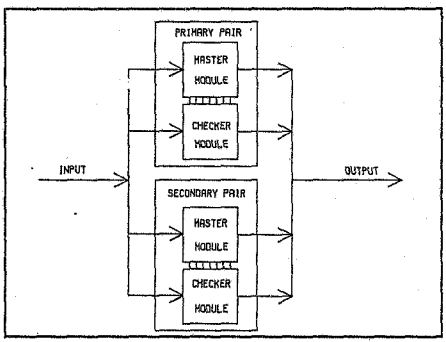


FIGURE 10 - The Quad-Modular Redundant Configuration [5]

Only the primary's master module is capable of activating the computer outputs. While the primary's master transmits data, its checker module compares external data and that presented to its disabled output drivers (See fig. 11). This technique is called "functional redundancy checking". If the primary's checker module detects an error, it initiates a procedure that disables the primary pair, and enables the shadow pair to take over the primary role. Systems which apply this technique are the Stratus transaction processing system and the Intel 432 general-purpose system (See appendix 1 for more details).

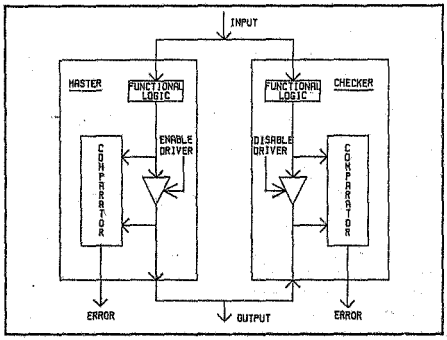


FIGURE 11 - Functional redundancy Checking [5]

Hybrid.

The essence of the hybrid system is the availability of spare units to replace those that are faulty. Spares can be provided for any system in which the faulty unit can be identified. The purpose of providing spares is to ensure that the system is returned to its original fault-tolerant state with the minimum of delay. The failed unit can then be removed for repair.

2.2.4 Software Redundancy

Software redundancy is the use of more software than is required by the application.

Software redundancy can range from the addition of small routines to perform validity checks on the data, to full replication of

all software (i.e. more than one complete software system), written by different programming teams.

There are many possible approaches to implementing redundancy in software, differing mainly in complexity. Validity checks are the simplest; in this approach, the values of key variables in calculations are monitored to pick up any deviation from the range of values that the variables may have.

Redundant software may be used to perform periodic testing of hardware, by applying algorithms to pre-defined data with known results. If erroneous results are produced by the hardware, then that particular piece of equipment is signalled as faulty.

Full replication of software may be used as a means to avoid error propagation, using voting. Identical copies of the software may be run concurrently in different processors (thereby including hardware redundancy as well), and the results compared. However, global faults such as electromagnetic interference may cause the same error to occur in all sets of the software. For this reason, the execution of the software may be staggered slightly in time, so that the same error is not induced in all copies of the program.

An expensive, but potentially reliable way of replicating software is to have different design teams each produce the programs knowing only the functional requirements of the system. This may even be taken so far as to have the teams use different programming languages. In this way, it is unlikely that the same code will be produced, and it will also be unlikely that the same programming mistakes will be made. Hence both common-mode errors and software errors have a greater probability of detection when such a system is in use.

Software may be used to perform the voting involved in hardware redundant systems. Each computer receives the same inputs, computes a result, and sends it to the other computers, receiving their answers in return. The majority voted

result is the used for output. This approach is known as SIFT [10], and will be discussed later.

Another software-redundant fault-tolerance technique, called "check-pointing", is used in some loosely-coupled systems, in which duplicated processors run the software at approximately the same time, but not with step-by-step synchronization. Software in these systems can periodically suspend normal program execution while each system compares its state with the state(s) of its companion(s) to determine if an error has occurred since the last check-point. If no error is detected, then the system saves its current state, and operation resumes. If an error is detected, then each system is "rolled back" to the previous (recorded) error-free state, and processing continues from that point. If the same error is detected at the next check-point, the failure is diagnosed as permanent.

2.2.5 Time Redundancy

Time redundancy is simply the use of more time than is needed to perform only the functions required by the application. All the fault-tolerance techniques already discussed involve the use of time redundancy:

- information-redundant systems must always perform checks to see if the data has to be corrected. Even if these checks are performed in hardware, some delay occurs. If correction is needed, then further delay is required

- hardware-redundant systems also perform correctness tests when they reach the voting stages of the each process, so that time redundancy is also evident here

- software-redundant systems can require many times the normal execution time if the entire software system is replicated. If only small diagnostic routines are used, then only a small increase in execution time will be necessary

Time redundancy can be used to aid in the determination of the nature of a fault; by repeating a calculation, it is possible to distinguish between permanent and transient faults.

2.3 Evaluation of Appropriate Fault-tolerance Techniques

Evaluation of systems is necessary to determine their suitability for a particular application. It is clear that a wide range of techniques are available for incorporation into the design of a specific fault tolerant computer system. It is naturally important to weigh up the various attributes of each approach and the trade-offs in a particular application. Generally, it is apparent that, as in all engineering, both qualitative and quantitative factors must be considered.

2.3.1 Qualitative Techniques

Qualitative comparisons describe trade-off issues and specific benefits of one technique or design over another. These are factors that can not be given numerical values, and can include:

- verifiability - the ability to determine that a system design performs the functions required of it

- testability - the ability to determine that a system is operating as it was designed to operate. Additional features are usually incorporated to make the system testable

- flexibility - the ability of a system to be used in many different application environments

Additional points which are considered in system evaluation are:

- faults that are covered by the system

- applications supported by the system

- technology - the capabilities of, and requirements for the system depend on the technology used

2.3.2 Quantitative Evaluation

Quantitative evaluation techniques derive values for [7]:

- fault coverage - the probability of detecting and handling all faults

- reliability - the probability of survival in the time span [to ,t], given that the system was operational at to

- availability - the probability that the system is available at time t

Numerous quantitative measures are taken into consideration in the above evaluations, including [13]:

- mean time to first failure
- mean time between failure
- mean down time
- availability
- computation reliability
- computation availability
- average computation to first failure
- average computation between failures

Also important in the evaluation of a technique are:

- performance - including throughput and response times
- cost - including purchasing price, maintenance cost and application engineering cost

Two widely used quantitative system evaluation techniques are [7]

- combinatorial modelling
- Markov modelling (See fig. 12)

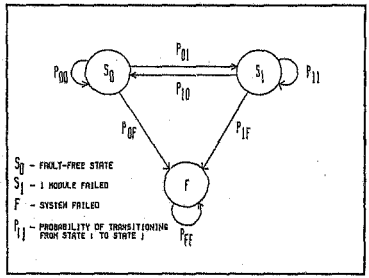


FIGURE 12 - A Markov Model [7]

2.4 Summary

The concept of fault-tolerance arises from the need to cater for the occurrence of faults in computer systems. Different applications require different aspects of the system to be made secure against the effect of faults.

The core principle, around which the sequence of fault-handling events is built, is that of redundancy. This redundancy may take the form of information redundancy, hardware redundancy, software redundancy and/or time redundancy.

In order to compare the relative merits of different fault-tolerant systems, a number of evaluation measures, both qualitative and quantitative, have been developed.

It is evident that a major portion of many fault-tolerant systems is the software which provides fault-tolerance functions. It is clearly necessary that this software be as reliable as possible, so that it does not diminish the reliability of the system as a whole. The next chapter gives a brief coverage of the techniques of good software design, or software engineering.

Chapter 3 - SOFTWARE DEVELOPMENT TECHNIQUES

3.1 Introduction

A major aspect in the production of any fault-tolerant computer system is clearly the development of reliable, well defined software. In view of this, relevant software development techniques are examined in this chapter.

The application of conventional development practices to software design has been shown [14] to lead to a 20% - 80% rule for the division of resources between definition and coding (20%) and testing and maintenance (80%). The lack of appropriate software design and development tools may lead to unstructured, poorly documented, and error-prone programs which are difficult to understand and expensive to maintain [14]. The increasing complexity and extent of applications of computers has reinforced the need for improved software development techniques.

However, with the application of modern software engineering practices, reductions in software costs, increases in programmers' productivity and reductions in error frequency of between 25% and 75% have been observed. Experiments indicate that the application of more systematic management, design and development techniques may lead to a 40% - 20% - 40% rule for the division of resources between definition/design, coding, and testing/maintenance respectively [14].

3.2 Structured Design

3.2.1 Design Methodologies

As applied to programming, design methodology consists of [15]

- establishing the definition of the problem
- specifying the data objects the software must manipulate

- specifying the operations that correspond to the manipulation of the data objects

- specifying the programs which must operate on the defined data objects

In order to control the complexity of the development sequence, it is necessary that specifications are initially represented in an abstract form, leading to the adoption of formal specification systems.

3.2.2 Design Principles - Overview

The four major design principles are [15], [14]:

Specification - identification of all the functions that the design must provide. Specifications formally define the functions and properties that a designed system must have. Formalized specifications are derived from the external requirements of the system.

Complexity decomposition - a structured organization of intellectually manageable steps or components of the design. The structure of an entity is given when the relationships between its components have been identified. The most widely accepted notion of modern programming techniques is the introduction of good structure into program and data design.

Guided design. A constrained and controlled process of construction of the design. The construction model consists of three rules which govern the development process:

- A program cannot be functionally specified until all its requirements are known

- The program's algorithm cannot be derived until the functional specifications are known

The environment for programs and objects form conditions which in turn may generate requirements through data type specifications to be satisfied by a lower level development step

Proof of correctness. Ideally, a proof should be possible for every program design and every data representation to ensure the design is consistent with its specifications. At present however, it is accepted that this goal is not practically attainable. Two types of proof are needed

- Proof of the program text
- Proof of the data representation

Proof is considerably aided by good documentation structure and the use of formalized specifications throughout the design sequence [15].

3.3 Structured Programming

It is clear that the easiest systems to maintain are those built up from manageably small modules, each of which is, as far as possible, independent of the others. This allows them to be taken out of the system, changed, and put back in the system without affecting the rest of the system.

In such a system, each module has its own job, which it performs only when given orders from above; it communicates only with its invoking module and with its invoked modules, to which it will, in turn, issue orders.

Coupling.

A good design therefore has the least possible coupling between modules. Three types of coupling have been defined [16]:

Data coupling. Data is passed as part of the invocation of the module and as part of returning control to the invoker. The coupling is improved if as few data exchanges as possible are used. This coupling has been found to be the best type.

Control coupling. In this form of coupling, status reports are passed between the invoking module and the called module, causing changes in the control pattern. This type of exchange should be kept to a minimum, for ease of understanding of program flow, and hence easier maintenance.

External/content/pathological coupling. This coupling arises when the execution of a module depends extensively on another module. Such coupling should be avoided, because of the confusion it can create in understanding of the program.

Cohesion.

A highly cohesive module, whose parts all contribute to a single function, is not likely to need much coupling to other modules. Six types of cohesion have been identified. From the worst to the best, they are [16]

Coincidental cohesion. The elements of the module cannot be seen as achieving any definable function.

Logical cohesion. Several similar functions are combined into one module.

Temporal cohesion. A variety of functions, which are executed at the same time, are combined into one module.

Procedural cohesion. Each chunk or procedure of a flowchart has been built into the same module.

Communicational cohesion. All functions in the module operate on the same data stream.

Functional cohesion. The module carries out one identifiable function.

Structured programming involves coding programs using a limited number of control structures to form highly cohesive units of code that are easily readable, and therefore more easily tested, maintained and modified than conventional programs.

Many tools which aid in program development are available. The prime tools are:

- The structure principle
- Specification extraction
- Tree structure diagrams
- Pseudo-code

3.3.1 Structure Theorem and Conventions [17]

The structure theorem states that any proper program (a program with one entry and one exit) is equivalent to a program that contains as control structures only

- sequences of two or more operations
- conditional branches to one of two operations (IF condition x THEN action a ELSE action b)
- repetition of an operation while a condition is true (DO WHILE condition x)

A large and complex program may be developed by the appropriate nesting of these three basic structures within each other. The logic flow of such a program always proceeds from the beginning to the end without arbitrary branching.

Two useful extensions to the three structures are

- DO UNTIL condition x

- DO CASE condition x

Several conventions are included as a supporting part of structured programming. For example, strict attention is paid to the indentation of the control structures on the printed page, so that logical relationships in the coding correspond to the physical position on the listing. Code is segmented into reasonable amounts (normally one segment or function per page). Segmentation continues down through the entire coding process.

The use of structured programming should provide many benefits, including fewer errors in the programming process, programs that are nearly self-documenting, and code that can be more easily read, modified and maintained.

3.3.2 Specification Extraction

A critical area in the design process is the establishment of the correct specification for the system. All design stems from this specification, so any errors or omissions will be propagated from it, into the final system. Two tools aid in the correct specification of the system:

- decision trees

- decision diagrams

Decision Trees. [16]

Decision trees are used as a tool to extract the correct decision logic from ambiguous specifications.

The technique is to identify conditions, actions, "unless", "however", "but",..... structures, greater than/less than ambiguities, and/or ambiguities, and undefined adjectives. This establishes areas which must be cleared up by the intended user of the system. Once clarification has been completed, a revised specification narrative is produced, and a decision tree is drawn up (See fig. 13).

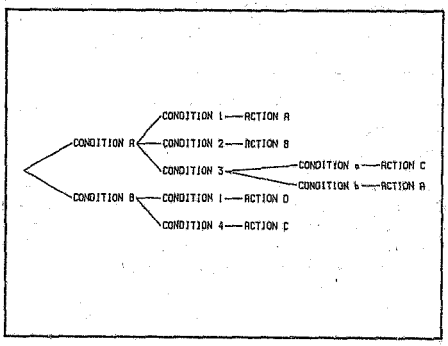


FIGURE 13 - A Decision Tree [16]

Decision Diagrams. [16]

Decision diagrams (See fig. 14) are exhaustive tables of all possibilities for every condition. They are used to specify exactly the action which must be taken for each combination of conditions.

C1-condition 1	Y	Y	Y	Y	N	N	N	N	there may be more than two
C2-condition 2	Y	Y	N	N	Y	Y	N	N	distinct values for each
C3-condition 3	Y	N	Y	N	Y	N	Y	N	condition
A1-action 1	X		X			X	X		action not taken
A2-action 2		X		X	X			X	action taken

Fig. 3.4 - A Decision Diagram [16]

3.3.4 Tree Structure Diagrams [18]

A useful graphic representation of a structured system is the tree structure diagram. A graphic representation allows easier visualization of the system, enabling the designer to more readily see improvements to the structure. Using this system, nodes on the diagram are shown as rectangles, such as those which follow:

object A

object B or object C

repeated object D

the null object

C1-condition 1	Y	Y	Y	Y	N	N	N	N
C2-condition 2	Y	Y	N	N	Y	Y	N	N
C3-condition 3	Y	N	Y	N	Y	N	Y	N
A1-action 1	X		X			X	X	
A2-action 2		X		X	X			X

there may be more than two distinct values for each condition

action not taken

action taken

FIGURE 14 - A Decision Diagram [16]

3.3.4 Tree Structure Diagrams [18]

A useful graphic representation of a structured system is the tree structure diagram. A graphic representation allows easier visualization of the system, enabling the designer to more readily see improvements to the structure. Using this system, nodes on the diagram are shown as rectangles, such as those which follow:

object A

object B or object C

repeated object D

the null object

Nodes are connected in a tree structure which indicates the relationship between nodes. An example is shown in figure 15.

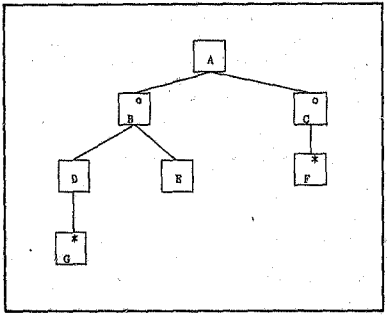


FIGURE 15 - A Tree Structure Diagram [18]

3.3.5 Pseudo-code

The primary purpose of pseudo-code is to enable an individual to express his thoughts in a form that uses native language prose, but expresses the control flow of the program in an unambiguous manner. Pseudo-code acts as a form of program documentation which is easy to maintain and not excessively time-consuming to produce.

"Structured words", such as IF, DO UNTIL etc., and indentation rules, are used to show control dependency. Natural language phrases are used to express thoughts [17].

3.4 Data Flow

3.4.1 Introduction

Data flow is the technique of connecting the functions of a system only by flows of the data within the system. Functions connected by data flow are not dependent on adjacent functions [19]. An independent job step can execute as long as its input data is available and it can dispose of its output data. This ensures that the program is easy to maintain.

(See appendix 2 for a more detailed description)

3.4.2 The use of Data Flow Techniques

The basic tool for utilizing data flow is the data flow diagram. A standard set of symbols is used to represent the flow of data through the elements of a system. The set includes:

- functions - processes which operate on the data
- flows of data
- stores of data
- external sources or sinks of data
- off-page connectors

Data dictionaries are used to provide standard descriptions of elements in the system.

To get from a data flow diagram to an hierarchical structure, one starts with the rawest form of input and traces it through the data flow until the point is reached where it can no longer be said to be input. Likewise, the output is traced back into the system until it can no longer be thought of as output. The middle piece of the system forms the transformation section.

Tree structure diagrams are used to represent the hierarchical system.

3.5 Top Down Design

Traditionally, top-down development involved the ordering of development, in each design phase, from the highest level to the lowest level, as shown in figure 16.

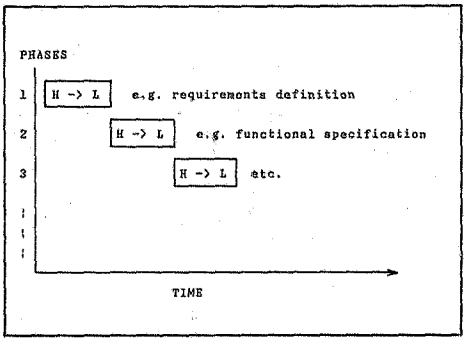


FIGURE 16 - Traditional Top-Down Development [20]

Recently, a modified approach to top-down system development has been proposed [20]. The new approach essentially combines top-down and bottom-up development.

Using the new approach, top-down design is defined as the technique of producing a crude skeleton version of a system, then adding and testing more complexity, piece by piece (See fig. 17) [21].

Course and fine versions of the system are developed in turn. This allows phases to run in parallel, since design teams do not usually concern themselves with phases in the development other than the one in which they specialize.

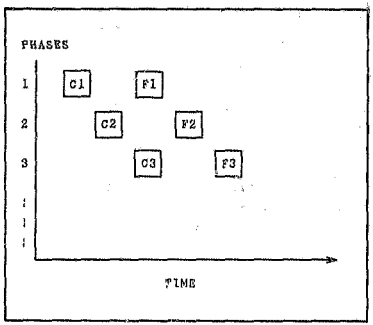


FIGURE 17 - Revised Top-Down Development [20]

With revised top-down development, the highest level of a system is coded and tested first. Since this unit will normally invoke lower level units, dummy code must be substituted temporarily for

them. Once the higher levels have been completed, work proceeds down the hierarchy until all coding is done.

The major advantage in top-down development is that it avoids the problem of interfacing many small modules. Also, it allows users to see reduced versions of the system so that they can offer comments at an early stage.

The quality of a system produced in this manner should be increased through earlier detection and elimination of design problems and coding errors [17].

3.6 Verification and Validation

The main objectives of the verification and validation processes are the identification and solving of software problems and high-risk issues as early in the software life cycle as possible.

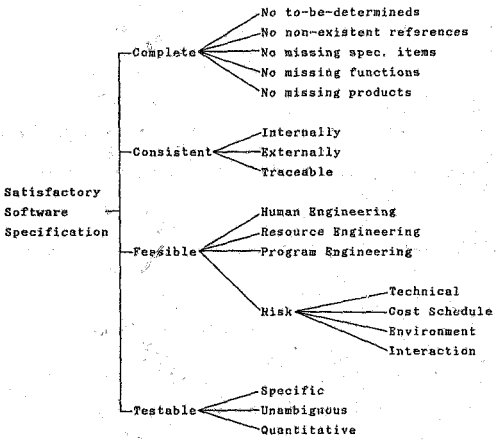
3.6.1 Definitions

Verification - "The process of determining whether or not the products of a given phase of the software development cycle fulfil the requirements established in the previous phase" [22].

Validation - "The process of evaluating software at the end of the software development process to ensure compliance with software requirements" [22].

3.6.2 Verification and Validation Criteria

The criteria are shown in figure 18, overleaf



NOTES:

- Consistency implies that a specification's provisions do not conflict with one another or with governing specifications and objectives. The specifications should be traceable so that misinterpretations and embellishments are avoided.

- If high-risk issues are not identified, then there is a high likelihood of disaster.

FIGURE 18 - Satisfactory Software Specification [22]

3.6.3 Verification and Validation Techniques [22]

Although manual techniques for verification and validation prevail, there is increasing development of automatic tools which improve the speed, reliability and consistency of the checking processes.

Manual techniques. These techniques may take the form of

- reading by someone other than the originator
- manual cross-referencing
- interviews with the originator of the specification
- checklists
- manual modelling in defined environments
- mathematical proofs

Automatic techniques. These may take the form of

- automated cross-referencing
- automated modelling
- prototypes

3.7 Debugging [23]

It is almost certain that, even after the most rigourous software development process, bugs will be found in the completed system.

The debugging process consists, in essence, of the following:

- describing the error

- (*) gathering data about the program's behaviour

- hypothesizing about the cause of the error, and taking steps to remedy it

- testing the hypothesis

- if the hypothesis did not work, then the processes is repeated from step (*)

(See appendix 3 for more details).

3.8 Summary

The use of modern software design techniques improves the quality of the software which is produced.

A number of techniques may be applied to improve the software design process. The most useful technique is that of structured design, as adapted for programming.

The steps involved while applying the technique of structured programming consist of:

- complete and accurate specification of the required software

- decomposition of the specification into manageable steps

- controlled construction of the software

- proof of the correctness of the software

At the end of each stage in the development, the results of the stage are verified (shown to follow from the results of the previous stage), and at the end of the design process, the software is validated (shown to comply with the software requirements).

Having now covered both an investigation into fault-tolerance, and a study of software development, the following chapter describes the experimental system produced to aid in the study of fault-tolerance techniques.

Chapter 4 - SYSTEM DESCRIPTION

4.1 Introduction

One of the prime objectives of this project was to produce a flexible experimental system which could be used to study various fault-tolerance techniques. In order to demonstrate its operational effectiveness, the system was to perform elementary real-time control of a servo system.

The possibility of constructing a system which utilized hardware to perform fault-tolerance functions was ruled out, for two main reasons

- Special processor boards would have to be developed, or additional boards would have to be designed to provide the mechanism for fault tolerance. This would require excessive additional effort, and not take advantage of the hardware available from commercial sources.

- It was clear that thoroughly tested commercial processor boards would be superior to any rapidly designed, yet complex, new hardware.

For these reasons, and for the obvious reason of flexibility, it was decided that the fault-tolerance functions would be implemented in software.

The path taken in the development of the system was to select a promising fault-tolerant architecture, and to use that as a basis from which other fault-tolerant configurations could be built through minor alterations to the software.

It was decided that the triple modular configuration was appropriate, mainly in view of several characteristics displayed by the system (as described in chapter 2) [24].

However, the software was designed and coded in a sufficiently modular form for small modifications to produce other system configurations.

The designed system provides a number of facilities for the tolerance of faults (See fig. 19)

- Task I/O voting
- Self and mutual testing
- Time staggered operation
- Device operation validation
- Watchdog timing
- Fault handling

These will be discussed in detail in the following sections.

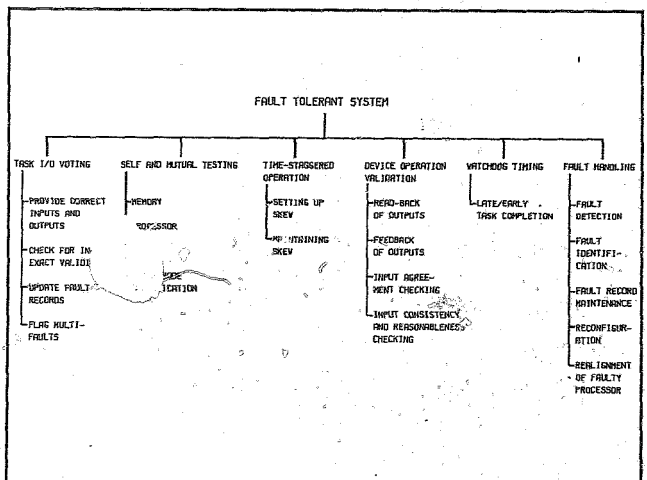


FIGURE 19 - System Description

4.2 Detailed System Description

The physical part of the experimental system is located in a 12-card Multibus-compatible rack, although provision is made for the nodes ultimately to be in separate racks, so improving reliability. This facility is accomplished by using a "model" of the MIL-STD-1553B bus communication standard for inter-node communication. The model is implemented in the experimental system using an additional processor and common memory. Hence, for the processing nodes to be separated, bus interface controllers need to be provided, and the software slightly modified.

The processing nodes in the experimental system each contain an elementary operating system which controls the execution of tasks and the fault-tolerance facilities. The operating system ensures that tasks are run at the correct time, that they run to completion, and that test tasks are invoked when there is sufficient time.

In addition to task control, the operating system provides a number of optional routines that the application tasks can use to perform fault-tolerant input, output, and memory access. Routines for scheduling and descheduling tasks are available.

These additional routines can be used as required by the application design. This provides the facility for choosing an acceptable execution time overhead, balanced against gains in reliability.

Each aspect of the system is introduced below.

4.2.1 Task I/O Voting

Task I/O voting is the most important of the fault tolerance measures, and since the system is to be used in a critical control environment, it must provide correct outputs at all times. In order to establish the validity of the outputs, it is necessary to have not only a means of comparison between proposed out-

puts, but also a means for determining the correct output. Hence three versions of critical outputs are produced, and by ensuring that at least the majority of the processors agree on an output value, the chance of an incorrect output clearly becomes very low.

When a task requires an input, it requests the correct value from the operating system. The operating system routine ascertains which type of data (input from external devices, input from previous calculations, or permanently stored data) has been requested, and executes an appropriate subroutine to provide the correct value. In the case of sensor inputs, the subroutine may be required to perform sensor operation checks before it returns an input value. These take the form indicated in the device operation validation section (4.2.4). If the required data is in RAM, then the subroutine writes the correct data into any location found to have the incorrect value. A two-out-of-three vote is applied whenever there are three values available.

Similarly, when a task has to output a value, it requests the operating system to perform the data validity checks and perform the actual output operation (See fig. 20, overleaf).

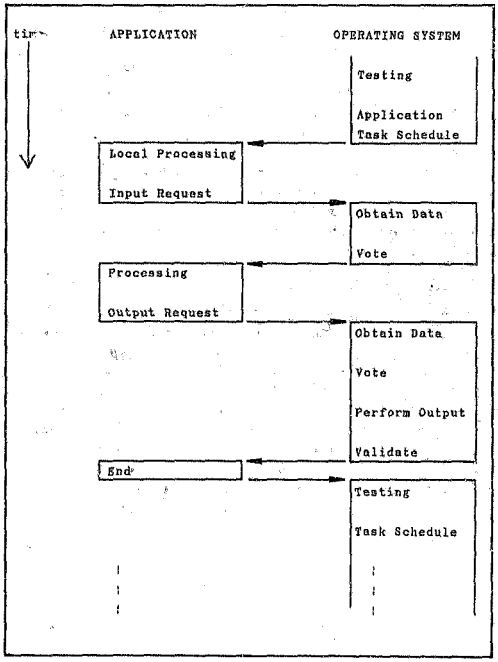


FIGURE 20 - Task I/O Voting

The voting operation provides three important pieces of information. Firstly, an error is signalled soon after its occurrence, allowing the operating system to begin appropriate action. Secondly, the erroneous information is marked as such, and can be regarded with suspicion if it has to be used for further calculations. Finally, the faulty node is identified, and can be ignored until it has been shown to be usable by the operating system.

There is a small possibility that two or all three of the processors will fail, and produce incorrect output values. When this happens, the voting mechanism indicates that it is incapable of resolving the dispute, and the operating system attempts to validate the system before it allows normal operation to continue.

There is also a possibility that results submitted for voting have an accuracy tolerance, that is to say, they may be slightly, but acceptably, incorrect. Such a situation may arise, for instance, when a task running in a node gets its input values from more than one sensor, causing them to be a little different. In such cases, it is necessary that the voting mechanism flags a fault only when the three submitted values are unacceptably different from each other. For this purpose, each data element has a tolerance value associated with it. The tolerance value is considered to be zero in the default case when none is specified, so that such a value need only be assigned to those data elements where inexactness is tolerable.

4.2.2 Self and Mutual Testing

Self- and mutual-testing of the processors helps to identify a faulty processor when a dispute arises. Such testing is also used to forestall the occurrence of a dispute by indicating a fault before a task is undertaken by the processor.

The testing is done in two circumstances. Firstly, testing is done whenever the processors are idle. This can provide advance warning of pending execution errors. Secondly, testing is undertaken when the operating system decides that it is necessary. Such an operation will occur when the operating system is to validate an input, or when a fault is detected, but the operating system is unsure of its originating processor, or wishes to make a more detailed diagnosis of the fault.

A number of subsystems are tested, in particular, the memory, the processor, the I/O equipment and the inter-node communication equipment, as well as any specialized equipment the node has. Some of this testing is written in machine code to maintain tight control over the system resources.

Self-testing is intended to find faults in the physical components of the computer system (See appendix 4). Algorithm faults and programming mistakes must be removed using conventional debugging techniques.

Error and Fault Detection.

It is clear that tests may be devised to detect virtually any error or fault. The question of which tests provide a significant improvement in reliability, against the resource usage required, has to be carefully considered before including any of them.

Various criteria establish the usefulness of a particular test, such as:

i - the probability that the type of malfunction which the test is designed to detect will actually occur, and the probability of its detection and correction using a particular technique

ii - the probable damage that could be caused by the malfunction

iii - the cost of additional storage and increased computing power requirements to include the test in the system

Testing in the Experimental System

A form of hybrid testing is utilized in the system discussed here. The system is functionally partitioned, and exercised using a set of pseudo-random numbers as data. If time permits (i.e. if the processor workload is low), this method tends toward the pseudo-exhaustive method, since the probability that all combinations of data and control have been tried tends to 1.

Processor Testing. The first items to be tested are the core instructions of the processor, namely

- memory to register transfer
- register to register comparison
- conditional branch

Next, instruction tests are performed, testing all possible processor instructions. Two methods are possible

- recomputation of a result by a different method, and comparison of the two results. Pseudo-random data is used.
- computation using known data, with pre-computed answers available for comparison

Memory Testing. Once it has been established that the processor is functioning correctly, testing of the memory sub-system proceeds.

The memory testing procedure is as follows:

- A block of memory is transferred to a temporary storage location. The copy is checked against the original to see if any error has occurred in the transfer.

- Pseudo-random numbers are generated and written into the block of memory to be tested, as well as another temporary storage area. These data are compared. The test can show up pattern dependent faults.

- A sequence of "sliding ones" is written into the block and read back. This technique is included in addition to the use of random numbers because it shows up both "stuck at" faults and cross-coupling faults, which are common types of memory fault.

- The original data is copied back to the memory block, and this is again checked against the duplicate.

Input and Output Device Testing. It was necessary to write specific test programs for the I/O facilities attached to the system.

Two situations are possible when testing input and output:

- Testing may be allowed to affect the devices attached to the output facility.

This situation is normally applicable only when the system is not in use, i.e. before it begins to execute application tasks. Tests which affect the attached devices are therefore normally run as pre-application system acceptance tests. The tests are able to fully exercise the output and input capabilities of the I/O facility, performing both readback and feedback tests if possible.

- Testing is not allowed to affect the devices attached to the I/O facility.

In this case, only limited testing is possible, such as writing to and reading from control registers without activating them, and testing input devices. It is also possible to produce small output variations if the resolution of the system is such that output is possible without being detrimental to the performance of the plant.

Communication System Testing. Special test programs were needed to test the communication system. Once again, the system may or may not be permitted to affect the output of the communications interface board.

It is possible to arrange the establishment of a communications link with another node for testing purposes only. In this case, the testing consists of passing known messages back and forth along the link, as well as the use of pseudo-random messages. All capabilities of the 1553B facility are tested.

Tests of other peripheral equipment need to be developed as appropriate for each additional item.

Mutual Testing. For the purposes of mutual testing, a node instructs another node as to the test(s) it must perform, and monitors the results. This provides added protection against the possible misinterpretation of test results due to faults in the node.

Application-Dependent Tests. Tests which are built into the application programs may be included, at the discretion of the programmer. Such tests can include

- recomputation of data by the same, an inverse or a different process, and comparison of the results

- tests to see if results satisfy mathematical or physical criteria

- checks based on estimates of behaviour

- special tests for a particular process or machine

Acceptance Tests. Acceptance tests for the system consist of a more comprehensive set of tests, plus the more exhaustive use of the standard tests. The acceptance tests are run as part of the preparation of the system for use.

Test Control.

The test control routines have two functions

- scheduling of tests

- maintenance of test records

Under no-error conditions, a standard routine of testing is followed and test records are updated.

When an error is detected, either by the self-test routines or by other methods, an attempt is made to establish the cause of the error. This is done by a systematic narrowing-down process, using the self-tests available. An error record, plus a fault record where appropriate, are generated.

4.2.3 Time Staggered Operation

Because of the noisy environment in which the system may operate, there is a high possibility of transient faults caused by interference such as power spikes and electromagnetic noise. To minimize the effects of such interference, time-staggered execution of tasks in the three processors is used. Although the processors are executing the same task at any time, this staggering ensures that code instructions are not executed simultaneously, and

therefore cannot be affected in the same way by common-mode interference. Errors in the results of all the tasks may occur, but they will be different, and therefore detectable.

Setting up of skew is necessary in two circumstances, namely at system power-up, and when a processor must be brought back into operation after fault recovery.

Since voting is done in software, there is no necessity for the processors to be brought into synchronization for this purpose. Results for voting are accessed by another processor, via the asynchronous communication system, when it is ready to do so.

There are two ways of maintaining time-staggered operation:

Staggered execution.

When using staggered execution (See fig. 21), all system clocks are synchronized to the same "clock-time", and each task begins at a different clock-time. This method requires that all task scheduling operating system calls are intercepted and modified to the new execution time, particular to each node, or that the task scheduling routine is rewritten to include a different correction factor for each node. Alternatively, all task scheduling calls must be written to include the different execution times. This implies different software for different nodes.

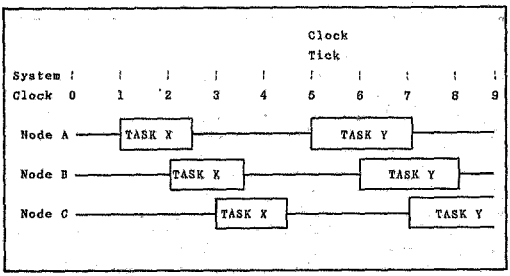


FIGURE 21 - Staggered Execution of Tasks

Staggered Clocks.

In the staggered clock method (See fig. 22), all system clocks are synchronized to the same clock-time, and are then staggered by the required amount. This means that the clock-time for each node is different.

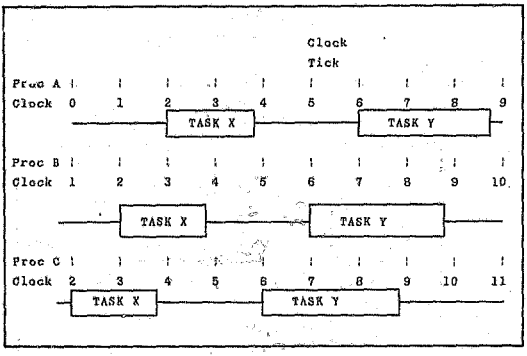


FIGURE 22 - Staggered Clocks

In this system, it is not necessary to intercept task scheduling calls or to rewrite the scheduling routines, because automatic staggering of execution times occurs.

This means that software can be exactly duplicated in all nodes. The node identification determines its clock-stagger position.

4.2.4 Device Operation Validation

Routines are provided to test the state of the output devices, as well as monitor the state of input devices. The output devices are tested by reading back the outputs sent to them to see if there is any discrepancy.

Where input devices are replicated, the values returned by them are compared, and a note made of any discrepancies.

Where replication is not present, the values returned are checked for reasonableness and consistency.

Output Device Validation.

An output channel is defined, for the purposes of this report, according to figure 23.

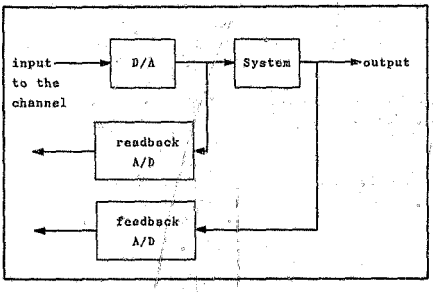


FIGURE 23 - An Output Channel

Either or both of the A/D converters may be absent from the output channel.

Two types of device operation validation are possible:

Readback. By reading back the output applied to the device by the I/O board, the system determines whether an output circuit fault has occurred. This may be either due to a fault on the I/O board, or a fault at the device inputs. In either case, the output channel has failed. It is possible, by examining the value of the incorrect output, to identify the type of fault - open circuit, short circuit or in between, but this is immaterial as the device is no longer usable in any case.

Feedback. If the readback signal is correct, then it is still possible for the device to have failed in some area other than the input circuit (such as a mechanical section). In this case, the feedback signal is different from that which can be predicted from the device characteristics.

Because the feedback signal to be expected is dependent on the nature of the application, it is up to the application program to provide checks on the feedback signal. An expected value is provided to the operating system whenever an output is requested, enabling it to decide whether an error has occurred.

If the feedback sensor itself has failed, producing invalid readings, this registers as an output channel failure because no control is possible.

An I/O data table is kept, informing the system of the configuration of each output channel. The table is consulted whenever output is to be performed.

If the device is such that readback of output values is not useful, then the table informs the operating system accordingly, and no readback is made. If readback is provided, then read-back testing is done whenever an output is sent to the device. Tests utilizing readback can also be made when no other tasks are being executed, or when specifically required by the operating system.

Similarly, if feedback is not present, then the operating system does not expect feedback error values from the application program.

Input Device Validation.

An input channel is defined in figure 24.

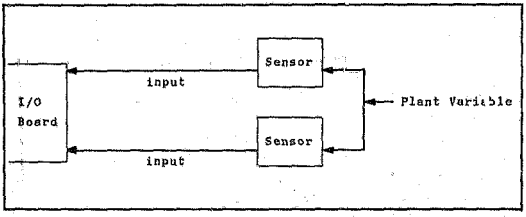


FIGURE 24 - An Input Channel

If a particular sensor is not replicated, then it is possible to detect its failure by checking its input values for reasonableness and consistency.

If a sensor is duplicated, then a fault is easier to detect, and if a sensor is triplicated, then the faulty sensor is easy to identify. In a dual input system, once a fault has been detected, the faulty sensor must be pin-pointed, or no useful data can be obtained about that plant variable. This identification is accomplished using the reasonableness and consistency tests such as those applied to single sensors.

It should be noted that the reasonableness and consistency tests may fail if a sensor input value is reasonable even though it is incorrect. In such a case, it is impossible to detect a single sensor failure or to identify the faulty sensor in a dual-sensor system.

4.2.5 Watchdog Timing

When a processor executes an incorrect instruction, or uses incorrect data, three things may happen. Firstly, no incorrect results occur (this is an unlikely option). Secondly, incorrect results occur, but the processor will exit the task as normal. Thirdly, the processor may enter an endless loop, or take an unpredictable direction of execution.

To detect such faults, watchdog timing is used. For this purpose, each processor provides a task-complete signal whenever it has completed the assigned task. In the first and second cases, the processor concludes its calculations, but at a time which is likely to be completely different to that of its counterparts. The time of occurrence of that processor's task-complete signal will therefore be significantly far from the signals of the other processors. In the third case, a task-complete signal may never be received from the faulty processor. An acceptable difference is defined, within which time all processors must have produced results, or an error is indicated.

The procedure is as shown in figure 25.

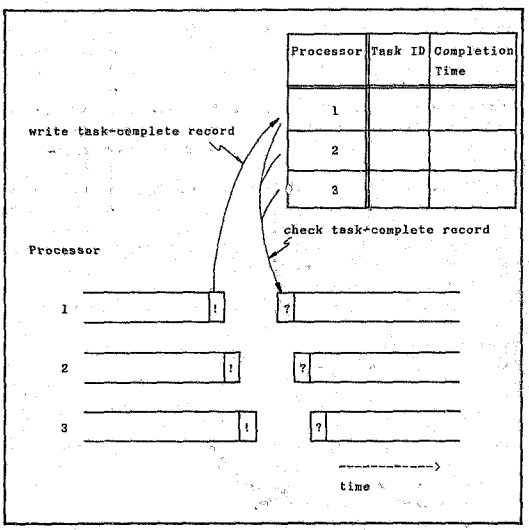


FIGURE 25 - Task Complete Records

After a task is complete, each operating system places a task-complete time in a task-completion table. Each operating system then compares the task-complete times to see if they are sufficiently close. Otherwise, an error is indicated. Furthermore, the task-completion allows the extra check that the processors were executing the correct task.

The primary use for the watchdog mechanism is to detect program flow errors, i.e. the situation where the program counter/instruction pointer has somehow obtained the incorrect instruction address. This usually leads to the processor never completing the current task, and also to its producing invalid data and becoming unusable. It could clearly never diagnose its own problems, so it is necessary for the other processors to take charge.

A secondary use for the mechanism is to detect errors that are missed by the other mechanisms due to the value of the error being acceptable to them. Such a situation can arise when the processor mistakenly arrives at the correct answer after it has executed an incorrect instruction or used incorrect data. The overtime or undertime error exhibited by the task can then show that something is faulty.

Watchdog timing thus provides a means of detecting errors as well as identifying the erring processor node.

4.2.6 Fault Handling

Faults are detected by the self testing facility as well as by the detection of an error by means of read-back of outputs, watchdog timing and voting. All error data are recorded in a data base which holds the historical fault manifestations and any other data that would help in the fault localization procedure.

Identification of the faulty node after the detection of an error is, in most cases, provided by the detection mechanism. Where this fails, the operating system instructs the nodes to perform self- and mutual-testing to attempt to identify the faulty node or nodes. This forms part of the diagnostic section of fault handling. Even where the fault has been localized by the error detection mechanisms, it is sometimes useful for further diagnostics to be performed so that system repairs can be made as soon as possible, and so that full records of failures are available for the system designer for design improvement. If

fault identification fails, then the system as a whole has failed, and must be restarted, after suitable repairs have been made.

Once a fault has been detected, it is necessary that all nodes be informed that an error has occurred, which data is suspect, and which processor or processors may have caused the error. This data is also maintained in the error-reporting data base. This enables healthy processors to continue without relying on possibly erroneous data.

Since many of the errors are likely to be caused by transient faults, it is not advisable to shut down a processor as soon as it errs. Instead, the system temporarily ignores that processor, while the operating system restores its internal state to one that is consistent with that of the healthy processors. Once this has been accomplished, and the processor brought back into loose skew synchronization, it may begin to execute application tasks once more, and provide useful input to the voters. A "black mark" is added to the record of the processor, so that a frequently erring node can be recognized, and shut down after an unacceptably high number of errors.

Once a processor has been taken out of the system, it is instructed to continue self testing to try and establish the cause of its errors. If the node continues to err this fact is recognized, and it remains out of the system. If, however, the node seems fault-free, then it may (in response to a status request) inform the system of its readiness to try again. The good processors aid the failed processor to recover by periodically resetting its state to one consistent with the functional system. Then it can perform the same calculations and see if they are correct. At no time is the node allowed to communicate spontaneously until it has been recognized as fit to do so. It may only communicate when specifically requested to do so by a running processor.

4.3 Summary

The system provides a number of facilities for the tolerance of faults:

- Task I/O voting, in which both input and output to and from memory, and input and output devices are compared before they are used by the application

- Self and mutual testing, where nodes test themselves and the other nodes in the system

- Time-staggered operation, which ensures that common-mode faults do not affect all processors in the same way

- Device operation validation, which ensures that devices are operating correctly

- Watchdog timing, which keeps track of the execution of all tasks, to ensure that there are no incorrect task executions

- Fault handling, to provide detection, identification and isolation of faults

Following on from the functional description of the system, the next chapter describes the hardware and the software.

Chapter 5 - SYSTEM DESIGN

5.1 Introduction

As was discussed previously, the composition of much of the experimental system was defined by the available equipment and tools. This meant that hardware development was kept to a minimum, consisting essentially of integration of available commercial equipment, and the construction of an input and output console for use with the servo system. The bulk of the design of the system therefore consisted of software development. The design process followed modern software engineering practices as far as possible, so that reliable software could be produced. The design sequence is covered section by section in this chapter, to illustrate the logical progression of the design steps.

5.2 Hardware Characteristics

The hardware consists of:

- Four Intel 80186-based single-board computers, used as the processing elements of the system, and for system monitoring

- a Feedback MS150 modular servo system, which served as the application system for control, as part of a demonstration of the use of the computer system

- a Data Translation DT732 analog input and output card, used to interface the fault-tolerant structure to the servo system

- an Intel 428 memory card for inter-node communication and system operation records

- a 12-card Multibus rack to house the equipment

Two terminals were available for node monitoring, and a personal computer with an appropriate loading facility was available for downloading of software to the processor nodes.

(For technical information on the more important components, see appendix 5).

5.3 Software Requirements

The software is required to provide eight major functions, namely

- Task I/O handling
- Self and mutual testing of nodes
- Time-staggered operation
- Watchdog timing
- Error handling
- Task control
- Inter-node communication
- System initialization

5.3.1 Task I/O Handling

I/O Device Checks.

To ensure that input and output devices are not used when they are faulty, it is important that regular checks on the I/O devices are made, especially when they are about to be used.

For output devices, readback and feedback checking of output channels must be supported, while for input devices, input data reasonableness and consistency checks must be provided.

I/O Request Servicing.

In order for the application system to make use of the fault-tolerance facilities, I/O request servicing must be provided. This will enable application modules to call operating system routines which perform input or output (as required) in a fault-tolerant manner.

To provide complete fault-tolerance facilities for the application programs, all types of task I/O, including memory modifications (i.e. RAM output) and memory data input, as well as other peripheral I/O types must be catered for in the I/O request servicing routines.

I/O Records.

Throughout the operating system, records of operations must be maintained, for two main reasons:

- to enable the operating system to perform fault diagnosis when necessary, and

- to allow system monitoring at various development phases and for maintenance and repair

In addition to the normal records, the I/O request servicing section of the system must maintain a list of the I/O devices together with their operational status and notes about peculiarities of the devices (such as replication of input devices and dual inputs to output devices).

5.3.2 Self and Mutual Testing

Both self testing and mutual testing must be included because of the possibility that the software in a node is corrupted. This corruption, when affecting the self-test software, may cause the node to erroneously report itself fault-free. The testing of a node by another node will detect this situation.

Test Descriptions.

All parts of the system should be exercised by the test programs, including the

- memory components
- processor subsystem
- I/O equipment
- inter-node communication equipment

and any other special equipment.

Test Records.

As before, error and fault records must be updated according to the results of the tests.

Also, records must be kept on the condition of devices, including whether or not they are operational.

5.3.3 Time-staggered Operation

In order to overcome the effects of common-mode faults, time-staggered operation must be used.

5.3.4 Watchdog Timing

After each task, the node must make available to other nodes the task number (identification) and completion time of the last task. This data will form the task-complete record for the task.

Thereafter, the node must check the total task-complete record set (from all nodes) to see if there are any discrepancies. Both task numbers and task complete-times must be checked.

5.3.5 Error Handling

The system must provide the facility to preserve system operation when an error has occurred. There should be the means to restore the system to its fault-free state wherever possible.

In order to facilitate fault isolation and to provide data for system repairs and improvement, it is necessary that the system perform fault diagnosis whenever it is suspected that a fault exists (namely when a node is isolated for testing).

5.3.6 Task Control

Task control is a most important part of the operating system. Task control must take care of the scheduling and descheduling of tasks, both application tasks and operating system tasks.

Routines must be provided by which application tasks are able to schedule and deschedule other application tasks.

A record must be made on the execution of each task. This will aid system testing, maintenance, repair and improvement.

5.3.7 Inter-node Communication

As was pointed out earlier, the inter-node communication carried out by modelling an actual system. Communication is imitated using common memory, but the format of the MIL-STD-1553B system protocol must be retained.

To be able to validate the communication system, it must be possible to establish test links.

Although modelling of the bus is not part of the experimental system, it is necessary for demonstration purposes. Procedures must exist which will take care of flags, and buffers, and perform other operations that would be handled by the communication board.

Bus activity should be recorded by the bus model procedures for the purposes of testing, maintenance and design improvement.

5.3.8 System Initialization

This suite of routines must be executed at power-up. The routines must establish the working environment for the operating system by initializing devices and performing system confidence tests.

(See appendix 7 for a more detailed description of the requirements of the software)

5.4 Functional Specification of the Software

The functional specification of the software (see appendix 8) was derived directly from the above software requirements.

A diagram of the software is shown in figure 26.

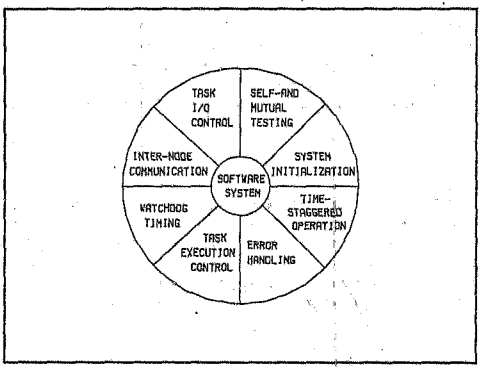


FIGURE 26 - The Software System

5.4.1 Task Input and Output

I/O Device Checks.

I/O device checks consist of readback and feedback checking, reasonableness and consistency checking, and error signalling.

- Readback is applied to outputs to ensure that their values are correct, while feedback checking is applied to ensure that the required effect is accomplished.

- Reasonableness and consistency checking are applied to inputs to ensure that the values which they return are within the ex-

pected range, and are not changed, from reading to reading, more than is possible for the device.

- Error signalling is accomplished in two parts. First, the detection of an error results in an error record being created, which alerts the operating system. Secondly, a status value is returned to the application program so that application-dependent corrective action may be performed if required.

I/O Request Servicing.

I/O request servicing provides the means by which application programs can utilize the fault-tolerance capabilities of the system. The servicing consists of voting and error signalling, and can cater for all types of input and output which can be performed by the system. The characteristics of the input or output are held in a data base, enabling the servicing routines to provide the appropriate action for any type of I/O required.

I/O Records.

The chief I/O records which are maintained by the system are the device characteristic records, which hold such information as reason bounds, maximum tolerable rates of change, and tolerance values.

In addition, error and fault records are maintained by the system. This allows the operating system to make decisions about the use of input and output channels.

5.4.2 Self and Mutual Testing

Tests.

A broad functional partitioning of the system was undertaken, to identify the various sections to which testing could be applied. Testing is broken into the sections of processor, memory, input and output equipment, and inter-node communication

equipment. In addition, there is the facility for invoking remote tests, and for servicing such remote invocations.

- Processor testing is accomplished by first testing critical instructions, and then using those instructions to check all other possible instructions.

- Memory tests consist of setting parts of memory to various known values, and reading them back to see if they have changed.

- The form of the input and output testing depends on the characteristics of the input and output devices, and special routines are needed for each different type.

- Inter-node communication equipment tests consist of passing known messages across the links, and checking that they are as they should be.

- Remote tests are made up of combinations of the standard tests, but the results from them are evaluated by the processor that invokes the test, rather than the one being tested.

Test scheduling is accomplished dynamically. Tests may be run whenever specifically required by the application program, or may be invoked by the operating system. Invocation by the operating system can be either in response to the occurrence of an error, or as a preventative measure whenever there is spare processing time.

The diagnosis of an error results in the creation of an error record, and if it is determined that a fault exists, a fault record is created. Faulty devices are marked as such.

5.4.3 Time-staggered Operation

The clocks in each node are set at initialization time to their staggered values. Periodic checks are made on the clocks to ensure that they do not drift unacceptably far apart. If excessive clock drift is detected, then the clocks are reset.

If the clock drift is such that a clock fault is indicated, then an error record is generated.

5.4.4 Watchdog Timing

When each task has been completed, the operating system in the node provides the other nodes with the task identification, and task completion time. Before executing the next task, the operating system checks that the correct task has just finished, and that it finished within an acceptable time. Errors are signalled when either of these tests shows a discrepancy.

5.4.5 Error Handling

Whenever an error is detected, the error handling mechanism is initiated. Consistency restoration, node reconfiguration and node resetting are provided by the mechanism, whenever such actions prove to be necessary.

Consistency restoration is accomplished in the voting process, when erroneous data is overruled by the votes of the correct processors. It is also accomplished for task errors, when the watchdog mechanism alerts the node to the incorrect execution of a task.

When a node has detected the (predetermined) number of similar errors that indicates that a fault is present, it performs fault pinpointing, to identify the fault. This is done using the information held in the error records, together with additional testing when necessary.

Node reconfiguration is invoked when other nodes determine that a node has erred more frequently than is acceptable. This involves an instruction to the node to perform self-testing. Meanwhile, the rest of the system continues, ignoring that node until its usefulness has been proved.

At this time, the good nodes reset the previously erring node to a state which is consistent with the state of the rest of the system. This implies that the node must be made to execute the correct next task, with the correct data.

The creation of error and fault records is also part of the error handling mechanism. Routines to accomplish this creation are invoked by any routine that discovers an error.

5.4.6 Task Control

Scheduling and descheduling of tasks is done via operating system "calls" that keep track of time usage. These routines can intercept any overlapping that might be caused by erroneous application program scheduling. The tasks to be executed are kept in a linked list, and are executed sequentially by the operating system.

Test and operating system tasks are handled in just the same way as application tasks. An "ending" task, that runs after every other task, performs operating system maintenance on the scheduling list. The task removes the last-executed task, and schedules tests if there is time. Then it initiates the next task.

It should be noted that the scheduling scheme only permits a task to be executed if it can be run to completion in the available time. No task suspension is allowed, and task priority is not supported. This avoids the problems involved with preserving the state of tasks when their execution is temporarily suspended.

To aid diagnostics, a list of all tasks that have run, with their initiation and completion times, is maintained.

5.4.7 Inter-Node Communication

In order to ease the use of the communication system, and to allow error interception, routines are provided for inter-node transmission and reception. These routines are expected to be used primarily when sensor inputs or control outputs are to be used, when the actual device is not attached to all nodes.

For the exchange of data for voting, another routine is provided, that makes the communication system transparent to the user.

The status of the communication controller is intercepted by the routines so that errors can be dealt with by the operating system rather than the application program.

Modelling of the bus is provided by a program resident in the experimental system supervisor node.

5.4.8 System Initialization

The operating system ensures that the environment is correctly initialized before application tasks are allowed to be executed. This procedure takes the form of the initialization of all devices, followed by checks to ensure that the initialization was successful.

Initialization of the environment is followed by extensive system testing to prove the usefulness of all the components of the system. If critical components fail the tests, then a warning that the system can not be used, is given, and the system halts. Error and fault records are generated as necessary.

5.5 Software Outline

5.5.1 Introduction

The software consists of a set of routines which are used by both the operating system and the application programs. Each node has a copy of these routines (See appendix 9 for complete descriptions of the software).

In addition to routines used by both the operating system and the application, a suite of operating system subroutines is available. These consist mainly of test routines that allow system validation, and record maintenance facilities.

The other routines in the system are available to the application programs, to allow task scheduling, task descheduling, inter-node communication, input validation and output validation. Using these routines, error interception is possible.

5.5.2 Operating System Routines

Start up.

At start-up, the first operating system routine is started. This routine initiates and tests the local node, then communicates with the other nodes to set up the clock system. After the clock initialization has been checked, the routine schedules the first application task. It is the responsibility of this task to schedule all other application tasks.

Task ending.

At the end of every task, control is transferred to an end-off routine, which ensures that the last task was correctly executed on all nodes. It then ascertains the delay needed before the next application task is to run, and if there is enough time, runs one or more of the test routines.

Test routines.

These test routines allow testing of the processor, memory, I/O facilities and inter-node communications equipment, of both the local node and any other node.

Whenever an error is detected, by either test routines or voting, a routine to handle the error is invoked. This routine makes a record of the error, and calls a fault-handling routine if more errors than acceptable have occurred.

Fault handling.

The fault-handling routine ascertains the nature of the fault, by performing more tests if necessary, and records the existence of the fault. This allows the status of parts of the system to be known to the operating system at any time. Hence it will not utilize identified faulty components.

5.5.3 General Routines

The rest of the routines are available to both the operating system, and the application programs.

Input and output.

The most important routines are those which allow validated output and input. These routines perform voting and checking of data. Input from, and output to memory is also supported by these routines.

When an input is required, the routine to validate input is called, and returns a validated piece of data, and a status value which informs the caller whether or not errors have occurred. The routine makes use of other routines that test the validity of the data with respect to reasonableness and consistency.

Similarly, the output routine performs the necessary output, and returns a status value. The routine makes use of routines that check feedback and readback signals if they are available.

Task handling.

A routine is provided for scheduling other tasks, or rescheduling the calling task. This routine checks that the request will cause no clashes. It then adds the task to the scheduling list.

Similarly, a routine is provided for removing a task from the scheduling list. This is useful for halting a repetitive task, and for clearing the scheduling list under error conditions.

Inter-node communications.

In order to simplify the use of the inter-node communication equipment, routines are provided for reception and transmission using the inter-node bus. Another routine is provided to perform mutual data exchange among all the nodes.

This is useful when equipment on the bus is not attached to one of the fault-tolerant nodes of the system, and for exchange of data for voting.

6.6 Summary

The hardware system consists of four single board computers, a common memory board, and an analog input and output board, all housed in a single 12-card Multibus rack.

The software was required to provide fault-tolerance mechanisms for the system, of which the major functions are

- Task I/O handling

- Self and mutual testing of nodes

- Time-staggered operation

- Watchdog timing

- Error handling

- Task control

- Inter-node communication

- System initialization

These requirements led to the functional specification of the software system, which provided a definition of the necessary functions.

The programs of the system consist of two types: those available to both the operating system and the application programs, and those available only to the operating system. The general routines consist of modules that allow validated input and output, task scheduling and descheduling, and inter-node communication. The operating system routines consist of test modules, and record maintenance modules.

Having covered the design of the system, the next chapter describes the way in which all parts were integrated to produce the total system.

Chapter 6 - SYSTEM INTEGRATION

6.1 Introduction

In order for the system to be used as an experimental tool, extensive control and monitoring of its behaviour had to be possible. These functions are provided by an extra processor, situated in the Multibus rack, together with the nodes of the fault-tolerant system. In addition to control and monitoring, this processor also provides simulation of the inter-node communication system. In the remainder of the chapter, this node will be referred to as the supervisor node.

6.2 Requirements for the Supervisor Node

The function of the supervisor node can be broken into two distinct sub-functions. These are

1) Handling of inter-node communication

2) Handling of system control and monitoring

6.2.1 Inter-node Communication

Several aspects of the communication system are controlled by the supervisor node. Firstly, the node has to service all the communication system initialization commands from the nodes. This involves the detection of the commands from each node, and the provision of appropriate responses to the nodes.

Secondly, the supervisor node has to service every communication system command from the nodes. It is therefore necessary to detect the commands, read the system command block to find the command block list, and provide the actions required by the node. This process is described in detail in a following section.

In order to support the handling of communication system commands, it is necessary to maintain a model of the communication system that provided an interface which is compatible with the intended communication protocol.

The model of the communication protocol is based on the external behaviour of an existing communication interface board. A reduced definition of this behaviour was derived and used as a specification for the modelling program. The reduced definition is given in appendix 10.

In addition to maintaining the functions of the communication system, the supervisor node also keeps a record of all bus activity, so that monitoring of the bus system is possible.

6.2.2 System Monitoring and Control

Monitoring of the system is provided by the supervisor node. This is facilitated by the maintenance of appropriate records in the common memory section of the system. It was decided that sufficient visibility of the system could be provided if it was possible to observe bus activity, task execution lists, and error and fault records.

In addition to the availability of these data structures for examination, it should be possible to observe any part of the common memory, so that system debugging and maintenance could be facilitated.

In order to control the environment of the system in such a way that it would be forced to use its fault-tolerance facilities, it should be possible to inject errors and faults into the system. This function is again provided by the supervisor node.

The terminal-handling section of the node, which allows the user to specify the desired supervisory actions, is menu driven, hence providing an easy means for system control and monitoring.

6.3 Description of the Supervisor Node

In accordance with the requirements for the supervisor, the node contains two main sections, namely a section for the handling of

bus activity, and a section for the handling of terminal commands for system control and monitoring.

The attention of the supervisor node is divided between the two functions. While the supervisor is waiting for input from the terminal, it is also scanning the flags which indicate that bus modelling is required. The time taken for the servicing of a bus request is so short that a user does not notice any effect at the terminal.

6.3.1 Communication Handling

In order to describe the operation of the communication handling section of the supervisor node, it is necessary first to give a description of the communication protocols and buffering system (Refer to appendix 10 for more details).

Format of the Communication.

All communication to and from the communication interface board is done through common memory. Within this memory, a number of structures exist. Using these structures, the node using the bus interface board can specify the actions it requires from the board. It can also obtain information about bus activity or the status of the bus interface board.

The first structure is known as the System Command Block, or SCB. This command block allows the node to control the operation of the communication interface board in a broad sense; the commands allow resetting of the board, reading of its status, causing it to begin execution of a sequence of commands, and causing it to suspend this execution or stop it altogether.

In addition, the SCB points to the locations of the other structures used in the control of the communication interface board.

The Command Block List, or CBL, is a structure which contains the detailed instructions for the bus interface board. These take

the form of Command Blocks (CBs), each of which describes an action which the interface board is to take, and provides memory locations for responses from the board.

The f[illegible] control structure is the Bus Event Queue (BEQ), which hold[illegible]t Descriptor Blocks (BEDBs). Each BEDB contains det[illegible] a bus event that applies to the node which the communi[illegible]erface board is serving.

The interface board can be in one of three states: bus controller, remote terminal, or bus monitor. There is only one bus controller, and it is the only node that can initiate bus activity. It is possible for bus controllership to be transferred between nodes, provided that the interface board attached to the node is capable of controlling the bus.

Remote terminals use the bus, but only when instructed to do so by the bus controller. Bus monitors never use the bus; they can only monitor its activity.

Since all bus activity is initiated by the bus controller, there is no need for there to be a BEQ in the bus controller memory structure. Therefore, BEQs apply only to remote terminals.

In addition to this restriction, some action commands apply only to bus controllers, while others apply only to remote terminals.

Communication Handling

The model for each communication interface consists of receive and transmit buffers, plus intermediate receive and transmit buffers which are situated in common memory. It is via these intermediate buffers that messages are passed to and from the supervisor node, because both the fault-tolerant node and the supervisor node have access to this memory, but not the local node memory.

In addition to these buffers, the supervisor node maintains records of the status of the interface board.

At initialization, the supervisor node provides the correct responses to commands from the nodes, even though it does not need the initialization information which they send it.

During - normal operation, the supervisor node waits until a flag is set in common memory, which indicates that a node requires servicing. It then determines which node requires the servicing, and resets the appropriate flag.

The processing sequence continues with the supervisor reading the System Command Block. If the command block does not require the execution of a Command Block List, then the supervisor node simply executes the command required by the System Command Block, and then terminates.

If execution of a CBL is required, then the supervisor node obtains the first Command Block from the list and services it appropriately. Processing of the CBL continues until a CB contains an instruction to suspend execution of the CBL, or one which indicates that the end of the list has been reached.

The action commands which are supported by the communication handling software are:

NOP - No action is taken, but this command is useful when manipulating CBLs.

CONFIGURE - Allows each node to configure its communication interface as a bus controller or as a remote terminal. The communication handling software ensures that only th first node which attempts to configure as a Bus Controller (BC) is successful.

BUS CONTROL ACCEPTANCE ENABLE - Used only by Remote Terminal Units (RTUs), this command is used to inform the bus interface

whether or not the node is prepared to accept the responsibility of controlling the bus system.

BUS CONTROL ACCEPTANCE DISABLE - Used to disallow acceptance of bus control.

SET SUBSYSTEM ERROR - Sets the subsystem error bit in the status word for the node.

CLEAR SUBSYSTEM ERROR - Resets the subsystem error bit in the status word for the node.

WRITE TO TX BUFFER - Copies data from the node into the transmit buffer in the interface memory.

READ FROM RX BUFFER - Copies data from the interface memory to the node.

TRANSMIT TO RTU - Used by the bus controller, this command causes an RTU to receive from the bus, and to place the received data into its receive buffer. This command can also be used to offer bus controllership to the RTU, or request it to send its current status to the controller.

RECEIVE FROM RTU - Used by the bus controller, this command causes an RTU to transmit the contents of its transmit buffer over the bus.

TRANSFER FROM RTU TO RTU - Used by the bus controller, this command causes one RTU to transmit the contents of its transmit buffer over the bus, and causes another RTU to receive that data.

6.3.2 System Monitoring and Control

By means of a menu-driven program, the user can specify errors and faults which he wishes to introduce into the system. Also, the user can cause the fault, error, bus activity or task execu-

tion records to be displayed. It is also possible to examine any part of the common memory, in bytes.

In order to introduce faults and errors into the system, a set of flags is maintained in the common memory area. At the request of the user, these flags are set or reset.

In the fault-tolerance software of the system nodes, these flags are examined at appropriate times, and system behaviour is modified according to their settings.

If an error flag is found to be set, then the program introduces an error into the test which should detect that error. Thereafter, it clears the error flag. Similarly, if a fault flag is found to be set, then the error is introduced, but the flag is not reset by the system software. Hence, the setting of errors causes only one erroneous operation, while the setting of a fault causes erroneous operations to continue until the user chooses to reset the fault flag.

The system can set any of the faults and errors shown in tables 1 and 2.

80186 CPU fault
80130 fault
8259 PIC fault
8274 MPSC fault
8255 PPI fault
Inter-node communication fault
Node operation fault
Memory fault (including the exact memory area which is faulty)
80186 initialization error
80130 initialization error
8259 initialization error
8274 initialization error
Communications initialization error
Core instruction error
Instruction error
Memory error
80130 interrupt error
80130 timer error
8259 PIC error
8274 MPSC error
8255 PPI error
Node isolation error
Node resetting error
Clock setting error
Scheduling error
Descheduling error
Data error
Remote test voting error
Remote test activation error
Task completion task error
Task completion time error
Next task voting error

TABLE 1 - Node Errors and Faults

Communication link fault, for links 0->1, 0->2 or 1->2
Input fault, for inputs 0, 1, 2, or 3
Output fault, for outputs 0, or 1
Communication link error, for links 0->1, 0->2, or 1->2
Input consistency error, for inputs 0, 1, 2, or 3
Input reason error, for inputs 0, 1, 2, or 3
Output readback error, for outputs 0, or 1
Output feedback error, for outputs 0, or 1

TABLE 2 - General System Errors and Faults

The display which is invoked to show the error records indicates the number of times each error has been detected, as well as the node in which it occurred, and the node(s) which detected the error.

The fault display shows only whether a fault is present or not, because they are regarded as permanent. It also shows the node in which the fault was found, and the node(s) which found it.

Bus activity is indicated by time, initiating node, action command, source node (if relevant) and destination node (if relevant).

The execution of tasks is recorded using start time, end time and task identification number.

6.4 System Testing

System testing was achieved by implementing a simple servo-control system using the fault-tolerant equipment. Three application tasks were introduced into the software. The first of these application tasks served to initialize the hardware required to

perform control of the servo system. In addition, this task scheduled the other two tasks.

The second task utilizes the analog input equipment to read a position request value from a potentiometer associated with the control system. The task is run periodically, so that changes in request value can be reflected in the position of the servo device.

The final application task performs actual control of the servo device. The request value is obtained from memory, and parameters of the servo device are measured via the analog input equipment. Elementary calculations are made using this information, and a signal, for application to the servo system, is derived. This value is sent to the servo via the analog output facility. Validation is applied to this output. This control task is also run periodically, but at a faster rate than the request value task.

6.5 Summary

The supervisor node provides two functions; namely the servicing of communication system requests, and handling of user requests from the terminal.

The communication system is imitated in such a way as to provide the same command format as would be present if the actual communication interface boards were used.

The supervisor node allows the user to specify errors and faults for injection into the system. Also, error, fault, bus activity and task execution records can be displayed. Common memory can be examined.

System testing was accomplished in a simple servo-control environment.

This concludes the description of the experimental system. The following chapter summarizes the work, and discusses several issues which arose in the development of the experimental system.

Chapter 7 - CONCLUSION

7.1 Summary

7.1.1 Fault Tolerance

It was pointed out earlier that fault tolerance is a technique which attempts to allow a computer system to operate correctly in the presence of faults. Such an approach has become necessary because of the serious effects which the failure of some computer systems could cause. These computer failures can have their origins as far back as the system specification phase of system development, or the cause can be as immediate as electromagnetic interference. Specification faults will be propagated through the system design, until they manifest themselves in a way which causes the computer to behave incorrectly. Implementation faults and component faults are also a problem, while external disturbances can clearly lead to system failure.

Fault tolerance involves the detection of errors, identification of the fault which caused the error, confinement of the damage caused by the fault, system recovery, and finally, system repair.

In order to accomplish these effects, replication of system resources is a key principle. This replication may take the form of information redundancy, hardware redundancy, software redundancy and/or time redundancy.

7.1.2 Software Development Techniques

One of the major premises adopted when designing fault-tolerant systems is that the software is fault-free. Hence, it is essential that extreme care is taken in the design of the software section of the system. The use of modern software development techniques improves the quality of the software which is produced. As has been pointed out, the steps which are generally applied when attempting to put good software design into practice are:

- complete and accurate specification of the required software

- decomposition of the specification into manageable steps

- construction of the software

- proof of software correctness

A number of tools and techniques are used to aid this process, many of which have been used in the present work.

7.1.3 System Description

The objective of the design described in this dissertation was to produce a flexible experimental system which could be used to study various fault-tolerance techniques. Because of limited hardware possibilities, as well as for flexibility, software implemented fault-tolerance was chosen as the implementation method, together with replication of processor boards.

The system produced provides a number of facilities for the tolerance of faults, including:

- task input and output voting

- self and mutual testing of the processor boards

- time-staggered operation for the avoidance of common-mode fault problems

- device operation validation, for input and output devices

- watchdog timing, in the form of task completion checks

- fault handling

7.1.4 System Design

The composition of much of the system was defined by available equipment and tools, and the hardware consists of several single-board computers, a common memory board, and an analog input and output board, all housed in a Multibus-compatible rack.

The program modules in the system consist of two types: those available to both the operating system, as well as to the application tasks, and those which are available only to the operating system. The general routines consist of modules that allow validation of input and output, task scheduling and descheduling, and inter-node communication. The operating system routines consist essentially of test modules and record maintenance modules.

7.1.5 System Integration

In order for the system to be used as an experimental tool, extensive control and monitoring had to be possible. These functions are provided by an extra processor, which also provided handling of inter-node communication. This, in turn, required the modelling of the intended communication protocols.

System testing was done by providing a small application program which controlled a modular servo system.

7.2 Discussion

7.2.1 Fault Tolerance

It was thought that an examination of the techniques of fault-tolerance would reveal a dominant technique which could be used as a basis for the design. Such was not the case, however; it became clear that almost all the different ways to achieve fault-tolerance functions had found use in commercial systems.

Furthermore, this variance was not constrained to the different application areas; in the field of transaction processing, for instance, multiple replication (the Tandem system) [12], pair-and-a-spare (the Stratus system) [25] and hot back-up (the Resilient system) [12] are all used.

Clearly, then, a very flexible design had to be used for the experimental system. Also, a single configuration had to be chosen for implementation on the system. Although triple modular redundancy was chosen, any of the other techniques could clearly be just as effective for some applications.

7.2.2 Software Development Techniques

The field of software engineering is so vast and new that only a few of the more important techniques could be applied. In addition, the fact that the programming "team" consisted of one person proved to be somewhat limiting, in that the benefit of an unbiased opinion was lacking.

The principle of structured top-down design appeared to be most applicable to the type of system which was to be constructed. Data flow techniques appeared to be most easily applicable to transaction processing systems, so they were not used in the design process.

Following the chosen structured design procedure proved to be useful: the system development steps could be seen to follow properly from one another. This allowed good consistency to be maintained in the design. The system functional specification, discussed in chapter 4, led to the software requirement specification (Appendix 7), which led to the software functional specification (Appendix 8), and then to the software description (Appendix 9).

At all stages, it was possible to apply verification by comparison with the previous stage.

7.2.3 Evaluation of the Experimental System

In order to allow flexible use of the experimental system, a large number of different fault tolerance facilities have been provided, which can be incorporated as the user desires.

It is also possible to produce different configurations by making small changes to the relevant software. During system testing, much was done using the dual system configuration, even though the software was based on the triple modular redundant structure. This supports the claim that the software is easily modifiable.

A number of relevant issues arose during construction and testing of the system. These are discussed briefly in the following paragraphs.

System Features.

I/O Record Adaptability. The application programs in the system know the input and output channels of the system only by the identifier of the variable holding the relevant value. The I/O record structure holds all equipment-dependent information which is necessary to perform the I/O function. Hence, it is possible to allocate any input or output to any appropriate input or output equipment in the system, simply by setting the I/O records accordingly.

Input Consistency. Since a major error-detection mechanism in the system is the application of voting, it is essential that input to all the nodes is consistent. If such is not the case, then two error-detection opportunities are lost: firstly, voting on inputs can not be used, and secondly, voting on any results which are based on the inputs is useless, because the results cannot be guaranteed to be the same.

Therefore, it is highly recommended that all application programs make use of the input and output validation procedures provided by the system.

System Limitations.

Control Algorithm Selection. It should be noted that the control system implemented using the experimental fault-tolerant computer is very simple. It was not the purpose of the project to produce an optimized control algorithm, but rather to show that the ideas could be used in the control of real systems. The complexity and sophistication of the application system is left to the application programmer.

Task Clashing Errors. The system was designed to report an error when a scheduling call would cause a clash with a previously scheduled task. It has been found that it may be useful to allow clashes to be reported simply by returning a status value to the task attempting to perform scheduling. In this way, the task can continue in its endeavour to perform scheduling with a different schedule time so as to avoid the clash. Alternatively, the application may wish to perform some other action.

Therefore, clashes are not errors as such, and should not be reported to the error handling system.

Synchronization.

Synchronization proved to be troublesome. This was mainly due to inaccurate specification of individual task execution times. This led to task execution over-runs in some cases. Since the node which controls the bus has to execute different instructions for every bus access than do the remote terminal nodes, these over-runs led to loss of synchronization.

A second scheduling problem was that relative scheduling was initially used (i.e. tasks were scheduled to run at the "present time" plus a predetermined offset). Again because the bus controller executes different instructions when the bus is used, this type of scheduling led to synchronization loss.

To combat these problems, two solutions were used. Firstly, the proper execution times were ascertained by observing the task execution records using the supervisor mode. A safety factor was added to these values, and the task records were updated to reflect the new execution times.

It is expected that this will be the best way of determining the execution times of any application program installed on the system.

Secondly, absolute scheduling was used. The schedule time for a repetitive task is calculated by referring to the number of times the task has already executed. This may lead to problems in a multi-application system where repetitive tasks are added and removed in an unpredictable way. Another method would have to be devised for such a case. However, the project was not intended to produce scheduling techniques for complex applications, but rather to provide the facility for scheduling and descheduling in any user-chosen manner.

7.3 Conclusion

Considerable study into the subject of fault-tolerance was undertaken, allowing familiarity with the principles, terminology, tools and techniques to be gained. In addition, the experimental system produced formed a useful tool, allowing many of the techniques to be tried out in an actual control environment.

Because of the extensive control and monitoring provided by the system, and also because of the simple control algorithm used, device control performance is somewhat slow. This means that the system itself could not be used in any but the most trivial control applications. However, the ideas behind the system, and also some of the software modules, could find use in real control systems.

REFERENCES

[1] AVIZIENIS, A., Fault-tolerance: The Survival Attribute of Digital Systems, Proceedings of the IEEE, Vol. 66, No. 10, October 1978.

[2] ANDERSON, C. A., Development of an Active Fly-by-wire Flight Control System, Advanced Control Technology and its Potential for Future Transport Aircraft, NASA, 1974.

[3] HUMPHRY, J. A., Fault Tolerance and Micros in the Real World, IEEE Micro, December 1984.

[4] Mc GILL, W. F., et al, Fault Tolerance in Continuous Process Control, IEEE Micro, December 1984.

[5] EMMERSON, R., et al, Fault Tolerance Achieved in VLSI, IEEE Micro, December 1984.

[6] ANDERSON, T., et al, Fault Tolerance Principles and Practice, Prentice/Hall International, 1982.

[7] JOHNSON, B. W., Fault-tolerant Microprocessor-based Systems, IEEE Micro, December 1984.

[8] BOEHM, B. W., Software Engineering, IEEE Transactions on Computers, Vol. 25, No. 12, December 1976.

[9] ARMSTRONG, C. V. W., et al, A Fault-tolerant Multi-microprocessor-based Computer System for Space-based Signal Processing, IEEE Micro, December 1984.

[10] WENSLEY, J. H., et al, SIFT: Design and Analysis of a Fault-tolerant Computer for Aircraft Control, Proceedings of the IEEE, Vol. 66, No. 10, October 1978.

[11] SCHMITTER, E. J., et al, The Basic Fault-tolerant System, IEEE Micro, February 1984.

REFERENCES

[12] KILLMON, P., Fault Tolerant Systems deal with Increased Loads, Computer Design, January 1984.

[13] OSAKI, S., Performance/Reliability Measures for Fault-tolerant Computing Systems, IEEE Transactions on Reliability, Vol. R-33, No. 4, October 1984.

[14] BERG, H. K., Towards a Uniform Design Methodology for Software, Firmware, and Hardware, The Use of Formal Specification of Software, (Ed.H. K. Berg and W. K. Giloi), Springer-Verlag, 1980, pp 1 - 38.

[15] BOYD, D. L., et al, An Overview of RDM: Rational Design Methodology, The Use of Formal Specification of Software, (Ed. H. K. Berg and W. K.Giloi), Springer-Verlag, 1980, pp 79 - 110.

[16] GANE, C., et al, Structured Systems Analysis: Tools and Techniques, Prentice-Hall, Inc., 1979.

[17] Improved Programming Technologies, Management Overview, IBM, IPTO Support Group.

[18] JACKSON, M., Applying the Jackson Techniques, Deltak Inc., 1979.

[19] STEVENS, W. P., Using Data Flow for Application Development, Byte, June 1985.

[20] SAMID, G., Modified Top-Down Design, Datamation, November 1981.

[21] GILB, T., Distinct Software: A Redundancy Technique for Reliable Software, INFOTECH State of the Art Report on Software Reliability, 1977, pp 117 - 133.

[22] BOEHM, B. W., Verifying and Validating Software Requirements and Design Specifications, IEEE Software, January 1984.

REFERENCES

[23] WILLIAMS, G., Debugging Techniques, Byte, June 1985.

[24] WENSLEY, J. H., Redundant Modules may be Best for Control Systems, Computer Design, April 1985.

[25] SERLIN, O., Fault-tolerant Systems in Commercial Applications, Computer, Vol. 17, No. 8, August 1984.

[26] MEYER, J. F., et al, Performance Evaluation of the SIFT Computer, IEEE Transactions on Computers, Vol C-29, No. 6, June 1980.

[27] MELLIAR-SMITH, P. M., et al, Formal Specification and Mechanical Verification of SIFT: A Fault-tolerant Flight Control System, IEEE Transactions on Computers, Vol C-31, No. 7, July 1982.

[28] LAMBRECHS, J. S. D., The Application of Micro-electronics in Fail-safe Systems, Ph. D. Thesis, University of the Witwatersrand, 1983.

[29] HOPKINS, A. L., et al, FTMP - A Highly Reliable Fault-tolerant Multiprocessor for Aircraft, Proceedings of the IEEE, Vol. 66, No. 10, October 1978.

[30] JOHNSON, B. W., et al, Fault Tolerant Computer System for the A129 Helicopter, IEEE Transactions on Aerospace and Electronic Systems, Vol. AES-21, No. 2, March 1985.

[31] JOHNSON, D., The Intel 432: A VLSI Architecture for Fault-tolerant Computer Systems, Computer, Vol. 17, No. 8, August 1984.

[32] Components Make Systems Fault Tolerant, Computer Design, June 1983.

[33] BRAUN, E. L., Digital Computer Design, Academic Press, New York, 1963, pp 521 - 553.

REFERENCES

[34] SEDMAK, R. M., Built-In Self-Test: Pass or Fail?, IEEE Design and Test of Computers, April 1985.

[35] BRAHME, D. et al., Functional Testing of Microprocessors, IEEE Transactions on Computers, Vol C-33, No. 6, June 1984.

[36] LEWIN, D., Theory and Design of Digital Computers, Thomas Nelson and Sons, London, 1972, pp 284 - 288.

[37] ILLMAN, R. J., Self Tested Data Flow Logic: A New Approach, IEEE Design and Test of Computers, April 1985.

[38] BASHKOW, T. R., et al., A Programming System for Detection and Diagnosis of Machine Malfunctions, IEEE Transactions on Electronic Computers, Feb 1963.

[39] Data Translation, User Manual for DT711, DT732 Analog Input and Analog I/O Boards, 1984.

[40] Feedback Instruments Ltd., Modular Servo Type MS105, Parts I and II.

[41] Feedback Instruments Ltd., Modular Servo Type MS150, Book 2: Circuit Notes and Maintenance, 1980.

[42] Feedback Instruments Ltd., Modular Servo Type MS150, Introductory Experiments.

[43] Intel, iSBC 186/03 Single Board Computer Data Sheet, 1984.

BIBLIOGRAPHY

1. ALLAN, R., Local-Net Architecture. Protocol Issues Heating Up, Electronic Design, 16 April 1981, pp 91 - 102.

2. ASEO, J., Approaches Differ for Fault-Tolerant Systems, Computer Design, September 1983.

3. AVACHE, J., et al, REBUS, A Fault-tolerant Distributed System for Industrial Real-time Control, IEEE Transactions on Computers, Vol. C-31, No. 7, 1982, pp 637 - 647.

4. BATES, K. H., et al, Shadowing Boosts System Reliability, Computer Design, April 1985.

5. BOEHM, B. W., et al, A Software Development Environment for Improving Reliability, IEEE Computer, June 1984, pp 30 - 44.

6. BORRIL, P. L., Microprocessor Bus Structures and Standards, IEEE Micro, February 1981, pp 84 - 95.

7. BRAYER, K., Implementation and Performance of Survivable Computer Communication with Autonomous Decentralized Control, IEEE Communications Magazine, July 1983.

8. CLARKE, E. M., Distributed Reconfiguration Strategies for Fault-tolerant Multiprocessor Systems, IEEE Transactions on Computers, Vol. C-31, No. 8, August 1982.

9. CLAY, C. W., Digital Electronic Flight Decks: The Outlook for Commercial Aviation, IEEE Transactions on Aerospace and Electronic Systems, Vol. AES-20, No. 2, March 1984.

10. CONDOM, P., Fly-by-wire Systems and CCV Design, Interavia, 1/1985, pp 50 - 51.

BIBLIOGRAPHY

11. CRISTIAN, F., Exception Handling and Software Fault-Tolerance, IEEE Transactions on Computers, Vol. C-31, No. 6, June 1982.

12. CROSSGROVE, A., Development and Applications of MIL-STD-1553, Aerospace Congress and Exposition, Los Angeles, California 1980.

13. DANIELS, R. G., Built-In Self-Test Trends in Motorola Microprocessors, IEEE Design and Test of Computers, April 1985.

14. DAVIS, R. T., Solutions Considered for Future High-speed Military Bus, Defense Electronics, Vol. 15, No. 5, May 1983.

15. DEPARTMENT OF DEFENSE: USA, MIL-STD-1553B, Aircraft Internal Time Division Command/Response Multiplex Data Bus, 21 September 1978.

16. ENGELLAND, J. D., The Evolving Revolutionary All-Electric Airplane, IEEE Transactions on Aerospace and Electronics, Vol AES-20, No. 2, March 1984.

17. FRIEDMAN, S., Manchester Coding Data Bus, Measurements and Control, Vol. 17, No. 3, Issue 99, June 1983.

18. FURLOW, B., Fewer Connections Translate into Fewer Failures, Computer Design, April 1985.

19. HOLTON, J. B., et al, Structured Top-Down Flowcharting, Datamation, Vol. 21 (5), May 1975.

20. IHARA, H., et al, Auto[illegible]ntralized Computer Control Systems, Computer, Vol. 17, No. [illegible]gust 1984.

21. JOHNSON, T. W., Design of a Digital Flight Control System using Area Multiplexing, IEEE NAECON, 1976.

22. KADIS, A., CMOS VLSI Increases Life Expectancy of Complex Systems, Computer Design, April 1985.

23. KAMEYAMA, M., et al, Design of Dependent-failure-tolerant Microcomputer Systems using TMR, IEEE Transactions on Computers, Vol. C-29, No. 2, February 1980, pp 202 - 206.

24. KANOPOULOS, N., A Totally Interactive Computer Program for Reliability Evaluation of Fault-tolerant Digital Systems, IEEE NAECON, 1983.

25. KOREN, I., et al, A New Approach to the Evaluation of the Reliability of Digital Systems, IEEE Transactions on Computers, Vol. C-29, No. 3, March 1980.

26. KUBAN, J. R., Self-Testing the Motorola MC 6804P2, IEEE Design and Test of Computers, May 1984.

27. LEDAMUN, D., et al, Exploring the Possibilities of the 1553B Data Bus, Electronic Engineering, Vol. 55, No. 675, March 1983.

28. LYMAN, J., Reliability's Big Promotion, Electronics Week, 11 March 1985.

29. MEYER, J. F., On Evaluating the Performability of Degradable Computing Systems, IEEE Transactions on Computers, Vol. C-29, No. 8, August 1980.

30. MIDDELBOE, S., Local Area Networks, Microprocessors and Microsystems, Vol. 6, No. 1, January 1982.

31. MINOTT, et al, Space Shuttle Digital Flight Control System, Advanced Control Technology and its Potential for Future Transport Aircraft, NASA, 1974.

32. MORGAN, L. F., Advanced Avionics Architectures for the 1980s: A Software View, IEEE Digital Avionics Conference, 1979.

BIBLIOGRAPHY

33. NAEGELE, T., Computer Controls 'Copters Avionics, Electronics Week, 12 August 1985.

34. NG, Y. W., et al, A Unified Reliability Model for Fault-Tolerant Computers, IEEE Transactions on Computers, Vol. C-29, No. 11, November 1980.

35. O'CONNER, P. D. T., Microelectronic System Reliability Prediction, IEEE Transactions on Reliability, Vol. R-32, No. 1, April 1983.

36. OSDER, S. S., The Implementation of Fail-Operative Functions in Integrated Digital Avionics Systems, Advanced Control Technology and its Potential for Future Transport Aircraft, NASA, 1974..

37. PINKOWITZ, D. C., MIL-STD-1553B: The Military Standard for Avionics Integration, Hearst Business Communications, Inc. / UTP Division, Vol. 26, No. 13, 28 March 1984.

38. PRADHAN, D. K., et al, A Fault-tolerant Communication Architecture for Distributed Systems, IEEE Transactions on Computers, Vol. C-31, No. 9, September 1982.

39. PRELL, E. M., et al, Building Quality and Productivity into a Large Software System, IEEE Software, July 1984.

40. RAGHAVENDRA, C. C., Fault Tolerance in Regular Network Architectures, IEEE Micro, December 1984.

41. Serial Digital Bus Heads for Industrial Systems, Electronic Design, Vol. 28, No. 19, 13 September 1980.

42. SHRIVASTAVA, S. K., Structuring Distributed Systems for Recoverability and Crash Resistance, IEEE Transactions on Software Engineering, Vol. SE-7, No. 4, July 1981.

43. SNEED, H. M., Software Renewal: A Case Study, IEEE Software, July 1984.

BIBLIOGRAPHY

44. SPITZER, C. R., The All-Electric Aircraft: A Systems View and Proposed NASA Research Program, IEEE Transactions on Aerospace and Electronic Systems, Vol. AES-20, No. 2, May 1984.

45. STALLINGS, W., Local Network Performance, IEEE Communications Magazine, Vol. 22, No. 2, February 1984.

46. THATTE, S. M., Test Generation for Microprocessors, IEEE Transactions on Computers, Vol. C-29, No. 6, June 1980.

47. TREACY, J. J., Flight Safety Issues of an All-Electric Aircraft, IEEE Transactions on Aerospace and Electronic Systems, Vol. AES-20, No. 3, May 1984.

48. TROPPER, C., Local Computer Network Technologies, Academic Press, New York, 1981.

49. TSAO, D. C., A Local Area Network Architecture Overview, IEEE Communications Magazine, Vol. 22, No. 8, August 1984.

50. VEATCH, M. H., Reliability/Logistics Analysis Techniques for Fault-Tolerant Architectures, IEEE NAECON, 1983.

51. VON BANN, J., Catastrophic Failure Modes Limit Redundancy Effectiveness, IEEE Transactions on Reliability, Vol. R-32, No. 5, December 1983.

52. WILLIAMS, D. G., Industrial Controller Joins the MIL-STD-1553 Bus, Electronic Design, 14 October 1982.

53. WILLIAMS, T. J., The Development of Reliability in Industrial Control Systems, IEEE Micro, December 1984.

54. WOOD, D., Jaguar to fly by Wire, Interavia, 3/1981.

55. ZELKOWITZ, M. V., et al, Software Engineering Practices in the US and Japan, IEEE Computer, June 1984, pp 57 - 66.

A1.1 Introduction

Numerous computer systems which utilize fault-tolerance techniques are commercially available. This is especially true in the fields of on-line transaction processing (OLTP) and control of machinery. Table A1-1 shows a number of commercial fault-tolerant systems.

Some of these systems stand out as being particularly important. They demonstrate the extensive use of fault-tolerance techniques in computers which perform functions that are critical in terms of safety or data integrity.

Among the most significant fault-tolerant computer systems, which will be reviewed in the following sections, are:

Control systems:

- The Software-Implemented Fault-Tolerance (SIFT) system
- The Fault-Tolerant Multi-Processor (FTMP) system
- The Triplex 32 system
- The Agusta 129 system

A signal processing system:

- The Fault-tolerant Array Signal Processor (FASP)

Transaction processing systems:

- The Tandem system
- The Stratus system
- The Resilient system

A REVIEW OF CURRENT FAULT-TOLERANT SYSTEMS APPENDIX 1

General purpose systems:

- The Intel 432 system.

- The Basic Fault-tolerant System (BFS)

SYSTEM	PURPOSE	PRIMARY FAULT-TOLERANCE MECHANISM
System 4000	OLTP	Replication Self-checking "I'm alive" messages
Power 55/5	OLTP	Replication Timeouts Voting
Parallel 300	OLTP	Hot back-up
Sequoia Systems	OLTP	Replication Self-checking
Stratus Computers	OLTP	Pair-and-a-spare Self-checking
Synapse N+1	OLTP	Replication Timeouts
Tandem Nonstop	OLTP	Replication "I'm alive" messages
Eternity	OLTP	Replication "I'm alive" messages Timeouts
Resilient	OLTP	Duplication "I'm alive" messages
Can't Fail 300	Process control	TMR SIFT

TABLE A1-1 - Fault-tolerant Systems [25]

SYSTEM	PURPOSE	PRIMARY FAULT-TOLERANCE MECHANISM
DAC-6000	Process control	Hot back-up Timeouts Cross diagnostics
Systemsafe/1000	Process control	Hot back-up Timeouts
Rebus	Process control	Replication Timeouts
Triplex 32	Process control	TMR
SIFT	Aircraft control	Replication Voting
FTMP	Aircraft control	Hybrid TMR Voting
Agusta 129	Aircraft control	Hot back-up Self-testing Timeouts Cross diagnostics
SPACE SHUTTLE	Flight control	Triple or quad redundancy Voting

TABLE A1-1 (cont.) - Fault-tolerant Systems

SYSTEM	PURPOSE	PRIMARY FAULT-TOLERANCE MECHANISM
JPL-Star	General	TMR with spares Voting
Intel 432	General	Pair-and-a-spare Functional redundancy
BFS	General	Replication Self-testing Cross diagnostics
Bell ESS	Telephone Switching	F ba[illegible]up or voting [illegible]king or cro[illegible]agnostics
FASP	Signal processing	Replication Timeouts

TABLE A1-1 (cont.) - Fault-tolerant Systems

A1.2 Control Systems

A1.2.1 The SIFT System [10][26][27]

The SIFT system is intended for use in aircraft control. In development of the system, failure modes (the different ways in which system components can fail) were not considered. Rather, only the distinction between failed and non-failed equipment was made. Low-level techniques for fault-tolerance, such as error detection and correction codes are not included in the design, since they offer little improvement.

The SIFT Concept of Fault-Tolerance.

System Overview. Reliability is achieved by having each iteration of a task independently executed by a number of processing modules. Each processor places the outputs of each iteration into memory allocated to that processor. A processor requiring this output determines the value to be used by reading the output generated by each processor which executed the iteration. Typically, a two-out-of-three vote is used, and errors are recorded for use by the executive system when determining which unit is faulty. Voting is minimized by considering data only at the beginning of each iteration. This means that processors may run in loose synchronization (such as to within 50 us of one another), allowing execution at slightly different times. The number of units which execute a task can vary. This is determined dynamically by the global executive task, which reconfigures the arrangement of the system when necessary.

Fault Isolation. Propagation of erroneous data is prevented by allowing each processor to write only to its own memory.

Fault Masking. Masking is achieved when necessary by majority voting between a suitable number of copies of any required data.

Scheduling. Two timing requirements are generally specified for control outputs:

- output to control actuators must be generated with a specific frequency

- the delay between the reading of sensors and the generation of outputs must not exceed a specified value

SIFT scheduling is a slight variant of a simple periodic method. Tasks are run at multiples of a base frequency, with the priority of a task determining its iteration rate.

Processor Synchronization. Even though processor synchronization is loose, clock drift or failure will result in the loss of this synchronization. To facilitate resynchronization, use is made of an algorithm which allows up to one third of the clocks to fail while still maintaining synchronization. The algorithm is as follows: each clock reads the values of all other clocks, as well as those clock's interpretations of the other clocks. If all readings for a particular clock do not agree, then it is ignored. The median of all valid clocks is found and used as the resynchronization value.

Reliability Prediction. The design goal was to attain a failure rate of less than 10^{-9} failures per hour for ten hours. Markov modelling was used to predict the reliability of the system, making the following assumptions:

- faults are uncorrelated, and distributed exponentially

- faults are permanent (transient faults are masked)

- the failure rate of the main processor modules is 10^{-4} per hour

- the failure rate of I/O processor modules and busses is 10^{-5} per hour

The reliability which was predicted from the calculations was acceptably high for the intended application.

The SIFT Hardware. (See fig. A1-1)

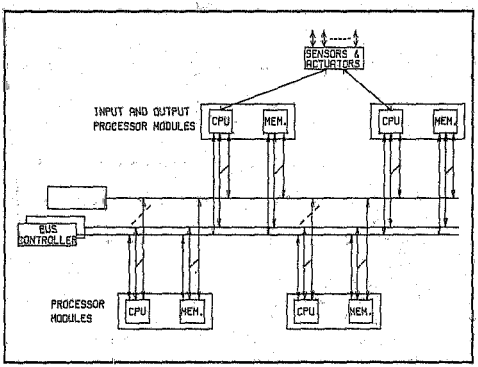


FIGURE A1-1 - The SIFT Hardware [28]

Standard units were used wherever possible. Processor-to-bus interfaces, bus-to-memory interfaces, and the busses were specially designed, however.

Interconnection system operation. Each bus controller continuously scans the processors to see if a bus operation is required. Similarly, each memory scans the busses to see if it is needed. Bus delays are insignificant, because of the small amount of data transfer that is required.

The SIFT Software.

The Application Software. Application software performs the actual flight control computations by means of iterative tasks. Input to, and output from the tasks is handled by the executive system.

The Executive Software. The executive system has several functions:

- It runs each task at its required rate

- It provides error-masked inputs

- It detects errors, and diagnoses their cause

- It reconfigures the system to avoid the use of failed components

Each processor runs a local reconfiguration task and an error-reporting task. Error reports are made to the global executive task, which decides on the necessary action, and places the appropriate command in a buffer. Local reconfiguration tasks read the buffer, and perform the required action.

Local executive tasks run each application task allocated to the processor, provide inputs to them, receive outputs from them, and report errors. Local executive tasks can be invoked by a running task, a clock interrupt, or a call from another local executive. Local executive tasks provide the following functions:

- error handling

- scheduling

- buffering

- voting

Fault Detection. An analysis of errors which have occurred can indicate which units are faulty. Each processor maintains a processor/bus error table, in the form of an m x n matrix, where m is the number of processors, and n is the number of busses. Each entry Xe (i,j) is the number of errors involving either processor i or bus j, as detected by processor p. The entries are compared with maximum tolerable numbers of errors, beyond which, faults are indicated. If the global executive is uncertain of the location of a fault, it can schedule diagnostic tasks.

Proof of Correctness.

Software correctness had to be proved mathematically, because of the vast number of combinations of possible states of the system. Because of the complexity of the system as a whole, models were used.

A1.2.2 The FTMP System [29]

The FTMP system is intended for use in aircraft control. Processor modules, with local cache memory, and memory modules, are connected by a redundant serial bus. Modules are associated into groups of three. Every module contains a voting element, and special hardware to prevent the propagation of faults from one module to another.

Rationale of the FTMP approach.

A failure rate of less than 10^{-9} failures per hour on a ten hour flight was required. Fault masking was to be used, and all system resources had to be verifiable during system operation.

A TMR-hybrid architecture is used, with graceful degradation. Operation is synchronous, allowing bit-by-bit hardware voting on all transactions. Modules can be reconfigured as necessary.

Theory of the FTMP.

Nominal Organization. Common memory holds high-level programs, while an extensive set of procedures, resident in cache memory, are used to interpret the programs. This cuts down on the amount of information that must be transferred when programs are loaded into a processor. The less unique information held in a processor's cache memory, the easier it is to perform reconfiguration.

Available processors examine a job queue, and select jobs which they have the resources to run. Hence, job allocation is dynamic, and adjusts itself to momentary load distribution and to module failures.

Access to I/O ports may be granted on a first-come first-served basis, or according to priority.

Redundant Organization. The FTMP uses redundant busses, three times the required nominal number of modules, plus spares. The modules are grouped into flexible triads, and three of the available busses also form a triad. Each member of a module triad uses a different bus for communication.

Bus guardian units govern the status of a module, consisting of power-on status, bus triad selection, transmission selection and some self-test configuration selection (See fig. A1-2).

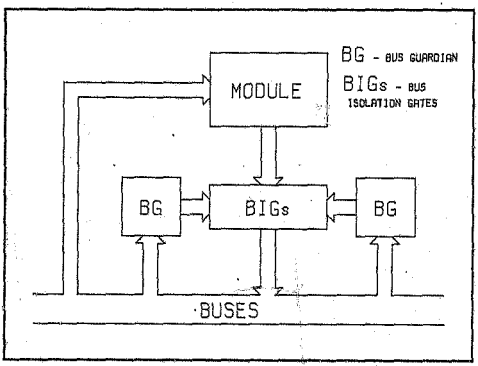


FIGURE A1-2 - An FTMP Module [29]

Both bus guardians must agree before power-on or bus transmission is permitted. Power, bus inputs and timing are separate for each bus guardian, and each is physically separate from other guardians and modules. Bus guardians are addressable as part of memory. Messages sent to bus guardians are commands which the bus guardians apply to their outputs until the command is superseded. The bus guardians are thus used as agents to convey the computer configuration authority to all elements of the computer.

The bus isolation gates are isolated from one another and their control lines are independent.

Processor and memory failures are handled as shown in figure A1-3.

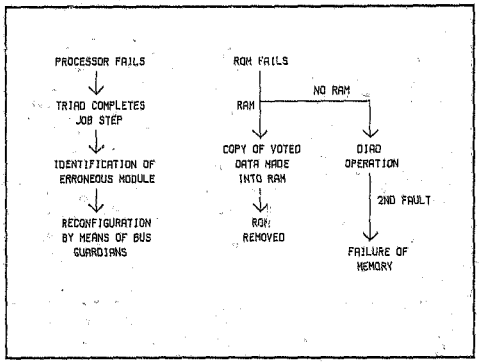


FIGURE A1-3 - Processor and Memory Failure Handling [29]

Synchronization. Synchronization is tight, allowing hardware voting, and easy programming. To achieve the required synchronization, the timing references are continuous and accurate. There are four clocks and clock lines, three of which are chosen by each processor for voting (See fig. A1-4).

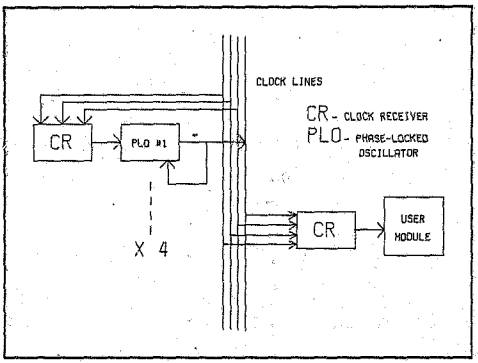


FIGURE A1-4 - The FTMP Clock System [29]

Failure of a clock distribution line appears as an oscillator failure, while failure of a clock receiver appears as a module failure.

Malfunction Management. The malfunction train is as shown in figure A1-5.

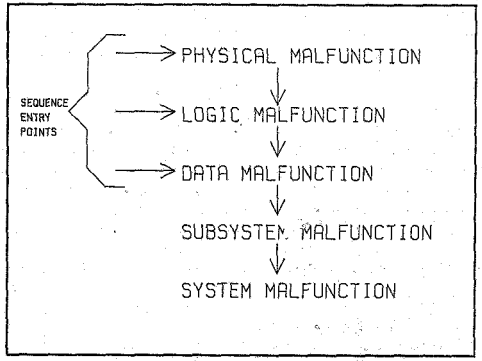


FIGURE A1-5 - The Malfunction Train [29]

Masking of errors holds malfunctions at a low level.

When a malfunctioning unit is identified, it is configured out of the system, and full fault-tolerance is restored. Reconfiguration may fail because of exhaustion of spares, malfunction of the reconfiguration mechanism, or the use of defective spares.

The FTMP reconfiguration mechanism is largely within the voters and bus guardian units. Failure of all guardian units in a single module can cause it to transmit on more than one bus, leading to errors.

Fault detection is accomplished when a disagreement occurs at a voter. Fault identification involves the discovery of the module, bus, or other element which failed. Redistribution of resources may be used to aid in this discovery.

Recovery is made by reassignment and reinitialization of modules. Program rollback is a secondary recovery process.

Survival and Dispatch Probability Models for the FTMP.

In order to evaluate the reliability of the FTMP system, the survival probability (the probability that the system will function correctly for the full duration of its use), and the dispatch probability (the probability that the system is put into use with sufficient resources to survive during its use) were calculated.

Survival Probability Models. Failures may arise from one of three sources:

- lack of perfect coverage - Markov modelling was used to predict the failure rate due to lack of perfect coverage. It was assumed that there were no latent faults at takeoff, that reconfiguration returns the system to a perfect state, that all failed busses are active, and that all undetected triple faults cause system failure. It was shown that the probability of failure due to imperfect coverage was dominant in the first 50 hours of system use. Most double failures could be tolerated, but some led to system failure. To achieve a system failure rate below 10^{-9} per hour for ten hours, the probability of latent faults at takeoff had to be below 10^{-6} per hour.

- exhaustion of spares - the minimum number of units which are necessary for the acceptable operation of the system, determines the point at which exhaustion of spares becomes significant. It was found that spare exhaustion became significant after 50 hours.

- Bus guardian unit transmission-enable mode failure - both BGUs would have to fail before the module could transmit incorrectly. It was found that the BGU failure rate was insignificant.

Dispatch Reliability. The "dispatch minimum compliment" is the minimum number of operational resources required at takeoff for sufficient system reliability. This number depends on the mission requirements. Dispatch reliability is the probability that this minimum compliment will be available at takeoff.

A1.2.3 Triplex 32 [4]

The Triplex 32 is a triple-modular redundant programmable controller, using Motorola 68000 microprocessors as processing elements. The controller contains three identical modules, each with its own CPU, memory, voting units and power supply. Each module monitors the operation of the others by means of a triplicated bus (See fig. A1-6). The modules are synchronized, and execute identical programs. If one module disagrees with the others, it is marked, and ignored.

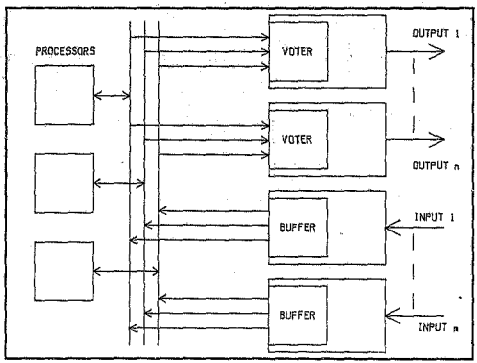


FIGURE A1-6 - The Triplex 32 Structure [4]

Faults are masked, and modular design permits quick replacement of faulty modules.

Faulty modules may be replaced without removing power. The replacement module is brought into synchronization by background software running on the other two modules, without interfering with the execution of the applications.

A1.2.4 Agusta 129 [30]

The Agusta 129 is a multiprocessor system which implements automated helicopter flight control, stability augmentation and navigation as well as many other functions. The system is dual redundant and is based on a MIL-STD-1553B bus (See fig. A1-7).

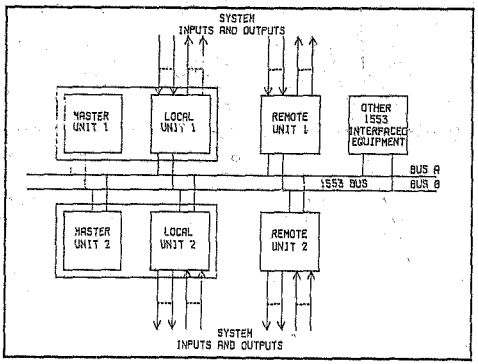


FIGURE A1-7 - The Agusta 129 Structure [30]

Each master unit has the architecture shown in fig. A1-8. Only one of the two units is active at any time, while the other per-

forms the same computations, and serves as a hot back-up. Each unit performs as much self-checking as time allows.

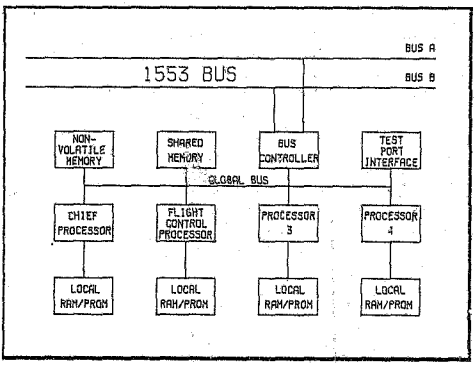


FIGURE A1-8 - The Agusta 129 Master Unit [30]

Fault-tolerance in this system depends largely on software implementation. The last resort check is cross-channel comparison between master units. If disagreement occurs, then the entire system is forced into a passive state. To limit the effect of mis-comparisons, more than one disagreement must occur before shut-down is allowed.

A1.3 The Fault-tolerant Array Signal Processor [9]

This system was developed to provide space-based digital signal processing. As such, the system required high reliability.

A1.3.1 Design Principles

In the FASP system, hardware resources are partitioned into processing modules which are interconnected to form a distributed system. Software is also partitioned, into tasks that run in a concurrent operating environment. Processing modules and tasks are under the control of a partitioned operating system. This partitioning is necessary because of the danger of failure in a central control system.

A1.3.2 System Architecture

The FASP comprises a number of identical hardware modules working together. The interconnection structure consists of a two-dimensional square matrix of busses.

FIGURE A1-9 - FASP System Operation [9]

System operation consists of the following levels (See fig. A1-9):

- the physical level - this includes direct memory access, and control of shared resources

- the data link level - in which a cyclic redundancy check is used to provide correction of data

- the network level - this provides system-wide communication between processing modules

- the task level - tasks communicate via mailboxes

- the applications level - several tasks interact to provide the necessary processing for a specific application

Faults at one level are usually corrected without involving other levels.

A1.4 Transaction Processing Systems

A1.4.1 The Tandem System [12]

The Tandem multiprocessor architecture eliminates single points of failure by removing master/slave relationships among processors, and by providing dual paths to all subsystems. It provides the ability to replace defective components without interrupting application programs. The key architectural features that underlie these capabilities are processor replication, dual-access I/O controllers, a redundant power system and a message-based operating system.

The Tandem Non-Stop system (See fig. A1-10) has from two to sixteen processors which communicate over a high-speed duplex 16-bit parallel bus system. Disk mirroring (identical disk copies) may be used.

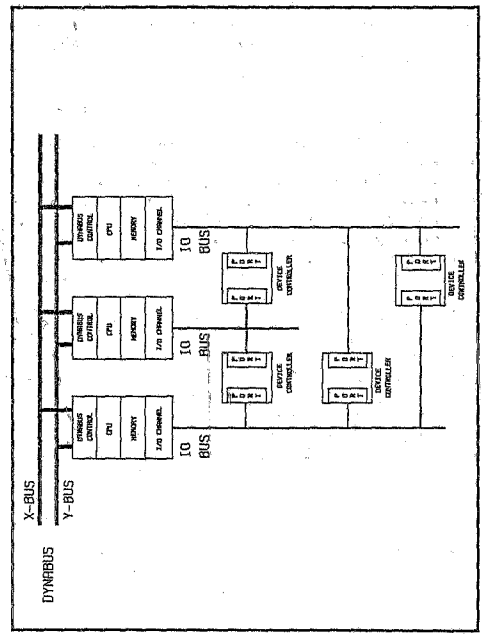


FIGURE A1-10 - The Tandem Non-Stop System [25]

Each processor is individually powered, and can shut down independently of the other system components. Each processor has a copy of a guardian operating system, which maintains tables that reflect the status of available system resources. Processors are required to broadcast "I'm alive" messages to all other processors, every second.

Check-Pointing.

For each running program, there is an identical, but semi-active back-up program in another processor. The back-up processor replaces the primary processor if it fails. The primary sends its back-up periodic check-point messages which define the state of the process at critical points in the computation. When the operating system activates the back-up, it resumes operation from the point which was defined at the last check-point.

Message System.

Isolation of user processes from configuration details is provided by forcing all inter process communication to be carried out via a message system. This means that communication is to logical devices, and the actual physical devices need not be controlled by the process. This isolation allows on-line repair, and graceful growth by plugging in additional equipment.

A1.4.2 The Stratus System [25]

Stratus developed the technique known as "pair-and-a-spare" [25], in which major functions are replicated four times. Each of two self-checking subsystems consist of a pair of identical units that receive the same inputs. If the outputs disagree, then the subsystem ceases to operate, and the spare carries the load. Normally, a subsystem and its spare run in close synchronization.

When a repaired subsystem is returned to service, an interrupt informs the CPUs that it is now available. The repaired subsystem is then brought back into synchronization.

The scheme requires no recovery from a fault, since work proceeds using the spare subsystem. To the user, the machine appears to be a conventional machine, requiring no special programming considerations.

A1.4.3 The Resilient System [12]

The system uses two computers, both running the same software, but with only one active. Central to the fault-tolerant nature of this system is the reconfiguration monitor that operates as a number of discrete tasks. At the base of the reconfiguration monitor is the kernel task, which monitors the current machine configuration and communicates with the other processor.

During normal operations, the kernel sends reassurance messages to the other CPU, and receives reassurance messages from it. When a reassurance message is not received by the back-up computer, the kernel takes control of the other system's applications.

The monitor runs in both computers. On detection of a failure in the other system, the monitor activates bus switches to take over peripherals from the failed system, and performs actions to load and restart the failed application or system.

A1.5 General Purpose Systems

A1.5.1 The Intel 432 System [31][5][32]

The iAPX 432 system is designed for large-scale real-time control and transaction processing. The system is based on quad-modular redundancy.

Very large scale integration (VLSI) replication is used in the system, to construct a family of configurations which cover a range of fault-tolerance levels. Fault detection and recovery are performed in the VLSI components. No additional logic or diagnostic software is needed.

In an iAPX 432-based system, there are three steps in responding to an error:

- error confinement
- error reporting
- error recovery

Error confinement. (See fig. A1-11)

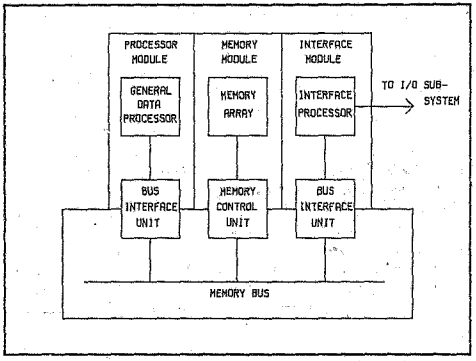


FIGURE A1-11 - Error Confinement Areas in the 432 System [5]

When an error is detected, it is confined to one of the system building blocks. Recovery and repair strategies are built around the block's replacement. When a module or bus has its confinement mechanisms activated, it becomes a self-checking unit. Detection mechanisms reside at every building block interface, and all data is checked as it flows across the interface between confinement areas.

The confinement areas are enforced by applying five different detection mechanisms:

- duplication - when more than one version of a data element is available, comparison is possible

- parity

- Hamming coding

- time-outs - data must be available within a certain time

- loop-back checks - suspected errors can be checked by trying the operation again

Error Reporting.

Upon detection of an error, a message is broadcast to all the nodes in the system, identifying the faulty confinement area, and the type of error. This message prevents other nodes from using the faulty data, and provides recovery information.

Error messages are broadcast over a set of serial busses that are independent of the busses used during normal operation. The error message busses are subject to the same fault-tolerance principles used in the rest of the system.

Error Recovery.

Each node reads the error report, and decides on the action to be taken.

Five redundancy mechanisms exist in the system:

- retry

- single-bit error correction

- shadow modules

- back-up busses

- spare memory bits

Single-bit memory errors are corrected by the memory control unit.

To guard against permanent faults, every self-checking module may be paired with another self-checking module of the same type. The pair of modules operate in lock-step synchronization, and provide back-up for all state information.

Such a configuration results in quad modular redundancy, because there are four identical units - two self-checking modules, each with a master and shadow.

Each memory bus may be paired with another memory bus. Both are used in normal operation.

If a permanent fault is found, the redundant resource is switched in to replace the failed unit. No centralized element controls the switch; each node knows which other node it is shadowing, and when that node is identified as faulty, the back-up node becomes active and takes over the operation.

After recovery is complete, hardware informs the software of the error and subsequent recovery actions. The system software then decides on the optimum configuration:

- full capability and fault-tolerance retention, using a spare

- decreased capability, but full fault-tolerance, by switching out the the faulty pair

- full capability, but decreased fault-tolerance, by using the shadow on its own

Levels of Fault-tolerance. (See fig. A1-12)

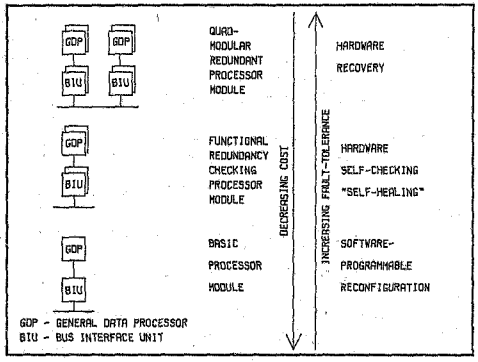


FIGURE A1-12 - Levels of Confinement in the 432 System [5]

The cheapest reconfiguration relies on the built-in fault-detection mechanisms such as bus parity, error-correcting codes across memory arrays, and access retry. Downtime is required, when a fault is detected, to reinitialize the system, run a diagnostic program to locate the fault and reconfigure the system to disconnect the faulty module. The system will function with a lighter load until a replacement for the faulty module is available. The data base may be corrupted unless special software techniques are used.

At the other end of the spectrum, the self-healing system uses functional redundancy checking to detect errors as they occur, and to limit any detected error to the confinement area in which it occurred. Normal program flow is then interrupted

for reconfiguration. Recovery time may be as long as a few seconds.

In 432 systems, the user is responsible for making I/O channels and controllers sufficiently secure, and for providing reliable clocks and power supplies.

A1.5.2 The Basic Fault-Tolerant System (BFS) [11]

The BFS is intended to find application in process control, transaction processing and remote equipment.

The Concept of the BFS.

A number of objectives were to be met by the BFS:

- Hardware faults only were to be tolerated

- Deterioration of system performance was permissible, but the system was to remain in working order for as long as practicable

- The redundancy inherent in a multi-microcomputer system was to be utilized, to minimize the need for extra redundancy

- Mutual monitoring by microcomputers was to be used

- Standard components were to be used

- A simple network structure was to be used

Hardware Structure of the BFS.

The BFS is a multi-microcomputer structure, in which the individual microcomputers are loosely coupled.

Each module has a private main memory, and includes interfaces for peripheral devices. There is no common memory. Functions can be delegated to any module that has the necessary resources.

The modules are arranged in a ring structure, with each module connected to its neighbours, and its neighbours' neighbours (See fig. A1-13). The data lines are bi-directional, and serial.

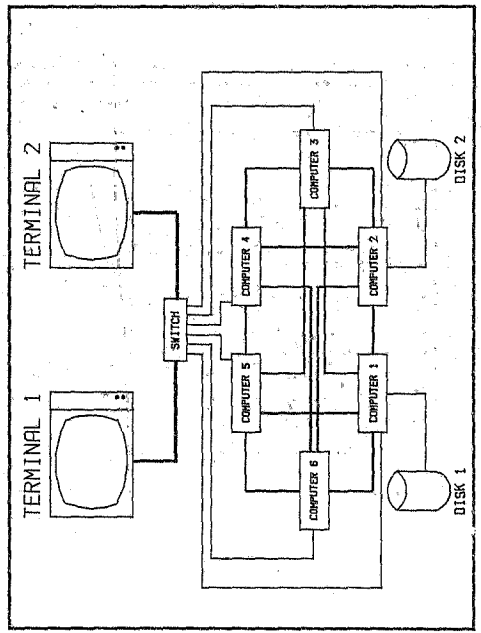


FIGURE A1-13 - The Structure of the BFS [11]

Modules each have a separate power supply, and use a six-card Multibus system.

Software of the BFS.

Each node has an individual operating system that makes it independent of the other nodes. Since the node hardware is identical, the same software can be used in them all (See fig. A1-14).

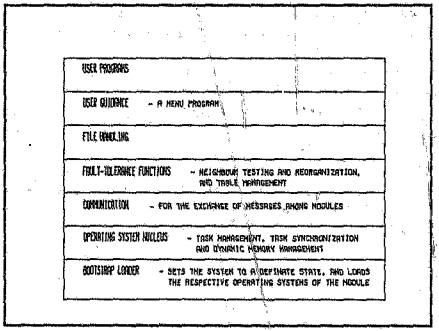


FIGURE A1-14 - The Software Structure of the BFS [11]

A1.6 Summary

Numerous systems apply fault-tolerance techniques, especially in the fields of on-line transaction processing and control of machinery.

An important system is the SIFT system, which is intended for aircraft control. The system achieves improved reliability by having each iteration of the task executed by a number of

processing modules. A processor requiring the results of any task determines the value by obtaining the results from each processor which executed the task, and performing voting.

Also important is the FTMP system, which is again intended for aircraft control. An hybrid triple-modular redundant structure is used, with voting being accomplished in hardware.

Other notable systems include the Triplex 32 programmable controller, the Agusta 129 flight control system, and the Fault-tolerant Array Signal Processor.

Notable transaction processing systems are the Tandem system, the Stratus system, and the Resilient system, while the Intel 432 system, and the Basic Fault-tolerant System are prime general-purpose systems.

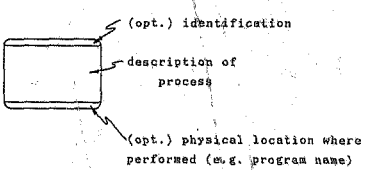
The study indicates that no dominant fault-tolerance technique has emerged. Some systems make use of replication, and others, back-up. Furthermore, there is no outstanding configuration within these techniques. It is therefore essential that systems be carefully designed for the particular application.

This appendix expands on the description of the data flow method of program development, given in Chapter 3.

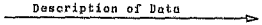
A2.1 Data Flow Diagrams

The basic tool for utilizing data flow is the data flow diagram. Five symbols are used [16], [19]

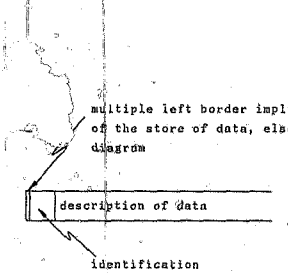
- rounded rectangles for functions (processes which transform flows of data)



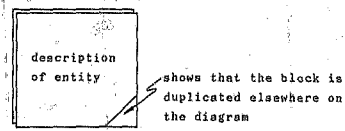
- arrows for flows of data



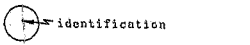
- open-ended rectangle for stores of data



- squares for external sources or sinks of data



- small circles for off-page connectors



Guidelines for drawing data flow diagrams are [16]

- identify the external entities involved

- identify the scheduled inputs and outputs involved

- identify the inquiries and on-demand requests for information that could arise

- using a large sheet of paper, place the primary external entity on the left-hand side and draw the data flows that arise, the processes that are logically necessary and the data stores that are probably required

- draw the first draft freehand, including everything except errors, exceptions and decisions

- check back with the list of inputs and outputs to see if everything is included. Note the exclusions

- produce a clearer second draft, using a template. Check and note exclusions

- get a user to examine the draft

- produce a lower-level explosion of each process. Include errors and exceptions. Incorporate changes in the top-level diagram

- produce a final top-level diagram

A2.2 Data Dictionaries

A data dictionary is a store of descriptions of all entities in a system.

Data dictionaries can provide

- ordered listings of all entries or various classes of entry, with full or partial detail
- composite records
- cross-referencing ability
- names from descriptions
- consistency and completeness checking
- generation of machine readable data definitions
- extraction of data dictionary entries from existing programs

Data elements are the smallest useful pieces of data. Data structures are made up of data elements and other data structures. Data flows and data stores are data structures in motion and at rest respectively.

Typical data dictionary contents are

Data element descriptions
- name
- description
- optional - aliases for the data element
  - related data elements
  - range of values and meaning of values
  - data length
  - encoding (ASCII, binary etc.)
  - other editing information

Data structure descriptions
- name
- description
- construction in terms of other entities:
  - optional entities - [xxxxxxxx]
  - alternate entities - {xxxxxxx,yyyyyyy}
  - iterations of entities - xxxxx * (0 - 10)
    possible numbers
    of iterations

Data flow descriptions
- name
- names of data structures that pass along it
- source and destination of data flow
- volume of flow
- physical implementation if available

Data store descriptions
- name
- names of data structures stored in it
- input and output data flows

Processes
- name
- names of inputs and outputs
- summary of the logic
- reference to the full functional specification

External entities
- name
- names of associated data flows
- if person(s) - Number
  - Identification
- if another system - Language
  - Hardware
  - Information source

Glossary entries
- description of esoteric terms

A2.3 Data Stores

It is usually easier to change the logic of a process than to change the structure of a data base. However, it is possible to refine data bases to their simplest form.

A2.4 Deriving a Structured Design from the logic model [16]

There are three main design objectives

- performance - in throughput, run-time, or response time
- control - security
- changeability - for enhancements and post-production debugging

The principle that the essential form of a program piece is

1 - Get input

2 - Perform transformation

3 - Produce output

is adhered to.

To get from a data flow diagram to an hierarchical structure, one starts with the rawest form of input and traces it through the data flow until the point is reached

where it can no longer be said to be input. Likewise, the output is traced back into the system until it can no longer be thought of as output. The middle piece of the system forms the transformation section.

Tree structure diagrams are used to represent the hierarchical system.

This appendix expands on the techniques of program debugging, which were briefly covered in Chapter 3.

A3.1 Guidelines While Writing Code

- avoid using GOTO
- design modular programs
- program for clarity and optimize later
- avoid system-dependent code
- produce good documentation

A3.2 Assumptions in Debugging

It is necessary to ensure that the bugs which show up on the execution of application programs are present in the program itself. Therefore it must be proved, or assumed that

- the computer is never at fault
- the system software is never at fault

A3.3 Debugging Practice

- keep a written record of debugging
- save a copy of the program before the latest fix
- make sure the listing is current
- test multiple variations of the program, each with only one change in it
- learn from negative results

- watch out for 1,I,l and 0,O

- be aware of the particular problems of a language

- rewriting may be faster than debugging

A3.4 Debugging Techniques

Techniques for eliminating visible errors

Indirect aid
- run the program
- look at the documentation
- browse through the debugging notebook

PRINT statements
- unconditional
- conditional
- to screen, printer or disk

Breakpoints.

Snapshots. - a printed record of the state of the machine

Finding hidden errors and verifying program correctness

Force and check with simple data.

Anticipating future errors (preventative debugging)

Sleeping debug instructions. - are invoked only when abnormal conditions occur

Firewalling. - check data before leaving modules

This appendix elaborates on the techniques of self testing in computer systems.

A4.1 Introduction

Errors in a computer system may be caused by [33]:

- Algorithm faults
- Programming mistakes
- Component faults

Test programs, which exercise the computer, are useful in detecting the failure of a component. Algorithm faults and programming mistakes must be removed using conventional debugging techniques.

Testing of microprocessors is made difficult because the working details of the microprocessor are not usually available to the user. For testing, therefore, the microprocessor is generally split up into functional units. An additional hindrance to microprocessor testing is the fact that many data and control paths within the machine are not directly accessible.

Built-In Self-Test (BIST) can be defined as the capability for a product (chip, multi-chip assembly or system) to test itself, with input stimulation or output evaluation, or both, being integral to the product and not requiring external test equipment [34].

The problem of self-testing is compounded by the fact that the instructions used for testing may themselves be faulty.

A4.2 Testing Theory

A4.2.1 Introduction

The models described in this section are used in the functional testing of microprocessors [35].

For test generation purposes, the microprocessor is represented in terms of its functional units. Then a fault model is developed for each of these functions.

A4.2.2 Instruction model

An instruction I is composed of a sequence of microinstructions $\{m_1, m_2, \ldots, m_k\}$. Each microinstruction is made up of microorders $(u_{j1}, u_{j2}, \ldots, u_{jq})$ which are executed in parallel.

k is the number of microinstructions in instruction I and q is the number of microorders in microinstruction j.

A4.2.3 Microprocessor model

A microprocessor is represented by a graph, which consists of nodes representing

- a register, or
- a set of equivalent registers, or
- the special nodes IN or OUT

IN represents the source of all control/data input, while OUT represents the sink of all control/data output.

The nodes are connected by directed edges if and only if there is an instruction which causes the transfer of data from one node to the other.

A4.2 Testing Theory

A4.2.1 Introduction

The models described in this section are used in the functional testing of microprocessors [35].

For test generation purposes, the microprocessor is represented in terms of its functional units. Then a fault model is developed for each of these functions.

A4.2.2 Instruction model

An instruction I is composed of a sequence of microinstructions $(m_1, m_2, \ldots, m_k)$. Each microinstruction is made up of microorders $(u_{j1}, u_{j2}, \ldots, u_{jq})$ which are executed in parallel.

k is the number of microinstructions in instruction I and q is the number of microorders in microinstruction j.

A4.2.3 Microprocessor model

A microprocessor is represented by a graph, which consists of nodes representing

- a register, or
- a set of equivalent registers, or
- the special nodes IN or OUT

IN represents the source of all control/data input, while OUT represents the sink of all control/data output.

The nodes are connected by directed edges if and only if there is an instruction which causes the transfer of data from one node to the other.

Registers are equivalent with respect to an instruction set if and only if any instruction which uses one of the registers could use another.

A4.2.4 Fault Models

The microprocessor is decomposed into the following units:

- register decoding function
- data transfer paths
- arithmetic and logic unit
- instruction sequencing and control function

Fault model for the register decoding function.

f (R_i) = the register decoding function d
= R_i in the fault-free case
= R_j or 0 or {R_i ,R_j} in the faulty case

Fault model for the data transfer function.

Under a fault

- any number of data transfer lines can be stuck at 0 or 1
- any pair of lines can be coupled

Fault model for the data manipulation function.

The model depends on the architecture of the microprocessor. The test set consists of instructions to transfer the data from memory to the source registers, instructions to perform the operation under test, and instructions to store the result from the destination to memory.

Fault model for the instruction sequencing function.

Under a fault, one or more of the following occurs

- one or more microorders is inactive
- microorders that are normally inactive become active
- incorrect microinstructions are active

A faulty instruction can be represented as

$F(I) = I + d^+ - d^-$

where d^+ is the set of extra microinstructions

d^- is the set of lost microinstructions

A4.3 Test Procedures

A4.3.1 Introduction

There are three types of routine used in the testing process

- test routines, which detect errors
- diagnostic routines, which locate faults
- executive routines, which control the overall process

Some parts of the machine to be tested must be operative in order to perform any program tests. In particular, once it is verified that the read instructions are fault-free, the correct functioning of the remaining instructions can be then tested by first reading codewords into the internal registers, followed by executing the instructions and reading all the registers.

A4.3.2 Error detection

To detect the presence of faults, it is essential that the effects of the faults show up as erroneous data.

Built-In Checks [36]

Such checks use special hardware and/or codes incorporated into the computer or system and are normally designed to detect a certain class of error only. The chief forms of this type of check are

- data transfer checks e.g. parity, Hamming etc.

- data storage checks e.g. memory duplication

- arithmetic checks e.g. register overflows, out-of-range checks etc.

Programmed Checks

The selection of programmed error detection and correction methods must be based on several factors [33]

- the probability of each type of malfunction and its detection and correction by a particular technique

- the probable damage produced by different malfunctions

- the cost of additional storage and increased computing speed requirements

The programmer must establish tolerances on allowable discrepancies, and the degree of confidence to be placed in a particular type of check. He should also decide on the frequency with which the various checks are to be applied.

Four testing methodologies have been applied [37]

- exhaustive testing - All combinations of data pattern and instructions are tested. This gives 100% fault coverage for all non-redundant faults, provided that they do not make the system sequential. The test time of this method is long, and grows exponentially as the width of the data flow increases.

- random pattern testing - A randomly chosen subset of the exhaustive test is used, sufficient to give an acceptably high fault coverage. However, prediction of fault coverage is difficult, and many structures are not random-pattern testable.

- pseudo-exhaustive testing - The method seeks to partition the system logically rather than physically, and then to exhaustively test each partition. This allows broader descriptions and models. However, the partitioning may be expensive to accomplish.

- hybrid testing - Logic can be divided into a number of cones, each consisting of all the logic that feeds a single output. It is not necessary to test the entire system, only each individual cone.

While the tests that exercise the machine are being carried out, it is useful to assign unique codewords to each register. Before the test is run, the registers are loaded with their codewords, and when the test is complete, the registers are checked to see that they are as they should be.

Certain basic instructions must be operative in order for tests to be carried out. These are

- memory to register transfer

- register to register comparison

- conditional branch

The instructions allow the loading and comparison of data, and the branching to an error handling routine if there is an error.

Comprehensive test sequences will utilize every command in the machine, and exercise every accessible element in the computer. The blocks of code can be scattered throughout the memory to increase the probability that an error will cause a transfer to an illegal memory location which will can be filled with an HALT instruction [38].

The test sequences perform :

- command tests - All commands with all possible instruction options are tested. Two methods are possible

(1) The result of an operation is obtained in two different ways; once using the instruction to be tested, and once using a sequence of other instructions that simulate that instruction. Different pseudo-random numbers are used as test data.

(2) Instructions are repeated with different data, and the results are compared with stored, precomputed answers.

- memory tests - A block of words is copied to a temporary location and the two blocks are compared. Next, test data are written into the block, and into another temporary location. The blocks are compared and then the original data is returned to the tested block.

- accumulator tests - The accumulator is caused to count by ones, and the end result is compared with the correct answer.

- addressable device and register tests - Specialized I/O test routines are used. The main machine must be operating correctly before these tests can be made.

- bus tests - To effectively test a bus, it is necessary to drive the bus from each possible register in turn, while collecting a signature in all receiving registers.

Tests may also be built into the application programs [36]. The basic procedures are [33]

- recomputation by the same, an inverse, or a different process and comparison of the results.

- tests to see if the results satisfy certain mathematical or physical criteria

- checks based on estimates of behaviour

- special checks for a particular process or machine

Careful consideration must be given to the number of operations to be covered by a single check. Tolerances should be provided because of truncation and roundoff.

Acceptance Checks [26]

Acceptance tests are designed to ensure that the system is fully operational and that all the system facilities are complete and working before acceptance of the system by the customer.

The acceptance procedure is generally based on a test cycle, comprising both engineering test programs and operational programs.

A4.3.3 Error Logging and Control

While there are no errors, the test sequences are repeated a specified number of times. Pseudo-random numbers are generated for test data. Status messages are printed. These messages typically specify the sequence just run successfully, the sequence which will be run next, and the con-

tents of the registers. Major errors will usually result in the tests being run in the incorrect order, or at erratic times.

When an error is detected, the error routine adds to an error count and outputs an error message. The test sequence is repeated to indicate whether the fault is transient or permanent. If two successive tests show the same error, then the fault is considered permanent. Otherwise it is transient, and the application program can continue.

A4.3.4 Error Diagnosis

A diagnostic program is usually employed to locate the source of an error once it is known to exist. It is useful to have separate diagnostic programs for different parts of the system. The location of a fault results in a record being made of the error's probable cause.

Diagnostic tests are not generally useful to locate faults in certain important control circuits or in the power supply.

A4.4 Preventative Maintenance

Preventative maintenance should be applied to systems, in addition to their self-testing. This maintenance will include marginal testing, where components of the system are driven at the extremes of their specifications. Such testing can show up the imminent failure of an ailing component, thereby allowing repairs to be made before errors are caused.

This appendix provides full details regarding the hardware used to implement the experimental system.

A5.1 The Single Board Computers [43]

A5.1.1 Overview

Typical uses include

i - multiprocessing single board computer

ii - mass storage front-end processor

iii - stand-alone single board system

A5.1.2 Functional Description

The SBC is functionally partitioned into six main sections

- central processor

The 80186 combines DMA, interval timers, clock generators and a programmable interrupt controller into one chip.

The 80186 instruction set includes all the 8086 instructions, while adding
block I/O
enter and leave subroutines
push immediate
multiply quick
array bounds checking
shift and rotate by immediate
pop and push all

The 80130 component provides the iRMX 86 nucleus primitives, timers and programmable interrupt

control. This adds to the 80186 instruction set, providing 35 operating system primitive functions, and supporting five new data types.

The 80186 provides three 16-bit programmable timers. Two of these are connected to four output pins, and can be used to count external events, time external events, generate non-repetitive waveforms, etc. As shipped, the timers provide baud rate generation. The third timer is not connected to output pins, and is useful for real-time coding, time-delay applications, prescaling the other timers, or as a DMA request source.

The 80130 provides three more 16-bit programmable timers. One is used as a baud rate generator, another as a system timer, and the third is reserved for use by the 80130's iRMX nucleus.

- memory

There are eight JEDEC 28-pin memory sites on the SBC. Four of the sites are EPROM sites, and four are RAM sites. The use of an expansion module will add four more RAM sites. Memory device type and size are jumper-selected.

The four EPROM sites are top-justified in the 1Mbyte address space, and must contain the power-up instructions. The four RAM sites are, by default, located starting at address 0.

Power-fail control and auxiliary power are provided for the protection of the RAM sites.

- small computer system interconnect (SCSI) peripheral interface

This interface consists of three 8-bit parallel ports, which may be configured as general-purpose I/O, SCSI or DMA-controlled centronics-compatible line printer interfaces.

- serial I/O

Two programmable communications interfaces are provided. 80186 timer outputs are used as software-selectable baud rate generators.

The mode of operation (asynchronous, byte synchronous or bisynchronous protocols), data format, control character format, parity and baud rate are under program control.

- interrupt control

27 on-board vectored interrupt levels, to service interrupts generated from 33 possible sources, are provided.

The 80186, 8259A, and 8274 PICs act as slaves to the 80130 master PIC.

The highest priority interrupt is the non-maskable interrupt (NMI) line, which is connected directly to the 80186 and is typically used to signal catastrophic events.

- MULTIBUS bus expansion

The SBC provides the [illegible]S system bus, which supports 8- and 16-bit SBCs and peripherals, with 20 or 24 address and 16 data lines

iLBX bus - a local bus extension, allowing on-board memory performance with off-board memory

iSBX bus - multi-module on-board expansion, allowing additional I/O functions to be added

A5.1.3 Operating System Support

iRMX 86 Release 6 can be used, or CP/M firmware can be supplied using the 80150.

A5.1.4 Development Environment

Using iRMX 86 Release 6, software development can be performed directly on the SBC, or by using one of the compatible development systems.

A5.2 The DT732 input/output board [39]

A5.2.1 Description

The I/O board is memory mapped, with a jumper-selectable base address over a 20-bit address range. A block of 16 contiguous address bytes, starting at the base address, provides access to various control, data and channel address registers required for programming purposes.

Analog input features

- 32 single-ended or 16 differential channels
- sample and hold circuitry
- fast A/D converter
- software programmable gain amplifier (gain = 1, 2, 4 or 8)
- jumper-selectable full scale input ranges (+/- 10 V, +/- 5 V, 0-10 V, 0-5 V)

- 4-20 mA current loop output

- +/- 0,06% non-linearity over full scale range

- 24 000 samples per second

A crystal-controlled "pacer" start clock is available for generating any one of ten jumper-selectable time periods from 1 ms to 1 s for synchronizing A/D conversions.

A variety of sampling modes is available

- repetitive single channel

- random access

- channel scan

Sampling may be triggered in three ways

- programmed I/O

- on board pacer sample timer

- external trigger

Separately enabled interrupts caused by either the end_of_conversion or end_of_ [illegible] conditions can be routed to the MULTIBUS.

Analog output features

- 2 channels

- 12-bit resolution

- jumper-selectable output ranges (+/- 10 V, +/- 5 V, 0-10 V, 0-5 V)

- 4-20 mA current loop for each output

Software

The I/O board operates under iRMX 80.

A5.2.2 Specifications

NOTE: References are to the handbook for the board [39]

Analog Inputs

input impedance - power on: 100 Mohm (min.)
- power off: 1,5 kohm (min.)
max input voltage without damage - power on: +/- 35 V
- power off: +/- 20 V
sample and hold - aperture time: 10 ns
- uncertainty time: 20 ns
- linearity: +/- 1/2 LSB
system non-linearity - gain = 1: 0,05% FSR +/- 1/2 LSB
- gain = 2,4,8: 0,07% FSR +/- 1/2 LSB
common mode voltage range: +/- 10,24 V
CMRR: 60 db
external trigger: TTL Schmitt trigger, edge sensitive
triggers on the falling edge
+5 V to 0 V for 200 ns (min.)
50 ns rise time (max.)
digital output encoding (jumper selectable):
- bipolar: offset binary or 2's compliment
- unipolar: straight binary
- current loop: straight binary

Analog Outputs

slew rate: 10 V/ us (no load capacitance)
settling time: 4 us to +/- 1 LSB
accuracy: +/- 0,05% FSR

output current: +/- 5 mA at +/- 10 V
short circuit protected
digital input encoding: straight binary
2's compliment
offset binary
output - impedance: 0,2 ohms
- capacitance: 1000 pF (max.)

User Connections

Output and input channels both use 50-pin edge connectors, with 2,54 mm spacing. One is used for the outputs, and two for the inputs.

(The connectors are 3M 3415-000 or TIH 312125)

Cables are flat or twisted pair.

(EP 036 or EP 086)

A5.2.3 User selectable Parameters

Parameter	Relevant Figs	Notes
Base address remove jumper 200-201	3.1 3.2 3.3	For 16-bit addresses,
A/D input range and configuration, and output coding	3.4 3.5 3.6	When the input range is changed the A/D converter should be recalibrated. For 4-20 mA inputs, the input must be differential. User-supplied 250 ohm 1/4 W +/- 2% 0,5 ppm/degree C shunt resistors should be used.

Parameter	Relevant Figs	Notes
D/A output range and input coding	3.7 3.8 3.9 3.10	Simultaneous voltage and current outputs are possible. Whenever the range is changed, the D/A converter should be recalibrated.
Internal pacer clock period	3.11 3.12	When an external trigger is used, the pacer clock must be disabled by disconnecting pin 40 from any other pins and connecting it to pin 39
Interrupts	3.13 3.14 3.15 3.16	INT A and INT B may be set to trigger at EOC, EOS or upon timeout of the pacer clock.
Transfer acknowledge delays	3.17	It is sometimes necessary to delay this signal match the XACK/ to the host to the computer timing
INH1/ and INH2/	3.19	If the I/O board occupies the signals same memory space as RAM or ROM, it may be configured to inhibit RAM (INH1/) or ROM (INH2/) when the I/O board is accessed
Factory Defaults	3.20 3.21 3.22 3.23	

A5.2.4 Input and Output Connections

Input Connections

Analog input connection schemes

- single-ended inputs - refer to figures 4.1 and 4.2
  - common-mode rejection is lost
  - use high level inputs (> 1 V FSR)
  - use short lead lengths (< 1 m)
- differential inputs - refer to figure 4.3
  - common-mode rejection is obtained
  - low-level inputs may be used
  - lead lengths are not restricted
- current loop inputs - refer to figure 4.4

Output Connections

For simultaneous voltage and current output, the jumpers must be set for current output. The current loop connects to "DAC x I out" and "DAC x I Rtn"; the voltage output connects to "DAC x V out" and "DAC x Ana Gnd". Only the current is calibratable. Refer to figure 4.5.

If the on-board dc-dc converter is to be used as power supply to the current loop, " + 15 V out" must be connected to "DAC x [illegible] V+".

A5.2.5 Architecture and Programming

The address base is set to 0F700H when shipped.

Memory Address Assignments

Refer to figure 5.1.

Analog Input Registers

Register	Relevant Fig
command	5.2
status	5.3
start channel and gain	5.4
last channel address	5.5
clear interrupts	5.6
ADC data	5.7

Analog Output Registers

The lower byte must be written first. The conversion takes place when the higher byte is written.

Analog Input Function

- random channel input - write to the start channel and gain register

  write to the command register

  conversion takes place

  subsequent channels are selected by a write to the start channel and gain register, before a read of the last converted channel

  a read of the high byte of the last converted channel enables the next conversion

- repetitive single channel conversion

  write to the start channel and gain register

  write to the command register

  reading the high byte enables the next conversion

- sequential input scan

  write to the start channel and gain register

  write to the last channel address register

  write to the command register

  reading the high byte enables the next conversion

(check that the "A/D conversion done" bit is set)

when the EOS bit is set, reading the high byte resets it

- A/D converter trigger

Unless an external trigger is specified, A/D conversions are started automatically by a software trigger whenever a data word is written, enabling the conversion.

Testing of the board is described in appendix 6.

A5.3 The Modular Servo System [40] [41] [42]

A5.3.1 Introduction

The servo system is a velocity lag position control system with a d.c. error channel using potentiometers, with provision for inserting linearizing networks to simulate a single time-constant system.

It is used for demonstrating and teaching automatic control techniques to students and technicians. It comprises modular units for individual study and for the construction of speed and position controls using d.c. error signals.

A5.3.2 Closed Loop Systems

Since a closed-loop system is "error operated", it contains the facility to compensate for any departure of the output from the required condition set by the input, since this departure changes the error causing a correcting signal to be applied to the forward path. Oscillations may set in, however, causing the system to become unstable.

A5.3.3 Motor Characteristics

a. Operation of the Motor

Armature control or field control can be selected by inserting appropriate links in the servo amplifier.

b. Tachogenerator Calibration

A gear system of ratio 30:1 rotates a low-speed shaft. The rotations of this shaft can be counted and the tachogenerator output measured, giving the voltage/speed ratio, Kg. This should be about 2,5 - 3,0 volts per 1000 r.p.m. of the motor.

A5.3.4 Speed Control of the Motor (Refer to fig. A5-1)

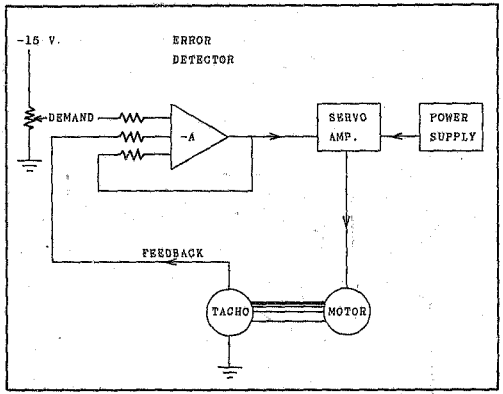


FIGURE A5-1 - Block Diagram of a Simple Speed Control Circuit

The steady-state operating conditions of a ideal system may be represented by

$O = KE$

where $\dot{\theta}$ = motor speed

K = forward path gain

E = error signal

Now $E = V_{ref} - K_g \dot{\theta}$

where V_{ref} = reference voltage

K_g = tachogenerator constant

so

$$\dot{\theta} = K(V_{ref} - K_g\dot{\theta})$$

or

$$\dot{\theta} = \frac{KV_{ref}}{1 + KK_g}$$

If the forward path gain K is large, and K_g is constant, then

$\dot{\theta}$ is proportional to V_{ref}

a. Properties of the Operational Amplifier Unit (Refer to fig. A5-2)

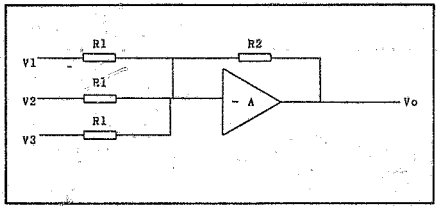


FIGURE A5-2 - The Op. Amp. Circuit

$Vo = -R2/R1\ (V1 + V2 + V3)$
for large A.

A zero setting is provided.

b. Simple Speed Control System

The servo amplifier requires a positive input to rotate the motor. Hence, the voltage applied to the inverting op. amp. must be negative.

The tachogenerator voltage must be connected to oppose the reference voltage.

A5.3.5 Position Control System (Refer to fig. A5-3)

For this purpose, a potentiometer is connected to the low speed motor output shaft.

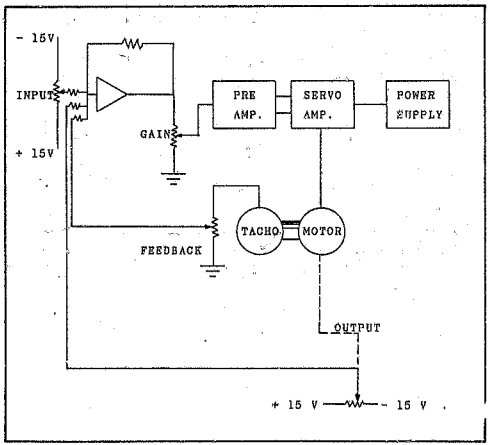


FIGURE A5-3 - Block Diagram of a Simple Position Control Circuit

The volts/degree ratio, Kc, is approximately 1V per 10 degrees, or 0,1 V per degree.

Due to inertia of the motor, the system may overshoot the required position. To reduce the settling time, an output can be taken from the tachogenerator of such polarity as to oppose the error signal.

A5.3.6 Circuit Notes

a. Operational Amplifier Unit

The unit consists of an op. amp. plus input and feedback components. There is provision for external feedback components.

The nominal output is +/- 10 V, +/- 5 mA max.

b. Pre-amplifier Unit

The pre-amplifier unit provides suitable positive-going signals for both inputs of the servo amplifier to enable the motor to be driven in both directions by a signal applied to this unit.

External compensation circuits can be plugged into the unit if required.

c. Servo Amplifier

The armature or field configuration is selected by inserting links in this unit.

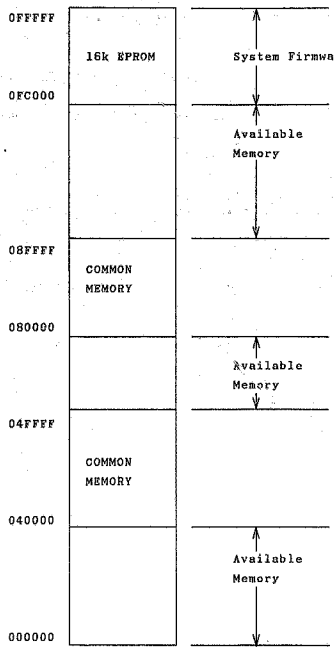
d. Motor/tachogenerator Unit

If the motor is connected for armature control, it should rotate at a current of about 0,9 A.

The tachogenerator ripple voltage is +/- 0,25 V peak at 1800 r.p.m.

This appendix describes the configuration and testing of the I/O board, including its insertion into the memory address space on the Multibus system, and the configuration of the input and output characteristics. Exact connections are given to provide a record of the configuration details.

A6.1. Memory Allocation



A6.2 I/O Board Address

The I/O board uses 12 bytes of contiguous memory, starting at a base address, which is jumper-selected to be on any 16-byte boundary. Since no other peripherals are to be used, 0E7000H was chosen.

A6.3 I/O Board Configuration

A6.3.1 Base Address

0E7000H

20-bit address, so insert jumper 200-201

ADDRx/	Value	Header #
4	0	
5	0	
6	0	
7	0	
8	0	
9	0	B4
A	0	
B	0	
C	1	
D	1	
E	1	
F	0	
10	0	
11	1	B3
12	1	
13	1	

A6.3.2 A/D Input Range, Configuration and Coding

Input range: +- 10 V

Insert jumpers - 58-71
60-62A

Remove jumpers - 58-59
60-61
62B-71

Configuration: differential

Insert jumpers - 23-24A
25-27
51-53B
53A-56
54-55B
55A-57

Remove jumpers - 24B-25
26-27
50-57
51-54
52-55A

Coding: offset binary

Insert jumper - 66-67

Remove jumper - 67-70

A6.3.3 D/A Output Range and Coding

Output range: +- 10 V

DAC 0

Insert jumpers - 3-5
6-7

Remove jumpers - 1-2
3-4
6-8
9A-10A
9B-10B

DAC 1

Insert jumpers - 15-17
18-19

Remove jumpers - 13-14
15-16
18-20
21A-22A
21B-22B

Coding: offset binary

DAC 0

Insert jumper - 43-44

Remove jumper - 42-43

DAC 1

Insert jumper - 46-47

Remove jumper - 45-46

A6.3.4 Pacer Clock Period

Trigger: factory default (1s)

Pins connected to pin 40 - 38

Pins not connected - 29,30,31,32,33,34,35,36,37

Interrupt: factory default (977us)

Pins connected to pin 41 - 28

Pins not connected - 30,31,32,33,34,35,36,37,38

A6.3.5 Interrupts

Int A: not connected
Int B: not connected

Pins not connected - 63,64,65,78,79,80,81,82,83,
84,85,86,87,96

A6.3.6 Transfer Acknowledge Delay

Factory default (0,05us)

Remove jumpers - 74B-76
75B-77
74A-73
75A-72

A6.3.7 Reference Jumpers

Factory default

Insert jumpers - 97-98
48-49

Remove jumper - 98-99

A6.3.8 Memory Inhibit Signals

RAM inhibit: not asserted

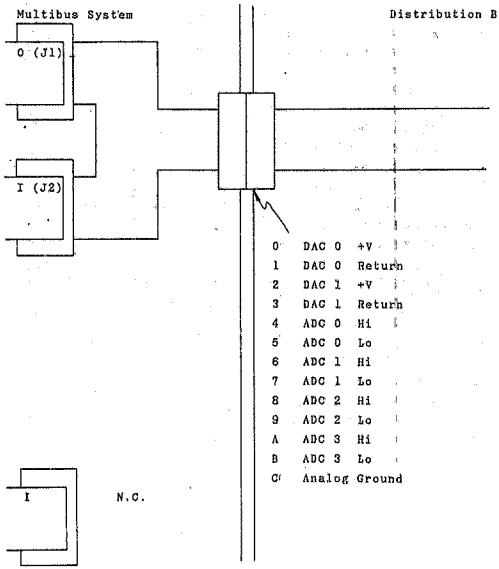
Remove jumper - K3-K4

ROM inhibit: not asserted

Remove jumper - K5-K6

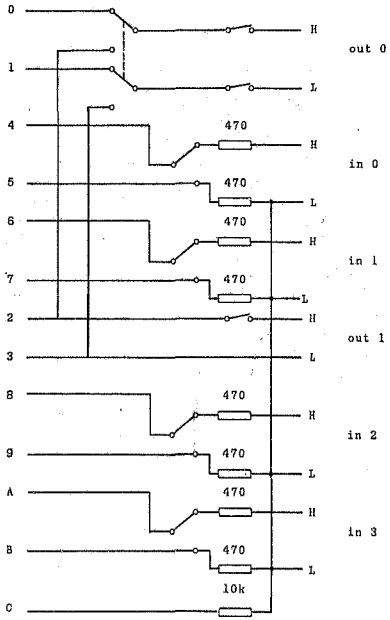
A6.4 Input/Output Connections

A6.4.1 Cabling Connections



Output (J1)						Input (J2)					
1	nc		nc		2	1	nc		nc		2
3	nc		nc		4	3	ana gnd	0	(0H)		4
5	nc		nc		6	5	ana gnd	8	(0L)		6
7	nc		nc		8	7	ana gnd	1	(1H)		8
9	nc		nc		10	9	ana gnd	9	(1L)		10
11	nc		nc		12	11	ana gnd	2	(2H)		12
13	nc		nc		14	13	ana gnd	10	(2L)		14
15	nc		nc		16	15	ana gnd	3	(3H)		16
17	nc		nc		18	17	ana gnd	11	(3L)		18
19	nc		nc		20	19	ana gnd	4	(4H)		20
21	nc		nc		22	21	ana gnd	12	(4L)		22
23	nc		nc		24	23	ana gnd	5	(5H)		24
25	nc		nc		26	25	ana gnd	13	(5L)		26
27	nc		nc		28	27	ana gnd	6	(6H)		28
29	nc		nc		30	29	ana gnd	14	(6L)		30
31	nc		nc		32	31	ana gnd	7	(7H)		32
33	nc		nc		34	33	ana gnd	15	(7L)		34
35	nc	1	V out		36	35	nc		nc		36
37	1 I rtn	1	I out		38	37	nc		nc		38
39	1 A gnd	1	LoopV+		40	39	dig gnd	ck	out		40
41	nc	0	V out		42	41	dig gnd	E.Tr	in		42
43	0 I rtn	0	I out		44	43	dig gnd	EOC	out		44
45	0 A gnd	0	LoopV+		46	45	dig gnd	EOS	out		46
47	ana gnd	Ana	gnd		48	47	ana gnd	ana	gnd		48
49	-15Vout	+15	Vout		50	49	-15Vout	+15	Vout		50

A6.4.2 Distribution Box Wiring



A6.4.3 D-connector configuration

Computer Pin	Computer Line	Description	Wire Colour	Dist. Box Pin
1	1	1 V out	White	3
2	3	1 I out		
3	5	1 Loop V+		
4	7	0 V out	Yellow	1
5	9	0 I out		
6	11	0 H	Black	5
7	13	0 L	Brown	6
8	15	1 H	Green	7
9	17	1 L	Purple	8
10	19	2 H	Blue	9
11	21	2 L	Orange	10
12	23	3 H	Turquoise	11
13	25	3 L	Pink	12
14	2	1 I rtn		
15	4	1 ana gnd	Red	4
16	6	nc		
17	8	0 I rtn		
18	10	0 ana gnd	Grey	2
19	12	ana gnd	Shield	13
20	14	ana gnd		
21	16	ana gnd		
22	18	ana gnd		
23	20	ana gnd		
24	22	ana gnd		
25	24	ana gnd		

A6.5 Testing

NOTE: Extensive reference is made to the handbook for the I/O board [39]

A6.5.1 Method of Calibration

D/A Calibration

Gain Adjust	Offset Adjust
R 16	R 17
R 31	R 32

- assumptions

  software to provide output codes is available

  output cable length < 3 m

  load impedance > 10 kohm

- equipment required

  precision voltmeter - 5 decimal places on the +/- 10 V range

  cables and connectors

- connections

  voltmeter + to DAC output

  voltmeter - to analog ground of DAC

- calibration

allow 1 hour for the system to warm up

refer to figure 7.1

unipolar offset (0-10 V and 0-5 V)

output 00H and adjust for 0 V

unipolar gain

output 0FFFH and adjust according to the table

bipolar offset (+/- 10 V and +/- 5 V)

output 00H and adjust for minus full scale

bipolar gain

output 0FFFH and adjust for full scale - 1 LSB

A/D Calibration

- assumptions

software to provide readings and display them is available

calibration is on channel 0, with all other channels returned to analog common

input cable length < 3 m

- equipment required

precision voltage source

cables and connectors

- single-ended operation - see the figure on page 2

- differential operation - see the figure on page 4

- calibration

allow 1 hour for the system to warm up

unipolar zero

set gain to 1

input 1 LSB and adjust the offset for 1 count

set gain to 8

input 1 LSB and adjust the amplifier offset for 1 count as shown in the table on page 6

unipolar full scale

set gain to 1

input full scale - 2 LSBs and adjust according to the table on page 6

bipolar zero

set gain to 1

input - full scale + 1 LSB and adjust the zero potentiometer according to the table on page 9

set gain to 8

adjust the auxiliary zero

bipolar full scale

set gain to 1

input full scale − 2 LSBs and adjust according to the table on page 6

A6.5.2 Calibration

Analog output required no calibration.

Analog input 0 was tied to analog output 0, and the number of erroneous readings was minimized by adjustment to the offset and gain.

After calibration, the average error rates were

Single-bit errors − 1 in 1833 samples

Multiple-bit errors − 0

A7.1 Introduction

The requirements for the software part of the system were generated by examination of the system functional specifications. This appendix provides the full Software Requirements document, discussed in chapter 5.

Eight major functions are required of the software:

- Task I/O handling
- Self and mutual testing of nodes
- Time-staggered operation
- Watchdog timing
- Error handling
- Task control
- Inter-node communication
- System initialization

A7.2 Task I/O Handling

Task I/O handling is the most important fault tolerance feature of the system. It is expected that this utility will provide the most meaningful system failure-avoidance capability, in that the task I/O handling routines will prevent the production of erroneous outputs, and also inhibit the spreading of errors through the application system software.

A7.2.1 I/O Device Checks

Because the system is intended for use in a control environment, input and output devices are of prime importance. No matter how fault tolerant the computer system is, undetected sensor and servo failures will override its effect and cripple the system as a whole. For this reason, it is important that regular checks on the I/O devices are made, especially when they are about to be used.

For output devices, readback and feedback checking of output channels must be supported, while for input devices, input reasonableness and consistency checks must be supported.

If errors are detected, these must be signalled so that the *operating system* can take appropriate action.

A7.2.2 I/O Request Servicing

In order for the application system to make use of fault-tolerance procedures, I/O request servicing must be provided. This will enable application modules to call operating system routines which perform input or output (as required) in a fault-tolerant manner.

The key aspect of I/O servicing is the application of voting on the data supplied by each node. Errors can thus be eliminated, and faulty nodes pinpointed.

It is necessary that, once a voting operation has been performed on an input to a task, all nodes use the same input value in their calculations; otherwise different answers may be produced, causing confusion in the voting process. (This requirement is based on the proposal that, given the same input data and the same process by which to transform that data, identical machines will produce identical answers).

To provide full facilities for application programs, all types of task I/O, including memory modifications (i.e. RAM output) and memory data input, as well as other peripheral I/O types must be catered for in the I/O request servicing package.

Of particular importance to the target application, multi-input servo device handling must be included.

A7.2.3 I/O Records

Throughout the various operating system sections, good records must be maintained, for two main reasons:

- to enable the operating system to perform fault diagnosis when necessary

- to allow system monitoring at various development phases and for maintenance and repair

As with other operating system sections, I/O records include two parts of the fault-tolerance record structure, namely error records and fault records. Error records will consist of notes of any errors detected, as well as any other information as may be determined. Similarly, fault records will consist of notes of all faults found by the mechanisms.

In addition, the I/O request servicing section of the system must maintain a list of the I/O devices together with their operational status and notes about peculiarities of the devices (such as replication of input devices and dual inputs to output devices).

A7.3 Self and Mutual Testing

Equipment-testing programs are primarily intended to forestall the occurrence of an error by detecting hardware faults at an early stage. The programs can also be used for fault pinpointing once an error has occurred.

Both self testing and mutual testing must be included because of the possibility that the software in a node is corrupted. This corruption, when affecting the self-test software, may cause the node to erroneously report itself fault-free. The testing of a node by another node will detect this situation.

A7.3.1 Test Descriptions

All parts of the system should be exercised by the test programs, including the

- memory components
- processor subsystem
- I/O equipment
- inter-node communication equipment

and any other special equipment.

Fixed data, designed to detect stuck-at faults, and random data, to give as high a pattern-dependent fault coverage as possible, must be used.

Programs should be provided to activate remote tests, and others should be designed to respond to these activations by performing the required test(s) and reporting back via the communications bus.

All routines must, on detection of an error, report this fact to the operating system.

A7.3.2 Test Scheduling

Equipment testing may be scheduled to take place in three circumstances.

When the cpu has no tasks to run, equipment testing must be undertaken to detect hardware faults as early as possible.

If the voting mechanism (or any other mechanism) indicates that a node has erred too often, then the node must remove itself from the operational system and perform self-tests until it has either failed to isolate a fault, or has pinpointed the fault as tightly as the tests allow.

Finally, when the voting mechanism is incapable of identifying the erring nodes after errors have been detected (i.e. when two or three nodes err simultaneously), then all three nodes must cease application execution, and perform self- and mutual-tests.

A7.3.3 Test Records

As before, error and fault records must be updated according to the results of the tests.

Also, records must be kept on the condition of devices, especially whether or not they may be used.

A7.4 Time-staggered Operation

This aspect of the system is key to the effectiveness of the voting mechanism. By ensuring that the nodes do not execute the same instruction at the same time, the probability of two nodes producing the same error due to a transient fault is made insignificant. Hence the voting mechanism can detect the errors.

A7.4.1 Setting up of Stagger

Procedures must be provided by which the relative clock times of the nodes can be set to different values. Also, it must be possible to check the stagger periodically so that clock drift does not move the clock positions too far from or close to one

another. Errors must be brought to the attention of the operating system.

A7.4.2 Time Stagger Records

Any errors uncovered by the time staggering set-up or check procedures should be noted in the error record structure.

A7.5 Watchdog Timing

This part of the system provides extra error and fault detection capability to the system.

A7.5.1 Task Completion Checks

After each task, the node must make available to other nodes the task number (identification) and completion time of the last task. *This data will form the task-complete record for the task.*

Thereafter, the node must check the total task-complete record set (from all nodes) to see if there are any discrepancies. Both task numbers and task complete-times must be checked.

Any discrepancies must be reported to the operating system for appropriate action.

A7.5.2 Timing Records

Again, errors must be noted in the error record structure.

A7.6 Error Handling

The system must provide the facility to preserve system operation when an error has occurred. There should be the means to restore the system to its fault-free state wherever *possible*.

A7.6.1 Node Errors

The first sign of a fault is the detection of a node error, by any of the means included in the system. When this occurs, the operating system must take appropriate action. This action is to be provided by the error handling routines.

If a node has erred only once (or an acceptable number of times), then a number of actions must be taken.

First, the error must be recorded in the error record data base. Secondly, the correct data must be provided to the erring node. Thirdly, nodes must begin voting on which task to execute.

If an unacceptable number of errors has occurred in one node, then more severe action must taken. Firstly, the error record base must be updated, and then examined to see if any conclusions can be drawn about the nature of the fault.

Then the node must be instructed to cease application program execution and to perform extensive self-testing. Once the self testing procedure is complete, the node may then begin execution of application programs if it has sufficient capability. The current schedule list must be provided to the node at this time, by the other nodes. The node may then provide data for voting, as before, but must be ignored until it has produced correct data for an adequate number of successive votes.

A7.6.2 Fault Diagnosis

In order to facilitate fault isolation and to provide data for system repairs and improvement, it is necessary that the system perform fault diagnosis whenever it is suspected that a fault exists (namely when a node is isolated for testing).

This process should be the systematic application of test procedures to isolate faults as much as possible. Once the fault is

pinpointed, an alternative resource may be used in place of the faulty resource if one is available.

A7.6.3 Error and Fault Records

Error handling routines must provide the bulk of the error and fault record manipulation. When an error report is made, the routines must update the relevant records, such as node-error counts, error-type counts etc.

A7.7 Task Control

A most important part of the operating system, task control routines, must be provided to take care of the scheduling and descheduling of tasks, both application tasks and operating system tasks.

A7.7.1 *Maintenance of Scheduling Lists*

Routines must be provided by which application tasks must be able to schedule and deschedule other application tasks. Routines must also maintain order in the lists for ease of task execution initiation.

Normal execution lists will include both operating system routines and application routines, while an exception list must be available when the node is instructed by the system to undergo testing.

A7.7.2 Performance of Scheduling

When no exceptions have been discovered, the normal list is followed, while in exceptional circumstances, the exception list becomes active.

If a node is identified as having erred, then, as a *precaution, the nodes will confer on each task before it is executed.* This will identify a fault if errors are then

detected. Also, this will prevent the execution of an incorrect task. If the cpu is too busy to handle the extra load, then the precaution must be forfeited, and the voting and watchdog mechanisms must be relied on to detect the execution of an incorrect task.

Abnormal conditions detected by any of these routines must be reported.

A7.7.3 Task Records

A record must be made on the execution of each task. This will aid system testing, maintenance, repair and improvement.

Errors must be recorded.

A7.8 Inter-node Communication

Although the communication is to be imitated using common memory, the correct format is to be retained so that in the intended application, little modification is necessary.

A7.8.1 Communication Handling

Routines must be provided by which modules may transmit and receive messages via the communications bus system. These routines must control all communication board-dependent functions, thus simplifying the communication procedure and providing an error interception method.

All errors and faults signalled by the communication board must be reported to the operating system.

A7.8.2 Test Links

It must be possible to establish test links for both communication system testing, and remote test initiation.

A7.8.3 Logging of Bus Activity

Although this function is not part of the system, it is necessary for the demonstration set-up. Procedures must exist which will take care of flags, buffers, etc. that would be handled by the communication board.

A7.8.4 Communication Records

Bus activity should be recorded by bus controllers for the purposes of testing, maintenance and design improvement.

Error and fault records must be updated when necessary.

A7.9 System Initialization

This suite of routines must be executed at power-up. The routines must establish the working environment for the operating system by initializing devices and performing system confidence tests.

A7.9.1 Initialization of Devices

All devices must be initialized.

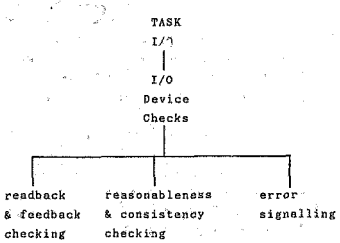
A7.9.2 System Confidence Test

Extensive pre-operation tests must be run. These must include extended versions of the self and mutual tests provided as well as enhanced tests to exercise peripherals, especially servos and sensors.

A7.9.3 Initialization Records

All errors and faults must be reported. Critical faults, which preclude the successful operation of the system, must be signalled to the operator.

This appendix gives the complete software functional specification, which was derived from the software requirement specification. Each section of the appendix starts with a diagram that shows how the described software fits into the system.



READBACK AND FEEDBACK CHECKING

Readback - If the output channel has a readback input channel, then this is noted in the I/O device record structure. The readback checking facility is called when an output is performed, and the application wishes to check that the output channel is operational. The readback facility examines the I/O device records to see if there is a readback input channel, and to get the tolerances associated with the output channel. It then compares the output value requested to the readback value read. If the values are sufficiently close, then the module returns a status of OK. If the values are too far apart, then the module creates an error and a fault record, and returns a status of NOT OK.

Feedback - The routine simply compares the actual feedback values with those that the application tells it to expect. The facility is used as an extra device check when required. The application program must supply the expected value. The feedback routine then obtains a feedback value and compares the two. If the difference is more than a tolerable value, then the feedback check routine returns a status of NOT OK and creates an error and a fault record. If the difference is acceptable, then the routine returns a status of OK.

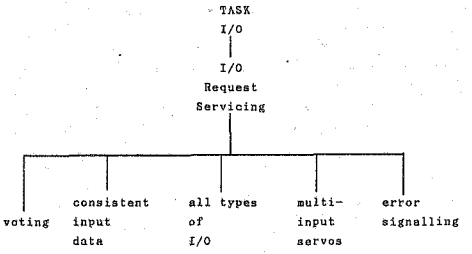
REASONABLENESS AND CONSISTENCY CHECKING

Reasonableness - Upper and lower bounds exist for input device readings. These values are stored in the I/O device records. When an input is measured, the returned value is compared with these values to see if the input is reasonable. If the value is out of bounds, the routine creates error and fault records, and returns a status of NOT OK. Otherwise a status of OK is returned.

Consistency - There is a maximum rate of change possible for each input channel. The value is held in the I/O device data base. A new input value is compared with the previous input value from that device. If the difference is too large, then the routine creates an error and a fault record, and returns a status value of NOT OK. Otherwise, the status value is OK.

ERROR SIGNALLING

Error signalling is accomplished in two parts. First, an error record is generated. This brings the error to the attention of the operating system via the record creation subroutines. Secondly, the application program is alerted by means of the status value. The application can decide on the next suitable course of action.



VOTING

When an application program wishes to obtain an important piece of information, it must use an input routine provided by the system. The application provides an identification of the required information. The system call returns the value and a status value. The routine consults the I/O device records to get the form of the input required. Each node that has a version of the required input stores it, and transmits it over the bus network in a sequence that ensures that stagger is maintained. If an acceptable number of errors is found, then the routine returns the voted value and a status of OK so that the application may continue. An error record is generated if there are errors. If a double (or a triple) error is detected, then the module returns a status of NOT OK, and the returned input is meaningless. An error record is generated. The application must decide on what action to take.

When an application program wishes to perform output in a fault-tolerant manner, it must use an output routine provided by the system. The application provides the data, and the identification of the output path. The system returns a status value. The routine determines the characteristics of the data to be output, from the I/O records. The routine then obtains the other versions of the data from the other nodes, votes, and performs the necessary output action.

CONSISTENT INPUT DATA

Consistency of input data is ensured, since the nodes vote on the same data.

ALL TYPES OF I/O

ROM and RAM - Each node has local ROM and RAM, which hold a version of the data. When an input from ROM or RAM is needed, the routine obtains all versions of the data via the communications bus. The voted version is returned to the application. If the routine finds that the result of the vote is different to the value held in its local RAM, then the correct value is written into the RAM, and an appropriate error record is generated.

When an output to RAM is needed, the routine obtains all versions of the data via the data bus, votes, and writes the result into memory.

Node sensor inputs - Most sensors are not replicated at each node. The I/O device records hold the configuration details. When sensor input is required, the routine checks the I/O device records to find out which nodes can provide input data. If the local node has the required inputs, then the routine reads the data and transmit the value to the other nodes. It also receives the data from the other nodes.

Averaging takes place where necessary, applying a tolerance appropriate to the device (also held in the I/O device records) to decide whether or not the input is valid. When there is only one sensor, it is the responsibility of the local node to pass the data to the other nodes.

Node servo outputs - Most servos are the single-input type, and are associated with a particular node. When output to such a servo is to be made, each node submits a value for the output, the local node [illegible] on the submissions, and performs the actual output. [illegible]i-input servos are present, the inputs are attached to [illegible]rent nodes. Again, each node submits a value, and each node with a physical output performs the actual operation.

Discrete sensor inputs - When the sensors are remote from the computing nodes, they are controlled via the communication bus. In this case, any of the computing nodes has the capability to transmit the control instruction. The current bus master issues the actual command.

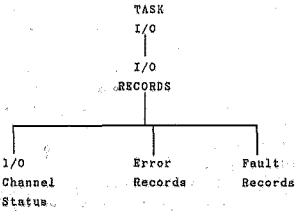
Discrete servo outputs- Similarly, servo control instructions are issued by the current bus master.

MULTI-INPUT SERVOS

Multi-input servos are catered for in the I/O request servicing routines.

ERROR SIGNALLING

Errors are reported to the application by means of status values, and to the operating system via the creation of error and fault records.



I/O CHANNEL STATUS

I/O channel records provide the information necessary for the operating system to perform I/O operations. Most of the information is invariant, and is programmed into permanent storage. The remainder of the data is established at run-time.

The records consist of an identifier, followed by type-dependent permanent parameters.

Memory - A memory record consists of an identifier marking the memory as ROM or RAM, followed by the node address(es).

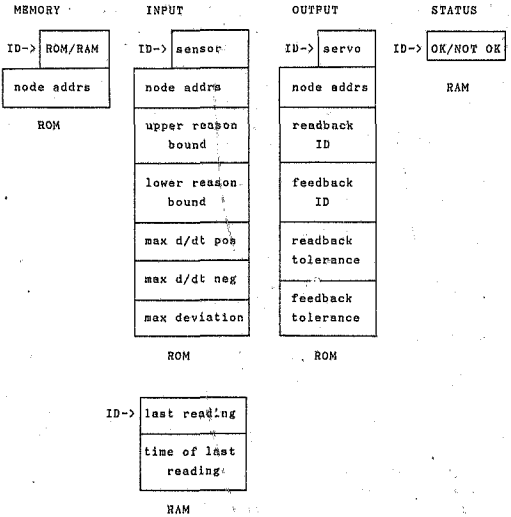
Input - An input device (sensor) record consists of an identifier marking the device as a sensor, followed by the node address(es) of the device, and the local address of the device. (Note that more than one sensor may be available to provide a particular piece of data). Also included are

a maximum positive rate of change and a maximum negative rate of change for the variable, thus defining the consistency limits for the variable. An upper and a lower bound for reasonability are stored so that an error can be signalled should the variable attain a value outside these bounds. An input deviation tolerance is included, which sets down the maximum deviation that is tolerated between like sensors.

Output - An output device record consists of an identifier, marking the device as a servo, followed by the node address(es) of the device, and its local address. The IDs of associated devices are stored next. If the output has a readback input associated with it, then the following information is the input ID. Otherwise it is an invalid value. If the output has a feedback input associated with it, then the following information is the input ID. Otherwise it is an invalid value. If there is readback, a readback tolerance follows, and if there is feedback then there is a feedback tolerance next.

For each input, there is a temporary storage space (i.e. in RAM) where the last measured value is stored, as well as a time when the value was measured. These values are used to determine whether or not the input is consistent.

For every device, there is a status record, showing whether or not the device can be used. This is also in RAM.



ERROR RECORDS

All error records conform to a standard format, so that manipulation of this data is simplified.

The error record must contain the ID of the data which was found to be erroneous, and its node's bus address, the method of detection which exposed the error (error type), and the bus address of the node which found the error.

Error

data ID
error type
node addr

The node address is necessary because data will have the same memory addresses in the final system.

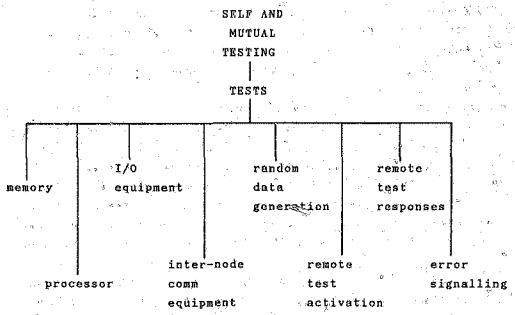
FAULT RECORDS

All fault records also conform to a standard format.

The fault record must contain the ID of the device which is faulty, the method of fault detection (fault type), the bus address of the node which found the fault, and the node-time at which the fault was found.

Fault

device ID
fault type
node addr



MEMORY

Memory testing is a time consuming process. If there is adequate cpu time available, all memory is tested. Otherwise, only representative pieces of each IC are tested. (Total IC failure is more common than individual memory cell failure). Complete memory tests are executed at power-up.

The memory testing procedure is:

1) A block of memory is transferred to a temporary memory location. The copy is checked against the original to see if any error has occurred in the transfer.

2) A sequence of "sliding ones" is written into the block, and read back. This shows "stuck-at" faults, and cross-coupling faults of both the memory and the bus system which provides access to it.

3) Pseudo-random numbers are written into the block of memory being tested, and into another temporary storage area. These data are compared. This test is intended to reveal pattern-dependent faults.

4) The original data is copied back to the memory block, and this is again checked against the duplicate.

If an error is detected at any of the stages, an error and a fault record are produced. The routine is called by the operating system, and returns a status of OK or NOT OK, as necessary, allowing immediate fault handling if the operating system so wishes.

At the end of memory tests, the processor communicates the results with the other nodes, so that they may update their records.

PROCESSOR

It is essential that the processor is operational, or the node is incapable of performing any useful work. The processor is therefore extensively tested before the system is put into operation, and is periodically re-tested to ensure that it remains operational.

The core instructions are tested first, because they are required to test the other instructions. The core instructions are

1) memory to register transfer

2) register to register comparison

3) conditional branch

If any of these instructions fail, then no further testing is possible, and the node is useless. Therefore the processor

sends an appropriate message to the operator, and operation terminates.

If the instructions are executed successfully, then testing of the processor continues. All possible processor commands are tested, using known data and precomputed results.

Power-up testing starts with processor checking; when this is complete, other tests are undertaken. When periodic processor checking is done, a status value is returned in addition to the error and fault records, which are produced whenever necessary.

At the end of the processor tests, the node communicates the results to the other nodes so that they may update their records.

I/O EQUIPMENT

Since the system may be used in different environments, it is possible to add or remove I/O equipment test routines as necessary. Therefore, the core I/O equipment test routine consists of a set of calls to all the appropriate sub-routines, which test the different types of equipment.

The present system requires two types of testing

1) analog input

2) analog output

Analog input - The first checks on the analog inputs are of the I/O board registers. All those registers which may be checked, by reading back, are loaded with appropriate values, and verified.

Next, all input channels are read, and their values checked for reasonableness. The I/O device records are consulted.

Analog output - All output channels with readback and/or feedback capabilities are set to a series of output values appropriate to the devices they control. Readback and feedback are performed, and the values checked against those which are expected.

If the results of the tests show errors, then both the output and the readback/feedback devices are marked as faulty, because it is not possible to distinguish between them with respect to the position of the fault.

Error and fault records are generated when an error is detected. A status value of OK or NOT OK is returned. At the end of I/O testing, the node communicates the results to the other nodes so that they may update their records.

INTER-NODE COMMUNICATION EQUIPMENT

The communication controller board is intelligent, and performs many of its own checks. The result of these checks is returned via a status word in the communication system control block.

On power-up, the board is given a software reset command, and the status is read to see if any faults are present.

When the system is operating, and the communication equipment is to be tested, a NOP command is issued, and the status is read.

Next, a communication link is established between two controllers. Known messages are passed between the nodes; any discrepancies or error reports are noted. Random data is used for the testing. All action commands that can be tested are verified.

Error and fault records are generated when an error is detected. A status value of OK or NOT OK is returned.

At the end of the inter-node communication equipment testing, the node communicates the results to the other nodes so that they may update their records.

RANDOM DATA GENERATION

Random data is used in many of the test procedures.

Pseudo-random numbers with a flat distribution curve are generated, using a seed obtained from an uninitialized RAM location.

REMOTE TEST ACTIVATION

In order to provide mutual testing, a set of routines must provide the necessary actions.

Remote test activation takes place according to a fixed sequence. This allows the establishment of a rendezvous between participating nodes.

The activating node sends a message to the activated node, instructing it to perform a sequence of tests. The activated node responds by performing the requested tests one by one, sending a report message back to the activating node after each test.

The activating node monitors the test results and generates appropriate error and fault records which it also later sends to the activated node so that its records can be updated.

REMOTE TEST RESPONSES

The activated node produces a report message consisting of the results of each test and the node's opinion of the

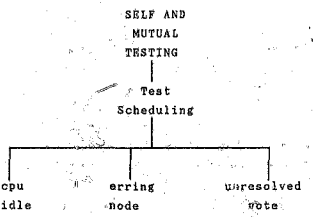
results. This enables the activating node to decide whether or not the activated node is capable of meaningful decisions.

The activated node then accepts any error or fault records from the activating node, performs self-tests if necessary, and creates its own error and fault records.

ERROR SIGNALLING

Self and mutual testing routines are invoked by the operating system. Therefore the returned status messages provide the operating system with the ability to decide on the course of action, according to the results.

Error and fault records are also generated.



CPU IDLE

All free cpu time is used for additional system confidence testing. This allows early detection of hardware failure. Tests are done in a cyclic manner, with as many tests as possible being fitted into the cpu-idle times. At the next cpu-idle time, testing resumes with the next test in the cycle.

For the purpose of fitting tests into free time, a table must hold the WITH-ERROR execution times for each test (i.e. including the time needed to create error and fault records).

The operating system checks on the available time, checks the required time for the next test, and executes the next test if time permits.

Appropriate error and fault records are generated.

ERRING NODE

When the error handling section of the system decides that a node has produced too many errors, it instructs the node to perform a sequence of self-tests. If the errors exhibited by the node are all of a particular type, then it is only necessary for the node to test the appropriate subsystem. If the testing reveals a fault, then the node reports the fault. If no fault is revealed, then the node must remain configured out of the system.

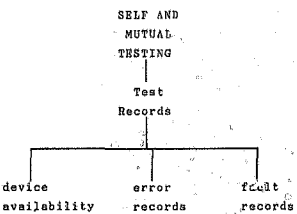
If the detected fault is not sufficiently serious to cause total node failure (e.g. duplicated sensor failure), then the node may be brought back into operation. Otherwise the node is no longer considered part of the system (e.g. in the case of processor failure).

UNRESOLVED VOTE

It would be wasteful to shut down the system after only one unresolved vote. The chances are great that the mis-vote was the result of a transient fault. Therefore, system shut-down *occurs only after an* unacceptable number of mis-votes have occurred.

In such a circumstance, it can be taken to imply that at least two nodes have a fault. Since the operational node cannot be pinpointed by the voting mechanism, *all nodes must perform* self- and mutual-tests in an effort to find the faults. If no faults are found, then the system may continue to operate.

After a further unacceptable number of votes, the system is considered inoperable, and is shut down.



DEVICE AVAILABILITY

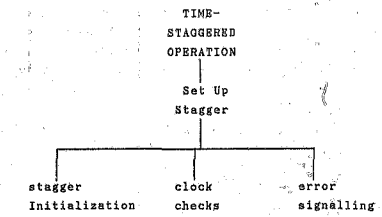
Device status is held in RAM, and is identified by the device ID. If any of the self or mutual tests determine that a device is not serviceable, then the device status is updated.

ERROR RECORDS

Error records conform with the standard format.

FAULT RECORDS

Fault records conform with the standard format.



STAGGER INITIALIZATION

After system initialization, the clocks are read by each node. The clock-time at which each node must reset its clock is determined by adding a set-up time to the read value of each clock. The set-up time is pre-determined, and consists of the time necessary for the calculations, the time differences between the reading of the clocks, the time for communication of the time readings, and a short safety time.

The procedure for each node is there

get all node times	calculate each zero-time	check zero-times with other nodes	wait	set zero time

----------> t

If the zero-time check reveals disagreement between the nodes, then an error record is generated, and the procedure is repeated. Multiple errors indicate a faulty node. The

operator is alerted, and the node is instructed to perform self- and mutual-tests. If these tests reveal serious node faults, then the node is left out of the system. Otherwise, corrective action is made, and the node is overruled in the zero-time vote.

CLOCK CHECKS

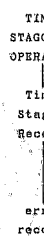
Immediately after stagger initialization, and also periodically, clock stagger checks are made.

The clocks are read and compared. If the difference is acceptable, then no action is taken.

If there is too large a discrepancy in the relative clock positions, then resynchronization is done, using the same procedure as for stagger initialization, but setting the time to an appropriate value. An error record is generated if the clock discrepancy reveals a serious clock error (i.e. greater than the expected drift rate would cause), and a status of NOT OK is returned.

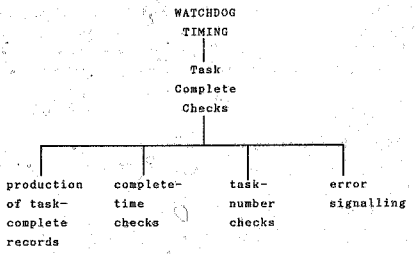
ERROR SIGNALLING

Errors are detected in the clock-check stage, and error records are generated in error conditions. The status value allows the operating system to take appropriate action.



ERROR RECORDS

Error records conform with the standard format.



PRODUCTION OF TASK-COMPLETE RECORDS

When each task is completed, the operating system in each node provides the other nodes with the task identifier and its task-completion time. This is done in a pre-determined sequence via the communication bus. A table is created, using the values obtained via the communication bus and the nodes own values. This table forms the task-complete records.

COMPLETE-TIME CHECKS

Each node, before executing the next task, reads each task-complete time from its task-complete records. The difference between the values is checked against a known maximum difference. If the times are sufficiently close, then no action is taken. If the times are too far apart, then an error record is generated.

TASK-NUMBER CHECKS

As an additional fault detection method, the task numbers of the just-completed tasks are compared. If they are not the same, then an error message is generated. The good nodes send

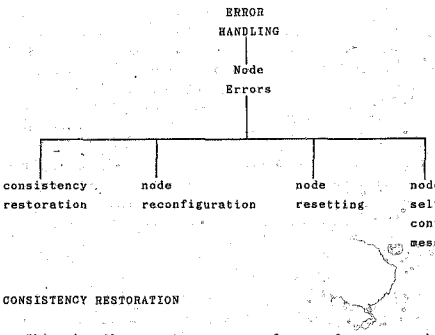
a message to the node which has executed the incorrect task, informing it of the correct next-task to execute. If a second task execution error occurs, then the node is instructed to perform self-validation. Otherwise no action is taken.

ERROR SIGNALLING

Errors are signalled when task numbers or task-completion times are found to be incorrect.

ERROR RECORDS

Error records conform with the standard format.



CONSISTENCY RESTORATION

This is the most common form of error handling. Inconsistencies take the form of incorrect data, which is detected by the voting mechanism. The voting mechanism overrules any single errors, thereby restoring consistency to the erring node.

If a node has executed an incorrect task, then the watchdog section detects this, and the node is informed of the correct next task, thereby restoring task consistency.

NODE RECONFIGURATION

Node reconfiguration is the process by which nodes are instructed to cease application execution, and to perform self-validation.

This instruction results from the detection of an unacceptable number of errors from a particular node, which is noticed in the error record and fault record handling routines.

When a node receives such an instruction, it begins an analysis of the error records which led to the action. Using the data obtained from the error records, the node performs appropriate self-tests of the faulty section. This leads to any existing fault being detected. Then a fault record is created, and the *task I/O table is modified to show* the faulty device.

Next, the node notifies the system that it has completed its tests, and it informs the system of any faults discovered. The other nodes respond by updating their fault records, and supplying the erring node with the current execution list.

NODE RESETTING

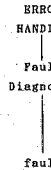
The erring node has been reset once it has performed all the necessary tests and received the *current execution list*.

The node then executes tasks as normal, submitting results in the usual way. If faults were pinpointed, then the faulty devices are not used. Otherwise, the node is used as normal.

In the voting procedure, healthy nodes ignore the contribution of the erring node as far as the vote is *concerned*, but count the number of correct submissions that the node makes. The count is reset to zero every time the node makes an error. After a pre-determined number of correct submissions, the system again uses the erring node, *resetting* the count of errors_since_last_validated to zero.

NODE SELF-CONFIDENCE MESSAGES

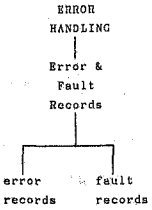
Node self-confidence is indicated in two ways. Firstly, when the self-testing is complete, the ready message indicates the status of the node. Secondly, the submission of correct data *for voting and task-complete checks* indicates whether or not the node is ready for use.



FAULT PINPOINTING

Fault pinpointing is the *process of localizing faults* by applying tests appropriate to the detected errors.

The error records indicate the ID of the errors detected. The pinpointing process ascertains the type of the error which caused the node isolation (i.e. ROM, RAM, servo, sensor etc.). Then the self-tests applicable to that device type are used to confirm the existence of the fault. Otherwise general tests are made to find the root cause of the error.



ERROR RECORDS

The *error handling routines provide the means* for the creation and maintenance of error records.

Whenever an error is detected, a call is made to an error record creation procedure. The procedure is passed

1) the ID of the data found to be in error

2) the type of error

The routine places this information, plus the detecting node's address into an error record. It then updates the appropriate error count, identifying it by error type and bus address. Hence ROM, RAM, sensor errors etc. for each node, are counted.

If the number of errors of a particular node/type combination exceeds the acceptable maximum, then the node isolation procedure is invoked.

FAULT RECORDS

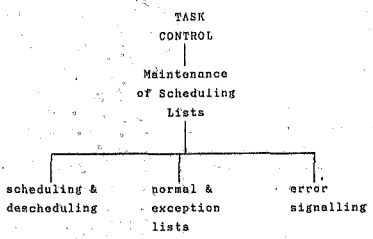
The fault handling routines provide the means for the creation and maintenance of fault records.

Whenever a fault has been pinpointed as far as is possible, using the testing procedures, the following information is available:

1) the ID of the device found to be faulty

2) the type of fault

The routine places this information, plus the detecting node's address and the detection time into a fault record. It then modifies the status of the device to show that it is not available for use.



SCHEDULING AND DESCHEDULING

Scheduling and descheduling of tasks is done via operating system calls that keep track of time usage.

The task schedule list is in the form of a linked list so that new entries can be inserted in the correct time-defined position. Hence, no list sorting is necessary.

Scheduling - When the scheduling of a task is desired, the program calls a scheduling routine. This routine requires the ID of the task to be scheduled, and the time of its execution.

The routine then obtains the expected task execution time (including error handling) from the task descriptor list. It then checks the task schedule list to see if the time between the start time and the end time of the task has already been allocated. If so, an error record is generated and a status of NOT OK is returned, so that the program attempting to schedule can take appropriate action.

If the time is vacant, then the task schedule list is updated to include the new task. A status of OK is returned.

Descheduling - When descheduling is required, the routine is called, and the task ID is passed. The descheduling procedure searches the task schedule list and removes the appropriate entry, thus making that time available for other tasks. If the task is not found, then an error record is created because this means that a task which should have been scheduled was not. A status of NOT OK is returned.

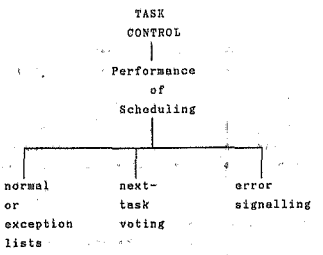
NORMAL AND EXCEPTION LISTS

Separate physical lists are not present. In exception conditions, the task schedule list is cleared, and those tasks which are appropriate to the exception condition are scheduled instead.

ERROR SIGNALLING

Scheduling and descheduling error reports are provided when such errors occur.

The status value returned to the calling program enables it to take appropriate action.



NORMAL OR EXCEPTION LISTS

Since only one list is to be supported, with its entries set appropriate to the conditions, there is no need for the scheduling routines to make any distinction.

Scheduling is accomplished after each task is completed, by examining the task schedule list. If there is adequate time before the next task, self-tests are run. Otherwise a call is made to schedule the next task.

In exception conditions, the operating system modifies the task schedule list to include only those tasks which are necessary for the test procedures, and for other functions needed to handle the exception.

After every execution, the task schedule list is modified by removing the task.

If the inter-task time check reveals a value less than 0, then a scheduling error record is generated.

NEXT-TASK VOTING

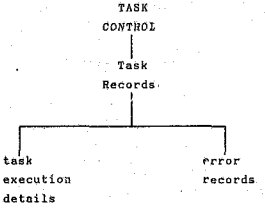
When a node has made the error of executing an incorrect task (detected by the watchdog mechanism), next-task voting is begun.

Before any task is executed, the task ID is communicated between the nodes, and a vote is taken to lessen the chances of another incorrect execution. Error records are produced when necessary.

If there is inadequate time before the next task, then the vote is omitted, and the other mechanisms are relied on for detection of errors.

ERROR SIGNALLING

Error records are generated when necessary.

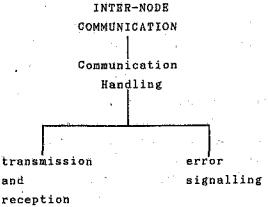


TASK EXECUTION DETAILS

To aid diagnostics etc., a list of tasks run, with their initiation times and completion times, is kept, and updated after the execution of each task.

ERROR RECORDS

Error records conform with the standard format.



TRANSMISSION AND RECEPTION

Transmission - A pointer to the data to be transmitted, a count of the number of bytes to be transmitted, and the destination of the message (ALL for broadcast) is passed to the routine. The routine writes the appropriate control into the SCB, sets up the command blocks and sets up the list of pointers. If it is the bus controller, then it then activates the bus interface board to start the transmission. Otherwise it must wait for an instruction to transmit, from the bus controller.

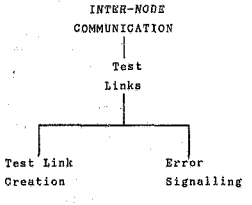
Reception - A pointer to the buffer into which the data is to be placed, and a maximum byte count are passed to the routine. The routine writes the appropriate control into the SCB, sets up the command blocks and the list of pointers, and, if it is the bus controller, then it activates the bus interface to start reception. Otherwise, it must wait for a receive instruction from the controller.

A multi-node co-operation routine allows the mutual exchange of data to be accomplished using only one call. This routine makes use of the transmission and reception routines to accomplish this effect.

The transmission and reception routines are provided for communication with non-fault-tolerant nodes, other fault-tolerant groups on the bus, and for inter-node communication in exception conditions. The multi-node co-operation routine is used the most.

ERROR SIGNALLING

Because the bus controller board is intelligent and provides *status and error signals*, these are examined to find communication errors. Then, normal error reports are generated, and, where a fault is identified, a fault record is also generated.

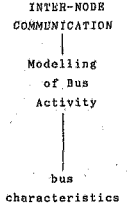


TEST LINK CREATION

Test link creation is facilitated by the use of the *transmission and reception routines.* When it is decided that the communication system is to be tested, or inter-node testing is to be done, the bus controller sends a message to the other nodes. This message merely establishes the *readiness of the other nodes, which respond* with an I'M READY message. A status of OK is then returned, so that testing can begin. If there is no response, or an incorrect or negative response is received, then a status of NOT OK is returned. Error records are generated.

ERROR SIGNALLING

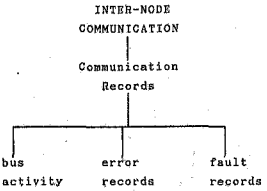
Error messages are generated whenever necessary.



BUS CHARACTERISTICS

The modelling routine maintains buffers according to the commands, and produces status and error messages in the format of the bus controller card.

The routine reads the control command in the system control block, and obtain the necessary data from it. It then accesses the command blocks and executes the appropriate actions according to the action commands, updating the necessary communication table entries, and transferring data between node buffers.



BUS ACTIVITY

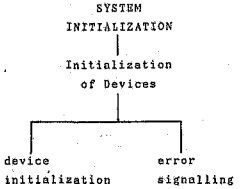
To aid diagnostics etc., records of bus activity are produced. For each communication, the record holds source, destination and time of the communication.

ERROR RECORDS

Error records conform with the standard format.

FAULT RECORDS

Fault records conform with the standard format.



DEVICE INITIALIZATION

Devices on the 186 board are initialized first, followed by initialization of all memory components, the analog I/O board and the inter-node bus controller.

After each initialization, a check back is performed to ensure that the device is ready for use. If it is not, then the initialization is re-tried, and an error message is produced. If the second try fails, then a fault record is generated, and the device may not be used. If this fault means that none of the system may be used, then the operator is alerted.

ERROR SIGNALLING

Failure to successfully initialize any device is reported first by an error record, then by a fault record.